INTRODUCTION TO INTEGER PROGRAMMING AND APPLICATIONS WITH julia



Luiz Antonio Nogueira Lorena
Luiz Henrique Nogueira Lorena
Antônio Augusto Chaves

Copyright © 2026 Luiz Antonio Nogueira Lorena, Luiz Henrique Nogueira Lorena, and Antônio Augusto Chaves.

DOI: https://doi.org/10.5281/zenodo.21227210

ISBN: 978-65-02-27111-7

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/

First printing, July 2026

Contents

Preface 15
How to use this book 15
Acknowledgement 16
Referencing this book 16

1 Introduction to Integer Programming 17
1.1 Basic model 17
1.2 Example 18
1.3 Applications 24
1.4 Final Remarks 25
1.5 Exercises 25

I Applications

2 Knapsack Problems 29
2.1 Knapsack problem applications 30
2.2 Example 30
2.3 The Binary Knapsack Problem 31
2.4 The Integer Knapsack Problem 36
2.5 Final Remarks 41
2.6 Exercises 42

3 Location Problems 43
3.1 The P-Median Problem 44
3.2 Covering problems 54
3.3 Final remarks 67
3.4 Exercises 67

4 Traveling Salesman Problem 71
4.1 TSP applications 72
4.2 The TSP model 72
4.3 TSP with no subtour elimination constraints 75
4.4 TSP with MTZ constraints 79
4.5 TSP with lazy constraints 83

4.6 Final remarks . . . 87
4.7 Exercises . . . 88

5 Graph Problems . . . 91
5.1 Maximum Independent Set Problem . . . 93
5.2 Vertex Covering Problem . . . 96
5.3 Graph Coloring Problem . . . 101
5.4 Graph Partitioning Problem . . . 106
5.5 Clique Partitioning Problem . . . 120
5.6 Final remarks . . . 127
5.7 Exercises . . . 127

II Advanced Techniques

6 Column Generation . . . 133
6.1 Column Generation technique . . . 133
6.2 The one-dimensional cutting stock problem . . . 134
6.3 The P-Median Problem . . . 149
6.4 Final remarks . . . 160
6.5 Exercises . . . 161

7 Lagrangian Relaxation . . . 163
7.1 Lagrangian Relaxation algorithm . . . 163
7.2 Simple Example of Lagrangian Relaxation . . . 168
7.3 Lagrangian Relaxation for P-Median Problem . . . 175
7.4 Lagrangian Relaxation for the Maximum Independent Set Problem . . . 184
7.5 Lagrangian relaxation for the Generalized Assignment Problem . . . 199
7.6 Final remarks . . . 202
7.7 Exercises . . . 203

8 Branch-and-Bound . . . 205
8.1 Branch-and-Bound Components . . . 205
8.2 Maximization example: Binary Knapsack Problem . . . 209
8.3 Minimization example: Traveling Salesman Problem . . . 221
8.4 Final remarks . . . 231
8.5 Exercises . . . 231

Bibliography 235

Index 243

Appendices 245

A Julia Introduction 245
A.1 Random number generation 245
A.2 Vectors and matrices 246
A.3 Structures and mutable structures 252
A.4 Flow control 253
A.5 Functions 254
A.6 Advanced example 257

B Exercise Solutions 261
B.1 Chapter 1 261
B.2 Chapter 2 269
B.3 Chapter 3 275
B.4 Chapter 4 302
B.5 Chapter 5 316
B.6 Chapter 6 335
B.7 Chapter 7 377
B.8 Chapter 8 423

List of Figures

1.1 Best IP value. 21
1.2 Best LP relaxation value. 23
1.3 Integer Programming applications. Source: image generated with ChatGPT. 24
1.4 *Flow Free* board example. 26
1.5 *Flow Free* boards. 26
2.1 A climber who had to solve the knapsack problem. Source: image generated with ChatGPT. 29
2.2 Graphical representation of the BKP solution. 33
2.3 Graphical representation of the LP-BKP solution. 36
2.4 Graphical representation of the IKP solution. 39
2.5 Graphical representation of the LP-IKP solution. 41
3.1 A policymaker analyzing optimal facility location. Source: image generated with ChatGPT. 43
3.2 PMP-C instance with 10 facilities candidates and 92 demand points.. 47
3.3 Optimal solution for PMP-C. 50
3.4 PMP-N instance with 102 locations. 51
3.5 Optimal solution for PMP-N. 54
3.6 SCLP instance with 102 locations. 56
3.7 Optimal solution for SCLP. 59
3.8 Optimal solution for MCLP. 63
3.9 Optimal solution for MCLP with random demands. 66
3.10 Graphical representation of the ambulance allocation problem. . . 68
4.1 A traveling salesman planning his trip. Source: image generated with ChatGPT. 71
4.2 TSP graph instance and different subset possibilities. 74
4.3 TSP instance. 76
4.4 Solution for TSP example using TSP-Rel model. 78

4.5 Solution for TSP example using TSP-MTZ model. 82
4.6 Lazy constraints flowchart. 83
4.7 Solution for TSP example using lazy constraints. 87
4.8 TSP instance `tsp_exercise.h5`. 88
4.9 Example weighted undirected graph. 89
5.1 Graph types. 91
5.2 Fundamental Graph problems. 92
5.3 MISP instance. 94
5.4 Solution for MISP example. 96
5.5 VCP instance. 99
5.6 Solution for VCP example. 101
5.7 GCP instance. 103
5.8 Solution for GCP example. 106
5.9 GPP instance. 109
5.10 Solution for GPP example. 112
5.11 Metis solution for GPP example. 115
5.12 WGPP instance. 117
5.13 Solution for WGPP example. 120
5.14 CPP instance. 123
5.15 Solution for CPP example. 127
5.16 Point feature map labeling as a MISP: (a) 3 points and 4 candidate positions for each point. (b) graph corresponding to points and their intersections as edges. (c) conflict-free solution. 128
5.17 Museum map for the vertex covering problem. 129
5.18 Map instance for the Graph Coloring Problem. 129
5.19 Circuit board instance for the Graph Partitioning Problem. 130
5.20 Social network instance for the Clique Partitioning Problem. . . . 130
6.1 Column Generation flowchart. 134
6.2 1D-CSP for panels. 134
6.3 Cutting stock instance: item types. 136
6.4 Solution for CSP example. 140

6.5 Convergence of the Pricing Subproblem for 1D-CSP. 148
6.6 Solution of the CG for 1D-CSP. 149
6.7 PMP instance. 153
6.8 Convergence of the Pricing Subproblem for PMP. 160
6.9 Solution of the CG for PMP. 160
7.1 Two-dimensional representation of the feasible set of model E. . 168
7.2 Lagrangian Dual Problem solution. 170
7.3 Lagrange Relaxation Convergence for model E. 175
7.4 Lagrange Relaxation Convergence for PMP. 184
7.5 Example graph for LagClus. 187
7.6 Metis partitioning of the example graph. 188
7.7 Lagrange Relaxation Convergence for MISP. 199
8.1 Relaxation bound for a maximization problem. 205
8.2 B&B flowchart. 206
8.3 B&B tree for BKP after initializing the root node. 210
8.4 B&B tree for BKP after evaluating node 1. 211
8.5 B&B tree for BKP after evaluating node 2. 211
8.6 B&B tree for BKP after evaluating node 4. 212
8.7 B&B tree for BKP after evaluating node 5. 212
8.8 B&B tree for BKP after evaluating node 6. 213
8.9 B&B tree for BKP after evaluating node 7. 214
8.10 B&B tree for BKP after evaluating node 8. 215
8.11 B&B tree for BKP after evaluating node 9. 216
8.12 B&B tree for BKP after evaluating node 3. 217
8.13 B&B tree for BKP after evaluating node 10. 218
8.14 B&B tree for BKP after evaluating node 11. 219
8.15 B&B tree for BKP after evaluating node 12. 219
8.16 B&B tree for BKP after evaluating node 13. 220
8.17 Bounds for a minimization problem. 221
8.18 Branching strategy for the TSP. 221
8.19 TSP instance with 7 cities. 222

8.20 Assignment problem relaxation solution at node 1. 224
8.21 B&B tree for TSP after evaluating node 1. 224
8.22 Assignment problem relaxation solution at node 2. 225
8.23 B&B tree for TSP after evaluating node 2. 225
8.24 Assignment problem relaxation solution at node 4. 226
8.25 B&B tree for TSP after evaluating node 4. 226
8.26 Assignment problem relaxation solution at node 5. 227
8.27 B&B tree for TSP after evaluating node 5. 227
8.28 Assignment problem relaxation solution at node 3. 228
8.29 B&B tree for TSP after evaluating node 3. 228
8.30 Assignment problem relaxation solution at node 6. 229
8.31 B&B tree for TSP after evaluating node 6. 229
8.32 Assignment problem relaxation solution at node 7. 230
8.33 B&B tree for TSP after evaluating node 7. 230
8.34 TSP instance for the Branch-and-Bound exercise. 232
8.35 MISP instance for the Branch-and-Bound exercise. 232
A.1 TSP solution obtained with Simulated Annealing. 260
B.1 Solution for the `flowfree5_1.csv` instance. 267
B.2 Solution for the `flowfree5_2.csv` instance. 268
B.3 Solution for the `flowfree6_1.csv` instance. 268
B.4 Graphical representation of the solution for the P-Median Problem (PMP-N) with n clients and $p = 4$ facilities. 277
B.5 Graphical representation of the solution for the Fault-tolerant P-Median Problem with $p = 3$ and $r = 2$. 281
B.6 Graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters. 283
B.7 Graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=0. 287
B.8 Graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=1. 289

B.9 Graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and p = 3. 292
B.10 Graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 0$. 296
B.11 Graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 1$. 297
B.12 Graphical representation of the solution for the TEAM model. . 302
B.13 Graphical representation of the solution of the TSP without subtour elimination constraints for `tsp_exercise.h5`. 305
B.14 Graphical representation of the solution of the TSP with MTZ constraints for `tsp_exercise.h5`. 308
B.15 Graphical representation of the solution of the TSP with lazy constraints for `tsp_exercise.h5`. 312
B.16 Graphical representation of the solution of the TSP with MTZ constraints for `tsp_exercise.csv`. 316
B.17 Visualization of the maximum independent set for dataset `d25_01.dat`. 320
B.18 Museum map with numbered rooms. 320
B.19 Graph representation of the museum. 321
B.20 Vertex cover solution for the museum problem. 323
B.21 Graph representation of the districts. 324
B.22 GCP solutiong for the districts problem represented as a graph. . 327
B.23 GCP solutiong for the districts problem represented on the map.. . . 327
B.24 Graph representation of the circuit represented as a graph. 328
B.25 GPP solution for the circuit dataset represented on graph. 331
B.26 Obtained circuit partitions. 332
B.27 Visualization of the CPP solution for the karate club dataset. . . . 335
B.28 Reduced cost convergence for the dataset `sjc_n=30_p=7_C=25.csv`.. . 344
B.29 RMIP solution for the dataset `sjc_n=30_p=7_C=25.csv`. 346
B.30 ILP solution for the dataset `sjc_n=30_p=7_C=25.csv`. 349

B.31 Reduced cost convergence for the dataset `karate.csv`. 359
B.32 RMIP solution for the dataset `karate.csv`. 361
B.33 ILP solution for the dataset `karate.csv`. 364
B.34 Reduced cost convergence for the dataset `tutte_graph.csv`. 371
B.35 RMIP solution for the dataset `tutte_graph.csv`. 373
B.36 ILP solution for the dataset `tutte_graph.csv`. 376
B.37 Graphical representation of the problem defined in Equation (B.13).. 379
B.38 Graphical representation of the problem defined in Equation (B.13).. 383
B.39 Convergence of the Lagrangian relaxation for the example problem.. 387
B.40 Convergence of the Lagrangian relaxation for the P-Median Problem for dataset `exercise_pmp.csv`. 394
B.41 Plot showing the convergence of lower and upper bounds for the Fault-tolerant P-Median Problem. 402
B.42 Plot showing the convergence of lower and upper bounds for the Capacitated P-Median Problem. 410
B.43 Convergence of the LagClus algorithm for the Maximum Independent Set Problem (MISP). 418
B.44 B&B search tree for BKP using DFS strategy. 429
B.45 B&B search tree for BKP using Best Bound strategy. 432
B.46 B&B search tree for TSP. 439
B.47 B&B search tree for MISP. 446
B.48 Triangle graph. 447
B.49 Branch-and-Cut Search Tree for MISP. 455

List of Tables

1.1 Real-world applications of Integer Programming. 24
2.1 Real-world applications of the Knapsack Problem. 30
3.1 Real-world applications of the P-Median Problem. 45
3.2 Real-world applications of the Set Covering Location Problem. . . 56
3.3 Real-world applications of the Maximal Coverage Location Problem.. 60
3.4 Dataset structure of the input file for the TEAM model. 69
4.1 Real-world applications of the Traveling Salesman Problem. 72
5.1 Real-world applications of the Maximum Independent Set Problem.. 93
5.2 Real-world applications of the Vertex Covering Problem. 97
5.3 Real-world applications of the Graph Coloring Problem. 101
5.4 Real-world applications of the Graph Partitioning Problem. 106
5.5 Real-world applications of the Clique Partitioning Problem. . . . 122
7.2 Derivation of the Lagrangian dual function segments. 170
7.3 Subgradient components for L_1 and L_2. 202
8.1 B&B component details. 207
B.1 Comparison of TSP solutions with different models. 312
B.2 Comparison of CPMP solutions with different models. 349
B.3 Comparison of CPP solutions with different models. 364
B.4 Comparison of GCP solutions with different models. 376
B.5 Optimal solutions of the Lagrangian subproblem over the intervals of λ. 380
B.6 Difference between DFS and Best Bound strategies. 432
B.7 Difference between Branch-and-Bound and Branch-and-Cut strategies for solving the MISP. 455

Preface

Welcome to **Introduction to Integer Programming and Applications with Julia**! This book is designed to provide a clear, practical, and application-focused introduction to the field of integer programming, with a modern computational approach using the Julia programming language.

Integer programming plays a central role in decision, making problems where variables are required to take on discrete values. It underpins solutions to a vast array of real-world challenges, ranging from logistics and scheduling to resource allocation and network design. Whether you're a student beginning your journey in optimization, a researcher exploring computational tools, or a practitioner seeking to apply integer programming in your domain, this book aims to bridge the gap between theory and practice.

What makes this book distinctive is its integration of mathematical modeling concepts with hands-on coding using Julia and the JuMP [1] optimization framework. Julia's syntax is expressive and readable, making it an ideal language for learning optimization while also offering performance suitable for solving complex, large-scale problems.

Throughout the chapters, you will explore foundational concepts in integer programming, from formulation and modeling strategies to solving techniques and application case studies. Each topic is illustrated with clear examples, accompanied by Julia code, to encourage experimentation and deeper understanding.

We invite you to use this book as a guide not only to master the principles of integer programming, but also to develop the computational skills needed to implement and solve real-world optimization problems effectively.

Welcome aboard, and enjoy the journey into the world of integer programming with Julia!

How to use this book

This book is structured into nine chapters, each focusing on different aspects of integer programming. The chapters are designed to be read sequentially, but you can also jump to specific topics of interest. Each chapter includes theoretical explanations, practical examples, and exercises to reinforce your understanding. The accompanying Julia code is available in the form of Jupyter notebooks, which

can be accessed on GitHub. We encourage you to run the code, experiment with it, and modify it to see how different approaches affect the solutions.

If this is your first encounter with Julia language, we recommend you start with Chapter 1, which provides a summary of the Julia language commands used in the previous chapters. For a detailed description of the language, you can refer to the books by [2], [3], and [4].

Acknowledgement

We want to express our gratitude to the students of the Integer Programming course (PO-203) at the **Federal University of São Paulo** and the Combinatorial Optimization course (CAP-254) at the **National Institute for Space Research** for their valuable suggestions and for solving the exercises presented in this text.

Referencing this book

Citation bibtex

```bibtex
@book{LorenaEtAl2026,
  title = {
    Introduction to Integer Programming and Applications with Julia
  },
  author = {
    Lorena, Luiz Antonio Nogueira and
    Lorena, Luiz Henrique Nogueira and
    Chaves, Ant{\\^o}nio Augusto
  },
  year = {2026},
  doi = {https://doi.org/10.5281/zenodo.21227210},
  url = {https://intro-ilp-julia.github.io/book/}
}
```

1 Introduction to Integer Programming

Optimization problems seek the best solution for an objective function subject to a set of constraints. When the objective and constraints are linear and the variables are continuous, the problem is known as **Linear Programming (LP)** [5]. In this case, the feasible region is a convex polyhedron formed by the intersection of linear half-spaces.

Integer Programming (IP), conversely, requires some or all variables to assume integer values. While IP models often maintain linear objectives and constraints [6], the requirement for integrality makes the feasible set a collection of discrete points, which significantly changes the complexity of the solution process.

Complex real-world scenarios can be modeled and solved using the following components commonly found in optimization problems:

- **Objective function**: the function to be optimized, either maximized or minimized, depending on the problem's requirements.
- **Constraints**: functions that restrict the values of the variables, usually expressed as equalities or inequalities. They define the feasible region of the problem.
- **Feasible set**: the set of all points that simultaneously satisfy all constraints; it represents the region of permissible solutions.
- **Domain of variables**: the range of values that the variables can assume. In Integer Programming, this domain is typically limited to a discrete subset of the real numbers.

1.1 Basic model

The standard form of an Integer Programming problem can be expressed as follows:

Definition 1.1 — IP model.

$$\max_{x} \quad \sum_{j=1}^{n} c_j x_j \tag{1.1.1}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1, ..., m \tag{1.1.2}$$

$$x_j \in \mathbb{Z}_{\geq 0} \qquad j = 1, ..., n \tag{1.1.3}$$

where:

- c_j objective coefficients
- x_j decision variables
- a_{ij} constraint coefficients
- b_i constraint right-hand side values

In Equation (1.1), the optimal value of the objective function is obtained by choosing values of x that satisfy all constraints. The objective function and constraints are linear, while the decision variables are integer-valued.

The optimal value obtained by solving the model is commonly denoted as z_{IP}, where

$$z_{\text{IP}} = \max_{x} \left\{ \sum_{j=1}^{n} c_j x_j \;\middle|\; \sum_{j=1}^{n} a_{ij} x_j \leq b_i, x_j \in \mathbb{Z}_{\geq 0} \right\} \tag{1.2}$$

Here, z_{IP} represents the maximum value of the objective function over all feasible integer solutions.

1.2 Example

We will explore a simple optimization problem in two forms: as an Integer Programming (IP) and as its Linear Programming (LP) relaxation. The following example will help illustrate the differences between integer and continuous optimization, both in terms of feasible solutions and optimal values.

Example 1.1 Consider two decision variables, x_1 and x_2, subject to a linear constraint $3x_1 + 5x_2 \geq 7$ and variable bounds. The objective is to minimize the linear function $7x_1 + 10x_2$.

We will first solve the problem with integer constraints, then relax these constraints to allow continuous values, and compare the results.

1.2.1 Integer Programming

The following example models a problem with two decision variables, x_1 and x_2, a linear constraint $3x_1 + 5x_2 \geq 7$, and integer limits for the variables, $x_1, x_2 \in \{0, 1\}$. We seek the smallest value of the objective function $7x_1 + 10x_2$ among all combinations of variable values. The model is:

$$\min_{x_1, x_2} \quad 7x_1 + 10x_2 \tag{1.3.1}$$

$$s.t. \quad 3x_1 + 5x_2 \geq 7 \tag{1.3.2}$$

$$x_1, x_2 \in \{0, 1\} \tag{1.3.3}$$

Listing 1.1 presents the code that formulates and solves the integer programming problem described above using Julia, the JuMP modeling language and the HiGHS solver. The code defines the decision variables, objective function, and constraints, then solves the model and prints the optimal solution.

```
using JuMP  # Modeling language
using HiGHS # Solver

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Decision variables
@variable(model, x1, Bin)
@variable(model, x2, Bin)

# Objective
@objective(model, Min, 7x1 + 10x2)

# Constraint
@constraint(model, 3x1 + 5x2 >= 7)

# Optimize
JuMP.optimize!(model)

# Print model
print("Model:\n$model")

# Extract solution
```

```
27 println("Solution:")
28 println("x1 = ", JuMP.value(x1))
29 println("x2 = ", JuMP.value(x2))
30 println("Objective value = ", JuMP.objective_value(model))
```

Listing 1.1: IP example.

The first step in the Julia code is to import the necessary packages. We use JuMP (Listing 1.1-1) for modeling the optimization problem and HiGHS (Listing 1.1-2) as the solver to find the optimal solution. Next, we create a new optimization model and specify HiGHS as the solver (Listing 1.1-5). We set the model to silent mode to suppress solver output (Listing 1.1-8). Then, we define the decision variables x_1 and x_2 as binary variables using (Listing 1.1-11). The objective function is defined to minimize in Listing 1.1-15. We add the constraint $3x_1 + 5x_2 \geq 7$ in Listing 1.1-18. Finally, we call `JuMP.optimize!(model)` to solve the optimization problem (Listing 1.1-21) and print the model (Listing 1.1-24) and solution values (Listing 1.1-27).

Listing 1.2 shows the expected output of the code, which includes the model formulation and the optimal solution values for x_1, x_2, and the objective function.

Output

```
Model:
Min 7 x1 + 10 x2
Subject to
 3 x1 + 5 x2 >= 7
 x1 binary
 x2 binary
Solution:
x1 = 1.0
x2 = 1.0
Objective value = 17.0
```

Listing 1.2: Output of the IP example.

Interpreting the solution

Figure 1.1 shows a two-dimensional representation of the model, highlighting the half-planes representing the constraints. The hatched area represents the intersection of the half-planes $3x_1 + 5x_2 \geq 7$ and $x_1, x_2 \in [0, 1]$. The objective function (dashed line) changes value with parallel displacements. The only integer feasible

point, $(1, 1)$, satisfies these constraints as well as $x_1, x_2 \in \{0, 1\}$. Therefore, this point is also the optimal solution to the problem, with an objective function value of $z_{\text{IP}} = 17$.

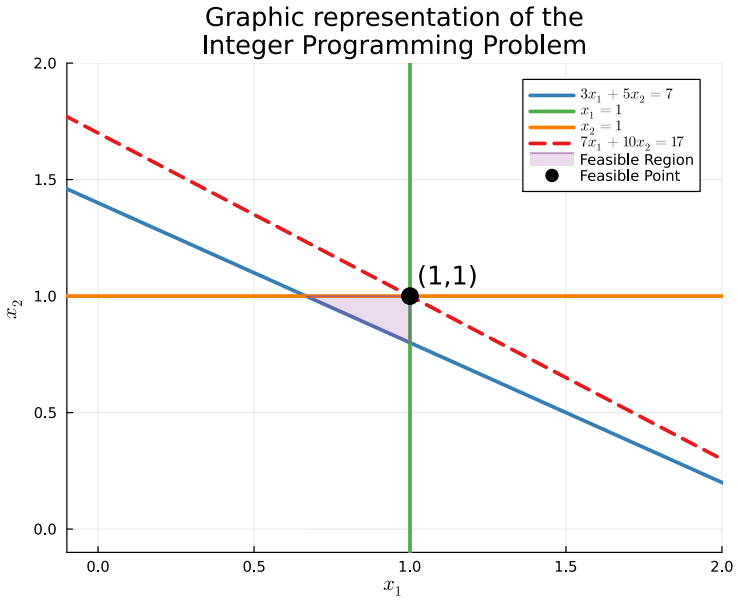


Figure 1.1: Best IP value.

1.2.2 Linear Programming

Integer Programming problems are generally difficult to solve because they belong to the NP-hard class [7]. A common approach to their solution is to relax some of their constraints and analyze the resulting problem. Linear Programming relaxation is the most common approach, and for this example, it can be formulated as:

$$\min_{x_1, x_2} \quad 7x_1 + 10x_2 \tag{1.4.1}$$

$$s.t. \quad 3x_1 + 5x_2 \geq 7 \tag{1.4.2}$$

$$x_1, x_2 \in [0, 1] \tag{1.4.3}$$

Listing 1.3 presents the code that formulates and solves the integer programming problem described above using Julia, the JuMP modeling language and the HiGHS solver. The main difference from the previous code is that the decision variables x_1 and x_2 are now defined as continuous variables with bounds between 0 and 1, instead of binary variables (Listing 1.3-10).

```
1 using JuMP  # Modeling language
2 using HiGHS # Solver
3
```

```
# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Decision variables
@variable(model, 0 <= x1 <= 1)
@variable(model, 0 <= x2 <= 1)

# Objective
@objective(model, Min, 7x1 + 10x2)

# Constraint
@constraint(model, 3x1 + 5x2 >= 7)

# Optimize
JuMP.optimize!(model)

# Print model
print("Model:\n$model")

# Extract solution
println("Solution:")
println("x1 = ", JuMP.value(x1))
println("x2 = ", JuMP.value(x2))
println("Objective value = ", JuMP.objective_value(model))
```

Listing 1.3: ILP example.

Listing 1.4 shows the expected output of the code, which includes the model formulation and the optimal solution values for x_1, x_2, and the objective function. The optimal solution for the linear programming relaxation is $x_1 \approx 0.67$ and $x_2 = 1.0$, with an objective function value of $z_{\text{LP}} \approx 14.67$. This solution is not integer, which illustrates the difference between the integer programming solution and its linear programming relaxation.

Output

```
Model:
Min 7 x1 + 10 x2
Subject to
```

```
 3 x1 + 5 x2 >= 7
 x1 >= 0
 x2 >= 0
 x1 <= 1
 x2 <= 1
Solution:
x1 = 0.6666666666666666
x2 = 1.0
Objective value = 14.666666666666666
```

Listing 1.4: Output of the ILP example.

Interpreting the solution

Here, the integer constraints $x_1, x_2 \in \{0, 1\}$ are replaced by the continuous bounds $x_1, x_2 \in [0, 1]$. Figure 1.2 shows the feasible region for the LP relaxation. Considering the movement of the objective function line, the best feasible point is $(0.67, 1)$, with an objective value of $z_{\text{LP}} = 14.67$. This point is not an integer solution, which illustrates the difference between the integer programming solution and its linear programming relaxation. The optimal value of the LP relaxation, z_{LP}, provides a lower bound on the optimal value of the original integer programming problem, z_{IP}. In this case, we have $z_{\text{LP}} \approx 14.67 \leq z_{\text{IP}} = 17$, demonstrating that the LP relaxation can be used to estimate how close the integer solution is to the optimal value.

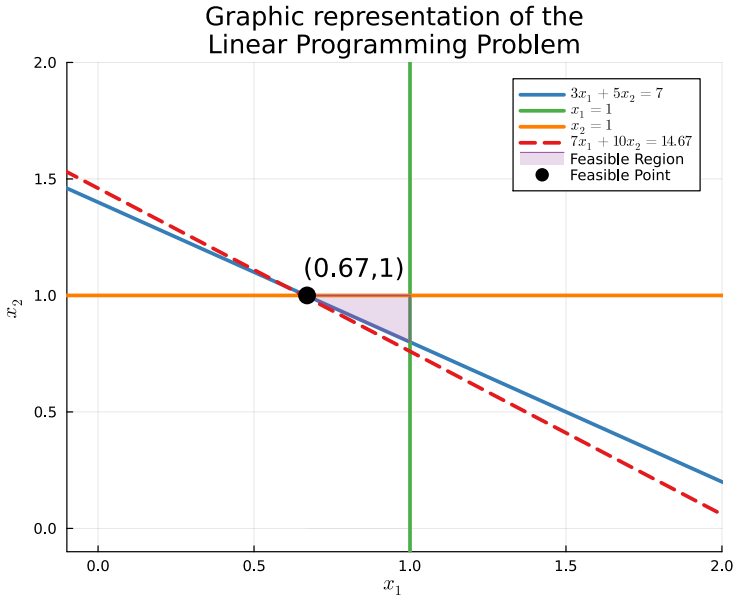


Figure 1.2: Best LP relaxation value.

1.3 Applications

The book *Model building in mathematical programming* [8] presents formulations for a variety of problems representing real-world applications that can be effectively addressed using mathematical programming and optimization techniques. Some notable applications highlighted in the book are presented in Figure 1.3 and described in more detail in Table 1.1.

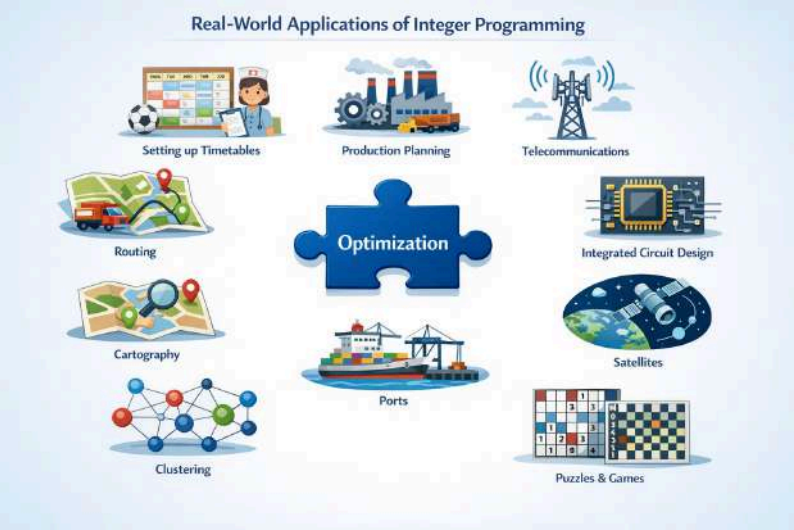


Figure 1.3: Integer Programming applications. Source: image generated with ChatGPT.

Application	Description
Setting up timetables	Scheduling for schools, nurses' shifts, games, drivers, and other scenarios requiring efficient allocation of time resources.
Production planning	Tasks such as machine scheduling and cutting stock optimization, aimed at improving production processes and resource utilization.
Telecommunications	Determining optimal locations for cell phone antennas to ensure effective coverage and communication infrastructure.
Routing	Distribution logistics, shortest-path determination, and optimization of transportation routes to minimize costs and maximize efficiency.

Application	Description
💻 **Integrated circuit design**	Optimizing the placement and routing of components on integrated circuits to enhance performance and reduce manufacturing costs.
🗺️ **Cartography**	Placement of labels on maps to ensure appropriate positioning, avoid overlap, and maintain readability.
⚓ **Ports**	Allocation of berths for ships in ports, considering vessel sizes, schedules, and available resources.
🛰️ **Satellites**	Optimization of daily satellite observation schedules to maximize coverage and data collection efficiency.
🔗 **Clustering**	Identification of clusters within complex networks using optimization techniques.
🎮 **Puzzles or Game Problems**	Formulating games and puzzles as graph problems, such as n-queens and Sudoku.

Table 1.1: Real-world applications of Integer Programming.

1.4 Final Remarks

This chapter provided an introduction to Integer Programming, highlighting its differences from Linear Programming and presenting a simple example to illustrate these concepts. We also discussed various real-world applications of Integer Programming and provided exercises to practice formulating and solving optimization problems using the JuMP package in Julia. In the next chapter, we will delve into the first practical application selected for this book: knapsack problems.

1.5 Exercises

Exercise 1.1 Suppose there are ten possible locations where health centers could be installed to serve a population. Express the following constraints on which centers can be selected:

a) You cannot select all centers;

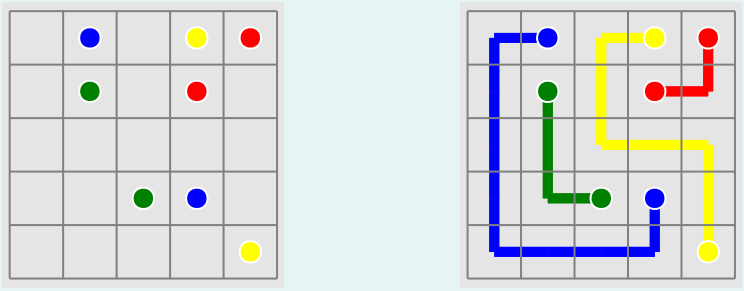
b) At least one center must be selected;

c) If center 5 is selected, then center 2 cannot be selected;

d) Center 10 can be selected only if center 7 is also selected;

e) Centers 3 and 8 must either both be selected or both not selected.

Exercise 1.2 *Flow Free* is a popular puzzle game available on iOS and Android, played on a square grid with simple rules. The goal is to connect matching colors with pipes to create a continuous flow. The objective is to connect all pairs of colors while covering the entire board. Figure 1.4 shows an example of a 5×5 puzzle and its corresponding solution.



(a) Initial State. (b) Solution.

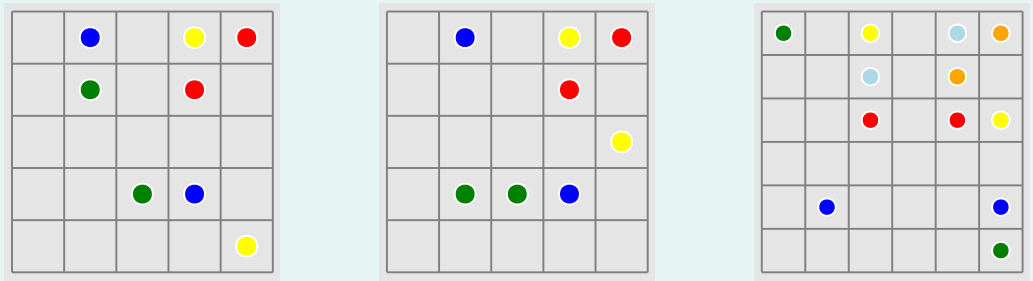
Figure 1.4: *Flow Free* board example.

An integer programming formulation for solving this problem (implemented in R language) can be found here. Use the JuMP package to formulate and solve the following problems (Figure 1.5):

a) Solve using the dataset `flowfree5_1.csv`;

b) Solve using the dataset `flowfree5_2.csv`;

c) Solve using the dataset `flowfree6_1.csv`.



(a) `flowfree5_1.csv`. (b) `flowfree5_2.csv`. (c) `flowfree6_1.csv`.

Figure 1.5: *Flow Free* boards.

I Applications

2 Knapsack Problems 29
2.1 Knapsack problem applications 30
2.2 Example 30
2.3 The Binary Knapsack Problem 31
2.4 The Integer Knapsack Problem 36
2.5 Final Remarks 41
2.6 Exercises 42

3 Location Problems 43
3.1 The P-Median Problem 44
3.2 Covering problems 54
3.3 Final remarks 67
3.4 Exercises 67

4 Traveling Salesman Problem . 71
4.1 TSP applications 72
4.2 The TSP model 72
4.3 TSP with no subtour elimination constraints 75
4.4 TSP with MTZ constraints 79
4.5 TSP with lazy constraints 83
4.6 Final remarks 87
4.7 Exercises 88

5 Graph Problems 91
5.1 Maximum Independent Set Problem . 93
5.2 Vertex Covering Problem 96
5.3 Graph Coloring Problem 101
5.4 Graph Partitioning Problem 106
5.5 Clique Partitioning Problem 120
5.6 Final remarks 127
5.7 Exercises 127

2 Knapsack Problems

The knapsack problem is a well-established and classical optimization problem. It involves selecting a subset of items to place in a knapsack, taking into account both their individual weights and values. The objective is to maximize the total value of the selected items while ensuring that their combined weight does not exceed the knapsack's capacity.

Figure 2.1 illustrates a scenario in which a climber faces the knapsack problem. The climber must decide which items to include in their knapsack based on their utility and the limited capacity available. This requires a careful evaluation of the weight and value of each item, followed by strategic decision-making to optimize the load carried.



Figure 2.1: A climber who had to solve the knapsack problem. Source: image generated with ChatGPT.

The book *Knapsack problems: algorithms and computer implementations* [9] serves as a valuable resource, offering an in-depth exploration of the various applications and extensions of the knapsack problem. It provides comprehensive insights into both the theoretical foundations and the practical applications of the problem across diverse fields. With detailed coverage of algorithms, optimization

techniques, and real-world case studies, the book serves as an essential guide for understanding and addressing knapsack-related challenges.

2.1 Knapsack problem applications

The knapsack problem, along with its numerous variations, has garnered significant attention in academic research due to its relevance in multiple fields such as logistics, resource allocation, and financial portfolio optimization (Table 2.1). Its combinatorial complexity, coupled with its practical significance, makes it a compelling and challenging subject of study.

Application	Description
🏭 **Production planning**	Choosing components or materials to maximize value while meeting cost or capacity limits.
🚚 **Logistics**	Optimizing the loading of goods or passengers while respecting weight or space constraints.
💰 **Marketing**	Allocating budget across campaigns to maximize return on investment.
🏥 **Healthcare**	Optimizing the selection of treatments or interventions under resource constraints.
🌳 **Environmental management**	Allocating resources to conservation efforts to maximize impact, such as biodiversity preservation.
🧬 **Bioinformatics**	Selecting subsets of biological elements (e.g., genes or proteins) that optimize specific properties.

Table 2.1: Real-world applications of the Knapsack Problem.

This chapter focuses on knapsack models involving binary and integer decision variables. The knapsack problem itself is a classic optimization problem in which the objective is to maximize the total value of items placed in a knapsack subject to a capacity constraint.

2.2 Example

In this chapter we will explore two knapsack models: the Binary Knapsack Problem and the Integer Knapsack Problem. We will present the mathematical

formulations for both models, their relaxations, and provide implementations in the Julia programming language.

The following example will be used to illustrate the concepts and formulations of both the binary and integer knapsack problems.

Example 2.1 A climber must decide which items to include in their knapsack based on their utility and the limited capacity available. The item values are given by the vector $v = (6, 8)$, the item weights are given by the vector $w = (4, 3)$, and the knapsack capacity is $W = 6$.

2.3 The Binary Knapsack Problem

In Binary Knapsack Problem (BKP), you are given a set of n items, each with a value and a weight. The task is to choose a subset of items that maximizes total value (Equation (2.1.1)) while ensuring the total weight does not exceed the knapsack's capacity (Equation (2.1.2)). The BKP is typically formulated as follows:

Definition 2.1 — BKP model.

$$\max_{x} \quad \sum_{i=1}^{n} v_i x_i \tag{2.1.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2.1.2}$$

$$x_i \in \{0, 1\} \qquad i = 1, ..., n \tag{2.1.3}$$

where:

v_i	item i value
x_i	decision variables (select item)
w_i	item i weight
W	knapsack capacity

This problem is also knows as 0–1 knapsask problem because each item can only be fully included or excluded; fractional selection is not permitted.

2.3.1 Solving the Binary Knapsack Problem

Listing 2.1 presents the Julia code to solve the BKP for the chapter's example. Listing 2.1-4 defines the problem data, including the item values, weights, knapsack capacity, and the number of items. The model is created using JuMP and the HiGHS solver (Listing 2.1-10). The decision variables are defined as binary (Listing 2.1-16), the objective function is set to maximize the total value of selected

items (Listing 2.1-19), and the constraint ensures that the total weight does not exceed the knapsack capacity (Listing 2.1-22). Finally, the model is solved and the solution is printed (Listing 2.1-31).

```
using JuMP  # Modeling language
using HiGHS # Solver

# Problem data
v = [6, 8]     # item value
w = [4, 3]     # item weight
W = 6          # knapsack capacity
n = length(w)  # number of items

# Binary model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Decision variables
@variable(model, x[1:n], Bin)

# Objective
@objective(model, Max, sum(v[i] * x[i] for i=1:n))

# Constraint
@constraint(model, sum(w[i] * x[i] for i=1:n) <= W)

# Print model
print("Model:\n$model")

# Solve model
JuMP.optimize!(model)

# Display solution
println("Solution:")
for i in 1:n
    println("x[$i] = ", JuMP.value(x[i]))
end
println("Objective = ", JuMP.objective_value(model))
```

Listing 2.1: Solving the BKP with Julia.

Listing 2.2 presents the output of the BKP model, showing the mathematical formulation of the model and the solution obtained after optimization.

Output

```
Model:
Max 6 x[1] + 8 x[2]
Subject to
 4 x[1] + 3 x[2] <= 6
 x[1] binary
 x[2] binary
Solution:
x[1] = 0.0
x[2] = 1.0
Objective = 8.0
```

Listing 2.2: Output of the BKP model.

Figure 2.2 presents a two-dimensional graphical representation of the problem, with each variable represented on an axis. Among the binary points, only $(0, 0)$, $(0, 1)$, and $(1, 0)$ are feasible. Therefore, the solution $(0, 1)$ is optimal, yielding an objective function value $z_{\text{BKP}} = 8$.

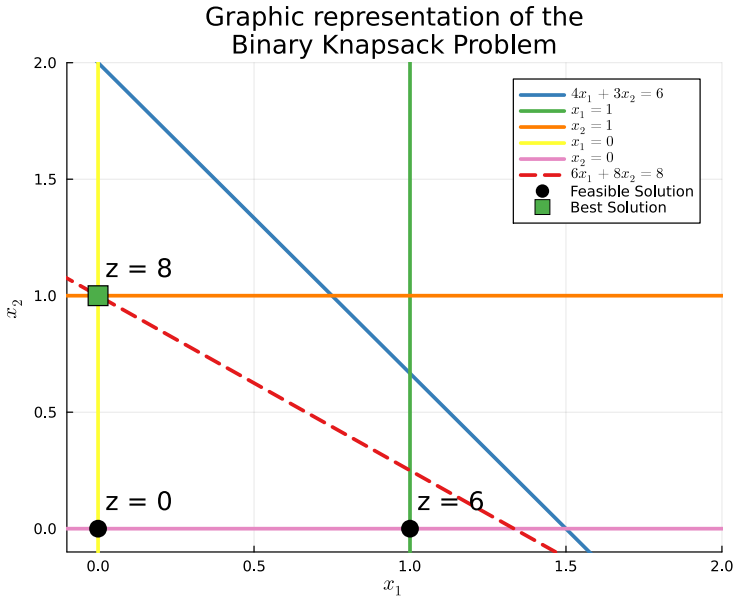


Figure 2.2: Graphical representation of the BKP solution.

2.3.2 Solving the Binary Knapsack Problem relaxation

The Linear Programming (LP) relaxation of the BKP is defined as:

Definition 2.2 — LP-BKP model.

$$\max_{x} \quad \sum_{i=1}^{n} v_i x_i \tag{2.2.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2.2.2}$$

$$x_i \in [0,1] \qquad i = 1, ..., n \tag{2.2.3}$$

where:

- v_i item i value
- x_i decision variables (select item fractionally)
- w_i item i weight
- W knapsack capacity

It can be observed that the only difference between the BKP model in Equation (2.1) and its LP relaxation is that the decision variables defined in Equation (2.2.3) can take any value between 0 and 1, allowing for fractional selection of items. This relaxation often provides an upper bound on the optimal value of the original BKP, and can be solved efficiently using linear programming techniques.

Listing 2.3 presents the Julia code to solve the LP relaxation of the BKP for the chapter's example. The code is similar to the one used for solving the BKP, with the main difference being that the decision variables are defined as continuous variables between 0 and 1 instead of binary variables (Listing 2.3-16).

```
1  using JuMP  # Modeling language
2  using HiGHS # Solver
3
4  # Problem data
5  v = [6, 8]     # item value
6  w = [4, 3]     # item weight
7  W = 6          # knapsack capacity
8  n = length(w)  # number of items
9
10 # LP relaxation model
11 model = JuMP.Model(HiGHS.Optimizer)
12
13 # Silent mode (solver output is not printed)
14 JuMP.set_silent(model)
15
16 # Decision variables
```

```julia
@variable(model, 0 <= x[1:n] <= 1)

# Objective
@objective(model, Max, sum(v[i] * x[i] for i=1:n))

# Constraint
@constraint(model, sum(w[i] * x[i] for i=1:n) <= W)

# Print model
print("Model:\n$model")

# Solve model
JuMP.optimize!(model)

# Display solution
println("Solution:")
for i in 1:n
    println("x[$i] = ", JuMP.value(x[i]))
end
println("Objective = ", JuMP.objective_value(model))
```

Listing 2.3: Solving the LP relaxation of BKP with Julia.

Listing 2.4 presents the output of the LP-BKP model, showing the mathematical formulation of the model and the solution obtained after optimization.

Output

```
Model:
Max 6 x[1] + 8 x[2]
Subject to
 4 x[1] + 3 x[2] <= 6
 x[1] >= 0
 x[2] >= 0
 x[1] <= 1
 x[2] <= 1
Solution:
x[1] = 0.75
x[2] = 1.0
Objective = 12.5
```

Listing 2.4: Output of the LP-BKP model.

Figure 2.3 shows a two-dimensional view of the problem. The hatched area represents the set of feasible solutions. For the LP relaxation, the optimal solution is $z_{\text{LP-BKP}} = 12.5$ for point $(0.75, 1)$.

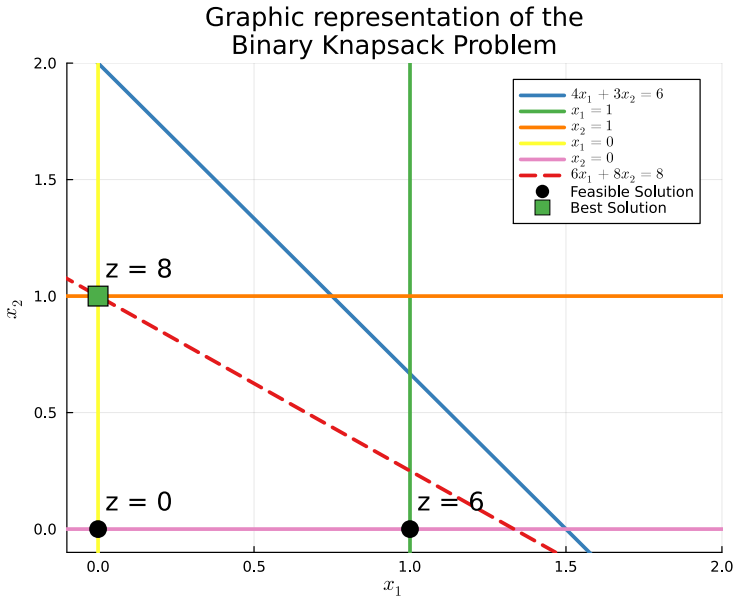


Figure 2.3: Graphical representation of the LP-BKP solution.

Note

It is well known in the literature [10] that the LP relaxation of this model can be solved by ordering the ratios $\frac{v_i}{w_i}$ in non-increasing order and selecting items accordingly (setting $x_i = 1$) until the capacity is reached.

Let k denote the index of the corresponding fractional variable, which is the only non-integer variable in the LP relaxation. Its value is computed as $x_k = \frac{W - \sum_{i=1}^{k-1} w_i}{w_k}$.

It can be observed from Figure 2.3 that the ordering considered for solving the LP relaxation was $\frac{v_2}{w_2} > \frac{v_1}{w_1}$, that is, $2.67 > 1.5$.

2.4 The Integer Knapsack Problem

We now consider the unbounded Integer Knapsack Problem (IKP). In this problem, you are allowed to select each item multiple times (Equation (2.3.3)), as long as the total weight does not exceed the knapsack's capacity. The IKP can be formulated as follows:

Definition 2.3 — IKP model.

$$\max_{x} \quad \sum_{i=1}^{n} v_i x_i \tag{2.3.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2.3.2}$$

$$x_i \in \mathbb{Z}_{\geq 0} \qquad i = 1, ..., n \tag{2.3.3}$$

where:

- v_i item i value
- x_i decision variables (select item multiple times)
- w_i item i weight
- W knapsack capacity

2.4.1 Solving the Integer Knapsack Problem

Listing 2.5 presents the Julia code to solve the IKP for the chapter's example. The code is similar to the one used for solving the BKP, with the main difference being that the decision variables are defined as non-negative integers instead of binary variables (Listing 2.5-16).

```
using JuMP  # Modeling language
using HiGHS # Solver

# Problem data
v = [6, 8]     # item value
w = [4, 3]     # item weight
W = 6          # knapsack capacity
n = length(w)  # number of items

# Create model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Decision variables
@variable(model, x[1:n] >= 0, Int)

# Objective
@objective(model, Max, sum(v[i] * x[i] for i=1:n))
```

```
21
22 # Constraint
23 @constraint(model, sum(w[i] * x[i] for i=1:n) <= W)
24
25 # Print model
26 print("Model:\n$model")
27
28 # Solve model
29 JuMP.optimize!(model)
30
31 # Display solution
32 println("Solution:")
33 for i in 1:n
34     println("x[$i] = ", JuMP.value(x[i]))
35 end
36 println("Objective = ", JuMP.objective_value(model))
```

Listing 2.5: Solving the IKP with Julia.

Listing 2.6 presents the output of the IKP model, showing the mathematical formulation of the model and the solution obtained after optimization.

Output

```
Model:
Max 6 x[1] + 8 x[2]
Subject to
 4 x[1] + 3 x[2] <= 6
 x[1] >= 0
 x[2] >= 0
 x[1] integer
 x[2] integer
Solution:
x[1] = 0.0
x[2] = 2.0
Objective = 16.0
```

Listing 2.6: Output of the IKP model.

Figure 2.4 illustrates a two-dimensional representation of the problem. The integer feasible points are highlighted. The optimal solution with value $z_{\text{IKP}} = 16$ corresponds to point $(0, 2)$.

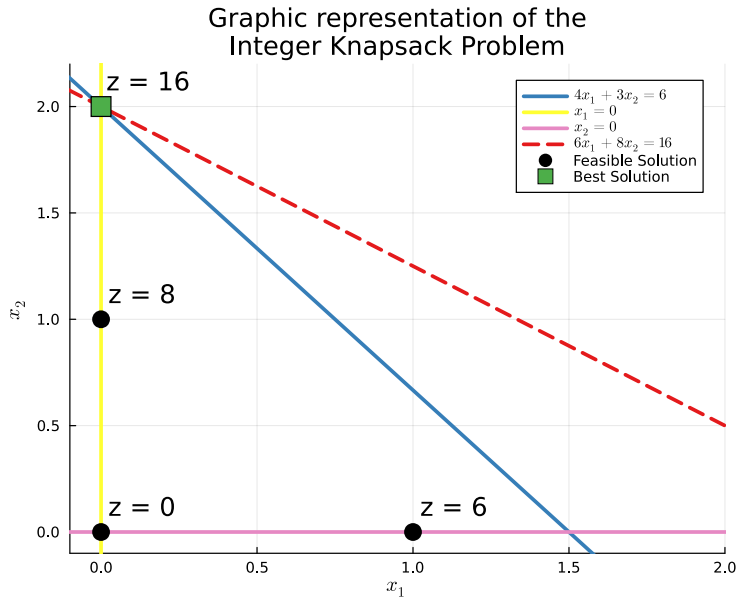


Figure 2.4: Graphical representation of the IKP solution.

2.4.2 Solving the Integer Knapsack Problem relaxation

Equation (2.4) is the LP relaxation of the IKP, where the decision variables can take any non-negative real value instead of being restricted to integers (Equation (2.4.3)). The formulation is as follows:

Definition 2.4 — LP-IKP model.

$$\max_{x} \quad \sum_{i=1}^{n} v_i x_i \tag{2.4.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2.4.2}$$

$$x_i \in \mathbb{R}_{\geq 0} \qquad i = 1, ..., n \tag{2.4.3}$$

where:

v_i	item i value
x_i	decision variables (select item fractionally)
w_i	item i weight
W	knapsack capacity

Listing 2.7 presents the Julia code to solve the LP relaxation of the IKP for the chapter's example. The code is similar to the one used for solving the IKP, with the main difference being that the decision variables are defined as non-negative continuous variables instead of integer variables (Listing 2.7-16).

```julia
using JuMP  # Modeling language
using HiGHS # Solver

# Problem data
v = [6, 8]     # item value
w = [4, 3]     # item weight
W = 6          # knapsack capacity
n = length(w)  # number of items

# LP relaxation model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Decision variables
@variable(model, x[1:n] >= 0)

# Objective
@objective(model, Max, sum(v[i] * x[i] for i=1:n))

# Constraint
@constraint(model, sum(w[i] * x[i] for i=1:n) <= W)

# Print model
print("Model:\n$model")

# Solve model
JuMP.optimize!(model)

# Display solution
println("Solution:")
for i in 1:n
    println("x[$i] = ", JuMP.value(x[i]))
end
println("Objective : ", JuMP.objective_value(model))
```

Listing 2.7: Solving the LP relaxation of IKP with Julia.

Listing 2.8 presents the output of the LP-IKP model, showing the mathematical formulation of the model and the solution obtained after optimization.

```
Output
Model:
Max 6 x[1] + 8 x[2]
Subject to
 4 x[1] + 3 x[2] <= 6
 x[1] >= 0
 x[2] >= 0
Solution:
x[1] = 0.0
x[2] = 2.0
Objective : 16.0
```

Listing 2.8: Output of the LP-IKP model.

Figure 2.5 illustrates a two-dimensional representation of the problem. The feasible region is highlighted, and the optimal solution with value $z_{\text{LP-IKP}} = 16$ corresponds to point $(0, 2)$, which is the same as the optimal solution of the original IKP in this case.

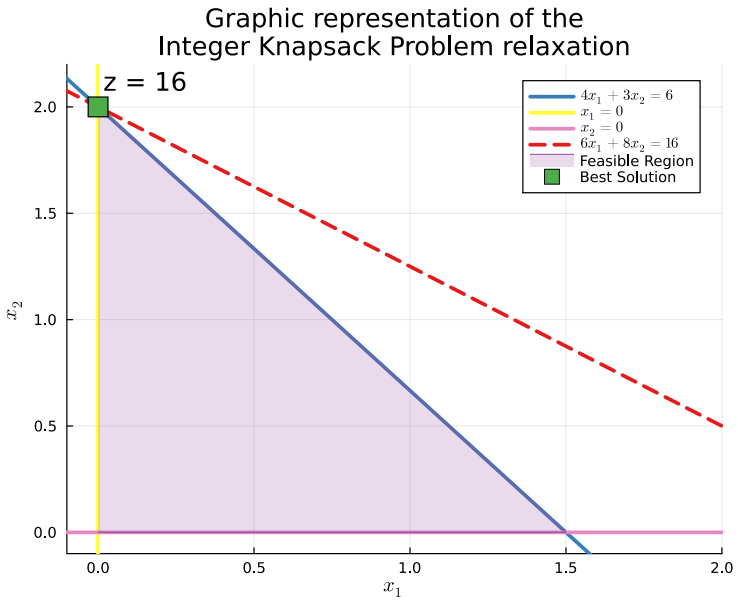


Figure 2.5: Graphical representation of the LP-IKP solution.

2.5 Final Remarks

In this chapter, we have provided an introductory study of knapsack problems, exploring their basic concepts and formulations. Readers are encouraged to

experiment with the solutions using the Julia programming language, which can help make the concepts more tangible and facilitate practical implementation.

Furthermore, in Chapter 8, we will introduce the Branch-and-Bound method, an exact technique widely used for solving combinatorial optimization problems, including the Knapsack Problem. This approach systematically explores the solution space, pruning branches that are unlikely to yield optimal solutions, and ultimately identifies the optimal solution to the Knapsack Problem.

2.6 Exercises

Exercise 2.1 Solve the following knapsack problems using the JuMP package. Create both binary, integer models and its relaxations. Compare the results obtained for each model and discuss the differences in the solutions and objective values.

a) Data:

- values: $v = (5, 7, 4, 3)$
- weights: $w = (16, 22, 12, 8)$
- capacity: $W = 30$

b) Data:

- values: $v = (4, 2, 10, 2, 1)$
- weights: $w = (12, 2, 4, 1, 1)$
- capacity: $W = 15$

c) Data:

- values: $v = (3, 9, 6, 3, 2)$
- weights: $w = (8, 5, 12, 6, 4)$
- capacity: $W = 17$

3 Location Problems

The logistics of distributing products and services have become increasingly important in strategic planning for both public and private sector organizations. Location problems play a key role in determining the optimal configuration for facility placement to meet the demands of a population [11].

Location problems involve decisions regarding where to install facilities in order to optimize specific criteria while serving customers. Facilities may include factories, warehouses, schools, and other similar entities, while customers may be warehouses, sales units, students, and so forth. These problems are also referred to as location-allocation problems, as they involve allocating customers to the facilities that are opened.



Figure 3.1: A policymaker analyzing optimal facility location. Source: image generated with ChatGPT.

The applications of facility location problems are widespread across both public and private sectors. In the public sector, the objective is often to maximize customer satisfaction while considering associated costs. Examples include the placement of schools, health posts, fire stations, ambulances, police vehicles, and bus stops. In the private sector, which typically involves fixed costs, applications often focus on the location of factories, warehouses, transmission towers, franchise stores, and similar facilities.

In some cases, constraints exist regarding the capacity of these facilities to provide services. Each customer has an associated demand that must be met by

the chosen facility, and the total demand served by a facility must not exceed its capacity. Problems that include such restrictions are referred to as location problems with capacity constraints. These constraints add complexity to the problem, requiring careful consideration when determining optimal facility locations.

In this chapter, we focus on three fundamental location problems:

- the P-Median Problem (PMP),
- the Set Covering Location Problem (SCLP),
- the Maximal Covering Location Problem (MCLP).

These problems are commonly encountered in the study of facility location. We begin with the PMP, which involves selecting p facilities (medians) from a given set of potential locations to minimize the total distance between each client and its nearest facility. We present the problem in detail and provide its mathematical formulation. To illustrate practical implementation, we demonstrate how to model the PMP using the Julia programming language.

Next, we examine the SCLP, which involves finding the minimum number of facilities required to cover a given set of clients. We describe its formulation and highlight its relevance in various applications. A Julia implementation of the set covering model is also provided.

Finally, we introduce the MCLP, which seeks to maximize the number of clients served within a given budget constraint. We explain its formulation and its relationship to the SCLP. A corresponding Julia implementation is provided to reinforce the concepts.

Throughout this chapter, the emphasis is on both understanding the theoretical aspects of these location problems and implementing them in Julia. By providing practical examples, readers gain hands-on experience in solving these fundamental facility location problems using a widely used optimization programming language.

3.1 The P-Median Problem

The P-Median problem (PMP) is a well-known location-allocation problem. It is used to determine the optimal placement of p facilities (like warehouses, hospitals, or service centers) to serve a set of n demand points, in such a way that the total distance (or cost) between demand points and their assigned facilities is minimized.

The PMP was first introduced by Hakimi [12], [13], [14], [15]. These early formulations of the problem set the foundation for further research and development in the field of location analysis.

The PMP has numerous practical applications, particularly in the areas of facility location planning, transportation and network optimization. By strategically selecting p centers, organizations can optimize their distribution networks, minimize transportation costs, and improve overall efficiency.

Over the years, researchers have developed various mathematical models, algorithms, and solution techniques to solve the PMP. These approaches aim to find an optimal or near-optimal solution that balances the trade-off between minimizing distances and selecting the appropriate number of facilities.

The PMP remains an active area of research, with ongoing efforts to develop new solution methods and explore its applications in different contexts. The problem's formulation and solutions continue to evolve, reflecting advancements in optimization techniques and the growing complexity of real-world location problems.

The PMP is combinatorial and NP-hard [16], which limits the size of the possible instances to be solved by exact methods. This fact justifies the development of heuristic methods to solve the problem [17], [18], [19]. Techniques based on Lagrangian relaxation, combined with subgradient optimization procedures, from a primal-dual point of view, also show great efficiency in solving this problem [20], [21], [22]. A column generation technique was successfully applied in [23], and a branch-and-price search appeared in the paper [24].

3.1.1 PMP applications

Table 3.1 presents a table summarizing real-world applications of the PMP across various domains.

Application	Description
🎓 **School location**	Finding the best location for schools [25].
🚑 **Ambulance location**	Finding the best location for ambulances [26].
🆘 **Disaster relief**	Hurricane disaster relief distribution [27].
♻️ **Recycling**	Optimize the location of ecopoints for waste oil and fats collection [28].
🔥 **Geostrategic location**	Find the location of natural gas sources [29].

Application	Description
Electrical distribution	Network design of electrical distribution systems [30].

Table 3.1: Real-world applications of the P-Median Problem.

The following sections will present two variants of the PMP:

- PMP-C: **classical PMP**, in which we work with separate locations for clients and facilities (client locations I, facility locations J)
- PMP-N: **PMP on a single set of locations**, in which clients and facilities share the same set of locations N

3.1.2 PMP-C: classical PMP

Equation (3.1) presents the Integer Programming formulation of the **classical PMP** formulation. The problem is modeled using two distinct sets: one set for clients $I = \{1, ..., n\}$ and another set for facilities $J = \{1, ..., m\}$.

Definition 3.1 — PMP-C model.

$$\min_{x,y} \quad \sum_{i=1}^{n}\sum_{i=1}^{m} d_{ij}x_{ij} \tag{3.1.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{m} y_j = p \tag{3.1.2}$$

$$\sum_{j=1}^{m} x_{ij} = 1 \qquad i = 1, ..., n \tag{3.1.3}$$

$$x_{ij} \leq y_j \qquad i = 1, ..., n, \quad j = 1, ..., m \tag{3.1.4}$$

$$x_{ij}, y_j \in \{0, 1\} \quad i = 1, ..., n, \quad j = 1, ..., m \tag{3.1.5}$$

where:

d_{ij}	distance between client i and facility j
x_{ij}	decision variables (assign client i to facility j)
y_j	decision variables (open facility j)
p	number of facilities to open

In Equation (3.1), the decision variable y_j is a binary variable that indicates whether facility j is opened (1) or not (0). The variable x_{ij} is also a binary variable that indicates whether client i is assigned to facility j (1) or not (0). The objective

function minimizes the total distance between clients and their assigned facilities (Equation (3.1.1)), while the constraints ensure that exactly p facilities are opened (Equation (3.1.2)), each client is assigned to exactly one facility (Equation (3.1.3)), and clients can only be assigned to open facilities (Equation (3.1.4)).

The PMP-C model can be solved by commercial solvers for relatively large instances (e.g., more than 1,000 points) within reasonable computational times.

3.1.2.1 Solving the PMP-C

To solve the PMP-C, we will use the following example.

Example 3.1 Consider an instance of PMP-C composed of $|I| = 92$ demand points and $|J| = 10$ facilities locations and $p = 3$ (Figure 3.2).



Figure 3.2: PMP-C instance with 10 facilities candidates and 92 demand points.

Listing 3.1 presents the Julia code to solve the PMP-C instance shown in Figure 3.2. The code reads the client and facility coordinates from CSV files (Listing 3.1-9), computes the distance matrix using the Haversine formula (Listing 3.1-17), and then formulates and solves the optimization model using JuMP and HiGHS (Listing 3.1-23). Finally, it extracts the solution (Listing 3.1-48) and plots it using a custom function defined in "utils/pmp-c_utils.jl" (Listing 3.1-68).

```
1 using JuMP      # Modeling language
2 using HiGHS     # Solver
3 using CSV       # Data handling
4 using Distances # Distance computations
5
```

```
# Include utility functions for plotting the solution
include("utils/pmp-c_utils.jl")

# Load latitude and longitude for facilities and clients
clients_coordinates = CSV.read("data/pmp_clients.csv",
CSV.Tables.matrix, header=false)
facilities_coordinates = CSV.read("data/pmp_facilities.csv",
CSV.Tables.matrix, header=false)

# Problem dimensions
n = size(clients_coordinates, 1)    # Number of clients
m = size(facilities_coordinates, 1) # Number of facilities

# Use Haversine distance to compute distance matrix
D = Distances.pairwise(Distances.Haversine(), clients_coordinates,
facilities_coordinates, dims=1)

# Number of facilities to select
p = 3

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Decision variables
@variable(model, y[1:m], Bin)
@variable(model, x[1:n, 1:m], Bin)

# Objective: minimize total distance
@objective(model, Min, sum(D[i, j] * x[i, j] for i in 1:n, j in
1:m))

# Constraint: exactly p facilities selected
@constraint(model, sum(y) == p)

# Constraint: each client assigned to one facility
@constraint(model, [i in 1:n], sum(x[i, j] for j in 1:m) == 1)

# Constraint: assignment only to open facilities
```

```
@constraint(model, [i in 1:n, j in 1:m], x[i, j] <= y[j])

# Solve the model
JuMP.optimize!(model)

# Extract solution
y_value = JuMP.value.(y)
x_value = JuMP.value.(x)
objective_value = JuMP.objective_value(model)

# Print objective value
println("Objective value: $objective_value meters")

# Print selected facilities
facility_ids = findall(y_value .> 0.5)
println("Selected Facilities (p=$p): $facility_ids")

# Print total client assignments for each selected facility
assignments = Dict()
for facility_id in facility_ids
    clients = findall(x_value[:, facility_id] .> 0.5)
    assignments[facility_id] = clients
    println("Facility: $facility_id | $(length(clients)) clients assigned")
end

# Plot the solution
map = plot_solution(clients_coordinates, facilities_coordinates, assignments, p)
display(map)
```

Listing 3.1: PMP-C Julia implementation.

Listing 3.2 presents the output of the PMP-C model, showing the solution obtained after optimization.

Output

```
Objective value: 20591.95914310779 meters
Selected Facilities (p=3): [5, 7, 8]
Facility: 5 | 43 clients assigned
Facility: 7 | 16 clients assigned
```

```
Facility: 8 | 33 clients assigned
```

Listing 3.2: Output of the PMP-C model.

Figure 3.3 presents the map of the solution for the PMP-C instance. The map visually represents the locations of the clients and the selected facilities, along with the assignments of clients to their nearest open facilities.

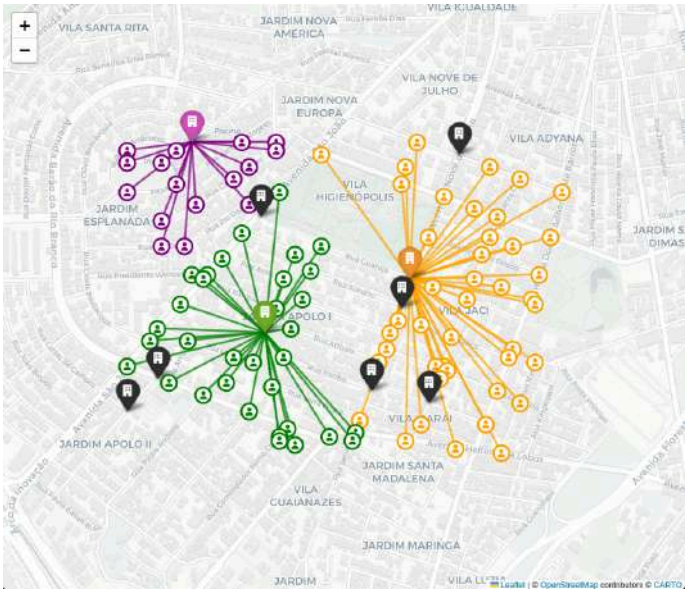


Figure 3.3: Optimal solution for PMP-C.

3.1.3 PMP-N: PMP on a single node set

The PMP-N formulation refers to a variant of the PMP in which both clients and facilities are chosen from the same set of locations (N), rather than from separate sets. This approach is particularly relevant in real-world scenarios where it is natural or necessary for service centers to be located at existing client sites, such as when selecting hospital locations among towns, distribution centers among retail outlets, or polling stations among neighborhoods.

Equation (3.2) presents the PMP-N model. The objective remains to select p facilities such that the total distance (or cost) of assigning each client to a facility is minimized.

Definition 3.2 — PMP-N model.

$$\min_{x,y} \quad \sum_{i=1}^{n}\sum_{i=1}^{m} d_{ij}x_{ij} \tag{3.2.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{m} x_{ij} = p \tag{3.2.2}$$

$$\sum_{j=1}^{m} x_{ij} = 1 \qquad i = 1, ..., n \tag{3.2.3}$$

$$x_{ij} \leq x_{ij} \qquad i = 1, ..., n, \quad j = 1, ..., m \tag{3.2.4}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, ..., n, \quad j = 1, ..., m \tag{3.2.5}$$

where:

- d_{ij} distance between client i and facility j
- x_{ij} decision variables (assign client i to facility j)
- p number of facilities to open

3.1.3.1 Solving the PMP-N

To solve the PMP-N, we will use the following example.

Example 3.2 Consider an instance of PMP-N composed of $|N| = 102$ locations that may serve both as client and as a potential facility site and $p = 3$ (Figure 3.4).



Figure 3.4: PMP-N instance with 102 locations.

Listing 3.3 presents the Julia code to solve the PMP-N instance shown in Figure 3.4. The code reads the coordinates for all locations from a CSV file (Listing 3.3-9), computes the distance matrix using the Haversine formula (Listing 3.3-15), and then formulates and solves the optimization model using JuMP and HiGHS (Listing 3.3-21). Finally, it extracts the solution (Listing 3.3-45) and plots it using a custom function defined in "utils/pmp-n_utils.jl" (Listing 3.3-64).

```
1  using JuMP      # Modeling language
2  using HiGHS     # Solver
3  using CSV       # Data handling
4  using Distances # Distance computations
5
6  # Include utility functions for plotting the solution
7  include("utils/pmp-n_utils.jl")
8
9  # Load latitude and longitude for all points
10 coordinates = CSV.read("data/coordinates.csv", CSV.Tables.matrix,
   header=false)
11
12 # Problem dimensions
13 n = size(coordinates, 1) # Number of points
14
15 # Use Haversine distance to compute distance matrix
16 D = Distances.pairwise(Distances.Haversine(), coordinates, dims=1)
17
18 # Number of facilities to select
19 p = 3
20
21 # Create the model
22 model = JuMP.Model(HiGHS.Optimizer)
23
24 # Silent mode (solver output is not printed)
25 JuMP.set_silent(model)
26
27 # Decision variables
28 @variable(model, x[1:n, 1:n], Bin)
29
30 # Objective: minimize total distance
31 @objective(model, Min, sum(D[i, j] * x[i, j] for i in 1:n, j in
   1:n))
32
```

```
# Constraint: exactly p facilities selected
@constraint(model, sum(x[j, j] for j in 1:n) == p)

# Constraint: each client assigned to one facility
@constraint(model, [i in 1:n], sum(x[i, j] for j in 1:n) == 1)

# Constraint: assignment only to open facilities
@constraint(model, [i in 1:n, j in 1:n], x[i, j] <= x[j, j])

# Solve the model
JuMP.optimize!(model)

# Extract solution
x_value = JuMP.value.(x)
objective_value = JuMP.objective_value(model)

# Print objective value
println("Objective value: $objective_value meters")

# Print selected facilities
facility_ids = findall(x_value[j, j] .> 0.5 for j in 1:n)
println("Selected Facilities (p=$p): $facility_ids")

# Print total client assignments for each selected facility
assignments = Dict()
for facility_id in facility_ids
    clients = findall(x_value[:, facility_id] .> 0.5)
    assignments[facility_id] = clients
    println("Facility: $facility_id | $(length(clients)) clients
assigned")
end

# Plot the solution
map = plot_solution(coordinates, assignments, p)
display(map)
```

Listing 3.3: PMP-N Julia implementation.

Listing 3.4 presents the output of the PMP-N model, showing the solution obtained after optimization.

```
Output
Objective value: 19211.026718733905 meters
Selected Facilities (p=3): [81, 92, 99]
Facility: 81 | 44 clients assigned
Facility: 92 | 26 clients assigned
Facility: 99 | 32 clients assigned
```

Listing 3.4: Output of the PMP-N model.

Figure 3.5 presents the map of the solution for the PMP-N instance. The map visually represents the locations of the clients and the selected facilities, along with the assignments of clients to their nearest open facilities.

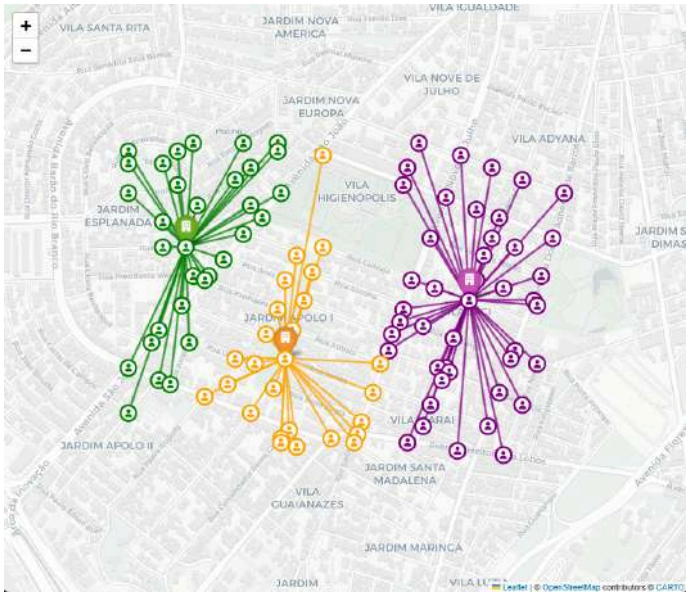


Figure 3.5: Optimal solution for PMP-N.

3.2 Covering problems

The Set Covering Location Problem (SCLP) and the Maximal Coverage Location Problem (MCLP) are two important problems in location analysis.

The SCLP, proposed by [31], focuses on finding the minimum number of facilities and their optimal locations to ensure that all demand points are within a critical service distance from at least one facility. The critical distance represents the maximum allowable distance or travel time that a user can travel to reach the nearest facility. The goal of the SCLP is to minimize the number of facilities while satisfying the coverage requirement for all demand points.

In some situations, it may not be feasible or economically viable to fully cover all demand points. In such cases, the MCLP, formulated by [32], offers an alternative approach. The MCLP aims to determine the locations for a predetermined number of facilities that maximize the coverage of the population, while ensuring that no demand points exceed the critical distance. Instead of focusing on complete coverage, the MCLP prioritizes maximizing the number of covered customers within the given constraints.

Both the SCLP and MCLP are important location problems that have numerous applications in various industries and sectors. These problems play a crucial role in strategic planning and decision-making processes, helping organizations optimize their facility locations to efficiently serve their target population while considering resource limitations and coverage constraints.

3.2.1 SCLP

Equation (3.3) presents the Integer Programming formulation of the SCLP. The model is defined using a single set of locations $N = \{1, ..., n\}$, which represents both potential facility sites and demand points. The decision variable x_j is a binary variable that indicates whether a facility is opened at location j (1) or not (0). The objective function minimizes the total number of facilities opened (Equation (3.3.1)), while the constraints ensure that each demand point i is covered by at least one open facility within the critical distance (Equation (3.3.2)). The coverage matrix c_{ij} indicates whether demand point i is covered by a facility at location j (1 if covered, 0 otherwise).

Definition 3.3 — SCLP model.

$$\min_{x} \quad \sum_{j=1}^{n} x_j \tag{3.3.1}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} c_{ij} x_j \geq 1 \quad i = 1, ..., n \tag{3.3.2}$$

$$x_j \in \{0, 1\} \quad j = 1, ..., n \tag{3.3.3}$$

where:

- x_i decision variables (select i as center)
- c_{ij} coverage matrix (client i is covered by facility j)

3.2.1.1 SCLP applications

Table 3.2 presents a wide range of practical applications for the SCLP. These applications help organizations minimize costs while ensuring essential services are accessible to the entire population.

Application	Description
🆘 **Emergency services**	Determining optimal locations for fire stations, hospitals, and police stations to ensure all areas are within a critical response distance.
📡 **Telecommunications**	Placing cell towers or Wi-Fi hotspots to guarantee coverage for all users in a region.
🚌 **Public transportation**	Locating bus stops or train stations so that all neighborhoods have access within a reasonable walking distance.
🚚 **Retail and logistics**	Selecting warehouse or store locations to serve all customers efficiently.
💧 **Utilities**	Siting water, electricity, or waste collection facilities to cover all demand points.

Table 3.2: Real-world applications of the Set Covering Location Problem.

3.2.1.2 Solving the SCLP

To solve the PMP-N, we will use the following example.

Example 3.3 Consider an instance of SCLP composed of $|N| = 102$ locations and $r = 200$ meters(Figure 3.6).

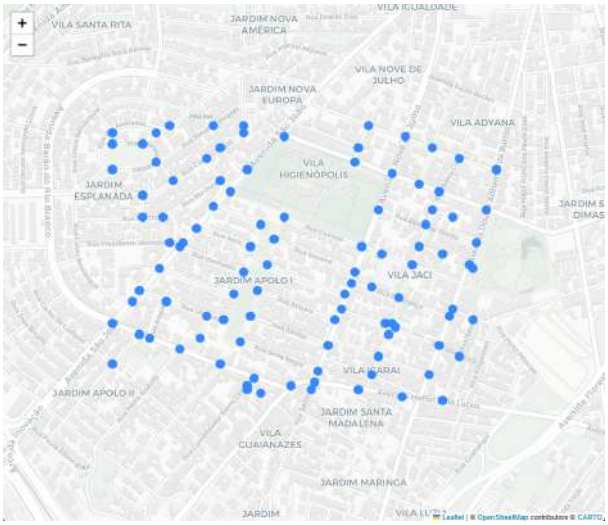


Figure 3.6: SCLP instance with 102 locations.

The first step before solving the SCLP is preprocessing the input data to create the coverage matrix $C = [c_{ij}]$:

$$c_{ij} = \begin{cases} 1 \text{ if facility } j \text{ can cover demand point } i \\ 0 \text{ otherwise} \end{cases} \tag{3.4}$$

Finally, we can model SCLP using JuMP. Listing 3.5 presents the code to solve the SCLP instance shown in Figure 3.6. The code reads the coordinates for all locations from a CSV file (Listing 3.5-8), computes the distance matrix (Listing 3.5-11), and then creates the coverage matrix based on a critical distance of $r = 200$ meters (Listing 3.5-17). Finally, it formulates and solves the optimization model using JuMP and HiGHS (Listing 3.5-23). After solving, it extracts the solution (Listing 3.5-41) and plots it using a custom function defined in "utils/sclp_utils.jl" (Listing 3.5-51).

```
1  using JuMP        # Modeling language
2  using HiGHS       # Solver
3  using CSV         # For reading CSV files
4
5  # Include utility functions for plotting the solution
6  include("utils/sclp_utils.jl")
7
8  # Load coordinates
9  coordinates = CSV.read("data/coordinates.csv", CSV.Tables.matrix,
   header=false)
10
11 # Create distance matrix
12 D = create_distance_matrix(coordinates)
13
14 # Define coverage radius (in meters)
15 radius = 200
16
17 # Create coverage matrix considering the defined radius
18 C = D .< radius
19
20 # Number of demand/facility points
21 n = size(D, 1)
22
23 # Create model
24 model = JuMP.Model(HiGHS.Optimizer)
25
26 # Silent mode (solver output is not printed)
```

```julia
JuMP.set_silent(model)

# Variables
@variable(model, x[1:n], Bin)

# Objective: minimize number of facilities opened
@objective(model, Min, sum(x))

# Constraint: each demand point is covered by at least one facility
@constraint(model, [i in 1:n], sum(C[i,j] * x[j] for j in 1:n) >= 1)

# Solve the model
JuMP.optimize!(model)

# Get solution vector
x_vals = JuMP.value.(x)

# Get indices of selected facilities
selected_facilities = findall(x_vals .> 0.5)

# Display solution
println("Number of facilities opened: ",
Int(JuMP.objective_value(model)))
println("Selected facility indices: ", selected_facilities)

# Display solution on map
map = plot_solution(selected_facilities, coordinates, radius)
display(map)
```

Listing 3.5: SCLP Julia implementation.

Listing 3.6 presents the output of the SCLP model, showing the solution obtained after optimization.

Output

```
Number of facilities opened: 11
Selected facility indices: [3, 10, 29, 30, 36, 47, 57, 58, 62, 88, 89]
```

Listing 3.6: Output of the SCLP model.

Figure 3.7 presents the map of the solution for the SCLP instance. The map visually represents the locations of the demand points and the selected facilities, along with the coverage areas defined by the critical distance.

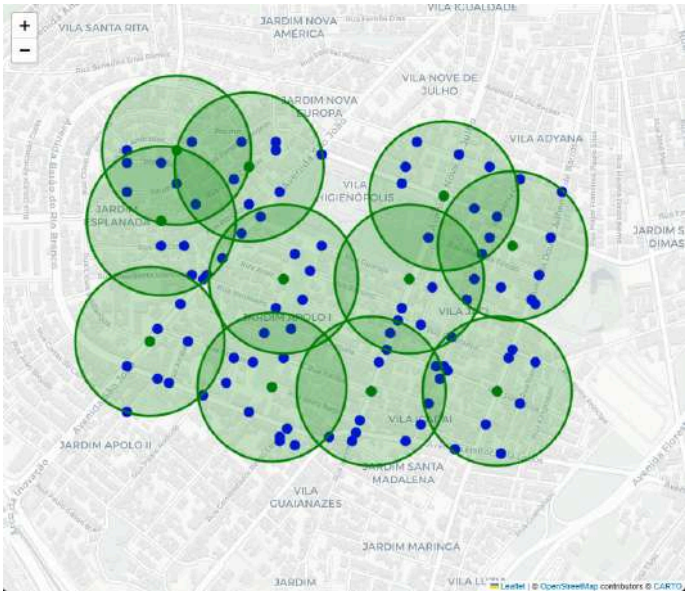


Figure 3.7: Optimal solution for SCLP.

3.2.2 The MCLP model

Equation (3.5) presents the Integer Programming formulation of the MCLP proposed by Church and ReVelle [32]. The model is defined using a single set of locations $N = \{1, ..., n\}$, which represents both potential facility sites and demand points.

The decision variable y_i is a binary variable that indicates whether demand point i is covered (1) or not (0), while x_j is a binary variable that indicates whether a facility is opened at location j (1) or not (0). The objective function maximizes the total demand covered (Equation (3.5.1)), while the constraints ensure that exactly p facilities are opened (Equation (3.5.2)) and that each demand point can only be covered if at least one facility that covers it is opened (Equation (3.5.3)). The coverage matrix c_{ij} indicates whether demand point i is covered by a facility at location j (1 if covered, 0 otherwise).

The MCLP model can be solved in commercial solvers for relatively large problems (greater than 1000 points) in reasonable time. New constraints can be included to obtain other variations of the MCLP model, such as the capacitated [33], with probabilistic coverage [34] or with cooperative covering [35].

Definition 3.4 — MCLP model.

$$\max_{x,y} \quad \sum_{i=1}^{n} d_i y_i \tag{3.5.1}$$

$$\text{s.t.} \quad \sum_{i=j}^{n} x_j = p \tag{3.5.2}$$

$$\sum_{j=1}^{n} c_{ij} x_j \geq y_i \quad i = 1, ..., n \tag{3.5.3}$$

$$x_j, y_i \in \{0, 1\} \quad i = 1, ..., n \tag{3.5.4}$$

where:

- d_i demand at client i
- y_i decision variables (client i is covered)
- x_j decision variables (select j as facility)
- p number of facilities to open
- c_{ij} coverage matrix (client i is covered by facility j)

3.2.2.1 MCLP applications

Table 3.3 presents a wide range of practical applications for the MCLP. These applications help organizations maximize their service reach and impact when resources are limited and full coverage is not feasible.

Application	Description
Healthcare	Placing a fixed number of clinics or vaccination centers to maximize the population served within a critical distance.
Emergency services	Locating a limited number of fire stations or ambulances to maximize coverage of high-risk areas.
Telecommunications	Deploying a set number of cell towers or Wi-Fi hotspots to maximize user coverage in urban or rural regions.
Retail and logistics	Selecting store or warehouse locations to maximize the number of customers served within a delivery radius.
Public safety	Siting surveillance cameras or police patrol zones to maximize coverage of vulnerable locations.

Table 3.3: Real-world applications of the Maximal Coverage Location Problem.

3.2.2.2 Solving the MCLP

To solve the MCLP, we will use the same instance used for previously solving the SCLP (Figure 3.6), considering a demand of 1 for each point, a critical distance of $r = 200$ meters, and $p = 3$ facilities to be opened.

Listing 3.7 presents the Julia code to solve the MCLP instance shown in Figure 3.6. The code reads the coordinates for all locations from a CSV file (Listing 3.7-8), computes the distance matrix (Listing 3.7-11), and then creates the coverage matrix based on a critical distance of $r = 200$ meters (Listing 3.7-17). Finally, it formulates and solves the optimization model using JuMP and HiGHS (Listing 3.7-23). After solving, it extracts the solution (Listing 3.7-41) and plots it using a custom function defined in "utils/mclp_utils.jl" (Listing 3.7-61).

```
1  using JuMP       # Modeling language
2  using HiGHS      # Solver
3  using CSV        # For reading CSV files
4
5  # Include utility functions for plotting the solution
6  include("utils/mclp_utils.jl")
7
8  # Load coordinates
9  coordinates = CSV.read("data/coordinates.csv", CSV.Tables.matrix,
   header=false)
10
11 # Create distance matrix
12 D = create_distance_matrix(coordinates)
13
14 # Define coverage radius (in meters)
15 radius = 200
16
17 # Number of facilities to select
18 p = 7
19
20 # Create coverage matrix considering the defined radius
21 C = D .< radius
22
23 # Number of demand/facility points
24 n = size(C, 1)
25
26 # Create a vector of equal demand for each point
27 d = ones(n)
```

```
28
29 # Create model
30 model = JuMP.Model(HiGHS.Optimizer)
31
32 # Silent mode (solver output is not printed)
33 JuMP.set_silent(model)
34
35 # Variables
36 @variable(model, x[1:n], Bin)
37 @variable(model, y[1:n], Bin)
38
39 # Objective: maximize covered demand
40 @objective(model, Max, sum(d[i]*y[i] for i in 1:n))
41
42 # Constraint: exactly p facilities are opened
43 @constraint(model, sum(x) == p)
44
45 # Constraint: each demand point is covered by at least one facility
46 @constraint(model, [i in 1:n], sum(C[i,j] * x[j] for j in 1:n) >=
   y[i])
47
48 # Solve the model
49 JuMP.optimize!(model)
50
51 # Get solution vectors
52 x_vals = JuMP.value.(x)
53 y_vals = JuMP.value.(y)
54
55 # Get indices of selected facilities
56 selected_facilities = findall(xi -> xi > 0.5, x_vals)
57 covered_points = findall(yi -> yi > 0.5, y_vals)
58
59 # Calculate coverage percentage
60 total_covered = length(covered_points)
61 coverage_percentage = round(100*total_covered/n, digits=2)
62
63 # Display solution
64 println("Facilities opened: $selected_facilities")
65 println("Points covered: $total_covered/$n ($coverage_percentage
   %)")
66
```

```
# Display solution on map
map = plot_solution(selected_facilities,covered_points, coordinates, radius)
display(map)
```

Listing 3.7: MCLP Julia implementation.

Listing 3.8 presents the output of the MCLP model, showing the solution obtained after optimization. The output includes the indices of the facilities that were opened and the number and percentage of demand points that are covered by these facilities.

Output

```
Facilities opened: [14, 28, 52, 59, 81, 82, 100]
Points covered: 90/102 (88.24 %)
```

Listing 3.8: Output of the MCLP model.

Figure 3.8 presents the map of the solution for the MCLP instance. The map visually represents the locations of the demand points, the selected facilities, and the coverage areas defined by the critical distance. It also indicates which demand points are covered by the opened facilities (green).

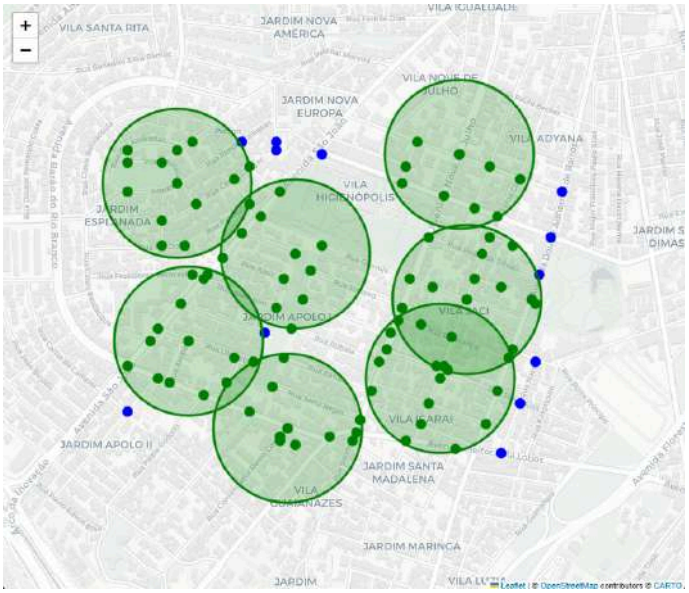


Figure 3.8: Optimal solution for MCLP.

Solving the MCLP with random demands

In Listing 3.9 we define the MCLP with random demands. The code is similar to the one presented in Listing 3.7, but instead of using a vector of equal demand for each point, we generate a random demand vector using the `rand` function (Listing 3.9-30) from the `Random` package (Listing 3.9-4) after setting a seed (Listing 3.9-6). The rest of the code remains unchanged, allowing us to solve the MCLP instance with random demands and visualize the solution on a map.

```
1  using JuMP   # Modeling language
2  using HiGHS  # Solver
3  using CSV    # For reading CSV files
4  using Random # For generating random demands
5
6  # Seed to create random demand
7  Random.seed!(42)
8
9  # Include utility functions for plotting the solution
10 include("utils/mclp_utils.jl")
11
12 # Load coordinates
13 coordinates = CSV.read("data/coordinates.csv", CSV.Tables.matrix,
   header=false)
14
15 # Create distance matrix
16 D = create_distance_matrix(coordinates)
17
18 # Define coverage radius (in meters)
19 radius = 200
20
21 # Number of facilities to select
22 p = 7
23
24 # Create coverage matrix considering the defined radius
25 C = D .< radius
26
27 # Number of demand/facility points
28 n = size(C, 1)
29
30 # Create a vector with a random demands
31 d = rand(n)
32
```

```
33 # Create model
34 model = JuMP.Model(HiGHS.Optimizer)
35
36 # Silent mode (solver output is not printed)
37 JuMP.set_silent(model)
38
39 # Variables
40 @variable(model, x[1:n], Bin)
41 @variable(model, y[1:n], Bin)
42
43 # Objective: maximize covered demand
44 @objective(model, Max, sum(d[i]*y[i] for i in 1:n))
45
46 # Constraint: exactly p facilities are opened
47 @constraint(model, sum(x) == p)
48
49 # Constraint: each demand point is covered by at least one facility
50 @constraint(model, [i in 1:n], sum(C[i,j] * x[j] for j in 1:n) >=
   y[i])
51
52 # Solve the model
53 JuMP.optimize!(model)
54
55 # Get solution vectors
56 x_vals = JuMP.value.(x)
57 y_vals = JuMP.value.(y)
58
59 # Get indices of selected facilities
60 selected_facilities = findall(xi -> xi > 0.5, x_vals)
61 covered_points = findall(yi -> yi > 0.5, y_vals)
62
63 # Calculate coverage percentage
64 total_covered = length(covered_points)
65 coverage_percentage = round(100*total_covered/n, digits=2)
66
67 # Display solution
68 println("Facilities opened: $selected_facilities")
69 println("Points covered: $total_covered/$n ($coverage_percentage
   %)")
70
71 # Display solution on map
```

```
map = plot_solution(selected_facilities,covered_points, coordinates,
radius)
display(map)
```

Listing 3.9: MCLP with random demands.

Listing 3.10 presents the output of the MCLP model, showing the solution obtained after optimization. The output includes the indices of the facilities that were opened and the number and percentage of demand points that are covered by these facilities. Note that the solution is different from the one obtained in Listing 3.8 due to the random generation of demands.

Output

```
Facilities opened: [14, 18, 47, 57, 59, 82, 98]
Points covered: 85/102 (83.33 %)
```

Listing 3.10: Output of the MCLP model.

Figure 3.9 presents the map of the solution for the MCLP instance with random demands. The map visually represents the locations of the demand points, the selected facilities, and the coverage areas defined by the critical distance. It also indicates which demand points are covered by the opened facilities (green). We can observe that the solution is different from Figure 3.8 in terms of selected centers.

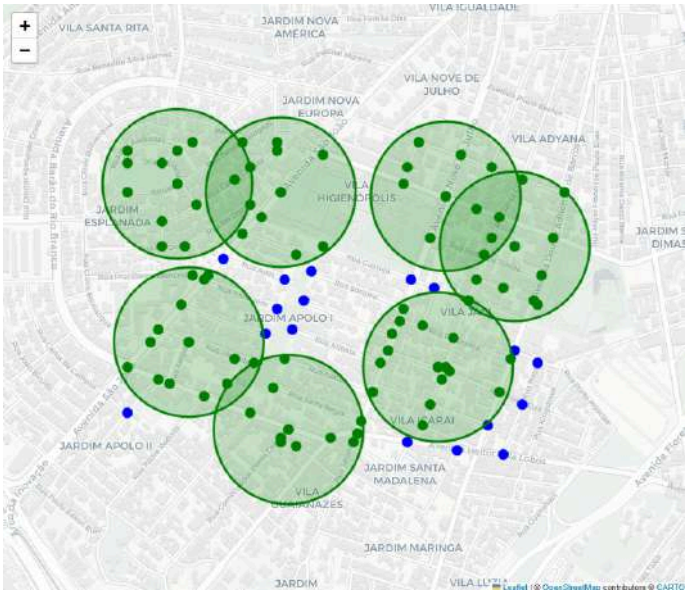


Figure 3.9: Optimal solution for MCLP with random demands.

3.3 Final remarks

Location problems have a wide range of applications and can be challenging to solve due to their complexity. In this chapter, we have introduced the PMP and two coverage problems, which are fundamental types of location problems. The PMP focuses on finding the optimal locations for a given number of facilities to minimize the distance between each customer and its closest facility. On the other hand, the coverage problems aim to determine the optimal locations of facilities to cover a set of points or areas.

We encourage the readers to explore and solve the exercises proposed in this chapter. By solving these exercises, you will gain understanding of the concepts and techniques used in location problems.

3.4 Exercises

Use the coordinates in file coordinates.csv to answer exercises 1-6. Implement the models using Julia with JuMP package.

Exercise 3.1 Formulate the P-Median Problem (PMP-N) and solve for 4 medians.

Exercise 3.2 Formulate the problem **Fault-tolerant P-Median Problem** [36] and solve for 3 medians (p) and 2 for the redundancy parameter (r).

Exercise 3.3 Formulate a coverage problem that finds the minimum number of centers that must be opened so that all other points are covered (SCLP), considering a coverage radius of 400 meters.

Exercise 3.4 Formulate the SCLP mathematical model from question 3 considering the intersections (overlap) between the covers, using the approach presented in the article [37], and a covering radius of 400 meters.

a) Solve using $z = 0$ for the overlap parameter;

b) Solve using $z = 1$ for the overlap parameter.

Exercise 3.5 Formulate a coverage problem that locates exactly three centers that maximize the demand covered in the other centers (MCLP), considering a coverage radius of 400 meters.

Exercise 3.6 Formulate the MCLP mathematical model from question 5 considering the intersections (overlap) between the covers, using the approach presented in the article [37], and a covering radius of 400 meters.

a) Solve using $z = 0$ for the overlap parameter;

b) Solve using $z = 1$ for the overlap parameter.

Exercise 3.7 Formulate the mathematical model known as TEAM model [38]. In this problem you need to allocate two type of ambulances to facilities so that the coverage of the demand is maximized. Figure 3.10 provides a graphical representation of the instance you will use. The districts with the population represents the demands, and the ambulance markers represents the facilities in which the ambulances can be placed. There are two type of ambulances:

- Basic care: provides emergency medical care to patients with non-life-threatening injuries or illnesses
- Advanced care: designed to provide care that is comparable to what a patient would receive in an emergency department

In our problem, basic care must be provided (the ambulance leaves the base and arrives at the point of care) in, for example, a maximum of 15 minutes. Advanced service must be carried out in a maximum of 25 minutes. If we have 4 ambulances for basic care and 2 for advanced care, find the best location for greater coverage of population.

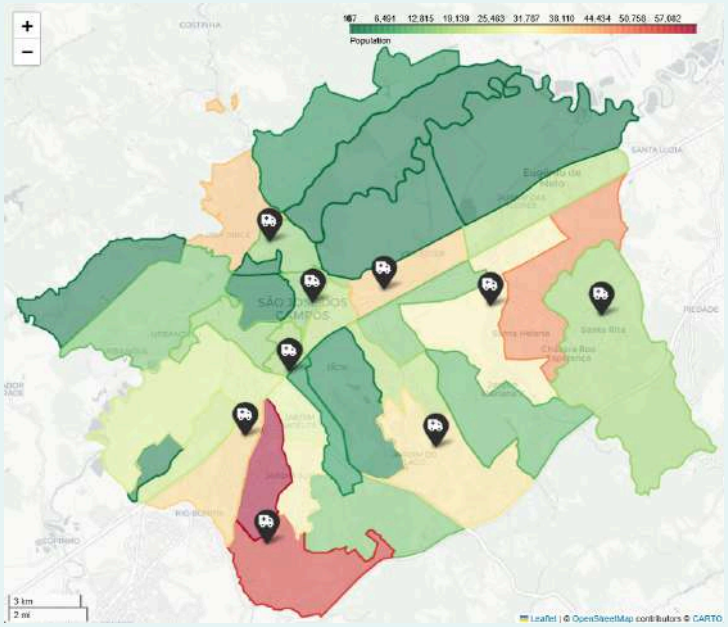


Figure 3.10: Graphical representation of the ambulance allocation problem.

The data used as input for TEAM model can be found sjc_ambulances.h5. It is in HDF5 format, stored with the structure presented in TABLE. The 'duration_matrix' was created calculating the travel time between the facilities and the centroid of each district using Openrouteservice API.

Data	Description
`districts_population`	Population of each district
`districts_geometry`	District polygons in wkt format
`facilities_geometry`	Points (lat,lon) of facilities in wkt format
`duration_matrix`	Route duration (minutes) between facilities and district centroids

Table 3.4: Dataset structure of the input file for the TEAM model.

4 Traveling Salesman Problem

In the last two decades, the operational, tactical and strategic aspects of the transport area have been presented as one of the areas benefited by the application of quantitative techniques to improve productivity. Fleet routing and scheduling problems are the main elements of many logistics systems, receiving much attention from researchers interested in finding good solutions for such complex combinatorial optimization problems. The vehicle routing problem intends to define routes for the vehicles, determining the customers to be visited, in a way that does not violate the restrictions and optimizes some objective function.

The **Traveling Salesman Problem** classic description (Figure 4.1) considers a set of cities, the distances between them and the "salesman" that must visit all of them, without passing through any of them more than once, and return to the origin traveling the shortest possible distance [39].



Figure 4.1: A traveling salesman planning his trip. Source: image generated with ChatGPT.

Obtaining an exact solution to this problem for a large number of cities can be computationally intractable. In these cases, heuristics are used. The web page https://www.math.uwaterloo.ca/tsp/ describes recent advances in TSP exact and heuristic solution.

4.1 TSP applications

Table 4.1 presents a wide range of practical applications for the TSP. These applications span across various industries and domains, demonstrating the versatility and importance of the TSP in solving real-world problems.

Application	Description
🚚 **Retail & logistics**	Design efficient routes in courier, food delivery, and ride-sharing services. It is also applied in fleet and warehouse routing to improve operational efficiency and reduce expenses.
🏭 **Manufacturing & Industrial Automation**	Optimize the movement of tools and machines in processes such as CNC machining, PCB drilling, and 3D printing. It helps reduce production time by minimizing unnecessary tool travel paths in automated systems and robotic operations.
📡 **Telecommunications & Networking**	Optimize maintenance routes and data collection paths in sensor networks. It also helps in planning efficient inspection schedules for infrastructure such as cell towers and communication systems.
🖥️ **Eletronic design**	VLSI and chip design to optimize wiring and minimize the total length of connections between components. It helps reduce signal delay and improve overall performance in integrated circuit layouts.

Table 4.1: Real-world applications of the Traveling Salesman Problem.

4.2 The TSP model

Dantzig et al. [40] were the first to present a mathematical formulation for the TSP. The model is based on the idea of defining binary decision variables that indicate whether an arc between two cities is included in the tour or not. The objective function minimizes the total distance traveled, while the constraints ensure that each city is visited exactly once and that no subtours are formed.

Given a graph $G = (V, E)$, where $V = \{1, ..., n\}$ with $n = |V|$, and arc costs d_{ij}, the authors in [40] proposed the following model, which is now known as the Dantzig-Fulkerson-Johnson (DFJ) formulation for the TSP:

Definition 4.1 — DFJ model.

$$\min_{x} \quad \sum_{i=1}^{n}\sum_{i=1}^{n} d_{ij}x_{ij} \tag{4.1.1}$$

$$\text{s.t.} \quad x_{ii} = 0 \qquad i = 1, ..., n \tag{4.1.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, ..., n \tag{4.1.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., n \tag{4.1.4}$$

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \quad S \subset V, 2 \leq |S| \leq n-1 \tag{4.1.5}$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, ..., n \tag{4.1.6}$$

where:

- d_{ij} distance between cities i and j
- x_{ij} decision variables (arc between cities i and j is included in the tour or not)
- S subset of cities used in the subtour elimination constraints

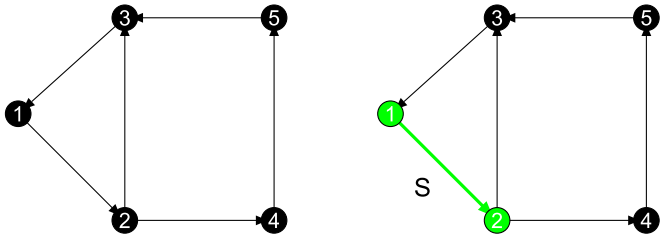
In the DFJ model, Equation (4.1.1) is the objective function that minimizes the total distance traveled. Equation (4.1.2) are constraints that prevent loops. Equation (4.1.3) and Equation (4.1.4) are constraints that ensure that each city is visited exactly once. Equation (4.1.5) are the subtour elimination constraints that prevent the formation of subtours, which are tours that do not include all cities. Finally, Equation (4.1.6) are binary constraints that ensure that the decision variables take values of either 0 or 1, indicating whether an arc is included in the tour or not.

Figure 4.2 illustrates the main idea behind subtour elimination constraint in Equation (4.1.5). The graph in Figure 4.2 (a) represents the cities and the arcs that can be used in the tour. A feasible solution must enter each city exactly once and leave each city exactly once (Equation (4.1.3) and Equation (4.1.4)). However, these degree constraints alone are not enough, because they may produce multiple disconnected cycles instead of a single tour.

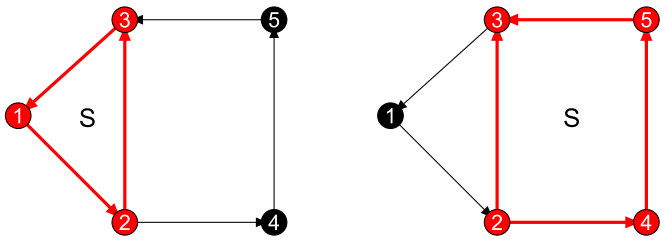
Equation (4.1.5) are constraints designed to prevent the formation of subtours. They ensure that for any subset of cities S, the number of arcs that connect cities

within S cannot be equal to $|S|$, which would indicate a subtour. Instead, at least one arc must connect S to the rest of the graph, ensuring that all cities are part of a single tour.

In Figure 4.2 (b) the subset $S = \{1, 2\}$ is valid because the number of arcs connecting cities within S is less than $|S| - 1$. However, Figure 4.2 (c) and Figure 4.2 (d) show different subsets S that the subtour elimination constraints will prevent to occur in the final solution.



(a) Graph instance. (b) $S = \{1, 2\}$.



(c) $S = \{1, 2, 3\}$. (d) $S = \{2, 3, 4, 5\}$.

Figure 4.2: TSP graph instance and different subset possibilities.

The TSP-DFJ model, however, has an exponential number of subtour elimination constraints (Equation (4.1.5)), since there are 2^n subsets of V. In practice, this model is often solved using a cutting-plane approach, where the model is initially solved without the subtour elimination constraints, and then violated subtour constraints are added iteratively until a valid tour is found.

There are alternative formulations for the TSP that use a polynomial number of constraints, such as the Miller-Tucker-Zemlin (MTZ) formulation [41]. The MTZ formulation introduces additional variables to help enforce the tour order and eliminate subtours without needing an exponential number of constraints. This model is introduced in Section 4.4.

Another alternative is to solve the TSP as an Assignment Problem, which also has a polynomial number of constraints, and then add subtour elimination con-

straints as lazy constraints during the solution process. This approach is discussed in Section 4.5.

In the next subsection we will show the importance of subtour elimination constraints by solving the TSP without them and showing that the obtained solution is not a valid tour, but rather a set of subtours.

4.3 TSP with no subtour elimination constraints

This section will show why the subtour elimination constraints are important to obtain a correct solution for the TSP. If we solve the TSP without subtour elimination constraints we are solving a relaxation (Equation (4.2)). The model becomes an Assignment Problem in which the assignment constraints over the decisions variables x_{ij} ensures that the salesman will enter and leave each city once.

Definition 4.2 — TSP-Rel model.

$$\min_{x} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} \tag{4.2.1}$$

$$\text{s.t.} \quad x_{ii} = 0 \qquad i = 1, ..., n \tag{4.2.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, ..., n \tag{4.2.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., n \tag{4.2.4}$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, ..., n \tag{4.2.5}$$

where:

- d_{ij} distance between cities i and j
- x_{ij} decision variables (arc between cities i and j is included in the tour or not)

The following example shows a TSP instance with 5 cities and the distance matrix D. We will solve the TSP without subtour elimination constraints (Equation (4.2)) and show that the obtained solution is not a valid tour, but rather a set of subtours.

Example 4.1 Consider a TSP instance with 5 cities (Figure 4.3) and the distance matrix:

$$D = \begin{pmatrix} 0.0 & 392.9 & 1250.8 & 529.5 & 1129.3 \\ 392.9 & 0.0 & 1030.1 & 331.6 & 1061.1 \\ 1250.8 & 1030.1 & 0.0 & 997.6 & 893.1 \\ 529.5 & 331.6 & 997.6 & 0.0 & 819.1 \\ 1129.3 & 1061.1 & 893.1 & 819.1 & 0.0 \end{pmatrix} \tag{4.3}$$

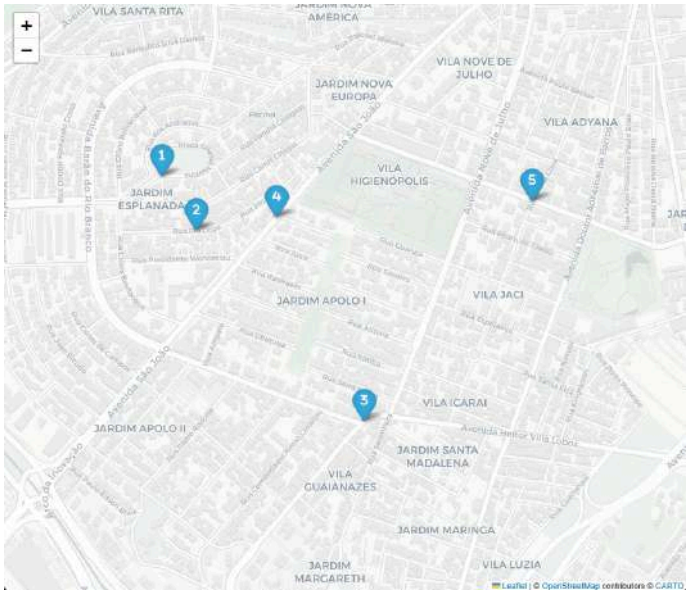


Figure 4.3: TSP instance.

Listing 4.1 presents the Julia code to solve the TSP-Rel model for the TSP instance with 5 cities. The code loads the distance matrix from an HDF5 file (Listing 4.1-5), defines the TSP-Rel model using JuMP (Listing 4.1-11), and solves it using the HiGHS solver. The obtained solution is then printed (Listing 4.1-36) and plotted on a map (Listing 4.1-57).

```
1  using JuMP  # Modeling language
2  using HiGHS # Solver
3  using HDF5  # For reading HDF5 files
4
5  # Load utility functions for plotting
6  include("utils/tsp_utils.jl")
7
8  # Load the distance matrix from the HDF5 file
9  D = h5read("data/tsp_example.h5", "distance_matrix")
10
11 # Number of locations
12 n = size(D, 1)
13
14 # Create the model
```

```
model = JuMP.Model(HiGHS.Optimizer)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, x[1:n, 1:n], Bin)

# Objective function: minimize total distance
@objective(model, Min, sum(D[i, j] * x[i, j] for i in 1:n, j in
1:n))

# Constraints: no self-loops
@constraint(model, [i=1:n], x[i, i] == 0)

# Constraints: each location visited once
@constraint(model, [j=1:n], sum(x[i, j] for i in 1:n) == 1)

# Constraints: each location departed once
@constraint(model, [i=1:n], sum(x[i, j] for j in 1:n) == 1)

# Solve
JuMP.optimize!(model)

# Get the values of the decision variables
x_opt = JuMP.value.(x)

# Extract the routes from the solution
visited = fill(false, n)
routes = Vector{Vector{Int}}()
for start in 1:n
    if !visited[start]
        route = Int[start]
        visited[start] = true
        nxt = findfirst(x_opt[start, :] .> 0.5)
        while nxt !== nothing && !visited[nxt]
            push!(route, nxt)
            visited[nxt] = true
            nxt = findfirst(x_opt[nxt, :] .> 0.5)
        end
        push!(routes, route)
        println("Route: ", route)
    end
```

```
end

# Get the optimal value of the objective function
z_opt = JuMP.objective_value(model)
println("Optimal value: $z_opt meters")

# Plot the solution on the map
fmap = plot_tsp_solution("data/tsp_example.h5", x_opt)
display(fmap)
```

Listing 4.1: Solving TSP-Rel with Julia.

Unfortunately, as can be seen in the code output (Listing 4.2) and in the map of the solution (Figure 4.4), solving as an Assignment problem may not result in the solution to the Traveling Salesman problem, as the requirement of forming a single connected tour may not be satisfied, whichleads to the emergence of subtours. In Figure 4.4 we have the occurrence of 2 subtours.

Output

```
Route: [1, 2, 4]
Route: [3, 5]
Optimal value: 3040.2 meters
```

Listing 4.2: Output of the TSP-Rel model.

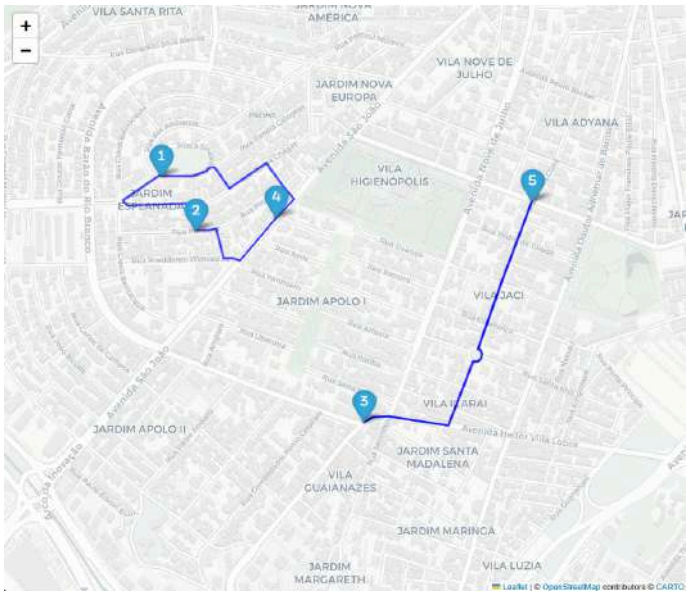


Figure 4.4: Solution for TSP example using TSP-Rel model.

The subtours must be broken so that we have a traveling salesman solution. One way to do this is to add new constraints known as Miller-Tucker-Zemlin (MTZ) [41] constraints to the Assignment Problem. This model will be introduced in the next section.

4.4 TSP with MTZ constraints

Miller, Tucker and Zemlin [41] proposed an alternative formulation for the TSP that uses a polynomial number of constraints. They introduced an additional set of variables u_i that represent the position of city i in the tour, and followed the intuition that if we select a road from city i to j ($x_{ij} = 1$), then the position of city j must be greater than the position of city i ($u_j \geq u_i + 1$).

To model the "If we use the road i to j, then j must be later than i." so the solver can understand they proposed the following constraints:

$$u_j \geq u_i + 1 + n(x_{ij} - 1) \quad i \neq j, \;\; i, j = 2, ..., n \tag{4.4}$$

An there are two cases to consider:

- If $x_{ij} = 1$, then the constraint becomes $u_j \geq u_i + 1$, which means that city j must be visited after city i in the tour.
- If $x_{ij} = 0$, then the constraint becomes $u_j \geq u_i + 1 - n$, which is always satisfied since u_i and u_j are between 1 and n.

Equation (4.5) presents the MTZ formulation for the TSP. This model introduces additional variables u_i to help enforce the tour order and eliminate subtours without needing an exponential number of constraints. The objective function and the degree constraints are similar to the TSP-Rel model, but the MTZ constraints (Equation (4.5.6)) ensure that no subtours can form by enforcing a specific ordering of the cities in the tour. These constraint work by assigning a unique value to each city (except the starting city) that represents its position in the tour. The constraints ensure that if there is an arc from city i to city j, then the position of city j must be greater than the position of city i, thus preventing the formation of subtours.

Equation (4.5.5) fix the tour to start from the first city. This is done to break the symmetry of the problem and reduce the search space. By fixing the first city as the starting point, we eliminate equivalent solutions that are just rotations of the same tour, which can significantly speed up the solver. Finally, Equation (4.5.8) represents the variables used by the subtour elimination constraints to enforce the order of the cities in the tour.

Definition 4.3 — TSP-MTZ model.

$$\min_{x,u} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} \tag{4.5.1}$$

$$\text{s.t.} \quad x_{ii} = 0 \qquad i = 1, ..., n \tag{4.5.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, ..., n \tag{4.5.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., n \tag{4.5.4}$$

$$u_1 = 1 \tag{4.5.5}$$

$$u_i - u_j + nx_{ij} \leq n - 1 \quad i \neq j, \; i, j = 2, ..., n \tag{4.5.6}$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, ..., n \tag{4.5.7}$$

$$2 \leq u_i \leq n \qquad i = 2, ..., n \tag{4.5.8}$$

where:

- d_{ij} distance between cities i and j
- x_{ij} decision variables (arc between cities i and j is included in the tour or not)
- u_i auxiliary variables to eliminate subtours

Listing 4.3 presents the Julia code to solve the TSP-MTZ model. The code loads the distance matrix from an HDF5 file (Listing 4.3-8), defines the TSP-MTZ model using JuMP (Listing 4.3-14), and solves it using the HiGHS solver. The obtained solution is then printed (Listing 4.3-50) and plotted on a map (Listing 4.3-62).

```julia
using JuMP  # Modeling language
using HiGHS # Solver
using HDF5  # For reading HDF5 files

# Load utility functions for plotting
include("utils/tsp_utils.jl")

# Load the distance matrix from the HDF5 file
D = h5read("data/tsp_example.h5", "distance_matrix")

# Number of locations
n = size(D, 1)

# Create model
```

```
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, x[1:n, 1:n], Bin)

# Define MTZ variables
@variable(model, 1 <= u[1:n] <= n)

# Objective function: minimize total distance
@objective(model, Min, sum(D[i,j] * x[i,j] for i in 1:n, j in 1:n))

# Avoid self-loops
@constraint(model, [i=1:n], x[i,i] == 0)

# Each column sums to 1 (each location visited once)
@constraint(model, [j=1:n], sum(x[i,j] for i in 1:n) == 1)

# Each row sums to 1 (each location departs once)
@constraint(model, [i=1:n], sum(x[i,j] for j in 1:n) == 1)

# Breaking symmetry by fixing the first city as the starting point
@constraint(model, u[1] == 1)

# MTZ constraints
@constraint(model, [i=2:n, j=2:n; i != j], u[j] >= u[i] + 1 + n *
(x[i,j] - 1))

# Run the solver
JuMP.optimize!(model)

# Get the values of the decision variables
x_opt = JuMP.value.(x)

# Extract the routes from the solution
route = [1]
while true
    to = findfirst(x_opt[route[end], :] .> 0.5)
    to == 1 ? break : push!(route, to)
```

```
end
println("Route: $route")

# Get the optimal value of the objective function
z_opt = JuMP.objective_value(model)
println("Optimal value: $z_opt meters")

# Plot the solution on the map
fmap = plot_tsp_solution("data/tsp_example.h5", x_opt)
display(fmap)
```

Listing 4.3: Solving TSP-MTZ with Julia.

Output

```
Route: [1, 2, 3, 5, 4]
Optimal value: 3664.7 meters
```

Listing 4.4: Output of the TSP-MTZ model.

Figure 4.5 shows the solution obtained by solving the TSP-MTZ model. As can be seen, the obtained solution is a valid tour that visits all cities exactly once and returns to the starting city, which is different from the solution obtained by solving the TSP-Rel model, which had subtours.

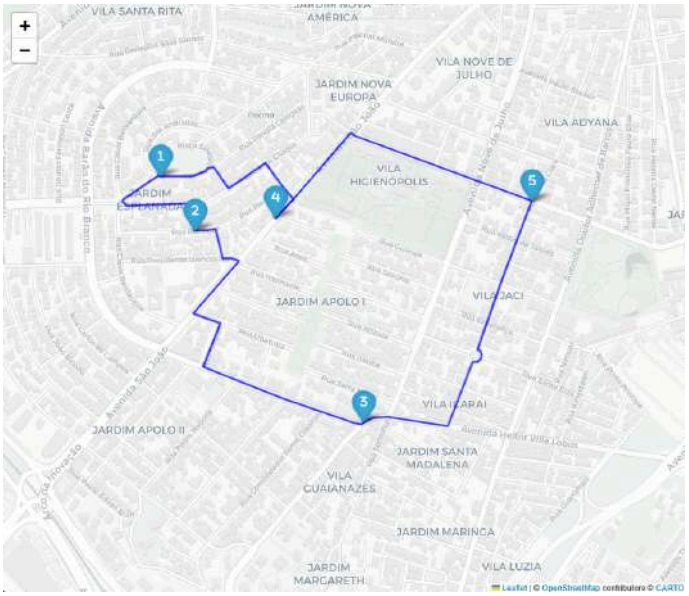


Figure 4.5: Solution for TSP example using TSP-MTZ model.

4.5 TSP with lazy constraints

Another way to eliminate the subtours is create an initial solution with the Assignment Problem (Equation (4.2)) and then check if the solution contains subtours. If it does, we can include the constraints that break the subtours as lazy constraints and solve the model again. This process (Figure 4.6) is repeated until all subtours are broken and a valid TSP solution is obtained.

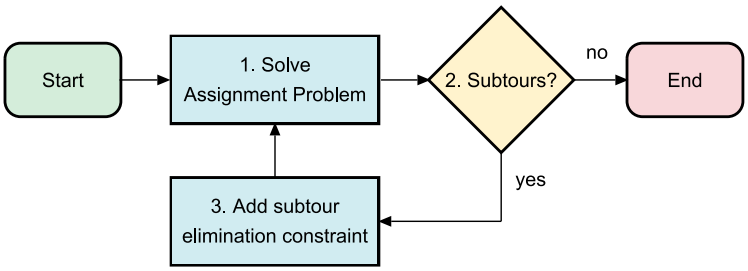


Figure 4.6: Lazy constraints flowchart.

Listing 4.5 presents the Julia code to solve the TSP using lazy constraints. The code defines a callback function (Listing 4.5-14) that checks for subtours in the current solution and adds lazy constraints to eliminate them. We create a graph representation of the solution and find the connected components to identify subtours. If a subtour is found, a lazy constraint is added to the model to eliminate it. The TSP model is then solved iteratively until a valid TSP solution is obtained.

The TSP model is defined similarly to the TSP-Rel model (Equation (4.2)), but without the subtour elimination constraints. The lazy constraint callback is set to the model (Listing 4.5-85), and the model is solved iteratively until a valid TSP solution is obtained.

```
1  using JuMP   # Modeling language
2  using GLPK   # Solver
3  using HDF5   # For reading HDF5 files
4  using Graphs # For graph operations
5
6  # Load utility functions for plotting
7  include("utils/tsp_utils.jl")
8
9  # Define lazy constraints callback function
10 function tsp_lazy_callback(cb_data)
11
12     # Get decision variable values
```

```
    x_val = [callback_value(cb_data, x[i,j]) for i in 1:n, j in 1:n]

    # Build an undirected graph
    g = Graphs.SimpleGraph(n)
    for i in 1:n, j in 1:n
        if i != j && x_val[i, j] > 0.5
            Graphs.add_edge!(g, i, j)
        end
    end

    # Find connected components
    cc = Graphs.connected_components(g)

    # If there are more than one connected components, we have a subtour
    if length(cc) > 1

        # Pick the smallest component (i.e., subtour)
        minTour = sort(cc, by=length)[1]

        # Initialize the left-hand side of the constraint
        subtourLhs = AffExpr()

        # Sum the decision variables corresponding to the subtour edges
        for i in minTour, j in minTour
            if i != j && x_val[i, j] > 0.5
                subtourLhs += x[i, j]
            end
        end

        # Add lazy constraint to eliminate subtour
        constraint = @build_constraint(subtourLhs <= length(minTour) - 1)

        # Print the constraint for debugging
        println("Adding lazy constraint: ", constraint.func, " <= ", constraint.set.upper)

        # Submit the lazy constraint to the model
```

```
49          JuMP.MOI.submit(model, JuMP.MOI.LazyConstraint(cb_data),
    constraint)
50      end
51  end
52
53  # Load the distance matrix from the HDF5 file
54  D = h5read("data/tsp_example.h5", "distance_matrix")
55
56  # Number of locations
57  n = size(D, 1)
58
59  # Create model
60  model = JuMP.Model(GLPK.Optimizer)
61
62  # Silent mode (solver output is not printed)
63  JuMP.set_silent(model)
64
65  # Define the decision variables
66  @variable(model, x[1:n, 1:n], Bin)
67
68  # Objective function: minimize total distance
69  @objective(model, Min, sum(D[i,j] * x[i,j] for i in 1:n, j in 1:n))
70
71  # Constraints to eliminate self-loops
72  @constraint(model, [i=1:n], x[i,i] == 0)
73
74  # Each column sums to 1 (each location visited once)
75  @constraint(model, [j=1:n], sum(x[i,j] for i in 1:n) == 1)
76
77  # Each row sums to 1 (each location departs once)
78  @constraint(model, [i=1:n], sum(x[i,j] for j in 1:n) == 1)
79
80  # Set the lazy constraint callback function to the model
81  JuMP.MOI.set(model, JuMP.MOI.LazyConstraintCallback(),
    tsp_lazy_callback)
82
83  # Run the solver
84  JuMP.optimize!(model)
85
86  # Get the values of the decision variables
87  x_opt = JuMP.value.(x)
```

```
88
89  # Extract the routes from the solution
90  route = [1]
91  while true
92      to = findfirst(x_opt[route[end], :] .> 0.5)
93      to == 1 ? break : push!(route, to)
94  end
95  println("Route: $route")
96
97  # Get the optimal value of the objective function
98  z_opt = JuMP.objective_value(model)
99  println("Optimal value: $z_opt meters")
100
101 # Plot the solution on the map
102 fmap = plot_tsp_solution("data/tsp_example.h5", x_opt)
103 display(fmap)
```

Listing 4.5: Solving TSP with lazy constraints in Julia.

Listing 4.6 shows the output of solving the TSP with lazy constraints. The solver identifies the subtours in the solution of the Assignment Problem and adds the corresponding lazy constraints to eliminate them. After a few iterations, a valid TSP solution is obtained, which is the same as the one obtained by solving the TSP-MTZ model.

Output

```
Adding lazy constraint: x[3,5] + x[5,3] <= 1.0
Adding lazy constraint: x[1,2] + x[2,1] <= 1.0
Route: [1, 2, 3, 5, 4]
Optimal value: 3664.7 meters
```

Listing 4.6: Output of the TSP with lazy constraints.

Figure 4.7 shows the solution obtained by solving the TSP with lazy constraints. As can be seen, the obtained solution is a valid tour that visits all cities exactly once and returns to the starting city, which is the same as the solution obtained by solving the TSP-MTZ model, demonstrating that both approaches are effective in eliminating subtours and finding a valid TSP solution.

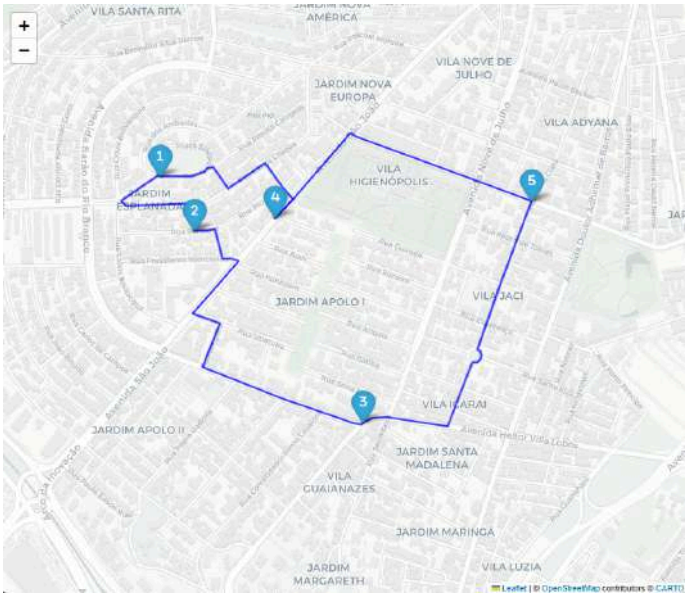


Figure 4.7: Solution for TSP example using lazy constraints.

4.6 Final remarks

The traveling salesman problem (TSP) is a fundamental problem in the field of vehicle routing and distribution. It involves finding the shortest possible route for a salesman to visit a set of cities and return to the starting city, while visiting each city exactly once. The TSP has been extensively studied, and numerous heuristic and exact methods have been developed to solve it.

In a sequel to this chapter, it would be valuable to explore the various versions of the Vehicle Routing Problem (VRP), as mentioned in [42]. VRP extend the TSP by incorporating additional constraints and objectives related to vehicle capacities, time windows, multiple depots, and other real-world considerations. These problems are highly relevant in logistics and transportation planning, as they involve optimizing the routes and schedules of vehicles to efficiently serve a set of customers or locations.

By examining different versions of VRPs, you can gain a deeper understanding of the complexities and variations that arise in real-world routing scenarios. You can also explore the various solution methods and algorithms specifically designed for solving VRPs. This would provide valuable insights into how to tackle more complex vehicle routing and distribution problems and contribute to the development of more efficient routing strategies.

4.7 Exercises

Formulate and solve the following exercises using the JuMP package.

Exercise 4.1 Consider the TSP instance from Figure 4.8, which is based on the dataset tsp_exercise.h5.

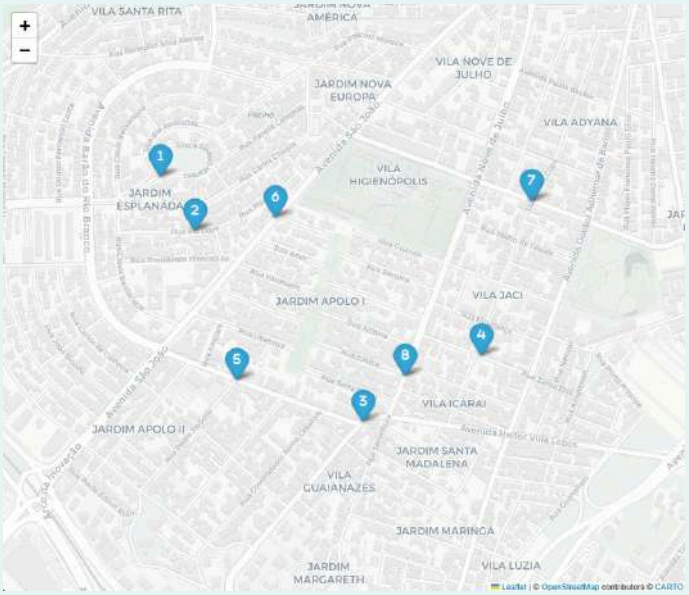


Figure 4.8: TSP instance `tsp_exercise.h5`.

a) Formulate and solve the TSP without subtour elimination constraints (Section 4.3).

b) Formulate and solve the TSP with Miller-Tucker-Zemlin constraints (Section 4.4).

c) Formulate and solve the TSP with lazy constraints (Section 4.5).

Compare the obtained solutions in terms of the optimal value of the objective function and the structure of the routes. Discuss the importance of subtour elimination constraints in obtaining a valid TSP solution.

Exercise 4.2 Use dataset tsp_exercise.csv (Figure 4.9) to solve the TSP with Miller-Tucker-Zemlin constraints.

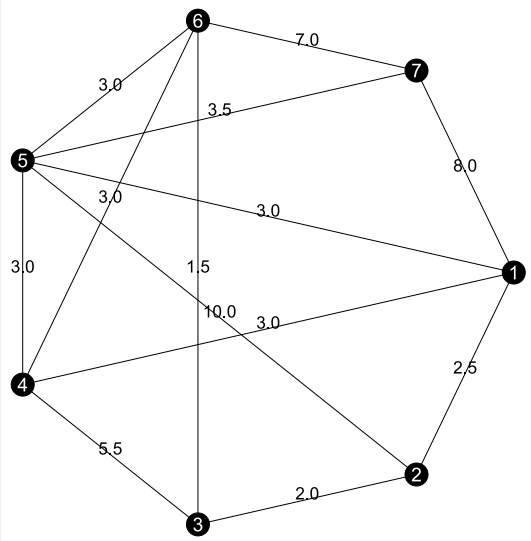
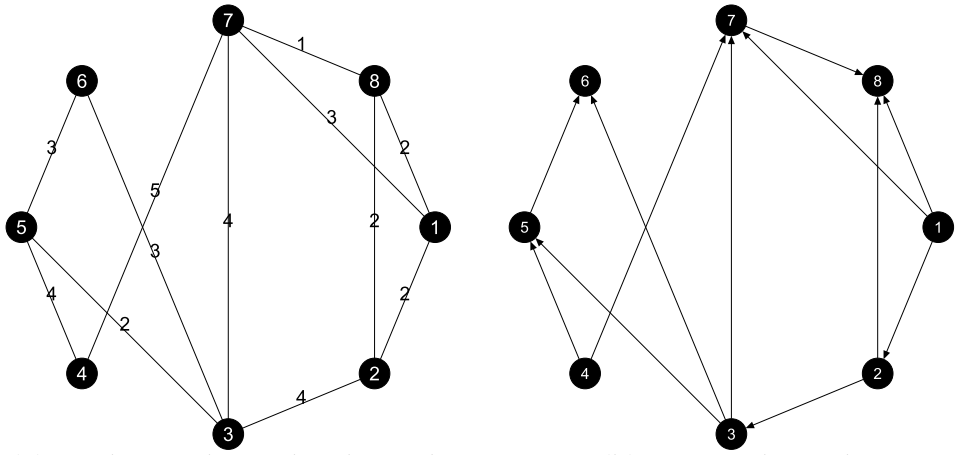


Figure 4.9: Example weighted undirected graph.

5 Graph Problems

Some IP problems can be formulated using a graph representation [43], [44]. A graph $G = (V, E)$ consists of two sets: V, the set of vertices, and E, the set of edges connecting pairs of vertices. Edges may be either directed or undirected, and they may also have associated weights.

In an undirected graph Figure 5.1 (a), if there is an edge $\{u, v\} \in E$, then vertices u and v are said to be adjacent, and the edge is incident to both vertices. In a directed graph Figure 5.1 (b), edges have a direction. If there is a directed edge $(u, v) \in E$, it means the edge goes from vertex u to vertex v. In this case, u is the tail and v is the head of the edge. Vertex v is called an out-neighbor of u, and u is an in-neighbor of v. The edge is incident from u to v.



(a) Undirected weighted graph. (b) Directed graph.

Figure 5.1: Graph types.

In this chapter, we will explore five fundamental problems defined on graphs that can be formulated as IP models (Figure 5.2). These problems include:

1. **Maximum Independent Set Problem (MISP)**: The MISP aims to find the largest possible subset of vertices in a graph such that no two vertices in the subset are adjacent. In other words, it seeks to find a maximum set of non-adjacent vertices.
2. **Vertex Cover Problem (VCP)**: The VCP involves finding the smallest possible subset of vertices in a graph such that every edge in the graph is incident to at

least one vertex in the subset. In other words, it aims to find a minimum set of vertices that "covers" all the edges.

3. **Graph Coloring Problem (GCP)**: The GCP focuses on assigning colors to the vertices of a graph such that no adjacent vertices have the same color. The objective is to minimize the total number of colors required to color the graph.
4. **Graph Partitioning Problem (GPP)**: The GPP seeks to divide the vertices of a graph into a specified number of disjoint subsets (partitions) such that the number of edges connecting vertices in different partitions is minimized.
5. **Clique Partitioning Problem (CPP)**: The CPP is a graph partitioning problem where the goal is to partition the vertices of a graph into a minimum number of cliques. A clique is a subset of vertices in a graph where every pair of vertices is connected by an edge.

All of these problems have important applications in various domains, including network design, scheduling, resource allocation, and optimization. By formulating these problems as IP models, we can leverage powerful optimization techniques and solvers to find optimal or near-optimal solutions.

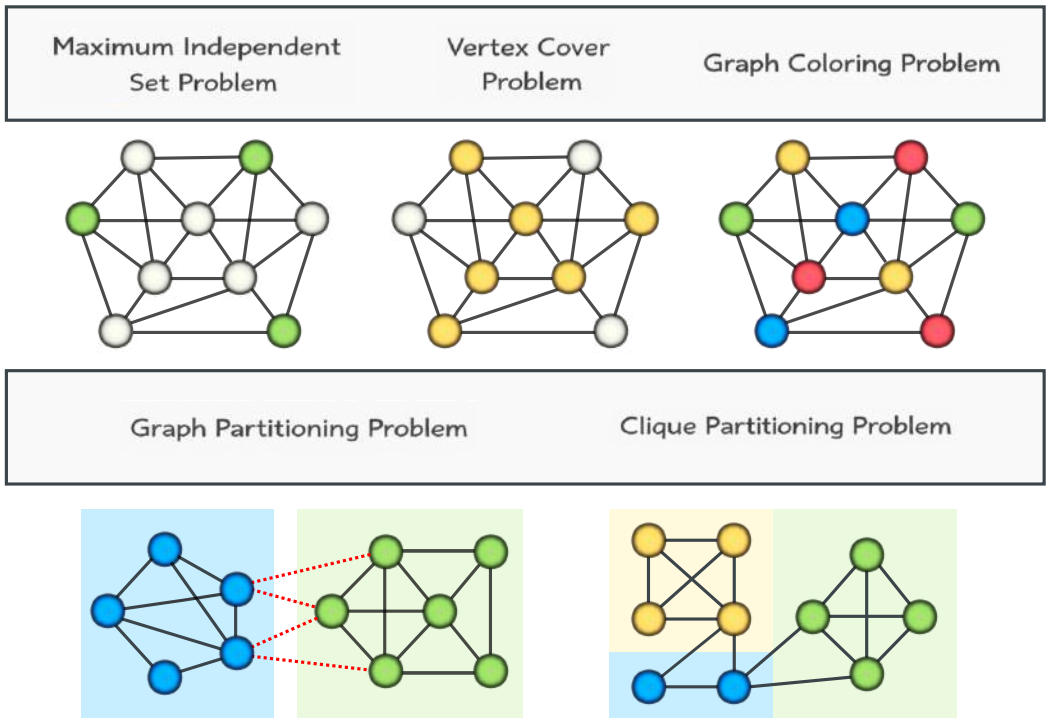


Figure 5.2: Fundamental Graph problems.

5.1 Maximum Independent Set Problem

Given an undirected graph $G = (V, E)$, an **independent vertex set** is a subset $S \subseteq V$ such that no two vertices in S are adjacent. The **Maximum Independent Set Problem** (MISP) asks for an independent set of maximum cardinality. Formally, the problem is:

$$\alpha(G) = \max\{|S| \mid S \subseteq V,\ \forall u, v \in S,\ \{u, v\} \notin E\} \tag{5.1}$$

where $\alpha(G)$ is the **independence number** of a graph, which is defined as the cardinality of the largest independent vertex set.

The MISP is NP-hard [45], implying that no polynomial-time algorithm is known for solving it on general graphs. Moreover, the problem is also difficult to approximate, since computing near-optimal solutions is computationally intractable in the general case [46].

5.1.1 MISP applications

Table 5.1 presents some real-world applications of the Maximum Independent Set Problem.

Application	Description
Telecommunications	Model non-interfering transmitters in a network where nearby nodes cause signal interference, maximizing simultaneous communication without collisions [47]
Social Network	It is used for tasks like selecting representative samples, reducing redundancy, and analyzing independent structures within social graphs [48]
Resource allocation	Selecting the largest possible combination of non-overlapping tasks or exams to eliminate timeline and structural conflicts [49]
Bioinformatics	It helps identify groups of genes or proteins that do not directly interact within biological networks [50]

Table 5.1: Real-world applications of the Maximum Independent Set Problem.

5.1.2 MISP model

Equation (5.2) presents an IP model for the Maximum Independent Set Problem. In this model, we define binary decision variables x_v for each vertex $v \in V$, where

$x_v = 1$ if vertex v is included in the independent set S, and $x_v = 0$ otherwise. The objective is to maximize the sum of the decision variables, which corresponds to maximizing the size of the independent set. The constraints (Equation (5.2.2)) ensure that no two adjacent vertices are included in the independent set, and that each decision variable is binary.

Definition 5.1 — MISP model.

$$\max_{x} \quad \sum_{v \in V} x_v \tag{5.2.1}$$

$$\text{s.t.} \quad x_u + x_v \leq 1 \quad \forall \{u, v\} \in E \tag{5.2.2}$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \tag{5.2.3}$$

where:

x_v decision variables (if the vertex v is included in the independent set S)

5.1.3 Solving MISP with Julia

The following example shows the MISP instance that will be used in this subsection.

Example 5.1 Consider the undirected graph G shown in Figure 5.3. We want to find the maximum independent set of this graph.

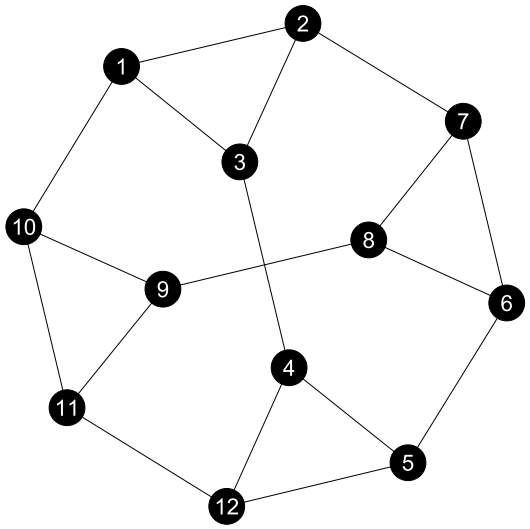


Figure 5.3: MISP instance.

Listing 5.1 presents a Julia implementation for solving the MISP instance shown in Figure 5.3. The code uses the JuMP package for modeling the optimization problem, the HiGHS solver for solving the model, and the Graphs package

for handling graph data structures. It begins by creating the graph instance (Listing 5.1-8), defining the MISP model (Listing 5.1-15) and solving it. Finally, it uses the function `plot_solution` (Listing 5.1-39), defined in `utils/misp_utils.jl`, to visualize the solution by highlighting the vertices that comprise the maximum independent set.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/misp_utils.jl")

# Create graph
graph = Graphs.smallgraph("truncatedtetrahedron")

# Auxiliary variables to simplify notation
V = 1:Graphs.nv(graph)  # Range of vertices
E = Graphs.edges(graph) # Graph edges

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, x[v in V], Bin)

# Objective function: maximize the total of selected vertices
@objective(model, Max, sum(x[v] for v in V))

# Independence constraint
@constraint(model, [e in E], x[e.src] + x[e.dst] <= 1)

# Run the solver
JuMP.optimize!(model)

# Independent set S
S = [v for v in V if JuMP.value(x[v]) > 0.5]

```

```
# Print solution
println("Maximum independent set (S): $S")

# Plot solution
plot_solution(graph, S)
```

Listing 5.1: Solving MISP with Julia.

Listing 5.2 presents the output of the MISP model for the given graph instance. The output indicates that the maximum independent set (S). This means that these vertices can be included in the independent set without any two of them being adjacent to each other.

Output

```
Maximum independent set (S): [1, 4, 6, 11]
```

Listing 5.2: Output of the MISP model.

Figure 5.4 presents the visual representation of the solution for the MISP instance. In the graph, the vertices that are part of the maximum independent set (S) are highlighted in red, while the other vertices are shown in black.

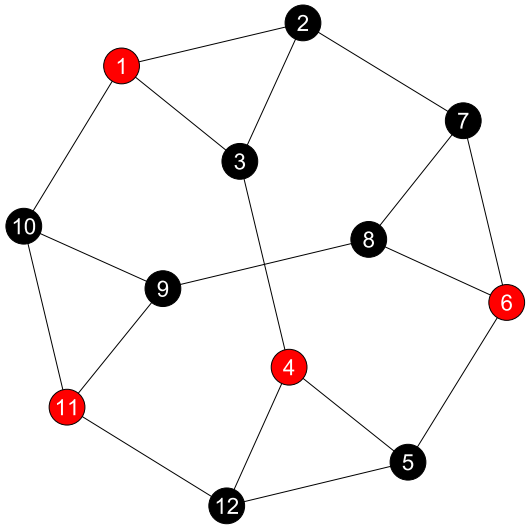


Figure 5.4: Solution for MISP example.

5.2 Vertex Covering Problem

Given an undirected graph $G = (V, E)$, a **vertex cover** is a subset of vertices $C \subseteq V$ such that every edge in the graph has at least one endpoint in C. In the **Vertex**

Covering Problem (VCP), we want to find a vertex cover such as the cardinality $|C|$ is minimal. Formally, the problem is:

$$\tau(G) = \min\{|C| \mid C \subseteq V,\ \forall\{u,v\} \in E,\ u \in C \vee v \in C\} \tag{5.3}$$

where $\tau(G)$ is the **vertex cover number** of a graph, which is defined as the cardinality of the smallest vertex cover.

The VCP is classified as NP-hard, meaning that while it is computationally easy to verify a given solution, finding an optimal solution is computationally difficult. Specifically, no polynomial-time algorithm is known to solve all instances of the problem efficiently.

5.2.1 Gallai's Theorem

There is a fundamental relationship between the Vertex Covering Problem and the Maximum Independent Set Problem, formalized by **Gallai's Theorem** [51]. This theorem establishes that for any undirected graph $G = (V, E)$, the size of a minimum vertex cover and the size of a maximum independent set are complementary:

$$\tau(G) + \alpha(G) = |V| \tag{5.4}$$

where $\tau(G)$ is the vertex cover number (size of the minimum vertex cover), and $\alpha(G)$ is the independence number (size of the maximum independent set).

This relationship means that if we find a maximum independent set S in graph G, then the complement set $V \setminus S$ forms a vertex cover, and vice versa. Consequently, solving one problem directly provides a solution to the other. If the maximum independent set has size k, then the minimum vertex cover has size $|V| - k$. This duality is computationally useful, as it allows algorithms designed for one problem to be adapted for the other by considering the complement of the solution.

5.2.2 VCP applications

Table 5.2 presents some real-world applications of the Vertex Covering Problem.

Application	Description
Telecommunications	The placement of application images on edge computing servers, where each vertex represents an edge node and the goal is to minimize the number of stored images while ensuring availability for all necessary communications [52]

Application	Description
🌐 **Social Network**	A vertex cover represents a set of influential individuals who can monitor or influence the entire network. It is used for tasks like information dissemination, viral marketing, and identifying key influencers by selecting a minimum set of vertices that can cover all connections in the social graph [53]
🗓️ **Resource allocation**	Select a minimum set of resources or time slots that can cover all tasks or events. It helps optimize job scheduling, exam timetabling, and resource usage by ensuring that every task or event is covered by at least one resource or time slot [54]
🧬 **Bioinformatics**	It helps identify a minimum set of genes or proteins that can interact with all other genes or proteins in a biological network. It is used to detect key regulatory elements and functional modules in protein–protein interaction and gene regulatory networks [55]

Table 5.2: Real-world applications of the Vertex Covering Problem.

5.2.3 VCP model

Equation (5.5) presents an IP model for the Vertex Covering Problem. In this model, we define binary decision variables x_v for each vertex $v \in V$, where $x_v = 1$ if vertex v is included in the vertex cover C, and $x_v = 0$ otherwise. The objective is to minimize the sum of the decision variables, which corresponds to minimizing the size of the vertex cover. The constraints (Equation (5.5.2)) ensure that for every edge in the graph, at least one of its endpoints is included in the vertex cover, and that each decision variable is binary.

Definition 5.2 — VCP model.

$$\min_{x} \quad \sum_{v \in V} x_v \tag{5.5.1}$$

$$\text{s.t.} \quad x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \tag{5.5.2}$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \tag{5.5.3}$$

where:

x_v decision variables (if the vertex v is included in the vertex cover C)

5.2.4 Solving VCP with Julia

The following example shows the VCP instance that will be used in this subsection.

Example 5.2 Consider the undirected graph G shown in Figure 5.5. We want to find the minimum vertex cover of this graph.

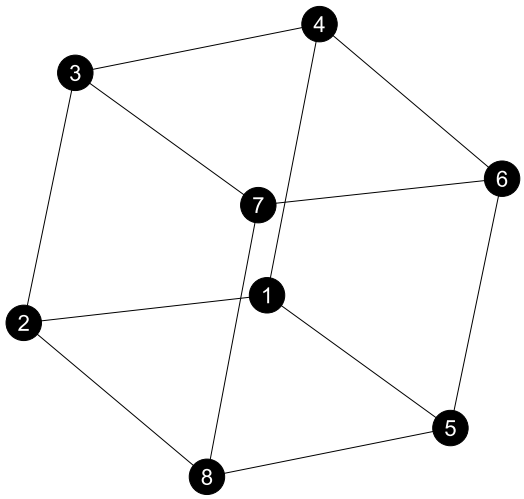


Figure 5.5: VCP instance.

Listing 5.3 presents a Julia implementation for solving the MISP instance shown in Figure 5.5. The code uses the JuMP package for modeling the optimization problem, the HiGHS solver for solving the model, and the Graphs package for handling graph data structures. It begins by creating the graph instance (Listing 5.3-8), defining the VCP model (Listing 5.3-15) and solving it. Finally, it uses the function `plot_solution` (Listing 5.3-39), defined in `utils/vcp_utils.jl`, to visualize the solution by highlighting the vertices that comprise the minimum vertex cover.

```
1  using JuMP   # Modeling language
2  using HiGHS  # Solver
3  using Graphs # Graphs package
4
5  # Load utility function for plotting
6  include("utils/vcp_utils.jl")
7
8  # Create graph
9  graph = Graphs.smallgraph("cubical")
10
11 # Auxiliary variables to simplify notation
12 V = 1:Graphs.nv(graph)  # Range of vertices
```

```
E = Graphs.edges(graph) # Graph edges

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, x[v in V], Bin)

# Objective function: minimize the total of selected vertices
@objective(model, Min, sum(x[v] for v in V))

# Vertex Cover constraint
@constraint(model, [e in E], x[e.src] + x[e.dst] >= 1)

# Run the solver
JuMP.optimize!(model)

# Vertex Cover C
C = [v for v in V if JuMP.value(x[v]) > 0.5]

# Print solution
println("Vertex Cover set (C): $C")

# Plot solution
plot_solution(graph, C)
```

Listing 5.3: Solving VCP with Julia.

Listing 5.4 presents the output of the VCP model for the given graph instance. The output indicates the vertex cover set (C). This means that these vertices are included in the vertex cover, ensuring that every edge in the graph is incident to at least one vertex in this set.

Output

```
Vertex Cover set (C): [2, 4, 5, 7]
```

Listing 5.4: Output of the VCP model.

Figure 5.6 presents the visual representation of the solution for the VCP instance. In the graph, the vertices that are part of the minimum vertex cover (C) are highlighted in red, while the other vertices are shown in black.

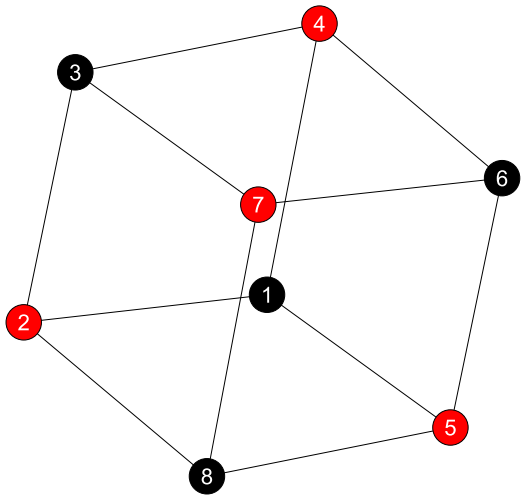


Figure 5.6: Solution for VCP example.

5.3 Graph Coloring Problem

The **Graph Coloring Problem** (GCP) involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color, using the minimum number of colors possible.

Given an undirected graph $G = (V, E)$, the goal is to assign a color to each $v \in V$ such that: if $(u, v) \in E$, then color o vertex u is different from vertex v color. The minimum number of colors needed to achieve this is called the chromatic number $\chi(G)$. Formally, the problem can be stated as:

$$\chi(G) = \min\{k \in \mathbb{N} \mid \exists c : V \to \{1, ..., k\},\ \forall \{u, v\} \in E,\ c(u) \neq c(v)\} \quad (5.6)$$

Finding the chromatic number is NP-hard. Even deciding whether a graph can be colored with just 3 colors is NP-complete.

5.3.1 GCP applications

Table 5.3 presents some real-world applications of the Graph Coloring Problem.

Application	Description
Register allocation	Assigning variables to a limited number of CPU registers. It maps an infinite number of temporary program variables to a limited set of CPU

Application	Description
	registers, ensuring variables with overlapping lifetimes do not share the same register [56]
Frequency assignment	Assigning frequencies to transmitters to avoid interference. In a WLAN environment, an access point is represented as a vertex in a graph, while an edge connects any two vertices whose access points interfere with each other if assigned the same or overlapping frequency [57]
Timetabling	Assigning classes to time slots and rooms without conflicts. The problem is modeled by representing each course as a vertex in a graph, while an edge connects any two vertices if the corresponding courses share common students or teachers (creating a scheduling conflict) [58]
Cloud computing	Assigning virtual machines to physical servers while minimizing resource contention and ensuring efficient utilization. The problem is modeled by representing each cloud task or virtual machine as a vertex in a graph, while an edge connects any two vertices that compete for the same limited resource (creating a conflict) [59]

Table 5.3: Real-world applications of the Graph Coloring Problem.

5.3.2 GCP model

Equation (5.7) presents an IP model for the Graph Coloring Problem. In this model, we define binary decision variables x_{vc} for each vertex $v \in V$ and color $c \in C$, where $x_{vc} = 1$ if vertex v is assigned color c, and $x_{vc} = 0$ otherwise. We also define binary decision variables y_c for each color $c \in C$, where $y_c = 1$ if color c is used in the coloring, and $y_c = 0$ otherwise. The objective is to minimize the sum of the y_c variables, which corresponds to minimizing the number of colors used. The constraints ensure that each vertex is assigned exactly one color (Equation (5.7.2)) and that adjacent vertices do not share the same color (Equation (5.7.3)). Finally, Equation (5.7.4) ensure that all decision variables are binary.

Definition 5.3 — GCP model.

$$\min_{x,y} \quad \sum_{c \in C} y_c \tag{5.7.1}$$

$$\text{s.t.} \quad \sum_{c \in C} x_{vc} = 1 \qquad \forall v \in V \tag{5.7.2}$$

$$x_{uc} + x_{vc} \leq y_c \qquad \forall \{u, v\} \in E \quad \forall c \in C \tag{5.7.3}$$

$$x_{vc} \in \{0, 1\} \qquad \forall v \in V \quad \forall c \in C \tag{5.7.4}$$

$$y_c \in \{0, 1\} \qquad \forall c \in C \tag{5.7.5}$$

where:

y_c decision variables (if color c is used in the coloring)

x_{vc} decision variables (if vertex v is assigned color c)

5.3.3 Solving GCP with Julia

The following example shows the GCP instance that will be used in this subsection.

Example 5.3 Consider the undirected graph G shown in Figure 5.7. We want to find the minimum number of colors needed to color the graph such that no two adjacent vertices share the same color.

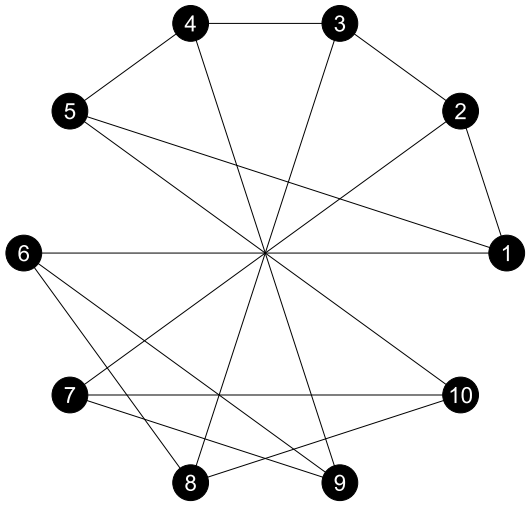


Figure 5.7: GCP instance.

Listing 5.5 presents a Julia implementation for solving the MISP instance shown in Figure 5.7. The code uses the JuMP package for modeling the optimization problem, the HiGHS solver for solving the model, and the Graphs package for handling graph data structures. It begins by creating the graph instance (Listing 5.5-8), defining the GCP model (Listing 5.5-16) and solving it. Finally, it

uses the function `plot_solution` (Listing 5.5-53), defined in `utils/gcp_utils.jl`, to visualize the solution by highlighting the vertices with their assigned colors.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/gcp_utils.jl")

# Create graph
graph = Graphs.smallgraph("petersen")

# Auxiliary variables to simplify notation
V = 1:Graphs.nv(graph)  # Range of vertices
C = 1:Graphs.nv(graph)  # Range of colors
E = Graphs.edges(graph) # Graph edges

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, y[c in C], Bin)
@variable(model, x[v in V, c in C], Bin)

# Minimize the total of selected colors
@objective(model, Min, sum(y[c] for c in C))

# Each vertex must be assigned exactly one color
@constraint(model, [v in V], sum(x[v,c] for c in C) == 1)

# Adjacent vertices cannot share the same color
@constraint(model, [e in E, c in C], x[e.src, c] + x[e.dst, c] <=
y[c])

# Run the solver
JuMP.optimize!(model)

# Chromatic number
```

```julia
X = JuMP.objective_value(model)
println("Chromatic number: $X")

# Extract partition from solution
partition = Vector{Vector{Int}}()
for c in C
    if JuMP.value(y[c]) > 0.5
        cluster = [v for v in V if JuMP.value(x[v,c]) > 0.5]
        push!(partition, cluster)
    end
end

println("Partition: ", partition)

# Plot solution
plot_solution(graph, partition)
```

Listing 5.5: Solving GCP with Julia.

Listing 5.6 presents the output of the GCP model for the given graph instance. The output indicates the chromatic number ($\chi(G)$), which is the minimum number of colors needed to color the graph such that no two adjacent vertices share the same color. Additionally, it shows the groups of vertices that are assigned the same color in the solution.

Output

```
Chromatic number: 3.0
Partition: [[2, 8, 9], [3, 5, 6, 7], [1, 4, 10]]
```

Listing 5.6: Output of the GCP model.

Figure 5.8 presents the visual representation of the solution for the GCP instance. In the graph, the vertices are colored according to their assigned colors, with different colors representing different groups of vertices that do not share the same color.

The GCP can be considered a clustering problem, as the independent sets form clusters. In the next sections we will look at two other clustering problems.

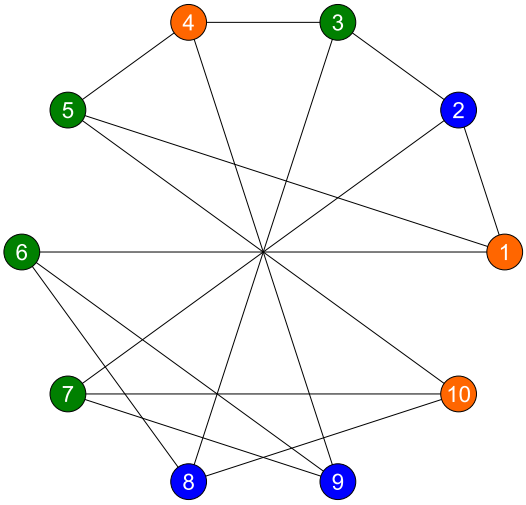


Figure 5.8: Solution for GCP example.

5.4 Graph Partitioning Problem

Given an undirected graph $G = (V, E)$ and a positive integer k, the **Graph Partitioning Problem** (GPP) asks for a partition of the vertex set V into k disjoint subsets (partitions) $V_1, V_2, ..., V_k$ such that the number of edges connecting vertices in different partitions is minimized. Formally, the problem can be stated as:

$$\text{k-cut}(G) = \min\left\{ \sum_{1 \le i < j \le k} |E(V_i, V_j)| \;\middle|\; \sqcup_{i=1}^{k} V_i = V \right\} \tag{5.8}$$

where k-cut(G) is the minimum number of edges (cut) that need to be removed to partition the graph into k disjoint subsets, $E(V_i, V_j)$ is the set of edges with one endpoint in V_i and the other endpoint in V_j.

The GPP is is NP-hard for both balanced and unbalanced partitions, and for weighted and unweighted graphs as well.

5.4.1 GPP applications

Table 5.4 presents some real-world applications of the Graph Partitioning Problem.

Application	Description
Telecommunications	It divides networks into smaller subnets to reduce physical link edge-cuts, minimizing cross-subnet communication overhead and transmission latency [60]

Application	Description
🌐 **Social Network**	Partitioning a social network into communities or clusters of users who are more closely connected to each other than to the rest of the network. It is used for community detection, influence analysis, and targeted marketing by identifying groups of users with similar interests or behaviors [61]
🗓️ **Scheduling**	Partitioning tasks or resources into groups to optimize scheduling and resource allocation. It helps in job scheduling, exam timetabling, and resource usage by minimizing the number of conflicts between different groups of tasks or resources [62]
🧬 **Bioinformatics**	It partitions large macromolecular or gene regulatory networks into isolated functional modules to uncover hidden cellular pathways and disease mechanisms [63]

Table 5.4: Real-world applications of the Graph Partitioning Problem.

5.4.2 GPP model

Equation (5.9) presents an IP model for the Graph Partitioning Problem. In this model, we define binary decision variables $x_{\{vp\}}$ for each vertex $v \in V$ and partition $p \in \{1, ..., k\}$, where $x_{\{vp\}} = 1$ if vertex v is assigned to partition p, and $x_{\{vp\}} = 0$ otherwise. We also define binary decision variables y_e for each edge $e \in E$, where $y_e = 1$ if edge e is cut (i.e., its endpoints are in different partitions), and $y_e = 0$ otherwise.

The objective is to minimize the sum of the y_e variables, which corresponds to minimizing the number of edges that are cut. The constraints ensure that each vertex is assigned to exactly one partition (Equation (5.9.2)), that the size of each partition is between a lower bound (L) and an upper bound (U) (Equation (5.9.3) and Equation (5.9.4)), and that an edge is cut if its endpoints are in different partitions (Equation (5.9.5) and Equation (5.9.6)). Finally, Equation (5.9.7) and Equation (5.9.8) ensure that all decision variables are binary.

Definition 5.4 — GPP model.

$$\min_{x,y} \quad \sum_{e \in E} y_e \tag{5.9.1}$$

$$\text{s.t.} \quad \sum_{p=1}^{k} x_{vp} = 1 \qquad \forall v \in V \tag{5.9.2}$$

$$\sum_{v \in V} x_{vp} \geq L \qquad p = 1, ..., k \tag{5.9.3}$$

$$\sum_{v \in V} x_{vp} \leq U \qquad p = 1, ..., k \tag{5.9.4}$$

$$y_e \geq x_{up} - x_{vp} \quad \forall e = \{u, v\} \in E \quad p = 1, ..., k \tag{5.9.5}$$

$$y_e \geq x_{vp} - x_{up} \quad \forall e = \{u, v\} \in E \quad p = 1, ..., k \tag{5.9.6}$$

$$x_{vp} \in \{0, 1\} \qquad \forall v \in V \quad p = 1, ..., k \tag{5.9.7}$$

$$y_e \in \{0, 1\} \qquad \forall e \in E \tag{5.9.8}$$

where:

y_e	decision variables (if edge e is cut)
x_{vp}	decision variables (if vertex v is assigned to partition p)
L	lower bound on the size of each partition
U	upper bound on the size of each partition

5.4.3 Solving GPP with Julia

The following example shows the GPP instance that will be used in this subsection.

Example 5.4 Consider the undirected graph $G = (V, E)$ with $n = |V| = 8$ vertices shown in Figure 5.9. We want to partition the graph into $k = 2$ disjoint subsets (partitions) such that the number of edges connecting vertices in different partitions is minimized. We also set a lower bound of $L = \lfloor \frac{n}{k} \rfloor = 4$ and an upper bound of $U = \lceil \frac{n}{k} \rceil = 4$ on the size of each partition.

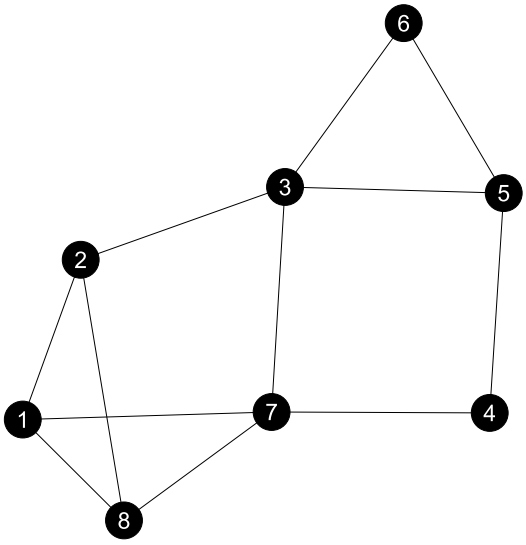


Figure 5.9: GPP instance.

Listing 5.7 presents a Julia implementation for solving the MISP instance shown in Figure 5.9. The code uses the JuMP package for modeling the optimization problem, the HiGHS solver for solving the model, and the Graphs package for handling graph data structures. It begins by creating the graph instance (Listing 5.7-8), defining the GPP model (Listing 5.7-37) and solving it. Finally, it uses the function `plot_solution` (Listing 5.7-87), defined in `utils/gpp_utils.jl`, to visualize the solution by highlighting the vertices with their assigned colors, and the edges that are cut.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/gpp_utils.jl")

# Create graph
graph = Graphs.Graph(8)

# Edges
edges = [(1,2), (1,7), (1,8), (2,3), (2,8), (3,5), (3,6), (3,7),
(4,5), (4,7), (5,6), (7,8)]

# Add edges
for (u, v) in edges
    Graphs.add_edge!(graph, u, v)
```

```
end

# Total of vertices
n = Graphs.nv(graph)

# Total of edges
m = Graphs.ne(graph)

# Number of partitions
k = 2

# Auxiliary variables to simplify notation
V = 1:Graphs.nv(graph) # Range of vertices
E = 1:Graphs.ne(graph) # Range of edges
P = 1:k                # Range of partitions

# Define lower and upper size limits for each partition
L = floor(Int, n / k)
U = ceil(Int, n / k)

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, y[e in E], Bin)
@variable(model, x[v in V, p in P], Bin)

# Objective: minimize number of cut edges
@objective(model, Min, sum(y[e] for e in E))

# Each vertex must be assigned to exactly one partition
@constraint(model, [v in V], sum(x[v, p] for p in P) == 1)

# Partition balance constraint
@constraint(model, [p in P], sum(x[v, p] for v in V) >= L)
@constraint(model, [p in P], sum(x[v, p] for v in V) <= U)

# Define y[e] = 1 if edge connects different partitions
```

```
for (e_id, e) in enumerate(Graphs.edges(graph))
    u, v = e.src, e.dst
    for p in P
        @constraint(model, y[e_id] >= x[u, p] - x[v, p])
        @constraint(model, y[e_id] >= x[v, p] - x[u, p])
    end
end

# Run the solver
JuMP.optimize!(model)

# Print solution
println("Edge cuts total: ", JuMP.objective_value(model))
edge_cuts = falses(m)
for (e_id, e) in enumerate(Graphs.edges(graph))
    if JuMP.value(y[e_id]) > 0.5
        println("Edge cut: {$(e.src),$(e.dst)}")
        edge_cuts[e_id] = true
    end
end

# Extract partition from solution
partition = Vector{Vector{Int}}()
for p in P
    cluster = [v for v in V if JuMP.value(x[v,p]) > 0.5]
    push!(partition, cluster)
end
println("Partition: ", partition)

# Plot solution
plot_solution(graph, partition, edge_cuts, k)
```

Listing 5.7: Solving GPP with Julia.

Listing 5.8 presents the output of the GPP model for the given graph instance. The output indicates the total number of edges that are cut (i.e., edges that connect vertices in different partitions) and the specific edges that are cut. Additionally, it shows the resulting partitions of the vertex set.

Output

```
Edge cuts total: 3.0
```

```
Edge cut: {2,3}
Edge cut: {3,7}
Edge cut: {4,7}
Partition: [[3, 4, 5, 6], [1, 2, 7, 8]]
```

Listing 5.8: Output of the GPP model.

Figure 5.10 presents the visual representation of the solution for the GPP instance. In the graph, the vertices are colored according to their assigned partitions, with different colors representing different partitions. The edges that are cut (i.e., edges that connect vertices in different partitions) are highlighted to show the connections between the partitions.

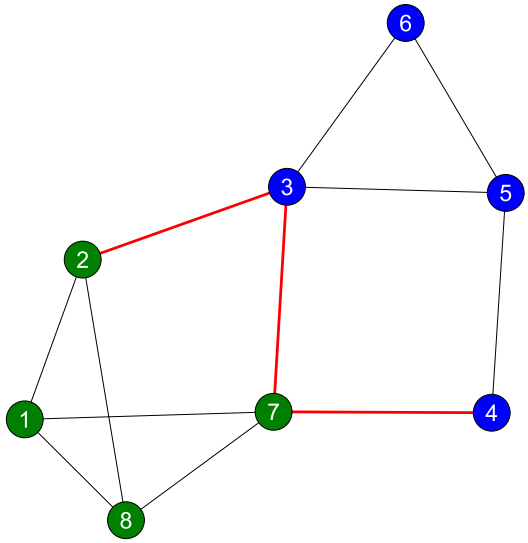


Figure 5.10: Solution for GPP example.

5.4.4 Solving GPP using Metis

Metis partitioning algorithm refers to a family of multilevel graph partitioning algorithms developed in [64]. It partitions a graph into k roughly equal-sized parts while minimizing the number of edges cut. Metis is scalable, working well on graphs with millions of nodes and edges.

It works in three main phases:

1. Coarsening Phase:
    - The original graph is gradually reduced in size by collapsing (contracting) pairs of vertices into supervertices.
    - Each coarsened graph is smaller but maintains the structure of the original

- The goal is to reduce the graph to a size that's easy to partition efficiently

Matching strategies like heavy-edge matching are used to contract edges with larger weights (which likely represent strong connections).

2. Initial Partitioning Phase
   - Once the graph is small enough, a simple partitioning algorithm (e.g., greedy or spectral method) is used to split it into k parts
   - This partition serves as the initial guess for the original graph.
3. Uncoarsening & Refinement Phase
   - The partition is projected back (uncoarsened) to each higher-resolution graph.
   - At each level, the partition is refined using local optimization heuristics: Kernighan–Lin (KL) and Fiduccia–Mattheyses (FM)
   - Refinement aims to reduce the number of edge cuts while keeping the partition balanced.

We are going to use the Metis.jl Julia package to solve the graph the example introduced in Figure 5.9 using $k = 2$ partitions. The code in Listing 5.9 uses the JuMP package for modeling the optimization problem, the HiGHS solver for solving the model, and the Graphs package for handling graph data structures. It begins by creating the graph instance (Listing 5.9-11) and solving the GPP with Metis heuristic (Listing 5.9-25). Finally, it uses the function `plot_solution` (Listing 5.9-44), defined in `utils/metis_utils.jl`, to visualize the solution by highlighting the vertices with their assigned colors, and the edges that are cut.

```
using Random # For random number generation
using Graphs # Graphs package
using Metis  # Metis package for graph partitioning

# Load utility function for plotting
include("utils/metis_utils.jl")

# Set seed for reproducibility
Random.seed!(0)

# Create graph
graph = Graphs.Graph(8)

```

```
# Edges
edges = [(1,2), (1,7), (1,8), (2,3), (2,8), (3,5), (3,6), (3,7),
(4,5), (4,7), (5,6), (7,8)]

# Add edges
for (src, dst) in edges
    Graphs.add_edge!(graph, src, dst)
end

# Number of partitions
k = 2

# Partition the graph using Metis
partition = Metis.partition(graph, k)

# Print edge cuts
edge_cuts = falses(Graphs.ne(graph))
for (e_id, e) in enumerate(Graphs.edges(graph))
    src, dst = Graphs.src(e), Graphs.dst(e)
    if partition[src] != partition[dst]
        println("Edge cut: {$src,$dst}")
        edge_cuts[e_id] = true
    end
end

# Print the partition
println("Partition:")
for i in 1:k
    println("Cluster $i: ", findall(partition .== i))
end

# Plot the solution
plot_solution(graph, partition, edge_cuts, k)
```

Listing 5.9: Solving GPP with Julia using Metis heuristic.

Listing 5.10 presents the output of the Metis heuristic for the GPP instance. The output shows the partitions of the vertex set as determined by the Metis algorithm. Each group corresponds to a partition, and the vertices in each group are listed together.

```
Output
Edge cut: {2,3}
Edge cut: {3,7}
Edge cut: {4,7}
Partition:
Cluster 1: [3, 4, 5, 6]
Cluster 2: [1, 2, 7, 8]
```

Listing 5.10: Output of the Metis heuristic.

Figure 5.11 presents the visual representation of the solution for the GPP instance as determined by the Metis heuristic. In the graph, the vertices are colored according to their assigned partitions, with different colors representing different partitions. The visualization allows us to see how the vertices are grouped together based on the Metis partitioning, and the edges that are cut (i.e., edges that connect vertices in different partitions) are highlighted to show the connections between the partitions.

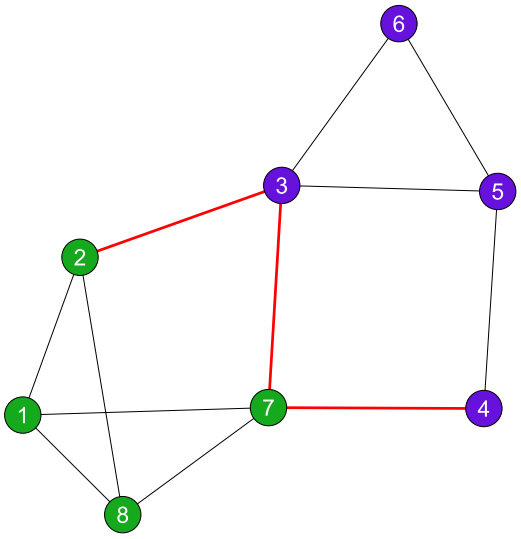


Figure 5.11: Metis solution for GPP example.

5.4.5 Weighted Graph Partitioning Problem

To solve GPP for weighted graphs we only need to modify Equation (5.9) objective function to consider the weights of the edges. Let w_e be the weight of edge $e \in E$. Equation (5.10) presents the **Weighted Graph Partitioning Problem** (WGPP). In this model, the objective is to minimize the sum of the weights of the cut edges, which corresponds to minimizing the total weight of the edges that are cut. The constraints remain the same as in the GPP model (Equation (5.9)).

Definition 5.5 — WGPP model.

$$\min_{x,y} \quad \sum_{e \in E} w_e y_e \tag{5.10.1}$$

$$\text{s.t.} \quad \sum_{p=1}^{k} x_{vp} = 1 \qquad \forall v \in V \tag{5.10.2}$$

$$\sum_{v \in V} x_{vp} \geq L \qquad p = 1, ..., k \tag{5.10.3}$$

$$\sum_{v \in V} x_{vp} \leq U \qquad p = 1, ..., k \tag{5.10.4}$$

$$y_e \geq x_{up} - x_{vp} \quad \forall e = \{u, v\} \in E \quad p = 1, ..., k \tag{5.10.5}$$

$$y_e \geq x_{vp} - x_{up} \quad \forall e = \{u, v\} \in E \quad p = 1, ..., k \tag{5.10.6}$$

$$x_{vp} \in \{0, 1\} \qquad \forall v \in V \quad p = 1, ..., k \tag{5.10.7}$$

$$y_e \in \{0, 1\} \qquad \forall e \in E \tag{5.10.8}$$

where:

w_e	edge e weight
y_e	decision variables (if edge e is cut)
x_{vp}	decision variables (if vertex v is assigned to partition p)
L	lower bound on the size of each partition
U	upper bound on the size of each partition

The following example shows the WGPP instance that will be used in this subsection.

Example 5.5 Consider the undirected graph $G = (V, E)$ with $n = |V| = 8$ vertices and weighted edges shown in Figure 5.12. We want to partition the graph into $k = 2$ disjoint subsets (partitions) such that the total weight of the edges connecting vertices in different partitions is minimized. We also set a lower bound of $L = \lfloor \frac{n}{k} \rfloor = 4$ and an upper bound of $U = \lceil \frac{n}{k} \rceil = 4$ on the size of each partition.

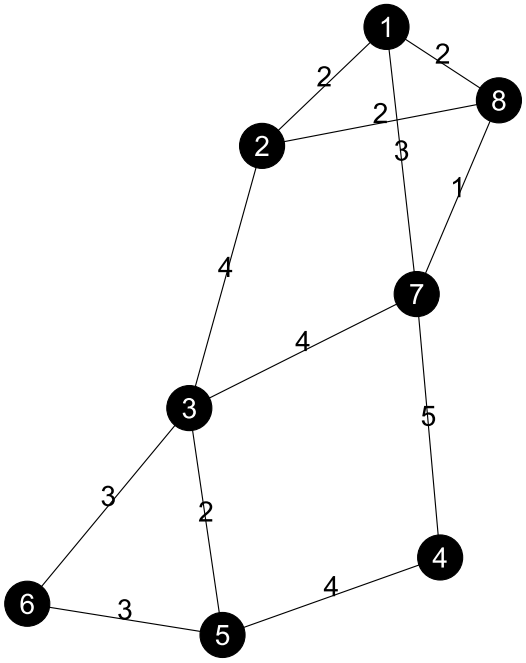


Figure 5.12: WGPP instance.

Listing 5.11 presents a Julia implementation to solve the WGPP instance shown in Figure 5.12. The code is similar to the one used for solving GPP, with the main difference being in the objective function of the model, which now considers the weights of the edges Listing 5.11-46.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package
using SimpleWeightedGraphs # For weighted graphs

# Load utility function for plotting
include("utils/wgpp_utils.jl")

# Create a weighted graph
graph = SimpleWeightedGraph(8)

# Add edges with weights
edges = [(1,2,2), (1,7,3), (1,8,2), (2,3,4), (2,8,2), (3,5,2),
(3,6,3), (3,7,4), (4,5,4), (4,7,5), (5,6,3), (7,8,1)]
for (src, dst, w) in edges
    Graphs.add_edge!(graph, src, dst, w)
end

# Total of vertices
```

```
n = Graphs.nv(graph)

# Total of edges
m = Graphs.ne(graph)

# Number of partitions
k = 2

# Auxiliary variables to simplify notation
V = 1:Graphs.nv(graph) # Range of vertices
E = 1:Graphs.ne(graph) # Range of edges
P = 1:k                # Range of partitions

# Define lower and upper size limits for each partition
L = floor(Int, n / k)
U = ceil(Int, n / k)

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, y[e in E], Bin)
@variable(model, x[v in V, p in P], Bin)

# Objective: minimize the weight of cut edges
@objective(model, Min, sum(graph.weights[e.src, e.dst] * y[e_id] for
(e_id, e) in enumerate(Graphs.edges(graph))))

# Each vertex must be assigned to exactly one partition
@constraint(model, [v in V], sum(x[v, p] for p in P) == 1)

# Partition balance constraint
@constraint(model, [p in P], sum(x[v, p] for v in V) >= L)
@constraint(model, [p in P], sum(x[v, p] for v in V) <= U)

# Define y[e] = 1 if edge connects different partitions
for (e_id, e) in enumerate(Graphs.edges(graph))
    u, v = e.src, e.dst
```

```
    for p in P
        @constraint(model, y[e_id] >= x[u, p] - x[v, p])
        @constraint(model, y[e_id] >= x[v, p] - x[u, p])
    end
end

# Run the solver
JuMP.optimize!(model)

# Print solution
println("Edge cuts total: ", JuMP.objective_value(model))
edge_cuts = falses(m)
for (e_id, e) in enumerate(Graphs.edges(graph))
    if JuMP.value(y[e_id]) > 0.5
        println("Edge cut: {$(e.src),$(e.dst)}")
        edge_cuts[e_id] = true
    end
end

# Extract partition from solution
partition = Vector{Vector{Int}}()
for p in P
    cluster = [v for v in V if JuMP.value(x[v,p]) > 0.5]
    push!(partition, cluster)
end
println("Partition: ", partition)

# Plot solution
plot_solution(graph, partition, edge_cuts, k)
```

Listing 5.11: Solving WGPP with Julia.

Listing 5.12 presents the output of the WGPP model for the given graph instance. The output indicates the total weight of the edges that are cut (i.e., edges that connect vertices in different partitions) and the specific edges that are cut. Additionally, it shows the resulting partitions of the vertex set.

Output

```
Edge cuts total: 12.0
Edge cut: {1,2}
Edge cut: {4,5}
```

```
Edge cut: {3,7}
Edge cut: {2,8}
Partition: [[2, 3, 5, 6], [1, 4, 7, 8]]
```

Listing 5.12: Output of the WGPP model.

Figure 5.13 presents the visual representation of the solution for the WGPP instance. In the graph, the vertices are colored according to their assigned partitions, with different colors representing different partitions. The edges that are cut (i.e., edges that connect vertices in different partitions) are highlighted to show the connections between the partitions, and the weights of the edges are displayed as labels.

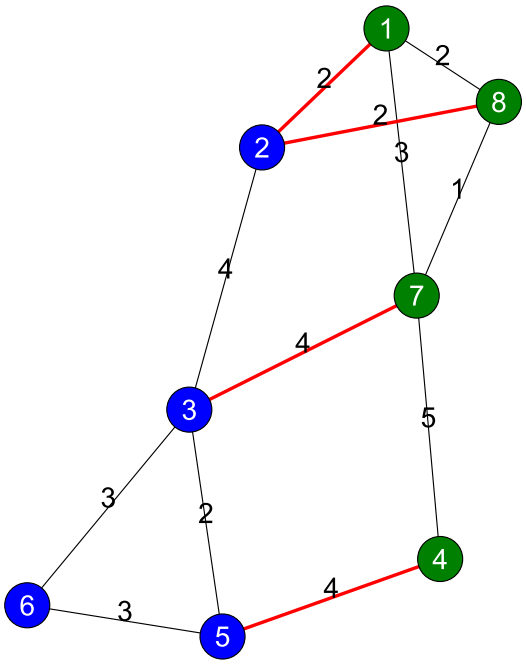


Figure 5.13: Solution for WGPP example.

5.5 Clique Partitioning Problem

The **Clique Partitioning Problem** (CPP) involves partitioning the vertex set of an undirected graph $G = (V, E)$ into disjoint subsets $V_1, V_2, l..., V_k$. The partitioning is guided by maximizing a metric denoted as **modularity** (Q). The intuition behind modularity is that a good partition should have many edges within subsets and few edges between subsets. The subsets obtained are also denoted as communities, and the problem is often referred to as **community detection** in the context of social networks [65].

The CPP is computationally hard (NP-hard). This makes it a difficult problem for large graphs or real-time applications.

5.5.1 Modularity

Modularity (Q) measures the density of edges within communities compared to a null model (a random graph with the same degree distribution) [65]. It can achieve values in the range of –1 to 1, where higher values indicate better partition quality:

- Q close to 1: many edges within subsets, few between.
- Q close to 0: no significant partition structure (similar to random)
- Q close to –1: anti-community structure (more edges between subsets than within).

The modularity is defined as:

$$Q = \sum_{u,v \in V} \left[\frac{A_{uv}}{2m} - \frac{d_u d_v}{4m^2} \right] \delta(c_u, c_v) \tag{5.11}$$

where:

- A_{uv} is the adjacency matrix of the graph (1 if edge exists, 0 otherwise)
- d_u is the degree of vertex u
- d_v is the degree of vertex v
- m is the total number of edges in the graph
- $\delta(c_u, c_v)$ is the Kronecker delta function, which is 1 if vertices u and v are in the same community (i.e., $c_u = c_v$) and 0 otherwise.

The modularity is important because:

- **Does not require knowing k:** Unlike the Graph Partitioning Problem (where you fix the number of groups), algorithms based on modularity automatically find the ideal number of communities that maximizes Q.
- **Identifies Cohesion:** It penalizes forced divisions in graphs that do not possess natural community structure.

This metric also has a known limitation: the resolution limit [66]. In very large graphs, it can fail to identify small communities, merging them into larger groups just to increase the overall Q value, even when they should be kept separate.

5.5.2 CPP applications

Table 5.5 presents some real-world applications of the Clique Partitioning Problem.

Application	Description
Healthcare	Group patients with similar characteristics into communities for risk analysis, disease detection, and treatment personalization. This helps identify groups of patients with similar health patterns, enabling more targeted and effective medical interventions [67]
Social Network	Partitioning a social network into communities or clusters of users who are more closely connected to each other than to the rest of the network. It is used for community detection, influence analysis, and targeted marketing by identifying groups of users with similar interests or behaviors [68]
Bioinformatics	Partitioning genes or proteins in a biological network into functional modules or communities. It is used to identify groups of genes or proteins that are more closely related to each other than to the rest of the network, which can help in understanding biological processes and disease mechanisms [69]
Cloud computing	Partitioning virtual machines or containers in a cloud environment to optimize resource usage and reduce costs. It helps in load balancing, fault tolerance, and efficient utilization of cloud resources by grouping workloads with similar characteristics [70]

Table 5.5: Real-world applications of the Clique Partitioning Problem.

5.5.3 CPP model

Equation (5.12) presents an Integer Linear Programming (ILP) model for the Clique Partitioning Problem (CPP). The model defines decision variables, an objective function to maximize modularity, and constraints to ensure that the partitioning is valid (i.e., vertices are grouped into communities without overlap). The objective function incorporates the modularity contribution of pairs of vertices, while the constraints ensure that the partitioning respects the structure of the graph and that each pair of vertices is assigned to the same community if they are connected.

Definition 5.6 — CPP model.

$$\max_{x} \quad \sum_{u \in V} \sum_{v \in V} q_{uv} x_{uv} \tag{5.12.1}$$

$$\text{s.t.} \quad x_{uv} + x_{vw} - x_{uw} \leq 1 \qquad u, v, w \in V \tag{5.12.2}$$

$$x_{uv} - x_{vw} + x_{uw} \leq 1 \qquad u, v, w \in V \tag{5.12.3}$$

$$-x_{uv} + x_{vw} + x_{uw} \leq 1 \qquad u, v, w \in V \tag{5.12.4}$$

$$x_{uv} \in \{0, 1\} \qquad \forall u, v \in V \tag{5.12.5}$$

where:

- q_{uv} is equal to $\frac{A_{uv}}{2m} - \frac{d_u d_v}{4m^2}$ (modularity contribution for vertices u and v)
- x_{uv} decision variables (1 if vertices u and v are in the same community, 0 otherwise)

5.5.4 Solving CPP with Julia

The following example shows the CPP instance that will be used in this subsection.

Example 5.6 Consider the undirected graph $G = (V, E)$ with $n = |V| = 24$ vertices and $m = |E| = 36$ edges shown in Figure 5.14. We want to partition the graph into communities such that the modularity of the partitioning is maximized.

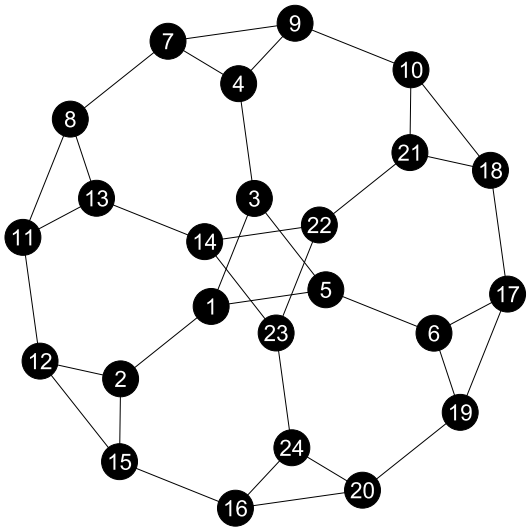


Figure 5.14: CPP instance.

Listing 5.13 presents a Julia implementation to solve the CPP instance shown in Figure 5.14. The code uses the JuMP package for modeling the optimization

problem, the HiGHS package as the solver, and the Graphs package for graph representation. It first creates the graph instance (Listing 5.13-8) and then defines the modularity matrix Q based on the graph structure (Listing 5.13-18). Next, it sets up the ILP model for CPP, defining the decision variables, objective function, and constraints (Listing 5.13-29). Finally, it runs the solver to find the optimal partitioning of the vertices and visualizes the solution using `plot_solution` (Listing 5.13-80), which colors the vertices according to their assigned communities and highlights the edges inside the communities.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/cpp_utils.jl")

# Create graph
graph = Graphs.smallgraph("truncatedcube")

# Get number of vertices and edges
n = Graphs.nv(graph)
m = Graphs.ne(graph)

# Auxiliary variables to simplify notation
V = 1:n

# Modularity matrix Q
Q = zeros(n,n)
for u in V
    for v in V
        du = Graphs.degree(graph, u)
        dv = Graphs.degree(graph, v)
        a_ij = Graphs.has_edge(graph, u, v) ? 1 : 0
        Q[u,v] = a_ij/(2*m) - (du * dv) / (4*m^2)
    end
end

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
```

```
JuMP.set_silent(model)

# Define the decision variables
@variable(model, x[u in V, v in V], Bin)

# Objective: maximize the weight of the selected edges
@objective(model, Max, sum(Q[u,v] * x[u,v] for u in V for v in V))

# Transitive Constraints: ensure that the selected edges form a
valid partition
for u in V
    for v in V
        for w in V
            @constraint(model,  x[u,v] + x[v,w] - x[u,w] <= 1)
            @constraint(model,  x[u,v] - x[v,w] + x[u,w] <= 1)
            @constraint(model, -x[u,v] + x[v,w] + x[u,w] <= 1)
        end
    end
end

# Run the solver
JuMP.optimize!(model)

# Get the objective (modularity)
modularity = JuMP.objective_value(model)
println("Modularity: ", modularity)

# Extract the optimal partition
println("Partition:")
partition = zeros(Int64, n)
partition_id = 1
for u in V
    # If vertex u is not yet assigned to a partition
    if partition[u] == 0
        print("Cluster $partition_id: ")
        for v in V
            # If vertex v is in the same partition as u
            if JuMP.value(x[u,v]) > 0.5
                # Assign both u and v to the same partition
                partition[v] = partition_id
                print("$v ")
```

```
            end
        end
        print("\n")
        partition_id += 1
    end
end

# Plot solution
plot_solution(graph, partition)
```

Listing 5.13: Solving CPP with Julia.

Listing 5.14 presents the output of the CPP model for the given graph instance. The output indicates the modularity of the obtained partitionin and lists the clusters of vertices that were identified by the model. Each cluster corresponds to a community in the graph.

Output

```
Modularity: 0.5416666666666666
Partition:
Cluster 1: 1 3 5
Cluster 2: 2 12 15
Cluster 3: 4 7 9
Cluster 4: 6 17 19
Cluster 5: 8 11 13
Cluster 6: 10 18 21
Cluster 7: 14 22 23
Cluster 8: 16 20 24
```

Listing 5.14: Output of the Metis heuristic.

Figure 5.15 presents the visual representation of the solution for the CPP instance. In the graph, the vertices are colored according to their assigned communities, with different colors representing different communities. The visualization allows us to see how the vertices are grouped together based on the partitioning that maximizes modularity.

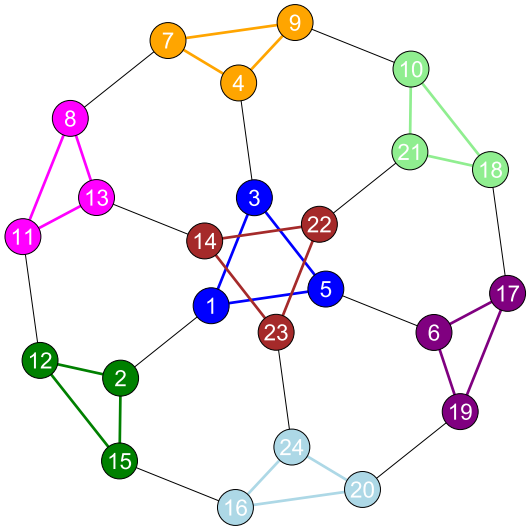


Figure 5.15: Solution for CPP example.

5.6 Final remarks

In this chapter, we examined five foundational graph problems (Maximum Independent Set, Vertex Cover, Graph Coloring, Graph Partitioning, and Clique Partitioning) and demonstrated how each can be formulated as an IP model. Despite their differences, these problems share a common structure that makes them well-suited for mathematical optimization techniques.

By expressing these combinatorial problems through IP formulations, we unlock the potential to apply general-purpose solvers to obtain high-quality solutions. This approach not only offers a unified modeling framework but also enables scalable and flexible solution methods that can be tailored to specific application contexts.

These graph problems are more than academic exercises; they have widespread applications in areas such as communication networks, scheduling, bioinformatics, and logistics. As such, mastering their IP formulations provides a valuable foundation for solving a broad range of real-world optimization challenges.

5.7 Exercises

Formulate and solve the following exercises using the Jump package.

Exercise 5.1 Solve the problem of point-feature cartographic label placement [71] as a maximum independent set of vertices problem, using the dataset

d25_01.dat presented in Figure 5.16 as input. The dataset follows the following format in Listing 5.15.

File description

```
# of points
# of candidate positions
# of labels in potential conflict with label 1 (degree of vertex 1)
labels in potential conflict with label 1
# of labels in potential conflict with label 2 (degree of vertex 2)
labels in potential conflict with label 2
```

Listing 5.15: Input format of `d25_01.dat`.

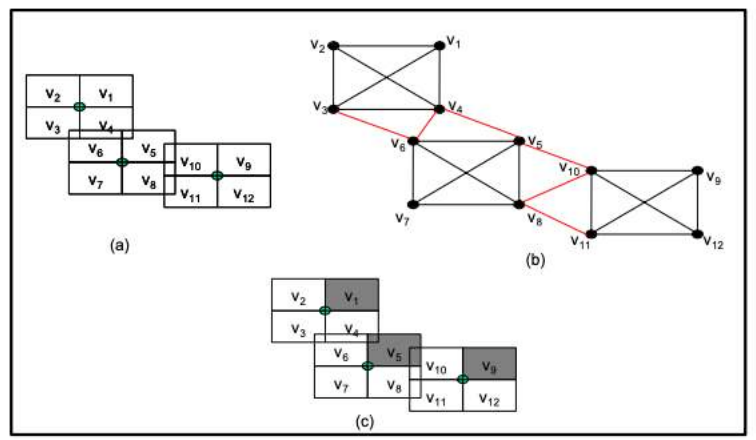


Figure 5.16: Point feature map labeling as a MISP: (a) 3 points and 4 candidate positions for each point. (b) graph corresponding to points and their intersections as edges. (c) conflict-free solution.

Note: the dataset was obtained from Point-feature cartographic label placement datasets.

Exercise 5.2 The director of a museum must decide how many guards should be employed to control its rooms. Each guard can guard the room in which he is stationed and adjacent ones through the doors between the rooms. Formulate an integer linear programming problem to give the minimum the number of guards so that all rooms are guarded (Vertex Covering Problem). Solve it for the map displayed in Figure 5.17.

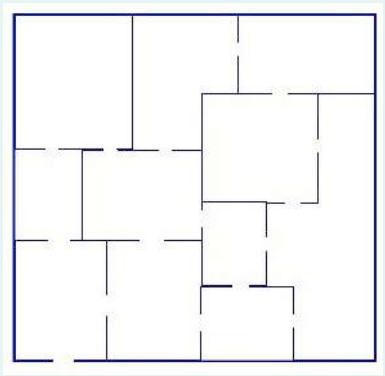


Figure 5.17: Museum map for the vertex covering problem.

Exercise 5.3 Figure 5.18 presents a map with 34 districts of São José dos Campos, Brazil. Formulate and solve the Graph Coloring Problem to color the map with the minimum number of colors such that no two adjacent districts share the same color. Use the dataset districts.csv as input.

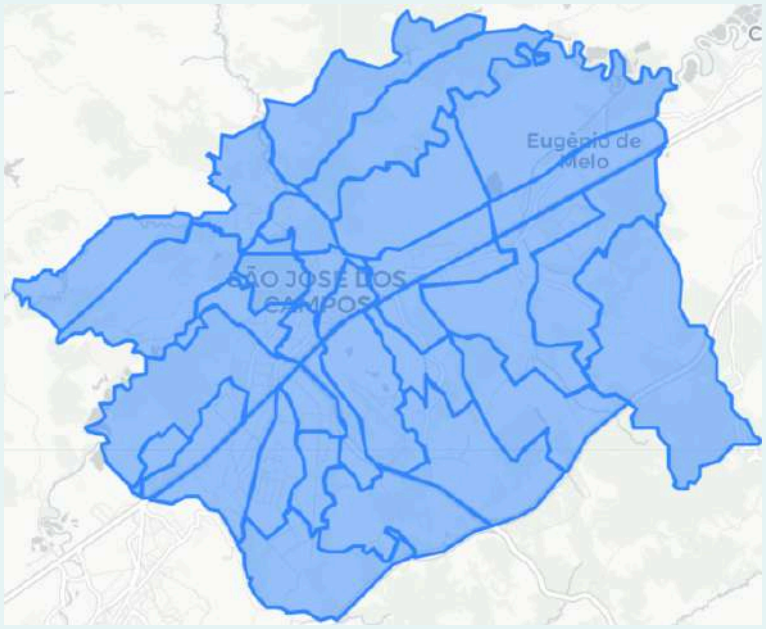


Figure 5.18: Map instance for the Graph Coloring Problem.

Exercise 5.4 The circuit board in Figure 5.19 has 48 components. Formulate and solve the Graph Partitioning Problem into 3 balanced groups, while minimizing the number cut edges. Use the graph dataset circuit.csv as input, which contains the list of components and their connections.

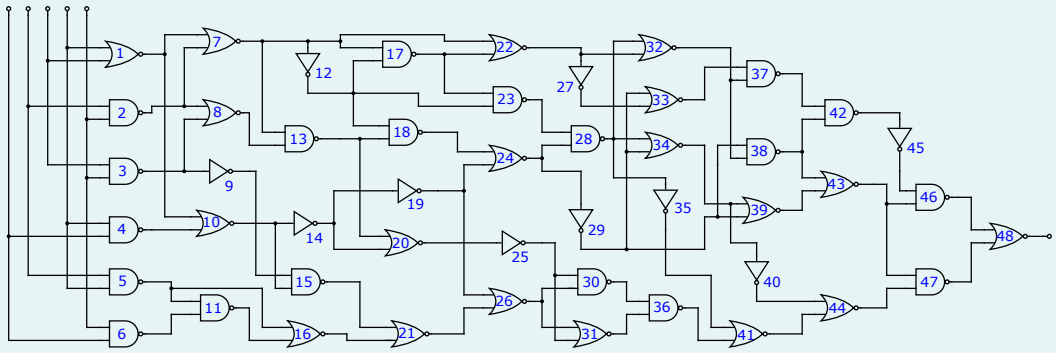


Figure 5.19: Circuit board instance for the Graph Partitioning Problem.

Exercise 5.5 Formulate and solve the Clique Partitioning Problem for the graph in Figure 5.20, which represents a social network of 34 individuals. The goal is to partition the graph into communities that maximize modularity. Use the dataset karate.csv as input, which contains the list of individuals and their connections.

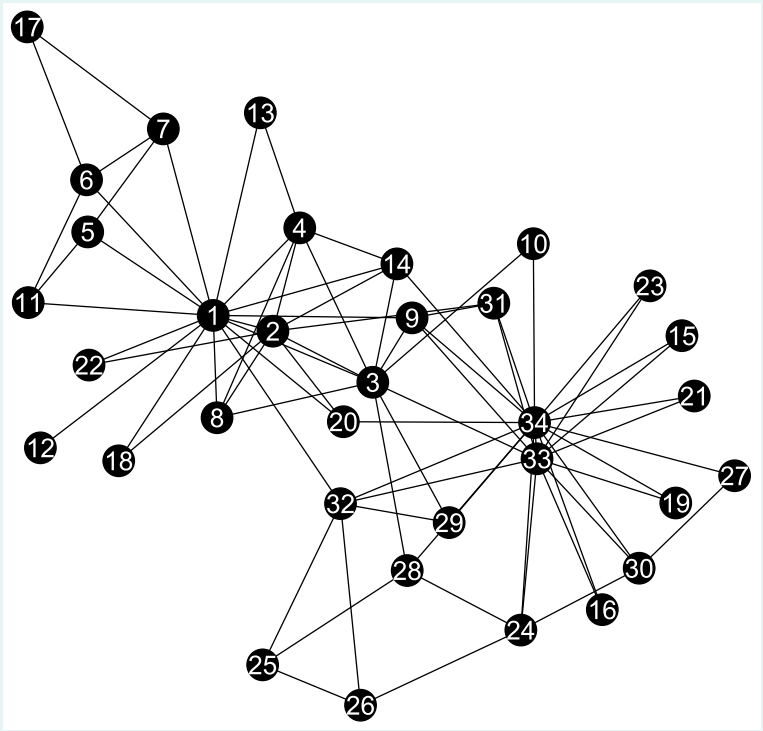


Figure 5.20: Social network instance for the Clique Partitioning Problem.

II Advanced Techniques

6 Column Generation 133
6.1 Column Generation technique 133
6.2 The one-dimensional cutting stock problem 134
6.3 The P-Median Problem 149
6.4 Final remarks 160
6.5 Exercises 161

7 Lagrangian Relaxation 163
7.1 Lagrangian Relaxation algorithm . . 163
7.2 Simple Example of Lagrangian Relaxation 168
7.3 Lagrangian Relaxation for P-Median Problem 175
7.4 Lagrangian Relaxation for the Maximum Independent Set Problem 184
7.5 Lagrangian relaxation for the Generalized Assignment Problem . 199
7.6 Final remarks 202
7.7 Exercises 203

8 Branch-and-Bound 205
8.1 Branch-and-Bound Components . . 205
8.2 Maximization example: Binary Knapsack Problem 209
8.3 Minimization example: Traveling Salesman Problem 221
8.4 Final remarks 231
8.5 Exercises 231

6 Column Generation

The treatment of large combinatorial problems has received a great advance, thanks to the development of faster and more efficient commercial and open-source optimization packages. Such tools allow that inherently complex problems can also be solved in acceptable computational time, through the use of combined techniques such as, for example, the **Column Generation** (CG) method applied to IP problems.

The CG idea, formalized through the Dantzig-Wolfe decomposition [72], was first practically applied in the context of the one-dimensional cutting stock problem [73], [74]. Since then, it has been explored in various applications, such as the stock cutting, vehicle routing, crew scale and VLSI circuit designs [75].

In this chapter we will introduce the CG technique for two IP problems. The first will be the cutting stock, as shown in [73]. Next, we will present the CG technique applied to the P-Median Problem, as shown in the work [23].

6.1 Column Generation technique

Column Generation is iterative decomposition technique for solving linear problems that typically have a prohibitively large number of variables (columns). The main idea of the algorithm is to remove all variables from the original model and then iteratively reintroduce a limited number of them until it is possible to prove that the smaller problem contains at least one optimal solution to the original problem.

Figure 6.1 presents a flowchart of the CG method. The algorithm starts with a small subset of columns (feasible solutions) and solves a Restricted Master Problem (RMP) to find the optimal solution and the corresponding dual variables. Then, it iteratively improves the solution by solving a pricing subproblem, which uses the dual variables from the RMP to identify new columns that can potentially improve the objective function. If a new column with negative reduced cost is found, it is added to the RMP and the process is repeated until no such column is found, indicating that the current solution is optimal for the original problem.

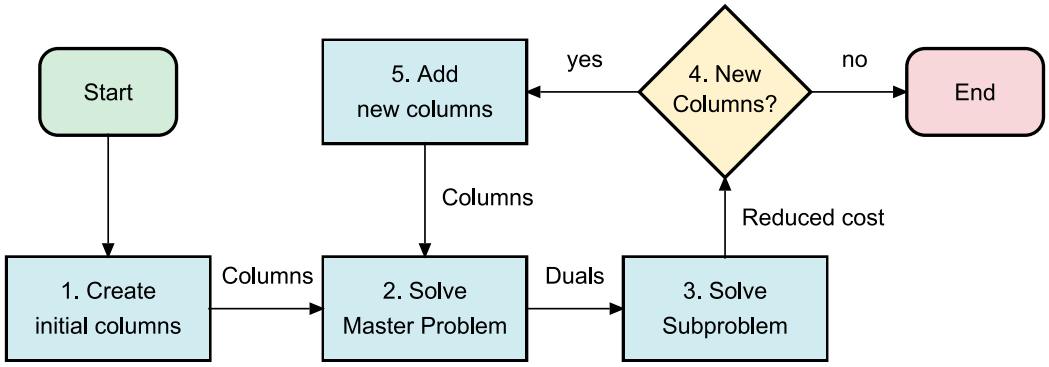


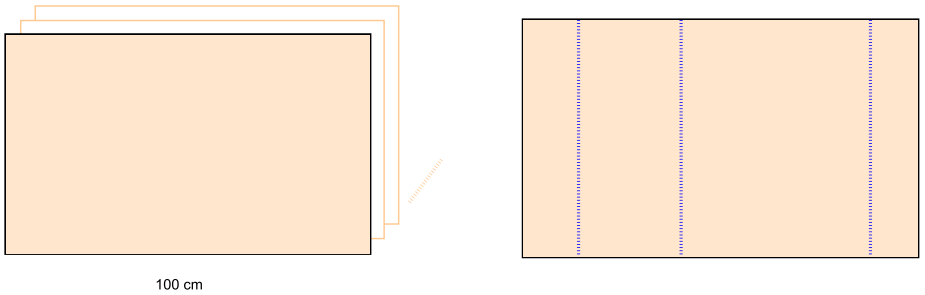
Figure 6.1: Column Generation flowchart.

In the next section, we will apply the CG technique to two IP problems. The first one is the cutting stock problem, which was the first practical application of CG.

6.2 The one-dimensional cutting stock problem

The **One-dimensional Cutting Stock Problem** (1D-CSP) involves cutting standard-sized stock materials (like rolls, rods, panels, or pipes) into smaller pieces of specified lengths to meet customer demands, while minimizing waste or the number of stock materials used [74].

Figure 6.2 introduces the 1D-CSP in the context of cutting panels of fixed length into smaller pieces to meet demand. The goal is to determine the optimal cutting patterns that minimize the number of panels used while satisfying the demand for each item type.



(a) Stock of panels. (b) Cutting pattern.

Figure 6.2: 1D-CSP for panels.

In the next section, we will present one of the first mathematical formulations for the 1D-CSP. We will also show how to solve a small instance in Julia, considering panel cutting as in Figure 6.2.

6.2.1 Kantorovich cutting stock model

Kantorovich [76] proposed a mathematical formulation for the cutting stock problem defined in Equation (6.1). The KCS model is a mixed-integer linear programming (MILP) formulation in which the objective is to minimize the number of panels used (Equation (6.1.1)) while satisfying the demand for each item type (Equation (6.1.2)) and ensuring that the total length of items cut from each panel does not exceed its length (Equation (6.1.3)).

Definition 6.1 — KCS model. Given a set of itens (I) and a set of panels (P).

$$\min_{x,y} \quad \sum_{p \in P} y_p \tag{6.1.1}$$

$$\text{s.t.} \quad \sum_{p \in P} x_{ip} \geq d_i \qquad \forall i \in I \tag{6.1.2}$$

$$\sum_{i \in I} l_i x_{ip} \leq L y_p \quad \forall p \in P \tag{6.1.3}$$

$$x_{ip} \in \mathbb{Z}_{\geq 0} \qquad \forall i \in I \quad \forall p \in P \tag{6.1.4}$$

$$y_p \in \{0, 1\} \qquad \forall p \in P \tag{6.1.5}$$

where:

y_p	decision variables (if panel p is used)
x_{ip}	decision variables (total items of type i are cut in panel p)
d_i	demand for items of type i
l_i	length of items of type i
L	length of the panels

Equation (6.1) formulation has some drawbacks:

1. As the number of item types or panel lengths increases, the model becomes harder to solve due to the exponential growth in possible patterns;
2. The model assumes perfect cutting without waste (other than trim loss) and does not account for practical constraints like setup times, machine limitations, or defects in raw material;
3. The model does not explicitly consider the generation of cutting patterns, which is a key aspect of solving the cutting stock problem efficiently. This is where techniques like column generation come into play.

In the next section, we will present a Julia implementation to solve the KCS model for a small instance of the cutting stock problem. After that, we will apply the CG technique to solve the same example.

6.2.2 Solving the KCS model with Julia

This subsection will consider the following example of the CSP.

Example 6.1 Given a set of itens types $I = \{1, 2, 3, 4, 5\}$, with lengths $l = (45, 30, 20, 15, 5)$, demands $d = (60, 88, 35, 110, 200)$, and a stock piece of length $L = 100$. Determine the optimal cutting patterns to meet the demand while minimizing the number of stock pieces used (Figure 6.3).

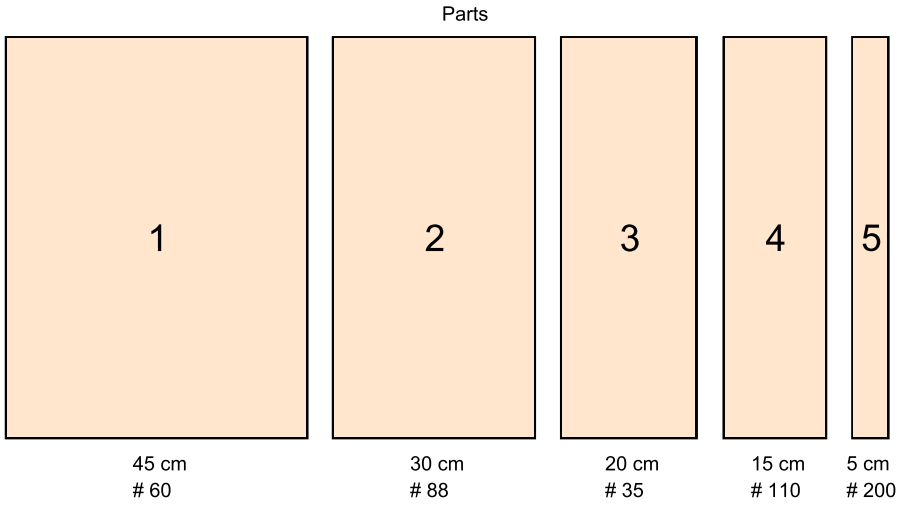


Figure 6.3: Cutting stock instance: item types.

Listing 6.1 presents the Julia code to solve the KCS model for the given example. The code uses the JuMP modeling language and the HiGHS solver. First, it defines the problem data (Listing 6.1-8), including the length of the stock piece, the number of stock pieces available, and the item types with their lengths and demands. It defines the decision variables, objective function, and constraints according to the KCS model (Listing 6.1-21). After solving the model, it extracts the solution, counts the unique cutting patterns used, and prints a production summary for each item type (Listing 6.1-45). Finally, it plots the cutting patterns using a utility function defined in `utils/1dcut_utils.jl` (Listing 6.1-88).

```
1 using JuMP   # Modeling language
2 using HiGHS  # Solver
3 using Printf # For formatted printing
4
5 # Utility functions for plotting the 1D cut solution
```

```
include("utils/1dcut_utils.jl")

# Items data
l = [45, 30, 20, 15, 5]     # Length of each item type
d = [60, 88, 35, 110, 200]  # Demand for each part type
n = length(d)               # Number of item types

# Panels data
L = 100                     # Length of each panel
m = 100                     # Total of panels available

# Auxiliary variables to simplify notation
I = 1:n # Range for item types
P = 1:m # Range for panels

# Create the model
model = JuMP.Model(HiGHS.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Variables: number of pieces of item i cut in panel p
@variable(model, x[i in I, p in P] >= 0, Int)

# Variables: binary variable indicating if panel p is used
@variable(model, y[p in P], Bin)

# Objective: minimize the number of panels used
@objective(model, Min, sum(y[p] for p in P))

# Demand satisfaction
@constraint(model, [i in I], sum(x[i,p] for p in P) >= d[i])

# Panel length capacity
@constraint(model, [p in P], sum(l[i] * x[i,p] for i in I) <= L *
y[p])

# Solve the model
JuMP.optimize!(model)

# Print solution
```

```
println("Number of panels used: ", JuMP.objective_value(model), " / $m")

# Extract solution
y_opt = JuMP.value.(y)
x_opt = ceil.(Int64,JuMP.value.(x))

# Find and count each unique patterns
unique_patterns_total = Dict{Vector{Int}, Int}()
for p in P
    if y_opt[p] > 0.5 # Only consider used rolls
        pattern = collect(x_opt[:, p])
        if haskey(unique_patterns_total, pattern)
            unique_patterns_total[pattern] += 1
        else
            unique_patterns_total[pattern] = 1
        end
    end
end

# Sort the unique patterns by their counts in descending order
unique_patterns_total = sort(collect(unique_patterns_total), by = x -> x[2], rev=true)

# Print the unique patterns and their counts
println("\nCutting patterns:")
println("="^28)
@printf("%-20s | %-9s\n", "Pattern (Qty Items)", "Total")
println("-"^28)
for (pattern, count) in unique_patterns_total
    @printf("%-20s | %.0f\n", pattern, count)
end

# Print the production summary
println("\nProduction summary:")
println("="^43)
@printf("%-4s | %-5s | %-8s | %-8s\n", "Type", "Demand", "Production", "Overproduction")
println("-"^43)
for i in I
    produced = sum(x_opt[i, p] for p in P)
```

```
84      @printf("T%-3d | %-6d | %-10d | %-8d\n", i, d[i], produced,
    produced - d[i])
85  end
86
87  # Plot the cutting patterns
88  plot_solution(unique_patterns_total)
```

Listing 6.1: Kantorovich model Julia implementation.

Listing 6.2 presents the output of the KCS model, including the number of panels used, the unique cutting patterns found, and a production summary for each item type. The output shows that 87 panels were used to meet the demand for all item types without overproduction. The unique cutting patterns and their counts are also displayed, along with a production summary that confirms that the demand for each item type was met exactly.

Output

```
Number of panels used: 87.0 / 100

Cutting patterns:
============================
Pattern (Qty Items)  | Total
----------------------------
[0, 3, 0, 0, 2]      | 29
[2, 0, 0, 0, 2]      | 28
[0, 0, 0, 6, 2]      | 18
[0, 0, 5, 0, 0]      | 7
[0, 0, 0, 0, 20]     | 2
[1, 0, 0, 2, 5]      | 1
[2, 0, 0, 0, 0]      | 1
[1, 1, 0, 0, 5]      | 1

Production summary:
=========================================
Type | Demand | Production | Overproduction
-----------------------------------------
T1   | 60     | 60         | 0
T2   | 88     | 88         | 0
T3   | 35     | 35         | 0
T4   | 110    | 110        | 0
T5   | 200    | 200        | 0
```

Listing 6.2: Output of the KCS model.

Figure 6.4 presents a figure that visually represents the cutting patterns used in the solution of the KCS model. The figure provides an intuitive understanding of how the stock pieces are being utilized to meet the demand for each item type.

Item	Cutting Patterns								
	1	2	3	4	5	6	7	8	
T1	0	2	0	0	0	1	2	1	
T2	3	0	0	0	0	0	0	1	
T3	0	0	0	5	0	0	0	0	
T4	0	0	6	0	0	2	0	0	
T5	2	2	2	0	20	5	0	5	
Total	29	28	18	7	2	1	1	1	87

Figure 6.4: Solution for CSP example.

In the next section, we will apply the CG technique to solve the same example presented in this subsection.

6.2.3 Column generation for the one-dimensional cutting stock problem

Column Generation is a method particularly well-suited for solving large-scale Cutting Stock Problems [73], [74], where the number of cutting patterns is prohibitively large and cannot be enumerated explicitly from the outset.

We now present a formal treatment of how CG is applied to the 1D-CSP, demonstrating how to adapt the general column generation algorithmic framework (Figure 6.1) to the one-dimensional cutting stock problem.

Instead of considering every possible pattern at once, we start with a small subset and iteratively add only those patterns that have the potential to improve our objective. The following subsections present core components of the column generation framework used to solve the problem.

Step 1. Creating initial columns

A column in the context of the cutting stock problem represents a specific cutting pattern, which is a way to cut a stock piece into smaller pieces to meet demand. The initial columns can be generated using simple heuristics, such as cutting one item type at a time (basic patterns) or randomly generating patterns that fit within the stock length. The key is to ensure that the initial set of columns provides a feasible solution to the problem, allowing us to start the column generation process.

Step 2. The Restricted Master Problem (RMP)

The RMP is a Linear Programming relaxation of the original problem, considering only a small subset of all possible cutting patterns.

Definition 6.2 — RMP model for the 1D-CSP.

$$\min_{y} \quad \sum_{p \in P} y_p \tag{6.2.1}$$

$$\text{s.t.} \quad \sum_{p \in P} a_{ip} y_p \geq d_i \qquad \forall i \in I \quad (\text{Dual} : \pi_i) \tag{6.2.2}$$

$$y_p \in \mathbb{R}_{\geq 0} \qquad \forall p \in P \tag{6.2.3}$$

where:

I	set of item types
P	set of cutting patterns (columns)
y_p	decision variables (number of times pattern p is used)
a_{ip}	number of items of type i in pattern p
d_i	demand for items of type i
π_i	dual prices for demand constraints

The RMP provides a lower bound on the optimal number of stock rolls and guides the generation of new patterns. The most critical output of the RMP are the dual variables π_i, which are utilized in the pricing subproblem. They represent how much the total number of stock rolls would change if the demand for item i increased by one unit.

Step 3. The Pricing Subproblem

The goal of the Pricing Subproblem is to identify a new column (cutting pattern) that could reduce the total number of patterns used. We look for a pattern that maximizes the value of the items included, where the "value" of each item is its dual price π_i from the RMP. This is modeled as a Knapsack Problem (Chapter 2).

Definition 6.3 — Pricing subproblem for the 1D-CSP.

$$\max_{x} \quad \sum_{i \in I} \pi_i x_i \tag{6.3.1}$$

$$\text{s.t.} \quad \sum_{i \in I} l_i x_i \leq L \tag{6.3.2}$$

$$x_i \in \mathbb{Z}_{\geq 0} \qquad \forall i \in I \tag{6.3.3}$$

where:

- I set of item types
- x_i decision variables (number of items of type i in the new pattern)
- π_i dual prices from the RMP
- l_i length of items of type i
- L length of the stock piece

Step 4. Check for new columns (Reduced Cost)

After solving the pricing subproblem, we check the reduced cost of the new pattern. If the reduced cost is negative, it means that adding this pattern to the RMP can potentially reduce the total number of stock pieces used. We then add this new pattern to the RMP and re-optimize. This process is repeated until no more patterns with negative reduced cost can be found, indicating that we have reached an optimal solution.

The reduced cost of a new pattern j can be calculated using the dual prices from the RMP and the original cost of using one more stock piece (which is typically 1 in the cutting stock problem). The formula for the reduced cost is:

$$\overline{c_j} = 1 - \sum_{i \in I} \pi_i x_i \tag{6.4}$$

6.2.3.1 Solving CG for 1D-CSP in Julia

Listing 6.3 presents the Julia code to solve the 1D-CSP using the Column Generation technique. The code is structured into functions that correspond to the steps of the CG algorithm: creating initial columns (Listing 6.3-45), solving the master problem (Listing 6.3-71), solving the pricing subproblem (Listing 6.3-106), and reporting the final solution (Listing 6.3-18). The main function `solve_1d_cutting_stock` (Listing 6.3-132) orchestrates the entire process, starting with a set of initial columns and iteratively adding new patterns until no more improving patterns can be found. Finally, it reports the solution and plots the results.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Random # For random number generation
using Printf # For formatted printing

# Include utility functions for drawing solutions
include("utils/cg_1dcut_utils.jl")

# Instance data structure
struct Instance
    L::Int               # Length of the panel
    lengths::Vector{Int} # Lengths of the items
    demands::Vector{Int} # Demands for each item
    n::Int               # Number of item types
    Instance(L, l, d) = new(L, l, d, length(l)) # Contructor
end

# Function to display the final solution
function report_solution(instance::Instance, columns, solution)
    # Columns sorted by quantity produced in descending order
    sorted_patterns = sortperm(collect(solution), rev=true)
    # Extract the amount cutting patterns
    println("\nCutting patterns:")
    println("="^28)
    @printf("%-20s | %-9s\n", "Pattern (Qty Items)", "Total")
    println("-"^28)
    produced = zeros(Int, instance.n)
    for p in sorted_patterns
        val = solution[p]
        if val > 0
            @printf("%-20s | %.0f\n", string(columns[p]), val)
            produced .+= columns[p] .* round(Int, val)
        end
    end
    # Print the production summary
    println("\nProduction summary:")
    println("="^43)
    @printf("%-4s | %-5s | %-8s | %-8s\n", "Type", "Demand",
"Production", "Overproduction")
    println("-"^43)
    for i in 1:instance.n
```

```
        @printf("T%-3d | %-6d | %-10d | %-8d\n", i,
instance.demands[i], produced[i], produced[i] -
instance.demands[i])
    end
end

# Function to create initial columns (cutting patterns)
function create_initial_columns(instance::Instance,
initial_columns)
    # Basic patterns (one item type by panel)
    columns = [ [i == j ? floor(Int, instance.L /
instance.lengths[i]) : 0 for i in 1:instance.n] for j in
1:instance.n ]
    # Add random cutting patterns
    for _ in 1:initial_columns
        new_column = zeros(Int, instance.n)
        # Loop to ensure we don't create a pattern with zero items
        while iszero(new_column)
            # Start with a full panel length
            new_column_length = instance.L
            # Randomly shuffle to create diverse patterns
            for i in shuffle(1:instance.n)
                # Select a random quantity that fits
                qty = rand(0:floor(Int, new_column_length /
instance.lengths[i]))
                new_column[i] = qty
                # Update the remaining length of the column
                new_column_length -= qty * instance.lengths[i]
            end
        end
        push!(columns, new_column)
    end
    # Ensure columns are unique
    return unique(columns)
end

# Function to solve the master problem (RMP)
function solve_master_problem(instance::Instance, columns;
integer=false)
    # Range of items and patterns
    I = 1:instance.n
    C = 1:length(columns)
```

```
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Create the decision variables
    if integer
        @variable(model, y[c in C] >= 0, Int)
    else
        @variable(model, y[c in C] >= 0)
    end
    # Minimize the total number of panels used
    @objective(model, Min, sum(y[c] for c in C))
    # For each item i, produce at least the required demand
    @constraint(model, demand_constraints[i in I], sum(columns[c][i] * y[c] for c in C) >= instance.demands[i])
    # Solve the model
    JuMP.optimize!(model)
    # Check the termination status
    if JuMP.termination_status(model) != JuMP.OPTIMAL
        error("RMP did not solve to optimality.")
    end
    # Return results
    if integer
        # Return integer solution (no duals for integer problems)
        return nothing, round.(Int, JuMP.value.(y))
    else
        # Return dual prices and linear programming solution
        return JuMP.dual.(demand_constraints), JuMP.objective_value(model)
    end
end

# Function to solve the pricing subproblem (knapsack problem)
function solve_subproblem(instance::Instance, π)
    # Range of items
    I = 1:instance.n
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Create the decision variables
```

```
115     @variable(model, x[i in I] >= 0, Int)
116     # Maximize the total "value" of the pattern
117     @objective(model, Max, sum(π[i] * x[i] for i in I))
118     # The number of items of type i cannot exceed the panel length
119     @constraint(model, sum(instance.lengths[i] * x[i] for i in I)
    <= instance.L)
120     # Solve the model
121     JuMP.optimize!(model)
122     # Check the termination status
123     if JuMP.termination_status(model) != JuMP.OPTIMAL
124         error("Pricing subproblem was not solved to optimality.")
125     end
126     # Calculate the reduced cost
127     reduced_cost = 1 - JuMP.objective_value(model)
128     # Return the reduced cost and the new column
129     return reduced_cost, round.(Int, value.(x))
130 end
131 
132 # Main function to execute the column generation process
133 function solve_cg_1dcsp(instance::Instance; initial_columns = 20,
    iteration_max=100, tolerance=-1e-6)
134     # 1. Create initial columns
135     columns = create_initial_columns(instance, initial_columns)
136     # Reduced cost history for plotting
137     reduced_cost_history = Float64[]
138     println("Initiating Column Generation...")
139     # Main column generation loop
140     for iteration in 1:iteration_max
141         # 2. Solve the Restricted Master Problem
142         π, lp_objective = solve_master_problem(instance, columns)
143         # 3. Solve the Pricing Subproblem
144         reduced_cost, new_column = solve_subproblem(instance, π)
145         # Store reduced cost history
146         push!(reduced_cost_history, reduced_cost)
147         # Print iteration details
148         @printf("Iteration %d: LP objective = %.2f | Reduced Cost =
    %.4f\n", iteration, lp_objective, reduced_cost)
149         # 4. Check if we found a new column
150         if reduced_cost < tolerance
151             push!(columns, new_column)
152         else
```

```
            println("Column Generation complete: no improving column found.")
            break
        end
    end
    # 5. Solve the final integer master problem
    _, ip_solution = solve_master_problem(instance, columns, integer=true)
    # Report the final solution
    report_solution(instance, columns, ip_solution)
    # Plot the reduced cost history
    plot_reduced_costs(reduced_cost_history)
    # Plot the final solution
    plot_solution(columns, ip_solution)
end

# Set the random seed for reproducibility
Random.seed!(42)
# Create an instance of the cutting stock problem and solve it
instance = Instance(100, [45, 30, 20, 15, 5], [60, 88, 35, 110, 200])
# Call the main function to solve the problem
solve_cg_1dcsp(instance, initial_columns=20, iteration_max=100, tolerance=-1e-6)
```

Listing 6.3: Solving 1D-CSP using CG in Julia.

Listing 6.4 presents the output of the Column Generation algorithm for the 1D-CSP example. The output includes the number of iterations, the LP objective value, and the reduced cost at each iteration. It also lists the cutting patterns found and their quantities, as well as a production summary for each item type. The results show that the CG method successfully found an optimal solution that meets the demand for all item types with minimal waste.

Output

```
Initiating Column Generation...
Iteration 1: LP objective = 87.76 | Reduced Cost = -0.0769
Iteration 2: LP objective = 87.07 | Reduced Cost = -0.0417
Iteration 3: LP objective = 86.90 | Reduced Cost = 0.0000
Column Generation complete: no improving column found.
```

```
Cutting patterns:
============================
Pattern (Qty Items)  | Total
----------------------------
[2, 0, 0, 0, 2]      | 30
[0, 2, 0, 1, 5]      | 20
[0, 3, 0, 0, 2]      | 16
[0, 0, 2, 4, 0]      | 15
[0, 0, 0, 6, 2]      | 5
[0, 0, 5, 0, 0]      | 1

Production summary:
=========================================
Type | Demand | Production | Overproduction
-----------------------------------------
T1   | 60     | 60         | 0
T2   | 88     | 88         | 0
T3   | 35     | 35         | 0
T4   | 110    | 110        | 0
T5   | 200    | 202        | 2
```

Listing 6.4: Output of the CG for 1D-CSP.

Figure 6.5 presents a figure that illustrates the convergence of the subproblem for the 1D-CSP. The plot shows how the reduced cost evolves over iterations, demonstrating the effectiveness of the column generation method in finding improving patterns until no more can be found.

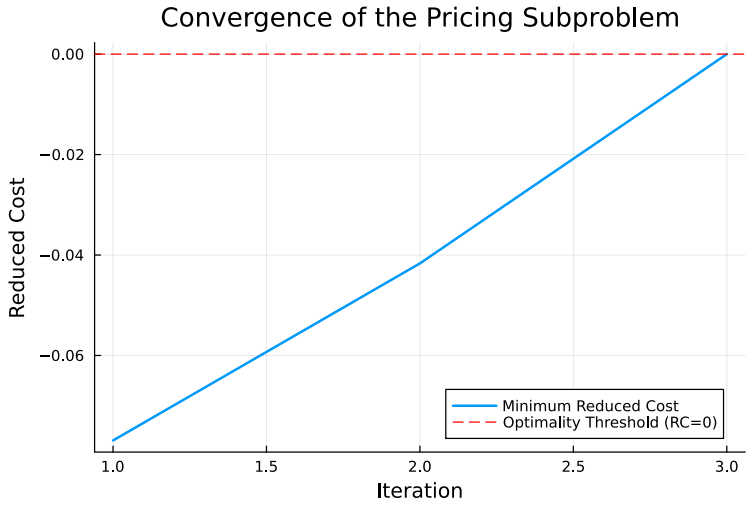


Figure 6.5: Convergence of the Pricing Subproblem for 1D-CSP.

Figure 6.6 presents a figure that visually represents the cutting patterns used in the solution of the 1D-CSP obtained through the Column Generation method. The figure provides an intuitive understanding of how the stock pieces are being utilized to meet the demand for each item type, and it can be compared to the solution obtained from the KCS model (Figure 6.4) to highlight the efficiency of the CG approach.

		Cutting Patterns						
		1	2	3	4	5	6	
Item	T1	2	0	0	0	0	0	
	T2	0	3	0	0	0	1	
	T3	0	0	0	4	3	0	
	T4	0	0	6	0	1	1	
	T5	2	2	2	4	4	11	
Total		30	29	18	8	1	1	87

Figure 6.6: Solution of the CG for 1D-CSP.

6.3 The P-Median Problem

Let us now examine the column generation process for a facility location problem, the P-Median Problem (PMP). This problem has already been dealt with in this text in Section 3.1.

The PMP is a facility location-allocation problem where the goal is to choose p facility locations (medians) out of a set of possible locations to minimize the total distance (or cost) between demand points and their nearest open facility.

Solving the full IP model for the PMP becomes intractable for large instances. Column generation is a technique that can be used in set-partitioning-style reformulations of the PMP to solve its LP relaxation efficiently.

The articles [23], [24] examined the column generation for this problem. Instead of modeling each client-facility assignment directly, the CG consider columns (patterns) that represents a feasible assignment of clients to a single facility.

In the next subsection we describe how to adapt the steps of the column generation technique for the PMP (Figure 6.1).

6.3.1 Column Generation for the PMP

To apply Column Generation, we reformulate the p-median problem as a **Set Covering Problem**. Instead of deciding individual client-to-facility assignments, we consider a "column" as a potential facility and the specific subset of clients it serves.

The following subsections present the core components of the column generation framework used to solve the problem.

Step 1. Create initial columns

The initial columns can be generated using simple heuristics, such as assigning each client to its nearest facility (greedy assignment) or randomly generating feasible assignments of clients to facilities. The key is to ensure that the initial set of columns provides a feasible solution to the problem, allowing us to start the column generation process.

Step 2. The Restricted Master Problem (RMP)

In the traditional model, we decide which points are centers and which customers are served by them. In Column Generation, we look at the problem from the perspective of cluster configurations. Each column represents a potential cluster: a median and the set of customers it serves.

Equation (6.5) presents the mathematical formulation of the Restricted Master Problem for the P-Median Problem. The objective is to minimize the total cost of the selected clusters (columns) while ensuring that each client is covered by at least one cluster and that exactly p clusters are selected (Equation (6.5.1)). The decision variables y_s indicate whether a particular cluster s is selected, and the parameters a_{is} indicate whether client i is served by cluster s. The dual variables π_i and σ are associated with the constraints and will be used in the pricing subproblem to generate new columns. Equation (6.5.2) ensures that each client is covered by at least one selected cluster, while Equation (6.5.3) ensures that exactly p clusters are selected.

Definition 6.4 — RMP model for the PMP.

$$\min_{y} \quad \sum_{s \in S} c_s y_s \tag{6.5.1}$$

$$\text{s.t.} \quad \sum_{s \in S} a_{is} y_s \geq 1 \quad \forall i \in I \quad (\text{Dual} : \pi_i) \tag{6.5.2}$$

$$\sum_{s \in S} y_s = p \quad (\text{Dual} : \sigma) \tag{6.5.3}$$

$$0 \leq y_s \leq 1 \quad \forall s \in S \tag{6.5.4}$$

where:

I	set of clients
S	set of columns
c_s	column cost (sum of distances from median to all clients in cluster)
y_s	decision variables (if column s is selected)
a_{is}	binary parameter indicating if client i is served by column s
p	number of facilities to open
π_i	dual prices for client coverage constraints
σ	dual prices for the number of facilities constraint

The most critical outputs of the RMP are the dual variables, which are utilized in the pricing subproblem:

- π_i: represents the marginal value of client i. A high value indicates that the client is currently expensive to serve, suggesting that the subproblem should identify a new column to cover the client more efficiently.
- σ: represents the opportunity cost of utilizing one of the limited p slots. A high value of σ signifies that the constraint to select exactly p facilities is substantially influencing the total cost.

Step 3. The Pricing Subproblem

The pricing subproblem seeks to identify a new column (a facility j and its associated clients) with the most negative reduced cost. The subproblem can be decomposed into independent problems for each potential facility location, where we determine the optimal set of clients to serve from that location based on the dual prices obtained from the RMP. For a fixed facility $j \in J$, the formulation is defined in Equation (6.6).

Definition 6.5 — Pricing subproblem for the PMP.

$$\min_{x} \quad \sum_{i \in I} (d_{ij} - \pi_i) x_i - \sigma \tag{6.6.1}$$

$$\text{s.t.} \quad x_i \in \{0, 1\} \qquad \forall i \in I \tag{6.6.2}$$

where:

- I set of clients
- d_{ij} distance from client i to facility j
- π_i dual prices from demand constraint of the RMP (benefit of serving client i)
- x_i binary decision variable indicating if client i is served by facility j
- σ dual price for the number of facilities constraint of the RMP (cost of opening a new facility)

Unlike other problems where the subproblem is a Knapsack or a complex Shortest Path, in the P-Median the subproblem is solved trivially by inspection:

1. For a fixed median j, you want $(d_{ij} - \pi_i)x_i$ to be as small as possible;
2. Since x_i is binary, you will set it to 1 only if the term in parentheses is negative;
3. If $(d_{ij} - \pi_i) > 0$, then it is not worth including that customer in the cluster, as it increases the reduced cost ($x_i = 0$).

Therefore, the contribution of each customer i to median j is:

$$v_i = \min(0, d_{ij} - \pi_i) \tag{6.7}$$

The final reduced cost for location j is:

$$\bar{c_j} = \left(\sum_{i \in I} v_i \right) - \sigma \tag{6.8}$$

> **Note**
>
> The dual variable π_i acts like a "price" that the customer is willing to pay to be covered. If the transportation cost d_{ij} is lower than the market price π_i, the connection is advantageous for the subproblem.

Algorithm to solve the Subproblem

Algorithm 6.1 presents the algorithmic steps to solve the pricing subproblem for the P-Median Problem. The algorithm iterates over all candidate locations,

computes the reduced cost for each location, and identifies the location with the smallest reduced cost. If the smallest reduced cost is negative, a new column is generated and added to the RMP; otherwise, the algorithm terminates as no improving columns can be found.

Subproblem heuristic

1 For each $j \in J$, compute the reduced cost $\bar{c}_j$
2 Identify the j^* with the smallest $\bar{c}_j$
3 **if** $\overline{c_{j^*}} < 0$ **then**
4 The binary vector a_{is} that represents the new column will have 1 for all customers where $d_{ij^*} - \pi_i < 0$ and 0 otherwise
5 The column cost will be the sum of the distances from the median to the customers in the cluster
6 Add this column to the RMP
7 **else**
8 There are no more columns that improve the solution, and the linear relaxation is solved.
9 **end**

Algorithm 6.1: Subproblem heuristic for the P-Median Problem.

6.3.2 Solving CG for the PMP in Julia

This subsection will consider the following example of the PMP.

Example 6.2 Given a set of points, a distance matrix, and a number of medians to select ($p = 3$), solve the PMP using Column Generation. Consider the instance from Figure 6.7.



Figure 6.7: PMP instance.

Listing 6.5 presents the Julia code to solve the PMP using the Column Generation technique. The code is structured into functions that correspond to the steps of the CG algorithm: creating initial columns (Listing 6.5-48), solving the master problem (Listing 6.5-69), solving the pricing subproblem (Listing 6.5-109), and reporting the final solution (Listing 6.5-37). The main function `solve_cg_pmp` (Listing 6.5-142) orchestrates the entire process, starting with a set of initial columns and iteratively adding new patterns until no more improving patterns can be found. Finally, it reports the solution and plots the results in a map using a function defined in `utils/cg_pmp_utils.jl` (Listing 6.5-8).

```julia
using JuMP       # Modeling language
using HiGHS      # Solver
using CSV        # for reading CSV files
using Distances  # for distance calculations
using Random     # for random number generation
using Printf     # For formatted printing

# Include utility functions for plotting solutions
include("utils/cg_pmp_utils.jl")

# Data structure to hold problem data
struct Instance
    p::Int64                          # Number of medians to select
    coordinates::Matrix{Float64}      # Coordinates of the clients
    distance_matrix::Matrix{Float64} # Distance matrix between clients
    n::Int64                          # Number of clients
    # Constructor
    Instance(p, coordinates, distance_matrix) = new(p, coordinates, distance_matrix, size(coordinates, 1))
end

# Define a structure to hold column information
struct Column
    cost::Float64   # Cost of the column (sum of distances)
    median::Int64   # Index of the median (facility) in this column
    client_assignments::BitVector
end

# Function to read coordinates and compute Haversine distance matrix
```

```
function read_data(file_path, p)
    # Read the CSV file
    coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)
    # Compute Haversine distance matrix
    distance_matrix = Distances.pairwise(Distances.Haversine(), coordinates, dims=1)
    return Instance(p, coordinates, distance_matrix)
end

# Function to display the final solution
function report_solution(instance::Instance, columns, solution)
    # Extract selected columns based on integer solution
    selected_columns = findall(collect(solution) .> 0.5)

    # Display the final solution
    println("\n--- FINAL P-MEDIAN SOLUTION ---")
    println("Selected facilities (p=$(instance.p)): ", [columns[c].median for c in selected_columns])
    println("Objective value: ", sum(columns[c].cost for c in selected_columns), " meters")
end

# Function to create initial columns (clusters)
function create_initial_columns(instance::Instance, initial_columns)
    columns = Column[]
    for _ in 1:initial_columns
        # Ensure we don't get an empty cluster
        client_assignments = falses(instance.n)
        while count(client_assignments) == 0
            client_assignments .= rand(Bool, instance.n)
        end
        # Get indices of assigned clients
        assigned_indices = findall(client_assignments)
        # Calculate sum of distances between clients and median
        costs = [sum(instance.distance_matrix[i, assigned_indices]) for i in assigned_indices]
        # Find the point with the minimum cost to be the median
        minimum_cost, idx = findmin(costs)
        median = assigned_indices[idx]
```

```
64          push!(columns, Column(minimum_cost, median,
client_assignments))
65      end
66      return columns
67  end
68
69  # Function to solve the master problem (RMP)
70  function solve_master_problem(instance::Instance, columns;
integer=false)
71      # Range of items and patterns
72      I = 1:instance.n
73      C = 1:length(columns)
74      # Create the model
75      model = JuMP.Model(HiGHS.Optimizer)
76      # Silent mode (solver output is not printed)
77      JuMP.set_silent(model)
78      # Create the decision variables
79      if integer
80          @variable(model, y[c in C] >= 0, Bin)
81      else
82          @variable(model, y[c in C] >= 0)
83      end
84      # Minimize the total cost of selected columns
85      @objective(model, Min, sum(y[c] * columns[c].cost for c in C))
86      if integer
87          # For integer solutions, we want exactly one column to
serve each client
88          @constraint(model, coverage_constraints[i in I],
sum(columns[c].client_assignments[i] * y[c] for c in C) == 1)
89      else
90          # For LP relaxation, we allow at least one column to serve
each client
91          @constraint(model, coverage_constraints[i in I],
sum(columns[c].client_assignments[i] * y[c] for c in C) >= 1)
92      end
93      # Exactly 'p' columns must be chosen
94      @constraint(model, p_facilities_constraints, sum(y) ==
instance.p)
95      # Solve the model
96      JuMP.optimize!(model)
97      # Check the termination status
98      if JuMP.termination_status(model) != JuMP.OPTIMAL
```

```
99          error("RMP did not solve to optimality.")
100     end
101     # Return results
102     if integer
103         return nothing, nothing, JuMP.value.(y)
104     else
105         return JuMP.dual.(coverage_constraints),
JuMP.dual(p_facilities_constraints), JuMP.objective_value(model)
106     end
107 end
108 
109 # Function to solve the pricing subproblem (heuristic)
110 function solve_subproblem(instance::Instance, π, σ)
111     I = J = 1:instance.n
112     best_reduced_cost = Inf
113     best_column = nothing
114     # Enumerate all potential medians j
115     for j in J
116         client_assignments = falses(instance.n)
117         cost = 0.0
118         rc = -σ
119         # Assign the median, update reduced cost
120         client_assignments[j] = true
121         rc -= π[j]
122         # Greedily assign client i if it reduces the reduced cost
123         for i in I
124             if i != j
125                 marginal = instance.distance_matrix[i, j] - π[i]
126                 if marginal < 0
127                     client_assignments[i] = true
128                     cost += instance.distance_matrix[i, j]
129                     rc += marginal
130                 end
131             end
132         end
133         # Track the column with the most negative reduced cost
134         if rc < best_reduced_cost
135             best_reduced_cost = rc
136             best_column = Column(cost, j, client_assignments)
137         end
138     end
```

```
    return best_reduced_cost, best_column
end

# Main function to execute the column generation process
function solve_cg_pmp(instance::Instance; initial_columns = 20, iteration_max=100, tolerance=-1e-6)
    # 1. Create initial columns
    columns = create_initial_columns(instance, initial_columns)
    # Reduced cost history for plotting
    reduced_cost_history = Float64[]
    println("Initiating Column Generation...")
    # Main column generation loop
    for iteration in 1:iteration_max
        # 2. Solve the Restricted Master Problem
        π, σ, lp_objective = solve_master_problem(instance, columns)
        # 3. Solve the Pricing Subproblem
        reduced_cost, new_column = solve_subproblem(instance, π, σ)
        # Store reduced cost history
        push!(reduced_cost_history, reduced_cost)
        # Print iteration details
        if iteration % 100 == 0
            @printf("Iteration %d: LP objective = %.2f | Reduced Cost = %.6f\n", iteration, lp_objective, reduced_cost)
        end
        # 4. Check if we found a new column
        if reduced_cost < tolerance
            push!(columns, new_column)
        else
            println("Column Generation complete: no improving column found.")
            break
        end
    end

    # 5. Solve the final integer master problem
    _, _, ip_solution = solve_master_problem(instance, columns, integer=true)

    # Report the final solution
    report_solution(instance, columns, ip_solution)

```

```
    # Plot the reduced cost history
    plot_reduced_costs(reduced_cost_history)

    # Plot a map of the final solution
    plot_solution(instance, columns, ip_solution)
end

# Set the random seed for reproducibility
Random.seed!(42)
# Load the instance data
instance = read_data("data/cg_pmp_coordinates.csv", 3)
# Call the main function to solve the problem
solve_cg_pmp(instance, initial_columns = 500, iteration_max=1000,
tolerance=-1e-6)
```

Listing 6.5: Solving PMP using CG in Julia.

Listing 6.6 presents the output of the Column Generation algorithm for the PMP example. The output includes the number of iterations, the LP objective value, and the reduced cost at each iteration. It also lists the selected facilities (medians) and the objective value of the final solution. The results show that the CG method successfully found an optimal solution that minimizes the total distance between clients and their assigned medians while respecting the constraint on the number of medians to be selected.

Output

```
Initiating Column Generation...
Iteration 100: LP objective = 5239.21 | Reduced Cost = -254.896396
Iteration 200: LP objective = 4774.60 | Reduced Cost = -290.644111
Iteration 300: LP objective = 4774.60 | Reduced Cost = -68.233999
Iteration 400: LP objective = 4774.60 | Reduced Cost = -126.648442
Iteration 500: LP objective = 4774.60 | Reduced Cost = -41.129886
Iteration 600: LP objective = 4774.60 | Reduced Cost = -58.985283
Iteration 700: LP objective = 4774.60 | Reduced Cost = -15.618303
Column Generation complete: no improving column found.

--- FINAL P-MEDIAN SOLUTION ---
Selected facilities (p=3): [26, 8, 18]
Objective value: 4774.5971343036135 meters
```

Listing 6.6: Output of the CG for PMP.

Figure 6.8 presents a figure that illustrates the convergence of the subproblem for the PMP. The plot shows how the reduced cost evolves over iterations, demonstrating the effectiveness of the column generation method in finding improving columns until no more can be found.

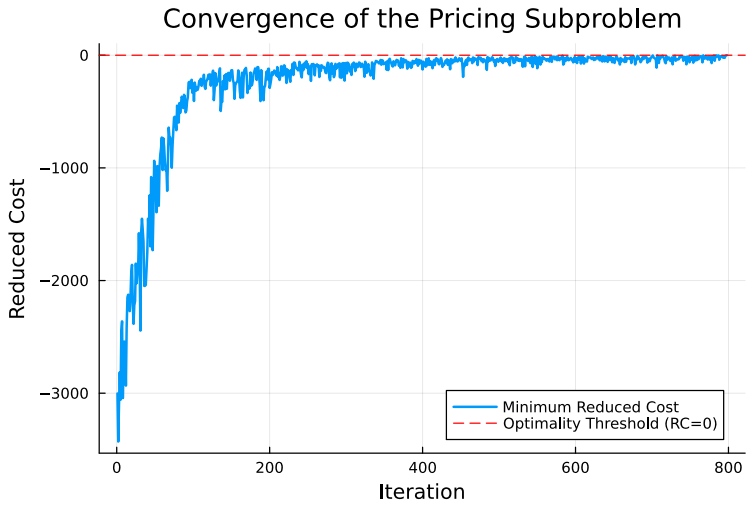


Figure 6.8: Convergence of the Pricing Subproblem for PMP.

Figure 6.9 presents a figure that visually represents the solution of the PMP obtained through the Column Generation method. The figure provides an intuitive understanding of the selected medians and their assigned clients.



Figure 6.9: Solution of the CG for PMP.

6.4 Final remarks

Column generation is an important technique used to solve integer programming problems. One extension of CG is its application within the framework of a

branch-and-bound search tree. This approach, known as Branch-and-Price or Branch-and-Cut-and-Price, combines the power of CG with the exploration of a branch-and-bound algorithm to solve integer programming problems.

In this approach, the branch-and-bound algorithm is used to explore the search tree, branching on variables and generating sub-problems. At each node of the search tree, the CG technique is applied to generate columns (variables) that improve the objective function value and contribute to the relaxation of the problem. These columns are then added to the LP relaxation of the subproblem, allowing for better solution quality and tighter bounds.

In the exercises of this chapter, you will have the opportunity to explore and apply CG to solve specific combinatorial optimization problems.

6.5 Exercises

Formulate and solve the following exercises using the Jump package.

Exercise 6.1 The work [23] presents the column generation for the Capacitated P-Median Problem.

a) Adapt the column generation code for CPMP, and run it for the dataset sjc_n=30_p=7_C=25.csv using $p = 7$ and $C = 25$.

b) Based on the columns generated in item (a), solve the Restricted Master Integer Problem (RMIP) by imposing integrality constraints on the master problem variables to obtain an integer solution.

c) Solve the original integer formulation of the Capacitated P-Median Problem and compare the results obtained in items (a) and (b). Discuss:

- The gap between the linear relaxation and the optimal integer solution;
- The gap between the RMIP solution and the optimal integer solution;
- The quality of the generated column set.

Exercise 6.2 The work [77] presents a Column Generation approach to the Clique Partitioning Problem in Complex Networks.

a) Implement the column generation process described by the authors, and test it in the dataset karate.csv.

b) Based on the columns generated in item (a), solve the Restricted Master Integer Problem (RMIP) by imposing integrality constraints on the master problem variables to obtain an integer solution.

c) Solve the original integer formulation of the Clique Partitioning Problem and compare the results obtained in items (a) and (b). Discuss:
   - The gap between the linear relaxation and the optimal integer solution;
   - The gap between the RMIP solution and the optimal integer solution;
   - The quality of the generated column set.

Exercise 6.3 The work [78] presents a column generation approach to the Graph Coloring Problem (see also the work [79]).

a) Implement the column generation described by the authors, and test it with the following graph: tutte_graph.csv.

b) Based on the columns generated in item (a), solve the Restricted Master Integer Problem (RMIP) by imposing integrality constraints on the master problem variables to obtain an integer solution.

c) Solve the original integer formulation of the Graph Coloring Problem and compare the results obtained in items (a) and (b). Discuss:
   - The gap between the linear relaxation and the optimal integer solution;
   - The gap between the RMIP solution and the optimal integer solution;
   - The quality of the generated column set.

7 Lagrangian Relaxation

Lagrangian Relaxation (LR) is a powerful method used in mathematical optimization to approximate difficult constrained optimization problems with simpler ones. By strategically relaxing "hard" constraints and incorporating them into the objective function through penalties, this technique can efficiently find good approximate solutions and provide valuable bounds on the true optimal value.

Its origins trace back to the 18th-century work of Joseph-Louis Lagrange on multipliers, with the modern version being significantly developed in the 1970s by Held and Karp for solving large-scale traveling salesman problems [80].

7.1 Lagrangian Relaxation algorithm

To explain the LR algorithm, we first need to define the integer programming problem we want to solve. Consider the integer programming problem presented in Equation (7.1).

Definition 7.1 — P model.

$$\min_{x} \quad \sum_{i=1}^{n} c_i x_i \tag{7.1.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} a_{ij} x_i \geq b_j \quad j = 1, ..., m_1 \tag{7.1.2}$$

$$\sum_{i=1}^{n} d_{ij} x_i = e_k \quad k = 1, ..., m_2 \tag{7.1.3}$$

$$x_i \in \mathbb{Z}_{\geq 0} \qquad \forall i = 1, ..., n \tag{7.1.4}$$

where:

c_i	cost coefficient of variable x_i
x_i	decision variable (what you are choosing)
a_{ij}	coefficient linking variable x_i to constraint j
b_j	right-hand side of constraint j
d_{ik}	coefficient linking variable x_i to constraint k
e_k	right-hand side of constraint k

Suppose the equality constraints Equation (7.1.3) are difficult to handle. To apply LR, we relax these constraints and move them into the objective. We introduce a vector of **Lagrange multipliers** λ_k for $k = 1, ..., m_2$, where each λ_k corresponds

to one of the equality constraints. These multipliers act as penalty terms in the objective function for violation of the relaxed constraints, resulting in the new objective:

$$\sum_{i=1}^{n} c_i x_i + \sum_{k=1}^{m_2} \lambda_k \left(e_k - \sum_{i=1}^{n} d_{ik} x_i \right) \tag{7.2}$$

The resulting relaxed optimization problem $L(\lambda)$, called the **Lagrangian relaxation** of the original problem, is defined as:

Definition 7.2 — Lagrangian Relaxation of P.

$$\min_{x} \quad \sum_{i=1}^{n} c_i x_i + \sum_{k=1}^{m_2} \lambda_k \left(e_k - \sum_{i=1}^{n} d_{ik} x_i \right) \tag{7.3.1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} a_{ij} x_i \geq b_j \qquad j = 1, ..., m_1 \tag{7.3.2}$$

$$x_i \in \mathbb{Z}_{\geq 0} \qquad i = 1, ..., n \tag{7.3.3}$$

where:

λ_k vector of Lagrange multipliers

In Equation (7.3), the original constraints Equation (7.1.3) are now relaxed, and the optimization problem is solved over the remaining constraints Equation (7.1.2) and Equation (7.1.4).

Since we obtain a parameterized family of optimization problems $L(\lambda)$, we need to optimize over the choice of λ to find the best possible bound on the original problem. This is done by solving the **Lagrangian dual** problem (D), which seeks to maximize the lower bound provided by the Lagrangian relaxation $z_{L(\lambda)}$ over all possible values of the Lagrange multipliers λ.

Definition 7.3 — Lagrangian Dual of P.

$$\max_{\lambda} \quad z_{L(\lambda)} \tag{7.4.1}$$

$$\text{s.t.} \quad \lambda_k \in \mathbb{R} \quad k = 1, ..., m_2 \tag{7.4.2}$$

where:

$z_{L(\lambda)}$ value of the Lagrangian relaxation for a given λ

λ_k vector of Lagrange multipliers

The dual function z_D is always concave, which means the dual problem can be solved efficiently using methods like the subgradient method.

7.1.1 Subgradient method

The subgradient method is the primary iterative algorithm used to solve the Lagrangian Dual problem D from Equation (7.4). It updates the Lagrange multipliers step by step, moving in the direction of a subgradient to seek the maximum of the dual function z_D.

Algorithm 7.2 summarizes the steps of the subgradient method for solving the Lagrangian dual problem. The algorithm iteratively updates the multipliers based on the subgradient of the dual function, which is computed from the solution of the Lagrangian relaxation at each iteration. The process continues until a stopping criterion is met, such as stagnation in improvement, a small subgradient norm, or reaching a maximum number of iterations.

Subgradient method

1. Set iteration counter $t = 0$ and initialize multipliers $\lambda^{(t)}$.
2. Solve the Lagrangian Relaxation $L(\lambda^{(t)})$ to obtain a lower bound $\text{LB}^{(t)}$.
3. Construct a feasible solution using a heuristic and update the best upper bound $\text{UB}^{(t)}$.
4. Compute the subgradient $s^{(t)}$.
5. Update the multipliers using the subgradient rule and project if necessary (see Note 7.3).
6. Increment t and repeat steps 2-6 until a stopping criterion is met (the improvement stagnates, subgradient norm is small, or max iterations reached).

Algorithm 7.2: Subgradient method.

Note

Projection of Lagrange multipliers

After updating the Lagrange multipliers using the subgradient rule, it is often necessary to project the updated multipliers back onto the feasible set defined by the constraints of the dual problem.

Table 7.1 summarizes the projection step for different types of constraints. For equality constraints, the multipliers can take any real value, so no projection is needed. However, for inequality constraints, the multipliers must be non-negative, which requires a projection step to ensure that the updated multipliers remain valid.

Constraint Type	Multiplier	Update Rule
$\sum_{i=1}^{n} d_{ik}x_i = e_k$	$\lambda_k \in \mathbb{R}$	$\lambda_k^{(t+1)} \leftarrow \lambda_k^{(t)} + \alpha^{(t)} g_k^{(t)}$
$\sum_{i=1}^{n} a_{ij}x_i \geq b_j$	$\lambda_j \geq 0$	$\lambda_j^{(t+1)} \leftarrow \max\left(0, \lambda_j^{(t)} + \alpha^{(t)} g_j^{(t)}\right)$

Table 7.1: Projection of Lagrange multipliers.

The next subsections will delve into details of some components of the subgradient method, including how to compute the subgradient, how to update multipliers and different possibilities of stopping criteria.

How to compute the subgradient (Step 4 of Algorithm 7.2)

After solving the Lagrangian subproblem at iteration t and obtaining the optimal solution $x_i^{(t)}$, a subgradient with respect to each λ_k is given by:

$$s_k^{(t)} = e_k - \sum_{i=1}^{n} d_{ik} x_i^{(t)} \tag{7.5}$$

This value represents the violation of the k-th relaxed constraint under the solution $x^{(t)}$, and it provides a direction for updating the corresponding Lagrange multiplier. The idea is to adjust each penalty λ_k based on the extent to which the corresponding constraint is violated.

How to update the multipliers (Step 5 of Algorithm 7.2)

Unlike optimization methods designed for differentiable functions, the subgradient method is not a descent (or ascent, for maximization) method. This means that each iteration does not necessarily improve the objective function value. In fact, it is possible that $z_{L(\lambda^{(t+1)})} < z_{L(\lambda^{(t)})}$, indicating a decrease in the dual function value. The direction of the negative subgradient may not point toward an improving or optimal solution.

As a result, the choice of step size $\alpha^{(t)}$ plays a critical role in ensuring that the algorithm converges to the optimal solution. Since line search techniques, commonly used in gradient-based methods, are not computationally feasible in most subgradient settings, step sizes are typically determined before the algorithm begins, based on a predefined strategy. Several common strategies include:

1. **Constant Step Size**: $\alpha^{(t)} = \alpha$. In this case, the algorithm may converge to a value near the optimal solution, but not necessarily to the optimum itself.

2. **Constant Step Length**: The step size is chosen so that the Euclidean distance between successive multiplier vectors remains constant. This is achieved by setting: $\alpha^{(t)} = \frac{\gamma}{\|s\|}$, where $\gamma > 0$ is a fixed constant.
3. **Diminishing Step Sizes**: The step sizes decrease with each iteration. Common choices that satisfy these conditions include:

$$\alpha^{(t)} = \frac{\gamma}{t} \quad \text{or} \quad \alpha^{(t)} = \frac{\gamma}{\sqrt{t}}, \quad \text{where } \gamma > 0 \text{ is a constant.} \tag{7.6}$$

4. **Held-Karp procedure**: The Held-Karp procedure is a classic method for choosing the step size in the subgradient method for Lagrangian Relaxation, especially in combinatorial optimization problems. The step size at iteration (t) is given by:

$$\alpha^{(t)} = \theta \left(\frac{\mathrm{UB}^{(t)} - \mathrm{LB}^{(t)}}{\left\| s^{(t)} \right\|^2} \right), \quad \theta \in (0, 2] \tag{7.7}$$

where $\mathrm{UB}^{(t)}$ is the current best upper bound, $\mathrm{LB}^{(t)}$ is the current best lower bound, $s^{(t)}$ is the subgradient, and θ is a parameter to reduce the convergence oscillation.

How to stop the algorithm (Step 6 of Algorithm 7.2)

Due to the non-monotonic nature of the subgradient method, defining an effective stopping criterion can be challenging. Since the objective function value is not guaranteed to improve at every iteration, it is not reliable to terminate the algorithm simply because progress appears to stall. Common approaches for terminating the algorithm include:

1. **Fixed Number of Iterations:** The simplest approach is to run the algorithm for a predetermined number of iterations. This is often used in practice when computational time is limited or when convergence behavior is not well known.
2. **Stalled Improvement (Best Objective Tracking):** The algorithm terminates when the best dual objective value found so far, LB, has not improved significantly over a given number of iterations. Because the method is not monotonic, it is essential to track this best value across all iterations. The algorithm can stop when the improvement in LB falls below a specified tolerance over a certain number of iterations.

3. **Duality Gap (Primal-Dual Convergence):** In many practical applications, a feasible solution to the original primal problem P is heuristically constructed from the solutions of the Lagrangian subproblems $L(\lambda)$. If such a solution is found with objective value UB, the algorithm can be terminated when the duality gap between the best upper bound and the best dual lower bound becomes sufficiently small: $\text{UB} - \text{LB} \leq \varepsilon$, where ε is a predefined tolerance.

7.2 Simple Example of Lagrangian Relaxation

We now illustrate the use of Lagrangian relaxation for the following model.

Definition 7.4 — E model.

$$\min_{x_1,x_2} \quad 7x_1 + 10x_2 \tag{7.8.1}$$

$$\text{s.t.} \quad 3x_1 + 5x_2 \geq 7 \tag{7.8.2}$$

$$x_1, x_2 \in \{0, 1\} \tag{7.8.3}$$

The E model in Equation (7.8) has only one linear constraint (Equation (7.8.2)) in addition to the binary restrictions (Equation (7.8.3)). Figure 7.1 shows a two-dimensional representation of the model, in which point $(1, 1)$ is the optimal solution with an objective function value of $z_{\text{E}} = 17$.

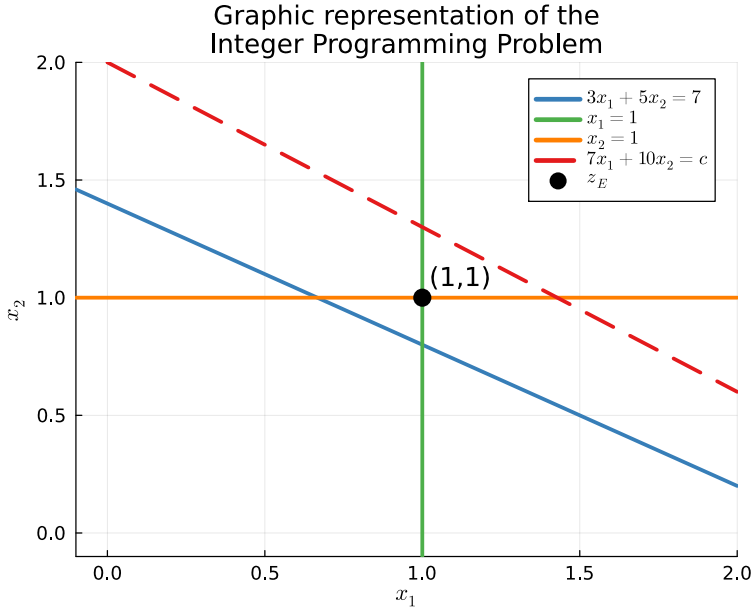


Figure 7.1: Two-dimensional representation of the feasible set of model E.

Lagrangian Relaxation of the E model

Let us now consider the Lagrangian Relaxation $L(\lambda)$ of model E with respect to the first constraint (Equation (7.8.2)). For a scalar Lagrangian multiplier $\lambda \geq 0$, we obtain the following model:

Definition 7.5 — Lagrangian Relaxation of model E.

$$\min_{x_1,x_2} \quad 7\lambda + (7-3\lambda)x_1 + (10-5\lambda)x_2 \tag{7.9.1}$$

$$\text{s.t.} \quad x_1, x_2 \in \{0,1\} \tag{7.9.2}$$

The Lagrangian Relaxation $L(\lambda)$ is defined for a scalar Lagrangian multiplier λ. Since the complicating constraint has been relaxed, it can be solved efficiently for the binary variables x. The optimal objective value, denoted $z_{L(\lambda)}$, is dependent on the choice of λ.

Lagrangian Dual of the E model

To find the best possible lower bound on the original problem, we need to solve the Lagrangian Dual problem D (Equation (7.10)), which involves maximizing $z_{L(\lambda)}$ over all non-negative values of λ. By evaluating $z_{L(\lambda)}$ for different values of λ, we can identify the optimal multiplier that provides the tightest lower bound on the original problem's optimal value.

Definition 7.6 — Lagrangian Dual of model E.

$$\max_{\lambda} \quad z_{L(\lambda)} \tag{7.10.1}$$

$$\text{s.t.} \quad \lambda \geq 0 \tag{7.10.2}$$

7.2.1 Solving the Lagrangian Dual problem

The solution of the dual problem D involves finding the multiplier $\lambda \geq 0$ that maximizes the lower bound $z_{L(\lambda)}$. Since $L(\lambda)$ is a minimization problem over binary variables, the optimal values of x_1 and x_2 depend on the signs of their respective coefficients:

$$z_{L(\lambda)} = 7\lambda + \underbrace{(7-3\lambda)}_{c_1} x_1 + \underbrace{(10-5\lambda)}_{c_2} x_2 \tag{7.11}$$

Since we are minimizing and $x_1, x_2 \in \{0,1\}$:

$$x_1 = \begin{cases} 1 \text{ if } \ c_1 \leq 0 \\ 0 \text{ otherwise,} \end{cases} \quad \text{and} \quad x_2 = \begin{cases} 1 \text{ if } \ c_2 \leq 0 \\ 0 \text{ otherwise,} \end{cases} \tag{7.12}$$

To find the boundaries, we set the coefficients to zero:

$$c_1 = 0 \quad \Rightarrow \quad 7 - 3\lambda = 0 \quad \Rightarrow \quad \lambda \approx 2.33 \tag{7.13.1}$$

$$c_2 = 0 \quad \Rightarrow \quad 10 - 5\lambda = 0 \quad \Rightarrow \quad \lambda = 2 \tag{7.13.2}$$

These values divide the possible values of λ into the three intervals: $[0, 2)$, $[2, 2.33)$, and $[2.33, \infty)$. Table 7.2 maps these intervals to the corresponding optimal binary decisions and the resulting linear expression for $z_{L(\lambda)}$. At the breakpoints $\lambda = 2$ and $\lambda = 2.33$, the Lagrangian subproblem admits multiple optimal solutions; the half-open interval convention is adopted for notational clarity.

Interval for λ	**Optimal solution (x_1, x_2)**	$z_{L(\lambda)}$
$[0, 2)$	$(0, 0)$	7λ
$[2, 2.33)$	$(0, 1)$	$2\lambda + 10$
$[2.33, \infty)$	$(1, 1)$	$-\lambda + 17$

Table 7.2: Derivation of the Lagrangian dual function segments.

Plotting these segments as a function of λ yields the graph in Figure 7.2. The dual function is the lower envelope of these linear functions, and is therefore concave and piecewise linear. As shown in the figure, the maximum value is the optimal value of the Lagrangian dual problem $z_D = 14.66$, that occurs at the intersection of the second and third segments where $\lambda = 2.33$.

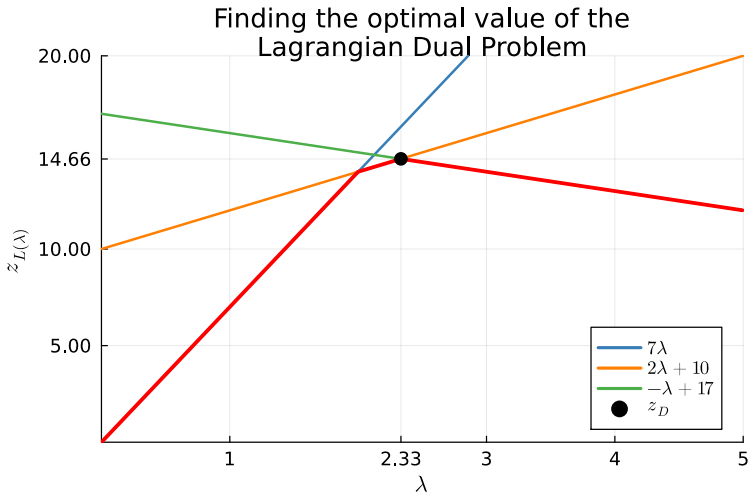


Figure 7.2: Lagrangian Dual Problem solution.

Lagrangian duality gap

The Lagrangian duality gap is defined as the difference between the optimal value of the original problem and the optimal value of the dual problem:

$$\text{Duality Gap} = z_E - z_D \tag{7.14}$$

For our example, the optimal value of the original problem is $z_E = 17$ and the optimal value of the dual problem is $z_D = 14.66$, so the duality gap is $17 - 14.66 = 2.34$. This gap quantifies how much the best lower bound (from the dual) underestimates the true optimal value of the original problem. In this case, the Lagrangian relaxation provides a lower bound that is 2.34 units below the optimal value, indicating that while it gives us useful information about the structure of the problem, it does not yield a tight bound in this instance.

7.2.2 Solving Lagrangian relaxation for PMP using Julia

Listing 7.1 presents the implementation of the Lagrangian relaxation for model E in Julia, using the JuMP package for optimization and HiGHS as the solver. The code defines functions to solve the original problem exactly (Listing 7.1-19), perform the Lagrangian relaxation (Listing 7.1-39), and plot the convergence of the bounds (Listing 7.1-6).

```julia
using JuMP   # For mathematical programming
using HiGHS  # HiGHS solver
using Printf # For formatted output
using Plots  # For plotting

# Function to plot convergence of LB and UB
function plot_convergence(lb_history, ub_history)
    p = plot(1:length(lb_history), [lb_history ub_history],
        label=["Lower Bound (LB)" "Upper Bound (UB)"],
        xlabel="Iteration",
        ylabel="Objective Value",
        title="Lagrangian Relaxation Convergence",
        linewidth=2,
        linestyle=[:dash :solid],
        legend=:bottomright)
    display(p)
end

# Function to solve the original integer program exactly
function solve_exact()
```

```
    # Create the model
    model =  JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define binary variables
    @variable(model, x[1:2], Bin)
    # Define the objective function
    @objective(model, Min, 7 * x[1] + 10 * x[2])
    # Define the constraint
    @constraint(model, 3 * x[1] + 5 * x[2] >= 7)
    # Solve the model
    JuMP.optimize!(model)
    # Get results
    x_opt =  JuMP.value.(x)
    obj_value = JuMP.objective_value(model)
    return x_opt, obj_value
end

# Function to perform Lagrangian relaxation
function solve_lagrangian(;iteration_max = 100,
show_convergence_plot = false)
    # Initialize variables
    x = [0, 0]  # Initial solution
    λ = 0.0     # Initial Lagrange multiplier
    λ_best = λ  # To track the best λ found
    α = 0.1     # Step size
    θ = 0.8     # A factor to update multiplier (Held-Karp)
    LB = -Inf   # Best lower bound found so far
    UB = Inf    # Best upper bound found so far
    LB_history = Float64[] # Store LB for plotting
    UB_history = Float64[] # Store UB for plotting

    # Logic to stop if no improvement in UB
    no_improve_limit = 10
    no_improve_counter = 0

    println("-"^14," Starting Lagrangian Relaxation ", "-"^14)
    println("Iter. |    λ    |  Lower Bound  |   Upper Bound  |
Gap")
    println("-"^60)

```

```
    for iteration in 1:iteration_max
        # Compute primal solution at current λ
        x[1] = (7 - 3 * λ) > 0 ? 1 : 0
        x[2] = (10 - 5 * λ) > 0 ? 1 : 0

        # Lagrangian dual value
        zL = 7 * λ + min(0, 7 - 3 * λ) + min(0, 10 - 5 * λ)

        # Check for improvement in LB
        if zL > LB
            LB = zL
            λ_best = λ              # Update the best λ found
            no_improve_counter = 0  # Reset counter
        else
            no_improve_counter += 1
            # Check if we have reached the limit of no improvement
            if no_improve_counter >= no_improve_limit
                println("-"^60)
                println("Lagrangian relaxation stopped. ",
                        "No improvement in LB for $no_improve_limit
iterations.")
                break
            end
        end

        # Feasible primal value
        if 3 * x[1] + 5 * x[2] >= 7
            zP = 7 * x[1] + 10 * x[2]
            UB = min(UB, zP)
        end

        # Store LB and UB values for plotting
        push!(LB_history, LB)
        push!(UB_history, UB)

        # Print iteration details
        if iteration % 10 == 0 || iteration == 1
            @printf("%5d | %7.4f | %13.4f | %14.4f | %9.3f\n",
iteration, λ, LB, UB, UB - LB)
        end

```

```
99          # Subgradient
100         s = 7 - 3 * x[1] - 5 * x[2]
101
102         # Update step size (Diminishing step size)
103         α = θ * ((UB - LB) / (s^2 + 1e-8))
104
105         # Update λ (project onto λ ≥ 0)
106         λ = max(0, λ - α * s)
107     end
108
109     # Solve exactly to find the optimal value
110     x0, z0 = solve_exact()
111
112     # Print results
113     println("-"^26, " Results ", "-"^25)
114     @printf("Optimal Dual Solution (λ = %4.2f): %4.2f\n", λ_best,
LB)
115     @printf("Optimal Primal Solution: x = [%d, %d] with objective
value %4.2f\n", x0[1], x0[2], z0)
116     @printf("Duality gap: %4.2f - %4.2f = %4.2f\n", z0, LB, z0 -
LB)
117
118     # Plot convergence if requested
119     if show_convergence_plot
120         plot_convergence(LB_history, UB_history)
121     end
122 end
123
124 solve_lagrangian(iteration_max = 100, show_convergence_plot = true)
```

Listing 7.1: Lagrangian Relaxation Julia implementation for model E.

Listing 7.2 presents the output of the Lagrangian relaxation algorithm when applied to model E. The output includes the iteration number, current value of the Lagrange multiplier λ, the best lower bound (LB) found so far, the best upper bound (UB) found so far, and the duality gap (UB - LB). The algorithm stops when there is no improvement in the lower bound for 10 consecutive iterations. Finally, it prints the optimal dual solution, the optimal primal solution, and the duality gap.

Output

```
-------------- Starting Lagrangian Relaxation --------------
Iter. |    λ    |  Lower Bound  |   Upper Bound  |   Gap
------------------------------------------------------------
    1 |  0.0000 |        0.0000 |        17.0000 |    17.000
   10 |  5.1510 |       11.8490 |        17.0000 |     5.151
   20 |  3.4105 |       14.5131 |        17.0000 |     2.487
   30 |  2.6241 |       14.5474 |        17.0000 |     2.453
   40 |  1.6741 |       14.6562 |        17.0000 |     2.344
------------------------------------------------------------
Lagrangian relaxation stopped. No improvement in LB for 10 iterations.
-------------------------- Results -------------------------
Optimal Dual Solution (λ = 2.34): 14.66
Optimal Primal Solution: x = [1, 1] with objective value 17.00
Duality gap: 17.00 - 14.66 = 2.34
```

Listing 7.2: Output of the Lagrangian Relaxation for model E.

Figure 7.3 shows the convergence plot of the lower bound (LB) and upper bound (UB) over iterations.

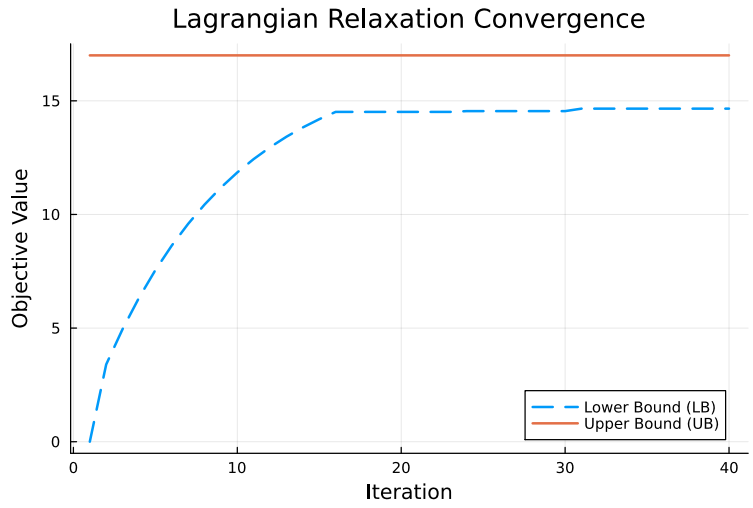


Figure 7.3: Lagrange Relaxation Convergence for model E.

7.3 Lagrangian Relaxation for P-Median Problem

The P-Median Problem (PMP) (Chapter 3) is a fundamental facility location model with numerous applications in logistics, telecommunications, and service system design. Despite its simple formulation, the problem is NP-hard and becomes computationally challenging for large instances.

In this section, we present a Lagrangian Relaxation approach for the PMP, together with a subgradient algorithm for optimizing the multipliers and a simple primal heuristic for obtaining feasible solutions.

7.3.1 The Lagrangian Relaxation model for PMP

Let $N = \{1, 2, 3, ...\}$ denote the set of demand points. For each pair $(i, j) \in N \times N$, let d_{ij} be the distance (or cost) of assigning demand point i to a facility located at point j. The p-median problem seeks to locate exactly p facilities and assign each demand node to one open facility so as to minimize total assignment cost. Equation (7.15) presents the standard node-based formulation for PMP.

Definition 7.7 — PMP model. Given the set of demand points N and the number of facilities to open p.

$$\min_{x} \quad \sum_{i \in N} \sum_{j \in N} d_{ij} x_{ij} \tag{7.15.1}$$

$$\text{s.t.} \quad \sum_{j \in N} x_{jj} = p \tag{7.15.2}$$

$$\sum_{j \in N} x_{ij} = 1 \qquad \forall i \in N \tag{7.15.3}$$

$$x_{ij} \leq x_{jj} \qquad \forall i, j \in N \tag{7.15.4}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in N \tag{7.15.5}$$

The p-median problem is difficult to solve because of the strong coupling between facility location and customer assignment decisions in the assignment constraints in Equation (7.15.3), which require that every demand node i be assigned to exactly one facility j. By relaxing these constraints, the problem becomes separable by facility location, allowing a much simpler structure.

Although other constraints, such as the facility-count constraint in Equation (7.15.2), are often relaxed in the literature, relaxing the assignment constraints leads to a particularly intuitive Lagrangian Relaxation.

We relax the assignment constraint by moving it into the objective function with a penalty vector λ_i (Lagrangian multipliers). The resulting Lagrangian relaxation $L(\lambda)$ is given in Equation (7.16).

Definition 7.8 — Lagrangian Relaxation for PMP model.

$$\min_{x} \quad \sum_{i\in N}\sum_{j\in N}(d_{ij}-\lambda_i)x_{ij}+\sum_{i\in N}\lambda_i \tag{7.16.1}$$

$$\text{s.t.} \quad \sum_{j\in N} x_{jj} = p \tag{7.16.2}$$

$$x_{ij} \le x_{jj} \qquad \forall i,j \in N \tag{7.16.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N \tag{7.16.4}$$

For fixed multipliers (λ), this problem provides a lower bound on the optimal value of the original PMP model in Equation (7.15).

Once the assignment constraints are relaxed, the problem decomposes by facility location. For a given open facility j, the optimal assignment decision for each demand point i is:

$$x_{ij} = \begin{cases} 1 \text{ if } \ d_{ij}-\lambda_i < 0 \\ 0 \text{ otherwise} \end{cases} \tag{7.17}$$

Opening facility j therefore yields a total contribution:

$$c_j = \sum_{i\in N} \min\big(0, d_{ij}-\lambda_i\big) \tag{7.18}$$

The value c_j represents the reduced cost of opening facility j. To satisfy the constraint that exactly p facilities must be opened, we select the p facilities with the smallest (most negative) values of c_j, and the resulting objective value of $z_{\{L(\lambda)\}}$ is a valid Lagrangian lower bound (LB).

The Lagrangian Dual (D) for PMP consists of maximizing the lower bound obtained from $L(\lambda)$ over all multiplier vectors λ.

Definition 7.9 — Lagrangian Dual for PMP model.

$$\max_{\lambda} \quad z_{L(\lambda)} \tag{7.19.1}$$

$$\text{s.t.} \quad \lambda_i \ge 0 \quad \forall i \in N \tag{7.19.2}$$

Since the dual function is concave but generally non-differentiable, it is typically solved using a subgradient method. For a given multiplier vector λ, let $x_{ij}^{(t)}$ denote an optimal solution of $L(\lambda)$ at iteration t. A subgradient component corresponding to demand node i is given by:

$$s_i^{(t)} = 1 - \sum_{j\in N} x_{ij}^{(t)} \quad \forall i \in N \tag{7.20}$$

The multipliers are updated iteratively according to:

$$\lambda_i^{(t+1)} = \lambda_i^{(t)} + \alpha^{(t)} g_i^{(t)} \quad \forall i \in N \tag{7.21}$$

where $\alpha^{(t)} > 0$ is a step size. A commonly used step-size rule is:

$$\alpha^{(t)} = \theta \left(\frac{\mathrm{UB}^{(t)} - \mathrm{LB}^{(t)}}{\left\| s^{(t)} \right\|^2} \right) \tag{7.22}$$

with $\theta \in (0, 2]$ and $\mathrm{UB}^{(t)}$ denoting the best known upper bound (feasible solution) at iteration t. The algorithm terminates when a stopping criterion such as a small duality gap or a maximum number of iterations is reached.

Primal Heuristic and Bounding

To obtain a feasible solution to the original p-median problem, a simple primal heuristic is applied. The p facilities selected in the Lagrangian subproblem are fixed as open, and each demand node is assigned to its nearest open facility. This produces a feasible solution and therefore a valid upper bound on the optimal objective value. Throughout the subgradient iterations, the best lower and upper bounds are tracked, and their difference provides a measure of solution quality.

7.3.1.1 Solving Lagrangian relaxation for PMP using Julia

Listing 7.3 presents a Julia implementation of the Lagrangian relaxation for the PMP. The code includes functions to load data (Listing 7.3-31), solve the original problem exactly (Listing 7.3-40), perform the Lagrangian relaxation with a subgradient method (Listing 7.3-68), and plot the convergence of bounds (Listing 7.3-18).

```julia
using JuMP          # For mathematical programming
using HiGHS         # HiGHS solver
using CSV           # For reading CSV files
using Distances     # Distance computations
using LinearAlgebra # For norm calculations
using Printf        # For formatted output
using Plots         # For plotting

struct Instance
    p::Int64
    coordinates::Matrix{Float64}
    distance_matrix::Matrix{Float64}
    n::Int64
    # Constructor
```

```
    Instance(p, coordinates, distance_matrix) = new(p, coordinates, distance_matrix, size(coordinates, 1))
end

# Function to plot convergence of LB and UB
function plot_convergence(lb_history, ub_history)
    p = plot(1:length(lb_history), [lb_history ub_history],
        label=["Lower Bound (LB)" "Upper Bound (UB)"],
        xlabel="Iteration",
        ylabel="Objective Value",
        title="Lagrangian Relaxation Convergence for PMP",
        linewidth=2,
        linestyle=[:dash :solid],
        legend=:bottomright)
    display(p)
end

# Function to load data from CSV and compute distance matrix
function load_data(file_path, p)
    # Load latitude and longitude data
    coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)
    # Compute Haversine distance matrix
    distance_matrix = Distances.pairwise(Distances.Haversine(), coordinates, dims=1)
    return Instance(p, coordinates, distance_matrix)
end

# Function to solve the PMP exactly
function solve_pmp_exact(instance::Instance)
    # Range for points
    N = 1:instance.n
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[i in N, j in N], Bin)
    # Objective: Minimize total distance
    @objective(model, Min, sum(instance.distance_matrix[i,j] * x[i,j] for i in N, j in N))
```

```
    # Constraint: Exactly p facilities opened
    @constraint(model, sum(x[j,j] for j in N) == instance.p)
    # Constraint: Each customer assigned to one facility
    @constraint(model, [i in N], sum(x[i,j] for j in N) == 1)
    # Constraint: Assignment only to open facilities
    @constraint(model, [i in N, j in N], x[i,j] <= x[j,j])
    # Run the solver
    JuMP.optimize!(model)
    # Return the optimal value
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        return JuMP.objective_value(model)
    else
        error("Optimal solution not found.")
    end
end

# Main function to solve PMP using Lagrangian Relaxation
function solve_pmp_lagrangian(file_path; p = 3, iteration_max =
100, gap_tolerance = 1e-6, show_convergence_plot = false)

    # Load instance
    instance = load_data(file_path, p)

    # Auxiliary variables
    n = instance.n
    N = 1:n
    D = instance.distance_matrix

    # Algorithm Initialization
    λ = zeros(n) # Initial Lagrange multipliers
    θ = 2.0      # A factor to update multiplier (Held-Karp)
    LB = -Inf    # Best lower bound found so far
    UB = Inf     # Best upper bound found so far
    LB_history = Float64[] # Store LB for plotting
    UB_history = Float64[] # Store UB for plotting

    # Logic to reduce θ if no improvement in LB
    no_improve_limit = 10
    no_improve_counter = 0

    println("-"^11," Starting Lagrangian Relaxation ", "-"^11)
```

```
92      println("Iter.  |   Lower Bound   |   Upper Bound   |   Gap
(%)")
93      println("-"^54)
94
95      for iteration in 1:iteration_max
96          #############################################
97          # 1. Solve the Lagrangian Relaxation L(λ) #
98
99          # 1a. Calculate attractiveness of each facility
100         c = [sum(min(0, D[i, j] - λ[i]) for i in N) for j in N]
101
102         # 1b. Select the p best facilities (smallest value)
103         best_facilities = sortperm(c)[1:p]
104
105         # 1c. Calculate the z(L(λ))
106         current_LB = sum(c[best_facilities]) + sum(λ)
107
108         # 1d. Update the Best Lower Bound (LB)
109         if current_LB > LB
110             LB = current_LB
111             no_improve_counter = 0
112         else
113             no_improve_counter += 1
114             # Reduce θ if progress stalls
115             if no_improve_counter >= no_improve_limit
116                 # Reset counter
117                 no_improve_counter = 0
118                 # Reduce θ to allow for finer adjustments
119                 θ = max(0.5, θ * 0.5)
120             end
121         end
122         push!(LB_history, LB)
123
124         ###########################################
125         # 2. Generate a Feasible Solution (UB) #
126
127         # 2a. For each customer, assign to the closest facility
128         current_UB = sum([minimum(D[i, best_facilities]) for i in
N])
129
130         # 2b. Update the best upper bound
```

```
        UB = min(UB, current_UB)
        push!(UB_history, UB)

        #############################
        # 3. Check for termination #
        relative_gap = 100 * (abs(UB - LB) / (UB + 1e-8))

        if relative_gap < gap_tolerance
            @printf("%6d | %15.3f | %15.3f | %9.3f\n", iteration, LB, UB, relative_gap)
            println("-"^54)
            @printf("Convergence achieved at iteration %d with gap %.3f%%.\n", iteration, relative_gap)
            break
        end

        # Print iteration details
        if iteration % 10 == 0 || iteration == 1
            @printf("%6d | %15.3f | %15.3f | %9.3f\n", iteration, LB, UB, relative_gap)
        end

        ###########################
        # 4. Update Multipliers #

        # 4a. Calculate the subgradient s_i = 1 - sum(x_ij)
        s = ones(n)
        for i in N
            for j in best_facilities
                if D[i, j] - λ[i] < 0
                    s[i] -= 1
                end
            end
        end

        # 4b. Determine the step size α_k (Held-Karp style)
        α = θ * ((UB - LB) / (norm(s)^2 + 1e-8))

        # 4c. Update the multipliers
        λ .+= α .* s
    end
```

```
169
170     # Final summary and Exact solve for comparison
171     exact_objective = solve_pmp_exact(instance)
172     println("Solving exact PMP for comparison...")
173     @printf("Final LB: %.3f | Exact: %.3f | Duality Gap: %.3f%%\n",
    LB, exact_objective, 100 * abs(exact_objective - LB)/
    exact_objective)
174
175     # Plot convergence if requested
176     if show_convergence_plot
177         plot_convergence(LB_history, UB_history)
178     end
179 end
180
181 # Example usage
182 solve_pmp_lagrangian("data/coordinates.csv", p = 3, iteration_max =
    300, gap_tolerance = 1e-6, show_convergence_plot = true)
```

Listing 7.3: Lagrangian Relaxation Julia implementation for PMP.

Listing 7.4 presents the output of the Lagrangian relaxation algorithm when applied to a PMP instance. The output includes the iteration number, current lower bound (LB), upper bound (UB), and the relative gap between them. The algorithm terminates when the relative gap falls below a specified tolerance, indicating convergence. Finally, it compares the final lower bound with the exact solution obtained by solving the original PMP model.

Output

```
----------- Starting Lagrangian Relaxation -----------
Iter.   |   Lower Bound   |   Upper Bound   |   Gap (%)
------------------------------------------------------
     1 |           0.000 |       27215.841 |   100.000
    10 |       12140.596 |       23591.169 |    48.538
    20 |       15108.058 |       23591.169 |    35.959
    30 |       16705.503 |       23591.169 |    29.187
    40 |       18181.478 |       23591.169 |    22.931
    50 |       18828.864 |       23591.169 |    20.187
    60 |       18948.300 |       23591.169 |    19.681
    70 |       18948.300 |       23228.585 |    18.427
    80 |       19092.205 |       23228.585 |    17.807
    90 |       19092.205 |       23228.585 |    17.807
```

```
   100 |       19092.205 |       23228.585 |    17.807
   110 |       19092.205 |       20369.729 |     6.272
   120 |       19211.024 |       19211.027 |     0.000
   124 |       19211.027 |       19211.027 |     0.000
-----------------------------------------------------
Convergence achieved at iteration 124 with gap 0.000%.
Solving exact PMP for comparison...
Final LB: 19211.027 | Exact: 19211.027 | Duality Gap: 0.000%
```

Listing 7.4: Output of the Lagrangian Relaxation for PMP.

Figure 7.4 shows the convergence plot of the lower bound (LB) and upper bound (UB) over iterations for the PMP instance. The plot illustrates how the bounds evolve and converge as the algorithm iterates.

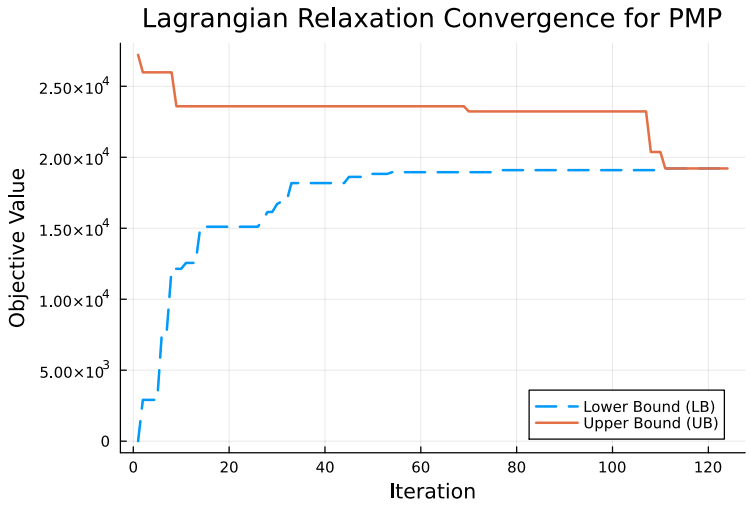


Figure 7.4: Lagrange Relaxation Convergence for PMP.

7.4 Lagrangian Relaxation for the Maximum Independent Set Problem

The Maximum Independent Set Problem (MISP) (Chapter 5) is a fundamental problem in graph theory. Given a graph, the goal is to find the largest possible set of vertices such that no two vertices in the set are connected by an edge. As it is NP-hard, exact solutions are often intractable for large graphs, making heuristic and approximation methods like Lagrangian relaxation valuable.

In [71], the authors proposed a technique named **LagClus** that explores the following characteristics of some combinatorial problems:

1. Many hard combinatorial problems can be written on a **conflict graph**.
2. These graphs are often **sparse and locally dense**.
3. If you **cut** the graph into clusters, most constraints are **intra-cluster**, not inter-cluster.

This approach leads to a powerful decomposition strategy:

- **Cluster Decomposition**: By partitioning the graph into clusters (subgraphs), most edge constraints are internal to clusters. Only a small number of constraints are "cut" between clusters.
- **Lagrangian Relaxation**: Relax the inter-cluster edge constraints, allowing each cluster to be solved independently. The relaxed constraints are penalized in the objective using Lagrange multipliers.
- **Iterative Coordination**: The multipliers are updated iteratively (e.g., via the subgradient method) to coordinate the solutions of the clusters, driving the overall solution toward feasibility.

This method leverages the sparsity and local density of real-world graphs, enabling efficient solution of large instances by focusing computational effort where constraints are most challenging.

7.4.1 LagClus Algorithm

Algorithm 7.3 presents the pseudocode for the **LagClus** algorithm. The algorithm begins by constructing the conflict graph and partitioning it into clusters. It then identifies the inter-cluster edges and initializes the Lagrange multipliers. The main loop iteratively solves the Lagrangian relaxation for each cluster, builds a relaxed solution, repairs it to a feasible solution, and updates the multipliers using a subgradient method until convergence is achieved. Finally, it returns the best feasible solution found.

LagClus Algorithm

```
1 Build conflict graph (G)
2 Partition (G) into clusters
3 Identify inter-cluster edges
4 Initialize multipliers (λ = 0)
5 while not converged do
```

```
LagClus Algorithm
    Solve MISP lagrangian relaxation in each cluster
    Build relaxed solution
    Repair to feasible MISP
    Update multipliers (subgradient)
end
Return best feasible solution
```

Algorithm 7.3: LagClus pseudocode.

The following subsections will show how the **LagClus** technique is adapted for solving MISP.

7.4.2 Solving MISP using LagClus

First, let's review the mathematical formulation of MISP. Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. The standard integer programming formulation for the MISP is given in Equation (5.2).

Definition 7.10 — MISP model. Given a graph $G = (V, E)$.

$$\max_{x} \quad \sum_{v \in V} x_v \tag{7.23.1}$$

$$\text{s.t.} \quad x_u + x_v \leq 1 \quad \forall \{u, v\} \in E \tag{7.23.2}$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \tag{7.23.3}$$

Here $x_v = 1$ if vertex v is included in the independent set, and 0 otherwise. The objective function maximizes the number of selected vertices, and the constraints ensure that no edge has both endpoints selected.

For the MISP, the **conflict graph** is the problem itself. Since two vertices cannot be selected simultaneously, the constraints of the model represent the conflict graph. Let's explain each step of **LagClus** using the graph from Figure 7.5.

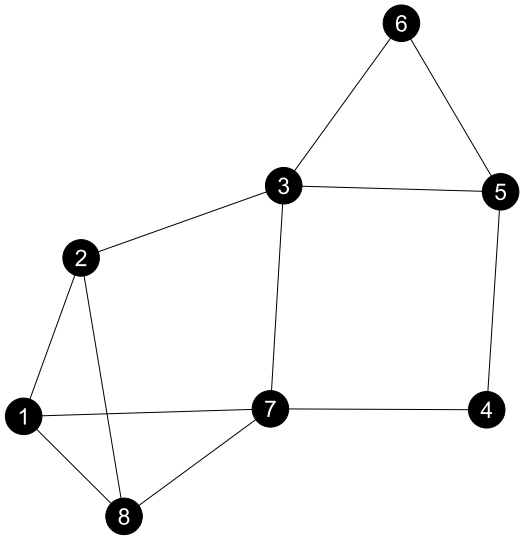


Figure 7.5: Example graph for LagClus.

Given the graph $G = (V, E)$ with $V = \{v_1, ..., v_8\}$ from Figure 7.5, the MISP model is given in Equation (7.24).

$$\max_{x} \quad \sum_{v \in V} x_v \tag{7.24.1}$$

$$\text{s.t.} \quad x_1 + x_2 \leq 1 \tag{7.24.2}$$

$$x_1 + x_7 \leq 1 \tag{7.24.3}$$

$$x_1 + x_8 \leq 1 \tag{7.24.4}$$

$$x_2 + x_3 \leq 1 \tag{7.24.5}$$

$$x_2 + x_8 \leq 1 \tag{7.24.6}$$

$$x_3 + x_5 \leq 1 \tag{7.24.7}$$

$$x_3 + x_6 \leq 1 \tag{7.24.8}$$

$$x_3 + x_7 \leq 1 \tag{7.24.9}$$

$$x_4 + x_5 \leq 1 \tag{7.24.10}$$

$$x_4 + x_7 \leq 1 \tag{7.24.11}$$

$$x_5 + x_6 \leq 1 \tag{7.24.12}$$

$$x_7 + x_8 \leq 1 \tag{7.24.13}$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \tag{7.24.14}$$

The next subsections will show how the **LagClus** technique is applied to the model in Equation (7.24).

First step: Metis partitioning

The first step in **LagClus** is to apply the Metis [64] technique. Figure 7.6 presents the obtained partitioning result. The graph is partitioned into two clusters: $C_1 = \{v_1, v_2, v_7, v_8\}$ and $C_2 = \{v_3, v_4, v_5, v_6\}$. The edges that connect vertices from different clusters are called inter-cluster edges. In this example, the inter-cluster edges are $\{2, 3\}$, $\{3, 7\}$ and $\{4, 7\}$. These edges correspond to the constraints Equation (7.24.5), Equation (7.24.9) and Equation (7.24.11), that will be relaxed in the Lagrangian relaxation step. The remaining edges are intra-cluster and will be handled within each cluster's subproblem.

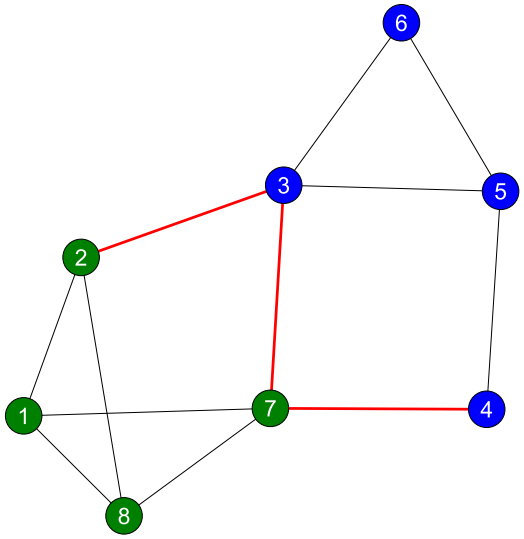


Figure 7.6: Metis partitioning of the example graph.

Second step: Lagrangian Relaxation

The inter-cluster constraints Equation (7.24.5), Equation (7.24.9) and Equation (7.24.11) from model in Equation (7.24) are relaxed by introducing lagrangian multipliers $\lambda \geq 0$:

$$x_2 + x_3 \leq 1 \quad \Rightarrow \quad \lambda_1(1 - x_2 - x_3) \tag{7.25.1}$$

$$x_3 + x_7 \leq 1 \quad \Rightarrow \quad \lambda_2(1 - x_3 - x_7) \tag{7.25.2}$$

$$x_4 + x_7 \leq 1 \quad \Rightarrow \quad \lambda_3(1 - x_4 - x_7) \tag{7.25.3}$$

The global Lagrangian objective becomes:

$$\max_x \sum_{i=1}^{8} x_i + \lambda_1(1 - x_2 - x_3) + \lambda_2(1 - x_3 - x_7) + \lambda_3(1 - x_4 - x_7) \tag{7.26}$$

The relaxed problem decomposes into **independent subproblems**. Each subproblem contains:

- its local variables
- all intra-cluster constraints
- the terms of the global Lagrangian objective involving its variables (penalties)

For the first subproblem (E_1), that represents cluster $C_1 = \{v_1, v_2, v_7, v_8\}$, the following penalties are considered: $-\lambda_1 x_2$, $-\lambda_2 x_7$ and $-\lambda_3 x_7$. Equation (7.27) presents the resulting subproblem model for cluster C_1.

Definition 7.11 — E_1 subproblem model. Given the graph $G = (V, E)$ and the cluster $C_1 = \{v_1, v_2, v_7, v_8\}$.

$$\max_x \quad x_1 + (1-\lambda_1)x_2 + (1-\lambda_2-\lambda_3)x_7 + x_8 \tag{7.27.1}$$

$$\text{s.t.} \quad x_1 + x_2 \leq 1 \tag{7.27.2}$$

$$x_1 + x_7 \leq 1 \tag{7.27.3}$$

$$x_1 + x_8 \leq 1 \tag{7.27.4}$$

$$x_2 + x_8 \leq 1 \tag{7.27.5}$$

$$x_7 + x_8 \leq 1 \tag{7.27.6}$$

$$x_v \in \{0,1\} \quad \forall v \in C_1 \tag{7.27.7}$$

For the second subproblem (E_2), that represents cluster $C_2 = \{v_3, v_4, v_5, v_6\}$, the following penalties are considered: $-\lambda_1 x_3$, $-\lambda_2 x_3$ and $-\lambda_3 x_4$. Equation (7.28) presents the resulting subproblem model for cluster C_2.

Definition 7.12 — E_2 subproblem model. Given the graph $G = (V, E)$ and the cluster $C_2 = \{v_3, v_4, v_5, v_6\}$.

$$\max_x \quad (1-\lambda_1-\lambda_2)x_3 + (1-\lambda_3)x_4 + x_5 + x_6 \tag{7.28.1}$$

$$\text{s.t.} \quad x_3 + x_5 \leq 1 \tag{7.28.2}$$

$$x_3 + x_6 \leq 1 \tag{7.28.3}$$

$$x_4 + x_5 \leq 1 \tag{7.28.4}$$

$$x_5 + x_6 \leq 1 \tag{7.28.5}$$

$$x_v \in \{0,1\} \quad \forall v \in C_2 \tag{7.28.6}$$

Because the relaxation decomposes, the Lagrangian value can be written as:

$$z_{E(\lambda)} = z_{E_1(\lambda)} + z_{E_2(\lambda)} + (\lambda_1 + \lambda_2 + \lambda_3) \tag{7.29}$$

where:

- $z_{E_1(\lambda)}$ is the optimal value of subproblem $E_1(\lambda)$
- $z_{E_2(\lambda)}$ is the optimal value of subproblem $E_2(\lambda)$
- the last term collects constants from the relaxed constraints

Third step: Lagrangian coordination (subgradient optimization

After decomposition, the subproblems $E_1(\lambda)$ and $E_2(\lambda)$ can be solved independently for a given multiplier vector (λ). However, their solutions may violate the relaxed inter-cluster constraints. To enforce coordination between clusters, the Lagrange multipliers are updated iteratively using a subgradient method.

Let $x^{(t)}$ denote the solution obtained at iteration (t) by solving all subproblems. The subgradient $s^{(t)} = \left(s_1^{(t)}, s_2^{(t)}, s_3^{(t)}\right)$ associated with each relaxed constraint is given by its violation:

$$s_1^{(t)} = 1 - \left(x_2^{(t)} + x_3^{(t)}\right), \tag{7.30.1}$$

$$s_2^{(t)} = 1 - \left(x_3^{(t)} + x_7^{(t)}\right), \tag{7.30.2}$$

$$s_3^{(t)} = 1 - \left(x_4^{(t)} + x_7^{(t)}\right). \tag{7.30.3}$$

The multipliers are then updated according to:

$$\lambda_k^{(t+1)} = \max\left(0, \lambda_k^{(t)} - \alpha^{(t)} g_k^{(t)}\right), \quad k = 1, 2, 3 \tag{7.31}$$

where ($\alpha^{(t)} > 0$) is the step size at iteration (t). A commonly used step-size rule is:

$$\alpha^{(t)} = \theta \left(\frac{\mathrm{UB}^{(t)} - \mathrm{LB}^{(t)}}{\left\| s^{(t)} \right\|^2} \right), \quad \theta \in (0, 2] \tag{7.32}$$

Here $\mathrm{UB}^{(t)} = z_{E(\lambda^{(t)})}$ is the upper bound given by the lagrangian relaxation, while $\mathrm{LB}^{(t)}$ is the best known feasible solution found so far.

This iterative process is repeated until a stopping criterion is met, such as a maximum number of iterations or convergence of the Lagrangian bound.

Fourth step: Primal recovery and local improvement

Since the solutions obtained from the relaxed problem may be infeasible for the original MISP, a primal recovery heuristic is applied at each iteration to construct a feasible independent set. This heuristic removes vertices involved in violated inter-cluster constraints until feasibility is restored, and may be followed by a local search to improve solution quality.

7.4.3 Solving LagClus for MISP using Julia

Listing 7.5 presents a Julia implementation of the **LagClus** technique for solving MISP. The code includes functions to plot bounds convergence (Listing 7.5-9), solve the original MISP exactly (Listing 7.5-22), perform the Metis partitioning (Listing 7.5-49), create a feasible solution from the Lagrangian solution (Listing 7.5-67), solve the Lagrangian subproblem for each cluster (Listing 7.5-89). The main function `solve_misp_lagclus` orchestrates the entire process (Listing 7.5-125), from partitioning to solving subproblems and updating multipliers. The implementation uses the JuMP package for mathematical programming, HiGHS as the solver, Graphs for graph operations, and Metis for graph partitioning.

```
using JuMP              # For mathematical programming
using HiGHS             # HiGHS solver
using Graphs            # Graphs package
using Metis             # For graph partitioning
using LinearAlgebra     # For norm calculation
using Printf            # For formatted output
using Plots             # For plotting convergence

# Function to plot convergence of LB and UB
function plot_convergence(lb_history, ub_history)
    p = plot(1:length(lb_history), [lb_history ub_history],
        label=["Lower Bound (LB)" "Upper Bound (UB)"],
        xlabel="Iteration",
        ylabel="Objective Value",
        title="LagClus Convergence for MISP",
        linewidth=2,
        linestyle=[:dash :solid],
        legend=:topright)
    display(p)
end

# Function to solve MISP exactly
function solve_misp_exact(graph)
    V = 1:Graphs.nv(graph)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
```

```
    # Variables
    @variable(model, x[v in V], Bin)
    # Objective: Maximize sum of x[i]
    @objective(model, Max, sum(x[v] for v in V))
    # Constraints: x[u] + x[v] <= 1 for each edge (u,v)
    for e in Graphs.edges(graph)
        u = Graphs.src(e)
        v = Graphs.dst(e)
        @constraint(model, x[u] + x[v] <= 1)
    end
    # Run the solver
    JuMP.optimize!(model)
    # Return the optimal value
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        return JuMP.objective_value(model)
    else
        error("Optimal solution not found.")
    end
end

# Function to partition graph using Metis
function metis_partition(graph, k)
    partition = Metis.partition(graph, k)
    clusters = [Int[] for _ in 1:k]
    for v in 1:Graphs.nv(graph)
        push!(clusters[partition[v]], v)
    end
    cut_edges = []
    for (e_id, e) in enumerate(Graphs.edges(graph))
        u = Graphs.src(e)
        v = Graphs.dst(e)
        if partition[u] != partition[v]
            push!(cut_edges, (u,v))
        end
    end
    return clusters, cut_edges
end

# Function to create a feasible solution from Lagrangian solution
function create_feasible_solution(graph, lag_solution)
    n = Graphs.nv(graph)
```

```
    feasible_solution = copy(lag_solution)
    # Greedily remove conflicts
    for u in 1:n
        if feasible_solution[u] == 1
            for v in Graphs.neighbors(graph, u)
                if feasible_solution[v] == 1
                    # Break tie by degree (drop node with higher degree)
                    if Graphs.degree(graph, u) ≥ Graphs.degree(graph, v)
                        feasible_solution[u] = 0
                    else
                        feasible_solution[v] = 0
                    end
                end
            end
        end
    end
    return feasible_solution
end

# Function to solve Lagrangian subproblem for a cluster
function solve_misp_lagrangian(graph, cluster, λ, solution)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[cluster], Bin)
    # Objective: Maximize sum of x[i]
    obj = sum(x[i] for i in cluster)
    # Add λ terms to objective and constraints
    for u in cluster
        for v in Graphs.neighbors(graph, u)
            if v in cluster
                if u < v
                    @constraint(model, x[u] + x[v] <= 1)
                end
            else
                if u < v
                    obj -= λ[(u,v)] * x[u]
```

```
                else
                    obj -= λ[(v,u)] * x[u]
                end
            end
        end
    end
    @objective(model, Max, obj)
    # Run the solver
    JuMP.optimize!(model)
    # Get the values of the decision variables
    for i in cluster
        solution[i] = JuMP.value(x[i])
    end
    return JuMP.objective_value(model)
end

# Main function to solve MISP using LagClus
function solve_misp_lagclus(graph; iteration_max = 2, k = 2,
gap_tolerance = 1e-6, show_convergence_plot = false)

    # Get Metis partition
    clusters, cut_edges = metis_partition(graph, k)

    # Initialize Lagrange multipliers and subgradient for cut edges
    λ = Dict{Tuple{Int,Int}, Float64}()
    s = Dict{Tuple{Int,Int}, Float64}()
    for (u,v) in cut_edges
        λ[(u,v)] = 0.0
        s[(u,v)] = 0.0
    end

    # Initialize parameters
    θ = 0.8   # A factor to update multiplier (Held-Karp)
    LB = -Inf # Upper bound
    UB = Inf  # Lower bound
    LB_history = [] # Lower bound history
    UB_history = [] # Upper bound history

    # Logic to reduce θ if no improvement in LB
    no_improve_limit = 10
    no_improve_counter = 0
```

```
149
150     # Print header
151     println("-"^11," Starting Lagrangian Relaxation ", "-"^11)
152     println("Iter.  |   Lower Bound   |   Upper Bound   |   Gap
(%)")
153     println("-"^54)
154
155     # Store the best feasible solution found
156     best_feasible_solution = zeros(Graphs.nv(graph))
157
158     # Main loop
159     for iteration in 1:iteration_max
160
161         #############################################
162         # 1. Solve the Lagrangian Relaxation L(λ) #
163
164         # 1a. Solve Lagrangian subproblems
165         lag_objective = 0.0
166         lag_solution = zeros(Graphs.nv(graph))
167
168         # Solve for each cluster
169         for cluster in clusters
170             lag_objective += solve_misp_lagrangian(graph, cluster,
λ, lag_solution)
171         end
172
173         # Add penalty for cut edges
174         lag_objective += sum(λ[(u,v)] for (u,v) in cut_edges)
175
176         # 1b. Update UB (Upper bound)
177         if lag_objective < UB
178             UB = lag_objective
179             no_improve_counter = 0
180         else
181             no_improve_counter += 1
182             # Reduce θ if progress stalls
183             if no_improve_counter >= no_improve_limit
184                 # Reset counter
185                 no_improve_counter = 0
186                 # Reduce θ to allow for finer adjustments
187                 θ = max(0.5, θ * 0.5)
```

```
            end
        end
        push!(UB_history, UB)

        ####################################
        # 2. Generate a Feasible Solution (LB) #

        # 2a. Create a feasible solution by greedy repair
        feasible_solution = create_feasible_solution(graph,
lag_solution)
        feasible_objective = sum(feasible_solution)
        if feasible_objective > LB
            LB = feasible_objective
            best_feasible_solution = copy(feasible_solution)
        end
        push!(LB_history, LB)

        ########################
        # 3. Check for termination #

        relative_gap = 100 * (abs(UB - LB) / (UB + 1e-8))

        # Print iteration results
        @printf("%6d | %15.3f | %15.3f | %9.3f\n", iteration, LB,
UB, relative_gap)

        # Check convergence
        if relative_gap < gap_tolerance
            @printf("Convergence achieved at iteration %d with gap
%.3f%%.\n", iteration, relative_gap)
            break
        end

        ######################
        # 4. Update Multipliers #

        # 4a. Calculate Lagrange multipliers
        s_norm = []
        for (u,v) in cut_edges
            s[(u,v)] = lag_solution[u] + lag_solution[v] - 1
            push!(s_norm, s[(u,v)])
```

```
226         end
227
228         # 4b. Adaptive step-size update (Held-Karp)
229         α = θ * ((UB - LB) / (norm(s_norm)^2 + 1e-8))
230
231         # 4c. Update the multipliers
232         for (u,v) in cut_edges
233             λ[(u,v)] = max(0.0, λ[(u,v)] + α * s[(u,v)])
234         end
235     end
236
237     # Final summary and Exact solve for comparison
238     exact_objective = solve_misp_exact(graph)
239     println("-"^54)
240     println("Solving exact MISP for comparison...")
241     @printf("Final UB: %.4f | Exact: %.4f | Duality Gap: %.3f%%\n",
    UB, exact_objective, 100 * abs(exact_objective- UB)/
    exact_objective)
242
243     # Plot convergence if requested
244     if show_convergence_plot
245         plot_convergence(LB_history, UB_history)
246     end
247 end
248
249 # Create graph
250 edges = [(1,2), (1,7), (1,8), (2,3),
251          (2,8), (3,5), (3,6), (3,7),
252          (4,5), (4,7), (5,6), (7,8)]
253 graph = Graph(8)
254 for (u,v) in edges
255     add_edge!(graph, u, v)
256 end
257
258 # Example usage
259 solve_misp_lagclus(graph, iteration_max = 200, k=2, gap_tolerance =
    1e-6, show_convergence_plot = true)
```

Listing 7.5: **LagClus** Julia implementation for MISP.

Listing 7.6 presents the output of the Lagrangian relaxation algorithm when applied to a MISP instance. The output includes the iteration number, current lower bound (LB), upper bound (UB), and the relative gap between them. The algorithm terminates when the relative gap falls below a specified tolerance, indicating convergence. Finally, it compares the final upper bound with the exact solution obtained by solving the original MISP model.

Output

```
----------- Starting Lagrangian Relaxation -----------
Iter.  |   Lower Bound   |   Upper Bound   |   Gap (%)
------------------------------------------------------
     1 |           3.000 |           4.000 |    25.000
     2 |           3.000 |           3.200 |     6.250
     3 |           3.000 |           3.040 |     1.316
     4 |           3.000 |           3.008 |     0.266
     5 |           3.000 |           3.002 |     0.053
     6 |           3.000 |           3.000 |     0.011
     7 |           3.000 |           3.000 |     0.002
     8 |           3.000 |           3.000 |     0.000
     9 |           3.000 |           3.000 |     0.000
    10 |           3.000 |           3.000 |     0.000
    11 |           3.000 |           3.000 |     0.000
    12 |           3.000 |           3.000 |     0.000
Convergence achieved at iteration 12 with gap 0.000%.
------------------------------------------------------
Solving exact MISP for comparison...
Final UB: 3.0000 | Exact: 3.0000 | Duality Gap: 0.000%
```

Listing 7.6: Output of the Lagrangian Relaxation for MISP.

Figure 7.7 shows the convergence plot of the lower bound (LB) and upper bound (UB) over iterations for the MISP instance. The plot illustrates how the bounds evolve and converge as the algorithm iterates.

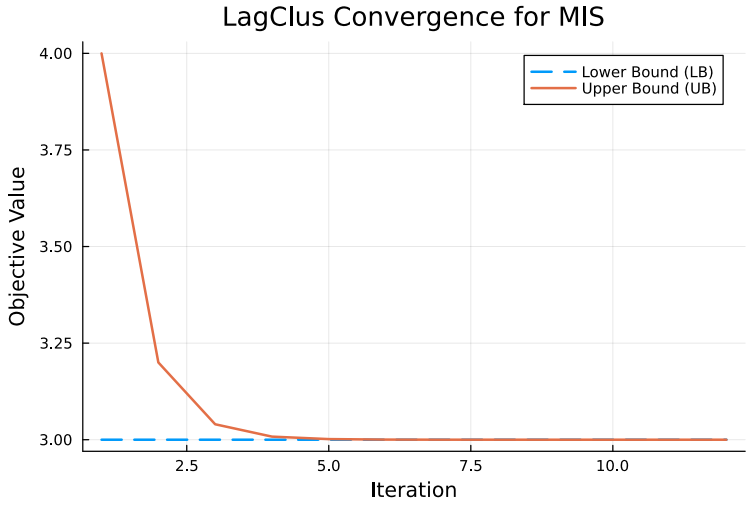


Figure 7.7: Lagrange Relaxation Convergence for MISP.

7.5 Lagrangian relaxation for the Generalized Assignment Problem

The Generalized Assignment Problem (GAP) is a classic combinatorial-optimization challenge that involves assigning a set of jobs to a group of agents at minimum cost, subject to each agent's limited capacity. Equation (7.33) presents the mathematical formulation of the GAP.

Definition 7.13 — GAP model. Given a set of agents $A = \{a_1, a_2, a_3, ...\}$ and a set of jobs $J = \{j_1, j_2, j_3, ...\}$.

$$\min_{x} \quad \sum_{a \in A} \sum_{j \in J} c_{aj} x_{aj} \tag{7.33.1}$$

$$\text{s.t.} \quad \sum_{a \in A} x_{aj} = 1 \qquad \forall j \in J \tag{7.33.2}$$

$$\sum_{j \in J} r_{aj} x_{aj} \leq C_a \qquad \forall a \in A \tag{7.33.3}$$

$$x_{aj} \in \{0, 1\} \qquad \forall a \in A, j \in J \tag{7.33.4}$$

where:

- c_{aj} cost of assigning agent a to job j
- x_{aj} binary variable indicating if agent a is assigned to job j
- r_{aj} resource consumption of assigning agent a to job j
- C_a capacity of agent a

Given a set of agents $A = \{a_1, a_2, a_3, ...\}$ and a set of jobs $J = \{j_1, j_2, j_3, ...\}$, the goal is to determine the optimal assignment of jobs to agents while minimizing the total cost and respecting the capacity constraints of each agent. The objective is to minimize the total assignment cost, where c_{aj} represents the cost of assigning job j to agent a. The constraints ensure that each job is assigned to exactly one agent (assignment constraints), and that the total resource consumption for each agent does not exceed its capacity C_a (capacity constraints). The decision variables x_{aj} are binary, indicating whether job j is assigned to agent a.

Due to its NP-hard nature, solving large-scale instances optimally is computationally difficult. Advanced techniques such as Lagrangian relaxation, combined with the subgradient method, provide effective strategies for finding high-quality solutions efficiently.

We can have more than one Lagrangian relaxation for a given problem. They can generate different bounds and one can be better than the other. In the next subsections, two Lagrangian relaxations for the GAP will be presented [20].

7.5.1 Relaxation 1: Relaxing the assignment constraints

We relax the assignment constraints in Equation (7.33.2) using a vector of Lagrange multipliers $\lambda = (\lambda_1, \lambda_2, \lambda_3, ...)$ associated with each job. The relaxed problem, denoted as $L_1(\lambda)$, is given in Equation (7.34). The objective function includes the original cost plus a penalty term for violating the assignment constraints. The ca pacity constraints and binary restrictions remain intact.

Definition 7.14 — $L_1(\lambda)$ model. Given a set of agents $A = \{a_1, a_2, a_3, ...\}$ and a set of jobs $J = \{j_1, j_2, j_3, ...\}$.

$$\min_{x} \quad \sum_{a \in A} \sum_{j \in J} (c_{aj} - \lambda_j) x_{aj} + \sum_{j \in J} \lambda_j \tag{7.34.1}$$

$$\text{s.t.} \quad \sum_{j \in J} r_{aj} x_{aj} \leq C_a \qquad \forall a \in A \tag{7.34.2}$$

$$x_{aj} \in \{0, 1\} \qquad \forall a \in A, j \in J \tag{7.34.3}$$

This separates across agents. For a fixed λ, each agent a solves the independent 0-1 knapsack problem model in Equation (7.35).

Definition 7.15 — $L_1(\lambda, a)$ model. For a vector of Lagrange multipliers λ.

$$\min_x \quad \sum_{j \in J} (c_{aj} - \lambda_j) x_{aj} \tag{7.35.1}$$

$$\text{s.t.} \quad \sum_{j \in J} r_{aj} x_{aj} \leq C_a \qquad \forall a \in A \tag{7.35.2}$$

$$x_{aj} \in \{0, 1\} \qquad \forall j \in J \tag{7.35.3}$$

The sum of the solutions across agents, plus the lagrangian multipliers, gives a lower bound on the original problem:

$$L_1(\lambda) = \sum_{a \in A} L_1(\lambda, a) + \sum_{j \in J} \lambda_j \tag{7.36}$$

7.5.2 Relaxation 2: Relaxing the capacity constraints

Alternatively, we can relax the capacity constraints in Equation (7.33.3) using a vector of Lagrange multipliers $\mu = (\mu_1, \mu_2, \mu_3, ...)$ associated with each agent. The relaxed problem, denoted as $L_2(\mu)$, is given in Equation (7.37). The objective function includes the original cost plus a penalty term for violating the capacity constraints. The assignment constraints and binary restrictions remain intact.

Definition 7.16 — $L_2(\mu)$ model. Given a set of agents $A = \{a_1, a_2, a_3, ...\}$ and a set of jobs $J = \{j_1, j_2, j_3, ...\}$.

$$\min_x \quad \sum_{a \in A} \sum_{j \in J} (c_{aj} + \mu_a r_{aj}) x_{aj} - \sum_{a \in A} \mu_a C_a \tag{7.37.1}$$

$$\text{s.t.} \quad \sum_{a \in A} x_{aj} = 1 \qquad \forall j \in J \tag{7.37.2}$$

$$x_{aj} \in \{0, 1\} \qquad \forall a \in A, j \in J \tag{7.37.3}$$

This separates across jobs. For each job j, we solve the model in Equation (7.38).

Definition 7.17 — $L_2(\mu, j)$ model. For a vector of Lagrange multipliers μ.

$$\min_x \quad \sum_{a \in A} (c_{aj} + \mu_a r_{aj}) x_{aj} \tag{7.38.1}$$

$$\text{s.t.} \quad \sum_{a \in A} x_{aj} = 1 \qquad \forall j \in J \tag{7.38.2}$$

$$x_{aj} \in \{0, 1\} \qquad \forall a \in A, j \in J \tag{7.38.3}$$

Note

1. Problem $L_1(\lambda)$ decomposes into n independent 0-1 knapsack problems, one for each agent. Although knapsack problems are NP-hard, they can often be solved efficiently in practice, even for large-scale instances, using dynamic programming or pseudo-polynomial algorithms.
2. Problem $L_2(\mu)$ can be solved in polynomial time, specifically $O(nm)$, by computing $\arg\min_a \left(c_{aj} + \mu_a r_{aj}\right)$ for each job j, and setting $x_{aj} = 1$ for the minimizing agent and 0 for the others.
3. We can note that $L_2(\mu)$ has an integrality property [81], while $L_1(\lambda)$ does not. This leads to $L_2(\mu) \leq L_1(\lambda)$. The integrality property means that solving the Lagrangian dual of $L_2(\mu)$ yields the same value as the linear programming relaxation of the original integer problem.

7.5.3 Solving the Dual Problem with the Subgradient Method

To obtain the best possible lower bound, we seek to maximize the duals for $L_1(\lambda)$ or $L_2(\mu)$. Since these functions are concave but non-differentiable, we apply the **subgradient method** presented in Algorithm 7.2. Table 7.3 summarizes the main differences between the two relaxations in terms of their dual functions, subgradients and multiplier update rules.

Component	L_1	L_2
Dual	$\max_{\lambda \in \mathbb{R}} z_{L_1(\lambda)}$	$\max_{\mu \geq 0} z_{L_2(\mu)}$
Subgradient	$s_j^{(t)} = 1 - \sum_{a \in A} x_{aj}^{(t)}$	$s_a^{(t)} = \sum_{j \in J} r_{aj} x_{aj}^{(t)} - C_a$
Update multipliers	$\lambda_j^{(t+1)} = \lambda_j^{(t)} + \alpha^{(t)} s_j^{(t)}$	$\mu_a^{(t+1)} = \max\left(0, \mu_a^{(t)} + \alpha^{(t)} s_a^{(t)}\right)$

Table 7.3: Subgradient components for L_1 and L_2.

7.6 Final remarks

Lagrangian Relaxation is a powerful and widely used technique for obtaining good bounds in Integer Programming (IP) problems. It has proven to be effective in providing tighter bounds compared to the linear programming relaxation,

making it a valuable tool for solving difficult combinatorial optimization problems.

The idea behind Lagrangian Relaxation is to relax some of the constraints of the original IP problem by introducing Lagrange multipliers (also known as dual variables). By relaxing these constraints, we can obtain a lower bound on the optimal objective value of the IP problem. The Lagrange multipliers are then updated iteratively using a subgradient-type optimization method, which further improves the lower bound.

Lagrangian Relaxation is related to the column generation method because both techniques involve solving a series of linear programming problems. In column generation, we generate columns (variables) at each node of the branch and bound search tree, while in Lagrangian Relaxation, we solve a series of relaxed linear programming problems by adjusting the Lagrange multipliers.

The exercises in this chapter will contribute to the understanding of the Lagrangian Relaxation technique. By solving these exercises, readers will gain practical experience in applying Lagrangian Relaxation to some combinatorial optimization problems and see firsthand how it can be used to obtain tighter bounds and improve the overall performance of solution algorithms.

Overall, Lagrangian Relaxation is a valuable addition to the toolbox of techniques for solving IP problems, and its combination with other methods like column generation can lead to powerful solution approaches for challenging real-world optimization problems.

7.7 Exercises

Formulate and solve the following exercises using the JuMP package.

Exercise 7.1 Consider the following model:

$$\max_{x_1,x_2} \quad x_1 + 2x_2 \tag{7.39.1}$$

$$\text{s.t.} \quad 2x_1 + x_2 \leq 2 \tag{7.39.2}$$

$$x_1, x_2 \in \{0, 1\} \tag{7.39.3}$$

a) Make a graphical representation of the problem and find the optimal solution and the linear programming relaxation.

b) Find the Lagrangian relaxation for the constraint $2x_1 + x_2 \leq 2$. Make a graphical representation of the dual's solution. Is there a duality gap?

c) Find the optimal solution of the dual using the subgradient method.

Exercise 7.2 Consider the following exercises related to P-Median Problems:

a) Apply the subgradient method to get a bound on the optimal solution of the P-Median Problem using $p = 3$. Use the following dataset: exercise_pmp.csv.

b) Apply the subgradient method to get a bound on the optimal solution of the fault-tolerant P-Median Problem [36] using $p = 5$ and $r = 2$. Use the following dataset: exercise_pmp.csv.

c) Apply the subgradient method to get a bound on the optimal solution of the Capacitated P-Median Problem using $p = 3$. Use the following dataset: exercise_cpmp.csv.

Exercise 7.3 Use the **LagClus** technique [71] to solve the Maximum Independent Set Problem (MISP) for the instance karate.csv. Is there a duality gap?

Exercise 7.4 Consider the Generalized Assignment Problem defined with the following data. Formulate the two Lagrangian relaxations presented in this chapter and solve by the subgradient method. Compare the limits and the optimal solution of the problem.

$$c_{aj} = \begin{pmatrix} 8 & 6 & 10 & 9 & 7 \\ 9 & 12 & 7 & 8 & 10 \\ 14 & 10 & 9 & 7 & 8 \end{pmatrix} \quad r_{aj} = \begin{pmatrix} 2 & 4 & 3 & 2 & 5 \\ 3 & 2 & 4 & 3 & 2 \\ 5 & 3 & 2 & 4 & 3 \end{pmatrix} \quad C_a = \begin{pmatrix} 10 \\ 9 \\ 11 \end{pmatrix}$$

8 Branch-and-Bound

This chapter introduces the **Branch-and-Bound** (B&B) algorithm, a powerful and widely used technique for solving Integer Programming (IP) and other difficult optimization problems. The central idea is to perform an implicit enumeration of the solution space. Instead of evaluating every feasible solution individually, which is often computationally intractable, the algorithm systematically partitions the problem into smaller subproblems and uses bounds to discard regions of the search space that cannot contain the optimal solution. This process relies on two fundamental operations: branching (partitioning the problem) and bounding (computing bounds). The method was first proposed by Land and Doig [82].

Figure 8.1 illustrates the central concept that makes this possible. It shows the feasible set for an original maximization problem (P) and the larger feasible region of its linear programming relaxation (LP relaxation). By optimizing over this larger, relaxed set, we can efficiently find an optimistic estimate of the optimal solution.

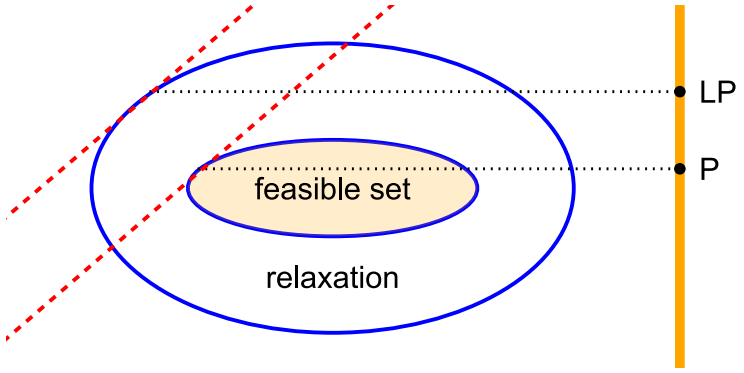


Figure 8.1: Relaxation bound for a maximization problem.

Since the feasible region of the original integer program is entirely contained within the feasible region of its LP relaxation, the optimal value of the relaxation, z_{LP}, provides a valid upper bound for the true optimal value z_{P} in maximization problems. This property forms the foundation of the B&B algorithm.

8.1 Branch-and-Bound Components

Figure 8.2 provides a flowchart of a generic B&B algorithm, illustrating its main components and the logical flow between them. The algorithm starts by finding an initial integer-feasible solution using a heuristic, which provides a starting in-

cumbent solution (Step 1). It then initializes the search with the original problem as the root node of the search tree (Step 2). The algorithm continues to process nodes from the active list until there are no more nodes to explore (Step 3). At each step, the algorithm selects a node (Step 4), solves its relaxation to obtain a bound (Step 5), check for better integer-feasible solutions to update the incumbent (Step 6), determines whether the node should be pruned (Step 7) or further branched based on the bound and the feasibility of the solution (Step 8). If a node is pruned, it is discarded from the search tree, which significantly reduces the number of nodes that need to be explored. If the node is not pruned, new subproblems are created by adding constraints, and these child nodes are added to the active list for further exploration (Step 9). The process continues iteratively until the active list is empty, at which point the best incumbent solution found is guaranteed to be the optimal integer solution to the original problem (Step 10).

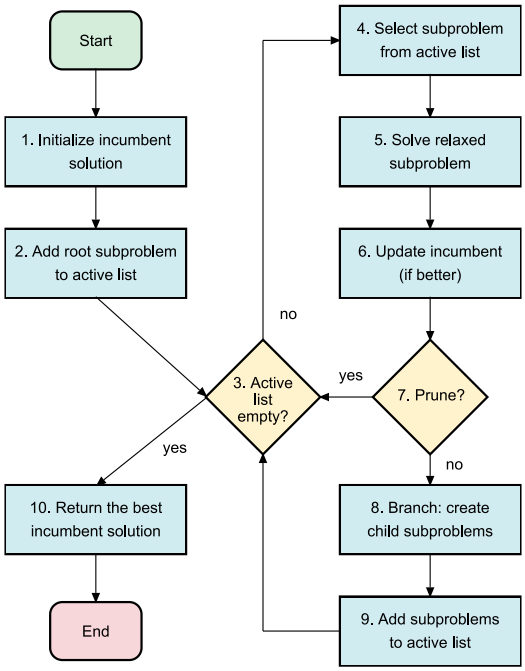


Figure 8.2: B&B flowchart.

Table 8.1 provides a detailed breakdown of the components of the generic B&B algorithm, outlining the key steps and their significance in the overall process. Each step is crucial for the efficiency and success of the algorithm, from finding an initial solution to managing the search tree and making decisions about pruning and branching.

Step	Detail
1	Use a heuristic to find an initial integer-feasible solution (incumbent solution). This can be done using various methods, such as greedy algorithms, local search, or metaheuristics. The quality of this initial solution can significantly impact the efficiency of the B&B algorithm, as it sets the initial bound.
2	Start with the original problem as the root node of the search tree. This node represents the entire feasible region of the problem without any additional constraints.
3	Continue the process until no active nodes remain to be explored. An active node is one that has been generated but not yet processed. The active list is typically managed using a data structure like a queue or a stack [83], depending on the search strategy.
4	Different strategies can be used to select the next node to process, such as Depth-First Search (DFS), Breadth-First Search (BFS), or Best-First Search (which selects the node with the most promising bound) [84]. The choice of strategy can affect the performance of the algorithm.
5	Solving the relaxation of a node typically involves solving a linear programming problem, which can be done efficiently using algorithms like the Simplex method or Interior Point methods. The solution to the relaxation provides an upper bound for maximization problems (or a lower bound for minimization problems) and helps determine whether to prune or branch further.
6	If we find a better integer-feasible solution, we update our incumbent solution and the global lower bound (for maximization problems) or upper bound (for minimization problems). This updated incumbent solution can help us prune more nodes in subsequent iterations, as it provides a tighter bound against which to compare the bounds of other nodes.
7	This step uses the bounds to make decisions about which branches of the tree are not worth exploring further. A node (and the entire branch below it) can be pruned for one of three reasons: • **Pruning by Bound**: If the upper bound of a node is less than or equal to the current incumbent value, we can discard this node.

Step	Detail
	Its optimistic estimate is already worse than a real solution we've already found, so it cannot possibly lead to a better one. • **Pruning by Infeasibility**: If a subproblem's relaxation has no feasible solution, then the original subproblem certainly doesn't either, and the node can be pruned. • **Pruning by Optimality**: If the solution to a node's relaxation happens to be integer-feasible, it is the optimal solution for that specific subproblem. We update our global lower bound (incumbent) if this solution is better and prune the node because no further branching can yield a better result in this path.
8	If the node is not pruned, we create new subproblems (child nodes) by adding constraints that restrict the feasible set. This action effectively splits the problem into smaller, mutually exclusive pieces. The effect of adding constraints means that the upper bound of a child node will always be less than or equal to the upper bound of its parent. The specific branching strategy can vary, such as branching on a variable (e.g., $x_i = 0$ or $x_i = 1$ for binary variables) or using more complex rules based on the structure of the problem.
9	The subproblems created by branching are added to the active list, which is a collection of nodes that have been generated but not yet processed. The algorithm will continue to select nodes from this active list and repeat the process of bounding and pruning until the list is empty. The management of the active list is crucial for the efficiency of the algorithm, as it determines the order in which nodes are explored.
10	The process of branching, bounding, and pruning continues iteratively until there are no more active nodes to explore. At this point, the best incumbent solution found during the search is guaranteed to be the optimal integer solution to the original problem.

Table 8.1: B&B component details.

In the next sections, we will see how to apply the B&B algorithm to different problems, starting with the Binary Knapsack Problem.

8.2 Maximization example: Binary Knapsack Problem

We will examine a B&B procedure for the Binary Knapsack Problem (BKP) considering the following example.

Example 8.1 Suppose we have the following instance of the BKP, considering a knapsack with capacity $W = 14$:

$$\max_{x} \quad 16x_1 + 22x_2 + 12x_3 + 8x_4 \tag{8.1.1}$$

$$\text{s.t.} \quad 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \tag{8.1.2}$$

$$x_i \in \{0, 1\} \qquad i = 1, ..., 4 \tag{8.1.3}$$

8.2.1 Step 1: Initialize incumbent solution

We start the B&B by finding an initial integer-feasible solution using a greedy heuristic. We can use the value-to-weight ratios to guide our selection of items to include in the knapsack. The ratios are calculated as follows:

$$\left(\frac{16}{5}, \frac{22}{7}, \frac{12}{4}, \frac{8}{3}\right) = (3.2, 3.14, 3, 2.66)$$

By considering the items in decreasing order of these ratios, we can fill the knapsack as follows:

- We first include item 1 (with ratio 3.2), which has a weight of 5 and a value of 16. The remaining capacity is $14 - 5 = 9$.
- Next, we include item 2 (with ratio 3.14), which has a weight of 7 and a value of 22. The remaining capacity is $9 - 7 = 2$.
- Item 3 (with ratio 3) cannot be included because its weight (4) exceeds the remaining capacity (2).
- Item 4 (with ratio 2.66) cannot be included because its weight (3) exceeds the remaining capacity (2).

Thus, the initial integer-feasible solution is $x = (1, 1, 0, 0)$, with a total value of $\text{LB} = 16 + 22 = 38$. We set this solution as the incumbent, yielding an initial global lower bound of $\text{LB}^* = 38$.

8.2.2 Step 2: Initialize search tree

The second step is to initialize the search tree with the original problem as the root node. This node represents the entire feasible region of the problem without any additional constraints. We will denote this node as node 1, and it will be

the first node to be processed in our B&B algorithm. This node is added to the active node list, which will keep track of all the nodes that we need to explore during the search process. Figure 8.3 presents the initial B&B tree, with the current incumbent value (LB*), the upper bound (UB), and the active node list.

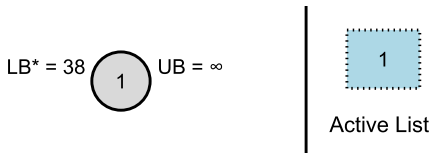


Figure 8.3: B&B tree for BKP after initializing the root node.

8.2.3 Steps 3 to 10: B&B main loop

Since the active node list is not empty, we enter the main loop of the B&B algorithm. We start by selecting the root node (node 1) from the active list to process. This is the first iteration of our search, and we will evaluate this node to determine whether it can be pruned or if we need to branch further.

Iteration 1) processing node 1

To compute an upper bound for node 1 (UB_1), we solve the Linear Programming (LP) relaxation of the BKP model, which yields the following upper bound and solution:

$$\text{UB}_1 = 44 \quad | \quad x = (1, 1, 0.5, 0)$$

Since $\text{UB}_1 >$ LB* and the solution to the LP relaxation is not integer-feasible, node 1 cannot be pruned. This means that there is still potential for finding a better solution in this branch of the search tree, and we need to branch further to explore the search space. Two choices must be made:

1. How to do the tree search? In this case, a **Depth-First Search** (DFS) strategy with pre-order traversal is adopted [84].
2. How to do the branching? In this case, we chose to **branch on the variable with a fractional value**.

We then branch based on the variable x_3, creating two new subproblems: one with the constraint $x_3 = 0$ (node 2) and another with the constraint $x_3 = 1$ (node 3). These nodes are added to the active list, and we will process them in the next iterations of the B&B algorithm.

Figure 8.4 presents the obtained B&B tree. For each node, the figure shows the best incumbent value found so far (LB*), the upper bound obtained by LP relax-

ation (UB), the solution of the LP relaxation. The branching decision is presented as edge labels, and the current Active Node List is presented as a stack.

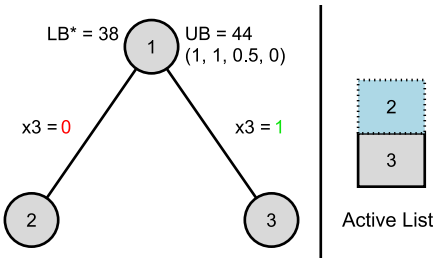


Figure 8.4: B&B tree for BKP after evaluating node 1.

Iteration 2) processing node 2

Node 2 corresponds to the subproblem obtained by fixing $x_3 = 0$. Solving its LP relaxation yields:

$$\text{UB}_2 = 43.33 \quad | \quad x = (1, 1, \mathbf{0}, 0.67)$$

Since $\text{UB}_2 > \text{LB}^*$, node 2 cannot be pruned. Branching is therefore performed on the fractional variable x_4, generating node 4 ($x_4 = 0$) and node 5 ($x_4 = 1$). The resulting search tree is shown in Figure 8.5.

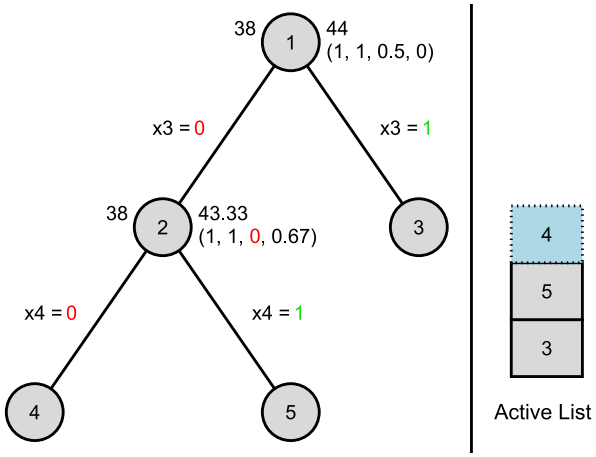


Figure 8.5: B&B tree for BKP after evaluating node 2.

Iteration 3) processing node 4

Node 4 corresponds to the subproblem obtained by fixing $x_3 = 0$ and $x_4 = 0$. Solving its LP relaxation yields:

$$\text{UB}_4 = 38 \quad | \quad x = (1, 1, \mathbf{0}, \mathbf{0})$$

Since $\text{UB}_4 \leq \text{LB}^*$, node 4 can be pruned. The resulting search tree is shown in Figure 8.6.

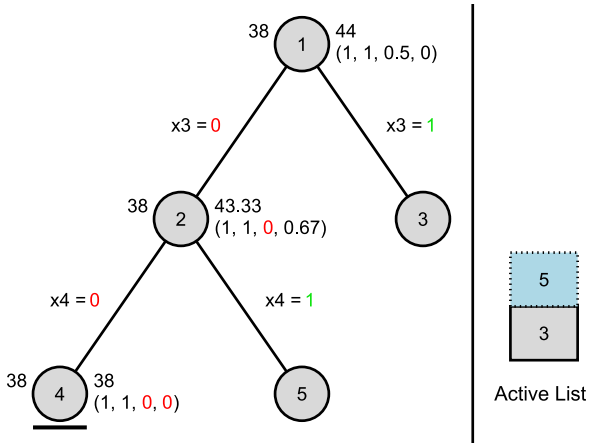


Figure 8.6: B&B tree for BKP after evaluating node 4.

Iteration 4) processing node 5

Node 5 corresponds to the subproblem obtained by fixing $x_3 = 0$ and $x_4 = 1$. Solving its LP relaxation yields:

$$\mathrm{UB}_5 = 42.86 \quad | \quad x = (1, 0.86, \mathbf{0}, \mathbf{1})$$

Since $\mathrm{UB}_5 > \mathrm{LB}^*$, node 5 cannot be pruned. Branching is therefore performed on the fractional variable x_2, generating node 6 ($x_2 = 0$) and node 7 ($x_2 = 1$). The resulting search tree is shown in Figure 8.7.

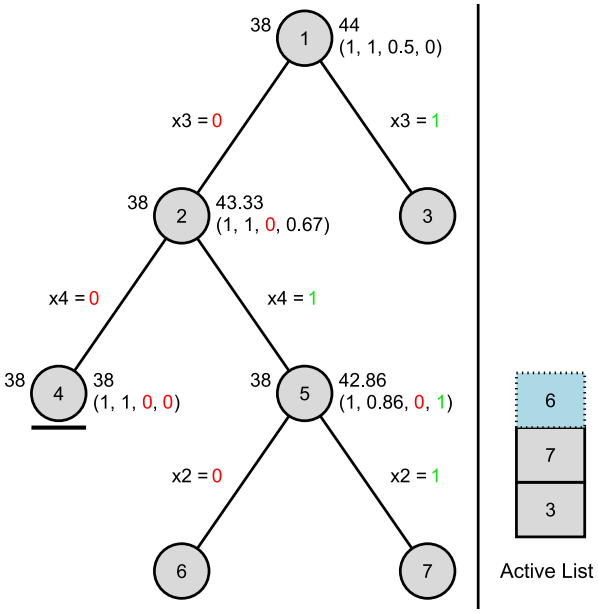


Figure 8.7: B&B tree for BKP after evaluating node 5.

Iteration 5) processing node 6

Node 6 corresponds to the subproblem obtained by fixing $x_3 = 0$, $x_4 = 1$, and $x_2 = 0$. Solving its LP relaxation yields:

$$\mathrm{UB}_6 = 24 \quad | \quad x = (1, 0, 0, 1)$$

Since $\mathrm{UB}_6 \leq \mathrm{LB}^*$, node 6 can be pruned. The resulting search tree is shown in Figure 8.8.

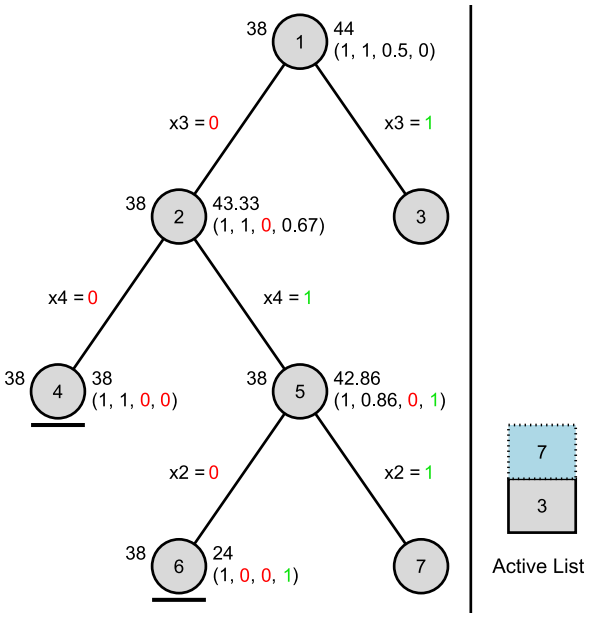


Figure 8.8: B&B tree for BKP after evaluating node 6.

Iteration 6) processing node 7

Node 7 corresponds to the subproblem obtained by fixing $x_3 = 0$, $x_4 = 1$, and $x_2 = 1$. Solving its LP relaxation yields:

$$\mathrm{UB}_7 = 42.8 \quad | \quad x = (0.8, 1, 0, 1)$$

Since $\mathrm{UB}_7 > \mathrm{LB}^*$, node 7 cannot be pruned. Branching is therefore performed on the fractional variable x_1, generating node 8 ($x_1 = 0$) and node 9 ($x_1 = 1$). The resulting search tree is shown in Figure 8.9.

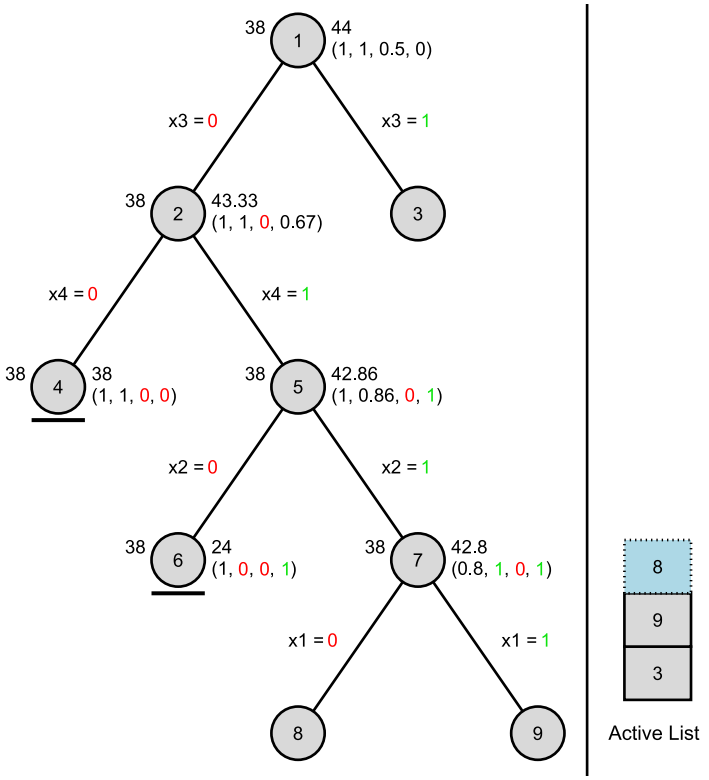


Figure 8.9: B&B tree for BKP after evaluating node 7.

Iteration 7) processing node 8

Node 8 corresponds to the subproblem obtained by fixing $x_3 = 0$, $x_4 = 1$, $x_2 = 1$, and $x_1 = 0$. Solving its LP relaxation yields:

$$\text{UB}_8 = 30 \quad | \quad x = (\mathbf{0}, \mathbf{1}, \mathbf{0}, \mathbf{1})$$

Since $\text{UB}_8 \leq \text{LB}^*$, node 8 can be pruned. The resulting search tree is shown in Figure 8.10.

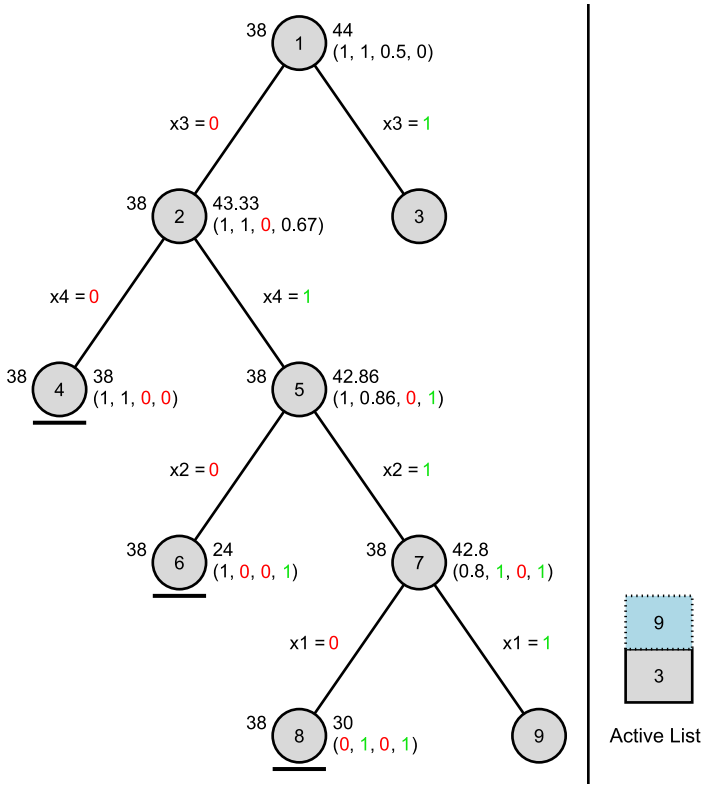


Figure 8.10: B&B tree for BKP after evaluating node 8.

Iteration 8) processing node 9

Node 9 corresponds to the subproblem obtained by fixing $x_3 = 0$, $x_4 = 1$, $x_2 = 1$, and $x_1 = 1$. Solving its LP relaxation yields:

$$\text{UB}_9 = 46 \quad | \quad x = (\mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1})$$

This solution, however, is infeasible because it violates the capacity constraint of the knapsack since $5(1) + 7(1) + 4(0) + 3(1) = 15$ which is greater than the knapsack capacity of 14. Therefore, we can prune this node, resulting in the search tree in Figure 8.11.

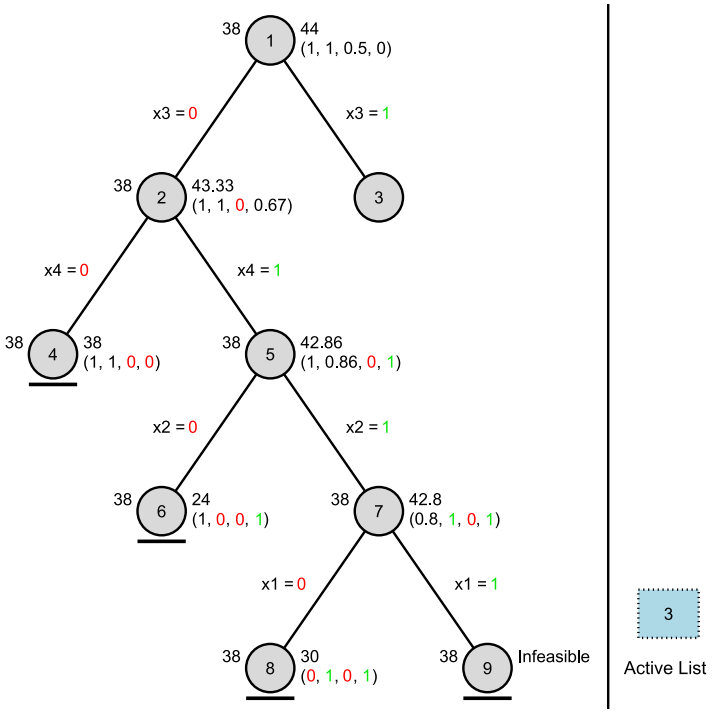


Figure 8.11: B&B tree for BKP after evaluating node 9.

Iteration 9) processing node 3

Node 3 corresponds to the subproblem obtained by fixing $x_3 = 1$. Solving its LP relaxation yields:

$$\mathrm{UB}_3 = 43.71 \quad | \quad x = (1, 0.71, \mathbf{1}, 0)$$

Since $\mathrm{UB}_3 > \mathrm{LB}^*$, node 3 cannot be pruned. Branching is therefore performed on the fractional variable x_2, generating node 10 ($x_2 = 0$) and node 11 ($x_2 = 1$). The resulting search tree is shown in Figure 8.12.

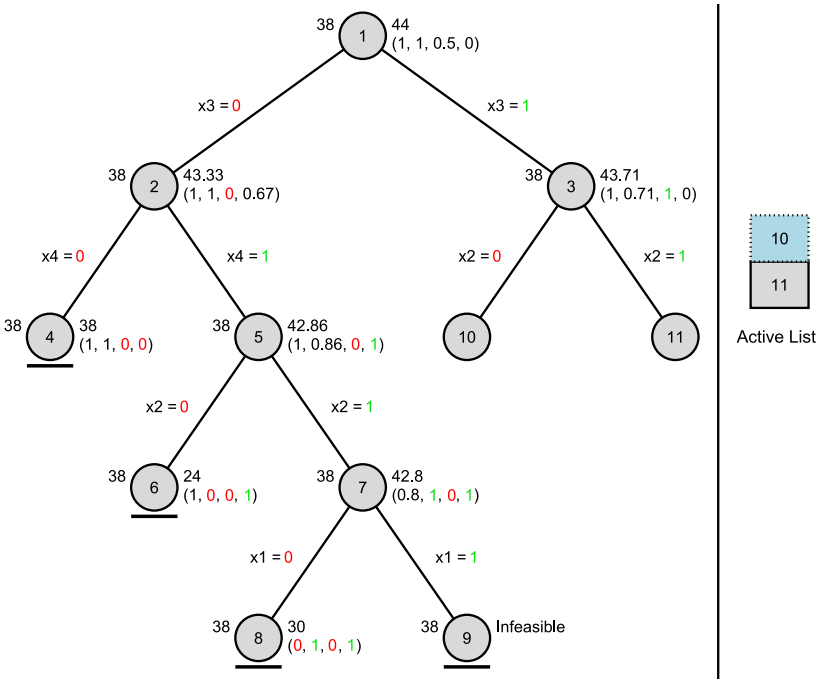


Figure 8.12: B&B tree for BKP after evaluating node 3.

Iteration 10) processing node 10

Node 10 corresponds to the subproblem obtained by fixing $x_3 = 1$ and $x_2 = 0$. Solving its LP relaxation yields:

$$\text{UB}_{10} = 36 \quad | \quad x = (1, \mathbf{0}, \mathbf{1}, 1)$$

Since $\text{UB}_{10} < \text{LB}^*$, node 10 can be pruned. The resulting search tree is shown in Figure 8.13.

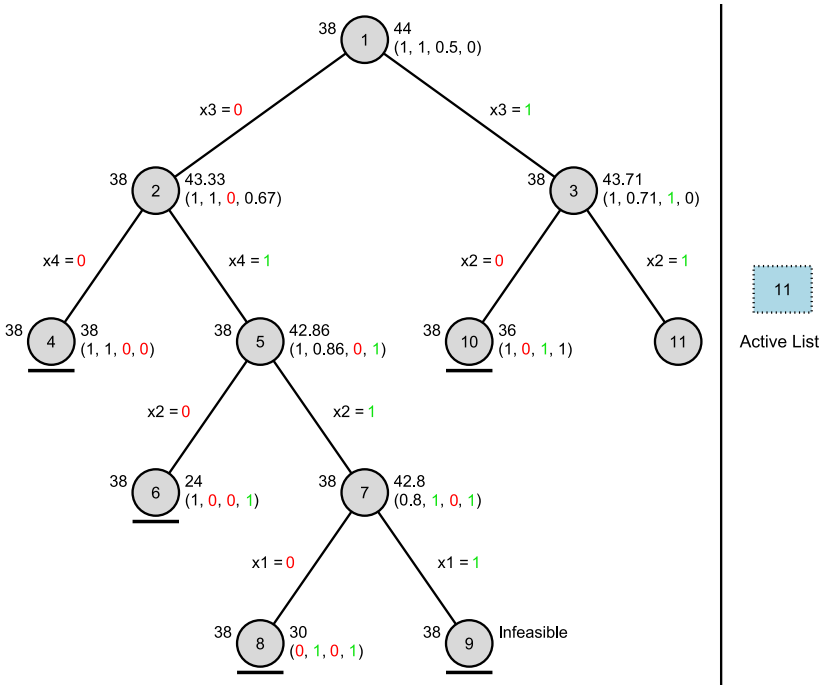


Figure 8.13: B&B tree for BKP after evaluating node 10.

Iteration 11) processing node 11

Node 11 corresponds to the subproblem obtained by fixing $x_3 = 1$ and $x_2 = 1$. Solving its LP relaxation yields:

$$\text{UB}_{11} = 36 \quad | \quad x = (0.6, \mathbf{1}, \mathbf{1}, 0)$$

Since $\text{UB}_{11} > \text{LB}^*$, node 11 cannot be pruned. Branching is therefore performed on the fractional variable x_1, generating node 12 ($x_1 = 0$) and node 13 ($x_1 = 1$). The resulting search tree is shown in Figure 8.14.

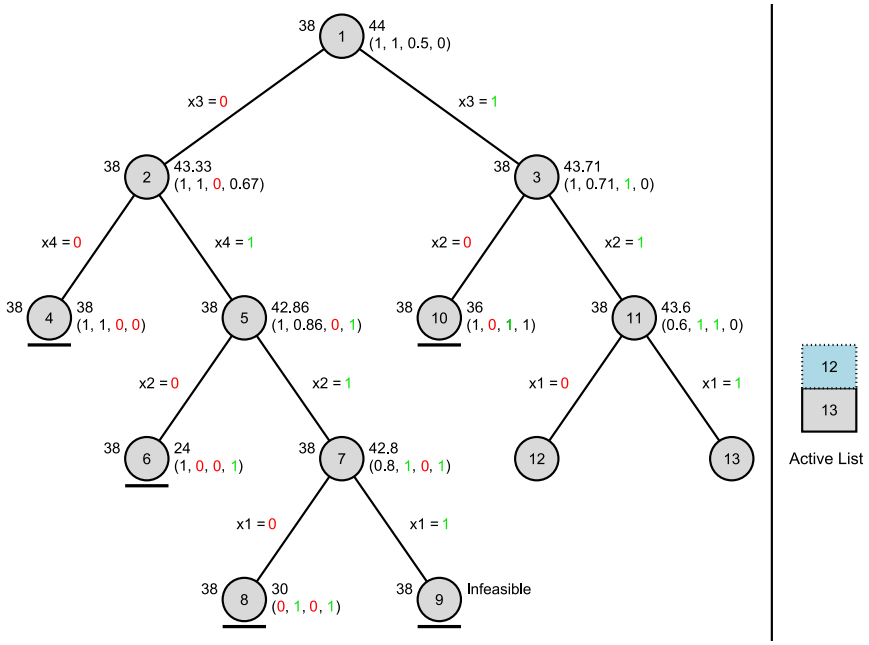


Figure 8.14: B&B tree for BKP after evaluating node 11.

Iteration 12) processing node 12

Node 12 corresponds to the subproblem obtained by fixing $x_3 = 1$, $x_2 = 1$, and $x_1 = 0$. Solving its LP relaxation yields:

$$\text{UB}_{12} = 42 \quad | \quad x = (\mathbf{0}, \mathbf{1}, \mathbf{1}, 1)$$

Since UB_{12} is integral, feasible and greater than our incumbent solution (LB*), we can update our incumbent solution to $\text{LB}^* = 42$ and prune this node. The resulting search tree is shown in Figure 8.15.

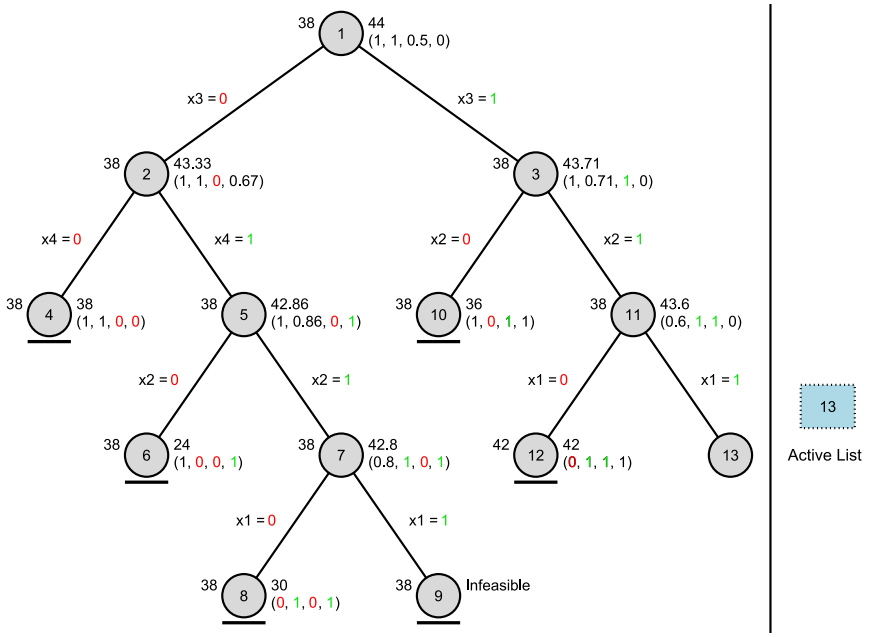


Figure 8.15: B&B tree for BKP after evaluating node 12.

Iteration 13) processing node 13

Finally, node 13 corresponds to the subproblem obtained by fixing $x_3 = 1$, $x_2 = 1$, and $x_1 = 1$. Solving its LP relaxation yields:

$$\text{UB}_{13} = 50 \quad | \quad x = (\mathbf{1}, \mathbf{1}, \mathbf{1}, 0)$$

This solution, however, is infeasible because it violates the capacity constraint of the knapsack since $5(1) + 7(1) + 4(1) + 3(0) = 16$, which exceeds the knapsack capacity of 14. Therefore, we can prune this node and since there are no more nodes in the active list, the B&B algorithm can terminate. The resulting search tree is shown in Figure 8.16, where the node that found the best solution (node 12) is highlighted in green.

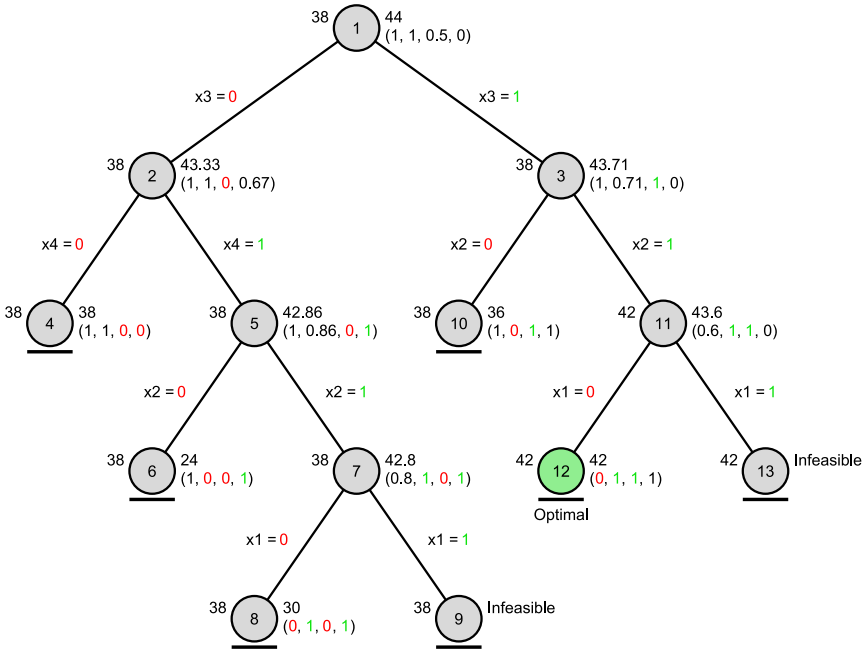


Figure 8.16: B&B tree for BKP after evaluating node 13.

> **Note**
>
> This example highlights a crucial aspect of the B&B method: the importance of the incumbent solution. Notice that our initial feasible solution of 38 was not improved upon until node 12, which occurred late in the search.
>
> A more effective approach is often to use strong heuristics specifically designed to find high-quality integer solutions during the search. This

helps the pruning process and can dramatically reduce the number of nodes that need to be explored.

8.3 Minimization example: Traveling Salesman Problem

Figure 8.17 illustrates the bounding process for a minimization problem, where the relaxation provides a lower bound and the feasible solution (incumbent) provides an upper bound. These bounds influence the tree search: the tighter they are, the more frequently they enable pruning.

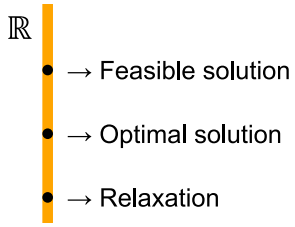


Figure 8.17: Bounds for a minimization problem.

Let us now examine a branch-and-bound method for the Traveling Salesman Problem (TSP), which was proposed in [85]. The idea focuses on solving the assignment problem in the traveling salesman formulation and breaking subtours through the branch-and-bound tree.

For example, if after solving the assignment problem the following subtour was identified: $1 \to 2 \to 3 \to 1$ (see Figure 8.18). Then you can create the tree nodes by forcing $x_{1,2} = 0$, $x_{2,3} = 0$ and $x_{3,1} = 0$. New subtours may appear on the new nodes and branching will be needed to break them. The tree is then formed up to the level until a salesman solution appears. Other solutions may appear and the tree must be traversed with free nodes until the desired optimal solution.

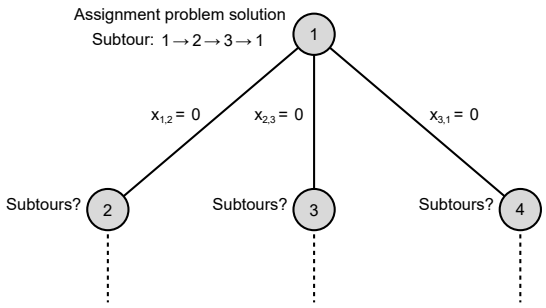


Figure 8.18: Branching strategy for the TSP.

In the next subsections, we will examine a B&B procedure for the TSP considering the following example.

Example 8.2 Consider the TSP instance shown in Figure 8.19, composed by $n = 7$ cities.

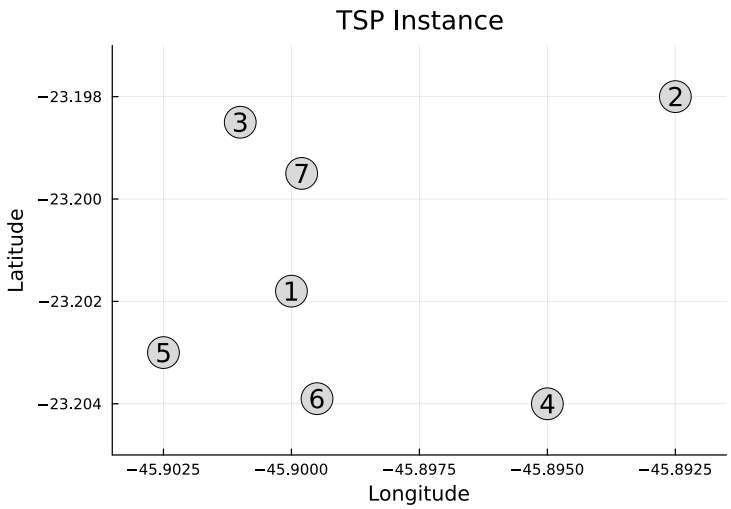


Figure 8.19: TSP instance with 7 cities.

To control the search process, we make the following decisions:

1. Depth-First Search with pre-order traversal as the search strategy;
2. Solve the assignment problem relaxation at each node to obtain bounds and guide the branching process;
3. Branch on the subtours identified in the solution of the assignment problem relaxation, by adding constraints to break these subtours in the child nodes;
4. Use the cost of the best integer-feasible solution found so far (incumbent) as an upper bound to prune nodes whose relaxation solution has a cost greater than or equal to this bound;
5. Continue the process until there are no more active nodes to explore, at which point the best incumbent solution found is guaranteed to be the optimal tour for the TSP instance.

8.3.1 Step 1: Initialize incumbent solution

To find an initial integer-feasible solution for the TSP, we can use the Multi-Start Nearest Neighbor heuristic with 2-opt local search, which is a simple and efficient method for constructing a tour. It consists of two main components:

1. Nearest Neighbor (NN) [86] — to quickly construct a feasible tour

- Start from an arbitrary city
- Choose the nearest unvisited city as the next city in the tour
- Appends to the tour
- Continues until all cities have been visited

2. 2-opt local search [87] — to improve the initial solution by removing inefficient edge crossings.
   - Select two edges in the route
   - Remove them
   - Reconnect the route by reversing the intermediate segment
   - Accept the change if it results in a shorter route

The heuristic can also use a multi-start strategy, meaning that the construction phase is repeated from every possible starting city, and the best solution found is retained as the incumbent solution.

The following incumbent solution is obtained $1 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 1$, with total cost of $\text{UB}^* = 3076.96$ meters. This solution provides an initial upper bound for the optimal tour cost, which will be used in the B&B algorithm to prune branches of the search tree that cannot yield a better solution.

8.3.2 Step 2: Initialize search tree

The second step is to initialize the search tree with the original problem as the root node (node 1) and add it to the active node list. This node represents the assignment problem relaxation of the TSP without any additional constraints.

8.3.3 Steps 3 to 10: B&B main loop

Since the active node list is not empty, we enter the main loop of the B&B algorithm. We start by selecting the root node (node 1) from the active list to process.

Iteration 1) processing node 1

Solving the assignment problem relaxation at node 1 yields a lower bound of $\text{LB}_1 = 2604.29$ meters and a solution composed by three subtours (Figure 8.20).

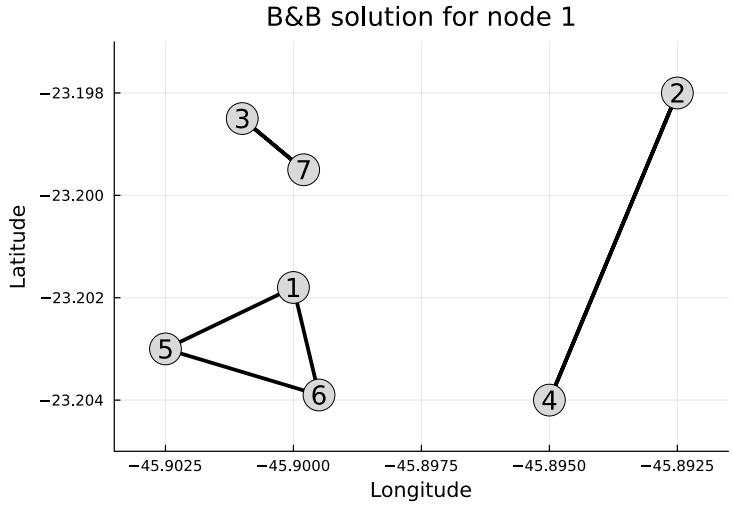


Figure 8.20: Assignment problem relaxation solution at node 1.

Since $\text{LB}_1 < \text{UB}^*$, node 1 cannot be pruned. Branching is therefore performed on the identified subtours. Any of the three subtours can be chosen. Let's choose the subtour $2 \to 4 \to 2$ to proceed. The resulting search tree is shown in Figure 8.21.

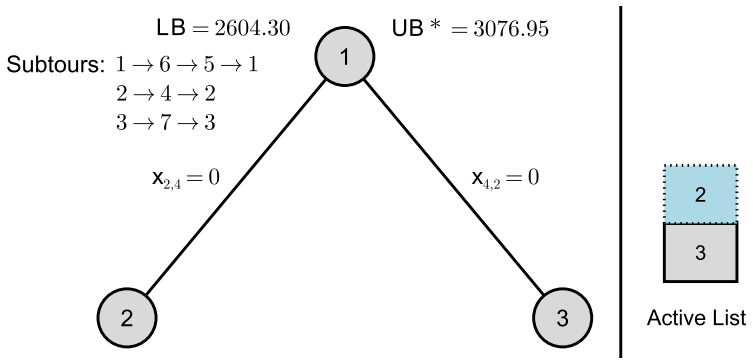


Figure 8.21: B&B tree for TSP after evaluating node 1.

Iteration 2) processing node 2

Node 2 corresponds to the assigned problem relaxation with constraint $x_{2,4} = 0$. Solving it yields a lower bound of $\text{LB}_2 = 2990.44$ meters and a solution composed by two subtours (Figure 8.22).

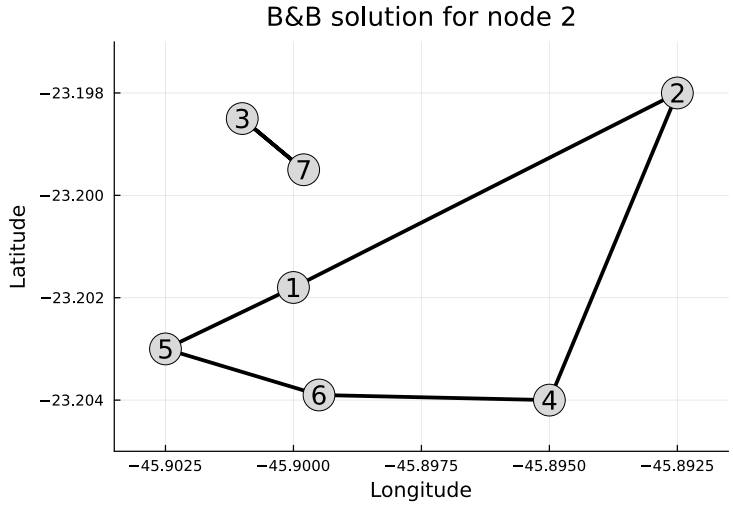


Figure 8.22: Assignment problem relaxation solution at node 2.

Since $\text{LB}_2 < \text{UB}^*$, node 2 cannot be pruned. Branching is therefore performed on the identified subtours. Let's choose the subtour $3 \to 7 \to 3$ to proceed. The resulting search tree is shown in Figure 8.23.

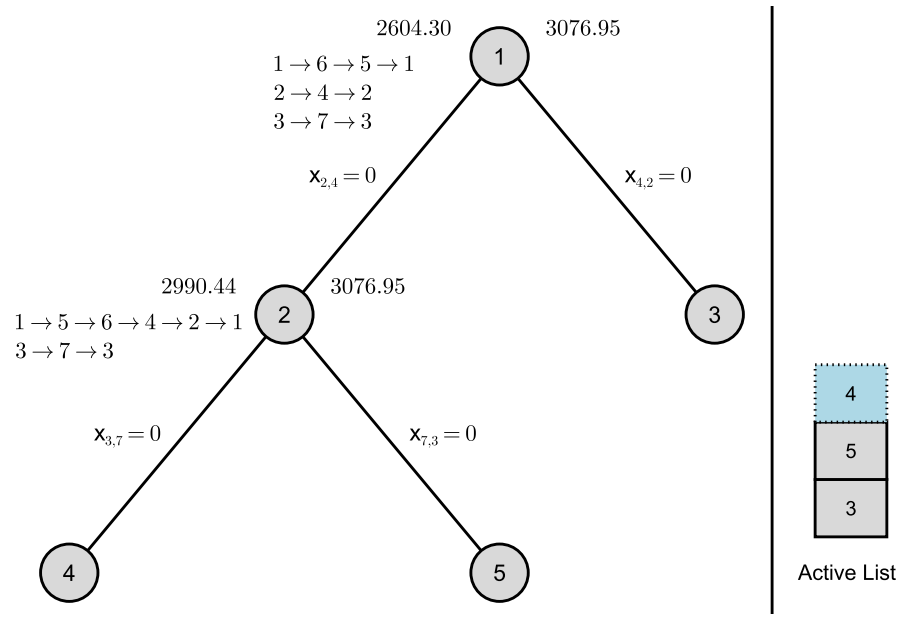


Figure 8.23: B&B tree for TSP after evaluating node 2.

Iteration 3) processing node 4

Node 4 corresponds to the assigned problem relaxation with constraints $x_{2,4} = 0$ and $x_{3,7} = 0$. Solving it yields a lower bound of $\text{LB}_4 = 3094.57$ meters and a solution composed by a single tour (Figure 8.24).

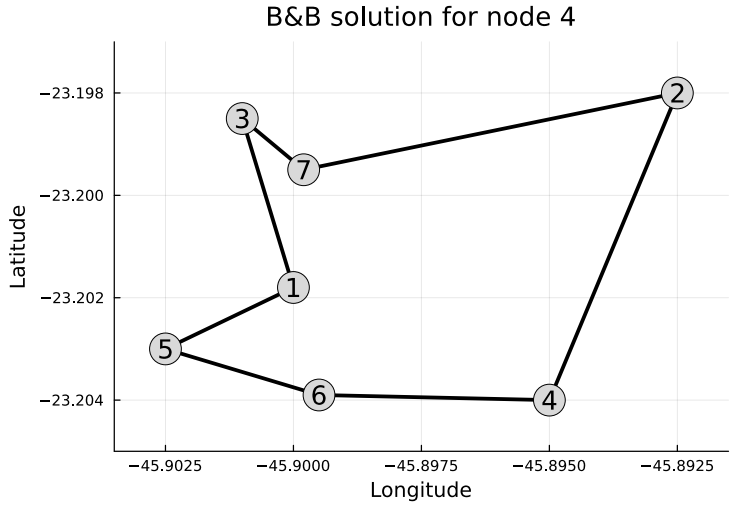


Figure 8.24: Assignment problem relaxation solution at node 4.

Since $\text{LB}_4 > \text{UB}^*$, node 4 can be pruned. The resulting search tree is shown in Figure 8.25.

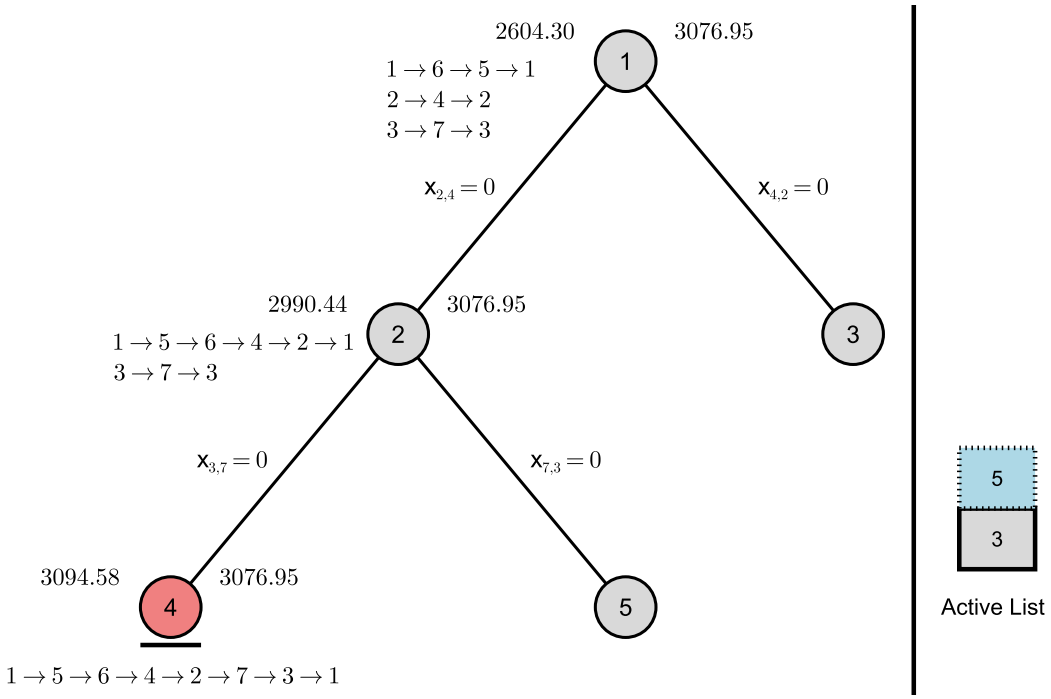


Figure 8.25: B&B tree for TSP after evaluating node 4.

Iteration 4) processing node 5

Node 5 corresponds to the assigned problem relaxation with constraints $x_{2,4} = 0$ and $x_{7,3} = 0$. Solving it yields a lower bound of $\text{LB}_5 = 3076.96$ meters and a solution composed by a single tour (Figure 8.26).

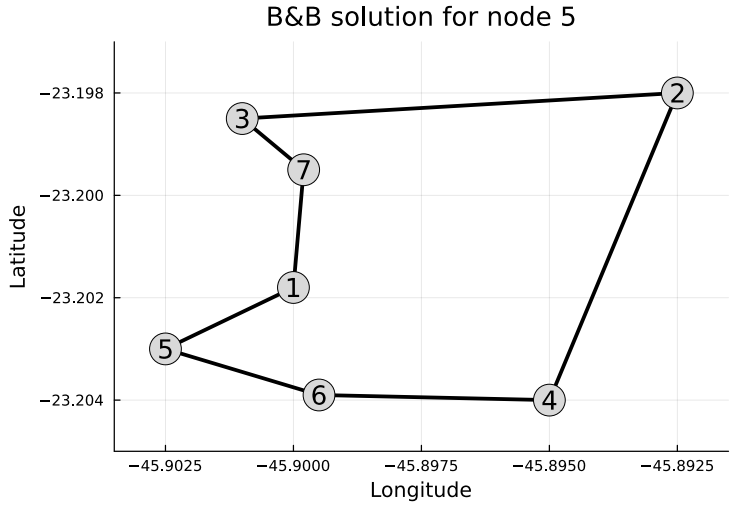


Figure 8.26: Assignment problem relaxation solution at node 5.

Since $\text{LB}_5 = \text{UB}^*$, node 5 can be pruned. The resulting search tree is shown in Figure 8.27.

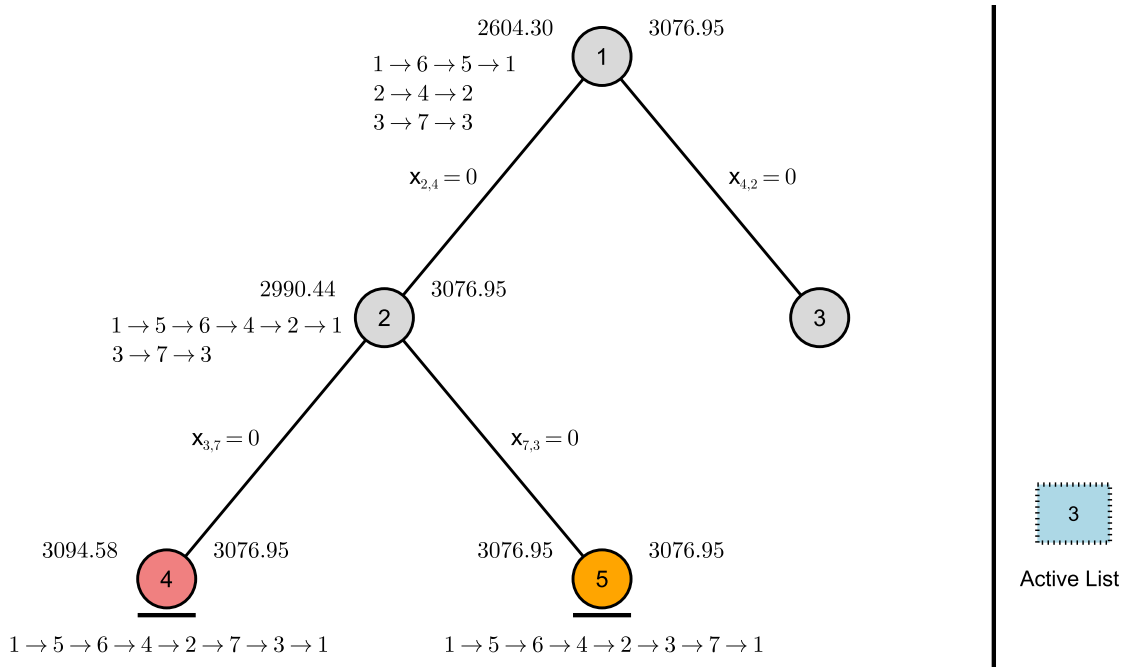


Figure 8.27: B&B tree for TSP after evaluating node 5.

Iteration 5) processing node 3

Node 3 corresponds to the assigned problem relaxation with constraint $x_{4,2} = 0$. Solving it yields a lower bound of $\text{LB}_3 = 2990.44$ meters and a solution composed by two subtours (Figure 8.28).

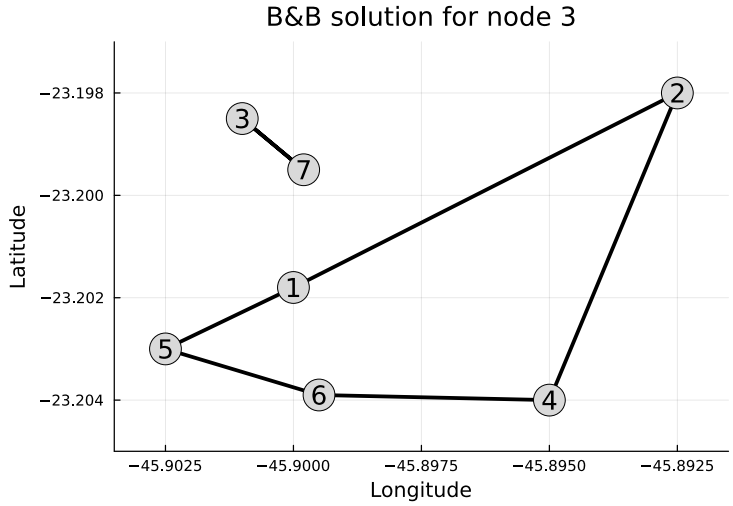


Figure 8.28: Assignment problem relaxation solution at node 3.

Since $\text{LB}_3 < \text{UB}^*$, node 3 cannot be pruned. Branching is therefore performed on the identified subtours. Let's choose the subtour $1 \to 5 \to 1$ to proceed. The resulting search tree is shown in Figure 8.29.

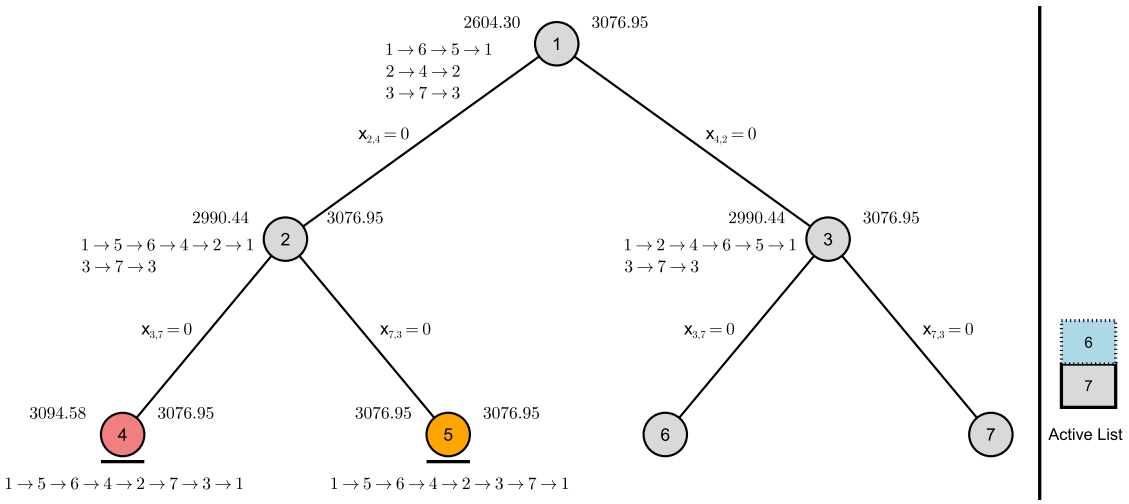


Figure 8.29: B&B tree for TSP after evaluating node 3.

Iteration 6) processing node 6

Node 6 corresponds to the assigned problem relaxation with constraints $x_{4,2} = 0$ and $x_{3,7} = 0$. Solving it yields a lower bound of $\text{LB}_6 = 3076.95$ meters and a solution composed by a single tour (Figure 8.30).

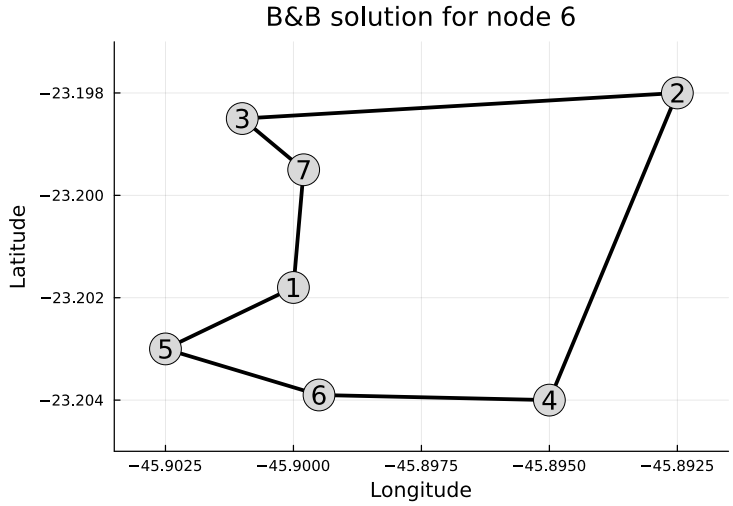


Figure 8.30: Assignment problem relaxation solution at node 6.

Since $\text{LB}_6 = \text{UB}^*$, node 6 can be pruned. The resulting search tree is shown in Figure 8.31.

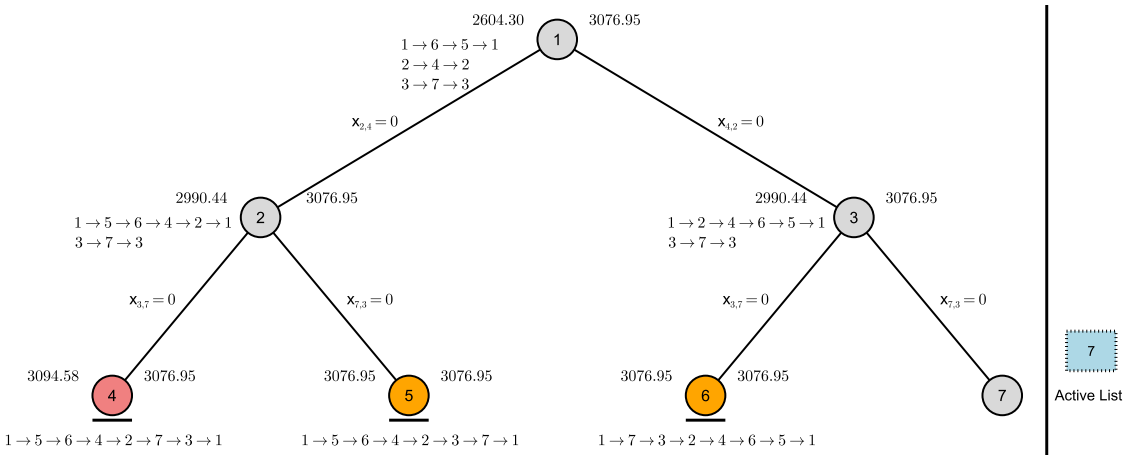


Figure 8.31: B&B tree for TSP after evaluating node 6.

Iteration 7) processing node 7

Node 7 corresponds to the assigned problem relaxation with constraints $x_{4,2} = 0$ and $x_{7,3} = 0$. Solving it yields a lower bound of $\text{LB}_7 = 3094.58$ meters and a solution composed by a single tour (Figure 8.32).

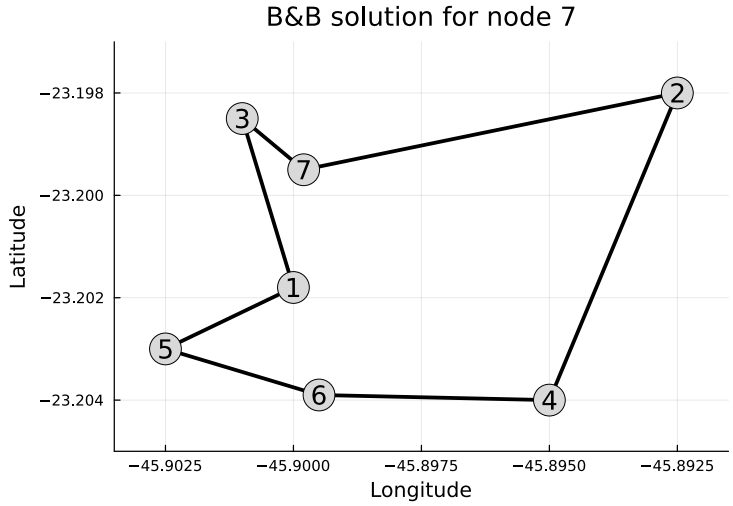


Figure 8.32: Assignment problem relaxation solution at node 7.

Since $LB_7 > UB^*$, node 7 can be pruned. The resulting search tree is shown in Figure 8.33. As the active node list is now empty, the B&B algorithm can terminate, and the best incumbent solution found is guaranteed to be the optimal tour for this TSP instance.

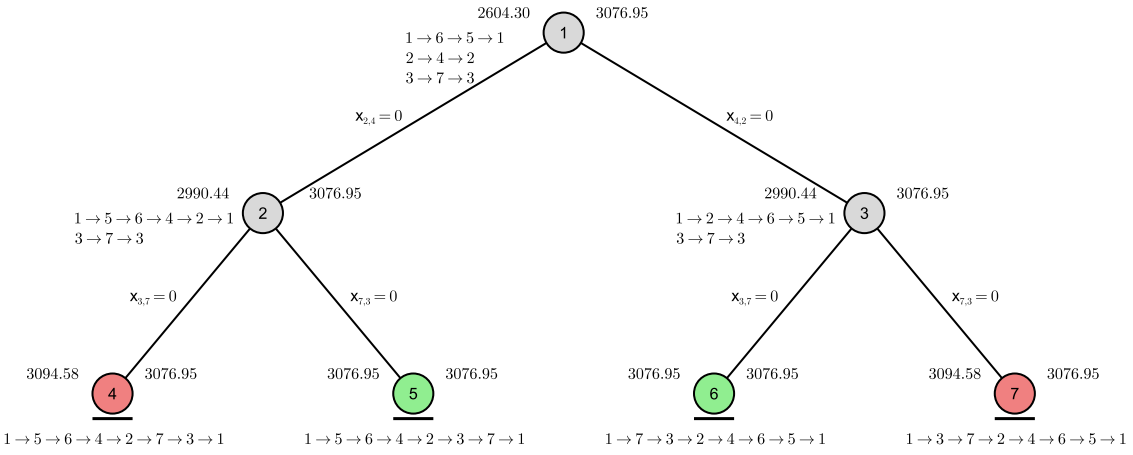


Figure 8.33: B&B tree for TSP after evaluating node 7.

> **Note**
>
> Note that there are two optimal solutions, as the distance matrix is symmetric. The solutions are inverses of each other (Nodes 5 and 6).

8.4 Final remarks

The branch-and-bound technique is one of the most fundamental and widely used methods for solving Integer Programming (IP) problems. Over the years, it has been refined and extended to tackle larger and more complex problems. Two prominent extensions are the branch and cut and the branch and price methods, which have significantly improved the state-of-the-art in solving IP problems [6].

The branch and cut method combines the branch-and-bound algorithm with cutting planes. Cutting planes are valid inequalities that strengthen the LP relaxation of the IP problem, providing tighter bounds on the optimal solution. By adding cutting planes at each node of the branch-and-bound search tree, the branch and cut method can eliminate non-promising regions of the search space and converge to the optimal solution more quickly.

On the other hand, the branch and price method combines branch-and-bound with column generation (or pricing). As we discussed earlier, column generation is a powerful technique for solving large-scale IP problems. By generating columns (variables) at each node of the search tree, the branch and price method can effectively explore the search space and find high-quality solutions.

Both branch and cut and branch and price have been successfully applied to a wide range of optimization problems, including vehicle routing, network design, cutting and packing, and many others. They have become indispensable tools in the field of operations research and combinatorial optimization.

The exercises at the end of the chapter will provide valuable hands-on experience and help deepen the understanding of the branch-and-bound technique. By solving these exercises, readers will gain practical insights into how to formulate and solve combinatorial optimization problems.

8.5 Exercises

Formulate and solve the following exercises using the JuMP package.

Exercise 8.1 Solve the following Binary Knapsack Problem instance using the Branch-and-Bound technique.

$$v = (8, 16, 20, 12, 6, 10, 4) \quad w = (3, 7, 9, 6, 3, 5, 2) \quad W = 17$$

a) Solving using Branch-and-Bound with DFS search strategy.

b) Solve using Branch-and-Bound with Best Bound search strategy.

c) Compare the results obtained in exercises (a) and (b). Is there an advantage of using the Best Bound search strategy instead of the DFS for this instance of the BKP?

Exercise 8.2 Solve the following Traveling Salesman Problem instance using the Branch-and-Bound technique. Use the dataset provided in the file tsp_instance.csv as input.

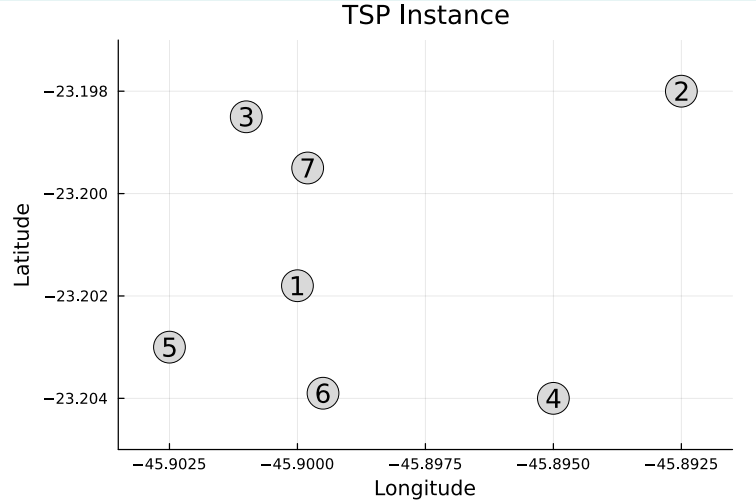


Figure 8.34: TSP instance for the Branch-and-Bound exercise.

Exercise 8.3 Solve the Maximum Independent Set Problem (MISP) using the Branch-and-Bound technique. Use the dataset icosahedral.csv as input. The following image illustrates the graph instance.

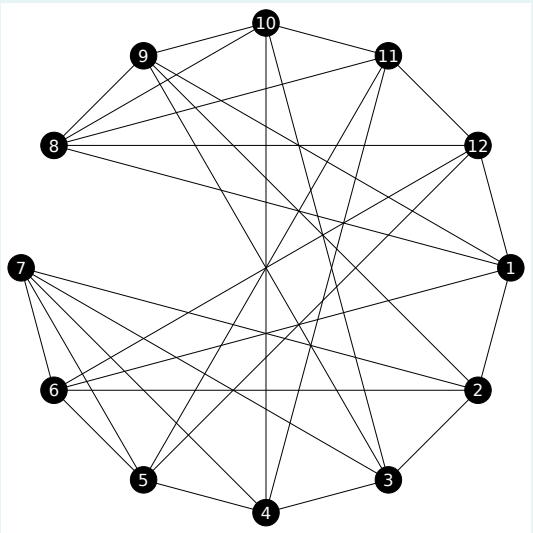


Figure 8.35: MISP instance for the Branch-and-Bound exercise.

a) Solve the MISP using Branch-and-Bound with Best Bound search strategy.

b) Solve the MISP using Branch-and-Cut technique.

c) Compare the results obtained in exercises (a) and (b). Is there an advantage of using the Branch-and-Cut technique?

Bibliography

[1] M. Lubin, O. Dowson, J. Dias Garcia, J. Huchette, B. Legat, and J. P. Vielma, "JuMP 1.0: Recent improvements to a modeling language for mathematical optimization," *Mathematical Programming Computation*, vol. 15, pp. 581–589, 2023, doi: 10.1007/s12532-023-00239-3.

[2] B. Lauwens and A. B. Downey, *Think Julia: how to think like a computer scientist*. O'Reilly Media, 2019.

[3] C. Kwon, *Julia programming for operations research*. Self-published, 2019. [Online]. Available: http://j4.io/

[4] N. Kalicharan, *Julia—bit by bit: programming for beginners*. Springer, 2021.

[5] G. B. Dantzig, *Linear programming and extensions*. Princeton University Press, 2016.

[6] L. A. Wolsey, *Integer programming*, 2nd ed. John Wiley & Sons, 2020.

[7] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*, 2nd ed. Courier Corporation, 1998.

[8] H. P. Williams, *Model building in mathematical programming*, 5th ed. John Wiley & Sons, 2013.

[9] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[10] G. B. Dantzig, "Discrete-variable extremum problems," *Operations Research*, vol. 5, no. 2, pp. 266–288, 1957, doi: 10.1287/opre.5.2.266.

[11] M. S. Daskin, *Network and discrete location: models, algorithms, and applications*. Hoboken, NJ: John Wiley & Sons, 2011.

[12] S. L. Hakimi, "Optimum locations of switching centers and the absolute centers and medians of a graph," *Operations Research*, vol. 12, no. 3, pp. 450–459, 1964, doi: 10.1287/opre.12.3.450.

[13] S. L. Hakimi, "Optimum distribution of switching centers in a communication network and some related graph theoretic problems," *Operations Research*, vol. 13, no. 3, pp. 462–475, 1965, doi: 10.1287/opre.13.3.462.

[14] C. S. ReVelle and R. W. Swain, "Central facilities location," *Geographical Analysis*, vol. 2, no. 1, pp. 30–42, 1970, doi: 10.1111/j.1538-4632.1970.tb00140.x.

[15] V. Marianov and D. Serra, "Median problems in networks," in *Foundations of location analysis*, New York, NY: Springer, 2011, pp. 39–59. doi: 10.1007/978-1-4419-7572-0_2.

[16] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*. in Series of Books in the Mathematical Sciences. San Francisco: W. H. Freeman, Company, 1979.

[17] M. B. Teitz and P. Bart, "Heuristic methods for estimating the generalized vertex median of a weighted graph," *Operations Research*, vol. 16, no. 5, pp. 955–961, 1968, doi: 10.1287/opre.16.5.955.

[18] M. G. Resende and R. F. Werneck, "A hybrid heuristic for the p-median problem," *Journal of Heuristics*, vol. 10, no. 1, pp. 59–88, 2004, doi: 10.1023/B:HEUR.0000026258.33459.df.

[19] W. Mu and D. Tong, "A spatial-knowledge-enhanced heuristic for solving the p-median problem," *Transactions in GIS*, vol. 22, no. 2, pp. 477–493, 2018, doi: 10.1111/tgis.12313.

[20] M. L. Fisher, "The Lagrangian relaxation method for solving integer programming problems," *Management Science*, vol. 50, no. 12, pp. 1861–1871, 2004, doi: 10.1287/mnsc.1040.0263.

[21] M. Guignard, "Lagrangean relaxation," *Top*, vol. 11, no. 2, pp. 151–200, 2003, doi: 10.1007/BF02579036.

[22] E. L. Senne and L. A. Lorena, "Lagrangean/surrogate heuristics for p-median problems," pp. 115–130, 2000, doi: 10.1007/978-1-4613-0307-1_7.

[23] L. A. Lorena and E. L. Senne, "A column generation approach to capacitated p-median problems," *Computers & Operations Research*, vol. 31, no. 6, pp. 863–876, 2004, doi: 10.1016/S0305-0548(03)00028-0.

[24] E. L. Senne, L. A. Lorena, and M. A. Pereira, "A branch-and-price approach to p-median location problems," *Computers & Operations Research*, vol. 32, no. 6, pp. 1655–1664, 2005, doi: 10.1016/j.cor.2003.12.008.

[25] N. D. Pizzolato, F. B. Barcelos, and L. A. Nogueira Lorena, "School location methodology in urban areas of developing countries," *International Transactions in Operational Research*, vol. 11, no. 6, pp. 667–681, 2004, doi: 10.1111/j.1475-3995.2004.00480.x.

[26] C. Vasic and B. Predic, "Using GIS in dynamic relocation of emergency ambulance vehicles," *International Journal of Research and Reviews in Computer Science*, vol. 2, no. 1, p. 211, 2011.

[27] M. J. Widener, "Modelling Hurricane disaster relief distribution with a hierarchical capacitated-median model: An analysis with extensions," 2009.

[28] J. V. L. Silva and N. F. dos Santos, "Location of ecopoints through integration p-medians model with geographical information system," *Gestão & Produção*, vol. 27, 2020, doi: 10.1590/0104-530x3676-20.

[29] T. Nogueira, R. Mendes, Z. Vale, and J. C. Cardoso, "Optimal location of natural gas sources in the Iberian system," in *Proceedings of the 6th WSEAS International Conference on Power Systems, Lisbon, Portugal*, 2006.

[30] A. M. Costa, P. M. França, and C. Lyra Filho, "Two-level network design with intermediate facilities: an application to electrical distribution systems," *Omega*, vol. 39, no. 1, pp. 3–13, 2011, doi: 10.1016/j.omega.2010.02.004.

[31] C. Toregas, R. Swain, C. ReVelle, and L. Bergman, "The location of emergency service facilities," *Operations Research*, vol. 19, no. 6, pp. 1363–1373, 1971, doi: 10.1287/opre.19.6.1363.

[32] R. Church and C. ReVelle, "The maximal covering location problem," in *Papers of the regional science association*, 1974, pp. 101–118. doi: 10.1007/BF02094836.

[33] J. R. Current and J. E. Storbeck, "Capacitated covering models," *Environment and Planning B: Planning and Design*, vol. 15, no. 2, pp. 153–163, 1988, doi: 10.1068/b150153.

[34] V. Marianov and D. Serra, "Probabilistic maximal covering location–allocation models for congested systems," *Journal of Regional Science*, vol. 38, no. 3, pp. 401–424, 1998, doi: 10.1111/0022-4146.00100.

[35] O. Berman, Z. Drezner, and D. Krass, "Discrete cooperative covering problems," *Journal of the Operational Research Society*, vol. 62, no. 11, pp. 2002–2012, 2011, doi: 10.1057/jors.2011.33.

[36] I. Vasilyev, A. V. Ushakov, N. Maltugueva, and A. Sforza, "An effective heuristic for large-scale fault-tolerant k-median problem," *Soft Computing*, vol. 23, no. 9, pp. 2959–2967, 2019, doi: 10.1007/s00500-018-3130-0.

[37] E. Araújo, A. Chaves, and L. Lorena, "A Mathematical Model for the Coverage Location Problem With Overlap Control," *Computers & Industrial Engineering*, p. 106548, Jun. 2020, doi: 10.1016/j.cie.2020.106548.

[38] D. Schilling, D. J. Elzinga, J. Cohon, R. Church, and C. ReVelle, "The TEAM/FLEET models for simultaneous facility and equipment siting," *Transportation Science*, vol. 13, no. 2, pp. 163–175, 1979, doi: 10.1287/trsc.13.2.163.

[39] W. J. Cook, *In pursuit of the traveling salesman: mathematics at the limits of computation*. Princeton University Press, 2012.

[40] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, 1954, doi: 10.1287/opre.2.4.393.

[41] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems," *Journal of the ACM*, vol. 7, no. 4, pp. 326–329, 1960, doi: 10.1145/321043.321046.

[42] P. Toth and D. Vigo, *Vehicle routing: problems, methods, and applications*, 2nd ed. SIAM, 2014.

[43] R. Diestel, *Graph theory*, 5th ed. Springer Berlin Heidelberg, 2017. doi: 10.1007/978-3-662-53622-3.

[44] M. Gondran, M. Minoux, and S. Vajda, *Graphs and algorithms*. John Wiley & Sons, Inc., 1984.

[45] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, Springer, 2009, pp. 219–241.

[46] M. Xiao and H. Nagamochi, "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017, doi: https://doi.org/10.1016/j.ic.2017.06.001.

[47] I. C. Paschalidis, F. Huang, and W. Lai, "A message-passing algorithm for wireless network scheduling," *IEEE/ACM Transactions on Networking*, vol. 23, no. 5, pp. 1528–1541, 2014.

[48] M. M. Daliri Khomami, A. Rezvanian, and M. R. Meybodi, "Maximum independent set in multiplex social networks and its application to influence maximization," *Scientific Reports*, vol. 15, no. 1, p. 16322, 2025.

[49] M. Buchem, L. Kleist, and D. Schmidt genannt Waldschmidt, "Scheduling with Machine Conflicts," in *Approximation and Online Algorithms*, P. Chalermsook and B. Laekhanukit, Eds., Cham: Springer International Publishing, 2022, pp. 36–60.

[50] M. Mooij and I. Kryven, "Finding large independent sets in networks using competitive dynamics," *Communications Physics*, vol. 8, no. 1, p. 235, 2025.

[51] T. Gallai, "Über extreme Punkt- und Kantenmengen," *Ann. Univ. Sci. Budapest, Eötvös Sect. Math*, vol. 2, pp. 133–138, 1959.

[52] A. Makris *et al.*, "Pro-active component image placement in Edge computing environments," *Future Generation Computer Systems*, vol. 157, pp. 344–359, 2024, doi: https://doi.org/10.1016/j.future.2024.04.007.

[53] J. Chen, R. Zhou, J. Wu, H. Zhang, and W. Gui, "Vertex cover of networks and its related optimization problems: An overview," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 7, pp. 11738–11758, 2024.

[54] L. Epstein, A. Levin, and G. J. Woeginger, "Vertex cover meets scheduling," *Algorithmica*, vol. 74, no. 3, pp. 1148–1173, 2016.

[55] U. Maulik, S. Basu, and S. Ray, "Identifying protein complexes in PPI network using non-cooperative sequential game," *Scientific reports*, vol. 7, no. 1, p. 8410, 2017.

[56] G. Chaitin, "Register allocation and spilling via graph coloring," *SIGPLAN Not.*, vol. 39, no. 4, pp. 66–74, Apr. 2004, doi: 10.1145/989393.989403.

[57] J. Riihijarvi, M. Petrova, and P. Mahonen, "Frequency allocation for WLANs using graph colouring techniques," in *Second Annual Conference on Wireless On-demand Network Systems and Services*, 2005, pp. 216–222.

[58] R. Ganguli and S. Roy, "A study on course timetable scheduling using graph coloring approach," *international journal of computational and applied mathematics*, vol. 12, no. 2, pp. 469–485, 2017.

[59] S. De, "An efficient technique of resource scheduling in cloud using graph coloring algorithm," *Global Transitions Proceedings*, vol. 3, no. 1, pp. 169–176, 2022.

[60] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: an experimental study," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1590–1603, 2018.

[61] R. Bhoot, T. Khare, M. Agarwal, S. Jaiswal, and Y. Simmhan, "Triparts: Scalable Streaming Graph Partitioning to Enhance Community Structure," *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 2992–3006, 2025.

[62] J. Paulachan, D. Onwuchekwa, and R. Obermaisser, "Incremental concurrent task scheduling in multi-cluster systems: Utilising graph partitioning and dynamic genome length genetic algorithm," *Procedia Computer Science*, vol. 257, pp. 927–936, 2025.

[63] Q. Xing *et al.*, "SNNcut: An Efficient Partitioning Method for Large-Scale Spiking Neural Networks Using Spike-Sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. , no. , p. 1, 2025, doi: 10.1109/TCAD.2025.3648616.

[64] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998, doi: 10.1137/S1064827595287997.

[65] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 26113, 2004, doi: 10.1103/PhysRevE.69.026113.

[66] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[67] M. Rostami, M. Oussalah, K. Berahmand, and V. Farrahi, "Community detection algorithms in healthcare applications: a systematic review," *IEEE Access*, vol. 11, pp. 30247–30272, 2023.

[68] A. Diboune, H. Slimani, H. Nacer, and K. Beghdad Bey, "A comprehensive survey on community detection methods and applications in complex information networks," *Social Network Analysis and Mining*, vol. 14, no. 1, p. 93, 2024.

[69] I. Manipur, M. Giordano, M. Piccirillo, S. Parashuraman, and L. Maddalena, "Community Detection in Protein-Protein Interaction Networks and Applications," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 20, no. 1, pp. 217–237, 2023, doi: 10.1109/TCBB.2021.3138142.

[70] S. Khan and S. Khan, "Latency aware graph-based microservice placement in the edge-cloud continuum," *Cluster Computing*, vol. 28, no. 2, p. 88, 2025.

[71] G. M. Ribeiro and L. A. N. Lorena, "Lagrangean relaxation with clusters for point-feature cartographic label placement problems," *Computers & Operations Research*, vol. 35, no. 7, pp. 2129–2140, 2008, doi: 10.1016/j.cor.2006.10.013.

[72] G. B. Dantzig and P. Wolfe, "Decomposition principle for linear programs," *Operations Research*, vol. 8, no. 1, pp. 101–111, 1960, doi: 10.1287/opre.8.1.101.

[73] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting-stock problem," *Operations Research*, vol. 9, no. 6, pp. 849–859, 1961, doi: 10.1287/opre.9.6.849.

[74] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting stock problem—Part II," *Operations Research*, vol. 11, no. 6, pp. 863–888, 1963, doi: 10.1287/opre.11.6.863.

[75] G. Desaulniers, J. Desrosiers, and M. M. Solomon, *Column generation*, vol. 5. Springer Science & Business Media, 2006. doi: 10.1007/b135457.

[76] L. V. Kantorovich, "Mathematical methods of organizing and planning production," *Management Science*, vol. 6, no. 4, pp. 366–422, 1960, doi: 10.1287/mnsc.6.4.366.

[77] D. Aloise, S. Cafieri, G. Caporossi, P. Hansen, S. Perron, and L. Liberti, "Column generation algorithms for exact modularity maximization in networks," *Physical Review E*, vol. 82, no. 4, p. 46112, 2010, doi: 10.1103/PhysRevE.82.046112.

[78] A. Mehrotra and M. A. Trick, "A column generation approach for graph coloring," *INFORMS Journal on Computing*, vol. 8, no. 4, pp. 344–354, 1996, doi: 10.1287/ijoc.8.4.344.

[79] G. Ribeiro Filho and L. A. N. Lorena, "Constructive genetic algorithm and column generation: an application to graph coloring," in *Proceedings of the APORS 2000 - The Fifth Conference of the Association of Asian-Pacific Operations Research Societies*, 2000.

[80] M. Held and R. M. Karp, "The traveling-salesman problem and minimum spanning trees," *Operations Research*, vol. 18, no. 6, pp. 1138–1162, 1970, doi: 10.1287/opre.18.6.1138.

[81] A. M. Geoffrion, "Lagrangean relaxation for integer programming," in *Approaches to integer programming*, Springer, 1974, pp. 82–114. doi: 10.1007/BFb0120690.

[82] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," in *50 Years of Integer Programming 1958-2008*, Springer, 2010, pp. 105–132. doi: 10.1007/978-3-540-68279-0_5.

[83] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[84] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[85] W. L. Eastman, "Linear programming with pattern constraints," Doctoral dissertation, 1958.

[86] E. L. Lawler, "The traveling salesman problem: a guided tour of combinatorial optimization," *Wiley-Interscience Series in Discrete Mathematics*, 1985.

[87] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.

[88] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*, Springer, 1987, pp. 7–15. doi: 10.1007/978-94-015-7744-1_2.

Index

B

Binary Knapsack Problem 31
Branch-and-Bound 205

C

Clique Partitioning Problem 120
Column Generation 133
Covering problems 54

G

Graph Coloring Problem 101
Graph Partitioning Problem 106

I

Integer Knapsack Problem 36
Integer Programming 19

L

Lagrangian Relaxation 163
Linear Programming 21

M

Maximal Coverage Location Problem . 59
Maximum Independent Set Problem . 93
Metis 112

O

One-dimensional Cutting Stock Problem. 134

P

P-Median Problem 44

S

Set Covering Location Problem 55

T

Traveling Salesman Problem 71

V

Vertex Covering Problem 96

W

Weighted Graph Partitioning Problem. . 115

A. Julia Introduction

A summary of the Julia language commands will be presented in this chapter, mainly those used in the previous chapters. For a detailed description of the language see the books by [2], [3] and [4].

A.1 Random number generation

A.1.1 Set seed

```
using Random
Random.seed!(42)
```

A.1.2 Generate a random number from a uniform distribution [0, 1)

```
rand()
```

Output

```
0.6293451231426089
```

A.1.3 Generate 4 random numbers from a uniform distribution [0, 1)

```
rand(4)
```

Output

```
4-element Vector{Float64}:
0.6293451231426089
0.4503389405961936
0.47740714343281776
0.7031298490032014
```

A.1.4 Generate a matrix (2x2) of random numbers from a uniform distribution [0, 1)

```
rand(2, 2)
```

```
Output
2×2 Matrix{Float64}:
0.0793339 0.492058
0.663758  0.713226
```

A.1.5 Generate 2 random numbers from a uniform distribution [0, 10]

```
rand(0:10, 2)
```

```
Output
2-element Vector{Int64}:
1
5
```

A.1.6 Generate a matrix (2x2) of random numbers from a uniform distribution [0, 10]

```
rand(0:10, 2, 2)
```

```
Output
2×2 Matrix{Int64}:
5 1
9 3
```

A.1.7 Generate a matrix (2x2) of random numbers from a normal distribution

```
randn(2, 2)
```

```
Output
2×2 Matrix{Float64}:
-0.715937 0.954845
-0.959029 0.378568
```

A.2 Vectors and matrices

The index of vectors and matrices starts at 1. There are data structures that we won't cover here, as well as packages that provide other classic structures DataStructures.jl.

A.2.1 Vectors

The following code snippets shows different ways to create and interact with vectors.

A.2.1.1 Vector creation: column vector

```
a = [1, 2, 3]
```

```
Output
3-element Vector{Int64}:
1
2
3
```

A.2.1.2 Vector creation: line vector

```
a = [1 2 3]
```

```
Output
1×3 Matrix{Int64}:
1 2 3
```

A.2.1.3 Vector creation: ones vector

```
a = ones(4)
```

```
Output
4-element Vector{Float64}:
1.0
1.0
1.0
1.0
```

A.2.1.4 Vector creation: zeros vector

```
a = zeros(4)
```

```
Output
4-element Vector{Float64}:
0.0
0.0
0.0
0.0
```

A.2.1.5 Vector creation: sequential

```
a = collect(1:4)
```

```
Output
4-element Vector{Int64}:
1
2
3
4
```

A.2.1.6 Vector creation: sequential vector with defined spacing

```
a = collect(1:2:6)
```

```
Output
3-element Vector{Int64}:
1
3
5
```

A.2.1.7 Vector creation: comprehension

```
a = [i for i in 1:3]
```

```
Output
3-element Vector{Int64}:
1
2
3
```

A.2.1.8 Vector operations: dimensions

```
a = [1, 2, 3]
size(a)
```

```
Output
3-element Vector{Int64}:
(3,)
```

A.2.1.9 Vector operations: transpose

```
a = [1, 2, 3]
a'
```

```
Output
1×3 adjoint(:)Vector{Int64}) with eltype Int64:
1 2 3
```

A.2.1.10 Vector operations: inner product

```
a = [1, 2, 3]
b = [4, 5, 6]
a' * b
```

```
Output
32
```

A.2.1.11 Vector operations: outer product

```
a = [1, 2, 3]
b = [4, 5, 6]
a * b'
```

```
Output
3×3 Matrix{Int64}:
 4  5  6
 8 10 12
12 15 18
```

A.2.1.12 Vector operations: element-wise product

```
a = [1, 2, 3]
b = [4, 5, 6]
a .* b
```

```
Output
3-element Vector{Int64}:
4
10
18
```

A.2.1.13 Vector operations: element-wise exponentiation

```
a = [1, 2, 3]
a .^ 4
```

```
Output
3-element Vector{Int64}:
1
16
81
```

A.2.2 Matrices

The following code snippets shows different ways to create and interact with matrices.

A.2.2.1 Matrix creation: manually

```
A = [1 2 3 4;
     5 6 7 8;
     9 10 11 12;
     13 14 15 16]
```

```
Output
4×4 Matrix{Int64}:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

A.2.2.2 Matrix creation: zero matrix

```
A = zeros(2, 3)
```

```
Output
2×3 Matrix{Float64}:
0.0 0.0 0.0
0.0 0.0 0.0
```

A.2.2.3 Matrix creation: ones matrix

```
A = ones(2, 3)
```

```
Output
2×3 Matrix{Float64}:
1.0 1.0 1.0
1.0 1.0 1.0
```

A.2.2.4 Matrix creation: comprehension

```
A = [i + j for i in 1:3, j in 1:3]
```

Output

```
3×3 Matrix{Int64}:
2 3 4
3 4 5
4 5 6
```

A.2.2.5 Matrix operations: dimensions

```
A = [1 2 3 4;
     5 6 7 8;
     9 10 11 12;
     13 14 15 16]
size(A)
```

Output

```
(4, 4)
```

A.2.2.6 Matrix operations: transpose

```
A = [1 2 3 4;
     5 6 7 8;
     9 10 11 12;
     13 14 15 16]
A
```

Output

```
4×4 adjoint(:)Matrix{Int64}) with eltype Int64:
1 5  9 13
2 6 10 14
3 7 11 15
4 8 12 16
```

A.2.2.7 Matrix operations: matrix-vector multiplication

```
A = [1 2;
3 4]
b = [5, 6]
A * b
```

```
Output
2-element Vector{Int64}:
17
3
```

A.2.2.8 Matrix operations: matrix-matrix multiplication

```
A = [1 2;
     3 4]
B = [5 6;
     7 8]
A * B
```

```
Output
2×2 Matrix{Int64}:
19 22
43 50
```

A.3 Structures and mutable structures

Structures are a way to create custom data types. They can be mutable or immutable. The following code snippets shows how to create and interact with structures.

A.3.1 Immutable structures

```
struct Point
  x::Float64
  y::Float64
end
point = Point(1.0, 2.0)
point.x = 3.0 # This will give an error
```

```
Output
setfield!: immutable struct of type Point cannot be changed
```

A.3.2 Mutable structures

```
mutable struct Point
  x::Float64
  y::Float64
end
```

```
point = MutablePoint(1.0, 2.0)
point.x = 3.0 # This works because is mutable
point
```

Output

```
MutablePoint(3.0, 2.0)
```

A.4 Flow control

A.4.1 If-else statements

```
x = 10
y = 12
if x < y
  println("$x is smaller than $y")
elseif x > y
  println("$x is greater than $y")
else
  println("$x is equal to $y")
end
```

Output

```
10 is smaller than 12
```

A.4.2 For loops

A.4.2.1 For loops with ranges

```
for i in 1:4
  print("$i ")
end
```

Output

```
1 2 3 4
```

A.4.2.2 For loops with vectors

```
a = [5, 2, 1, 8]
for i in a
  print("$i ")
end
```

```
Output
5 2 1 8
```

A.4.2.3 For loops with enumerate

```
a = [5, 2, 1, 8]
for (id, value) in enumerate(a)
  println("$id) $value")
end
```

```
Output
1) 5
2) 2
3) 1
4) 8
```

A.4.3 While loops

```
i = 1
while i <= 4
    print("$i ")
    i += 1
end
```

```
Output
1 2 3 4
```

A.5 Functions

A.5.1 Custom functions

```
function my_sum(x,y)
  return x+y
end
my_sum(10,12)
```

```
Output
22
```

A.5.2 Sum: by column

```
A = [8 1;
     5 6]
```

```
sum(A, dims=1)
```

```
Output
1×2 Matrix{Int64}:
13  7
```

A.5.3 Sum: by row

```
A = [8 1;
     5 6]
sum(A, dims=2)
```

```
Output
2×1 Matrix{Int64}:
 9
11
```

A.5.4 Find: indices of the elements that satisfy a condition

```
v = [1, 0, 0, 1]
findall(v .> 0)
```

```
Output
2-element Vector{Int64}:
1
4
```

A.5.5 Sorting: ascending

```
sort([2,3,1])
```

```
Output
3-element Vector{Int64}:
1 2 3
```

A.5.6 Sorting: descending

```
sort([2,3,1], rev=true)
```

```
Output
3-element Vector{Int64}:
3 2 1
```

A.5.7 Sorting: sorting order

```
sortperm([2,3,1])
```

Output

```
3-element Vector{Int64}:
3
1
2
```

A.5.8 Sorting: by line

```
A = [8 7;
     5 6]
sort(A, dims=1)
```

Output

```
2×2 Matrix{Int64}:
5 6
8 7
```

A.5.9 Sorting: by column

```
A = [8 7;
     5 6]
sort(A, dims=2)
```

Output

```
2×2 Matrix{Int64}:
7 8
5 6
```

A.5.10 Sorting: order for lines

```
sortperm(A, dims = 1)
```

Output

```
2×2 Matrix{Int64}:
2 4
1 3
```

A.6 Advanced example

In this subsection we are going to use Julia to implement the Simulated Annealing (SA) [88] metaheuristic to approximately solve the Traveling Salesman Problem [39]. We are going to implement the following pseudocode:

Algorithm A.4: Simulated Annealing

procedure SIMULATED ANNEALING(s, T, T_{min}, α, L)
  $s^* \leftarrow s$
  while $T > T_{min}$ **do**
    for $l \leq L$ **do**
      $l \leftarrow l + 1$
      $s' \leftarrow N(s)$
      $\Delta \leftarrow f(s') - f(s)$
      if $(\Delta < 0)$ or $\left(\text{rand}(0,1) < e^{-\frac{\Delta}{T}}\right)$ **then**
        if $f(s') < f(s^*)$ **then**
          $s^* \leftarrow s'$
        end
        $s \leftarrow s'$
      end
    end
    $T \leftarrow \alpha * T$
  end
  return s^*
end

The following Julia Code implements the Simulated Annealing for TSP. Figure A.1 presents a plot of the best route found by the algorithm.

```julia
using Random          # For reproducibility
using DelimitedFiles # To read CSV files
using Distances       # For distance calculations
using Plots           # For plotting the route

# Structure to hold the problem data
struct ProblemData
  distance_matrix # Distance matrix
  coordinates     # Vector of coordinates
end

# Structure to hold the solution
mutable struct Solution
  route     # Vector of city indices
  objective # Objective value
end
```

```
17
18 # Function to plot the route of a solution
19 function plot_route(s, data)
20   # Base plot: the cities
21   p = plot(data.coordinates[:, 1],
22            data.coordinates[:, 2],
23            seriestype = :scatter,
24            label = "Cities",
25            legend = :topleft)
26   # Add label for each city
27   for i in 1:length(sBest.route)
28     x = data.coordinates[sBest.route[i], 1]
29     y = data.coordinates[sBest.route[i], 2]
30     annotate!(x, y + 0.0005,
31               text(string(sBest.route[i]), 8))
32   end
33   # Extract ordered route coordinates
34   xs = data.coordinates[sBest.route, 1]
35   ys = data.coordinates[sBest.route, 2]
36   # Include the starting point (loop)
37   xf = vcat(xs, xs[1])
38   yf = vcat(ys, ys[1])
39   # Draw lines between the cities
40   plot!(xf, yf, seriestype=:path,
41         color=:red, label="Best Route", lw=2)
42   # Show the plot
43   display(p)
44 end
45
46 # Function to calculate the objective value
47 function f(s, data)
48   s.objective = 0.0
49   n = length(s.route)
50   for i in 1:n-1
51     s.objective += data.distance_matrix[s.route[i],
s.route[i+1]]
52   end
53   # Add distance from last to first city
54   s.objective += data.distance_matrix[s.route[n], s.route[1]]
55 end
56
```

```
57 # Neighbor function (Swap two cities)
58 function N(s, data)
59   # Create a new solution based on the current one
60   sNew = deepcopy(s)
61   n = length(s.route)
62   # Randomly select two indices
63   i, j = rand(1:n), rand(1:n)
64   while i == j
65     j = rand(1:n)
66   end
67   # Swap the cities
68   sNew.route[i], sNew.route[j] = sNew.route[j], sNew.route[i]
69   # Recalculate the objective value for the new solution
70   f(sNew, data)
71   return sNew
72 end
73
74 # Simulated Annealing algorithm for TSP
75 function SA(s, T, Tmin, α, L, data)
76   sBest = deepcopy(s)
77   while T > Tmin
78     for l in 1:L
79       # Generate a neighbor solution by swapping two cities
80       sNew = N(s, data)
81       # Calculate the change in objective value
82       Δ = sNew.objective - s.objective
83       # If the new solution is better or accepted by probability
84       if Δ < 0 || rand() < exp(-Δ / T)
85         s = deepcopy(sNew)
86         if s.objective < sBest.objective
87           sBest = deepcopy(s)
88         end
89       end
90     end
91     T *= α  # Cool down the temperature
92   end
93   return sBest
94 end
95
96 # Load the problem data from CSV files
97 coordinates = readdlm("data/tsp_dataset.csv", ',')
```

```
98
99  # Create a distance matrix from the coordinates using Distances.jl
100 distance_matrix = pairwise(Haversine(6371.0), coordinates; dims =
    1)
101
102 # Initialize the problem data structure
103 data = ProblemData(distance_matrix, coordinates)
104
105 # Create an initial solution with a random route
106 s = Solution(shuffle!(collect(1:size(coordinates, 1))), Inf)
107
108 # Calculate the objective value for the initial solution
109 f(s, data)
110
111 # Run the Simulated Annealing algorithm
112 Random.seed!(42) # For reproducibility
113 sBest = SA(s, 1000.0, 100.0, 0.95, 10, data)
114
115 # Print the best solution found
116 println("Best route: ", sBest.route)
117 println("Best objective value: $(sBest.objective) km")
118
119 # Plot the best route
120 plot_route(sBest, data)
```

Output

```
Best route: [5, 3, 2, 4, 1]
Best objective value: 3.1003739990197374 km
```

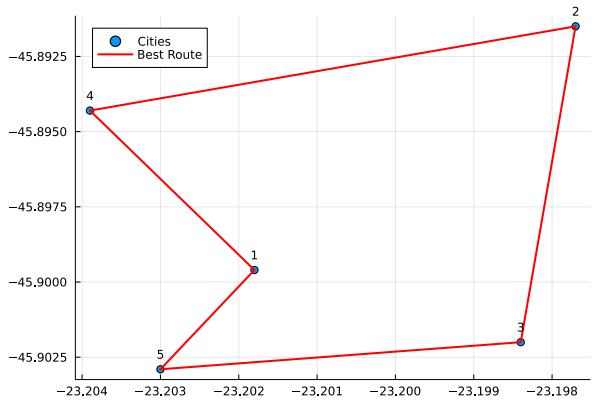


Figure A.1: TSP solution obtained with Simulated Annealing.

B. Exercise Solutions

All the code in this book is available in the form of Jupyter notebooks, which can be accessed on GitHub.

B.1 Chapter 1

B.1.1 Exercise 1.1

Suppose there are ten possible locations where health centers could be installed to serve a population. Express the following constraints on which centers can be selected:

$$x_i = \begin{cases} 1 \text{ if you select center } i \\ 0 \text{ otherwise} \end{cases} \tag{B.1}$$

B.1.1.1 a) You cannot select all centers

$$\sum_{i=1}^{10} x_i \leq 9 \tag{B.2}$$

B.1.1.2 b) At least one center must be selected

$$\sum_{i=1}^{10} x_i \geq 1 \tag{B.3}$$

B.1.1.3 c) If center 5 is selected, then center 2 cannot be selected

$$x_2 + x_5 \leq 1 \tag{B.4}$$

B.1.1.4 d) Center 10 can be selected only if center 7 is also selected

$$x_{10} \leq x_7 \tag{B.5}$$

B.1.1.5 e) Centers 3 and 8 must either both be selected or both not selected

$$x_3 = x_8 \tag{B.6}$$

B.1.2 Exercise 1.2

Listing B.1 presents a Julia code to solve the Flow Free puzzle using an integer programming model. The code defines a function `solve_flow_free` that takes as input a dataset containing the terminal points of the puzzle and the grid size. The

function creates a JuMP model, defines binary variables for cell colors and edge selections, and sets up constraints to ensure valid solutions. Finally, it solves the model and visualizes the solution using a utility function.

```
using JuMP       # Modeling language
using HiGHS      # Solver
using CSV        # Data handling
using DataFrames # Data handling

# Utility function to draw solution
include("utils/flowfree_utils.jl")

# Function to solve the Flow Free problem
function solve_flow_free(data; grid_size = 5)

  total_cells = grid_size^2
  total_colors = maximum(data.color)
  total_directions = 4  # (up, down, right, left)

  # Location and color of terminal points on board
  t = zeros(Int64, total_cells)
  for row in eachrow(data)
    id = coordinates_to_id(row.X, row.Y, grid_size)
    t[id] = row.color
  end

  # Ranges for variables
  T = 1:total_cells
  C = 1:total_colors
  D = 1:total_directions

  # Create model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Current activate color in the cell
  @variable(model, cell_color[i in T, k in C], Bin)

  # Neighbor cells connected (up, down, right, left)
  @variable(model, edge_selected[i in T, j in D], Bin)
```

```
39
40    # Neighbor cells color
41    @variable(model, edge_color[i in T, j in D, k in C], Bin)
42
43    # Objective: Maximize the sum of the number of cells connected by
      color
44    @objective(model, Max, sum(edge_color[i,j,k] for i in T,
45                                                    j in D,
46                                                    k in C))
47
48    ###################################################
49    # Creation of type 1 constraints: cell coverage
50    ###################################################
51    for i in T
52      if t[i] != 0
53        # For terminal cells, only one edge can be selected
54        @constraint(model,
55          sum(edge_selected[i,j] for j in D) == 1
56        )
57      else
58        # For non-terminal cells, two edges must be selected
59        @constraint(model,
60          sum(edge_selected[i,j] for j in D) == 2
61        )
62      end
63    end
64
65    ########################################
66    # Creation of type 2 constraints:
67    #    Limit edge Constraints (each edge can only have 1 color)
68    # Logic:
69    #   a) Blocking the creation of color variables to
70    #      ends not allocated (edge_selected[i,j] == 0)
71    #   b) Allows only one color on the ends that are
72    #      allocated (if edge_selected[i,j] == 1, then
73    #      sum (edge_color ijk) <= 1)
74    ########################################
75    for i in T
76      for j in D
77        @constraint(model,
78          sum(edge_color[i,j,k] for k in C) <= edge_selected[i,j]
```

```
79        )
80      end
81    end
82
83    ###################################
84    # Creation of type 3 constraints: Horizontal Connectivity
85    # Note: it was not necessary to treat the horizontal ends
86    # of the corners of the board, because they will be set to
87    # zero by other restrictions (those of type 5).
88    # Remembering that:
89    # 1 - up
90    # 2 - down
91    # 3 - right
92    # 4 - left
93    ###################################
94    for i in 1:(total_cells-1)
95      for k in C
96        # Connects the right edge of cell to the
97        # left edge of the next cell
98        @constraint(model,
99          edge_color[i,3,k] == edge_color[i+1,4,k]
100       )
101     end
102   end
103
104   ########################################################
105   # Creation of type 4 constraints: Vertical Connectivity
106   ########################################################
107   for i in 1:(total_cells-grid_size)
108     for k in C
109       # Connects the bottom edge of cell to the top
110       # edge of the cell below
111       @constraint(model,
112         edge_color[i,2,k] == edge_color[i+grid_size,1,k]
113       )
114     end
115   end
116
117   ##################################################################
118   # Creation of type 5 constraints: Boundary Edges (extreme nodes)
119   ##################################################################
```

```
120
121   # Top of the board: north end of cells must be disabled
122   for i in 1:grid_size
123     @constraint(model, edge_selected[i,1] == 0)
124   end
125
126   # Bottom of the board: south end of cells must be disabled
127   for i in (total_cells-grid_size+1):total_cells
128     @constraint(model, edge_selected[i,2] == 0)
129   end
130
131   # Right part of board: east end of cells must be turned off
132   for i in grid_size:grid_size:total_cells
133     @constraint(model, edge_selected[i,3] == 0)
134   end
135
136   # Left part of board: left end of cells must be turned off
137   for i in 1:grid_size:(total_cells-grid_size+1)
138     @constraint(model, edge_selected[i,4] == 0)
139   end
140
141   ##################################################################
142   # Creation of type 6 constraints: Single color in Terminal Cells
143   ##################################################################
144   for i in T
145     if t[i] != 0
146       for k in C
147         if k == t[i]
148           @constraint(model,sum(edge_color[i,j,k] for j in D) == 1)
149           @constraint(model,cell_color[i,k] == 1)
150         else
151           @constraint(model,sum(edge_color[i,j,k] for j in D) == 0)
152         end
153       end
154     end
155   end
156
157   ####################################################
158   # Creation of type 7 constraints: Same color per cell
159   # & Pick 1 constraints
160   ####################################################
```

```
161
162   # Each cell can only have one color
163   for i in T
164     @constraint(model, sum(cell_color[i,k] for k in C) == 1)
165   end
166
167   ##################################################
168   # The sum of the cell border color indication variables
169   # must be equal to 2 or equal to zero for each color
170   #  a) Will be equal to 2 if that color is activated
171   #     in the cell, that is, the variable p will be 1
172   #     (that's why the restriction applies 2*p[cell,color],
173   #     which will be equal to 2)
174   #  b) Will be zero if that color is not enabled in the
175   #     cell (and therefore 2*p[cell,color], which will equal 0)
176   ##################################################
177   for i in T
178     if t[i] == 0
179       for k in C
180         @constraint(model,
181           sum(edge_color[i,j,k] for j in D) == 2 * cell_color[i,k]
182         )
183         end
184       end
185   end
186
187   # Solve the model
188   JuMP.optimize!(model)
189
190   # Get values of the variables (color of the activated edge)
191   edge_color_opt = JuMP.value.(edge_color)
192
193   # Display the board results
194   p = draw_free_flow_solution(grid_size, data, edge_color_opt)
195
196   # Display the solution
197   display(p)
198 end
```

Listing B.1: Julia code to solve a Flow Free instance.

B.1.2.1 a) Solve using the dataset `flowfree5_1.csv`

```
# Load data
data = CSV.read("data/flowfree5_1.csv", DataFrame)
# Solve the problem and display the solution
solve_flow_free(data, grid_size = 5)
```

Listing B.2: Solving the Flow Free instance `flowfree5_1.csv`.

Figure B.1 presents the solution for the `flowfree5_1.csv` instance.

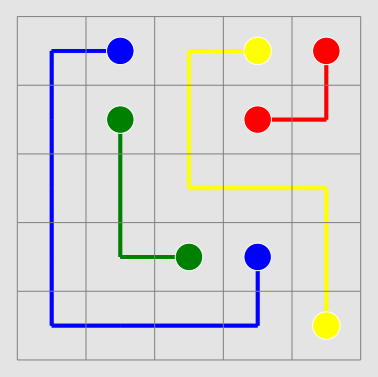


Figure B.1: Solution for the `flowfree5_1.csv` instance.

B.1.2.2 b) Solve using the dataset `flowfree5_2.csv`

We can use the same code presented in Listing B.2 to solve the instance `flowfree5_2.csv`. Just change the name of the file in the line where we load the data.

```
# Load data
data = CSV.read("data/flowfree5_2.csv", DataFrame)
# Solve the problem and display the solution
solve_flow_free(data, grid_size = 5)
```

Listing B.3: Solving the Flow Free instance `flowfree5_2.csv`.

Figure B.2 presents the solution for the `flowfree5_2.csv` instance.

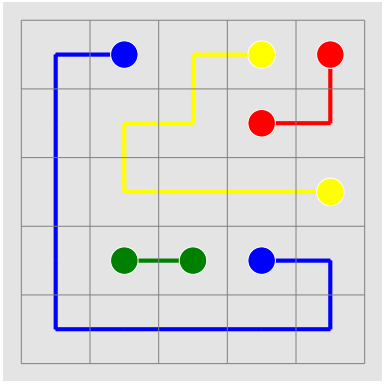


Figure B.2: Solution for the `flowfree5_2.csv` instance.

B.1.2.3 c) Solve using the dataset `flowfree6_1.csv`

We can use the same code presented in Listing B.2 to solve the instance `flowfree6_1.csv`. Just change the name of the file in the line where we load the data.

```
1 # Load data
2 data = CSV.read("data/flowfree6_1.csv", DataFrame)
3 # Solve the problem and display the solution
4 solve_flow_free(data, grid_size = 6)
```

Listing B.4: Solving the Flow Free instance `flowfree6_1.csv`.

Figure B.3 presents the solution for the `flowfree6_1.csv` instance.

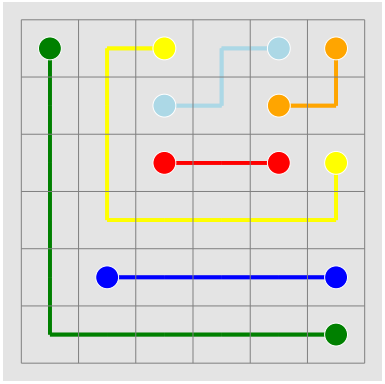


Figure B.3: Solution for the `flowfree6_1.csv` instance.

B.2 Chapter 2

B.2.1 Exercise 2.1

Listing B.5 presents the Julia code to solve binary, integer models and its relaxations for the knapsack problem giving an instance. The code defines functions to solve both the Binary Knapsack Problem (BKP) and the Integer Knapsack Problem (IKP), as well as their linear relaxations. The results are stored in a DataFrame for easy comparison.

```
using JuMP       # Modeling language
using HiGHS      # Solver
using DataFrames # Data handling

# Function to solve the Binary Knapsack Problem (BKP) and its linear relaxation
function solve_bkp(v, w, W; relaxation = false)
  model = JuMP.Model(HiGHS.Optimizer)
  JuMP.set_silent(model)
  n = length(v)
  if relaxation
    @variable(model, 0 <= x[1:n] <= 1)
  else
    @variable(model, x[1:n], Bin)
  end
  @objective(model, Max, sum(v[i] * x[i] for i in 1:n))
  @constraint(model, sum(w[i] * x[i] for i in 1:n) <= W)
  JuMP.optimize!(model)
  objective_value = JuMP.objective_value(model)
  solution = JuMP.value.(x)
  weight = sum(w[i] * solution[i] for i in 1:n)
  return objective_value, solution, weight
end

# Function to solve the Integer Knapsack Problem (IKP) and its linear relaxation
function solve_ikp(v, w, W; relaxation = false)
  model = JuMP.Model(HiGHS.Optimizer)
  JuMP.set_silent(model)
  n = length(v)
  if relaxation
    @variable(model, x[1:n] >= 0)
```

```
  else
    @variable(model, x[1:n] >= 0, Int)
  end
  @objective(model, Max, sum(v[i] * x[i] for i in 1:n))
  @constraint(model, sum(w[i] * x[i] for i in 1:n) <= W)
  JuMP.optimize!(model)
  objective_value = JuMP.objective_value(model)
  solution = JuMP.value.(x)
  weight = sum(w[i] * solution[i] for i in 1:n)
  return objective_value, solution, weight
end

# Function to solve all four problems and store results in a
DataFrame
function solve_knapsack_problems(v, w, W)
  results = DataFrame(
    Problem = ["BKP", "LP-BKP", "IKP", "LP-IKP"],
    Objective = [0.0, 0.0, 0.0, 0.0],
    Solution = [Float64[], Float64[], Float64[], Float64[]],
    Weight = [0.0, 0.0, 0.0, 0.0]
  )
  results[1, :Objective], results[1, :Solution], results[1, :Weight]
= solve_bkp(v, w, W)
  results[2, :Objective], results[2, :Solution], results[2, :Weight]
= solve_bkp(v, w, W; relaxation = true)
  results[3, :Objective], results[3, :Solution], results[3, :Weight]
= solve_ikp(v, w, W)
  results[4, :Objective], results[4, :Solution], results[4, :Weight]
= solve_ikp(v, w, W; relaxation = true)
  return results
end
```

Listing B.5: Julia code to solve binary, integer models and its relaxations for the knapsack problem.

B.2.1.1 a) Solving the following knapsack problem:

$$v = (5, 7, 4, 3) \quad w = (16, 22, 12, 8) \quad W = 30$$

Listing B.6 presents the code to solve the first knapsack problem using the functions defined in Listing B.5. The data for values, weights, and capacity are

provided as tuples. The code calls the `solve_knapsack_problems` function with the specified data and prints the results.

```
# Data
v = [5, 7, 4, 3]    # values
w = [16, 22, 12, 8] # weights
W = 30              # capacity
# Solve the problems and print results
results = solve_knapsack_problems(v, w, W)
println(results)
```

Listing B.6: Solving the first knapsack problem.

Discussion

Listing B.7 presents the results obtained for the first knapsack problem. The table shows the objective value, solution vector, and total weight for each of the four problem variants: BKP, BKP(LP), IKP, and IKP(LP).

Output

Row	Problem	Objective	Solution	Weight
	String	Float64	Array…	Float64
1	BKP	10.0	[0.0, 1.0, -0.0, 1.0]	30.0
2	BKP(LP)	10.1818	[0.0, 0.454545, 1.0, 1.0]	30.0
3	IKP	10.0	[0.0, 0.0, 1.0, 2.0]	28.0
4	IKP(LP)	11.25	[0.0, 0.0, 0.0, 3.75]	30.0

Listing B.7: Results for the first knapsack problem.

In BKP, only one copy of each item is allowed. The optimal solution selects items 2 and 4, exactly filling the knapsack to capacity (30 units) and achieving an objective value of 10. In BKP(LP), you can select fractional parts of a item. Here, item 2 is partially selected (about 0.45), and items 3 and 4 are fully selected, slightly increasing the objective value to 10.18. The knapsack is fully utilized. The gap is $z_{\text{BKP(LP)}} - z_{\text{BKP}} = 10.18 - 10 = 0.18$.

In IKP, multiple copies of each item are allowed, but the optimal solution selects one of item 3 and two of item 4, reaching a total value of 10 and a weight of 28 (not fully utilizing the capacity due to integer constraints). In IKP(LP), both repetition and fractional quantities are allowed. The solution allocates all capacity to item

4 (3.75 units), achieving the highest objective value (11.25) and fully utilizing the knapsack. The gap is $z_{\text{IKP(LP)}} - z_{\text{IKP}} = 11.25 - 10 = 1.25$.

The results illustrate three important optimization concepts:

a) Relaxation increases the objective value (for maximization) enlarging the feasible region, so the objective value can only improve (or remain equal)

b) The hierarchy of objective values reflects the progressive enlargement of the feasible region: binary constraints are the most restrictive, LP relaxation removes integrality, integer multiplicities allow repeated items, and full relaxation allows both repetition and fractional quantities.

c) Both LP solutions tend to exploit the best density items (4 and 3)

$$\left(\frac{v_1}{w_1}, \frac{v_2}{w_2}, \frac{v_3}{w_3}, \frac{v_4}{w_4}\right) = \left(\frac{5}{16}, \frac{7}{22}, \frac{4}{12}, \frac{3}{8}\right) = (0.31^4, 0.32^3, 0.33^2, 0.38^1) \quad \text{(B.7)}$$

These results demonstrate how modeling assumptions strongly influence both the structure of the optimal solution and the achievable profit.

B.2.1.2 b) Solving the following knapsack problem:

$$v = (4, 2, 10, 2, 1) \quad w = (12, 2, 4, 1, 1) \quad W = 15$$

Listing B.8 presents the code to solve the second knapsack problem using the same functions defined in Listing B.5. The data for values, weights, and capacity are updated to reflect the new instance. The code calls the `solve_knapsack_problems` function with the new data and prints the results.

```
# Data
v = [4, 2, 10, 2, 1] # values
w = [12, 2, 4, 1, 1] # weights
W = 15               # capacity
# Solve the problems and print results
results = solve_knapsack_problems(v, w, W)
println(results)
```

Listing B.8: Solving the second knapsack problem.

Discussion

Listing B.9 presents the results obtained for the second knapsack problem. The table shows the objective value, solution vector, and total weight for each of the four problem variants: BKP, BKP(LP), IKP, and IKP(LP).

Output

Row	Problem	Objective	Solution	Weight
	String	Float64	Array…	Float64
1	BKP	15.0	[0.0, 1.0, 1.0, 1.0, 1.0]	8.0
2	BKP(LP)	17.3333	[0.583333, 1.0, 1.0, 1.0, 1.0]	15.0
3	IKP	36.0	[0.0, 0.0, 3.0, 3.0, 0.0]	15.0
4	IKP(LP)	37.5	[0.0, 0.0, 3.75, 0.0, 0.0]	15.0

Listing B.9: Results for the second knapsack problem.

In BKP a notable observation is that the knapsack is not completely filled, since only 8 of the 15 capacity units are used. This happens because the remaining item (item 1) is too heavy to fit in the remaining space. The binary restriction significantly limits the feasible region and leads to the smallest objective value among all models. In the relaxation BKP(LP), there is an increase in the objective value and the knapsack is fully filled. The improvement occurs because the LP relaxation enlarges the feasible region by allowing fractional decisions. Therefore, the LP relaxation provides an upper bound for the binary problem ($z_{\text{BKP(LP)}} \geq z_{\text{BKP}}$), and the gap is $17.33 - 15 = 2.33$.

In the integer version (IKP) multiple copies of each item are allowed. The objective value rises dramatically to 36 because item 3 has the best value-to-weight ratio (10/4 = 2.5) and item 4 efficiently fills the remaining capacity. Compared to BKP, the feasible region is much larger because items can be repeated. This explains the large increase in profit. Thus, allowing repetitions fundamentally changes the structure of the optimal solution. Finally, in the relaxed integer model IKP(LP), the optimal strategy is to allocate all capacity to the item with the highest value-to-weight ratio. Since item 3 dominates (10/4 = 2.5) the solution selects 3.75 of this item. This is the largest objective value among all models because the feasible region is the largest possible. Again, the LP relaxation provides an upper bound ($z_{\text{IKP(LP)}} gez_{\text{IKP}}$) and the gap is relatively small $z_{\text{IKP(LP)}} - z_{\text{IKP}} = 37.5 - 36 = 1.5$.

B.2.1.3 c) Solving the following knapsack problem:

$$v = (3, 9, 6, 3, 2) \quad w = (8, 5, 12, 6, 4) \quad W = 17$$

Listing B.10 presents the code to solve the third knapsack problem using the same functions defined in Listing B.5. The data for values, weights, and capacity are updated to reflect the new instance. The code calls the `solve_knapsack_problems` function with the new data and prints the results.

```julia
# Data
v = [3, 9, 6, 3, 2]  # values
w = [8, 5, 12, 6, 4] # weights
W = 17               # capacity
# Solve the problems and print results
results = solve_knapsack_problems(v, w, W)
println(results)
```

Listing B.10: Solving the third knapsack problem.

Discussion

Listing B.11 presents the results obtained for the third knapsack problem. The table shows the objective value, solution vector, and total weight for each of the four problem variants: BKP, BKP(LP), IKP, and IKP(LP).

Output

Row	Problem	Objective	Solution	Weight
	String	Float64	Array…	Float64
1	BKP	15.0	[0.0, 1.0, 1.0, 0.0, 0.0]	17.0
2	BKP(LP)	15.0	[0.0, 1.0, 0.166667, 1.0, 1.0]	17.0
3	IKP	27.0	[0.0, 3.0, 0.0, 0.0, -0.0]	15.0
4	IKP(LP)	30.6	[0.0, 3.4, 0.0, 0.0, 0.0]	17.0

Listing B.11: Results for the third knapsack problem.

In BKP, only one copy of each item is allowed. The optimal solution selects items 2 and 3, reaching a total value of 15 and a weight of 17 (fully utilizing the capacity). In BKP(LP), you can select fractional parts of a item. Here, item 3 is partially selected (about 0.16), and items 2, 4 and 5 are fully selected, acheiving the same objective value to 15 and fully utilizing the knapsack. The gap is $z_{\text{BKP(LP)}} - z_{\text{BKP}} = 17 - 17 = 0$. The LP solution is fractional, yet it achieves exactly the same objective value as the integer optimum. This means there are multiple optimal LP solutions, one of which happens to be fractional.

In IKP, multiple copies of each item are allowed, but the optimal solution is composed only by items of type 2 since they have the highest value-to-weight ratio $\left(\frac{v_2}{w_2} = 1.8\right)$ and the solution is limited by the weight of this item. In IKP(LP), both repetition and fractional quantities are allowed. The solution allocates all capacity

to item 3 (about 3.4 units), achieving the highest objective value (30.6) and fully utilizing the knapsack. The gap is $z_{\text{IKP(LP)}} - z_{\text{IKP}} = 17 - 15 = 2$.

B.3 Chapter 3

B.3.1 Exercise 3.1

Listing B.12 provides a Julia code to solve the P-Median Problem with n clients and p facilities (PMP-N) for the instance `coordinates.csv`. The code uses the JuMP modeling language and the HiGHS solver to find the optimal location of facilities that minimizes the total distance between clients and their assigned facilities. The solution is visualized using a custom plotting function that shows the selected facilities and their assigned clients on a map.

```
using JuMP      # Modeling language
using HiGHS     # Solver
using CSV       # Data handling
using Distances # Distance computations

# Include utility functions for plotting the solution
include("utils/pmp-n_utils.jl")

# Function to solve the P-Median Problem with n clients and p
facilities
function solve_pmp_n(file_path; p = 2)
  # Load latitude and longitude for all points
  coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)

  # Problem dimensions
  n = size(coordinates, 1)

  # Index sets
  I, J = 1:n, 1:n

  # Use Haversine distance to compute distance matrix
  D = Distances.pairwise(Distances.Haversine(), coordinates, dims=1)

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)
```

```
28
29    # Decision variables
30    @variable(model, x[i in I, j in J], Bin)
31
32    # Objective: minimize total distance
33    @objective(model, Min, sum(D[i, j] * x[i, j] for i in I, j in J))
34
35    # Constraint: exactly p facilities selected
36    @constraint(model, sum(x[j, j] for j in J) == p)
37
38    # Constraint: each client assigned to one facility
39    @constraint(model, [i in I], sum(x[i, j] for j in J) == 1)
40
41    # Constraint: assignment only to open facilities
42    @constraint(model, [i in I, j in J], x[i, j] <= x[j, j])
43
44    # Solve the model
45    JuMP.optimize!(model)
46
47    # Extract solution
48    x_value = JuMP.value.(x)
49    objective_value = JuMP.objective_value(model)
50
51    # Print objective value
52    println("Objective value: $objective_value meters")
53
54    # Print selected facilities
55    facility_ids = findall(x_value[j, j] .> 0.5 for j in J)
56    println("Selected Facilities (p=$p): $facility_ids")
57
58    # Print total client assignments for each selected facility
59    assignments = Dict()
60    for facility_id in facility_ids
61      clients = findall(x_value[i, facility_id] .> 0.5 for i in I)
62      assignments[facility_id] = clients
63      println("Facility: $facility_id | $(length(clients)) clients
    assigned")
64    end
65
66    # Plot the solution
67    fmap = plot_solution(coordinates, assignments, p)
```

```
  display(fmap)
end

# Example usage
solve_pmp_n("data/coordinates.csv", p = 4)
```

Listing B.12: P-Median Problem (PMP-N) solution for $p = 4$.

Listing B.13 provides the output of the code above, showing the objective value, selected facilities, and the number of clients assigned to each facility for the P-Median Problem with n clients and p=4 facilities (PMP-N) model.

Output

```
Objective value: 16528.64615458511 meters
Selected Facilities (p=4): [18, 78, 92, 99]
Facility: 18 | 24 clients assigned
Facility: 78 | 23 clients assigned
Facility: 92 | 23 clients assigned
Facility: 99 | 32 clients assigned
```

Listing B.13: Output of the P-Median Problem with n clients and $p = 4$ facilities (PMP-N) model.

Figure B.4 provides a graphical representation of the solution for the P-Median Problem (PMP-N) with n clients and $p = 4$ facilities. The figure shows the locations of the selected facilities and their assigned clients on a map, illustrating the spatial distribution of the solution.

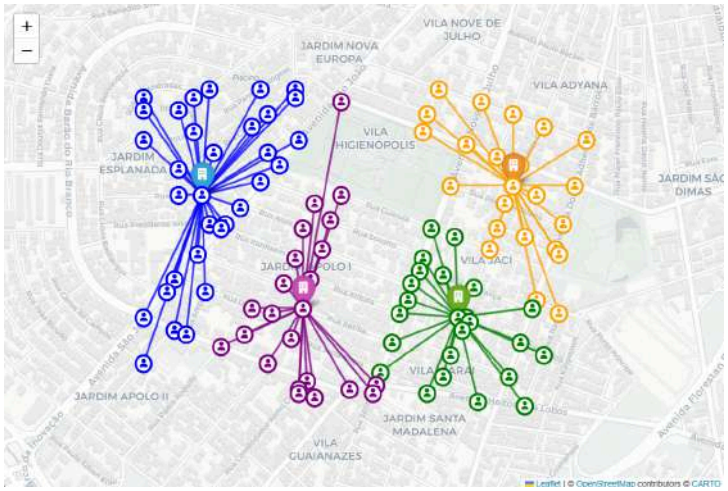


Figure B.4: Graphical representation of the solution for the P-Median Problem (PMP-N) with n clients and $p = 4$ facilities.

B.3.2 Exercise 3.2

We use the following mathematical model for the Fault-tolerant P-Median Problem [36] and solve for 3 medians (p) and 2 for the redundancy parameter (r).

$$\min \sum_{i \in N} \sum_{j \in N} d_{ij} x_{ij} \tag{B.8.1}$$

$$\text{s.t.} \sum_{j \in N} x_{jj} = p \tag{B.8.2}$$

$$\sum_{j \in N} x_{ij} = r \quad \forall i \in N \tag{B.8.3}$$

$$x_{ij} \leq x_{jj} \quad \forall i \in N, j \in N \tag{B.8.4}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in N, j \in N \tag{B.8.5}$$

Listing B.14 provides a Julia code to solve the Fault-tolerant P-Median Problem for the instance `coordinates.csv` with $p = 3$ medians and $r = 2$ redundancy. The code uses the JuMP modeling language and the HiGHS solver to find the optimal location of facilities that minimizes the total distance while ensuring that each client is covered by at least two facilities. The solution is visualized using a custom plotting function that shows the selected facilities and their assigned clients on a map, illustrating the fault-tolerant nature of the solution.

```julia
using JuMP      # Modeling language
using HiGHS     # Solver
using CSV       # Data handling
using Distances # Distance computations

# Importing Python libraries for plotting
include("utils/pmp-fault_utils.jl")

# Function to solve the Fault-tolerant P-Median Problem
function solve_fault_tolerant_pmp(file_path; p = 3, r = 2)
  # Load latitude and longitude for all points
  coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)

  # Problem dimensions
  n = size(coordinates, 1)

  # Index sets
  I, J = 1:n, 1:n

  # Use Haversine distance to compute distance matrix
```

```
  D = Distances.pairwise(Distances.Haversine(), coordinates, dims=1)

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Decision variables
  @variable(model, x[i in I, j in J], Bin)

  # Objective: minimize total distance
  @objective(model, Min, sum(D[i, j] * x[i, j] for i in I, j in J))

  # Constraint: exactly p facilities selected
  @constraint(model, sum(x[j, j] for j in J) == p)

  # Constraint: assignment only to open facilities
  @constraint(model, [i in I, j in J], x[i, j] <= x[j, j])

  # Constraint: each node i must be covered r times (including
possibly by itself)
  @constraint(model, [i in I], sum(x[i, j] for j in J) == r)

  # Solve the model
  JuMP.optimize!(model)

  # Extract solution
  x_value = JuMP.value.(x)
  objective_value = JuMP.objective_value(model)

  # Print objective value
  println("Objective value: $objective_value meters")

  # Print selected facilities
  facility_ids = findall(x_value[j, j] .> 0.5 for j in J)
  println("Selected Facilities (p=$p): $facility_ids")

  # Print total client assignments for each selected facility
  assignments = Dict()
  for facility_id in facility_ids
```

```
    clients = findall(x_value[i, facility_id] .> 0.5 for i in I)
    assignments[facility_id] = clients
    println("Facility: $facility_id | $(length(clients)) clients assigned")
  end

  # Plot the solution
  fmap = plot_solution(coordinates, assignments, p)
  display(fmap)
end

# Example usage
solve_fault_tolerant_pmp("data/coordinates.csv", p = 3, r = 2)
```

Listing B.14: Fault-tolerant P-Median Problem (PMP-N) solution for $p = 3$ and $r = 2$.

Listing B.15 provides the output of the code above, showing the objective value, selected facilities, and the number of clients assigned to each facility for the Fault-tolerant P-Median Problem with $p = 3$ medians and $r = 2$ redundancy.

Output

```
Objective value: 62426.06000659928 meters
Selected Facilities (p=3): [7, 12, 55]
Facility: 7 | 52 clients assigned
Facility: 12 | 50 clients assigned
Facility: 55 | 102 clients assigned
```

Listing B.15: Output of the Fault-tolerant P-Median Problem with $p = 3$ and $r = 2$.

Figure B.5 provides a graphical representation of the solution for the Fault-tolerant P-Median Problem with $p = 3$ and $r = 2$. The figure shows the locations of the selected facilities and their assigned clients on a map, illustrating the spatial distribution of the solution.

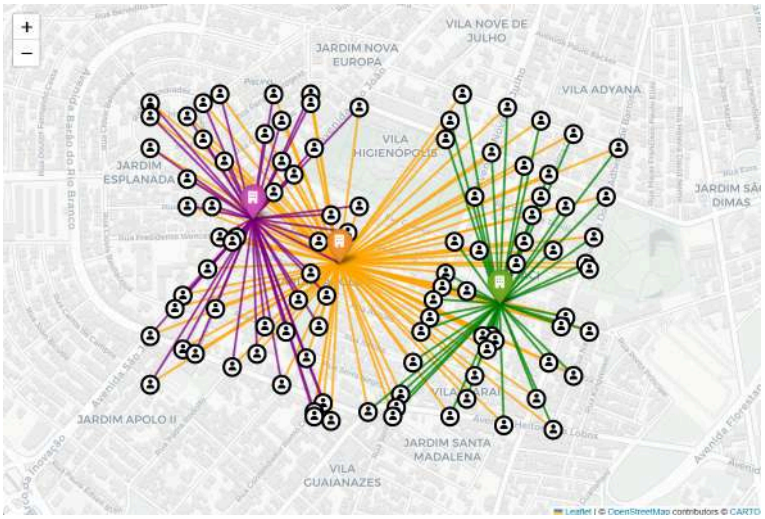


Figure B.5: Graphical representation of the solution for the Fault-tolerant P-Median Problem with $p = 3$ and $r = 2$.

B.3.3 Exercise 3.3

Listing B.16 provides a Julia code to solve the Set Covering Location Problem (SCLP) for the instance `coordinates.csv` with a coverage radius of 400 meters. The code uses the JuMP modeling language and the HiGHS solver to find the optimal location of facilities that minimizes the number of facilities opened while ensuring that all demand points are covered within the specified radius. The solution is visualized using a custom plotting function that shows the selected facilities and their coverage areas on a map, illustrating how the solution effectively covers all demand points.

```
using JuMP  # Modeling language
using HiGHS # Solver
using CSV   # Data handling

# Include utility functions for plotting the solution
include("utils/sclp_utils.jl")

function solve_sclp(file_path; radius = 200)
  # Load coordinates
  coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)

  # Create distance matrix
  D = create_distance_matrix(coordinates)

```

```
  # Create coverage matrix considering the defined radius
  C = D .< radius

  # Number of demand/facility points
  n = size(D, 1)

  # Index sets
  I, J = 1:n, 1:n

  # Create model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Variables
  @variable(model, x[j in J], Bin)

  # Objective: minimize number of facilities opened
  @objective(model, Min, sum(x[j] for j in J))

  # Constraint: each demand point is covered by at least one
facility
  @constraint(model, [i in I], sum(C[i,j] * x[j] for j in J) >= 1)

  # Solve the model
  JuMP.optimize!(model)

  # Get solution vector
  x_vals = JuMP.value.(x)

  # Get indices of selected facilities
  selected_facilities = [j for j in J if x_vals[j] > 0.5]

  # Display solution
  println("Number of facilities opened: ",
Int(JuMP.objective_value(model)))
  println("Selected facility indices: ", selected_facilities)

  # Display solution on map
  fmap = plot_solution(selected_facilities, coordinates, radius)
```

```
  display(fmap)
end

# Example usage
solve_sclp("data/coordinates.csv", radius = 400)
```

Listing B.16: Solving the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters.

Listing B.17 provides the output of the code above, showing the number of facilities opened and the indices of the selected facilities for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters.

Output

```
Number of facilities opened: 4
Selected facility indices: [1, 12, 50, 73]
```

Listing B.17: Output of the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters.

Figure B.6 provides a graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters. The figure shows the locations of the selected facilities and their coverage areas on a map, illustrating how the solution effectively covers all demand points within the specified radius.

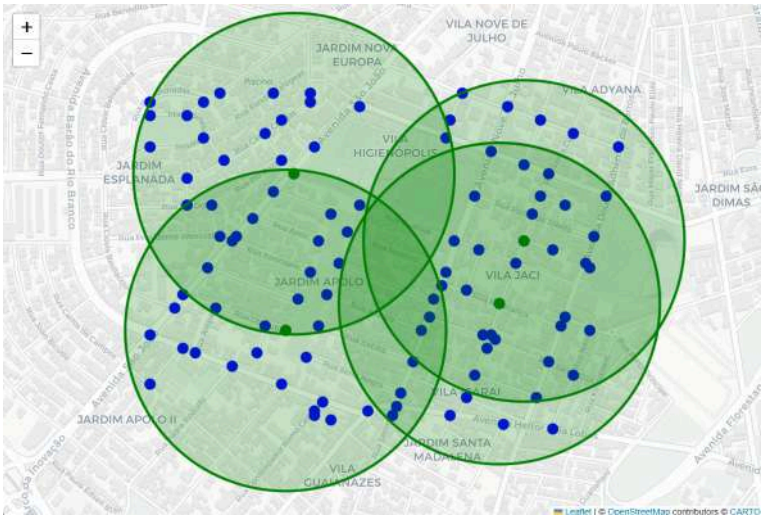


Figure B.6: Graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters.

B.3.4 Exercise 3.4

We use the following mathematical model for the SCLP considering the intersections (overlap) between the covers, using the approach presented in the article [37].

$$\min \sum_{j \in J} x_j + u \tag{B.9.1}$$

$$\text{s.t.} \sum_{j \in J} c_{ij} x_j \geq 1 \qquad \forall i \in I \tag{B.9.2}$$

$$\sum_{i \in I} \sum_{j \in J} (x_i + x_j - 1) \left| \Psi_{ij} - z \right| \leq u \tag{B.9.3}$$

$$x_j \in \{0, 1\} \qquad \forall j \in J \tag{B.9.4}$$

$$u \in \mathbb{R} \tag{B.9.5}$$

The continuous variable u is used to represent overlap penalties, c_{ij} is a boolean coverage matrix that represents the if facility (j) covers point (i). Finally, Ψ_{ij} represents the Jaccard Matrix, wich measures overlap between the points covered by facilities (i) and (j), and is calculated as:

$$\Psi_{ij} = \frac{|S_i \cap S_j|}{|S_i \cup S_j|} \tag{B.10}$$

where S_i is the set of points covered by a facility at point (i). Finally, z is the overlap-control parameter:

- (z=0): Highly overlapping facilities are favored.
- (z=1): Large overlap increases the penalty, spreading coverage zones and reducing unnecessary overlaps

Why they use $(x_i + x_j - 1)$?

The authors try to control the pairwise similarity structure among selected facilities. The paper explicitly discusses this. They intentionally allow negative contributions when neither facility is selected. This creates a reward/penalty mechanism:

- selected pairs contribute positively,
- unselected pairs contribute negatively.

The authors use this to bias the optimization toward or against overlap patterns depending on (z). So although unconventional from a pure MILP modeling perspective, it is deliberate in their formulation.

B.3.4.1 a) Solving the SCLP with overlap control using $z = 0$ for the overlap parameter

Listing B.18 provides a Julia code to solve the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and an overlap parameter $z = 0$ for the instance `coordinates.csv`. The code uses the JuMP modeling language and the HiGHS solver to find the optimal location of facilities that minimizes the number of facilities opened while ensuring that all demand points are covered within the specified radius, while also controlling for overlap between coverage areas. The solution is visualized using a custom plotting function that shows the selected facilities and their coverage areas on a map, illustrating how the solution effectively covers all demand points while minimizing unnecessary overlaps.

```julia
using JuMP      # Modeling language
using HiGHS     # Solver
using CSV       # Data handling
using Distances # Distance computations

# Include utility functions for plotting and distance matrix creation
include("utils/sclp_utils.jl")

# Function to solve SCLP with overlap control
function solve_sclp_overlap(file_path; radius = 200, z = 1)
  # Load coordinates
  coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)

  # Create distance matrix
  D = create_distance_matrix(coordinates)

  # Create coverage matrix considering the defined radius
  C = D .< radius

  # Calculate jaccard's matrix between points coverages
  Ψ = Distances.pairwise(Jaccard(), eachrow(C))

  # Number of demand/facility points
  n = size(C, 1)

  # Index sets
  I, J = 1:n, 1:n
```

```
28
29    # Create model
30    model = JuMP.Model(HiGHS.Optimizer)
31
32    # Silent mode (solver output is not printed)
33    JuMP.set_silent(model)
34
35    # Variables
36    @variable(model, x[j in J], Bin)
37    @variable(model, u)
38
39    # Objective: minimize number of facilities opened
40    @objective(model, Min, sum(x[j] for j in J) + u)
41
42    # Constraint: each demand point is covered by at least one
      facility
43    @constraint(model, [i in I], sum(C[i,j] * x[j] for j in J) >= 1)
44
45    # Constraint: Jaccard distance condition
46    @constraint(model, sum((x[i] + x[j] - 1) * abs(Ψ[i,j] - z) for i
      in I, j in J) <= u)
47
48    # Solve the model
49    JuMP.optimize!(model)
50
51    # Get solution vector
52    x_vals = JuMP.value.(x)
53
54    # Get indices of selected facilities
55    selected_facilities = [j for j in J if x_vals[j] > 0.5]
56
57    # Display solution
58    println("Number of facilities opened: ",
      length(selected_facilities))
59    println("Selected facility indices: ", selected_facilities)
60
61    # Display solution on map
62    fmap = plot_solution(selected_facilities, coordinates, radius)
63    display(fmap)
64 end
65
```

```
# Example usage z = 0 (more overlap)
solve_sclp_overlap("data/coordinates.csv", radius = 400, z = 0)
```

Listing B.18: Solving the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=0.

Listing B.19 provides the output of the code above, showing the number of facilities opened and the indices of the selected facilities for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and an overlap parameter $z = 0$.

Output

```
Number of facilities opened: 4
Selected facility indices: [1, 7, 12, 73]
```

Listing B.19: Output of the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=0.

Figure B.7 provides a graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and an overlap parameter $z = 0$. The figure shows the locations of the selected facilities and their coverage areas on a map, illustrating how the overlap parameter z affects the coverage.

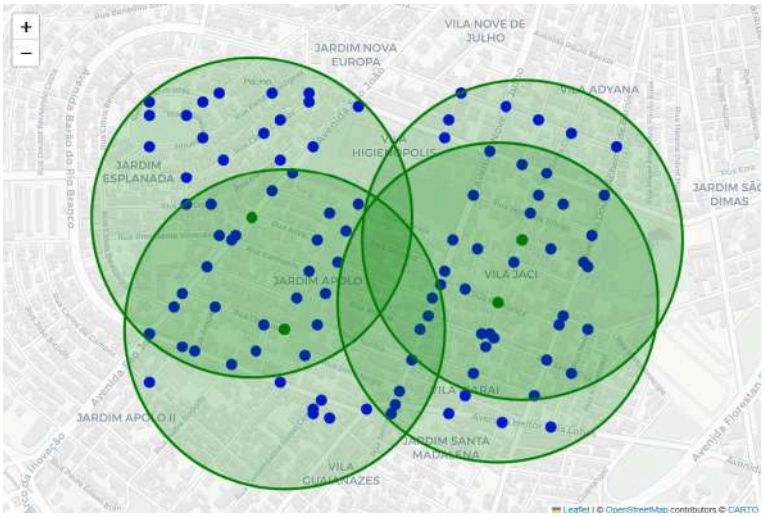


Figure B.7: Graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=0.

B.3.4.2 b) Solving the SCLP with overlap control using $z = 1$ for the overlap parameter

Listing B.20 provides a Julia code to solve the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and an overlap parameter $z = 1$ for the instance `coordinates.csv`. It uses the code provided in Listing B.18 but with a different value for the overlap parameter, which penalizes large overlaps between coverage areas.

```
# Example usage z = 1 (less overlap)
solve_sclp_overlap("data/coordinates.csv", radius = 400, z = 1)
```

Listing B.20: Solving the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=1.

Listing B.21 provides the output of the code above, showing the number of facilities opened and the indices of the selected facilities for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and an overlap parameter $z = 1$.

Output

```
Number of facilities opened: 4
Selected facility indices: [5, 43, 50, 97]
```

Listing B.21: Output of the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=1.

Figure B.8 provides a graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and an overlap parameter $z = 1$. The figure shows the locations of the selected facilities and their coverage areas on a map, illustrating how the overlap parameter z affects the coverage.

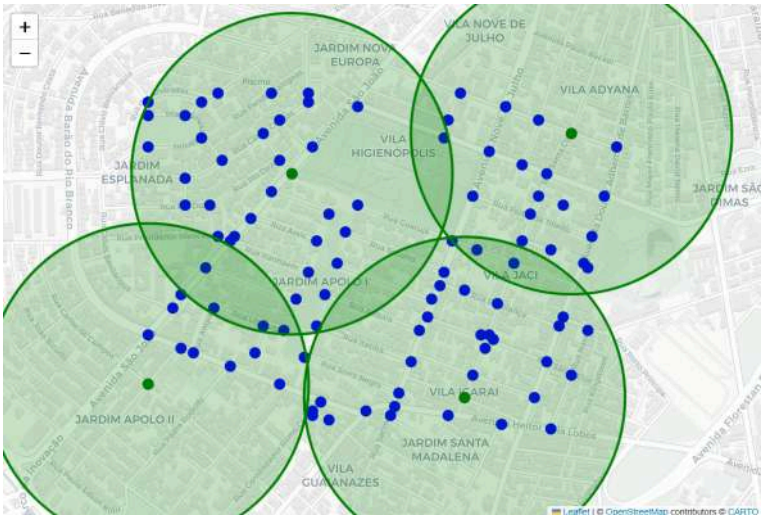


Figure B.8: Graphical representation of the solution for the Set Covering Location Problem (SCLP) with a coverage radius of 400 meters and overlap parameter z=1.

B.3.5 Exercise 3.5

Listing B.22 provides a Julia code to solve the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and $p = 3$ facilities for the instance `coordinates.csv`. The code uses the JuMP modeling language and the HiGHS solver to find the optimal location of facilities that maximizes the coverage of demand points within the specified radius while limiting the number of facilities opened to three. The solution is visualized using a custom plotting function that shows the selected facilities and their covered demand points on a map, illustrating how the solution effectively maximizes coverage given the constraints.

```julia
using JuMP  # Modeling language
using HiGHS # Solver
using CSV   # For reading CSV files

# Include utility functions for plotting the solution
include("utils/mclp_utils.jl")

# Function to solve the MCLP given coverage radius and number of facilities
function solve_mclp(file_path; radius = 400, p = 3)
  # Load coordinates
  coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)

  # Create distance matrix
  D = create_distance_matrix(coordinates)
```

```
15
16    # Create coverage matrix considering the defined radius
17    C = D .< radius
18
19    # Number of demand/facility points
20    n = size(C, 1)
21
22    # Index sets
23    I, J = 1:n, 1:n
24
25    # Create a vector of equal demand for each point
26    d = ones(n)
27
28    # Create model
29    model = JuMP.Model(HiGHS.Optimizer)
30
31    # Silent mode (solver output is not printed)
32    JuMP.set_silent(model)
33
34    # Variables
35    @variable(model, x[j in J], Bin)
36    @variable(model, y[i in I], Bin)
37
38    # Objective: maximize covered demand
39    @objective(model, Max, sum(d[i]*y[i] for i in I))
40
41    # Constraint: exactly p facilities are opened
42    @constraint(model, sum(x[j] for j in J) == p)
43
44    # Constraint: each demand point is covered by at least one
      facility
45    @constraint(model, [i in I], sum(C[i,j] * x[j] for j in J) >=
      y[i])
46
47    # Solve the model
48    JuMP.optimize!(model)
49
50    # Get solution vectors
51    x_vals = JuMP.value.(x)
52    y_vals = JuMP.value.(y)
53
```

```
  # Get indices of selected facilities
  selected_facilities = [j for j in J if x_vals[j] > 0.5]
  covered_points = [i for i in I if y_vals[i] > 0.5]

  # Calculate coverage percentage
  total_covered = length(covered_points)
  coverage_percentage = round(100*total_covered/n, digits=2)

  # Display solution
  println("Facilities opened: $selected_facilities")
  println("Points covered: $total_covered/$n ($coverage_percentage %)")

  # Display solution on map
  fmap = plot_solution(selected_facilities,covered_points, coordinates, radius)
  display(fmap)
end

# Example usage
solve_mclp("data/coordinates.csv", radius = 400, p = 3)
```

Listing B.22: Solving the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and p = 3.

Listing B.23 provides the output of the code above, showing the indices of the facilities opened and the number and percentage of demand points covered for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and $p = 3$ facilities.

Output

```
Facilities opened: [7, 40, 73]
Points covered: 101/102 (99.02 %)
```

Listing B.23: Output of the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and $p = 3$.

Figure B.9 provides a graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and $p = 3$ facilities. The figure shows the locations of the selected facilities and the demand

points they cover on a map, illustrating how the solution effectively maximizes coverage given the constraints.

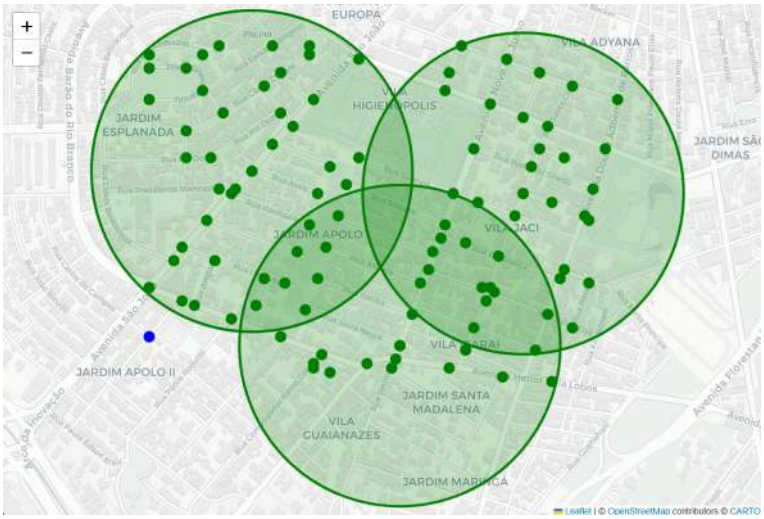


Figure B.9: Graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and p = 3.

B.3.6 Exercise 3.6

We use the following mathematical model for the MCLP considering the intersections (overlap) between the covers, using the approach presented in the article [37].

$$\max \sum_{i \in I} y_i - u \tag{B.11.1}$$

$$\text{s.t.} \quad \sum_{i \in I} x_i = p \tag{B.11.2}$$

$$\sum_{j \in J} c_{ij} x_j \geq y_i \qquad \forall i \in I \tag{B.11.3}$$

$$\sum_{i \in I} \sum_{j \in J} \left(x_i + x_j - 1\right) \left|\Psi_{ij} - z\right| \leq u \tag{B.11.4}$$

$$x_j, y_i \in \{0, 1\} \qquad \forall i \in I, j \in J \tag{B.11.5}$$

$$u \in \mathbb{R} \tag{B.11.6}$$

B.3.6.1 a) Solving the MCLP with overlap control using $z = 0$ for the overlap parameter

Listing B.24 provides a Julia code to solve the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and an overlap parameter $z = 0$ for the instance `coordinates.csv`. The code uses the JuMP modeling language

and the HiGHS solver to find the optimal location of facilities that maximizes the coverage of demand points within the specified radius while limiting the number of facilities opened to three and controlling for overlap between coverage areas. The solution is visualized using a custom plotting function that shows the selected facilities and their covered demand points on a map, illustrating how the solution effectively maximizes coverage given the constraints while minimizing unnecessary overlaps.

```
using JuMP      # Modeling language
using HiGHS     # Solver
using CSV       # For reading CSV files
using Distances # Distance computations

# Include utility functions for plotting the solution
include("utils/mclp_utils.jl")

# Function to solve the MCLP with a given coverage radius and number of facilities
function solve_mclp(file_path; radius = 400, p = 3, z = 1)
  # Load coordinates
  coordinates = CSV.read(file_path, CSV.Tables.matrix, header=false)

  # Create distance matrix
  D = create_distance_matrix(coordinates)

  # Create coverage matrix considering the defined radius
  C = D .< radius

  # Calculate jaccard's matrix between points coverages
  Ψ = Distances.pairwise(Jaccard(), eachrow(C))

  # Number of demand/facility points
  n = size(C, 1)

  # Index sets
  I, J = 1:n, 1:n

  # Create a vector of equal demand for each point
  d = ones(n)

  # Create model
```

```
33  model = JuMP.Model(HiGHS.Optimizer)
34
35  # Silent mode (solver output is not printed)
36  JuMP.set_silent(model)
37
38  # Variables
39  @variable(model, x[j in J], Bin)
40  @variable(model, y[i in I], Bin)
41  @variable(model, u)
42
43  # Objective: maximize covered demand
44  @objective(model, Max, sum(d[i]*y[i] for i in I) - u)
45
46  # Constraint: exactly p facilities are opened
47  @constraint(model, sum(x[j] for j in J) == p)
48
49  # Constraint: each demand point is covered by at least one
    facility
50  @constraint(model, [i in I], sum(C[i,j] * x[j] for j in J) >=
    y[i])
51
52  # Constraint: Jaccard distance condition
53  @constraint(model, sum((x[i] + x[j] - 1) * abs(Ψ[i,j] - z) for i
    in I, j in J) <= u)
54
55  # Solve the model
56  JuMP.optimize!(model)
57
58  # Get solution vectors
59  x_vals = JuMP.value.(x)
60  y_vals = JuMP.value.(y)
61
62  # Get indices of selected facilities
63  selected_facilities = [j for j in J if x_vals[j] > 0.5]
64  covered_points = [i for i in I if y_vals[i] > 0.5]
65
66  # Calculate coverage percentage
67  total_covered = length(covered_points)
68  coverage_percentage = round(100*total_covered/n, digits=2)
69
70  # Display solution
```

```
  println("Facilities opened: $selected_facilities")
  println("Points covered: $total_covered/$n ($coverage_percentage %)")

  # Display solution on map
  fmap = plot_solution(selected_facilities,covered_points, coordinates, radius)
  display(fmap)
end

# Example usage z=0 (more overlap)
solve_mclp("data/coordinates.csv", radius = 400, p = 3, z = 0)
```

Listing B.24: Solving the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 0$.

Listing B.25 provides the output of the code above, showing the indices of the facilities opened and the number and percentage of demand points covered for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and an overlap parameter $z = 0$.

Output

```
Facilities opened: [7, 38, 101]
Points covered: 99/102 (97.06 %)
```

Listing B.25: Output of the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 0$.

Figure B.10 provides a graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and an overlap parameter $z = 0$. The figure shows the locations of the selected facilities and the demand points they cover on a map, illustrating how the overlap parameter z affects the coverage and the distribution of selected facilities.

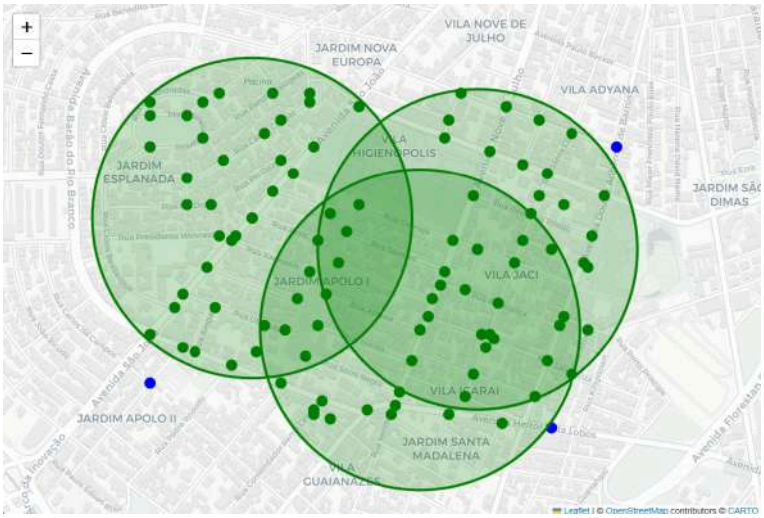


Figure B.10: Graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 0$.

B.3.6.2 b) Solving the MCLP with overlap control using $z = 1$ for the overlap parameter

Listing B.26 provides a Julia code to solve the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and an overlap parameter $z = 1$ for the instance `coordinates.csv`. It uses the model presented in Listing B.24, but with a different value for the overlap parameter z.

```
# Example usage z=1 (less overlap)
solve_mclp("data/coordinates.csv", radius = 400, p = 3, z = 1)
```

Listing B.26: Solving the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 1$.

Listing B.27 provides the output of the code above, showing the indices of the facilities opened and the number and percentage of demand points covered for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and an overlap parameter $z = 1$.

Output

```
Facilities opened: [5, 60, 72]
Points covered: 54/102 (52.94 %)
```

Listing B.27: Output of the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 1$.

Figure B.11 provides a graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and an overlap parameter $z = 1$. The figure shows the locations of the selected facilities and the demand points they cover on a map, illustrating how the overlap parameter z affects the coverage and the distribution of selected facilities, with less overlap compared to the case when $z = 0$.

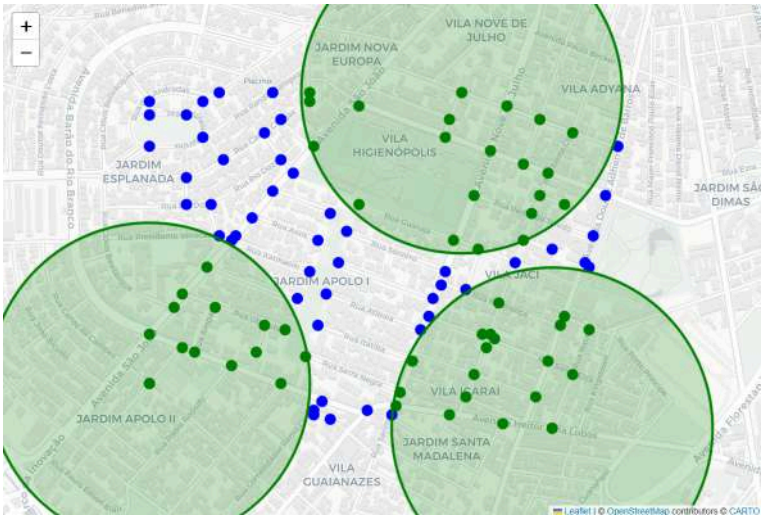


Figure B.11: Graphical representation of the solution for the Maximal Coverage Location Problem (MCLP) with a coverage radius of 400 meters and overlap control using $z = 1$.

B.3.7 Exercise 3.7

Listing B.28 provides a Julia code to solve the TEAM model for the instance `ambulances.h5`. The code uses the JuMP modeling language and the HiGHS solver to determine the optimal location of basic and advanced care ambulances while maximizing the population coverage. The solution is visualized using a custom plotting function that shows the selected facilities and their coverage areas on a map, illustrating how the solution effectively covers the population given the constraints of ambulance types and their respective coverage times.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using HDF5   # For reading .h5 files
using Printf # For formatted printing

# Include utility functions for plotting
include("utils/team_utils.jl")

```

```
# Function to solve the ambulance location problem with basic and advanced care
function solve_team_model(file_path; basic_time = 15, basic_total = 4,
                          advanced_time = 25, advanced_total = 2)

  # Load problem data from HDF5 file
  # - duration_matrix: matrix representing route duration
  # - polulation: vector of population at each demand point
  duration_matrix = h5read(file_path, "duration_matrix")
  population = h5read(file_path, "districts_population")

  # Problem parameters
  demands = length(population)          # Number of demand points
  facilities = size(duration_matrix,1) # Number of facilities

  # Index sets
  F = 1:facilities
  D = 1:demands

  # Create coverage matrix for basic care
  C_B = duration_matrix .<= basic_time

  # Create coverage matrix for advanced care
  C_A = duration_matrix .<= advanced_time

  # Create model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Variables
  @variable(model, xB[f in 1:facilities], Bin)
  @variable(model, xA[f in 1:facilities], Bin)
  @variable(model, y[d in 1:demands], Bin)

  # Objective: maximize covered population
  @objective(model, Max, sum(population[d] * y[d] for d in D))

```

```
47   # Constraint: total number of ambulances for basic and advanced
     care
48   @constraint(model, sum(xB) == basic_total)
49   @constraint(model, sum(xA) == advanced_total)
50
51   # Demand coverage constraints for basic care
52   @constraint(model,
53               [d in D],
54               sum(C_B[f, d] * xB[f] for f in F) >= y[d])
55
56   # Demand coverage constraints for advanced care
57   @constraint(model,
58               [d in D],
59               sum(C_A[f, d] * xA[f] for f in F) >= y[d])
60
61   # Advanced only if basic exists
62   @constraint(model, [f in F], xA[f] <= xB[f])
63
64   # Solve model
65   JuMP.optimize!(model)
66
67   # Extract results
68   xB_opt = JuMP.value.(xB)
69   xA_opt = JuMP.value.(xA)
70   y_opt = JuMP.value.(y)
71   z_opt = Int(JuMP.objective_value(model))
72
73   # Calculate total covered population
74   total_population = sum(population)
75   coverage = round((z_opt / total_population) * 100, digits=2)
76
77   println("=== Optimization Results ===\n")
78   println("Population covered by both types of ambulances: ",
79           "$z_opt / $total_population ($coverage%)")
80
81   println("\n=== Facilities selected for each ambulance type
     ===\n")
82   selected_basic = findall(xB_opt .> 0.5)
83   println("Basic Care: $selected_basic")
84
85   selected_advanced = findall(xA_opt .> 0.5)
```

```
  println("Advanced Care: $selected_advanced")

  println("\n=== Ambulances coverage by district ===\n")
  println("="^60)
  println(" District  |  Basic Care  |  Advanced Care  |
Population")
  println("="^60)

  for d in D
    percent = round((population[d]/total_population)*100, digits=2)
    if y_opt[d] < 0.5
      @printf("%10d | %12s | %15s | %5s (%2.2f%%)\n", d,
              "Not Covered", "Not Covered",
              population[d], percent)
    else
      basic = [j for j in selected_basic if C_B[j, d]]
      advanced = [j for j in selected_advanced if C_A[j, d]]
      @printf("%10d | %12s | %15s | %5s (%2.2f%%)\n", d,
              "$basic", "$advanced",
              population[d], percent)
    end
  end

  # Plot the solution on a map
  fmap = plot_solution(file_path, xB_opt, xA_opt, y_opt)
  display(fmap)
end

# Example usage
solve_team_model("data/sjc_ambulances.h5",
                 basic_time = 15, basic_total = 4,
                 advanced_time = 25, advanced_total = 2)
```

Listing B.28: Solving the TEAM model for instance `ambulances.h5`.

Listing B.29 provides the output of the code above, showing the optimization results for the TEAM model, including the population covered by both types of ambulances, the facilities selected for each ambulance type, and the coverage of ambulances by district.

Output

```
=== Optimization Results ===

Population covered by both types of ambulances: 639288 / 680044
(94.01%)

=== Facilities selected for each ambulance type ===

Basic Care: [4, 5, 8, 9]
Advanced Care: [4, 8]

=== Ambulances coverage by district ===

============================================================
 District  |  Basic Care  |  Advanced Care  |  Population
============================================================
         1 |          [5] |          [4, 8] | 40909 (6.02%)
         2 |  Not Covered |     Not Covered |   342 (0.05%)
         3 |       [4, 5] |          [4, 8] | 16012 (2.35%)
         4 |          [5] |          [4, 8] |   167 (0.02%)
         5 |       [4, 5] |          [4, 8] | 15544 (2.29%)
         6 |  Not Covered |     Not Covered |   696 (0.10%)
         7 |       [4, 5] |          [4, 8] | 21916 (3.22%)
         8 |       [4, 5] |          [4, 8] | 39506 (5.81%)
         9 |  Not Covered |     Not Covered |   536 (0.08%)
        10 |       [8, 9] |          [4, 8] | 32782 (4.82%)
        11 |    [4, 5, 8] |          [4, 8] | 14260 (2.10%)
        12 |          [8] |          [4, 8] | 21086 (3.10%)
        13 |       [4, 8] |          [4, 8] |  2560 (0.38%)
        14 |       [4, 8] |          [4, 8] | 13866 (2.04%)
        15 |          [4] |          [4, 8] | 56532 (8.31%)
        16 |          [4] |          [4, 8] | 33184 (4.88%)
        17 |          [4] |          [4, 8] | 27804 (4.09%)
        18 |          [4] |          [4, 8] | 62190 (9.14%)
        19 |          [4] |          [4, 8] | 38549 (5.67%)
        20 |          [4] |          [4, 8] | 23647 (3.48%)
        21 |          [4] |          [4, 8] |   839 (0.12%)
        22 |  Not Covered |     Not Covered | 13453 (1.98%)
        23 |  Not Covered |     Not Covered | 25467 (3.74%)
        24 |    [4, 5, 8] |          [4, 8] | 18671 (2.75%)
        25 |          [5] |             [4] |  3718 (0.55%)
```

```
 26 |          [5] |             [4] |   219 (0.03%)
 27 |       [4, 5] |          [4, 8] | 16856 (2.48%)
 28 |          [9] |          [4, 8] | 47193 (6.94%)
 29 |          [8] |          [4, 8] |  5447 (0.80%)
 30 |          [8] |          [4, 8] | 35828 (5.27%)
 31 |          [8] |          [4, 8] | 11763 (1.73%)
 32 |          [9] |          [4, 8] | 20613 (3.03%)
 33 |  Not Covered |     Not Covered |   262 (0.04%)
 34 |          [9] |          [4, 8] | 17627 (2.59%)
```

Listing B.29: Output of the TEAM model.

Figure B.12 provides a graphical representation of the solution for the TEAM model. The figure shows the locations of the selected facilities for basic and advanced care ambulances, as well as the demand points they cover on a map. This visualization helps to understand how the solution effectively covers the population given the constraints of ambulance types and their respective coverage times.

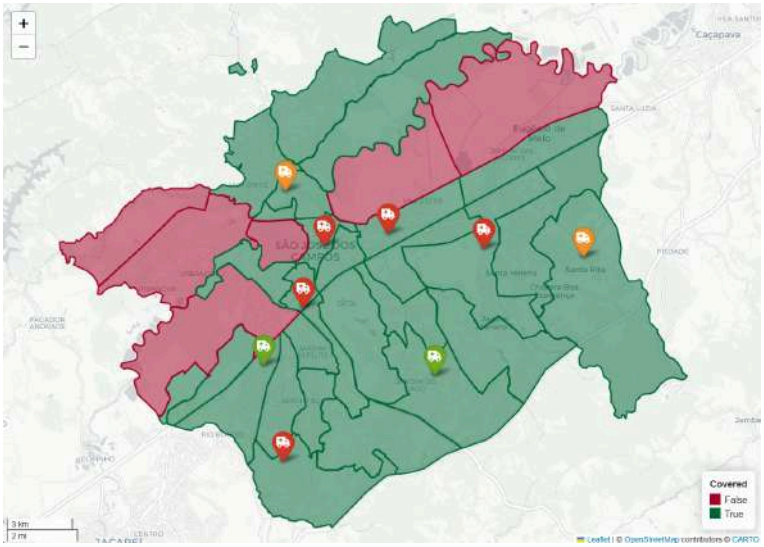


Figure B.12: Graphical representation of the solution for the TEAM model.

B.4 Chapter 4

B.4.1 Exercise 4.1

B.4.1.1 a) TSP without subtour elimination constraints

Listing B.30 presents a Julia implementation of the TSP without subtour elimination constraints, using the HiGHS solver. The code reads the distance matrix from

an HDF5 file, formulates the TSP model, and extracts the routes from the solution. The optimal value of the objective function is also printed.

```
using JuMP  # Modeling language
using HiGHS # Solver
using HDF5  # For reading HDF5 files

# Load utility functions for plotting
include("utils/tsp_utils.jl")

# Function to solve the TSP without subtour elimination constraints
function solve_tsp_no_subtour(file_path)
  # Load the distance matrix from the HDF5 file
  D = h5read(file_path, "distance_matrix")

  # Number of locations
  n = size(D, 1)

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)
  JuMP.set_silent(model)

  # Define the decision variables
  @variable(model, x[1:n, 1:n], Bin)

  # Objective function: minimize total distance
  @objective(model, Min, sum(D[i, j] * x[i, j] for i in 1:n, j in
1:n))

  # Constraints: no self-loops
  @constraint(model, [i=1:n], x[i, i] == 0)

  # Constraints: each location visited once
  @constraint(model, [j=1:n], sum(x[i, j] for i in 1:n) == 1)

  # Constraints: each location departed once
  @constraint(model, [i=1:n], sum(x[i, j] for j in 1:n) == 1)

  # Solve
  JuMP.optimize!(model)

  # Get the values of the decision variables
```

```
  x_opt = JuMP.value.(x)

  # Extract the routes from the solution
  visited = fill(false, n)
  routes = Vector{Vector{Int}}()
  for start in 1:n
    if !visited[start]
      route = Int[start]
      visited[start] = true
      nxt = findfirst(x_opt[start, :] .> 0.5)
      while nxt !== nothing && !visited[nxt]
        push!(route, nxt)
        visited[nxt] = true
        nxt = findfirst(x_opt[nxt, :] .> 0.5)
      end
      push!(routes, route)
      println("Route: ", route)
    end
  end

  # Get the optimal value of the objective function
  z_opt = JuMP.objective_value(model)
  println("Optimal value: $z_opt meters")

  # Plot the solution on the map
  fmap = plot_tsp_solution(file_path, x_opt)
  display(fmap)
end

# Example usage
solve_tsp_no_subtour("data/tsp_exercise.h5")
```

Listing B.30: Solving the TSP without subtour elimination constraints for `tsp_exercise.h5`.

Listing B.31 shows the output of the TSP without subtour elimination constraints for the dataset `tsp_exercise.h5`. The solution consists of three routes, and the optimal value of the objective function is 3240.8 meters.

Output

```
Route: [1, 6, 2]
```

```
Route: [3, 8, 5]
Route: [4, 7]
Optimal value: 3240.8 meters
```

Listing B.31: Output of the TSP without subtour elimination constraints for `tsp_exercise.h5`.

Figure B.13 presents a graphical representation of the solution of the TSP without subtour elimination constraints for the dataset `tsp_exercise.h5`. The figure shows the three routes obtained from the solution, illustrating the presence of subtours due to the lack of subtour elimination constraints.

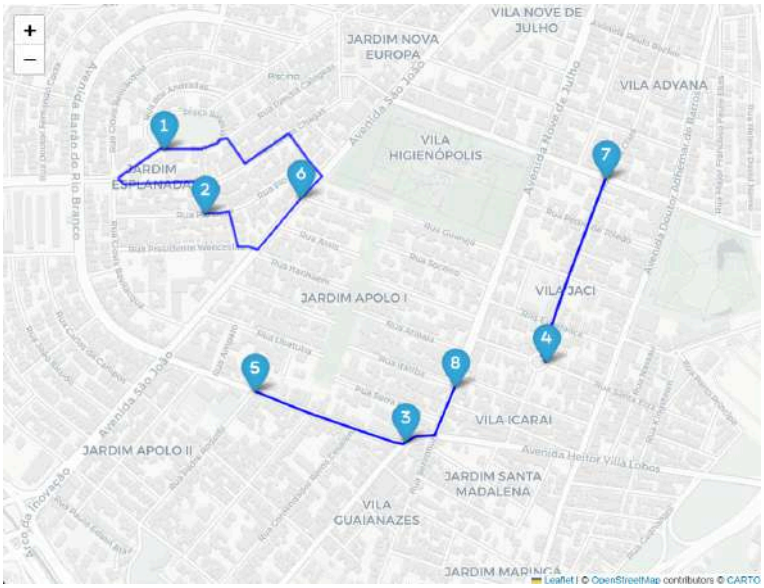


Figure B.13: Graphical representation of the solution of the TSP without subtour elimination constraints for `tsp_exercise.h5`.

B.4.1.2 b) TSP with Miller-Tucker-Zemlin constraints

Listing B.32 provides a Julia implementation of the TSP with Miller-Tucker-Zemlin (MTZ) constraints, using the HiGHS solver. The code reads the distance matrix from an HDF5 file, formulates the TSP model with MTZ constraints, and extracts the route from the solution. The optimal value of the objective function is also printed.

```
1 using JuMP  # Modeling language
2 using HiGHS # Solver
3 using HDF5  # For reading HDF5 files
4
```

```
# Load utility functions for plotting
include("utils/tsp_utils.jl")

# Function to solve the TSP with MTZ subtour elimination constraints
function solve_tsp_mtz(file_path)
  # Load the distance matrix from the HDF5 file
  D = h5read(file_path, "distance_matrix")

  # Number of locations
  n = size(D, 1)

  # Create model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Define the decision variables
  @variable(model, x[1:n, 1:n], Bin)

  # Define MTZ variables
  @variable(model, 1 <= u[1:n] <= n)

  # Objective function: minimize total distance
  @objective(model, Min, sum(D[i,j] * x[i,j] for i in 1:n, j in
1:n))

  # Avoid self-loops
  @constraint(model, [i=1:n], x[i,i] == 0)

  # Each column sums to 1 (each location visited once)
  @constraint(model, [j=1:n], sum(x[i,j] for i in 1:n) == 1)

  # Each row sums to 1 (each location departs once)
  @constraint(model, [i=1:n], sum(x[i,j] for j in 1:n) == 1)

  # Breaking symmetry by fixing the first city as the starting point
  @constraint(model, u[1] == 1)

  # MTZ constraints
```

```
  @constraint(model, [i=2:n, j=2:n; i != j], u[j] >= u[i] + 1 + n *
(x[i,j] - 1))

  # Run the solver
  JuMP.optimize!(model)

  # Get the values of the decision variables
  x_opt = JuMP.value.(x)

  # Extract the routes from the solution
  route = [1]
  while true
    to = findfirst(x_opt[route[end], :] .> 0.5)
    to == 1 ? break : push!(route, to)
  end
  println("Route: $route")

  # Get the optimal value of the objective function
  z_opt = JuMP.objective_value(model)
  println("Optimal value: $z_opt meters")

  # Plot the solution on the map
  fmap = plot_tsp_solution(file_path, x_opt)
  display(fmap)
end

# Example usage
solve_tsp_mtz("data/tsp_exercise.h5")
```

Listing B.32: Solving the TSP with MTZ constraints for `tsp_exercise.h5`.

Listing B.33 shows the output of the TSP with MTZ constraints for the dataset `tsp_exercise.h5`. The solution consists of a single route that visits all locations, and the optimal value of the objective function is 3693.3 meters.

Output

```
Route: [1, 2, 5, 3, 8, 4, 7, 6]
Optimal value: 3693.2999999999984 meters
```

Listing B.33: Output of the TSP with MTZ constraints for `tsp_exercise.h5`.

Figure B.14 presents a graphical representation of the solution of the TSP with MTZ constraints for the dataset `tsp_exercise.h5`. The figure shows the single route obtained from the solution, illustrating that the MTZ constraints successfully eliminate subtours and yield a valid TSP solution.

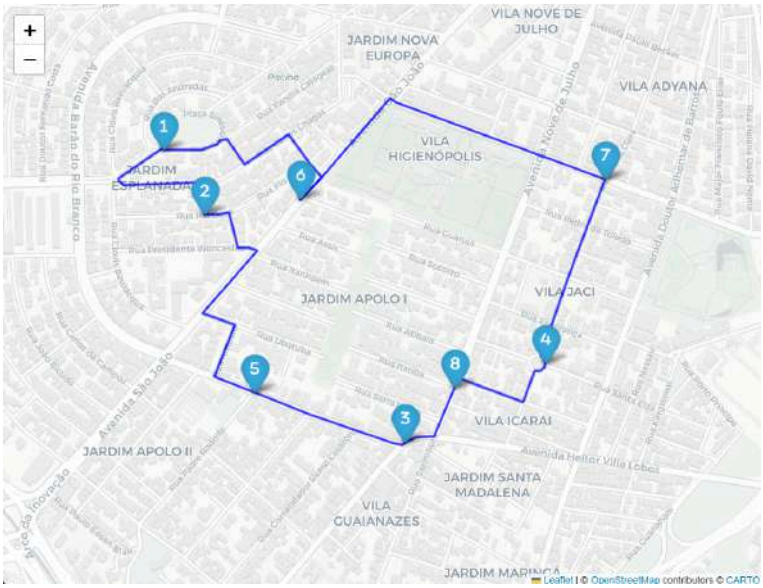


Figure B.14: Graphical representation of the solution of the TSP with MTZ constraints for `tsp_exercise.h5`.

B.4.1.3 c) TSP with lazy constraints

Listing B.34 provides a Julia implementation of the TSP with lazy constraints, using the GLPK solver. The code reads the distance matrix from an HDF5 file, formulates the TSP model with lazy constraints, and extracts the route from the solution. The optimal value of the objective function is also printed.

```
using JuMP   # Modeling language
using GLPK   # Solver
using HDF5   # For reading HDF5 files
using Graphs # For graph operations

# Load utility functions for plotting
include("utils/tsp_utils.jl")

# Define lazy constraints callback function
function tsp_lazy_callback(cb_data)
  # Get decision variable values
  x_val = [callback_value(cb_data, x[i,j]) for i in 1:n, j in 1:n]

```

```
14    # Build an undirected graph
15    g = Graphs.SimpleGraph(n)
16    for i in 1:n, j in 1:n
17      if i != j && x_val[i, j] > 0.5
18        Graphs.add_edge!(g, i, j)
19      end
20    end
21
22    # Find connected components
23    cc = Graphs.connected_components(g)
24
25    # If there are more than one connected components, we have a
      subtour
26    if length(cc) > 1
27      # Pick the smallest component (i.e., subtour)
28      minTour = sort(cc, by=length)[1]
29
30      # Initialize the left-hand side of the constraint
31      subtourLhs = AffExpr()
32
33      # Sum the decision variables corresponding to the subtour edges
34      for i in minTour, j in minTour
35        if i != j && x_val[i, j] > 0.5
36          subtourLhs += x[i, j]
37        end
38      end
39
40      # Add lazy constraint to eliminate subtour
41      constraint = @build_constraint(subtourLhs <= length(minTour) -
      1)
42
43      # Print the constraint for debugging
44      println("Adding lazy constraint: ", constraint.func, " <= ",
      constraint.set.upper)
45
46      # Submit the lazy constraint to the model
47      JuMP.MOI.submit(model, JuMP.MOI.LazyConstraint(cb_data),
      constraint)
48    end
49  end
50
```

```
# Load the distance matrix from the HDF5 file
D = h5read("data/tsp_exercise.h5", "distance_matrix")

# Number of locations
n = size(D, 1)

# Create model
model = JuMP.Model(GLPK.Optimizer)

# Silent mode (solver output is not printed)
JuMP.set_silent(model)

# Define the decision variables
@variable(model, x[1:n, 1:n], Bin)

# Objective function: minimize total distance
@objective(model, Min, sum(D[i,j] * x[i,j] for i in 1:n, j in 1:n))

# Constraints to eliminate self-loops
@constraint(model, [i=1:n], x[i,i] == 0)

# Each column sums to 1 (each location visited once)
@constraint(model, [j=1:n], sum(x[i,j] for i in 1:n) == 1)

# Each row sums to 1 (each location departs once)
@constraint(model, [i=1:n], sum(x[i,j] for j in 1:n) == 1)

# Set the lazy constraint callback function to the model
JuMP.MOI.set(model, JuMP.MOI.LazyConstraintCallback(),
tsp_lazy_callback)

# Run the solver
JuMP.optimize!(model)

# Get the values of the decision variables
x_opt = JuMP.value.(x)

# Extract the routes from the solution
route = [1]
while true
  to = findfirst(x_opt[route[end], :] .> 0.5)
```

```
  to == 1 ? break : push!(route, to)
end
println("Route: $route")

# Get the optimal value of the objective function
z_opt = JuMP.objective_value(model)
println("Optimal value: $z_opt meters")

# Plot the solution on the map
fmap = plot_tsp_solution("data/tsp_exercise.h5", x_opt)
display(fmap)
```

Listing B.34: Solving the TSP with lazy constraints for `tsp_exercise.h5`.

Listing B.35 shows the output of the TSP with lazy constraints for the dataset `tsp_exercise.h5`. The solution consists of a single route that visits all locations, and the optimal value of the objective function is 3693.3 meters. The output also includes the lazy constraints that were added during the optimization process to eliminate subtours.

Output

```
Adding lazy constraint: x[4,7] + x[7,4] <= 1.0
Adding lazy constraint: x[3,5] + x[5,3] <= 1.0
Adding lazy constraint: x[1,2] + x[2,1] <= 1.0
Route: [1, 2, 5, 3, 8, 4, 7, 6]
Optimal value: 3693.2999999999997 meters
```

Listing B.35: Output of the TSP with lazy constraints for `tsp_exercise.h5`.

Figure B.15 presents a graphical representation of the solution of the TSP with lazy constraints for the dataset `tsp_exercise.h5`. The figure shows the single route obtained from the solution, illustrating that the lazy constraints successfully eliminate subtours and yield a valid TSP solution.

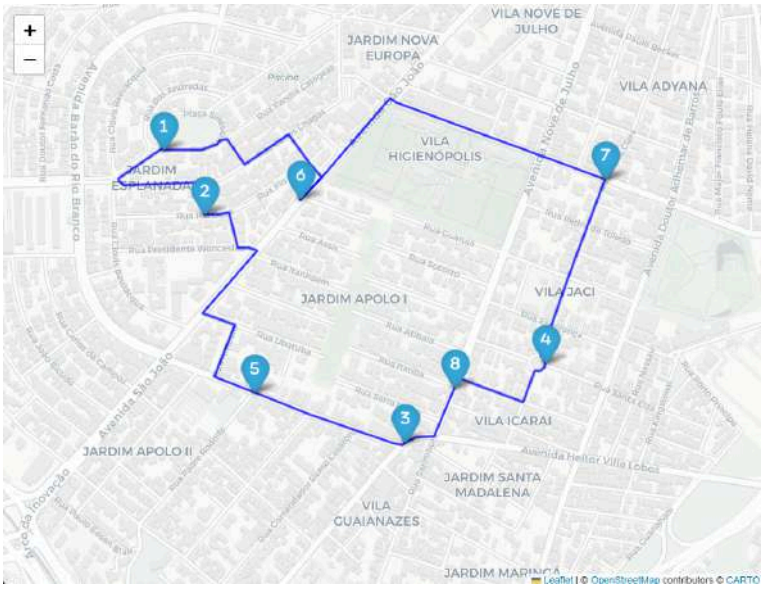


Figure B.15: Graphical representation of the solution of the TSP with lazy constraints for `tsp_exercise.h5`.

Discussion

Table B.1 summarizes the results obtained from solving the TSP without subtour elimination constraints, with Miller-Tucker-Zemlin constraints, and with lazy constraints. The table compares the optimal value of the objective function and the structure of the routes for each method.

Method	Objective	Solution
TSP-Rel	3240.8	[1, 6, 2],[3, 8, 5],[4, 7]
TSP-MTZ	3693.30	[1, 2, 5, 3, 8, 4, 7, 6]
TSP-Lazy	3693.30	[1, 2, 5, 3, 8, 4, 7, 6]

Table B.1: Comparison of TSP solutions with different models.

For TSP-Rel (No Subtour Elimination), the solution consists of multiple disconnected routes (subtours), such as [1, 6, 2], [3, 8, 5], and [4, 7]. The objective value (3240.8) is lower, but this is not a valid TSP tour since it does not visit all cities in a single cycle.

In the TSP-MTZ and TSP-Lazy (With Subtour Elimination) context, both methods yield a single valid tour: [1, 2, 5, 3, 8, 4, 7, 6]. The objective value (3693.30) is higher than the TSP-Rel, but the solution is feasible for the TSP, visiting all cities exactly once in a single cycle.

Subtour elimination constraints are essential in TSP formulations to prevent the occurrence of multiple disconnected cycles (subtours) in the solution. Without these constraints, the solver may find solutions that minimize the total distance but do not form a valid tour, as seen in the TSP-Rel result. Methods like MTZ and Lazy Constraints ensure that the solution is a single tour covering all cities, which is the true requirement of the TSP.

B.4.2 Exercise 4.2

Listing B.36 provides a Julia implementation of the TSP with MTZ constraints, using the HiGHS solver. The code reads the distance matrix from a CSV file, formulates the TSP model with MTZ constraints, and extracts the route from the solution. The optimal value of the objective function is also printed.

```
using JuMP                  # Modeling language
using HiGHS                 # Solver
using CSV                   # CSV file handling
using DataFrames            # Dataframe handling
using Graphs                # For graph data structures
using SimpleWeightedGraphs  # For weighted graphs
using Karnak                # For drawing graphs
using Colors                # For color manipulation

# Utility functions for TSP solution plotting
function plot_tsp_solution(graph::SimpleWeightedGraph,
x_opt::Matrix{Float64})
    # Extract the route from the solution
    route = [(coord[1], coord[2]) for coord in findall(x_opt .>
0.5)]

    # Define a function to check if an edge is part of the route
    function is_in_route(e, route)
        return (e.src, e.dst) in route || (e.dst, e.src) in route ?
true : false
    end

    # Draw graph
    @drawsvg begin
        background("white")
        sethue("black")
        fontsize(20)
        drawgraph(
```

```
            graph,
            vertexshapesizes = 12,
            vertexlabels = 1:Graphs.nv(graph),
            edgelabels = [graph.weights[e.src, e.dst] for e in
Graphs.edges(graph)],
            edgelabelfontsizes = 18,
            edgestrokecolors = [is_in_route(e, route) ?
Colors.RGB(1, 0, 0) : Colors.RGB(0, 0, 0) for e in
Graphs.edges(graph)],
            edgestrokeweights = [is_in_route(e, route) ? 3 : 1 for e
in Graphs.edges(graph)]
        )
    end
end

# Function to solve TSP using MTZ formulation
function solve_tsp_mtz(file_path)

    # Load the CSV file containing the TSP data
    df = CSV.read(file_path, DataFrame)

    # Load graph
    graph = SimpleWeightedGraph(df.From, df.To, df.Distance)

    # Number of vertices
    n = nv(graph)

    # Create model
    model = JuMP.Model(HiGHS.Optimizer)

    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)

    # Define the decision variables
    @variable(model, x[1:n, 1:n], Bin)

    # Define the MTZ variables
    @variable(model, u[2:n])

    # Objective function: minimize total distance
```

```
    @objective(model, Min, sum(Graphs.has_edge(graph, i, j) ? graph.weights[i, j] * x[i,j] : floatmax(Float64) * x[i,j] for i in 1:n, j in 1:n))

    # Constraints to eliminate self-loops
    @constraint(model, [i=1:n], x[i,i] == 0)

    # Each column sums to 1 (each location visited once)
    @constraint(model, [j=1:n], sum(x[i,j] for i in 1:n) == 1)

    # Each row sums to 1 (each location departs once)
    @constraint(model, [i=1:n], sum(x[i,j] for j in 1:n) == 1)

    # MTZ constraints
    @constraint(model, [i=2:n, j=2:n], n * x[i,j] - (n - 1) <= u[j] - u[i])

    # Run the solver
    JuMP.optimize!(model)

    # Get the values of the decision variables
    x_opt = JuMP.value.(x)

    # Extract the routes from the solution
    route = [1]
    while true
        to = findfirst(x_opt[route[end], :] .> 0.5)
        to == 1 ? break : push!(route, to)
    end
    println("Route: $route")

    # Get the optimal value of the objective function
    z_opt = JuMP.objective_value(model)
    println("Optimal value: $z_opt meters")

    # Plot the solution on the map
    plot_tsp_solution(graph, x_opt)
end

# Example usage
solve_tsp_mtz("data/tsp_exercise.csv")
```

Listing B.36: Solving the TSP with MTZ constraints for `tsp_exercise.csv`.

Listing B.37 shows the output of the TSP with MTZ constraints for the dataset `tsp_exercise.csv`. The solution consists of a single route that visits all locations, and the optimal value of the objective function is 22.0 meters.

Output

```
Route: [1, 7, 5, 4, 6, 3, 2]
Optimal value: 22.0 meters
```

Listing B.37: Output of the TSP with MTZ constraints for `tsp_exercise.csv`.

Figure B.16 presents a graphical representation of the solution of the TSP with MTZ constraints for the dataset `tsp_exercise.csv`. The figure shows the single route obtained from the solution, illustrating that the MTZ constraints successfully eliminate subtours and yield a valid TSP solution.

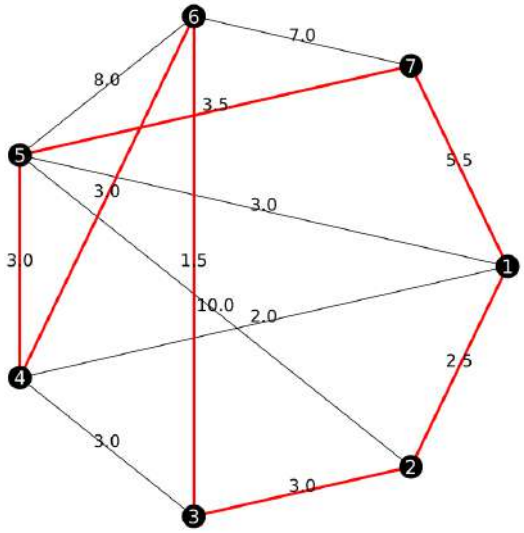


Figure B.16: Graphical representation of the solution of the TSP with MTZ constraints for `tsp_exercise.csv`.

B.5 Chapter 5

B.5.1 Exercise 5.1

Listing B.38 provides a complete implementation of the Maximum Independent Set Problem (MISP) using the JuMP modeling language and the HiGHS solver in Julia. The code includes functions to load a graph from a dataset, solve the MISP, and plot the solution.

```julia
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package
using Karnak # Graphs Visualization package
using Colors # Colors package

# Load utility function for plotting
include("utils/misp_utils.jl")

# Function to load dataset from a file
function load_dataset(file_path)
  # Open the file for reading
  f = open(file_path, "r")

  # Read the first line to get the number of points
  points = parse(Int, readline(f))

  # Read the next line to get the total of label positions
  positions = parse(Int, readline(f))

  # Calculate the total of labels
  labels = points * positions

  # Create a graph with the specified number of labels
  graph = Graphs.Graph(labels)

  # Vertex id
  vertex_id = 1

  # Read the next line to get the edge list
  while !eof(f)

    # Read the next vertex degree
    _ = parse(Int, readline(f))

    # Read the neighbors of the vertex
    neighbors = split(strip(readline(f)),' ')

    # Add edges to the graph for each neighbor
    for v in neighbors
      Graphs.add_edge!(graph, vertex_id, parse(Int, v))
```

```
    end

    # Increment the vertex id
    vertex_id += 1
  end

  close(f) # Close the file after reading

  return graph
end

# Function to solve the Maximum Independent Set Problem (MISP)
function solve_misp(file_path)
  # Load the graph from the file
  graph = load_dataset(file_path)

  # Total of vertices
  n = Graphs.nv(graph)

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Define the decision variables
  @variable(model, x[1:n], Bin)

  # Objective function: maximize the total of selected vertices
  @objective(model, Max, sum(x))

  # Independence constraint
  @constraint(model, [i = 1:n, j = 1:n; Graphs.has_edge(graph, i,
j)], x[i] + x[j] <= 1)

  # Run the solver
  JuMP.optimize!(model)

  # Independent set S
  S = findall(JuMP.value.(x) .>= 0.5)

```

```
  # Print solution
  println("Maximum independent set (S) with $(length(S)) vertices:")

  # Print 10 vertices per line
  for i in 1:length(S)
    i % 10 == 0 ? println() : print(S[i], " ")
  end

  # Plot solution
  plot_solution(graph, S)
end

# Example usage
solve_misp("data/d25_01.dat")
```

Listing B.38: Solving the MISP for dataset `d25_01.dat`.

Listing B.39 shows the output of the above code when executed with the dataset `d25_01.dat`. The output includes the maximum independent set (S) with 24 vertices.

Output

```
Maximum independent set (S) with 24 vertices:
2 8 12 15 19 23 28 32 34
43 45 49 53 58 64 67 72 76
85 91 94 99
```

Listing B.39: Output of the MISP for dataset `d25_01.dat`.

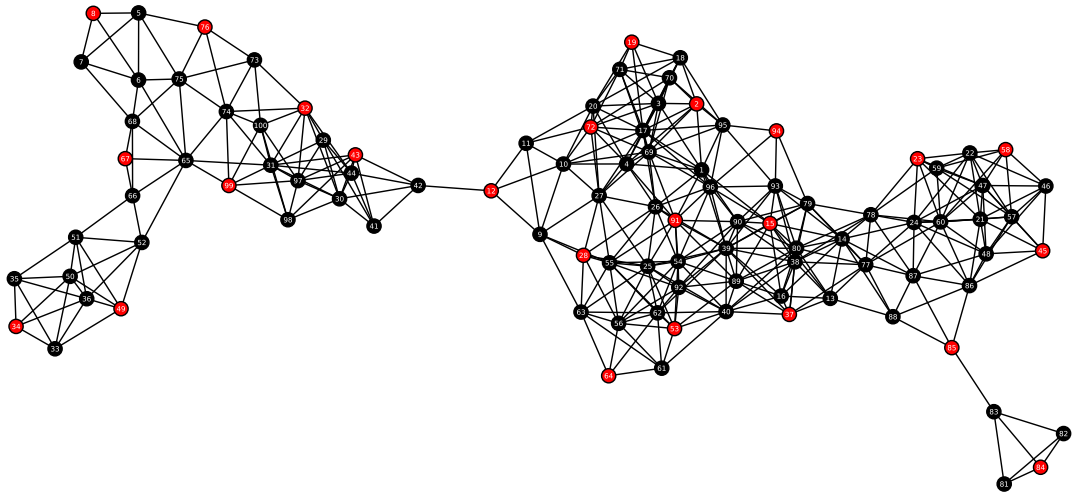


Figure B.17: Visualization of the maximum independent set for dataset `d25_01.dat`.

B.5.2 Exercise 5.2

The first step to solve this problem is to map each museum room to a number so that a graph could be constructed. Each room will be a vertex of the graph and the edges are placed for pairs of vertices that represent rooms with a door between them.

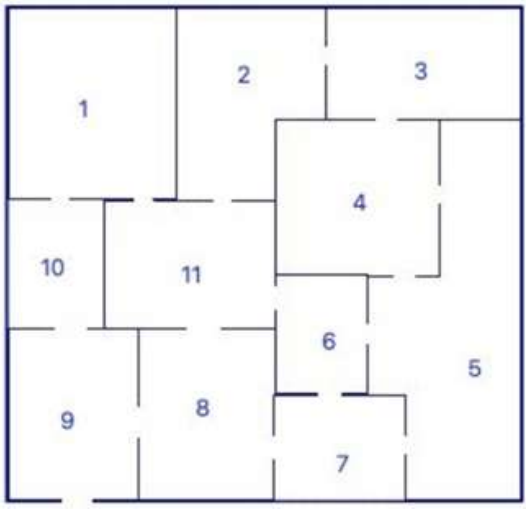


Figure B.18: Museum map with numbered rooms.

Now we represent the museum as a graph, and the problem of finding the minimum number of guards is equivalent to finding the minimum vertex cover of the graph.

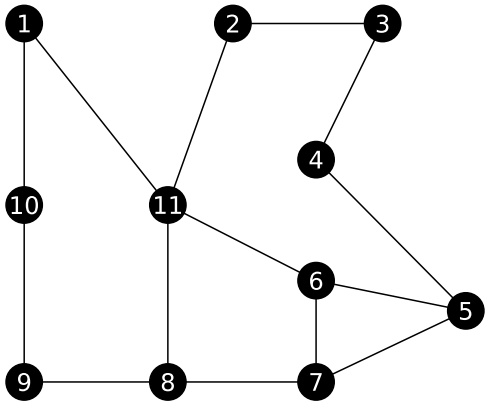


Figure B.19: Graph representation of the museum.

Listing B.40 provides a complete implementation of the Vertex Covering Problem (VCP) using the JuMP modeling language and the HiGHS solver in Julia. The code includes functions to create a graph representation of the museum, solve the VCP, and plot the solution.

```julia
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/vcp_utils.jl")

# Function to solve the Vertex Covering Problem (VCP)
function solve_vcp(graph, positions; plot_graph_solution = false)
  # Total of vertices
  n = Graphs.nv(graph)

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Define the decision variables
  @variable(model, x[1:n], Bin)

  # Objective function: minimize the total of selected vertices
  @objective(model, Min, sum(x))
```

```
24
25    # Vertex Cover constraint
26    @constraint(model, [i = 1:n, j = 1:n; Graphs.has_edge(graph, i,
      j)], x[i] + x[j] >= 1)
27
28    # Run the solver
29    JuMP.optimize!(model)
30
31    # Vertex Cover C
32    C = findall(JuMP.value.(x) .>= 0.5)
33
34    # Print solution
35    println("Vertex Cover set (C): $C")
36
37    # Plot graph solution if requested
38    if plot_graph_solution
39      fig = plot_solution(graph, positions, C)
40      display(fig)
41    end
42  end
43
44  # Create graph
45  graph = Graphs.Graph(11)
46
47  # Edges
48  edges = [(1,10), (1,11), (2,3), (2,11),
49           (3,4), (4,5), (5,6), (5,7),
50           (6,7), (6,11), (7,8), (8,9),
51           (8,11), (9,10)]
52
53  # Add edges
54  for (u, v) in edges
55    Graphs.add_edge!(graph, u, v)
56  end
57
58  # Manually define the vertices positions for the layout
59  positions = [Point(0,0), Point(138,0),
60               Point(237,0), Point(193,90),
61               Point(292,190), Point(193,170),
62               Point(193,237), Point(95,237),
63               Point(0,237), Point(0,120),
```

```
64              Point(95,120)]
65
66 # Solve the Vertex Covering Problem
67 solve_vcp(graph, positions, plot_graph_solution = true)
```

Listing B.40: Solving the VCP for the museum.

Listing B.41 shows the output of the above code when executed with the graph representation of the museum. The output includes the vertex cover set (C) that represents the rooms where guards should be placed.

Output

```
Vertex Cover set (C): [3, 5, 6, 8, 10, 11]
```

Listing B.41: Output of the VCP solution for the museum.

Figure B.20 provides a visualization of the vertex cover solution for the museum problem, showing the rooms where guards should be placed.

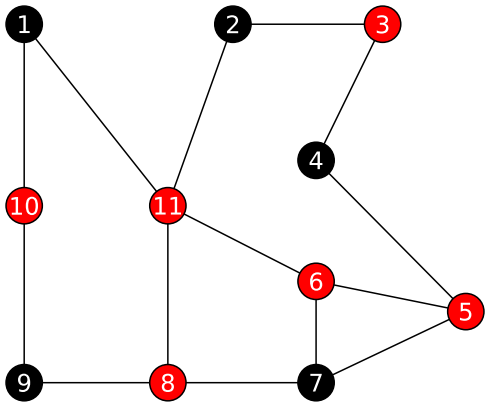


Figure B.20: Vertex cover solution for the museum problem.

B.5.3 Exercise 5.3

The first step to solve this problem is to map each district to a number so that a graph could be constructed (Figure B.21). Each district will be a vertex of the graph and the edges are placed for pairs of vertices that represent districts with a common border.



Figure B.21: Graph representation of the districts.

Now, based on the graph, we can solve the problem as a Graph Coloring Problem (GCP). Listing B.42 provides a complete implementation of the GCP using the JuMP modeling language and the HiGHS solver in Julia. The code includes functions to load the graph from a dataset, solve the GCP, and plot the solution both as a graph and on a map.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/gcp_utils.jl")

# Function to solve the Graph Coloring Problem (GCP)
function solve_gcp(graph_file, map_file, map_background_file;
plot_graph_solution = true, plot_map_solution = true)

  # Create graph
  graph = Graphs.loadgraph(graph_file)

  # Total of vertices
  n = Graphs.nv(graph)

  # Create the model
```

```
18  model = JuMP.Model(HiGHS.Optimizer)
19
20  # Silent mode (solver output is not printed)
21  JuMP.set_silent(model)
22
23  # Define the decision variables
24  @variable(model, x[1:n, 1:n], Bin)
25  @variable(model, y[1:n], Bin)
26
27  # Objective function: minimize the total of selected colors
28  @objective(model, Min, sum(y))
29
30  # Each vertex must be assigned exactly one color
31  @constraint(model, [u = 1:n], sum(x[u, :]) == 1)
32
33  # If a vertex uses color c, then color c is active
34  @constraint(model, [u = 1:n, c = 1:n], x[u, c] <= y[c])
35
36  # No two adjacent vertices share the same color
37  @constraint(model,
38              [u = 1:n, v = u+1:n, c = 1:n; Graphs.has_edge(graph,
u, v)],
39              x[u, c] + x[v, c] <= 1)
40
41  # Run the solver
42  JuMP.optimize!(model)
43
44  if JuMP.termination_status(model) != JuMP.OPTIMAL
45      error("No optimal solution found.")
46  end
47
48  # Chromatic number
49  chromatic_number = Int(round(JuMP.objective_value(model)))
50  println("Chromatic number: $chromatic_number")
51
52  # Extract the optimal color assignments
53  x_opt = JuMP.value.(x)
54  y_opt = JuMP.value.(y)
55
56  # Create solution array
57  solution = zeros(Int, n)
```

```
  selected_colors = findall(y_opt .>= 0.5)
  for (id, pos) in enumerate(selected_colors)
    ids = findall(x_opt[:, pos] .>= 0.5)
    solution[ids] .= id
    println("Group $id: $ids")
  end

  # Plot graph solution if requested
  if plot_graph_solution
    fig = plot_solution_graph(graph, map_background_file, solution)
    display(fig)
  end

  # Plotting map solution if requested
  if plot_map_solution
    fmap = plot_solution_map(map_file, solution)
    display(fmap)
  end
end

# Example usage
solve_gcp("data/districts.csv", "data/districts.geojson", "images/
districts.png")
```

Listing B.42: Julia code to solve the GCP for the districts.

Listing B.43 shows the output of the above code when executed with the graph representation of the districts. The output includes the chromatic number and the groups of districts that can be colored with the same color.

Output

```
Chromatic number: 4
Group 1: [2, 4, 9, 18, 21, 27, 28, 29, 31]
Group 2: [6, 7, 10, 16, 19, 26, 30]
Group 3: [3, 8, 12, 14, 17, 20, 22, 24, 25]
Group 4: [1, 5, 11, 13, 15, 23, 32, 33, 34]
```

Listing B.43: Output of the Graph Coloring solution.

Figure B.22 provides a visualization of the GCP solution for the districts problem represented as a graph, showing the different groups of districts colored with different colors.

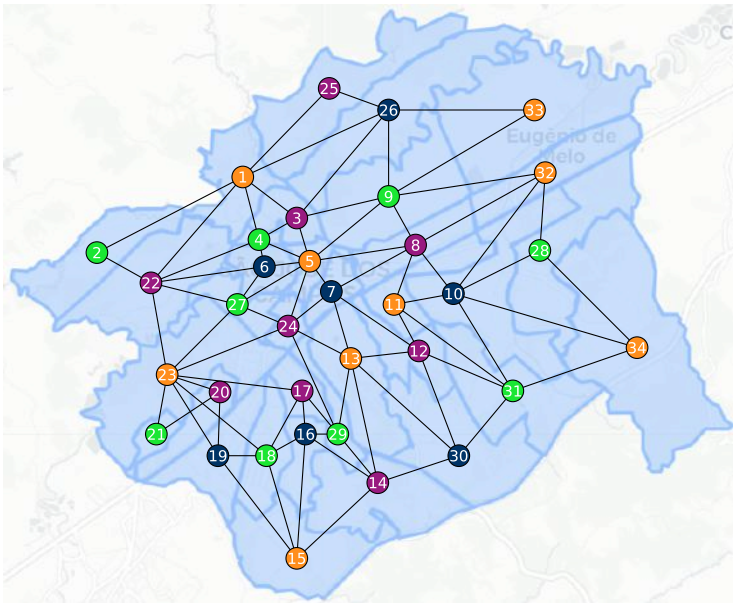


Figure B.23: GCP solutiong for the districts problem represented as a graph.

Figure B.23 provides a visualization of the GCP solution for the districts problem represented on a map, showing the different groups of districts colored with different colors.

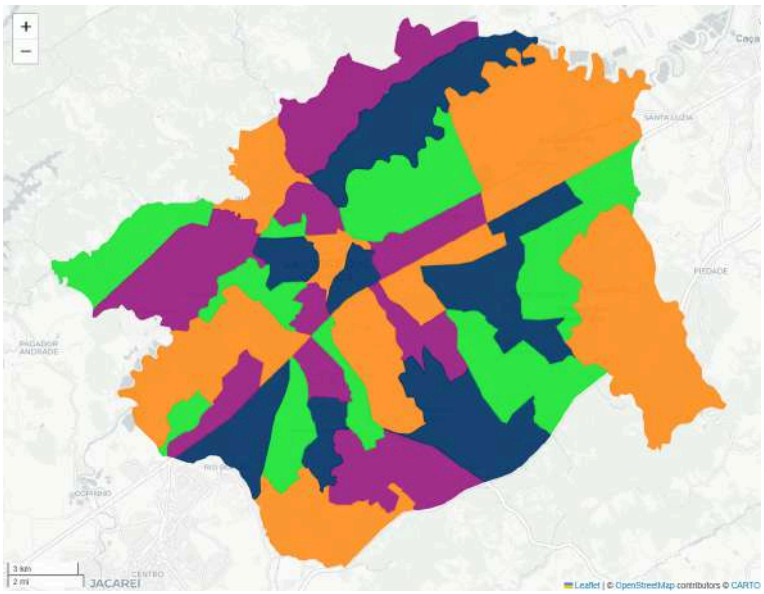


Figure B.23: GCP solutiong for the districts problem represented on the map.

B.5.4 Exercise 5.4

The first step to solve this problem as the Graph Partitioning Problem (GPP) is representing the graph as shown in Figure B.24.

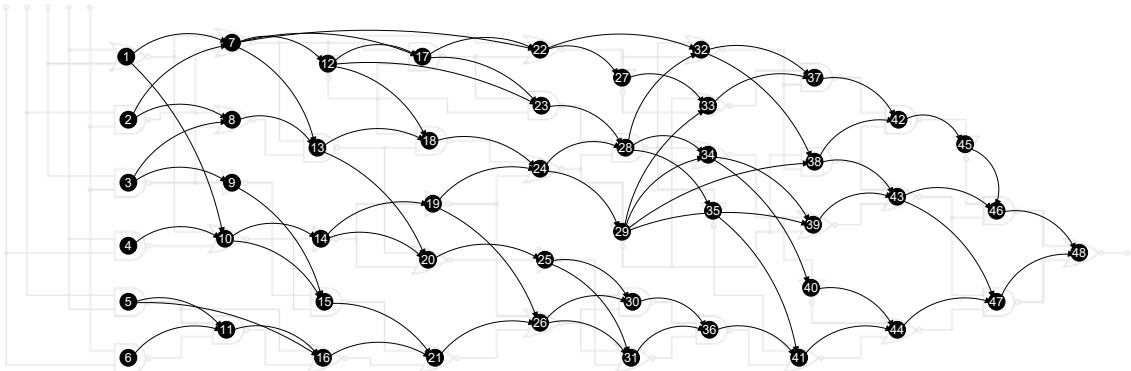


Figure B.24: Graph representation of the circuit represented as a graph.

The following code solves the GPP for the circuit dataset. Listing B.44 provides a complete implementation of the GPP using the JuMP modeling language and the HiGHS solver in Julia. The code includes functions to load the graph from a dataset, solve the GPP and plot the solution.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/gpp_utils.jl")

# Function to solve the Graph Partitioning Problem (GPP)
function solve_gpp(graph_file; k = 2, circuit_background = nothing,
plot_graph_solution = true)

  # Load graph from file
  graph = Graphs.loadgraph(graph_file)

  # Total of vertices
  n = Graphs.nv(graph)

  # Total of edges
  m = Graphs.ne(graph)

```

```
  # Define lower and upper size limits for each partition
  L = floor(Int, n / k)
  U = ceil(Int, n / k)

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Define the decision variables
  # x[u,i] = 1 if vertex u is in partition i
  @variable(model, x[1:n, 1:k], Bin)
  # y[e] = 1 if edge e is a cut edge
  @variable(model, y[1:m], Bin)

  # Objective: minimize number of cut edges
  @objective(model, Min, sum(y))

  # Each vertex must be assigned to exactly one partition
  @constraint(model, [u in 1:n], sum(x[u, p] for p in 1:k) == 1)

  # Partition balance constraint
  @constraint(model, [p in 1:k], sum(x[u, p] for u in 1:n) >= L)
  @constraint(model, [p in 1:k], sum(x[u, p] for u in 1:n) <= U)

  # Define y[e] = 1 if edge connects different partitions
  for (e_id, e) in enumerate(Graphs.edges(graph))
    u, v = Graphs.src(e), Graphs.dst(e)
    for p in 1:k
      @constraint(model, y[e_id] >= x[u, p] - x[v, p])
      @constraint(model, y[e_id] >= x[v, p] - x[u, p])
    end
  end

  # Run the solver
  JuMP.optimize!(model)

  # Print solution
  println("Edge cuts total: ", round(Int,
JuMP.objective_value(model)))
```

```
60
61    # Get partitions
62    partitions = [findall(JuMP.value.(x[:,id]) .> 0.5) for id in 1:k]
63
64    # Get cut edges
65    cut_edges = []
66    for (e_id, e) in enumerate(Graphs.edges(graph))
67      if JuMP.value(y[e_id]) > 0.5
68        push!(cut_edges, (Graphs.src(e), Graphs.dst(e)))
69      end
70    end
71
72    # Print results
73    println("\nEdge cuts:")
74    for (id, cut_edge) in enumerate(cut_edges)
75      print("$cut_edge ")
76      if id % 8 == 0
77        println()
78      end
79    end
80
81    println("\n\nPartitions:")
82    for (id, partition) in enumerate(partitions)
83      println("Partition $id ($(length(partition)) components): ",
partition)
84    end
85
86    # Plot graph solution if requested
87    if plot_graph_solution
88      fig = plot_solution(graph, partitions, y, k; circuit_background
= circuit_background)
89      display(fig)
90    end
91  end
92
93  # Solve the Graph Partitioning Problem
94  solve_gpp("data/circuit.csv", k = 3, circuit_background = "images/
circuit_background.png")
```

Listing B.44: Julia code to solve the GPP for the circuit dataset.

Listing B.45 shows the output of the above code when executed with the graph representation of the circuit dataset. The output includes the total number of edge cuts, the list of cut edges, and the components in each partition.

Output

```
Edge cuts total: 11

Edge cuts:
(1, 10) (9, 15) (13, 20) (18, 24) (19, 24) (23, 28) (28, 32) (29, 33)
(32, 38) (36, 41) (37, 42)

Partitions:
Partition 1 (16 components): [1, 2, 3, 7, 8, 9, 12, 13, 17, 18, 22, 23,
27, 32, 33, 37]
Partition 2 (16 components): [4, 5, 6, 10, 11, 14, 15, 16, 19, 20, 21,
25, 26, 30, 31, 36]
Partition 3 (16 components): [24, 28, 29, 34, 35, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48]
```

Listing B.45: Output of the GPP for the circuit dataset.

Figure B.25 provides a visualization of the GPP solution for the circuit dataset represented as a graph, showing the different partitions colored with different colors and the cut edges highlighted.

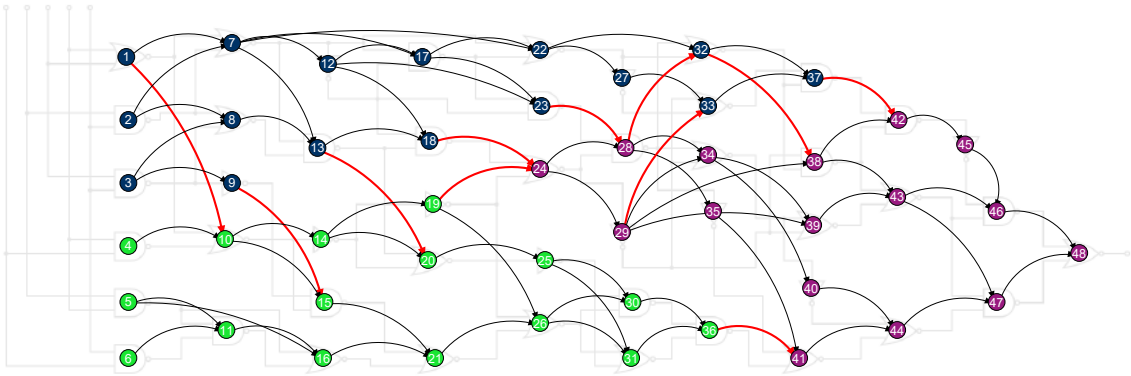


Figure B.25: GPP solution for the circuit dataset represented on graph.

Figure B.26 provides a visualization of the GPP solution, showing circuit with partitions colored with different colors and the cut edges highlighted.

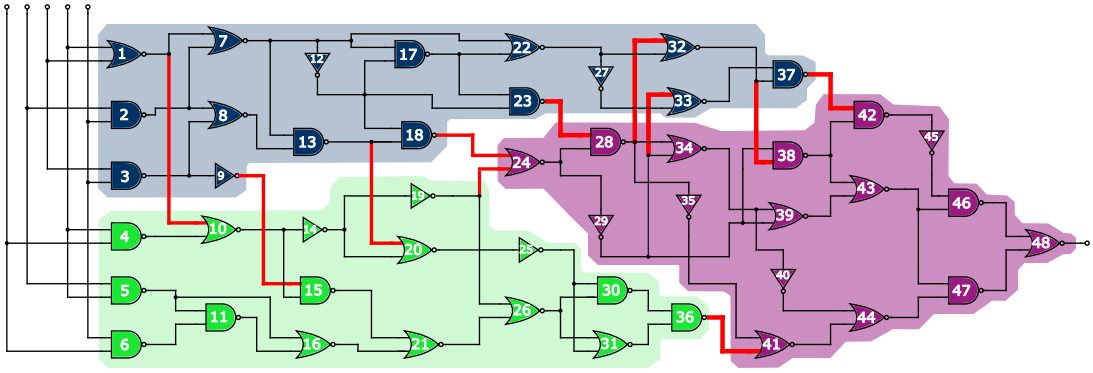


Figure B.26: Obtained circuit partitions.

B.5.5 Exercise 5.5

Listing B.46 provides a complete implementation of the Clique Partitioning Problem (CPP) using the JuMP modeling language and the HiGHS solver in Julia. The code includes functions to load the graph from a dataset, solve the CPP, and plot the solution.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Load utility function for plotting
include("utils/cpp_utils.jl")

# Function to solve the Clique Partitioning Problem (CPP)
function solve_cpp(graph_file)

  # Load the graph from the file
  graph = Graphs.loadgraph(graph_file)

  # Get number of vertices and edges
  n = Graphs.nv(graph)
  m = Graphs.ne(graph)

  # Auxiliary variables to simplify notation
  V = 1:n

  # Modularity matrix Q
  Q = zeros(n,n)
```

```
  for u in V
    for v in V
      du = Graphs.degree(graph, u)
      dv = Graphs.degree(graph, v)
      a_ij = Graphs.has_edge(graph, u, v) ? 1 : 0
      Q[u,v] = a_ij/(2*m) - (du * dv) / (4*m^2)
    end
  end

  # Create the model
  model = JuMP.Model(HiGHS.Optimizer)

  # Silent mode (solver output is not printed)
  JuMP.set_silent(model)

  # Define the decision variables
  @variable(model, x[u in V, v in V], Bin)

  # Objective: maximize the weight of the selected edges
  @objective(model, Max, sum(Q[u,v] * x[u,v] for u in V for v in V))

  # Transitive Constraints: ensure that the selected edges form a
valid partition
  for u in V
    for v in V
      for w in V
        @constraint(model,  x[u,v] + x[v,w] - x[u,w] <= 1)
        @constraint(model,  x[u,v] - x[v,w] + x[u,w] <= 1)
        @constraint(model, -x[u,v] + x[v,w] + x[u,w] <= 1)
      end
    end
  end

  # Run the solver
  JuMP.optimize!(model)

  # Get the objective (modularity)
  modularity = JuMP.objective_value(model)
  println("Modularity: ", modularity)

  # Extract the optimal partition
```

```
  println("Partition:")
  partition = zeros(Int64, n)
  partition_id = 1
  for u in V
    # If vertex u is not yet assigned to a partition
    if partition[u] == 0
      print("Cluster $partition_id: ")
      for v in V
        # If vertex v is in the same partition as u
        if JuMP.value(x[u,v]) > 0.5
          # Assign both u and v to the same partition
          partition[v] = partition_id
          print("$v ")
        end
      end
      print("\n")
      partition_id += 1
    end
  end
  # Plot solution
  plot_solution(graph, partition)
end

# Example usage
solve_cpp("data/karate.csv")
```

Listing B.46: Julia code to solve the CPP for the karate club dataset.

Listing B.47 shows the output of the above code when executed with the karate club dataset. The output includes the modularity of the partition and the clusters of vertices in each partition.

Output

```
Modularity: 0.41978961209730403
Partition:
Cluster 1: 1 2 3 4 8 12 13 14 18 20 22
Cluster 2: 5 6 7 11 17
Cluster 3: 9 10 15 16 19 21 23 27 30 31 33 34
Cluster 4: 24 25 26 28 29 32
```

Listing B.47:

Figure B.27 provides a visualization of the CPP solution for the karate club dataset, showing the different clusters colored with different colors.

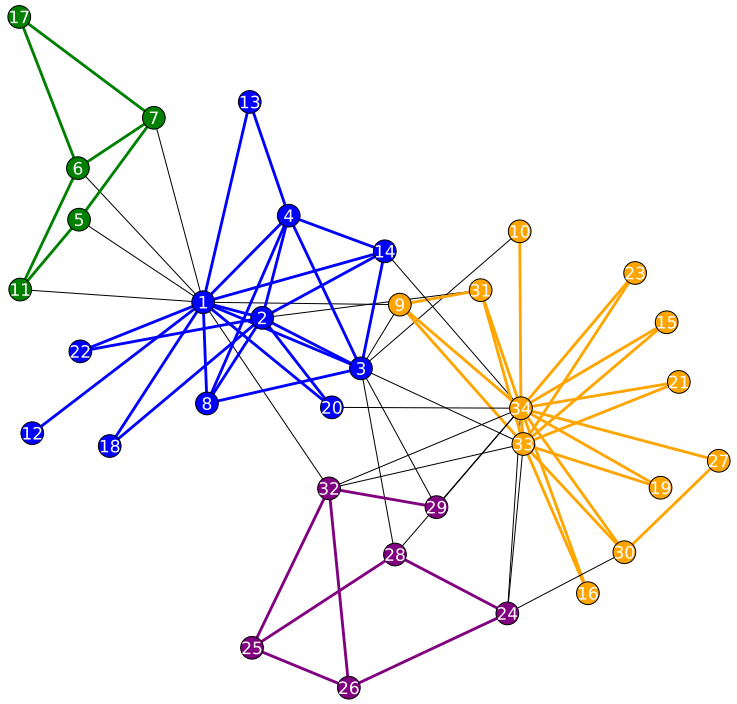


Figure B.27: Visualization of the CPP solution for the karate club dataset.

B.6 Chapter 6

B.6.1 Exercise 6.1

To solve the Capacitated P-Median Problem (CPMP) using Column Generation (CG), we will implement the following steps in Julia:

1. Define a column: a column represents a potential solution to the subproblem, which in this case is a specific assignment of clients to a facility (median). Each column will have an associated cost and a binary vector indicating which clients are assigned to that facility.
2. Create initial columns: we will generate a set of initial columns randomly, ensuring that they respect the capacity constraints of the facilities.
3. Solve the Restricted Master Problem (RMP): this involves solving a linear programming relaxation of the master problem using the current set of columns. The RMP will provide us with the following dual prices:

- π: dual prices associated with the constraints that ensure each client is served by at least one facility.
- σ: dual price associated with the constraint that ensures exactly p facilities are chosen.

4. Solve the Pricing Subproblem: using the dual prices obtained from the RMP, we will solve a subproblem to find new columns with negative reduced cost. This involves solving a binary knapsack problem for each facility to determine if there is a more cost-effective way to assign clients to that facility where the objective is to minimize the reduced cost:

$$\text{rc} = \sum_{c=1}^{n} (d_{c,f} - \pi_c) * x_c - \sigma \tag{B.12}$$

5. Update the master problem: if a new column with negative reduced cost is found, we will add it to the master problem and repeat the process from step 3. If no new column with negative reduced cost is found, we have reached the optimal solution for the linear programming relaxation of the master problem.

B.6.1.1 a) Solving CPMP using Column Generation

Listing B.48 presents the Julia code that implements the CG approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The code includes functions to create initial columns, solve the RMP, solve the pricing subproblem, and perform the column generation loop. The final output includes the optimal objective value of the RMP and a plot showing the convergence of the reduced cost over iterations.

```
using JuMP       # Modeling language
using HiGHS      # Solver
using CSV        # For reading CSV files
using DataFrames # For handling data frames
using Distances  # For distance calculations
using Random     # For random number generation
using Printf     # For formatted printing

# Utility functions for plotting convergence
include("utils/cg_pmp-n_utils.jl")

# Structure to hold problem data
struct CPMPData
```

```
    n::Int64                        # Number of points
    p::Int64                        # Number of facilities to open
    C::Int64                        # Capacity of each facility
    coordinates::Matrix{Float64} # Coordinates of points
    distance::Matrix{Float64}    # Distance matrix between points
    demand::Vector{Int64}        # Demand at each point
    # Constructor
    function CPMPData(file_path::String; p::Int64=7, C::Int64=25)
        df = CSV.read(file_path, DataFrame)
        coordinates = Matrix{Float64}(df[:, [1, 2]])
        distance_matrix = pairwise(Haversine(), coordinates,
dims=1)
        demand = Vector{Int64}(df[:, 3])
        new(nrow(df), p, C, coordinates, distance_matrix, demand)
    end
end

# Structure to hold column information
struct CPMPColumn
    cost::Float64
    median::Int64
    assignments::Vector{Bool}
end

# Structure to hold the solution of the RMP
struct CPMPSolution
    y::Vector{Float64}
    objective::Float64
    duals::Dict{String, Any}
end

# Optimized function to create initial columns
function cpmp_create_initial_columns(data::CPMPData, total::Int64)
    columns = Vector{CPMPColumn}(undef, total)
    for id in 1:total
        assignments = falses(data.n)
        current_capacity = 0
        while true
            for i in shuffle(1:data.n)
                if rand() > 0.5 && (current_capacity +
data.demand[i] <= data.C)
```

```
                    assignments[i] = true
                    current_capacity += data.demand[i]
                end
            end
            if sum(assignments) > 0
                break
            end
        end
        # Find the median (facility) with minimum total assignment cost
        min_cost = Inf
        median = 1
        for i in 1:data.n
            if assignments[i] == 1
                cost = sum(assignments[j] * data.distance[i, j] for j in 1:data.n)
                if cost < min_cost
                    min_cost = cost
                    median = i
                end
            end
        end
        columns[id] = CPMPColumn(min_cost, median, assignments)
    end
    return columns
end

# Function to solve the Restricted Master Problem (RMP)
function cpmp_solve_restricted_master_problem(data::CPMPData, columns::Vector{CPMPColumn})
    # Number of clients, facilities and columns
    nC = nF = data.n
    ncols = length(columns)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, 0 <= y[1:ncols] <= 1)
    # Objective: Minimize the total cost of the selected columns
```

```
90      @objective(model, Min, sum(y[i] * columns[i].cost for i in
    1:ncols))
91      # Constraint: Each client 'j' must be served by at least one
    selected column
92      @constraint(model, covering_satisfaction[j=1:nC],
    sum(columns[i].assignments[j] * y[i] for i in 1:ncols) >= 1)
93      # Constraint: Ensures that exactly 'p' columns are chosen
94      @constraint(model, p_median_satisfaction, sum(y) == data.p)
95      # Solve the model
96      JuMP.optimize!(model)
97      # Check the results and return dual prices
98      if JuMP.termination_status(model) == JuMP.OPTIMAL
99          y_values = JuMP.value.(y)
100         objective = JuMP.objective_value(model)
101         π = JuMP.dual.(covering_satisfaction)
102         σ = JuMP.dual(p_median_satisfaction)
103         duals = Dict("π" => π, "σ" => σ)
104         return CPMPSolution(y_values, objective, duals)
105     else
106         error("RMP was not solved to optimality.")
107     end
108 end
109
110 # Function to solve the Pricing Subproblem for each facility and
    generate new columns
111 function cpmp_solve_pricing_subproblem(data::CPMPData,
    columns::Vector{CPMPColumn}, rmp_solution::CPMPSolution;
    tolerance::Float64 = 1e-6)
112     # Extract dual prices from the RMP solution
113     π = rmp_solution.duals["π"]
114     σ = rmp_solution.duals["σ"]
115     # Number of clients and facilities
116     nC = data.n
117     nF = data.n
118     # Initialize variables to track the best column found
119     min_reduced_cost = typemax(Float64)
120     # Now we solve 1-knapsack problem for each facility
121     for f in 1:nF
122         # Create model
123         model = JuMP.Model(HiGHS.Optimizer)
124         # Silent mode (solver output is not printed)
125         JuMP.set_silent(model)
```

```
126         # Unidimensional variable: does client 'c' go to facility
    'f'?
127         @variable(model, x[1:nC], Bin)
128         # Objective: Minimize the reduced cost of the column
129         @objective(model, Min, sum((data.distance[c, f] - π[c]) *
    x[c] for c in 1:nC) - σ)
130         # Capacity constraint for facility 'f'
131         @constraint(model, sum(data.demand[c] * x[c] for c in 1:nC)
    <= data.C)
132         # Solve the model
133         JuMP.optimize!(model)
134         # Check the results and calculate the reduced cost
135         if JuMP.termination_status(model) == JuMP.OPTIMAL
136             reduced_cost = JuMP.objective_value(model)
137             # If the reduced cost is strictly negative, the column
    is advantageous
138             if reduced_cost < -tolerance
139                 # Extract the clients assigned to this facility
140                 assignments = JuMP.value.(x) .> 0.5
141
142                 # Check if the assignments are valid (not all
    zeros)
143                 is_invalid = sum(assignments) == 0
144
145                 # Check if is duplicated
146                 is_duplicate = any(c -> c.assignments ==
    assignments, columns)
147
148                 # Prevent creating columns with no assignments and
    avoid duplicates
149                 if !is_invalid && !is_duplicate
150                     min_reduced_cost = min(min_reduced_cost,
    reduced_cost)
151                     # Calculate the actual cost of the column (sum
    of distances of active clients)
152                     cost = sum(data.distance[c, f] for c in 1:nC if
    assignments[c])
153                     # Add the new column to the list of columns
154                     push!(columns, CPMPColumn(cost, f,
    assignments))
155                 end
156             end
```

```
157         end
158     end
159     return min_reduced_cost
160 end
161 
162 # Function to solve the CPMP using column generation
163 function cpmp_solve_gc(data::CPMPData; initial_columns_total::Int64 = 20, iterations::Int64 = 100, tolerance::Float64 = 1e-6, print_feedback_interval::Int64 = 10)
164 
165     println("--- Starting Column Generation ---\n")
166 
167     println("| Iteration | Reduced Cost | RMP Objective | Columns |")
168     println("|-----------|--------------|---------------|---------|")
169 
170     # DataFrame to store results of each iteration
171     results = DataFrame(Iteration = Int[], ReducedCost = Float64[], RMPObjective = Float64[], Columns = Int[])
172 
173     # Vector to store reduced costs for each iteration
174     reduced_cost_history = []
175 
176     # --- 1. Create initial columns ---
177     columns = cpmp_create_initial_columns(data, initial_columns_total)
178 
179     best_solution = nothing
180 
181     # --- 2. Column generation loop ---
182     for iteration in 1:iterations
183 
184         # --- 3. Solve the Restricted Master Problem (RMP) ---
185         rmp_solution = cpmp_solve_restricted_master_problem(data, columns)
186 
187         # --- 4. Solve the Pricing Subproblem ---
188         reduced_cost = cpmp_solve_pricing_subproblem(data, columns, rmp_solution, tolerance = tolerance)
189 
190         # --- 5. Check if a new pattern was found ---
```

```
191         if reduced_cost < -tolerance
192             if iteration == 1 || iteration %
print_feedback_interval == 0
193                 @printf("| %9d | %12.2f | %13.3f | %7d |\n",
194                         iteration,
195                         reduced_cost,
196                         rmp_solution.objective,
197                         length(columns))
198             end
199             push!(reduced_cost_history, reduced_cost)
200             best_solution = deepcopy(rmp_solution)
201             # Add results to DataFrame
202             push!(results, (iteration, reduced_cost,
rmp_solution.objective, length(columns)))
203         else
204             println("\n--- Column Generation Complete ---\n")
205             println("No more profitable columns found. Optimal LP
solution reached.")
206             break
207         end
208     end
209     # --- 6. Generate reduced cost Plot ---
210     cpmp_plot_reduced_costs(reduced_cost_history)
211     return best_solution, columns, results
212 end
213
214 # Set random seed for reproducibility
215 Random.seed!(42)
216
217 # Read data and compute distance matrix
218 data = CPMPData("data/sjc_n=30_p=7_C=25.csv"; p = 7, C = 25)
219
220 # Example usage
221 relaxed_solution, columns, results = cpmp_solve_gc(data,
initial_columns_total = 1500, iterations = 1500, tolerance = 1e-6,
print_feedback_interval = 1);
222
223 println("\nFinal RMP Objective: ", relaxed_solution.objective)
```

Listing B.48: Julia code that implements the CG approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`.

Listing B.49 presents the output of the CG approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The output includes the iteration number, reduced cost, RMP objective value, and the evolution of the number of columns at each iteration. The process continues until no more profitable columns can be found, indicating that the optimal LP solution has been reached.

Output

```
--- Starting Column Generation ---

| Iteration | Reduced Cost | RMP Objective | Columns |
|-----------|--------------|---------------|---------|
|         1 |     -1195.58 |      5452.409 |    1521 |
|         2 |      -669.71 |      4492.844 |    1545 |
|         3 |      -595.48 |      3751.066 |    1564 |
|         4 |      -351.03 |      3629.761 |    1586 |
|         5 |      -224.31 |      3323.915 |    1602 |
|         6 |      -163.99 |      3127.170 |    1617 |
|         7 |      -174.30 |      3022.213 |    1628 |
|         8 |      -179.16 |      2907.673 |    1646 |
|         9 |       -76.07 |      2881.754 |    1650 |
|        10 |       -64.43 |      2881.754 |    1658 |
|        11 |       -15.88 |      2877.364 |    1663 |
|        12 |       -78.09 |      2877.364 |    1672 |
|        13 |       -47.12 |      2877.364 |    1673 |
|        14 |       -20.52 |      2877.364 |    1674 |

--- Column Generation Complete ---

No more profitable columns found. Optimal LP solution reached.

Final RMP Objective: 2877.363898621487
```

Listing B.49: Output of the CG approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`.

Figure B.28 presents a plot showing the convergence of the reduced cost over iterations for the CG approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The plot illustrates how the reduced cost evolves as new columns are generated and added to the master problem, eventually converging to zero when no more profitable columns can be found.

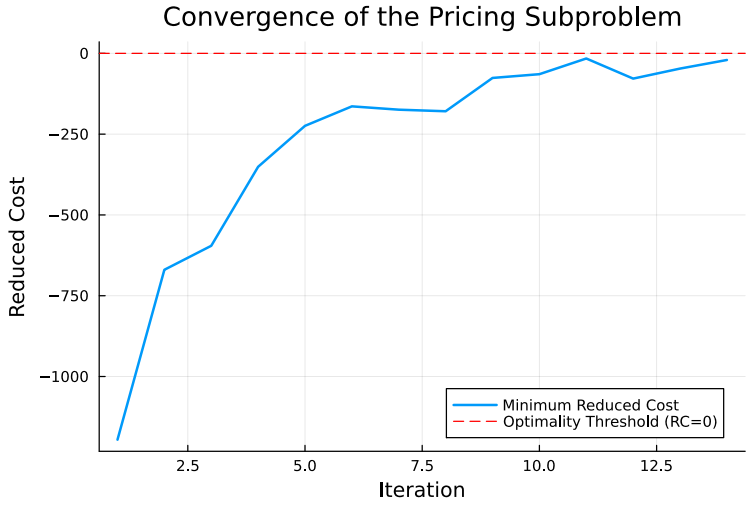


Figure B.28: Reduced cost convergence for the dataset `sjc_n=30_p=7_C=25.csv`.

B.6.1.2 b) Solving RMIP using columns generated by CG

Listing B.50 presents the Julia code that implements the RMIP to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv` using the columns generated in part (a). The code includes a function to solve the RMIP by imposing integrality constraints on the master problem variables. The output includes the optimal objective value of the RMIP, the selected columns (facilities opened), and the clients served by each selected facility. Additionally, a plot is generated to visualize the solution.

```
# Function to solve the Restricted Master Integer Problem (RMIP)
function cpmp_solve_restricted_master_integer_problem(data::CPMPData, columns::Vector{CPMPColumn})
    # Number of clients, facilities and columns
    nC = nF = data.n
    ncols = length(columns)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, y[1:ncols], Bin)
    # Objective: Minimize the total cost of the selected columns
    @objective(model, Min, sum(y[i] * columns[i].cost for i in 1:ncols))
    # Constraint: Each client 'j' must be served by at least one selected column
```

```
    @constraint(model, covering_satisfaction[j=1:nC], sum(columns[i].assignments[j] * y[i] for i in 1:ncols) >= 1)
    # Constraint: Ensures that exactly 'p' columns are chosen
    @constraint(model, p_median_satisfaction, sum(y) == data.p)
    # Solve the model
    JuMP.optimize!(model)
    # Check the results and return dual prices
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        # Extract the objective value
        objective = JuMP.objective_value(model)
        println("RMIP solution found with objective value: ", objective)
        solution = Dict()
        # Extract the selected columns
        selected_columns = columns[JuMP.value.(y) .> 0.5]
        # Order selected columns by their median index for better readability
        selected_columns = sort!(selected_columns, by = col -> col.median)
        println("Selected columns (facilities opened): ", [col.median for col in selected_columns])
        # For each selected column, print the assigned clients
        for col in selected_columns
            assigned_clients = [j for j in 1:nC if col.assignments[j]]
            println("Facility $(col.median) serves clients: ", assigned_clients)
            solution[col.median] = assigned_clients
        end
        # Plot the solution
        cpmp_plot_solution(data.coordinates, solution, data.p)
    else
        error("RMP was not solved to optimality.")
    end
end

# Example usage of the integer version of the RMP
cpmp_solve_restricted_master_integer_problem(data, columns)
```

Listing B.50: Julia code that implements RMIP to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`.

Listing B.51 presents the output of the RMIP approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The output includes the optimal objective value of the RMIP, the selected columns (facilities opened), and the clients served by each selected facility.

Output

```
RMIP solution found with objective value: 2947.040874259389
Selected columns (facilities opened): [2, 5, 6, 12, 18, 25, 26]
Facility 2 serves clients: [2, 13, 17, 24, 28, 29]
Facility 5 serves clients: [5, 15, 23]
Facility 6 serves clients: [1, 6, 14, 20]
Facility 12 serves clients: [8, 12, 16]
Facility 18 serves clients: [4, 18, 19, 22, 27]
Facility 25 serves clients: [3, 7, 9, 11, 25]
Facility 26 serves clients: [10, 21, 26, 30]
```

Listing B.51: Output of the RMIP approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`.

Figure B.29 presents a plot visualizing the solution obtained from the RMIP approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The plot shows the locations of the clients and the facilities opened, with lines connecting each facility to the clients it serves. This visualization helps to understand the spatial distribution of the solution and how clients are assigned to facilities.

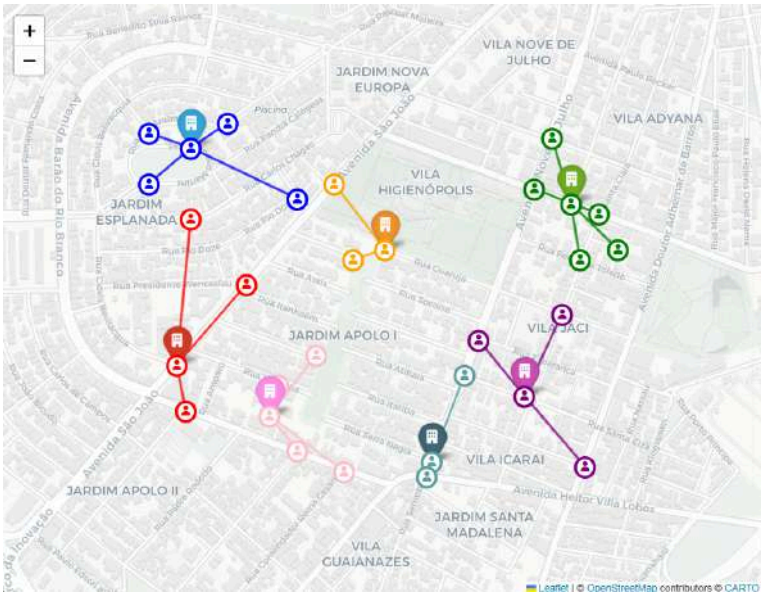


Figure B.29: RMIP solution for the dataset `sjc_n=30_p=7_C=25.csv`.

B.6.1.3 c) Solving CPMP using ILP

Listing B.52 presents the Julia code that implements an Integer Linear Programming (ILP) approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The code includes a function to solve the ILP by defining binary variables for client assignments and facility openings, along with the necessary constraints. The output includes the optimal objective value of the ILP, the selected columns (facilities opened), and the clients served by each selected facility. Additionally, a plot is generated to visualize the solution.

```
# Function to solve the CPMP using Integer Linear Programming (ILP)
function cpmp_solve_ilp(data::CPMPData)
    nC = nF = data.n
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[c in 1:nC,f in 1:nF], Bin)
    # Objective: Minimize the total cost of the selected columns
    @objective(model, Min, sum(x[c, f] * data.distance[c, f] for c
in 1:nC, f in 1:nF))
    # Exactly 'p' facilities must be opened
    @constraint(model, sum(x[f, f] for f in 1:nF) == data.p)
    # Each client must be assigned to exactly one facility
    @constraint(model, [c in 1:nC], sum(x[c, f] for f in 1:nF) == 1)
    # A client can only be assigned to an open facility (if x[c, f]
= 1, then x[f, f] must be 1)
    @constraint(model, [c in 1:nC, f in 1:nF], x[c, f] <= x[f, f])
    # Each facility has a capacity constraint
    @constraint(model, [f in 1:nF], sum(data.demand[c] * x[c, f] for
c in 1:nC) <= data.C * x[f, f])
    # Solve the model
    JuMP.optimize!(model)
    # Check the results
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        objective = JuMP.objective_value(model)
        println("Optimal solution found with objective value: ",
objective)
        x_values = JuMP.value.(x)
        solution = Dict()
        medians = [f for f in 1:nF if x_values[f, f] > 0.5]
```

```
        println("Optimal medians (facilities opened): ", sort!(medians))
        for median in medians
            assigned_clients = [c for c in 1:nC if x_values[c, median] > 0.5]
            println("Facility $median serves clients: ", assigned_clients)
            solution[median] = assigned_clients
        end
        # Plot the solution
        cpmp_plot_solution(data.coordinates, solution, data.p)
    else
        error("No optimal solution found")
    end
end

# Example usage of the ILP approach
cpmp_solve_ilp(data)
```

Listing B.52: Julia code that implements ILP to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`.

Listing B.53 presents the output of the ILP approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The output includes the optimal objective value of the ILP, the selected columns (facilities opened), and the clients served by each selected facility.

Output

```
Optimal solution found with objective value: 2898.4216409686173
Optimal medians (facilities opened): [2, 6, 12, 15, 25, 26, 27]
Facility 2 serves clients: [2, 13, 17, 24, 28, 29]
Facility 6 serves clients: [1, 6, 14, 20]
Facility 12 serves clients: [8, 12, 16]
Facility 15 serves clients: [5, 15, 23]
Facility 25 serves clients: [3, 7, 9, 11, 25]
Facility 26 serves clients: [10, 21, 26, 30]
Facility 27 serves clients: [4, 18, 19, 22, 27]
```

Listing B.53: Output of the ILP approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`.

Figure B.30 presents a plot visualizing the solution obtained from the ILP approach to solve the CPMP for the dataset `sjc_n=30_p=7_C=25.csv`. The plot shows the locations of the clients and the facilities opened, with lines connecting each facility to the clients it serves. This visualization helps to understand the spatial distribution of the solution and how clients are assigned to facilities.

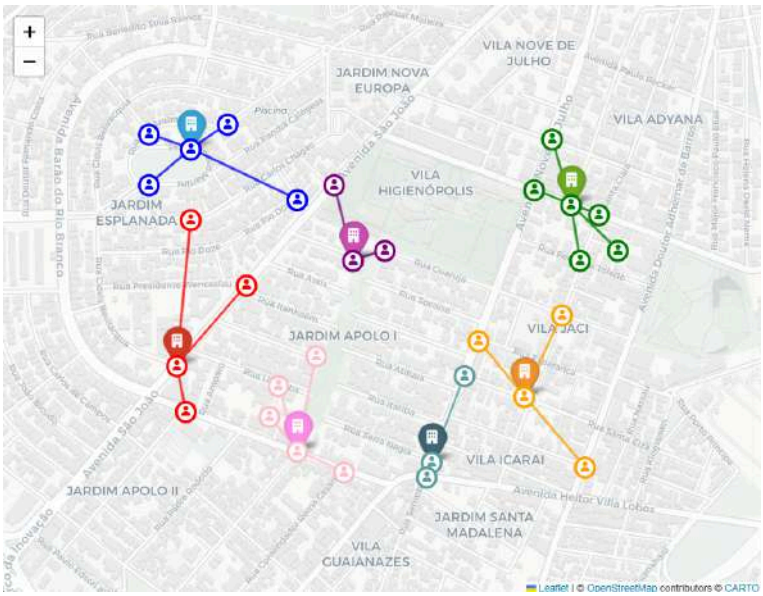


Figure B.30: ILP solution for the dataset `sjc_n=30_p=7_C=25.csv`.

Discussion

Table B.2 presents the objective values obtained by the Restricted Master Problem (RMP), the Restricted Master Integer Problem (RMIP), and the Integer Linear Programming (ILP) model, together with the corresponding integrality gap.

Technique	Objective	Gap (%)
CG	2835.50	2.17
RMIP	2898.42	0.00
ILP	2898.42	N/A

Table B.2: Comparison of CPMP solutions with different models.

The Column Generation approach produced a final objective value of 2835.50, providing a lower bound with an integrality gap of 2.17% relative to the optimal integer solution. The columns generated during the column generation process were sufficient for the RMIP to recover the global optimum, yielding the same

objective value as the original ILP model (2898.42) without requiring additional branching procedures.

B.6.2 Exercise 6.2

To solve the Clique Partitioning Problem (CPP) using Column Generation (CG), we implement the following steps in Julia:

1. Define a column: a column represents a potential community (subset of vertices), whose cost is given by the modularity contribution of that community.
2. Create initial columns: generate an initial set of columns randomly, together with singleton columns containing only one vertex. The singleton columns guarantee that the initial Restricted Master Problem is feasible, since every vertex can always be assigned to its own community.
3. Solve the Restricted Master Problem (RMP): solve the linear programming relaxation of the master problem using the current set of columns. The RMP provides the dual prices associated with the partitioning constraints that ensure each vertex is assigned to exactly one community.
4. Solve the pricing subproblem: using the dual prices obtained from the RMP, solve a pricing subproblem to identify a new community with positive reduced cost. The pricing problem searches for a subset of vertices whose addition to the RMP can improve the objective value. Since the modularity contribution of a community contains quadratic terms, binary variables and a linearization scheme are used to formulate the pricing problem as a mixed-integer linear program.
5. Generate and add a new column: if the pricing subproblem finds a community with positive reduced cost, the corresponding column is generated, its modularity contribution is computed, and the column is added to the RMP.
6. Apply dual stabilization: to reduce oscillations in the dual prices and improve convergence, a smoothing procedure is applied by combining the current dual solution with the dual solution from the previous iteration.
7. Repeat the column generation process: re-solve the RMP with the expanded set of columns, obtain updated dual prices, and solve the pricing subproblem

again. This iterative process continues until no column with positive reduced cost can be found.

B.6.2.1 a) Solving CPP using Column Generation

Listing B.54 presents the Julia code that implements the CG approach to solve the CPP for the dataset `karate.csv`. The code includes functions to create initial columns, solve the RMP, solve the pricing subproblem, and perform the column generation loop. The final output includes the optimal objective value of the RMP and a plot showing the convergence of the reduced cost over iterations.

```
using JuMP       # Modeling language
using HiGHS      # Solver
using DataFrames # For handling data frames
using Graphs     # For graph data structures
using Random     # For random number generation
using Printf     # For formatted printing

# Utility functions for plotting convergence
include("utils/cg_cpp_utils.jl")

# Structure to hold problem data
struct CPPData
    graph::Graphs.SimpleGraph
    Q::Matrix{Float64}
    # Constructor
    function CPPData(file_path::String)
        graph = Graphs.loadgraph(file_path)
        n, m = nv(graph), ne(graph)
        # Create modularity matrix Q
        C1 = 1.0 / (2 * m)
        C2 = 1.0 / (4 * m^2)
        Q = zeros(Float64,n,n)
        d = Graphs.degree(graph)
        for u in 1:n
            for v in 1:n
                Q[u,v] = Graphs.has_edge(graph, u, v) * C1 - (d[u] * d[v]) * C2
            end
        end
        return new(graph, Q)
    end
```

```
end

# Structure to hold column information
struct CPPColumn
    cost::Float64
    assignments::Vector{Bool}
end

# Structure to hold the solution of the RMP
struct CPPSolution
    y::Vector{Float64}
    objective::Float64
    duals::Dict{String, Any}
end

# Function to create initial columns for the column generation process
function cpp_create_initial_columns(data::CPPData, total::Int)
    n = nv(data.graph)
    columns = CPPColumn[]
    seen = Set{BitVector}()
    while length(columns) < total
        assignments = BitVector(rand(Bool, n))
        # Prevent creating columns with no assignments
        if !any(assignments)
            continue
        end
        # Prevent creating duplicate columns
        if assignments in seen
            continue
        end
        push!(seen, copy(assignments))
        cost = sum(
            data.Q[i,j] * assignments[i] * assignments[j]
            for i in 1:n, j in 1:n
        )
        push!(columns, CPPColumn(cost, assignments))
    end
    # Add singleton assignments for each vertex to ensure feasibility
    for v in 1:n
```

```
70          assignments = falses(n)
71          assignments[v] = true
72          cost = data.Q[v,v]
73          push!(columns, CPPColumn(cost, assignments))
74      end
75      return columns
76  end
77
78  # Function to solve the restricted master problem
79  function cpp_solve_restricted_master_problem(data::CPPData,
    columns::Vector{CPPColumn})
80      n = nv(data.graph)
81      n_cols = length(columns)
82      # Create the model
83      model = JuMP.Model(HiGHS.Optimizer)
84      # Silent mode (solver output is not printed)
85      JuMP.set_silent(model)
86      # Variables
87      @variable(model, 0 <= y[1:n_cols] <= 1)
88      # Objective: Maximize modularity
89      @objective(model, Max, sum(columns[c].cost * y[c] for c in
    1:n_cols))
90      # Constraints
91      @constraint(model, partition_satisfaction[j=1:n],
    sum(columns[i].assignments[j] * y[i] for i in 1:n_cols) == 1)
92      # Solve the model
93      JuMP.optimize!(model)
94      # Check the results and return dual prices
95      if JuMP.termination_status(model) == JuMP.OPTIMAL
96          y_values = JuMP.value.(y)
97          objective = JuMP.objective_value(model)
98          π = JuMP.dual.(partition_satisfaction)
99          duals = Dict("π" => π)
100         return CPPSolution(y_values, objective, duals)
101     else
102         error("RMP was not solved to optimality.")
103     end
104 end
105
106 # Function to solve the pricing subproblem
```

```
function cpp_solve_pricing_subproblem(data::CPPData,
columns::Vector{CPPColumn}, rmp_solution::CPPSolution;
tolerance::Float64 = 1e-6)
    # Number of vertices in the graph
    n = nv(data.graph)
    # Extract dual prices from the RMP solution
    π = rmp_solution.duals["π"]
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, y[i=1:n], Bin)
    @variable(model, z[i=1:n, j=i+1:n], Bin)
    # Linearization: z[i,j] = y[i] * y[j]
    @constraint(model, [i=1:n, j=i+1:n], z[i,j] <= y[i])
    @constraint(model, [i=1:n, j=i+1:n], z[i,j] <= y[j])
    @constraint(model, [i=1:n, j=i+1:n], z[i,j] >= y[i] + y[j] - 1)
    # Objective
    objective = AffExpr()
    for i in 1:n
        # Diagonal values Qii * xi
        JuMP.add_to_expression!(objective, data.Q[i,i], y[i])
        # Off-diagonal terms
        for j in i+1:n
            JuMP.add_to_expression!(objective,
                                    2.0 * data.Q[i,j],
                                    z[i,j])
        end
        # Duals
        JuMP.add_to_expression!(objective,
                                π[i],
                                y[i])
    end
    @objective(model, Max, objective)
    # Solve the model
    JuMP.optimize!(model)
    # Check the results
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        # Extract the solution
        reduced_cost = JuMP.objective_value(model)
```

```
146         if reduced_cost > tolerance
147             # Extract vertices assigned to this column
148             assignments = JuMP.value.(y) .> 0.5
149             # Check if the assignments are valid (not all zeros)
150             is_invalid = sum(assignments) == 0
151             # Check if is duplicated
152             is_duplicate = any(c -> c.assignments == assignments,
columns)
153             # Prevent creating columns with no assignments and
avoid duplicates
154             if !is_invalid && !is_duplicate
155                 # Calculate the cost of the new column
156                 cost = 0.0
157                 for i in 1:n
158                     for j in 1:n
159                         if assignments[i] && assignments[j]
160                             cost += data.Q[i,j]
161                         end
162                     end
163                 end
164                 # Create the new column
165                 new_column = CPPColumn(cost, assignments)
166                 # Add the new column to the list of columns
167                 push!(columns, new_column)
168             end
169         end
170         return reduced_cost
171     else
172         error("Pricing subproblem was not solved to optimality.")
173     end
174 end
175
176 # Function to solve the CPP using column generation
177 function cpp_solve_gc(data::CPPData; initial_columns_total::Int64 =
20, iterations::Int64 = 100, tolerance::Float64 = 1e-6,
print_feedback_interval::Int64 = 10)
178
179     println("--- Starting Column Generation ---\n")
180
181     println("| Iteration | Reduced Cost | RMP Objective | Columns
|")
```

```
println("|-----------|--------------|---------------|---------|")

    # DataFrame to store results of each iteration
    results = DataFrame(Iteration = Int[], ReducedCost = Float64[], RMPObjective = Float64[], Columns = Int[])

    # Vector to store reduced costs for each iteration
    reduced_cost_history = []

    # --- 1. Create initial columns ---
    columns = cpp_create_initial_columns(data, initial_columns_total)
    # columns = cpp_create_initial_columns(data)

    # Store best solution found
    best_solution = nothing

    # Dual Stabilization (Smoothing)
    # π_prev = zeros(nv(data.graph))
    π_prev = 0.1 .* rand(nv(data.graph))
    α = 0.3 # Smoothing factor (0.3 to 0.5 is usually optimal)

    # --- 2. Column generation loop ---
    for iteration in 1:iterations

        # --- 3. Solve the Restricted Master Problem (RMP) ---
        rmp_solution = cpp_solve_restricted_master_problem(data, columns)

        # Stabilize the duals before feeding them to the pricing subproblem
        if iteration > 1
            rmp_solution.duals["π"] = α * rmp_solution.duals["π"] + (1 - α) * π_prev
        end
        π_prev = copy(rmp_solution.duals["π"])

        # --- 4. Solve the Pricing Subproblem ---
        reduced_cost = cpp_solve_pricing_subproblem(data, columns, rmp_solution, tolerance = tolerance)

```

```
        # --- 5. Check if a new pattern was found ---
        if reduced_cost > tolerance
            if iteration == 1 || iteration % print_feedback_interval == 0
                @printf("| %9d | %12.3f | %13.3f | %7d |\n",
                        iteration,
                        reduced_cost,
                        rmp_solution.objective,
                        length(columns))
            end
            push!(reduced_cost_history, reduced_cost)
            best_solution = deepcopy(rmp_solution)
            # Add results to DataFrame
            push!(results, (iteration, reduced_cost, rmp_solution.objective, length(columns)))
        else
            println("\n--- Column Generation Complete ---\n")
            println("No more profitable columns found. Optimal LP solution reached.")
            break
        end
    end
    # --- 6. Generate reduced cost Plot ---
    cpp_plot_reduced_costs(reduced_cost_history)
    return best_solution, columns, results
end

# Set random seed for reproducibility
Random.seed!(42)

# Read data and compute modularity matrix
data = CPPData("data/karate.csv")

# Example usage
relaxed_solution, columns, results = cpp_solve_gc(data, initial_columns_total = 500, iterations = 200, tolerance = 1e-4, print_feedback_interval = 50);

println("\nFinal RMP Objective: ", relaxed_solution.objective)
```

Listing B.54: Julia code that implements the CG approach to solve the CPP for the dataset `karate.csv`.

Listing B.55 presents the output of the CG approach to solve the CPP for the dataset `karate.csv`. The output includes the iteration number, reduced cost, RMP objective value, and the evolution of the number of columns at each iteration. The process continues until no more profitable columns can be found, indicating that the optimal LP solution has been reached.

Output

```
--- Starting Column Generation ---

| Iteration | Reduced Cost | RMP Objective | Columns |
|-----------|--------------|---------------|---------|
|         1 |        0.160 |         0.074 |     535 |
|        50 |        0.028 |         0.420 |     582 |
|       100 |        0.007 |         0.420 |     629 |
|       150 |        0.003 |         0.420 |     667 |

--- Column Generation Complete ---

No more profitable columns found. Optimal LP solution reached.

Final RMP Objective: 0.41978961209730414
```

Listing B.55: Output of the CG approach to solve the CPP for the dataset `karate.csv`.

Figure B.31 presents a plot showing the convergence of the reduced cost over iterations for the CG approach to solve the CPP for the dataset `karate.csv`. The plot illustrates how the reduced cost evolves as new columns are generated and added to the master problem, eventually converging to zero when no more profitable columns can be found.

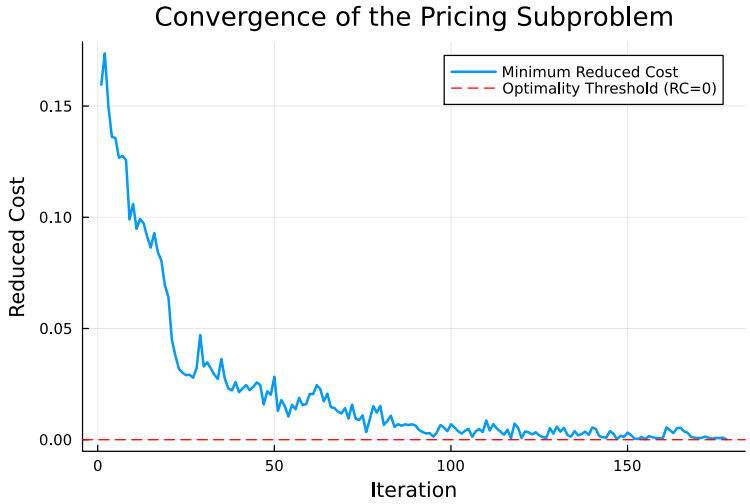


Figure B.31: Reduced cost convergence for the dataset `karate.csv`.

B.6.2.2 b) Solving RMIP using columns generated by CG

Listing B.56 presents the Julia code that implements the RMIP to solve the CPP for the dataset `karate.csv` using the columns generated in part (a). The code includes a function to solve the RMIP by imposing integrality constraints on the master problem variables. The output includes the optimal objective value of the RMIP, the selected columns (communities), and the vertices assigned to each community. Additionally, a plot is generated to visualize the solution.

```
# Function to solve the Restricted Master Integer Problem (RMIP)
function cpp_solve_restricted_master_integer_problem(data::CPPData, columns::Vector{CPPColumn})
    n = nv(data.graph)
    n_cols = length(columns)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, y[1:n_cols], Bin)
    # Objective: Maximize modularity
    @objective(model, Max, sum(columns[c].cost * y[c] for c in 1:n_cols))
    # Constraints
    @constraint(model, [j=1:n], sum(columns[i].assignments[j] * y[i] for i in 1:n_cols) == 1)
    # Solve the model
```

```
    JuMP.optimize!(model)
    # Check the results and return dual prices
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        objective = JuMP.objective_value(model)
        println("RMIP solution found with objective value: ",
objective)
        solution = zeros(Int64,n)
        selected_columns = columns[JuMP.value.(y) .> 0.5]
        for (idx, col) in enumerate(selected_columns)
            assigned_vertices = [j for j in 1:n if
col.assignments[j]]
            println("Community $(idx): ", assigned_vertices)
            solution[assigned_vertices] .= idx
        end
        # Plot the solution
        cpp_plot_solution(data.graph, solution)
    else
        error("RMP was not solved to optimality.")
    end
end

# Example usage of the integer version of the RMP
cpp_solve_restricted_master_integer_problem(data, columns)
```

Listing B.56: Julia code that implements RMIP to solve the CPP for the dataset `karate.csv`.

Listing B.57 presents the output of the RMIP approach to solve the CPP for the dataset `karate.csv`. The output includes the optimal objective value and the vertices assigned to each community.

Output

```
RMIP solution found with objective value: 0.41978961209730414
Community 1: [9, 10, 15, 16, 19, 21, 23, 27, 30, 31, 33, 34]
Community 2: [5, 6, 7, 11, 17]
Community 3: [24, 25, 26, 28, 29, 32]
Community 4: [1, 2, 3, 4, 8, 12, 13, 14, 18, 20, 22]
```

Listing B.57: Output of the RMIP approach to solve the CPP for the dataset `karate.csv`.

Figure B.32 presents a plot visualizing the solution obtained from the RMIP approach to solve the CPP for the dataset `karate.csv`. The plot shows the communities formed.

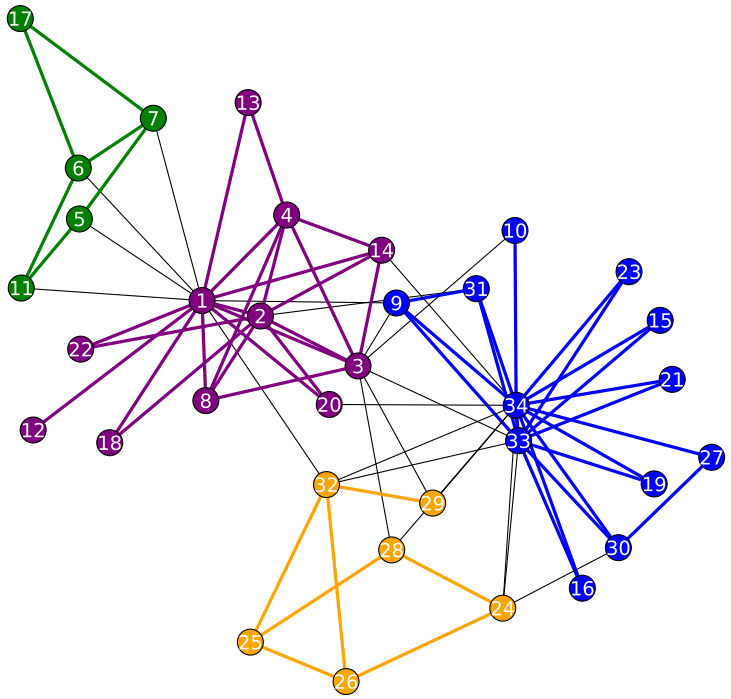


Figure B.32: RMIP solution for the dataset `karate.csv`.

B.6.2.3 c) Solving CPMP using ILP

Listing B.58 presents the Julia code that implements an Integer Linear Programming (ILP) approach to solve the CPP for the dataset `karate.csv`. The code includes a function to solve the ILP by defining binary variables for vertex assignments and community memberships, along with the necessary constraints. The output includes the optimal objective value of the ILP, the communities formed, and the vertices assigned to each community. Additionally, a plot is generated to visualize the solution.

```
# Function to solve the CPP using Integer Linear Programming (ILP)
function cpp_solve_ilp(data::CPPData)
    # Get number of vertices and edges
    n = Graphs.nv(data.graph)
    m = Graphs.ne(data.graph)
    # Auxiliary variables to simplify notation
    V = 1:n
    # Modularity matrix Q
```

```
    Q = zeros(n,n)
    d = Graphs.degree(data.graph)
    for u in V
        for v in V
            a_ij = Graphs.has_edge(data.graph, u, v) ? 1 : 0
            Q[u,v] = a_ij/(2*m) - (d[u] * d[v]) / (4*m^2)
        end
    end
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define the decision variables
    @variable(model, x[u in V, v in V], Bin)
    # Objective: maximize the weight of the selected edges
    @objective(model, Max, sum(Q[u,v] * x[u,v] for u in V for v in V))
    # Transitive Constraints: ensure that the selected edges form a valid partition
    for u in V
        for v in V
            for w in V
                @constraint(model,  x[u,v] + x[v,w] - x[u,w] <= 1)
                @constraint(model,  x[u,v] - x[v,w] + x[u,w] <= 1)
                @constraint(model, -x[u,v] + x[v,w] + x[u,w] <= 1)
            end
        end
    end
    # Run the solver
    JuMP.optimize!(model)
    # Get the objective (Modularity)
    modularity = JuMP.objective_value(model)
    println("Optimal solution found with objective value: ", modularity)
    # Extract the optimal partition
    partition = zeros(Int64, n)
    partition_id = 1
    for u in V
        # If vertex u is not yet assigned to a partition
        if partition[u] == 0
            print("Community $partition_id: ")
```

```
            for v in V
                # If vertex v is in the same partition as u
                if JuMP.value(x[u,v]) > 0.5
                    # Assign both u and v to the same partition
                    partition[v] = partition_id
                    print("$v ")
                end
            end
            print("\n")
            partition_id += 1
        end
    end
    # Plot the solution
    cpp_plot_solution(data.graph, partition)
end

# Example usage of the ILP approach
cpp_solve_ilp(data)
```

Listing B.58: Julia code that implements ILP to solve the CPP for the dataset `karate.csv`.

Listing B.59 presents the output of the ILP approach to solve the CPP for the dataset `karate.csv`. The output includes the optimal objective value and the communities formed with their corresponding vertices.

Output

```
Optimal solution found with objective value: 0.41978961209730403
Community 1: 1 2 3 4 8 12 13 14 18 20 22
Community 2: 5 6 7 11 17
Community 3: 9 10 15 16 19 21 23 27 30 31 33 34
Community 4: 24 25 26 28 29 32
```

Listing B.59: Output of the ILP approach to solve the CPP for the dataset `karate.csv`.

Figure B.33 presents a plot visualizing the solution obtained from the ILP approach to solve the CPP for the dataset `karate.csv`. The plot shows the communities formed.

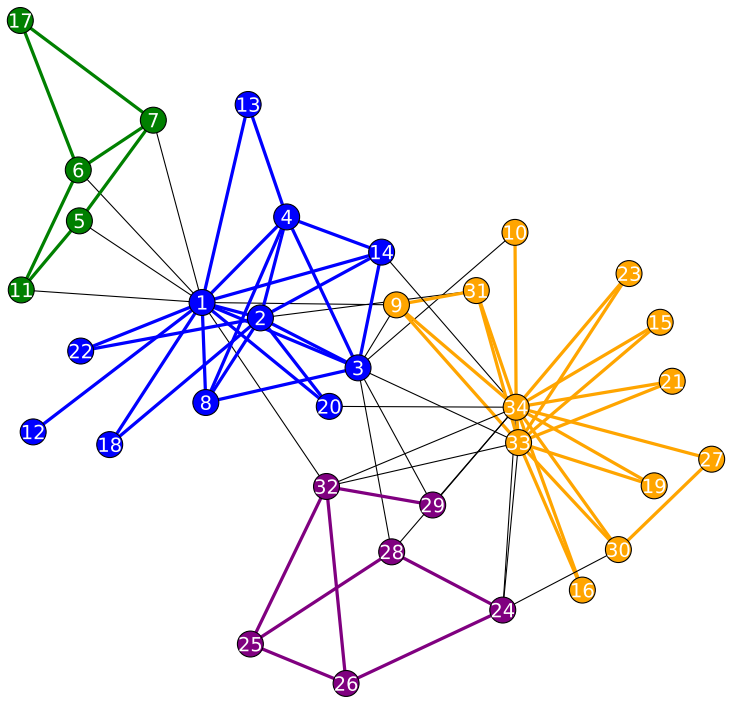


Figure B.33: ILP solution for the dataset `karate.csv`.

Discussion

Table B.3 presents the objective values obtained by the Restricted Master Problem (RMP), the Restricted Master Integer Problem (RMIP), and the Integer Linear Programming (ILP) model, together with the corresponding integrality gap.

Technique	Objective	Gap (%)
CG	0.42	0.00
RMIP	0.42	0.00
ILP	0.42	N/A

Table B.3: Comparison of CPP solutions with different models.

The Column Generation approach produced a final objective value of 0.42, providing a lower bound with an integrality gap of 0.00% relative to the optimal integer solution. The columns generated during the column generation process were sufficient for the RMIP to recover the global optimum, yielding the same objective value as the original ILP model (0.42) without requiring additional branching procedures.

B.6.3 Exercise 6.3

To solve the Graph Coloring Problem (GCP) using Column Generation (CG), we will implement the following steps in Julia:

1. Define a column: a column represents a potential color class (subset of vertices).
2. Create initial columns: create singleton columns containing only one vertex. The singleton columns guarantee that the initial Restricted Master Problem is feasible, since every vertex can always be assigned to its own color class.
3. Solve the Restricted Master Problem (RMP): solve the linear programming relaxation of the master problem using the current set of columns. The RMP provides the dual prices associated with the coloring constraints that ensure each vertex is assigned to exactly one color class.
4. Solve the pricing subproblem: using the dual prices obtained from the RMP, solve the pricing subproblem to identify new columns (color classes) with negative reduced cost. These columns are added to the RMP to improve the solution.
5. Iterate: repeat steps 3 and 4 until no columns with negative reduced cost are found, indicating that the optimal solution of the RMP has been reached.

B.6.3.1 a) Solving GCP using Column Generation

Listing B.60 presents the Julia code that implements the CG approach to solve the GCP for the dataset `tutte_graph.csv`. The code includes functions to create initial columns, solve the RMP, solve the pricing subproblem, and perform the column generation loop. The final output includes the optimal objective value of the RMP and a plot showing the convergence of the reduced cost over iterations.

```
using JuMP       # Modeling language
using HiGHS      # Solver
using DataFrames # For handling data frames
using Graphs     # For graph data structures
using Printf     # For formatted printing

# Utility functions for plotting convergence
include("utils/cg_gcp_utils.jl")

# Structure to hold the solution of the RMP
```

```
struct GCPSolution
    selected_columns::Vector{Float64}
    objective::Float64
    duals::Dict{String, Any}
end

# Function to create initial columns for the column generation process
function gcp_create_initial_columns(graph::Graphs.Graph)
    # Each vertex is an independent set
    n = nv(graph)
    columns = Vector{BitVector}(undef, n)
    for i in 1:n
        independent_set = falses(n)
        independent_set[i] = true
        columns[i] = independent_set
    end
    return columns
end

# Function to solve the restricted master problem
function gcp_solve_restricted_master_problem(graph::Graphs.Graph, columns::Vector{BitVector})
    V = 1:nv(graph)
    C = 1:length(columns)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables (independent set)
    @variable(model, 0 <= y[c in C] <= 1)
    # Objective: Minimize total colors used
    @objective(model, Min, sum(y))
    # Constraints: Each vertex must be covered by at least one selected independent set
    @constraint(model, cover[v in V], sum(columns[c][v] * y[c] for c in C) >= 1)
    # Solve the model
    JuMP.optimize!(model)
    # Check the results and return dual prices
    if JuMP.termination_status(model) == JuMP.OPTIMAL
```

```
48          selected_columns = JuMP.value.(y)
49          objective = JuMP.objective_value(model)
50          π = JuMP.dual.(cover)
51          duals = Dict("π" => π)
52          return GCPSolution(selected_columns,
53                             objective,
54                             duals)
55      else
56          error("RMP was not solved to optimality.")
57      end
58  end
59
60  # Function to solve the pricing subproblem
61  function gcp_solve_pricing_subproblem(graph::Graphs.Graph,
    columns::Vector{BitVector}, rmp_solution::GCPSolution;
    tolerance::Float64 = 1e-6)
62      V = 1:nv(graph)
63      π = rmp_solution.duals["π"]
64      # Create the model
65      model = JuMP.Model(HiGHS.Optimizer)
66      # Silent mode (solver output is not printed)
67      JuMP.set_silent(model)
68      # Variables
69      @variable(model, x[v in V], Bin)
70      # Objective: Maximize the dual weights collected
71      @objective(model, Max, sum(π[v] * x[v] for v in V))
72      # Adjacency constraint: adjacent vertices cannot both be
    selected
73      for e in Graphs.edges(graph)
74          @constraint(model, x[e.src] + x[e.dst] <= 1)
75      end
76      # Solve the model
77      JuMP.optimize!(model)
78      # Check the results
79      if JuMP.termination_status(model) == JuMP.OPTIMAL
80          reduced_cost = 1.0 - JuMP.objective_value(model)
81          # If the reduced cost is negative, we have found a new
    column to add
82          if reduced_cost < -tolerance
83              # Extract vertices assigned to this column
84              independent_set = JuMP.value.(x) .> 0.5
```

```
85
86              # Check if the assignments are valid (not all zeros)
87              is_invalid = sum(independent_set) == 0
88
89              # Check if is duplicated
90              is_duplicate = any(c -> c == independent_set, columns)
91
92              # Prevent creating columns with no assignments or
duplicates
93              if !is_invalid && !is_duplicate
94                  push!(columns, independent_set)
95              end
96          end
97          return reduced_cost
98      else
99          error("Pricing subproblem was not solved to optimality.")
100     end
101 end
102
103 # Function to solve the GCP using column generation
104 function gcp_solve_gc(graph::Graphs.Graph; iterations::Int64 = 100,
tolerance::Float64 = 1e-6, print_feedback_interval::Int64 = 10)
105
106     println("--- Starting Column Generation ---\n")
107
108     println("| Iteration | Reduced Cost | RMP Objective | Columns
|")
109
println("|-----------|--------------|---------------|---------|")
110
111     # DataFrame to store results of each iteration
112     results = DataFrame(Iteration = Int[], ReducedCost = Float64[],
RMPObjective = Float64[], Columns = Int[])
113
114     # Vector to store reduced costs for each iteration
115     reduced_cost_history = []
116
117     # --- 1. Create initial columns ---
118     columns = gcp_create_initial_columns(graph)
119
120     # Store best solution found
121     best_solution = nothing
```

```
122
123     # --- 2. Column generation loop ---
124     for iteration in 1:iterations
125         # --- 3. Solve the Restricted Master Problem (RMP) ---
126         rmp_solution = gcp_solve_restricted_master_problem(graph,
columns)
127
128         # --- 4. Solve the Pricing Subproblem ---
129         reduced_cost = gcp_solve_pricing_subproblem(graph, columns,
rmp_solution; tolerance = tolerance)
130
131         # --- 5. Check if a new pattern was found ---
132         if reduced_cost < -tolerance
133             if iteration == 1 || iteration %
print_feedback_interval == 0
134                 @printf("| %9d | %12.3f | %13.3f | %7d |\n",
135                         iteration,
136                         reduced_cost,
137                         rmp_solution.objective,
138                         length(columns))
139             end
140             push!(reduced_cost_history, reduced_cost)
141             best_solution = deepcopy(rmp_solution)
142             # Add results to DataFrame
143             push!(results, (iteration, reduced_cost,
rmp_solution.objective, length(columns)))
144         else
145             println("\n--- Column Generation Complete ---\n")
146             println("No more profitable columns found. Optimal LP
solution reached.")
147             break
148         end
149     end
150     # --- 6. Generate reduced cost Plot ---
151     gcp_plot_reduced_costs(reduced_cost_history)
152     return best_solution, columns, results
153 end
154
155 # Example usage
156 graph = Graphs.loadgraph("data/tutte_graph.csv")
157
158 # Run the column generation algorithm for GCP
```

```
159 relaxed_solution, columns, results = gcp_solve_gc(graph, iterations
    = 500, tolerance = 1e-3, print_feedback_interval = 50);
160
161 println("\nFinal RMP Objective: ", relaxed_solution.objective)
```

Listing B.60: Julia code that implements the CG approach to solve the GCP for the dataset `tutte_graph.csv`.

Listing B.61 presents the output of the CG approach to solve the GCP for the dataset `tutte_graph.csv`. The output includes the iteration number, reduced cost, RMP objective value, and the evolution of the number of columns at each iteration. The process continues until no more profitable columns can be found, indicating that the optimal LP solution has been reached.

Output

```
--- Starting Column Generation ---

| Iteration | Reduced Cost | RMP Objective | Columns |
|-----------|--------------|---------------|---------|
|         1 |      -18.000 |        46.000 |      47 |
|        50 |       -0.984 |         3.165 |      96 |
|       100 |       -0.465 |         2.760 |     146 |
|       150 |       -0.159 |         2.639 |     196 |
|       200 |       -0.083 |         2.553 |     246 |
|       250 |       -0.055 |         2.508 |     296 |

--- Column Generation Complete ---

No more profitable columns found. Optimal LP solution reached.

Final RMP Objective: 2.5005387931034484
```

Listing B.61: Output of the CG approach to solve the GCP for the dataset `tutte_graph.csv`.

Figure B.34 presents a plot showing the convergence of the reduced cost over iterations for the CG approach to solve the GCP for the dataset `tutte_graph.csv`. The plot illustrates how the reduced cost evolves as new columns are generated and added to the master problem, eventually converging to zero when no more profitable columns can be found.

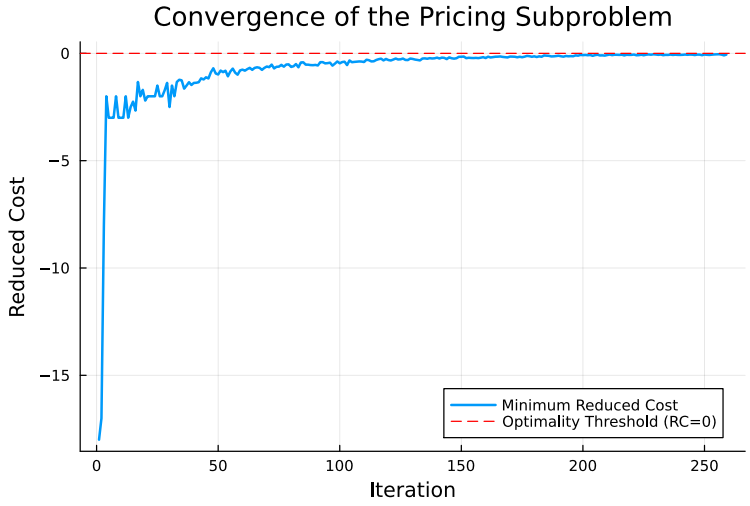


Figure B.34: Reduced cost convergence for the dataset `tutte_graph.csv`.

B.6.3.2 b) Solving RMIP using columns generated by CG

Listing B.62 presents the Julia code that implements the RMIP to solve the GCP for the dataset `tutte_graph.csv` using the columns generated in part (a). The code includes a function to solve the RMIP by imposing integrality constraints on the master problem variables. The output includes the optimal objective value of the RMIP, the selected columns (colors), and the vertices assigned to each color. Additionally, a plot is generated to visualize the solution.

```
# Function to solve the Restricted Master Integer Problem (RMIP)
function gcp_solve_restricted_master_integer_problem(graph::Graphs.SimpleGraph, columns::Vector{BitVector})
    V = 1:nv(graph)
    C = 1:length(columns)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables (independent set)
    @variable(model, y[c in C], Bin)
    # Objective: Minimize total colors used
    @objective(model, Min, sum(y))
    # Constraints: Each vertex must be covered by at least one selected independent set
    @constraint(model, cover[v in V], sum(columns[c][v] * y[c] for c in C) >= 1)
```

```
    # Solve the model
    JuMP.optimize!(model)
    # Check the results and return dual prices
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        objective = JuMP.objective_value(model)
        println("RMIP solution found with objective value: ",
objective)
        solution = zeros(Int, length(V))
        selected_columns = [c for c in C if JuMP.value(y[c]) > 0.5]
        for (idx, col) in enumerate(selected_columns)
            vertices = findall(columns[col])
            println("Color $(idx): ", vertices)
            solution[vertices] .= idx
        end
        gcp_plot_solution(graph, solution)
    else
        error("RMP was not solved to optimality.")
    end
end

# Example usage of the integer version of the RMP
gcp_solve_restricted_master_integer_problem(graph, columns)
```

Listing B.62: Julia code that implements RMIP to solve the GCP for the dataset `tutte_graph.csv`.

Listing B.63 presents the output of the RMIP approach to solve the GCP for the dataset `tutte_graph.csv`. The output includes the optimal objective value of the RMIP, the selected columns (colors), and the vertices assigned to each color.

Output

```
RMIP solution found with objective value: 2.9999999999999867
Color 1: [2, 3, 4, 10, 13, 15, 16, 18, 22, 26, 28, 30, 33, 34, 37, 39,
42, 45, 46]
Color 2: [1, 5, 7, 9, 14, 17, 19, 20, 22, 25, 27, 29, 31, 36, 38, 40,
41, 43]
Color 3: [2, 4, 6, 8, 11, 12, 14, 17, 21, 23, 24, 29, 32, 34, 35, 37,
44, 46]
```

Listing B.63: Output of the RMIP approach to solve the GCP for the dataset `tutte_graph.csv`.

Figure B.35 presents a plot showing the RMIP solution for the dataset `tutte_graph.csv`. The plot illustrates the assignment of colors to vertices based on the selected columns from the RMIP solution.

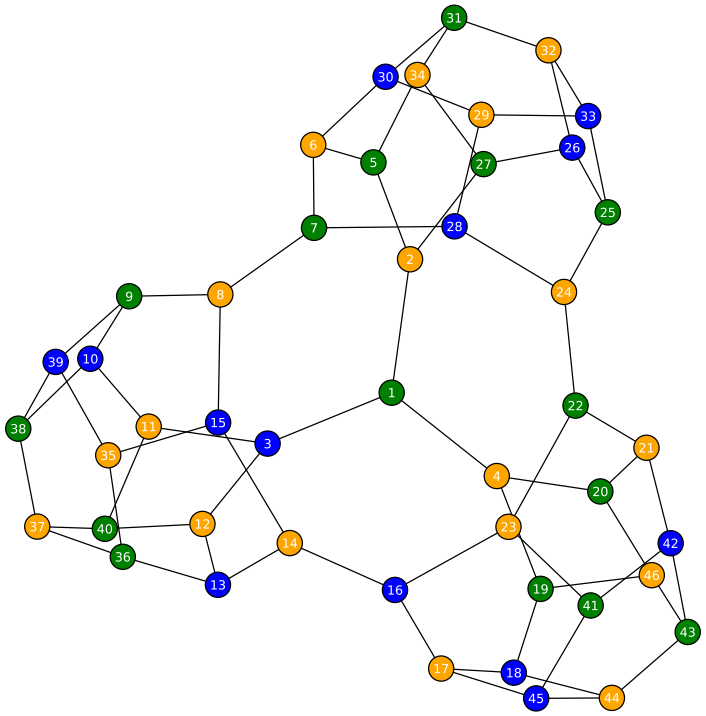


Figure B.35: RMIP solution for the dataset `tutte_graph.csv`.

B.6.3.3 c) Solving GCP using ILP

Listing B.64 presents the Julia code that implements ILP to solve the GCP for the dataset `tutte_graph.csv`. The code includes a function to solve the GCP using Integer Linear Programming (ILP) by defining decision variables, constraints, and the objective function. The output includes the optimal objective value of the ILP, the selected colors, and the vertices assigned to each color. Additionally, a plot is generated to visualize the solution.

```
# Function to solve the GCP using Integer Linear Programming (ILP)
function gcp_solve_ilp(graph::Graphs.SimpleGraph)
    # Range of vertices
    N = 1:Graphs.nv(graph)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define the decision variables
```

```
    @variable(model, x[u in N, c in N], Bin)
    @variable(model, y[c in N], Bin)
    # Objective function: minimize the total of selected colors
    @objective(model, Min, sum(y))
    # Each vertex must be assigned to exactly one color
    @constraint(model, [u in N], sum(x[u, c] for c in N) == 1)
    # If a vertex uses color c, then color c is active
    @constraint(model, [u in N, c in N], x[u, c] <= y[c])
    # No two adjacent vertices share the same color
    for e in Graphs.edges(graph)
        u, v = e.src, e.dst
        for c in N
            @constraint(model, x[u, c] + x[v, c] <= 1)
        end
    end
    # Run the solver
    JuMP.optimize!(model)
    # Check the results
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        # Get the objective (Chromatic number)
        objective = Int(round(JuMP.objective_value(model)))
        println("Optimal solution found with objective value:
$objective")
        # Extract the optimal color assignments
        x_opt = JuMP.value.(x)
        y_opt = JuMP.value.(y)
        solution = zeros(Int, length(N))
        # Print the color assignments
        selected_colors = [c for c in N if y_opt[c] > 0.5]
        for (id, pos) in enumerate(selected_colors)
            vertices = [v for v in N if x_opt[v, pos] > 0.5]
            println("Color $id: $vertices")
            solution[vertices] .= id
        end
        # Plot the graph with the assigned colors
        gcp_plot_solution(graph, solution)
    else
        error("Model was not solved to optimality.")
    end
end

```

```
# Example usage of the ILP approach
gcp_solve_ilp(graph)
```

Listing B.64: Julia code that implements ILP to solve the GCP for the dataset `tutte_graph.csv`.

Listing B.65 presents the output of the ILP approach to solve the GCP for the dataset `tutte_graph.csv`. The output includes the optimal objective value of the ILP, the selected colors, and the vertices assigned to each color.

Output

```
Optimal solution found with objective value: 3
Color 1: [7, 9, 16, 29, 31, 36, 40, 42, 44]
Color 2: [2, 3, 4, 6, 10, 13, 15, 18, 21, 23, 24, 26, 33, 34, 37, 39,
45, 46]
Color 3: [1, 5, 8, 11, 12, 14, 17, 19, 20, 22, 25, 27, 28, 30, 32, 35,
38, 41, 43]
```

Listing B.65: Output of the ILP approach to solve the GCP for the dataset `tutte_graph.csv`.

Figure B.36 presents a plot showing the ILP solution for the dataset `tutte_graph.csv`. The plot illustrates the assignment of colors to vertices based on the optimal solution obtained from the ILP model.

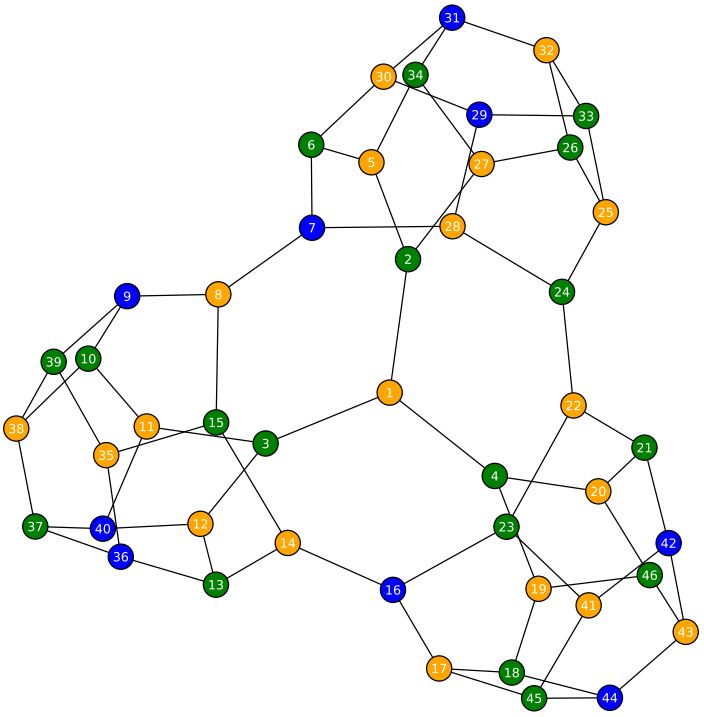


Figure B.36: ILP solution for the dataset `tutte_graph.csv`.

Discussion

Table B.4 presents the objective values obtained by the Restricted Master Problem (RMP), the Restricted Master Integer Problem (RMIP), and the Integer Linear Programming (ILP) model, together with the corresponding integrality gap.

Technique	Objective	Gap (%)
CG	2.5	16.66
RMIP	3	0.00
ILP	3	N/A

Table B.4: Comparison of GCP solutions with different models.

The column generation approach produced a final RMP objective value of 2.5, providing a lower bound with an integrality gap of 16.66% relative to the optimal integer solution. Since the RMIP and the original ILP model achieved the same objective value (3), the integrality gap is 0.00%. This indicates that the linear relaxation provided by the RMP was not integral for this instance, but the set of columns generated during the column generation process was sufficient to

recover the optimal integer solution. Consequently, the RMIP matched the ILP optimum without requiring any additional branching procedures.

B.7 Chapter 7

B.7.1 Exercise 7.1

The following subsections consider the model from Equation (B.13).

$$E: \quad \max_{x_1,x_2} \quad x_1 + 2x_2 \tag{B.13.1}$$

$$\text{s.t.} \quad 2x_1 + x_2 \leq 2 \tag{B.13.2}$$

$$x_1, x_2 \in \{0, 1\} \tag{B.13.3}$$

B.7.1.1 a) Make a graphical representation of the problem and find the optimal solution and the linear programming relaxation.

Listing B.66 presents the Julia code that implements the graphical representation of the problem defined in Equation (B.13). The code includes the definition of the constraints, objective function, and feasible region. It also plots the constraints, objective function, and feasible points, highlighting the optimal solution and the linear programming relaxation.

```
using Plots, LaTeXStrings
gr()

# Define colors
cb_blue   = "#377eb8"
cb_orange = "#ff7f00"
cb_green  = "#4daf4a"
cb_purple = "#984ea3"
cb_black  = "#000000"
cb_gray   = "#bdbdbd"
cb_red    = "#e41a1c"


cb_new   = "#aeb42dff"
cb_red    = "#e41a1c"
cb_gray   = "#bdbdbd"
cb_black  = "#000000"

# Define x values
x = 0:0.01:2
```

```
21
22 # Define constraint lines
23 line1 = -2 .* x .+ 2        # x2 = -2x1 + 2
24 line2 = zero.(x)            # x2 = 0
25 line3 = one.(x)             # x2 = 1
26 line_obj = -0.5 .* x .+ 1.5 # x2 = -0.5x1 + 1.5 (dashed red)
27
28 # Plot constraints
29 plot(x, line1, color=cb_blue, label=L"2x_1 + x_2 = 2", linewidth=3)
30 plot!([0, 0], [0, 2], color=cb_purple, label=L"x_1 = 0",
   linewidth=3)
31 plot!([1, 1], [0, 2], color=cb_green, label=L"x_1 = 1", linewidth=3)
32 plot!(x, line2, color=cb_new, label=L"x_2 = 0", linewidth=3)
33 plot!(x, line3, color=cb_orange, label=L"x_2 = 1", linewidth=3)
34
35 # Dashed red line (objective)
36 plot!(x, line_obj, color=cb_red, linewidth=3, linestyle=:dash,
   label=L"x_1 + 2x_2 = c")
37
38 # Fill between: Region 1 (between x₂=0 and x₂=1, for x₁ in [0,0.5])
39 x_fill1 = 0:0.01:0.5
40 fill1_top = one.(x_fill1)
41 fill1_bot = zero.(x_fill1)
42 plot!(x_fill1, fill1_top, fillrange=fill1_bot, fillalpha=0.3,
   color=cb_gray, label="")
43
44 # Fill between: Region 2 (between x₂=0 and x₂=-2x₁+2, for x₁ in
   [0.5,1])
45 x_fill2 = 0.5:0.01:1.0
46 fill2_top = -2 .* x_fill2 .+ 2
47 fill2_bot = zero.(x_fill2)
48 plot!(x_fill2, fill2_top, fillrange=fill2_bot, fillalpha=0.3,
   color=cb_gray, label="Feasible Region")
49
50 # Bullet points
51 scatter!([1], [0], color=cb_black, markersize=6, label="Feasible
   Point")
52 scatter!([0], [1], color=cb_black, markershape = :diamond,
   markersize=6, label=L"z_{IP}")
53 # Insert a point at (0.5,1) with black color and square type
54 scatter!([0.5], [1], color=cb_black, markershape = :square,
   markersize=6, label=L"z_{LP}")
```

```
55
56 # Annotate points (optional)
57 annotate!(1.05, 0.1, text("(1,0)", :left))
58 annotate!(0.05, 1.1, text("(0,1)", :left))
59
60 # Axis labels and layout
61 xlabel!(L"x_1")
62 ylabel!(L"x_2")
63 title!("Graphical representation")
64 xlims!(0, 2)
65 ylims!(0, 2)
66 plot!(legend=true, grid=true, size=(600,500))
```

Listing B.66: Julia code to create the graphical representation of model presented in Equation (B.13).

Figure B.37 presents a plot showing the graphical representation of the problem defined in Equation (B.13). The plot illustrates the constraints, objective function, and feasible region, highlighting the optimal solution and the linear programming relaxation.

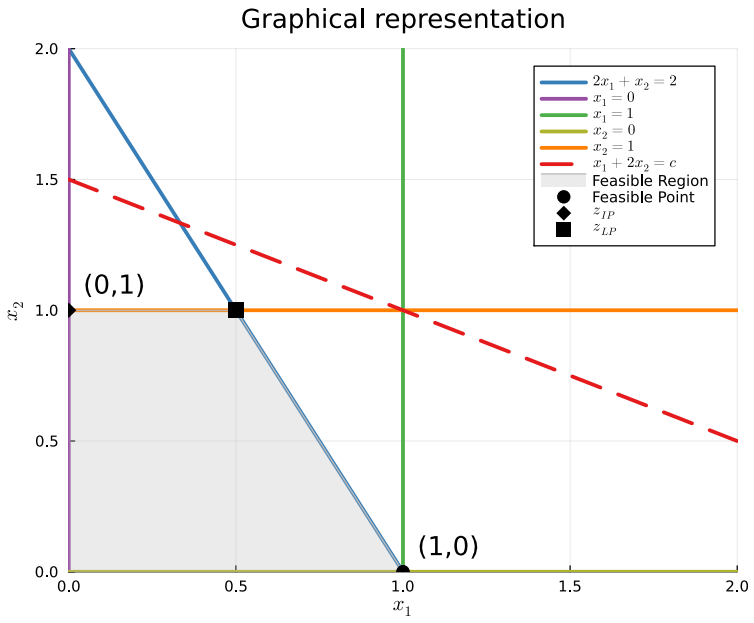


Figure B.37: Graphical representation of the problem defined in Equation (B.13).

B.7.1.2 b) Find the Lagrangian relaxation for the constraint $2x_1 + x_2 \leq 2$. Make a graphical representation of the dual's solution. Is there a duality gap?

The Lagrangian relaxation is:

$$\mathrm{L}(\lambda): \quad \max_{x_1,x_2} \quad 2\lambda + (1-2\lambda)x_1 + (2-\lambda)x_2 \tag{B.14.1}$$

$$\text{s.t.} \quad x_1, x_2 \in \{0,1\} \tag{B.14.2}$$

and the dual problem is:

$$D: \min_{\lambda \geq 0} z_{L(\lambda)} \tag{B.15}$$

We can find the optimal value of the dual problem (z_D) graphically by plotting the function $z_{L(\lambda)}$ for $\lambda \geq 0$ and identifying its minimum point.

Since $L(\lambda)$ is a maximization problem over binary variables, the optimal values of x_1 and x_2 depend on the signs of their respective coefficients:

$$z_{L(\lambda)} = 2\lambda + \underbrace{(1-2\lambda)}_{c_1} x_1 + \underbrace{(2-\lambda)}_{c_2} x_2 \tag{B.16}$$

Since we are maximizing and $x_1, x_2 \in \{0,1\}$:

- x_1 should be 1 if $c_1 \geq 0$ and 0 if $c_1 < 0$.
- x_2 should be 1 if $c_2 \geq 0$ and 0 if $c_2 < 0$.

To find the boundaries, we set the coefficients to zero:

- $c_1 = 0 \Rightarrow 1 - 2\lambda = 0 \Rightarrow \boldsymbol{\lambda = 0.5}$
- $c_2 = 0 \Rightarrow 2 - \lambda = 0 \Rightarrow \boldsymbol{\lambda = 2}$

These values divide the possible values of λ into the three intervals: $[0, 0.5)$, $[0.5, 2)$, and $[2, \infty)$. Table B.5 maps these intervals to the corresponding optimal binary decisions and the resulting linear expression for $z_{L(\lambda)}$. At the breakpoints $\lambda = 0.5$ and $\lambda = 2$, the Lagrangian subproblem admits multiple optimal solutions; the half-open interval convention is adopted for notational clarity.

$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{\lambda}$	$\boldsymbol{z_{L(\lambda)}}$
0	0	$\lambda \geq 2$	2λ
0	1	$0.5 \leq \lambda < 2$	$\lambda + 2$
1	1	$0 \leq \lambda < 0.5$	$3 - \lambda$

Table B.5: Optimal solutions of the Lagrangian subproblem over the intervals of λ.

Plotting these segments as a function of λ yields the graph in the following figure. As can be seen, the minimum value is obtained at the point where the function's definition changes, at $\lambda = 0.5$. Therefore, the optimal value of the dual problem is $z_{\mathrm{D}} = L(0.5) = 2.5$. The duality gap for this problem is

$$z_{\mathrm{D}} - z_{\mathrm{IP}} = 2.5 - 2 = 0.5. \tag{B.17}$$

This gap indicates that the Lagrangian relaxation does not provide the same optimal value as the original integer problem. The value of the dual's solution ($z_{\mathrm{D}} = 2.5$) corresponds to the solution of the linear programming relaxation (z_{LP}).

Listing B.67 presents the Julia code that implements the graphical representation of the optimal value of the Lagrangian Dual Problem. The code includes the definition of the piecewise linear function $z_{L(\lambda)}$ and plots the segments corresponding to different intervals of λ. It also highlights the optimal dual solution at $\lambda = 0.5$, where the minimum value is achieved.

```
using Plots, LaTeXStrings

# Define colors
cb_blue   = "#377eb8"
cb_orange = "#ff7f00"
cb_green  = "#4daf4a"

# Define the domain
λ = 0:0.01:4

# Define the piecewise functions
f1(x) = 2x
f2(x) = x + 2
f3(x) = -x + 3

# Base plot (empty, to set axis)
plot(
    xlabel = L"\lambda", ylabel = L"z_{L(\lambda)}",
    xlims = (0, 4), ylims = (0, 7.5),
    xticks = [0.5, 1, 2, 3, 4],
    yticks = [2.5, 5, 7.5],
    legend = false,
    grid = true,
    framestyle = :origin,
    title = "Finding the optimal value of the\nLagrangian Dual
Problem"
```

```
)

# Blue segments (full domain)
plot!(λ, f1.(λ), color = cb_blue, label=L"2\lambda",
linestyle=:dash, lw=2)
plot!(λ, f2.(λ), color = cb_orange, label=L"\lambda+2",
linestyle=:dash, lw=2)
plot!(λ, f3.(λ), color = cb_green, label=L"3-\lambda",
linestyle=:dash, lw=2)

# Red segments (limited domain to overwrite parts)
plot!(2:0.01:4, f1.(2:0.01:4), color = :red, label="", lw=3)
plot!(0.5:0.01:2, f2.(0.5:0.01:2), color = :red, label="", lw=3)
plot!(0:0.01:0.5, f3.(0:0.01:0.5), color = :red, label="", lw=3)

# Highlight the optimal dual solution
scatter!([0.5], [2.5], color=:black, markersize=8, label=L"z_D")

plot!(legend=true, grid=true, size=(600,500))
```

Listing B.67: Julia code to create the graphical representation of optimal value of the Lagrangian Dual Problem.

Figure B.38 presents a plot showing the graphical representation of the optimal value of the Lagrangian Dual Problem. The plot illustrates the piecewise linear function $z_{L(\lambda)}$ and highlights the optimal dual solution at $\lambda = 0.5$, where the minimum value is achieved.

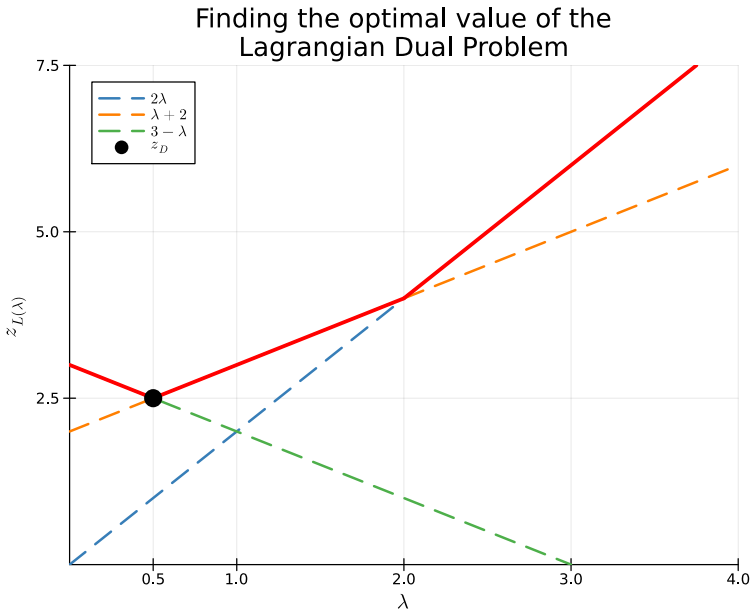


Figure B.38: Graphical representation of the problem defined in Equation (B.13).

B.7.1.3 c) Find the optimal solution of the dual using the subgradient method.

Listing B.68 presents the Julia code that implements the subgradient method to find the optimal solution of the dual problem. The code includes functions to solve the original integer program exactly, perform Lagrangian relaxation, and iterate through the subgradient method to update the Lagrange multiplier. It also tracks the lower and upper bounds for convergence analysis.

```
using JuMP   # Modeling language
using HiGHS  # Solver
using Printf # For formatted output

# Utility functions for plotting bounds convergence
include("utils/lr_utils.jl")

# Function to solve the original integer program exactly
function solve_exact()
    # Create the model
    model =  JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define binary variables
    @variable(model, x[1:2], Bin)
```

```
    # Define the objective function
    @objective(model, Max, x[1] + 2 * x[2])
    # Define the constraint
    @constraint(model, 2 * x[1] + x[2] <= 2)
    # Solve the model
    JuMP.optimize!(model)
    # Get results
    x_opt =  JuMP.value.(x)
    obj_value = JuMP.objective_value(model)
    return x_opt, obj_value
end

# Function to perform Lagrangian relaxation
function solve_lagrangian(iterations = 100)
    # Initialize variables
    x = [0, 0]  # Initial solution
    λ = 0.0     # Initial Lagrange multiplier
    λ_best = λ  # To track the best λ found
    α = 0.1     # Step size
    LB = -Inf   # Best lower bound found so far
    UB = Inf    # Best upper bound found so far
    LB_history = Float64[] # To store LB values for plotting
    UB_history = Float64[] # To store UB values for plotting

    # Logic to stop if no improvement in UB for a certain number of iterations
    no_improve_limit = 10
    no_improve_counter = 0

    println("-------------- Starting Lagrangian Relaxation --------------")
    println("Iter. |   λ    |  Lower Bound  |   Upper Bound  | Gap")
    println("-"^59)

    for t in 1:iterations
        # Compute primal solution at current λ
        x[1] = (1 - 2λ > 0) ? 1 : 0
        x[2] = (2 - λ > 0) ? 1 : 0

        # Lagrangian dual value
```

```
54          zL = 2λ + max(0, 1 - 2λ) + max(0, 2 - λ)
55
56          # Check for improvement in UB
57          if zL < UB
58              UB = zL
59              λ_best = λ              # Update the best λ found
60              no_improve_counter = 0  # Reset counter if we have an
improvement
61          else
62              no_improve_counter += 1
63              # Check if we have reached the limit of no improvement
64              if no_improve_counter >= no_improve_limit
65                  println("-"^59)
66                  println("Lagrangian relaxation stopped. ",
67                          "No improvement in UB for $no_improve_limit
iterations.")
68                  break
69              end
70          end
71
72          # Feasible primal value
73          if 2x[1] + x[2] <= 2
74              zP = x[1] + 2x[2]
75              LB = max(LB, zP)
76          end
77
78          # Store LB and UB values for plotting
79          push!(LB_history, LB)
80          push!(UB_history, UB)
81
82          # Print iteration details
83          @printf("%5d | %5.4f | %13.4f | %14.4f | %9.3f\n", t, λ,
LB, UB, UB - LB)
84
85          # Subgradient
86          s = 2 - (2x[1] + x[2])
87
88          # Update step size (Diminishing step size)
89          α = 1 / (1 + t)
90
91          # Update λ (project onto λ ≥ 0)
```

```
        λ = max(0, λ - α * s)
    end
    return UB, λ_best, LB_history, UB_history
end

# Solve using Lagrangian relaxation
UB, λ_best, LB_history, UB_history = solve_lagrangian()

# Solve exactly to find the optimal value
x0, z0 = solve_exact()

println("------------------------- Results
-------------------------")
@printf("Optimal Dual Solution (λ = %4.2f): %4.2f\n", λ_best, UB)
@printf("Optimal Primal Solution: x = [%d, %d] with objective value
%4.2f\n", x0[1], x0[2], z0)
@printf("Duality gap: %4.2f - %4.2f = %4.2f\n", UB, z0, UB - z0)

# Plot convergence of LB and UB
plot_convergence("Lagrangian Relaxation Convergence",LB_history,
UB_history)
```

Listing B.68: Julia code to implement the subgradient method for the Lagrangian Dual Problem.

Listing B.69 presents the output of the Lagrangian relaxation for the example problem defined in Equation (B.13). The output includes the iteration number, current value of the Lagrange multiplier (λ), lower bound, upper bound, and the gap between the upper and lower bounds. The process continues until no improvement in the upper bound is observed for a specified number of iterations, at which point the algorithm stops. The final results include the optimal dual solution, optimal primal solution, and the duality gap.

Output

```
-------------- Starting Lagrangian Relaxation -------------
Iter. |   λ    |  Lower Bound  |   Upper Bound  |   Gap
------------------------------------------------------------
    1 | 0.0000 |          -Inf |         3.0000 |       Inf
    2 | 0.5000 |        2.0000 |         2.5000 |     0.500
    3 | 0.1667 |        2.0000 |         2.5000 |     0.500
    4 | 0.4167 |        2.0000 |         2.5000 |     0.500
```

```
    5 | 0.6167 |        2.0000 |         2.5000 |     0.500
    6 | 0.4500 |        2.0000 |         2.5000 |     0.500
    7 | 0.5929 |        2.0000 |         2.5000 |     0.500
    8 | 0.4679 |        2.0000 |         2.5000 |     0.500
    9 | 0.5790 |        2.0000 |         2.5000 |     0.500
   10 | 0.4790 |        2.0000 |         2.5000 |     0.500
   11 | 0.5699 |        2.0000 |         2.5000 |     0.500
-------------------------------------------------------------
Lagrangian relaxation stopped. No improvement in UB for 10 iterations.
------------------------- Results -------------------------
Optimal Dual Solution (λ = 0.50): 2.50
Optimal Primal Solution: x = [0, 1] with objective value 2.00
Duality gap: 2.50 - 2.00 = 0.50
```

Listing B.69: Output of the Lagrangian relaxation for the example problem.

Figure B.39 presents a plot visualizing the convergence of the lower and upper bounds obtained from the Lagrangian relaxation for the example problem defined in Equation (B.13). The plot shows how the lower bound (LB) and upper bound (UB) evolve over iterations, highlighting the point at which the algorithm stops due to no improvement in the upper bound for a specified number of iterations. The final optimal dual solution and the corresponding duality gap are also indicated on the plot.

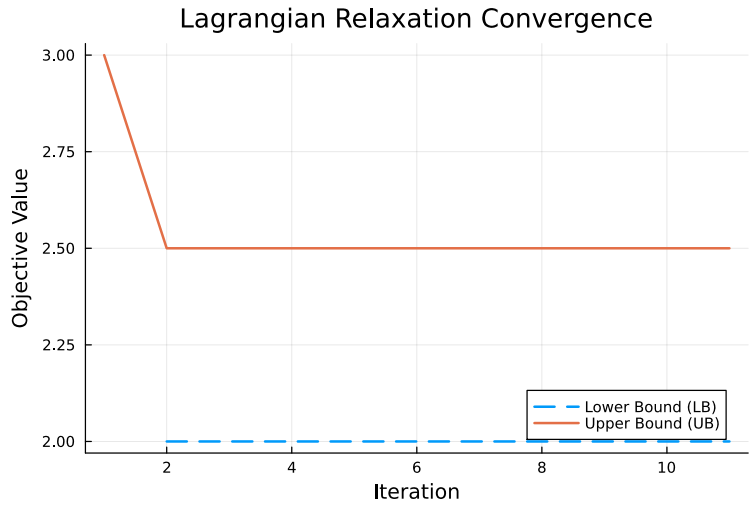


Figure B.39: Convergence of the Lagrangian relaxation for the example problem.

B.7.2 Exercise 7.2

B.7.2.1 a) Apply the subgradient method to get a bound on the optimal solution of the P-Median Problem using $p = 3$ and dataset: `exercise_pmp.csv`

Listing B.70 presents the Julia code that implements the subgradient method to solve the P-Median Problem for the dataset `exercise_pmp.csv`. The code includes functions to load the data, compute distances, perform Lagrangian relaxation, and iterate through the subgradient method to update the Lagrange multipliers. It also tracks the lower and upper bounds for convergence analysis.

```
using JuMP          # Modeling language
using HiGHS         # Solver
using CSV           # For reading CSV files
using Distances     # Distance computations
using LinearAlgebra # For dot/norm calculations
using Printf        # For formatted output

# Utility functions for plotting bounds convergence
include("utils/lr_utils.jl")

# Data structure to hold the p-median problem data
struct PMedianData
    D::Matrix{Float64} # Distance matrix
    p::Int             # Number of facilities
    n::Int             # Number of clients/locations
    # Constructor
    function PMedianData(file_path::String; p::Int)
        # Load latitude and longitude data
        coordinates = CSV.read(file_path, CSV.Tables.matrix,
header=true)
        # Compute Haversine distance matrix
        D = pairwise(Haversine(), coordinates, dims=1)
        return new(D, p, size(D, 1))
    end
end

# Function to improve the chosen facilities using local search
function improve_facilities(D, chosen)
    n = size(D,1)
    # Initialize current solution and cost
```

```
    current = copy(chosen)
    current_cost = sum(minimum(D[:, current], dims=2))
    # Main loop for local search
    improved = true
    while improved
        improved = false
        best_cost = current_cost
        best_set = current
        # Create a set for quick membership checking
        current_set = Set(current)
        # Iterate over all possible swaps
        for facility_out in current
            for facility_in in 1:n
                # Skip if the facility is already in the current set
                if facility_in in current_set
                    continue
                end
                candidate = copy(current)
                # Perform the swap
                idx = findfirst(==(facility_out), candidate)
                candidate[idx] = facility_in
                # Calculate the cost of the candidate solution
                candidate_cost = sum(minimum(D[:, candidate], dims=2))
                # Check if this candidate solution is better than the best found so far
                if candidate_cost < best_cost
                    best_cost = candidate_cost
                    best_set = candidate
                    improved = true
                end
            end
        end
        # Update current solution if an improvement was found
        if improved
            current = best_set
            current_cost = best_cost
        end
    end
    return current, current_cost
```

```
end

# Function to solve the PMP exactly
function solve_pmp_exact(data::PMedianData)
    # Index range
    N = 1:data.n
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[i in N, j in N], Bin)
    # Objective: Minimize total distance
    @objective(model, Min, dot(data.D, x))
    # Constraint: Exactly p facilities opened
    @constraint(model, sum(x[j,j] for j in N) == data.p)
    # Constraint: Each customer assigned to one facility
    @constraint(model, [i in N], sum(x[i,j] for j in N) == 1)
    # Constraint: Assignment only to open facilities
    @constraint(model, [i in N, j in N], x[i,j] <= x[j,j])
    # Run the solver
    JuMP.optimize!(model)
    # Return the optimal value
    return JuMP.objective_value(model)
end

# Function to solve PMP using Lagrangian Relaxation
function solve_pmp_lagrangian(file_path; p = 3, iteration_max=500,
gap_tolerance = 1e-6, print_feedback_iteration = 10,
local_search_iteration = 10)
    # Load problem data
    data = PMedianData(file_path, p = p)

    # Initialization
    λ = zeros(data.n)
    θ, LB, UB = 2.0, -Inf, Inf
    LB_history, UB_history = Float64[], Float64[]
    no_improve, limit = 0, 10

    println("-"^11," Starting Lagrangian Relaxation ", "-"^11)
```

```
106     println("Iter.  |   Lower Bound   |   Upper Bound   |   Gap
(%)")
107     println("-"^54)
108 
109     # Main loop
110     for iteration in 1:iteration_max
111         # --- Step 1: Lower Bound ---
112 
113         # 1a. Calculate attractiveness of each facility
114         costs = data.D .- λ
115         facility_scores = vec(sum(min.(0.0, costs), dims=1))
116 
117         # 1b. Select the p best facilities (smallest value)
118         chosen = sortperm(facility_scores)[1:data.p]
119 
120         # 1c. Calculate the LB
121         current_LB = sum(facility_scores[chosen]) + sum(λ)
122 
123         # 1d. Update Best LB
124         if current_LB > LB
125             LB, no_improve = current_LB, 0
126         else
127             no_improve += 1
128             if no_improve >= limit
129                 θ = max(0.5, θ * 0.95)
130                 no_improve = 0
131             end
132         end
133 
134         # 1e. Store LB history
135         push!(LB_history, LB)
136 
137         # --- Step 2: Upper Bound (Heuristic) ---
138 
139         # 2a. Two strategies: every `local_search_iteration`
iterations, improve the facilities;
140         #     otherwise, assign each customer to the closest
facility
141         if iteration % local_search_iteration == 0
142             # Local search to improve the chosen facilities
```

```
            improved_chosen, current_UB = improve_facilities(data.D, chosen)
        else
            # For each customer, assign to the closest facility
            current_UB = sum(minimum(data.D[:, chosen], dims=2))
        end

        # 2b. Update the best upper bound if this heuristic solution is better
        UB = min(UB, current_UB)

        # 2c. Store upper bound history
        push!(UB_history, UB)

        # 2d. Stopping Criterion
        gap = isfinite(UB) ? abs(UB - LB) / (UB + 1e-10) : Inf
        if isfinite(gap) && gap <= gap_tolerance
            @printf("%6d | %15.3f | %15.3f | %9.3f\n", iteration, LB, UB, 100 * gap)
            println("-"^54)
            println("Convergence achieved: Duality gap is below tolerance.")
            break
        end

        # --- Step 3: Update Multipliers ---
        # s_i = 1 - (times the client i was served in the relaxed problem)

        # 3a. Calculate subgradient
        x_relaxed = (costs[:, chosen] .< 0)
        # x_relaxed = (costs[:, improved_chosen] .< 0)
        s = 1.0 .- vec(sum(x_relaxed, dims=2))

        # 3b. Determine the step size α_k (Held-Karp style)
        α = θ * (abs(UB - LB) / (norm(s)^2 + 1e-10))

        # 3c. Update the multipliers
        λ .+= α .* s

        # Print iteration details
```

```
        if iteration == 1 || iteration % print_feedback_iteration
== 0
            @printf("%6d | %15.3f | %15.3f | %9.3f\n", iteration,
LB, UB, 100 * gap)
        end
    end

    # Final summary and Exact solve for comparison
    println("-"^54)
    println("Solving exact PMP for comparison...")
    exact_objective = solve_pmp_exact(data)
    @printf("Final LB: %.3f | Exact: %.3f | Duality Gap: %.3f
(%.3f%%)\n", LB, exact_objective, abs(exact_objective - LB), 100 *
abs(exact_objective - LB)/exact_objective)

    # Plot convergence history
    plot_convergence("Lagrangian Relaxation\nConvergence for PMP",
LB_history, UB_history)
end

# Example usage
solve_pmp_lagrangian("data/exercise_pmp.csv", p=3, iteration_max =
500, gap_tolerance = 1e-4, print_feedback_iteration = 10,
local_search_iteration = 10)
```

Listing B.70: Julia code to implement the subgradient method to solve the P-Median Problem for dataset `exercise_pmp.csv`.

Listing B.71 presents the output of the Lagrangian relaxation for the P-Median Problem using the dataset `exercise_pmp.csv`. The output includes the iteration number, lower bound, upper bound, and the gap percentage between the upper and lower bounds. The process continues until convergence is achieved, indicated by the duality gap falling below a specified tolerance. The final results include the best lower bound, exact solution, and the duality gap.

Output

```
----------- Starting Lagrangian Relaxation -----------
Iter.   |   Lower Bound   |   Upper Bound   |   Gap (%)
-----------------------------------------------------
     1 |           0.000 |     1105032.279 |   100.000
    10 |      364622.007 |      499653.789 |    27.025
```

```
    20 |      378322.099 |      499653.789 |    24.283
    30 |      475081.593 |      499653.789 |     4.918
    40 |      496400.884 |      499653.789 |     0.651
    50 |      499525.953 |      499653.789 |     0.026
    58 |      499607.511 |      499653.789 |     0.009
-----------------------------------------------------
Convergence achieved: Duality gap is below tolerance.
-----------------------------------------------------
Solving exact PMP for comparison...
Final LB: 499607.511 | Exact: 499653.789 | Duality Gap: 46.278 (0.009%)
```

Listing B.71: Output of the Lagrangian relaxation for the P-Median Problem for dataset `exercise_pmp.csv`.

Figure B.40 presents a plot visualizing the convergence of the lower and upper bounds obtained from the Lagrangian relaxation for the P-Median Problem using the dataset `exercise_pmp.csv`. The plot shows how the lower bound (LB) and upper bound (UB) evolve over iterations, highlighting the point at which convergence is achieved, indicated by the duality gap falling below a specified tolerance. The final best lower bound and exact solution are also indicated on the plot.

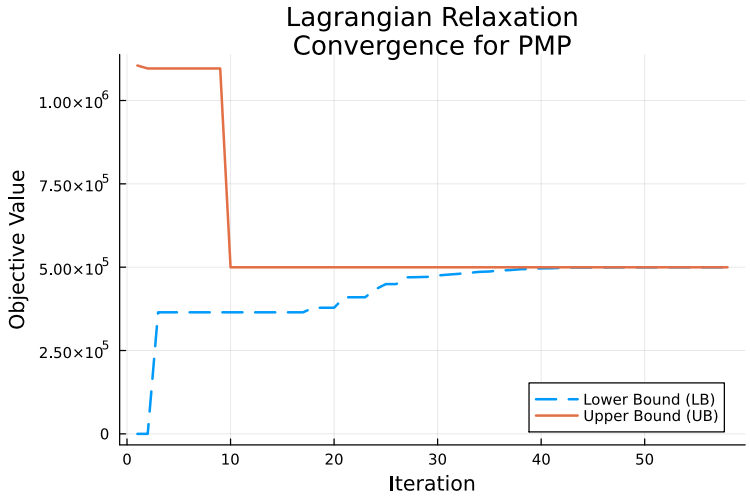


Figure B.40: Convergence of the Lagrangian relaxation for the P-Median Problem for dataset `exercise_pmp.csv`.

B.7.2.2 b) Apply the subgradient method to get a bound on the optimal solution of the fault-tolerant P-Median Problem [36] using $p = 5$, $r = 2$ and dataset: `exercise_pmp.csv`

Given the following formulation of the Fault-tolerant P-Median Problem:

$$\text{FT-PMP}: \quad \min_x \quad \sum_{i\in N}\sum_{j\in N} d_{ij}x_{ij} \tag{B.18.1}$$

$$\text{s.t.} \quad \sum_{j\in N} x_{jj} = p \tag{B.18.2}$$

$$\sum_{j\in N} x_{ij} = r \qquad \forall i \in N \tag{B.18.3}$$

$$x_{ij} \leq x_{jj} \qquad \forall i,j \in N \tag{B.18.4}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N \tag{B.18.5}$$

where N is the set of customers/facilities, d_{ij} is the distance from customer i to facility j, and r is the number of facilities that each customer is assigned to.

We relax constraint Equation (B.18.3) using Lagrangian multipliers $\lambda_i \in \mathbb{R}$ for each customer i. The resulting Lagrangian relaxation is:

$$L(\lambda): \quad \min_x \sum_{i\in N}\sum_{j\in N}(d_{ij} - \lambda_i)x_{ij} + r\sum_{i\in N}\lambda_i \tag{B.19.1}$$

$$\text{s.t.} \quad \sum_{j\in N} x_{jj} = p \tag{B.19.2}$$

$$x_{ij} \leq x_{jj} \qquad \forall i,j \in N \tag{B.19.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N \tag{B.19.4}$$

The subgradient for updating the Lagrangian multipliers is given by:

$$s_i = r - \sum_{j\in N} x_{ij} \tag{B.20}$$

Interpretation of the subgradient:

- If $s_i > 0$, it means that customer i is currently assigned to fewer than r facilities, so we need to increase λ_i to encourage more assignments for that customer.
- If $s_i < 0$, it means that customer i is currently assigned to more than r facilities, so we need to decrease λ_i to discourage excess assignments for that customer.

- If $s_i = 0$, it means that customer i is currently assigned to exactly r facilities, so we do not need to adjust λ_i for that customer.

The following code solves the Lagrangian relaxation of the Fault-tolerant P-Median Problem.

Listing B.72 presents the Julia code that implements the subgradient method to solve the Fault-tolerant P-Median Problem for the dataset `exercise_pmp.csv`. The code includes functions to load the data, compute distances, perform Lagrangian relaxation, and iterate through the subgradient method to update the Lagrange multipliers. It also tracks the lower and upper bounds for convergence analysis.

```
using JuMP          # Modeling language
using HiGHS         # Solver
using CSV           # For reading CSV files
using Distances     # Distance computations
using LinearAlgebra # For norm calculations
using Printf        # For formatted output

# Utility functions for plotting bounds convergence
include("utils/lr_utils.jl")

# Data structure to hold the Fault-tolerant P-Median Problem data
struct FTPMedianData
    D::Matrix{Float64} # Distance matrix
    p::Int             # Number of facilities
    r::Int             # Fault tolerance level
    n::Int             # Number of clients/locations
    # Constructor
    function FTPMedianData(file_path::String; p::Int, r::Int)
        # Load latitude and longitude data
        coordinates = CSV.read(file_path, CSV.Tables.matrix,
header=true)
        # Compute Haversine distance matrix
        D = pairwise(Haversine(), coordinates, dims=1)
        return new(D, p, r, size(D, 1))
    end
end

# Function to solve the FT-PMP exactly
function solve_ft_pmp_exact(data::FTPMedianData)
    # Index range
```

```
    N = 1:data.n
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[i in N, j in N], Bin)
    # Objective: Minimize total distance
    @objective(model, Min, dot(data.D, x))
    # Constraint: Each customer assigned to r facilities
    @constraint(model, [i in N], sum(x[i,j] for j in N) == data.r)
    # Constraint: Assignment only to open facilities
    @constraint(model, [i in N, j in N], x[i,j] <= x[j,j])
    # Constraint: Exactly p facilities opened
    @constraint(model, sum(x[j,j] for j in N) == data.p)
    # Run the solver
    JuMP.optimize!(model)
    return JuMP.objective_value(model)
end

# Function to evaluate the cost of a given solution S
function evaluate_solution(data, S)
    cost = 0.0
    for i in 1:data.n
        dists = data.D[i, collect(S)]
        cost += sum(partialsort(dists, 1:data.r))
    end
    return cost
end

# Function to perform local search by swapping facilities
function local_search_swap(data, initial_open)
    S = Set(initial_open)
    # Initialize current cost
    current_cost = evaluate_solution(data, S)
    # Main loop for local search
    improved = true
    while improved
        improved = false
        # Best improvement found in this iteration
        best_delta = 0.0
```

```
71          best_move = nothing
72          # Generate lists of open and closed facilities
73          open_facilities = collect(S)
74          closed_facilities = setdiff(1:data.n, open_facilities)
75          # Try swapping each open facility with each closed facility
76          for f in open_facilities
77              for g in closed_facilities
78                  # Create a new solution by swapping f with g
79                  S_new = copy(S)
80                  delete!(S_new, f)
81                  push!(S_new, g)
82                  # Evaluate the cost of the new solution
83                  new_cost = evaluate_solution(data, S_new)
84                  # Calculate the improvement (delta)
85                  delta = current_cost - new_cost
86                  # Check if this is the best improvement found so
    far
87                  if delta > best_delta + 1e-6
88                      best_delta = delta
89                      best_move = (f,g,new_cost)
90                  end
91              end
92          end
93          # If an improvement was found, update the current solution
94          if best_move !== nothing
95              f,g,new_cost = best_move
96              delete!(S,f)
97              push!(S,g)
98              current_cost = new_cost
99              improved = true
100         end
101     end
102     return collect(S), current_cost
103 end
104
105 # Main function to solve FT-PMP using Lagrangian Relaxation
106 function solve_ft_pmp_lagrangian(file_path; p = 3, r = 2,
    iteration_max = 500, gap_tolerance = 1e-6, print_feedback_iteration
    = 10, local_search_iteration = 10)
107     # Load problem data
108     data = FTPMedianData(file_path, p = p, r = r)
```

```
109
110     # Initialization
111     λ = zeros(data.n)
112     θ, LB, UB = 2.0, -Inf, Inf
113     LB_history, UB_history = Float64[], Float64[]
114     no_improve, limit = 0, 10
115
116     println("--- Starting Lagrangian Relaxation for Fault-tolerant
    P-Median ---")
117     println("Iter. |     Lower Bound     |     Upper Bound     |
    Gap (%)")
118     println("-"^66)
119
120     # Main loop
121     for iteration in 1:iteration_max
122         # --- Step 1: Lower Bound ---
123
124         # 1a. Calculate attractiveness of each facility
125         costs = data.D .- λ
126         facility_scores = vec(sum(min.(0.0, costs), dims=1))
127
128         # 1b. Select the p best facilities (smallest value)
129         chosen = sortperm(facility_scores)[1:data.p]
130
131         # 1c. Calculate the LB
132         current_LB = sum(facility_scores[chosen]) + r * sum(λ)
133
134         # 1d. Update Best LB
135         if current_LB > LB
136             LB = current_LB
137             no_improve = 0
138         else
139             no_improve += 1
140             if no_improve >= limit
141                 θ = max(0.5, θ * 0.5)
142                 no_improve = 0
143             end
144         end
145
146         # 1e. Store LB history
147         push!(LB_history, LB)
```

```
148
149         # --- Step 2: Upper Bound (Heuristic) ---
150
151         # 2a. Local search every `local_search_iteration`
    iterations, otherwise assign each customer to the "r" closest
    facilities
152         if iteration % local_search_iteration == 0
153             # Local search to improve the chosen facilities
154             improved_chosen, current_UB = local_search_swap(data,
    chosen)
155         else
156             # For each customer, assign them to the "r" closest
    facilities
157             current_UB = sum(sum(partialsort(data.D[i, chosen],
    1:data.r)) for i in 1:data.n)
158         end
159
160         # 2b. Update the best upper bound if this heuristic
    solution is better
161         UB = min(UB, current_UB)
162
163         # 2c. Store upper bound history
164         push!(UB_history, UB)
165
166         # 2d. Check convergence
167         gap = isfinite(UB) ? abs(UB - LB) / (UB + 1e-10) : Inf
168         if isfinite(gap) && gap <= gap_tolerance
169             @printf("%5d | %19.3f | %19.3f | %9.3f\n", iteration,
    LB, UB, 100 * gap)
170             println("-"^66)
171             println("Convergence achieved: Duality gap is below
    tolerance.")
172             break
173         end
174
175         # --- 3. Subgradient update ---
176
177         # 3a. Calculate subgradient
178         x_relaxed = (costs[:, chosen] .< 0)
179         s = data.r .- vec(sum(x_relaxed, dims=2))
180
181         # 3b. Determine the step size α_k (Held-Karp style)
```

```
182         α = θ * abs(UB - LB) / (norm(s)^2 + 1e-10)
183
184         # 3c. Update the multipliers
185         λ .+= α .* s
186
187         # Print iteration results
188         if iteration == 1 || iteration % print_feedback_iteration
    == 1
189             @printf("%5d | %19.3f | %19.3f | %9.3f\n", iteration,
    LB, UB, 100 * gap)
190         end
191     end
192
193     # Final summary and Exact solve for comparison
194     println("-"^66)
195     println("Solving exact FT-PMP for comparison...")
196     exact_objective = solve_ft_pmp_exact(data)
197     @printf("Final LB: %8.3f | Exact: %8.3f | Duality Gap: %5.3f
    (%5.3f%%)\n", LB, exact_objective, abs(exact_objective - LB), 100 *
    abs(exact_objective - LB)/exact_objective)
198
199     # Plot convergence
200     plot_convergence("Lagrangian Relaxation\nConvergence for FT-
    PMP", LB_history, UB_history)
201 end
202
203 # Example usage
204 solve_ft_pmp_lagrangian("data/exercise_pmp.csv", p = 5, r = 2,
    iteration_max = 500, gap_tolerance = 1e-4, print_feedback_iteration
    = 20, local_search_iteration = 10)
```

Listing B.72: Julia code to implement the subgradient method to solve the Fault-tolerant P-Median Problem for dataset `exercise_pmp.csv`.

Listing B.73 presents the output of the Lagrangian relaxation for the Fault-tolerant P-Median Problem using the dataset `exercise_pmp.csv`. The output includes the iteration number, lower bound, upper bound, and the gap percentage between the upper and lower bounds. The process continues until convergence is achieved, indicated by the duality gap falling below a specified tolerance. The final results include the best lower bound, exact solution, and the duality gap.

Output

```
--- Starting Lagrangian Relaxation for Fault-tolerant P-Median ---
Iter. |     Lower Bound     |     Upper Bound     |     Gap (%)
------------------------------------------------------------------
    1 |               0.000 |         2199112.825 |   100.000
   21 |         1013421.779 |         1184550.274 |    14.447
   41 |         1169956.922 |         1184550.274 |     1.232
   61 |         1184028.228 |         1184550.274 |     0.044
   81 |         1184341.234 |         1184550.274 |     0.018
  101 |         1184406.542 |         1184550.274 |     0.012
  121 |         1184426.491 |         1184550.274 |     0.010
  126 |         1184438.234 |         1184550.274 |     0.009
------------------------------------------------------------------
Convergence achieved: Duality gap is below tolerance.
------------------------------------------------------------------
Solving exact FT-PMP for comparison...
Final LB: 1184438.234 | Exact: 1184550.274 | Duality Gap: 112.040
(0.009%)
```

Listing B.73: Output of the Lagrangian Relaxation for Fault-tolerant P-Median Problem.

Figure B.41 presents a plot visualizing the convergence of the lower and upper bounds obtained from the Lagrangian relaxation for the Fault-tolerant P-Median Problem using the dataset `exercise_pmp.csv`.

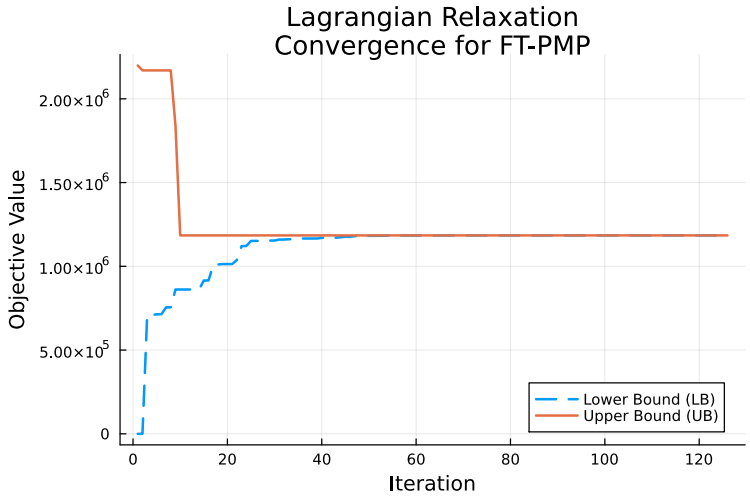


Figure B.41: Plot showing the convergence of lower and upper bounds for the Fault-tolerant P-Median Problem.

B.7.2.3 c) Apply the subgradient method to get a bound on the optimal solution of the Capacitated P-Median Problem using p = 3 and dataset `exercise_cpmp.csv`

Given the following formulation of the Capacitated P-Median Problem (CPMP):

$$\text{CPMP}: \quad \min_{x} \quad \sum_{i\in N}\sum_{j\in N} d_{ij}x_{ij} \tag{B.21.1}$$

$$\text{s.t.} \quad \sum_{j\in N} x_{jj} = p \tag{B.21.2}$$

$$\sum_{j\in N} x_{ij} = 1 \qquad \forall i \in N \tag{B.21.3}$$

$$x_{ij} \leq x_{jj} \qquad \forall i,j \in N \tag{B.21.4}$$

$$\sum_{i\in N} c_i x_{ij} \leq C x_{jj} \qquad \forall j \in N \tag{B.21.5}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N \tag{B.21.6}$$

where N is the set of customers/facilities, d_{ij} is the distance from customer i to facility j, c_i is the demand of customer i, and C is the capacity of facility j.

We relax the assignment constraints in Equation (B.21.3) using Lagrange multipliers λ_i for each customer i. The resulting Lagrangian relaxation, rearranged to group by facility j, is:

$$\text{L}(\lambda): \quad \min_{x} \sum_{j\in N}\left[\sum_{i\in N}(d_{ij}-\lambda_i)x_{ij}\right] + \sum_{i\in N}\lambda_i \tag{B.22.1}$$

$$\text{s.t.} \quad \sum_{j\in N} x_{jj} = p \tag{B.22.2}$$

$$x_{ij} \leq x_{jj} \qquad \forall i,j \in N \tag{B.22.3}$$

$$\sum_{i\in N} c_i x_{ij} \leq C x_{jj} \qquad \forall j \in N \tag{B.22.4}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in N \tag{B.22.5}$$

The problem now decomposes into $|N|$ independent subproblems, one for each facility j. Each subproblem is a $0/1$ knapsack problem that determines whether to open facility j and which customers to assign to it, based on the modified costs $(d_{ij} - \lambda_i)$.

Listing B.74 presents the Julia code that implements the subgradient method to solve the Capacitated P-Median Problem for the dataset `exercise_cpmp.csv`. The code includes functions to load the data, compute distances, perform Lagrangian

relaxation, and iterate through the subgradient method to update the Lagrange multipliers. It also tracks the lower and upper bounds for convergence analysis.

```
using JuMP          # Modeling language
using HiGHS         # Solver
using LinearAlgebra # For norm calculations
using CSV           # For reading CSV files
using DataFrames    # For handling DataFrame operations
using Distances     # Distance computations
using Printf        # For formatted output

# Utility functions for plotting bounds convergence
include("utils/lr_utils.jl")

# Data structure to hold the CPMP data
struct CPMPData
    D::Matrix{Float64}       # Distance matrix
    p::Int                   # Number of facilities
    demands::Vector{Float64} # Demand of each client
    C::Float64               # Capacity of each facility
    n::Int                   # Number of clients/locations
    # Constructor
    function CPMPData(file_path::String; p::Int, C::Float64)
        # Load latitude and longitude data
        file = CSV.read(file_path, DataFrame)
        coordinates = Matrix(file[:, [:Latitude, :Longitude]])
        demands = Vector(file[:, :Demand])
        # Compute Haversine distance matrix
        D = pairwise(Haversine(), coordinates, dims=1)
        return new(D, p, demands, C, size(D, 1))
    end
end

# Function to solve the p-median problem exactly
function solve_cpmedian_exact(data::CPMPData)
    N = 1:data.n
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[i in N, j in N], Bin)
```

```
    # Objective: Minimize total distance
    @objective(model, Min, dot(data.D, x))
    # Constraint: Each customer assigned to one facility
    @constraint(model, [i in N], sum(x[i,j] for j in N) == 1)
    # Constraint: Assignment only to open facilities
    @constraint(model, [i in N, j in N], x[i,j] <= x[j,j])
    # Constraint: Exactly p facilities opened
    @constraint(model, sum(x[j,j] for j in N) == data.p)
    # Constraint: Capacity limit for each facility
    @constraint(model, [j in N], sum(data.demands[i] * x[i, j] for
i in N) <= data.C * x[j,j])
    # Run the solver
    JuMP.optimize!(model)
    # Return the optimal value
    if termination_status(model) == JuMP.OPTIMAL
        return JuMP.objective_value(model)
    else
        error("Optimal solution not found.")
    end
end

# Function to solve subproblem for a given facility j in the
relaxed problem
function solve_subproblem_j(modified_costs, data::CPMPData)
    n = size(modified_costs, 1)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, y[1:n], Bin)
    # Objective: Minimize total modified cost for facility j
    @objective(model, Min, sum(modified_costs[i] * y[i] for i in
1:n))
    # Constraint: Total demand assigned to facility j cannot exceed
capacity
    @constraint(model, sum(data.demands[i] * y[i] for i in 1:n) <=
data.C)
    # Run the solver
    JuMP.optimize!(model)
    # Return the optimal value
    if termination_status(model) == JuMP.OPTIMAL
```

```
77          return JuMP.objective_value(model), [JuMP.value(y[i])>0.5 ?
    1 : 0 for i in 1:n]
78      end
79  end
80
81  # Function to solve subproblem for a given a set of facilities from
    the relaxed problem
82  function solve_cpmedian_exact_subset(selected_facilities,
    data::CPMPData)
83      N = 1:data.n
84      M = 1:length(selected_facilities)
85      # Create the model
86      model = JuMP.Model(HiGHS.Optimizer)
87      # Silent mode (solver output is not printed)
88      JuMP.set_silent(model)
89      # Variables
90      @variable(model, x[i in N, j in M], Bin)
91      # Objective: Minimize total distance
92      @objective(model, Min, sum(data.D[i, selected_facilities[j]] *
    x[i,j] for i in N, j in M))
93      # Constraint: Each customer assigned to one facility
94      @constraint(model, [i in N], sum(x[i,j] for j in M) == 1)
95      # Constraint: Capacity limit for each facility
96      @constraint(model, [j in M], sum(data.demands[i] * x[i, j] for
    i in N) <= data.C)
97      # Run the solver
98      JuMP.optimize!(model)
99      # Return the optimal value
100     if termination_status(model) == JuMP.OPTIMAL
101         return JuMP.objective_value(model),
102                 [JuMP.value(x[i,j])>0.5 ? selected_facilities[j] : 0
    for i in N, j in M]
103     else
104         error("Optimal solution for subset not found.")
105     end
106 end
107
108 # Main Lagrangian Relaxation function
109 function solve_cpmedian_lagrangian(file_path; p = 3, capacity =
    3516, iterations = 50, gap_tolerance = 1e-6,
    print_feedback_iteration = 20)
110     # Load problem data
```

```
111     data = CPMPData(file_path, p = p, C = capacity)
112 
113     # Initialization
114     λ = zeros(data.n)
115     θ, LB, UB = 0.15, -Inf, Inf
116     LB_history, UB_history = Float64[], Float64[]
117     no_improve, limit = 0, 10
118     best_feasible = nothing # Store best feasible solution found
119 
120     # Counter logic for robust convergence: reduce θ after 3
    iterations without improvement in LB
121     no_improve_limit = 3
122     no_improve_counter = 0
123 
124     println("--- Starting Lagrangian Relaxation for Capacitated P-
    Median ---")
125     println("Iter. |     Lower Bound     |     Upper Bound     |
    Gap (%)")
126     println("-"^63)
127 
128     # Main loop
129     for iteration in 1:iterations
130         # ---- 1. Solve the relaxed problem to get a lower bound
131 
132         # 1a. Modify costs with current λ
133         modified_costs = data.D .- λ
134 
135         # 1b. Solve subproblems for each facility
136         facility_costs = zeros(data.n)
137         facility_assignments = zeros(Int, data.n, data.n)
138         for facility_id in 1:data.n
139             facility_costs[facility_id],
140             facility_assignments[facility_id,:] =
    solve_subproblem_j(modified_costs[:, facility_id], data)
141         end
142 
143         # 1c. Select p facilities with lowest modified cost
144         chosen_facilities = sortperm(facility_costs)[1:p]
145 
146         # 1d. Calculate the relaxed solution
147         x_relaxed =
    vec(sum(facility_assignments[chosen_facilities, :], dims=1))
```

```
148
149         # 1e. Calculate Lagrangian Lower Bound
150         current_LB = sum(λ) +
sum(facility_costs[chosen_facilities])
151
152         # 1f. Update LB and step size factor
153         if current_LB > LB
154             LB = current_LB
155             no_improve_counter = 0
156         else
157             no_improve_counter += 1
158             if no_improve_counter >= no_improve_limit
159                 θ *= 0.95
160                 # θ = max(0.5, θ * 0.5)
161                 no_improve_counter = 0
162             end
163         end
164
165         # 1g. Store LB history
166         push!(LB_history, LB)
167
168         # ---- 2. Heuristic to find a feasible solution and get an
upper bound
169
170         # 2a. Repair heuristic based on Lagrangian regret
171         current_UB, best_feasible =
solve_cpmedian_exact_subset(chosen_facilities, data)
172
173         # 2b. Update UB
174         UB = min(UB, current_UB)
175
176         # 2c. Store UB history
177         push!(UB_history, UB)
178
179         # 2d. Check convergence
180         gap = isfinite(UB) ? abs(UB - LB) / (UB + 1e-10) : Inf
181         if isfinite(gap) && gap <= gap_tolerance
182             @printf("%5d | %19.3f | %19.3f | %9.3f\n", iteration,
LB, UB, 100 * gap)
183             println("-"^66)
184             println("Convergence achieved: Duality gap is below
tolerance.")
```

```
            break
        end

        # ---- 3. Subgradient update

        # 3. Calculate subgradient
        s = ones(data.n) - x_relaxed

        # 3a. Calculate step size
        α = θ * (UB - LB) / (norm(s)^2 + 1e-10)

        # 3b. Update Lagrange multipliers
        λ .+= α .* s
        λ .= max.(0.0, λ) # Ensure non-negativity

        # Print iteration results
        if iteration == 1 || iteration % print_feedback_iteration
== 1
            @printf("%5d | %14.4f | %14.4f | %9.3f%%\n", iteration,
LB, UB, 100 * gap)
        end
    end

    # Final summary and Exact solve for comparison
    exact_objective = solve_cpmedian_exact(data)
    println("-"^63)
    println("Solving exact CPMP for comparison...")
    @printf("Final LB: %8.3f | Exact: %8.3f | Duality Gap: %8.3f
(%5.3f%%)\n", LB, exact_objective, (exact_objective - LB),
((exact_objective - LB) / exact_objective) * 100)

    # Plot convergence
    plot_convergence("Lagrangian Relaxation\nConvergence for CPMP",
LB_history, UB_history)
end

# Example usage
solve_cpmedian_lagrangian("data/exercise_cpmp.csv", p = 3, capacity
= 3516.0, iterations = 150, gap_tolerance = 1e-4,
print_feedback_iteration = 20)
```

Listing B.74: Julia code to implement the subgradient method to solve the Capacitated P-Median Problem for dataset `exercise_cpmp.csv`.

Listing B.75 presents the output of the Lagrangian relaxation for the Capacitated P-Median Problem using the dataset `exercise_cpmp.csv`. The output includes the iteration number, lower bound, upper bound, and the gap percentage between the upper and lower bounds. The process continues until convergence is achieved, indicated by the duality gap falling below a specified tolerance. The final results include the best lower bound, exact solution, and the duality gap.

Output

```
--- Starting Lagrangian Relaxation for Capacitated P-Median ---
Iter. |     Lower Bound     |     Upper Bound     |  Gap (%)
------------------------------------------------------------------
    1 |          0.0000 |    126816.6489 |   100.000%
   21 |     64025.3436 |     83545.6408 |    23.365%
   41 |     67737.2846 |     83545.6408 |    18.922%
   61 |     70273.1421 |     83545.6408 |    15.887%
   81 |     71566.4974 |     80855.7658 |    11.489%
  101 |     72108.9813 |     80744.6979 |    10.695%
  121 |     72404.8102 |     80744.6979 |    10.329%
  141 |     72651.2776 |     80744.6979 |    10.023%
------------------------------------------------------------------
Solving exact CPMP for comparison...
Final LB: 72721.408 | Exact: 77837.716 | Duality Gap: 5116.308 (6.573%)
```

Listing B.75: Output of the Lagrangian Relaxation for Capacitated P-Median Problem.

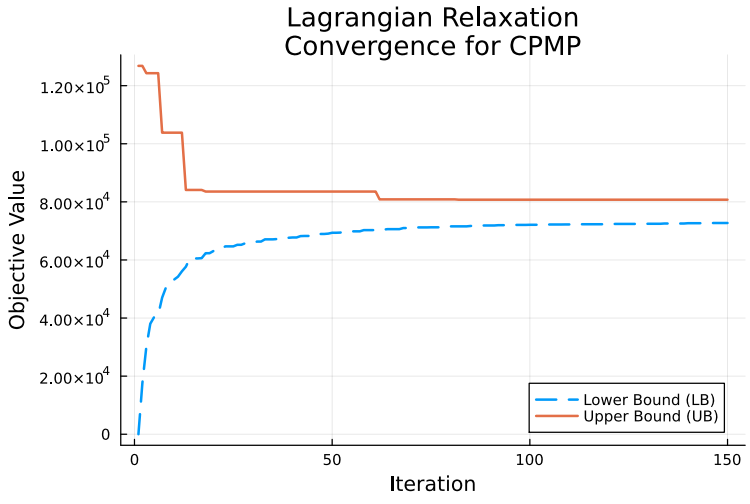


Figure B.42: Plot showing the convergence of lower and upper bounds for the Capacitated P-Median Problem.

B.7.3 Exercise 7.3

Listing B.76 presents the Julia code that implements the LagClus algorithm to solve the Maximum Independent Set Problem (MISP). The code includes functions to solve the MISP exactly, partition the graph using Metis, create a feasible solution from the Lagrangian solution, and solve the Lagrangian subproblem for each cluster. The main function orchestrates the LagClus algorithm, iterating through clusters and updating Lagrange multipliers to converge towards an optimal solution.

```
using JuMP          # For mathematical programming
using HiGHS         # HiGHS solver
using Graphs        # Graphs package
using Metis         # For graph partitioning
using LinearAlgebra # For norm calculation
using Printf        # For formatted output

# Utility functions for plotting bounds convergence
include("utils/lr_utils.jl")

# Function to solve MISP exactly
function solve_misp_exact(graph)
    n = Graphs.nv(graph)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Variables
    @variable(model, x[1:n], Bin)
    # Objective: Maximize sum of x[i]
    @objective(model, Max, sum(x[i] for i in 1:n))
    # Constraints: x[u] + x[v] <= 1 for each edge (u,v)
    for e in Graphs.edges(graph)
        u = Graphs.src(e)
        v = Graphs.dst(e)
        @constraint(model, x[u] + x[v] <= 1)
    end
    # Run the solver
```

```
    JuMP.optimize!(model)
    # Return the optimal value
    if termination_status(model) == JuMP.OPTIMAL
        return JuMP.objective_value(model)
    else
        error("Optimal solution not found.")
    end
end

# Function to partition graph using Metis
function metis_partition(graph, k)
    partition = Metis.partition(graph, k)
    clusters = [Int[] for _ in 1:k]
    for v in 1:Graphs.nv(graph)
        push!(clusters[partition[v]], v)
    end
    cut_edges = []
    for (e_id, e) in enumerate(Graphs.edges(graph))
        u = Graphs.src(e)
        v = Graphs.dst(e)
        if partition[u] != partition[v]
            push!(cut_edges, (u,v))
        end
    end
    return clusters, cut_edges
end

# Function to create a feasible solution from Lagrangian solution
function create_feasible_solution(graph, lag_solution)
    n = Graphs.nv(graph)
    feasible_solution = copy(lag_solution)
    # Greedily remove conflicts
    for u in 1:n
        if feasible_solution[u] == 1
            for v in Graphs.neighbors(graph, u)
                if feasible_solution[v] == 1
                    # Break tie by degree (drop node with higher
degree)
                    if Graphs.degree(graph, u) ≥
Graphs.degree(graph, v)
                        feasible_solution[u] = 0
```

```
                    else
                        feasible_solution[v] = 0
                    end
                end
            end
        end
    end
    return feasible_solution
end

# Function to solve Lagrangian subproblem for a cluster
function solve_lagrangian_misp(graph, cluster, λ, solution)

    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)

    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)

    # Variables
    @variable(model, x[cluster], Bin)

    # Objective: Maximize sum of x[i]
    obj = sum(x[i] for i in cluster)

    # Add λ terms to objective and constraints
    for u in cluster
        for v in Graphs.neighbors(graph, u)
            if v in cluster
                if u < v
                    @constraint(model, x[u] + x[v] <= 1)
                end
            else
                if u < v
                    obj -= λ[(u,v)] * x[u]
                else
                    obj -= λ[(v,u)] * x[u]
                end
            end
        end
    end
```

```
109     @objective(model, Max, obj)
110 
111     # Run the solver
112     JuMP.optimize!(model)
113 
114     # Get the values of the decision variables
115     for i in cluster
116         solution[i] = JuMP.value(x[i])
117     end
118     return JuMP.objective_value(model)
119 end
120 
121 # Main function to solve MISP using LagClus
122 function solve_lagclus_misp(graph; max_iterations = 2, k = 2,
    gap_tolerance = 1e-4, print_feedback_iteration = 10)
123 
124     # Get Metis partition
125     clusters, cut_edges = metis_partition(graph, k)
126 
127     # Initialize Lagrange multipliers for cut edges
128     λ = Dict{Tuple{Int,Int}, Float64}()
129     for (u,v) in cut_edges
130         λ[(u,v)] = 0.1
131     end
132 
133     # Initialize parameters
134     LB = -Inf # Lower bound
135     UB = Inf  # Upper bound
136     LB_history = [] # Lower bound history
137     UB_history = [] # Upper bound history
138     θ = 0.8   # A factor to update multiplier (Held-Karp)
139 
140     # Counter logic for robust convergence
141     # Halve θ if UB doesn't improve for 10 iterations
142     no_improve_limit = 10
143     no_improve_counter = 0
144 
145     println("------ Starting Optimized Lagrangian Relaxation
    ------")
146     println("Iteration  |  Lower Bound  |  Upper Bound  |   Gap
    (%)")
```

```
147     println("-"^54)
148 
149     best_feasible_solution = zeros(Graphs.nv(graph))
150 
151     for iteration in 1:max_iterations
152 
153         # Solve Lagrangian subproblems
154         lag_objective = 0.0
155         lag_solution = zeros(Graphs.nv(graph))
156         for cluster in clusters
157             lag_objective += solve_lagrangian_misp(graph, cluster,
λ, lag_solution)
158         end
159         lag_objective += sum(λ[(u,v)] for (u,v) in cut_edges)
160 
161         # Update UB (Upper bound)
162         if lag_objective < UB
163             UB = lag_objective
164             no_improve_counter = 0
165         else
166             no_improve_counter += 1
167             # Held-Karp Rule: Reduce θ if progress stalls
168             if no_improve_counter >= no_improve_limit
169                 θ *= 0.5
170                 no_improve_counter = 0
171             end
172         end
173         UB = min(lag_objective, UB)
174 
175         # Create a feasible solution by greedy repair
176         feasible_solution = create_feasible_solution(graph,
lag_solution)
177         feasible_objective = sum(feasible_solution)
178 
179         if feasible_objective > LB
180             LB = feasible_objective
181             best_feasible_solution = copy(feasible_solution)
182         end
183 
184         # Store history
185         push!(LB_history, LB)
```

```
        push!(UB_history, UB)

        # Compute gap
        gap = (UB - LB) / UB

        # Print iteration results
        if iteration == 1 || iteration % print_feedback_iteration
== 1
            @printf("%10d | %13.4f | %13.4f | %7.3f\n",
                    iteration, LB, UB, gap * 100)
        end

        # Check convergence
        if gap < gap_tolerance
            println("-"^54)
            println("Converged achieved at iteration $iteration by
gap tolerance.")
            break
        end

        # Update Lagrange multipliers
        s = []
        for (u,v) in cut_edges
            grad = lag_solution[u] + lag_solution[v] - 1
            λ[(u,v)] = max(0.0, λ[(u,v)] + θ * grad)
            push!(s, grad)
        end
        s = norm(s)

        # Check for small gradient norm
        if s < 1e-6
            println("Subgradient norm small: ", s)
            break
        end

        # Adaptive step-size update (Held-Karp)
        θ = 0.2 * ((UB - LB) / (s + 1e-8))
    end

    # Final summary and Exact solve for comparison
    println("-"^54)
```

```julia
    println("Solving exact MIS for comparison...")
    exact_objective = solve_misp_exact(graph)
    @printf("Final UB: %.4f | Exact: %.4f | Duality Gap: %.3f%%\n",
            UB, exact_objective, 100 * abs(exact_objective- UB)/exact_objective)

    # Plot convergence
    plot_convergence("LagClus Convergence for MISP", LB_history, UB_history, :topright)
end

# Load graph from CSV file
graph = Graphs.loadgraph("data/karate.csv")

# Example usage
solve_lagclus_misp(graph, max_iterations = 200, k=2, gap_tolerance = 1e-4, print_feedback_iteration = 5)
```

Listing B.76: Julia code to implement the LagClus algorithm for the Maximum Independent Set Problem (MISP).

Listing B.77 presents the output of the LagClus algorithm for the Maximum Independent Set Problem (MISP). The output includes the iteration number, lower bound, upper bound, and the gap percentage between the upper and lower bounds. The process continues until convergence is achieved, indicated by the duality gap falling below a specified tolerance. The final results include the best upper bound, exact solution, and the duality gap.

Output

```
------ Starting Optimized Lagrangian Relaxation ------
Iteration  |  Lower Bound  |  Upper Bound  |   Gap (%)
-----------------------------------------------------
         1 |       20.0000 |       20.5000 |   2.439
         6 |       20.0000 |       20.0323 |   0.161
        11 |       20.0000 |       20.0158 |   0.079
        16 |       20.0000 |       20.0077 |   0.039
        21 |       20.0000 |       20.0038 |   0.019
        26 |       20.0000 |       20.0019 |   0.009
-----------------------------------------------------
Converged achieved at iteration 26 by gap tolerance.
```

```
------------------------------------------------------
Solving exact MIS for comparison...
Final UB: 20.0019 | Exact: 20.0000 | Duality Gap: 0.009%
```

Listing B.77: Output of the LagClus algorithm for the Maximum Independent Set Problem (MISP).

Figure B.43 presents a plot visualizing the convergence of the lower and upper bounds obtained from the LagClus algorithm for the Maximum Independent Set Problem (MISP). The plot shows how the lower bound (LB) and upper bound (UB) evolve over iterations, highlighting the point at which convergence is achieved, indicated by the duality gap falling below a specified tolerance.

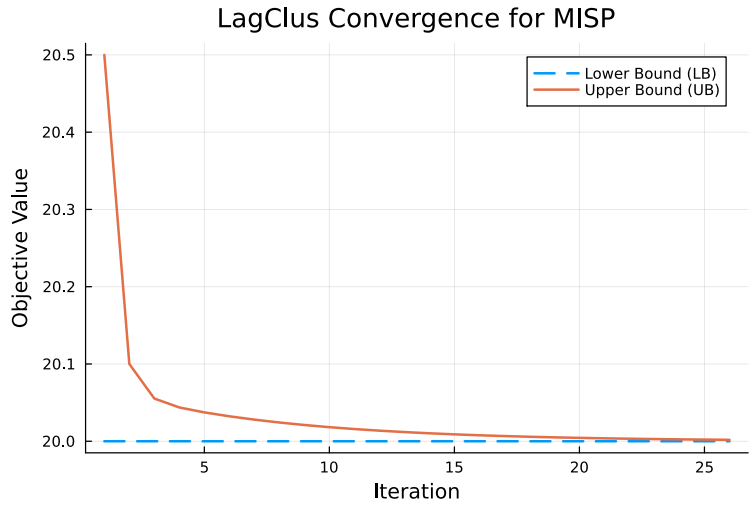


Figure B.43: Convergence of the LagClus algorithm for the Maximum Independent Set Problem (MISP).

B.7.4 Exercise 7.4

To provide a benchmark for comparison, we first solve the original problem to find the true optimal integer solution using JuMP. Then, we solve the first and second relaxation that provide lower bounds for the original problem. Listing B.78 presents the Julia code that implements the GAP Optimal Assignment Problem and its Lagrangian Relaxations. The code includes functions to solve the original integer program exactly, perform Lagrangian relaxation for both assignment and capacity constraints, and iterate through the subgradient method to update the Lagrange multipliers. It also tracks the lower bounds for convergence analysis.

```
1 using JuMP          # Modeling language
```

```
using HiGHS          # Solver
using LinearAlgebra # For norm calculations

# GAP Optimal Assignment Problem
function solve_gap_optimal(c,w,C)
    # Problem dimensions
    n, m = size(c)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define binary variables
    @variable(model, x[1:n, 1:m], Bin)
    # Objective: minimize total assignment cost
    @objective(model, Min, sum(c[i, j] * x[i, j] for i in 1:n, j in
1:m))
    # Constraint: each job must be assigned to exactly one agent
    @constraint(model, [j in 1:m], sum(x[i, j] for i in 1:n) == 1)
    # Constraint: agent capacities
    @constraint(model, [i in 1:n], sum(w[i, j] * x[i, j] for j in
1:m) <= C[i])
    # Solve the problem
    JuMP.optimize!(model)
    # Display results
    println("\nOptimal assignment and costs:")
    for i in 1:n, j in 1:m
        if JuMP.value(x[i, j]) > 0.5
            println("Job $j assigned to Agent $i (Cost = $(c[i,
j]), Workload = $(w[i, j]))")
        end
    end
    println("\nTotal Cost: ", JuMP.objective_value(model))
end

#--- Lagrangian Relaxation 1: Dualizing Assignment Constraints ---

# Solve the knapsack problem for each agent given the modified
costs
function solve_knapsack(costs, weights, capacity)
    n_items = length(costs)
    model = JuMP.Model(HiGHS.Optimizer)
```

```
    JuMP.set_silent(model)
    @variable(model, y[1:n_items], Bin)
    @objective(model, Min, dot(costs, y))
    @constraint(model, dot(weights, y) <= capacity)
    JuMP.optimize!(model)
    return JuMP.objective_value(model), JuMP.value.(y)
end

# Subgradient method for Lagrangian Relaxation 1
function subgradient_lr1(c, w, C)
    n,m = size(c)
    λ = zeros(m)
    max_iter = 200
    best_lower_bound = -Inf

    for k in 1:max_iter
        # Solve subproblems (n knapsack problems)
        total_subproblem_obj = 0
        x_sol = zeros(n, m)
        for i in 1:n
            costs_i = c[i, :] .- λ
            weights_i = w[i, :]
            obj_i, x_i = solve_knapsack(costs_i, weights_i, C[i])
            total_subproblem_obj += obj_i
            x_sol[i, :] = x_i
        end

        lower_bound = sum(λ) + total_subproblem_obj
        best_lower_bound = max(best_lower_bound, lower_bound)

        # Subgradient update
        s = 1 .- sum(x_sol, dims=1)[:]
        if norm(s) < 1e-6
            break
        end

        α = 1.0 / k # Step size
        λ .+= α .* s
    end
    return best_lower_bound
end
```

```
80
81 #--- Lagrangian Relaxation 2: Dualizing Capacity Constraints ---
82
83 # Subgradient method for Lagrangian Relaxation 2
84 function subgradient_lr2(c, w, C)
85     n,m = size(c)
86     μ = zeros(n)
87     max_iter = 200
88     best_lower_bound = -Inf
89
90     for k in 1:max_iter
91         # Solve subproblems (m simple assignment problems)
92         total_subproblem_obj = 0
93         x_sol = zeros(n, m)
94         for j in 1:m
95             costs_j = c[:, j] + μ .* w[:, j]
96             min_cost, min_idx = findmin(costs_j)
97             total_subproblem_obj += min_cost
98             x_sol[min_idx, j] = 1
99         end
100
101        lower_bound = -dot(μ, C) + total_subproblem_obj
102        best_lower_bound = max(best_lower_bound, lower_bound)
103
104        # Subgradient update
105        s = sum(w .* x_sol, dims=2)[:] - C
106        if norm(s) < 1e-6
107            break
108        end
109
110        α = 1.0 / k # Step size
111        μ = max.(0, μ + α .* s)
112    end
113    return best_lower_bound
114 end
115
116 # Cost matrix c[i, j]
117 c = [
118     8  6 10  9  7;
119     9 12  7  8 10;
120    14 10  9  7  8
```

```
]

# Workload matrix w[i, j]
w = [
    2 4 3 2 5;
    3 2 4 3 2;
    5 3 2 4 3
]

# Capacity for each agent
C = [10, 9, 11]

# Run everything
println("Solving GAP Optimal Assignment Problem:")
solve_gap_optimal(c, w, C)

println("\n--- Lagrangian Relaxation ---")
lr1_bound = subgradient_lr1(c, w, C)
println("Lower Bound from Relaxation 1: ", lr1_bound)

lr2_bound = subgradient_lr2(c, w, C)
println("Lower Bound from Relaxation 2: ", lr2_bound)
```

Listing B.78: Julia code to implement the GAP Optimal Assignment Problem and its Lagrangian Relaxations.

Listing B.79 presents the output of the GAP Optimal Assignment Problem and the results of the two Lagrangian relaxations. The output includes the optimal assignment of jobs to agents, the associated costs and workloads, the total cost of the optimal assignment, and the lower bounds obtained from both relaxation methods.

Output

```
Solving GAP Optimal Assignment Problem:

Optimal assignment and costs:
Job 1 assigned to Agent 1 (Cost = 8, Workload = 2)
Job 2 assigned to Agent 1 (Cost = 6, Workload = 4)
Job 3 assigned to Agent 2 (Cost = 7, Workload = 4)
Job 4 assigned to Agent 3 (Cost = 7, Workload = 4)
Job 5 assigned to Agent 3 (Cost = 8, Workload = 3)
```

```
Total Cost: 36.0

--- Lagrangian Relaxation ---
Lower Bound from Relaxation 1: 29.36515474060723
Lower Bound from Relaxation 2: 35.2
```

Listing B.79: Output of GAP Optimal Assignment Problem and Lagrangian Relaxations.

Discussion

Comparing the two methods reveals a clear winner for this specific problem instance:

- **Relaxation 1 Lower Bound**: 29.365
- **Relaxation 2 Lower Bound**: 35.2
- **Optimal Cost**: 36.0

Relaxation 2 provides a much tighter (better) lower bound, coming very close to the true optimal cost. This indicates that relaxing the capacity constraints (Relaxation 2) results in a problem that is a better approximation of the original GAP than relaxing the assignment constraints (Relaxation 1). The choice of which constraints to relax is crucial and can significantly impact the quality of the resulting bound. In this case, maintaining the "one agent per job" rule while penalizing capacity violations proved to be a more effective estimation strategy.

B.8 Chapter 8

B.8.1 Exercise 8.1

B.8.1.1 a) Solving BKP using Branch-and-Bound with DFS search strategy

Listing B.80 presents a Julia code to solve the Binary Knapsack Problem (BKP) using the Branch-and-Bound (B&B) framework with a Depth-First Search (DFS) strategy. The code defines the necessary data structures, functions for finding an initial incumbent solution, solving the LP relaxation, checking pruning conditions, branching, and determining optimality. It also includes an example of how to use the `BnB.jl` framework to solve a specific instance of the BKP.

```
using BnB   # BnB Framework
```

```julia
using JuMP  # Modeling language
using HiGHS # Solver

# Structure to represent the BKP data
struct BKPData <: BnB.BnBData
    v::Vector{Int64} # values
    w::Vector{Int64} # weights
    W::Int64         # capacity
    n::Int           # number of items
    # Constructor
    function BKPData(;v::Vector{Int64}, w::Vector{Int64}, W::Int64)
        return new(v, w, W, length(v))
    end
end

# Structure to represent the branch constraint
struct FixVariable <: BnB.BnBBranchConstraint
    i::Int64     # Variable index
    value::Int64 # Value to fix
end

# Function to find an initial incumbent solution using a greedy
heuristic based on value-to-weight ratio
function BnB.custom_incumbent(data::BKPData)
    # Value-to-weight ratios
    ratios = data.v ./ data.w
    # Sort items by decreasing ratio
    order = sortperm(ratios, rev=true)
    # Initialize solution vector
    solution = zeros(Int, data.n)
    # Greedy filling based on sorted order
    remaining_capacity = data.W
    incumbent_objective = 0.0
    for i in order
        if data.w[i] <= remaining_capacity
            solution[i] = 1
            remaining_capacity -= data.w[i]
            incumbent_objective += data.v[i]
        end
    end
```

```
    return BnB.BnBSolution(solution=solution,
objective=incumbent_objective)
end

# Function to solve the LP relaxation of the BKP for a given node
function BnB.custom_relaxation(node::BnB.BnBNode, bnb::BnB.BnBCore)
    model = JuMP.Model(HiGHS.Optimizer)
    JuMP.set_silent(model)
    @variable(model, 0 <= x[i in 1:bnb.data.n] <= 1)
    @objective(model, Max, sum(bnb.data.v[i] * x[i] for i in
1:bnb.data.n))
    @constraint(model, sum(bnb.data.w[i] * x[i] for i in 1:bnb.data.n)
<= bnb.data.W)
    # Add constraints for fixed variables
    for branch in node.branch_constraints
        @constraint(model, x[branch.i] == branch.value)
    end
    JuMP.optimize!(model)
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        return BnB.BnBSolution(solution=JuMP.value.(x),
objective=JuMP.objective_value(model))
    end
end

# Function to check pruning conditions for a given node and update its
status accordingly
function BnB.custom_prune(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # 1. Prune by Infeasibility
    if isnothing(node.relaxation.solution)
        return BnB.PrunedByInfeasibility
    end
    # 2. Prune by Integrality
    if !any(x -> 1e-5 < x < (1 - 1e-5), node.relaxation.solution)
        return BnB.PrunedByIntegrality
    end
    # 3. Prune by Bound
    if node.relaxation.objective <= bnb.incumbent.objective
        return BnB.PrunedByBound
    end
    return BnB.Active
end
```

```
# Function to select the branches and the values to fix for a given
node
function BnB.custom_branch(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Create two branches: one excluding the edge and one including the
edge
    branch_constraints = Vector{BnB.BnBBranchConstraint}()
    # Find all fractional variables in the solution
    fractional_variables = findall(x -> 1e-5 < x < (1 - 1e-5),
node.relaxation.solution)
    # Create branches
    for var in fractional_variables
        left_branch = FixVariable(var, 0)
        right_branch = FixVariable(var, 1)
        # Check if the variable is already fixed to 0.0 in the current
node
        if left_branch in node.branch_constraints || right_branch in
node.branch_constraints
            continue
        end
        push!(branch_constraints, left_branch)  # Left branch: exclude
the item
        push!(branch_constraints, right_branch) # Right branch: include
the item
        return branch_constraints
    end
end

# Function to determine if a node contains an optimal solution
function BnB.custom_is_optimal_solution(node::BnB.BnBNode,
bnb::BnB.BnBCore)
    # Check if the node has a valid solution
    if isnothing(node.relaxation.solution)
        return false
    end
    # Check if the solution is integral (i.e., all variables are either
0 or 1)
    is_integral = !any(x -> 1e-5 < x < (1 - 1e-5),
node.relaxation.solution)
    # Check if the relaxation value matches the best incumbent
objective value within a tolerance
    is_optimal = isapprox(node.relaxation.objective,
bnb.incumbent.objective; atol=1e-6)
```

```
    # It is optimal if it has an integral solution and its relaxation
value matches the incumbent objective value
    if is_integral && is_optimal
        return true
    end
    return false
end

# Example
data = BKPData(
    v = [8, 16, 20, 12, 6, 10, 4],
    w = [3, 7, 9, 6, 3, 5, 2],
    W = 17
)

# Define plot options for this example
custom_plot_options = BnB.BnBPlotOptions(
    figure_size = (750, 1100),
    node_size = (130, 90),
    node_label_size = 14,
    edge_label_size = 14,
    show_legend = true,
    legend_position = :bottom,
    branch_label = (branch::FixVariable) -> "x[$(branch.i)] =
$(branch.value)"
)

# Solve the BKP using the BnB framework
solution = BnB.solve(data, is_maximization = true, search_strategy
= :dfs, custom_plot_options = custom_plot_options)
```

Listing B.80: Julia code to solve BKP with B&B using DFS search strategy.

Listing B.81 presents the output of the BKP solved with BnB using the DFS search strategy. The output includes the total time taken, the best solution found, the objective value, the number of nodes explored and pruned, pruning statistics, and a visual representation of the search tree with nodes labeled by their objective values and pruning status.

Output

```
========== Branch-and-Bound Completed ==========
```

```
Total time: 0.1189730167388916 seconds
Best solution: [1.0, -0.0, 1.0, 0.0, 0.0, 1.0, -0.0]
Objective value: 38.0

Nodes explored: 15
Nodes pruned: 7
Pruning statistics:
  - Pruned by infeasibility: 5
  - Pruned by integrality: 1
  - Pruned by bound: 1

================================================
Search Tree
================================================

└── Node 1 (39.56)
    ├── Node 2 (38.0) ✂
    └── Node 3 (39.43)
        ├── Node 4 (38.0) ⭐
        └── Node 5 (38.67)
            ├── Node 6 (38.0)
            │   ├── Node 8 (38.0)
            │   │   ├── Node 10 (38.0)
            │   │   │   ├── Node 12 (38.0)
            │   │   │   │   ├── Node 14 (36.0) 🔒
            │   │   │   │   └── Node 15 (-Inf) 🚫
            │   │   │   └── Node 13 (-Inf) 🚫
            │   │   └── Node 11 (-Inf) 🚫
            │   └── Node 9 (-Inf) 🚫
            └── Node 7 (-Inf) 🚫

Legend:

  🔒 : Pruned by Integrality
  🚫 : Pruned by Infeasibility
  ✂ : Pruned by Bound
  ⭐ : Optimal

================================================
```

Listing B.81: Output of BKP solved with BnB using DFS search strategy.

Figure B.44 presents a visual representation of the B&B search tree for the BKP using the DFS strategy. The tree illustrates the branching decisions, objective values at each node, and pruning status, providing insight into the search process and how the algorithm navigates through the solution space.

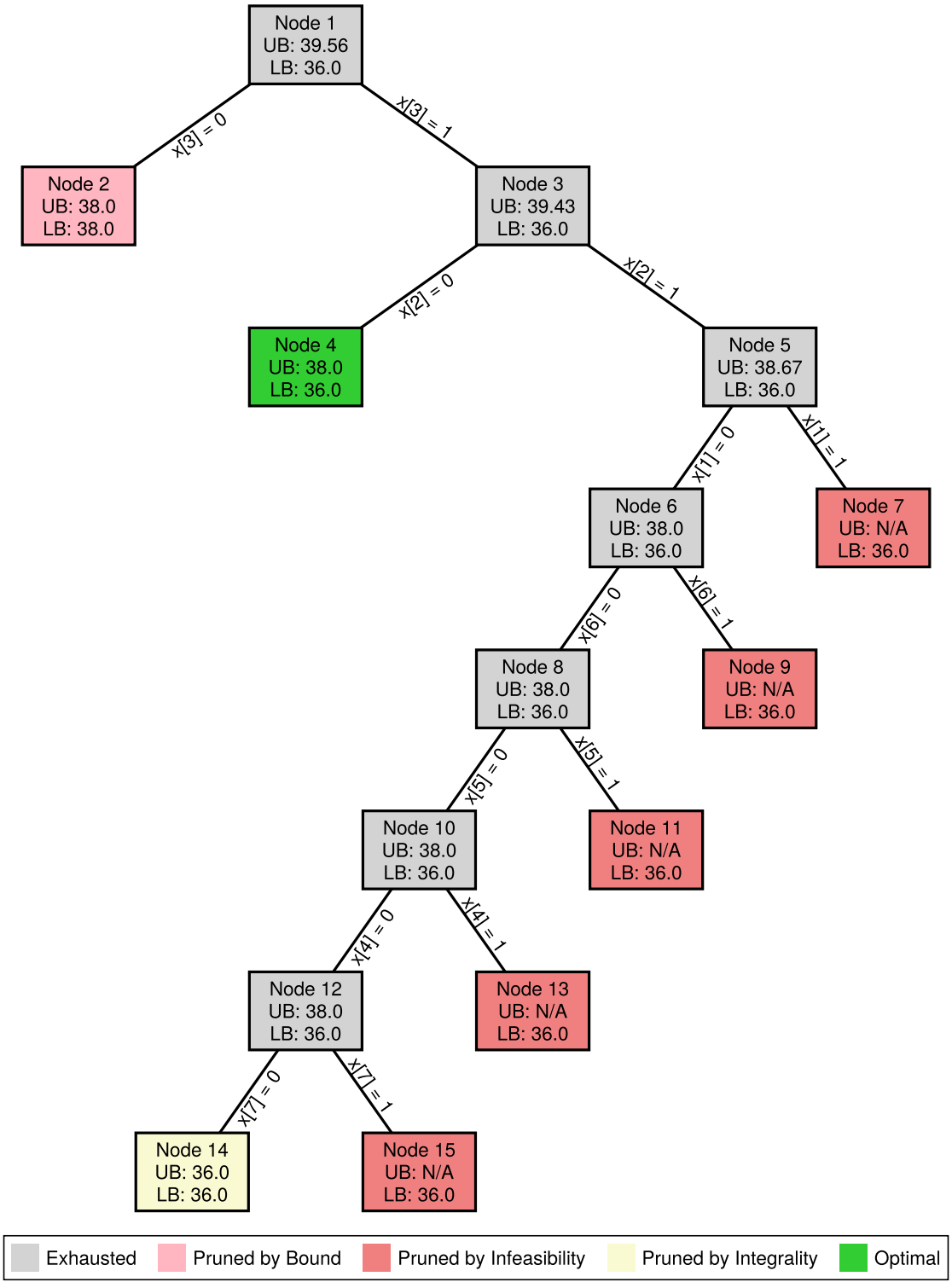


Figure B.44: B&B search tree for BKP using DFS strategy.

B.8.1.2 b) Solving BKP using B&B with Best Bound search strategy

Listing B.82 presents a Julia code to solve the Binary Knapsack Problem (BKP) using the Branch-and-Bound (B&B) framework with a Best Bound search strategy.

```
# Define plot options for this example
custom_plot_options = BnB.BnBPlotOptions(
    figure_size = (750, 650),
    node_size = (130, 90),
    node_label_size = 14,
    edge_label_size = 14,
    show_legend = true,
    legend_position = :bottom,
    branch_label = (branch::FixVariable) -> "x[$(branch.i)] =
$(branch.value)"
)

# Solve the BKP using the BnB framework
solution = BnB.solve(data, is_maximization = true, search_strategy
= :best_bound, custom_plot_options = custom_plot_options)
```

Listing B.82: Julia code to solve BKP using B&B with Best Bound search strategy.

Listing B.83 presents the output of the BKP solved with B&B using the Best Bound search strategy. The output includes the total time taken, the best solution found, the objective value, the number of nodes explored and pruned, pruning statistics, and a visual representation of the search tree with nodes labeled by their objective values and pruning status.

Output

```
========== Branch-and-Bound Completed ==========

Total time: 0.02105402946472168 seconds
Best solution: [1.0, -0.0, 1.0, 0.0, 0.0, 1.0, -0.0]
Objective value: 38.0

Nodes explored: 13
Nodes pruned: 6
Pruning statistics:
  - Pruned by infeasibility: 2
  - Pruned by integrality: 0
  - Pruned by bound: 4
```

```
================================================
Search Tree
================================================

└── Node 1 (39.56)
    ├── Node 2 (38.0)
    │   ├── Node 8 (38.0) ✂
    │   └── Node 9 (38.0)
    │       ├── Node 12 (38.0) ✂
    │       └── Node 13 (36.86) ✂
    └── Node 3 (39.43)
        ├── Node 4 (38.0) ⭐
        └── Node 5 (38.67)
            ├── Node 6 (38.0)
            │   ├── Node 10 (38.0) ✂
            │   └── Node 11 (-Inf) 🚫
            └── Node 7 (-Inf) 🚫

Legend:

  🔒 : Pruned by Integrality
  🚫 : Pruned by Infeasibility
  ✂ : Pruned by Bound
  ⭐ : Optimal

================================================
```

Listing B.83: Output of BKP solved with B&B using Best Bound search strategy.

Figure B.45 presents a visual representation of the B&B search tree for the BKP using the Best Bound strategy. The tree illustrates the branching decisions, objective values at each node, and pruning status, providing insight into the search process and how the algorithm navigates through the solution space.

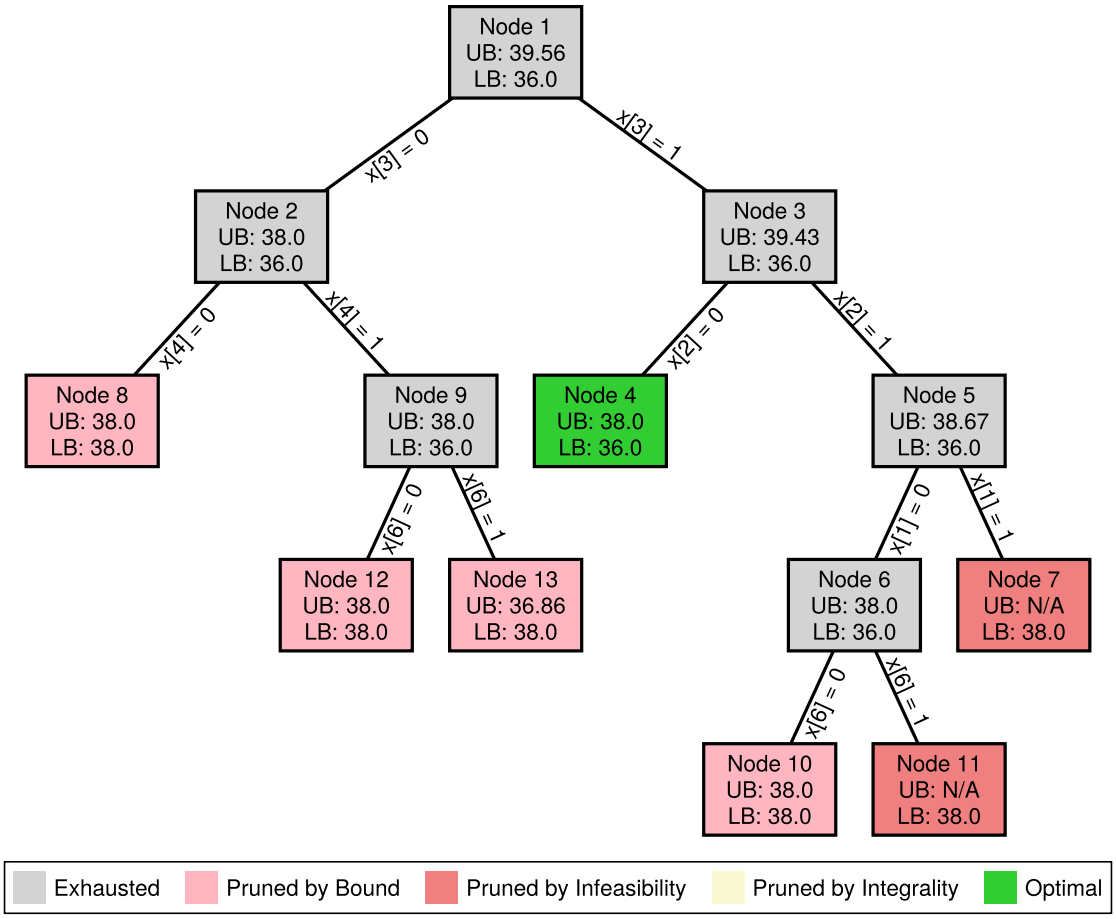


Figure B.45: B&B search tree for BKP using Best Bound strategy.

B.8.1.3 c) Compare the results obtained in exercises (a) and (b). Is there an advantage of using the Best Bound search strategy instead of the DFS for this instance of the BKP?

Yes. Based on these results (Table B.6), the **Best Bound** strategy appears to have a clear advantage over **DFS** for this particular Binary Knapsack instance.

Metric	DFS (A)	Best Bound (B)	Better
Time	0.143 s	0.043 s	Best Bound
Nodes explored	15	13	Best Bound
Nodes pruned	7	6	Similar
Pruned by bound	1	4	Best Bound
Best objective	38.0	38.0	Tie

Table B.6: Difference between DFS and Best Bound strategies.

Both methods found the same optimal solution (value 38.0), but Best Bound reached it with:

- fewer explored nodes,
- substantially more pruning by bound,
- and a much shorter runtime.

DFS explores one branch deeply before considering alternatives. In the BNB tree, after reaching Node 6, DFS continues exploring nodes 8, 10, 12, 14 and 15 before returning to other promising parts of the tree. As a result, DFS spends effort investigating nodes whose upper bounds are not particularly attractive. The consequence is:

- only 1 node pruned by bound,
- several infeasibility detections occurring late,
- 15 explored nodes.

The Best Bound strategy always selects the open node with the highest LP relaxation bound. After the root:

- Node 3 (39.43) is preferred over Node 2 (38.0).
- Node 5 (38.67) is preferred over weaker nodes.

Therefore, the search concentrates first on the most promising regions of the tree.

B.8.2 Exercise 8.2

Listing B.84 presents a Julia code to solve the Traveling Salesman Problem (TSP) using the Branch-and-Bound (B&B) framework. The code defines the necessary data structures, functions for finding an initial incumbent solution, solving the LP relaxation, checking pruning conditions, branching, and determining optimality. It also includes an example of how to use the `BnB.jl` framework to solve a specific instance of the TSP.

```
using BnB       # BnB Framework
using JuMP      # Modeling language
using HiGHS     # Solver
using CSV       # Data handling
using Distances # Distance computations

# Structure to represent the TSP data
struct TSPData <: BnB.BnBData
    coordinates::Matrix{Float64}     # City coordinates
```

```
    distance_matrix::Matrix{Float64} # Precomputed distance matrix
    n::Int                           # Number of cities
    # Constructor
    function TSPData(;file_path::String)
        coordinates = CSV.read(file_path, CSV.Tables.matrix,
header=false)
        distance_matrix = Distances.pairwise(Distances.Haversine(),
coordinates, dims=1)
        return new(coordinates, distance_matrix, size(coordinates, 1))
    end
end

# Structure to represent the branch constraint
struct FixVariable <: BnB.BnBBranchConstraint
    i::Int64     # Variable index i
    j::Int64     # Variable index j
    value::Int64 # Value to fix
end

# Function to find an initial incumbent solution using the Nearest
Neighbor heuristic
function BnB.custom_incumbent(data::TSPData)
    visited = falses(data.n)
    solution = zeros(Int, data.n)
    # Start from the first city
    current_city = 1
    solution[1] = current_city
    visited[current_city] = true
    incumbent_objective = 0.0
    for k in 2:data.n
        # Find the nearest unvisited city
        next_city = argmin([visited[j] ? Inf :
data.distance_matrix[current_city, j] for j in 1:data.n])
        solution[k] = next_city
        visited[next_city] = true
        incumbent_objective += data.distance_matrix[current_city,
next_city]
        current_city = next_city
    end
    # Add the distance to return to the starting city
    incumbent_objective += data.distance_matrix[current_city,
solution[1]]
```

```
    return BnB.BnBSolution(solution=solution,
objective=incumbent_objective)
end

# Function to solve the LP relaxation of the TSP for a given node
function BnB.custom_relaxation(node::BnB.BnBNode, bnb::BnB.BnBCore)
    model = JuMP.Model(HiGHS.Optimizer)
    JuMP.set_silent(model)
    @variable(model, x[i in 1:bnb.data.n, j in 1:bnb.data.n], Bin)
    @objective(model, Min, sum(bnb.data.distance_matrix[i, j] * x[i, j]
for i in 1:bnb.data.n, j in 1:bnb.data.n))
    @constraint(model, [i in 1:bnb.data.n], x[i, i] == 0)
    @constraint(model, [j in 1:bnb.data.n], sum(x[i, j] for i in
1:bnb.data.n) == 1)
    @constraint(model, [i in 1:bnb.data.n], sum(x[i, j] for j in
1:bnb.data.n) == 1)
    # Add constraints for fixed variables
    for branch in node.branch_constraints
        @constraint(model, x[branch.i, branch.j] == branch.value)
    end
    JuMP.optimize!(model)
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        # Extract the routes from the solution
        x_val = JuMP.value.(x)
        # Identify cycles in the solution to check for integrality and
to identify subtours
        visited = fill(false, bnb.data.n)
        cycles = Vector{Vector{Int}}()
        for start in 1:bnb.data.n
            if !visited[start]
                cycle = Int[start]
                visited[start] = true
                nxt = findfirst(x_val[start, :] .> 0.5)
                while nxt !== nothing && !visited[nxt]
                    push!(cycle, nxt)
                    visited[nxt] = true
                    nxt = findfirst(x_val[nxt, :] .> 0.5)
                end
                push!(cycles, cycle)
            end
        end
        # return cycles, JuMP.objective_value(model)
```

```
        return BnB.BnBSolution(solution=cycles,
objective=JuMP.objective_value(model))
    end
end

# Function to check pruning conditions for a given node and update its
status accordingly
function BnB.custom_prune(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # 1. Prune by Infeasibility
    if isnothing(node.relaxation.objective)
        return BnB.PrunedByInfeasibility
    end
    # 2. Prune by Integrality
    if length(node.relaxation.solution) == 1
        return BnB.PrunedByIntegrality
    end
    # 3. Prune by Bound
    if node.relaxation.objective >= bnb.incumbent.objective
        return BnB.PrunedByBound
    end
    return BnB.Active
end

# Function to select the branches and the values to fix for a given
node
function BnB.custom_branch(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Create two branches: one excluding the edge and one including the
edge
    branch_constraints = Vector{BnB.BnBBranchConstraint}()
    # Sort cycles by length to prioritize branching on the smallest
subtour
    subtours = sort(node.relaxation.solution, by=length)
    # Select the first edge of the smallest subtour for branching
    for subtour in subtours
        # Iterate through the edges of the subtour and find the first
edge that is not already fixed to 0.0 in the current node
        for i in 1:(length(subtour) - 1)
            left_branch = FixVariable(subtour[i], subtour[i+1], 0)
            right_branch = FixVariable(subtour[i+1], subtour[i], 0)
            # Check if the edge is already fixed to 0 in the current
node
```

```
            if left_branch in node.branch_constraints || right_branch
in node.branch_constraints
                continue
            end
            push!(branch_constraints, left_branch)  # Left branch:
exclude the edge
            push!(branch_constraints, right_branch) # Right branch:
exclude the reverse edge
            return branch_constraints
        end
    end
end

# Function to determine if a node contains an optimal solution
function BnB.custom_is_optimal_solution(node::BnB.BnBNode,
bnb::BnB.BnBCore)
    # Check if the node has a valid solution
    if isnothing(node.relaxation.solution)
        return false
    end
    # Check if the solution is integral (i.e., forms a single cycle)
    is_integral = length(node.relaxation.solution) == 1
    # Check if the relaxation value matches the best incumbent
objective value within a tolerance
    is_optimal = isapprox(node.relaxation.objective,
bnb.incumbent.objective; atol=1e-6)
    # A node is considered optimal if it has an integral solution and
its relaxation value matches the incumbent objective value
    if is_integral && is_optimal
        return true
    end
    return false
end

# Example
data = TSPData(file_path="data/tsp_instance.csv")

# Define plot options for this example
custom_plot_options = BnB.BnBPlotOptions(
    figure_size = (750, 400),
    node_size = (140, 90),
    node_label_size = 14,
```

```
    edge_label_size = 14,
    show_legend = true,
    legend_position = :bottom,
    branch_label = (branch::FixVariable) -> "x[$(branch.i),$(branch.j)]
= $(branch.value)"
)

# Solve the TSP using the BnB framework
solution = BnB.solve(data, is_maximization = false, search_strategy
= :best_bound, custom_plot_options = custom_plot_options)
```

Listing B.84: Julia code to solve the TSP using B&B.

Listing B.85 presents the output of the Traveling Salesman Problem (TSP) solved with the Branch-and-Bound (B&B) framework. The output includes the total time taken, the best solution found, the objective value, the number of nodes explored and pruned, pruning statistics, and a visual representation of the search tree with nodes labeled by their objective values and pruning status.

Output

```
========== Branch-and-Bound Completed ==========

Total time: 0.24185681343078613 seconds
Best solution: [[1, 6, 4, 2, 7, 3, 5]]
Objective value: 2866.791690832384

Nodes explored: 7
Nodes pruned: 2
Pruning statistics:
  - Pruned by infeasibility: 0
  - Pruned by integrality: 0
  - Pruned by bound: 2

================================================
Search Tree
================================================

└── Node 1 (2196.5)
    ├── Node 2 (2838.7)
    │   ├── Node 4 (2866.79) ⭐
    │   └── Node 5 (2868.28) ✂
```

```
    └── Node 3 (2838.7)
        ├── Node 6 (2868.28) ✂
        └── Node 7 (2866.79) ⭐

Legend:

  🔒 : Pruned by Integrality
  🚫 : Pruned by Infeasibility
  ✂ : Pruned by Bound
  ⭐ : Optimal

================================================
```

Listing B.85: Output of the Branch-and-Bound algorithm for the TSP.

Figure B.46 presents a visual representation of the B&B search tree for the Traveling Salesman Problem (TSP). The tree illustrates the branching decisions, objective values at each node, and pruning status, providing insight into the search process and how the algorithm navigates through the solution space.

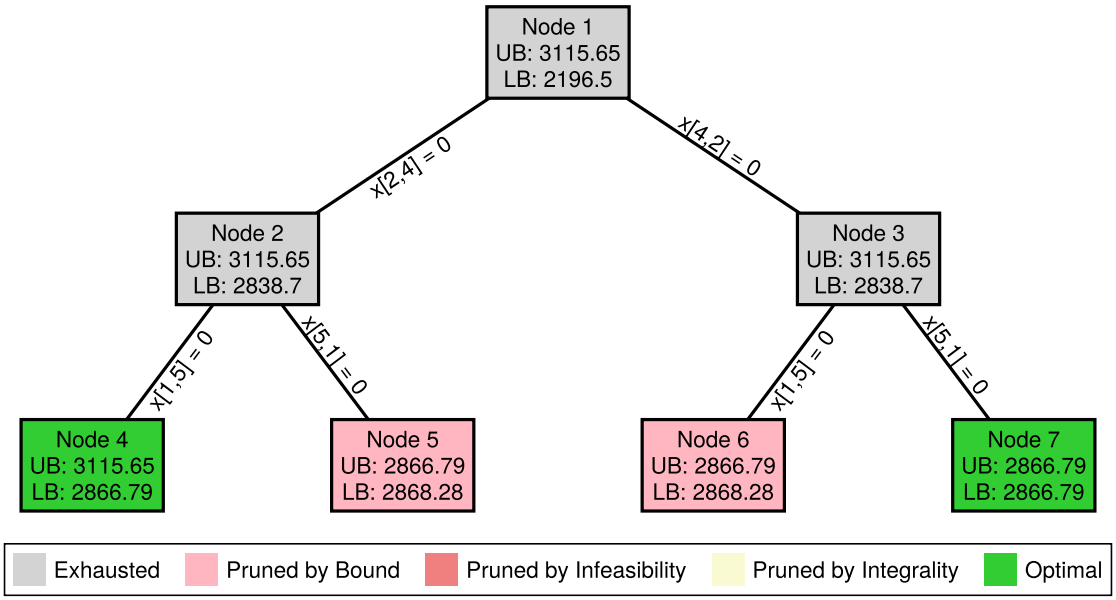


Figure B.46: B&B search tree for TSP.

B.8.3 Exercise 8.3

B.8.3.1 a) Solve the MISP using Branch-and-Bound technique

Listing B.86 presents a Julia code to solve the Maximum Independent Set Problem (MISP) using the Branch-and-Bound (B&B) framework. The code defines the necessary data structures, functions for finding an initial incumbent solution, solving the LP relaxation, checking pruning conditions, branching, and determining optimality. It also includes an example of how to use the `BnB.jl` framework to solve a specific instance of the MISP.

```
using BnB    # BnB Framework
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package

# Structure to represent the MISP data
struct MISPData <: BnB.BnBData
    graph::Graphs.SimpleGraph # Graph structure
    n::Int64                  # Number of vertices
    m::Int64                  # Number of edges
    # Constructor
    function MISPData(graph::Graphs.SimpleGraph)
        n = Graphs.nv(graph)
        m = Graphs.ne(graph)
        return new(graph, n, m)
    end
end

# Structure to represent the branch constraint
struct FixVariable <: BnB.BnBBranchConstraint
    v::Int64     # Variable index
    value::Int64 # Value to fix
end

# Function to find an initial incumbent solution using a greedy
heuristic
function BnB.custom_incumbent(data::MISPData)
    graph = data.graph
    n = data.n
    # Initialize the independent set and a set to track selected
vertices
```

```
    independent_set = Set{Int}()
    selected = falses(n)
    # Iterate over vertices in order of degree (lowest degree first)
    for v in sort(1:n, by=v -> Graphs.degree(graph, v))
        if !selected[v]
            push!(independent_set, v)
            selected[v] = true
            # Mark neighbors as selected to prevent their inclusion
            for neighbor in Graphs.neighbors(graph, v)
                selected[neighbor] = true
            end
        end
    end
    return BnB.BnBSolution(solution=independent_set,
objective=length(independent_set))
end

# Function to solve the LP relaxation of the MISP for a given node
function BnB.custom_relaxation(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define the decision variables
    @variable(model, 0 <= x[v in 1:bnb.data.n] <= 1)
    # Objective function: maximize the total of selected vertices
    @objective(model, Max, sum(x[v] for v in 1:bnb.data.n))
    # Independence constraint
    for e in Graphs.edges(bnb.data.graph)
        @constraint(model, x[e.src] + x[e.dst] <= 1)
    end
    # Add constraints for fixed variables
    for branch in node.branch_constraints
        @constraint(model, x[branch.v] == branch.value)
    end
    # Run the solver
    JuMP.optimize!(model)
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        return BnB.BnBSolution(solution=JuMP.value.(x),
objective=JuMP.objective_value(model))
    end
```

```
end

# Function to check pruning conditions for a given node and update its status accordingly
function BnB.custom_prune(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # 1. Prune by Infeasibility
    if isnothing(node.relaxation.solution)
        return BnB.PrunedByInfeasibility
    end
    # 2. Prune by Integrality
    if !any(x -> 1e-5 < x < (1 - 1e-5), node.relaxation.solution)
        return BnB.PrunedByIntegrality
    end
    # 3. Prune by Bound
    if node.relaxation.objective <= bnb.incumbent.objective
        return BnB.PrunedByBound
    end
    return BnB.Active
end

# Function to select the branches and the values to fix for a given node
function BnB.custom_branch(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Create two branches: one excluding the edge and one including the edge
    branch_constraints = Vector{BnB.BnBBranchConstraint}()
    # Find all fractional variables in the solution
    fractional_variables = findall(x -> 1e-5 < x < (1 - 1e-5), node.relaxation.solution)
    # Create branches
    for var in fractional_variables
        left_branch = FixVariable(var, 0)
        right_branch = FixVariable(var, 1)
        # Check if the variable is already fixed in the current node
        if left_branch in node.branch_constraints || right_branch in node.branch_constraints
            continue
        end
        push!(branch_constraints, left_branch)  # Left branch: exclude the vertice
        push!(branch_constraints, right_branch) # Right branch: include the vertice
```

```
        return branch_constraints
    end
end

# Function to determine if a node is optimal by checking if it has an integral solution and if its relaxation value matches the incumbent objective value
function BnB.custom_is_optimal_solution(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Check if the node has a valid solution
    if isnothing(node.relaxation.solution)
        return false
    end
    # Check if the solution is integral (i.e., all variables are either 0 or 1)
    is_integral = !any(x -> 1e-5 < x < (1 - 1e-5), node.relaxation.solution)
    # Check if the relaxation value matches the best incumbent objective value within a tolerance
    is_optimal = isapprox(node.relaxation.objective, bnb.incumbent.objective; atol=1e-6)
    # It is optimal if it has an integral solution and its relaxation value matches the incumbent objective value
    if is_integral && is_optimal
        return true
    end
    return false
end

# Example
graph = Graphs.smallgraph("icosahedral")
data = MISPData(graph)

# Define plot options for this example
custom_plot_options = BnB.BnBPlotOptions(
    figure_size = (950, 950),
    node_size = (95, 90),
    node_label_size = 14,
    edge_label_size = 14,
    show_legend = true,
    legend_position = :bottom,
```

```
    branch_label = (branch::FixVariable) -> "x[$(branch.v)] =
$(branch.value)"
)

# Solve the MISP using the BnB framework
solution = BnB.solve(data, is_maximization = true, search_strategy
= :best_bound, custom_plot_options = custom_plot_options)
```

Listing B.86: Julia code to solve the MISP using B&B technique.

Listing B.87 presents the output of the Maximum Independent Set Problem (MISP) solved with the Branch-and-Bound (B&B) framework. The output includes the total time taken, the best solution found, the objective value, the number of nodes explored and pruned, pruning statistics, and a visual representation of the search tree with nodes labeled by their objective values and pruning status.

Output

```
========== Branch-and-Bound Completed ==========

Total time: 0.28378987312316895 seconds
Best solution: Set([5, 3, 1])
Objective value: 3.0

Nodes explored: 23
Nodes pruned: 3
Pruning statistics:
  - Pruned by infeasibility: 0
  - Pruned by integrality: 2
  - Pruned by bound: 1

================================================
Search Tree
================================================

└── Node 1 (6.0)
    ├── Node 2 (5.5)
    │   ├── Node 4 (5.0)
    │   │   ├── Node 6 (4.5)
    │   │   │   ├── Node 8 (4.0)
    │   │   │   │   ├── Node 14 (3.5)
```

```
    │   │   │   │   │   ├── Node 20 (3.0) ⭐
    │   │   │   │   │   └── Node 21 (2.0) 🔒
    │   │   │   │   └── Node 15 (2.5) ✂
    │   │   │   └── Node 9 (3.0) ⭐
    │   │   └── Node 7 (3.5)
    │   │       ├── Node 16 (3.0) ⭐
    │   │       └── Node 17 (3.0) ⭐
    │   └── Node 5 (4.0)
    │       ├── Node 10 (3.5)
    │       │   ├── Node 22 (3.0) ⭐
    │       │   └── Node 23 (3.0) ⭐
    │       └── Node 11 (3.0) ⭐
    └── Node 3 (4.0)
        ├── Node 12 (3.5)
        │   ├── Node 18 (3.0) ⭐
        │   └── Node 19 (2.0) 🔒
        └── Node 13 (3.0) ⭐

Legend:

  🔒 : Pruned by Integrality
  🚫 : Pruned by Infeasibility
  ✂ : Pruned by Bound
  ⭐ : Optimal

==================================================
```

Listing B.87: Output of the B&B algorithm for the MISP.

Figure B.47 presents a visual representation of the B&B search tree for the Maximum Independent Set Problem (MISP). The tree illustrates the branching decisions, objective values at each node, and pruning status, providing insight into the search process and how the algorithm navigates through the solution space.

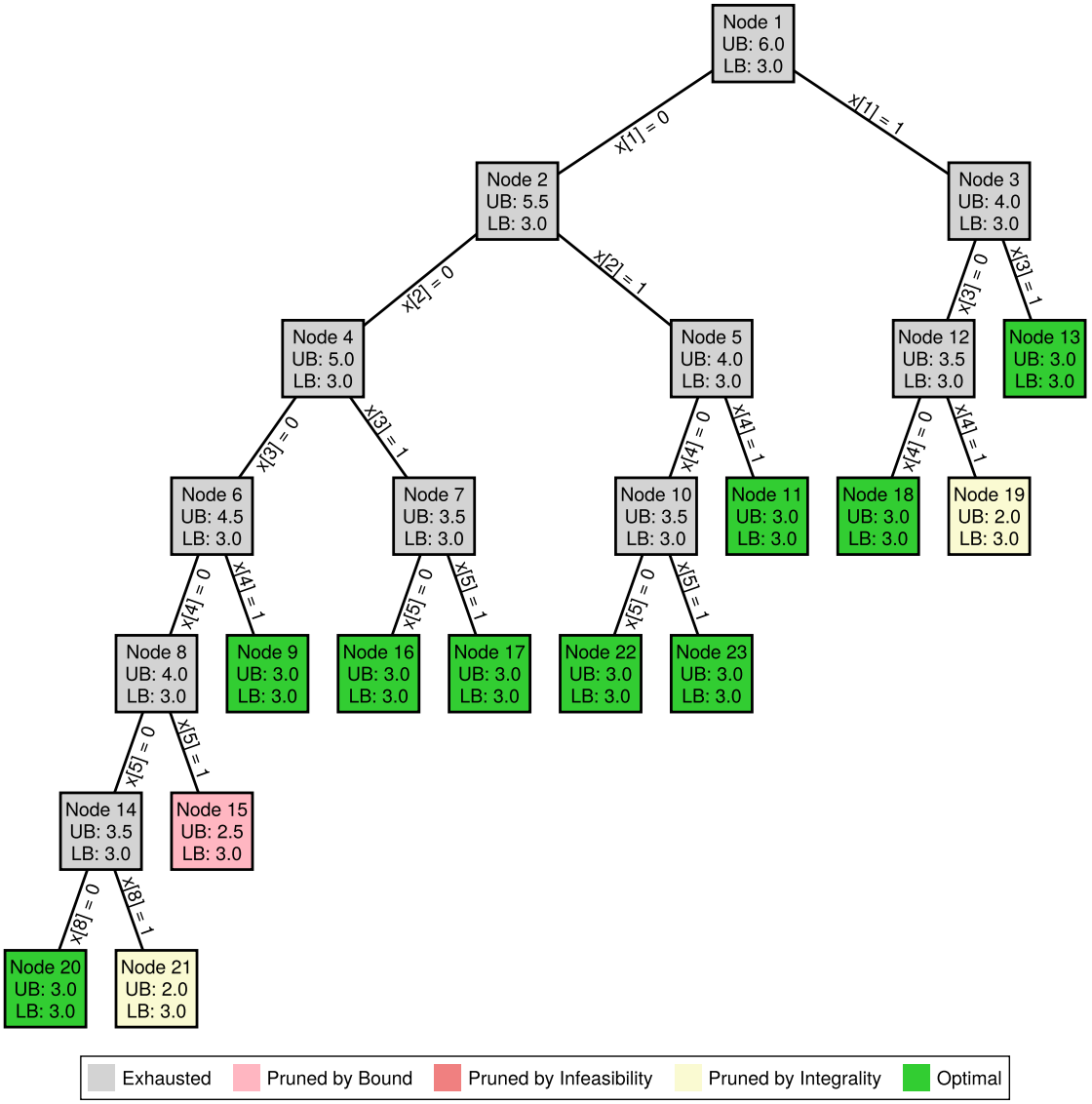


Figure B.47: B&B search tree for MISP.

B.8.3.2 b) Solve the MISP using Branch-and-Cut technique

The Branch-and-Cut technique is an extension of the Branch-and-Bound method that incorporates cutting planes to improve the LP relaxation of the problem.

Branch-and-Cut for MISP

Given an undirected graph $G = (V, E)$, an **independent vertex set** is a subset $S \subseteq V$ such that no two vertices in S are adjacent. The **Maximum Independent Set Problem** (MISP) asks for an independent set of maximum cardinality. Formally, the model is defined as:

$$\max_{x} \quad \sum_{v \in V} x_v \tag{B.23.1}$$

$$\text{s.t.} \quad x_u + x_v \leq 1 \quad \forall \{u, v\} \in E \tag{B.23.2}$$

$$x_v \in \{0, 1\} \qquad \forall v \in V \tag{B.23.3}$$

The constraint $x_u + x_v \leq 1$ for all edges $(u, v) \in E$ ensures that no two adjacent vertices are selected in the independent set. The binary variable x_v indicates whether vertex v is included in the independent set (1) or not (0).

While this formulation is intuitively simple, its Linear Programming (LP) relaxation is notoriously weak. For example, consider the triangle in Figure B.48. The LP relaxation of the MISP for this graph yields a solution of $x_1 = x_2 = x_3 = 0.5$ with objective value 1.5, which is not integral and does not correspond to any valid independent set.

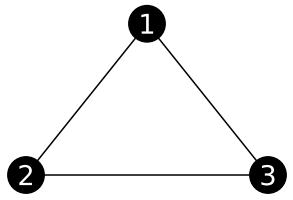


Figure B.48: Triangle graph.

To prevent such fractional solutions, we can add **clique inequalities** as **Cutting Planes**. A clique in a graph is a subset of vertices such that every two distinct vertices are adjacent. For any clique $C \subseteq V$, the following inequality must hold:

$$\sum_{v \in C} x_v \leq 1 \tag{B.24}$$

The clique constraint restricts the sum strictly to ≤ 1, cutting off the $x_1 = x_2 = x_3 = 0.5$ fractional solution.

A separation algorithm is used to identify violated clique inequalities in the current LP solution. If a violated clique inequality is found, it is added to the model as a cutting plane, and the LP relaxation is resolved. This process continues iteratively until no more violated clique inequalities can be found, resulting in a tighter LP relaxation that better approximates the convex hull of feasible integer solutions.

By injecting these cuts, the upper bound provided by the LP relaxation drops rapidly. This allows the algorithm to prune massive subtrees early because their upper bounds fall below the value of an already discovered feasible integer

solution, dramatically speeding up the overall solution time compared to pure branch-and-bound.

Algorithm B.5 presents a pseudo-code for the Branch-and-Cut algorithm applied to the MISP, outlining the steps of solving the LP relaxation, checking for integrality, separating cuts, and branching on fractional variables.

Branch-and-Cut Algorithm

1 Solve the current LP relaxation.

2 Check for integrality: If the solution happens to be fully integer (0 or 1), we record it if it's the best found so far, and prune this branch.

3 Separate Cuts: If the solution is fractional, look for violated inequalities (Cliques).

> If a violated cut is found, add it to the LP and go back to Step 1 (without branching yet!).
> If no more useful cuts can be found, branch on a fractional variable x_v, creating two new nodes.

Algorithm B.5: Branch-and-Cut for MISP.

Branch-and-Cut for MISP Julia Implementation

Listing B.88 presents a Julia code to solve the Maximum Independent Set Problem (MISP) using the Branch-and-Cut technique. The code defines the necessary data structures, functions for finding an initial incumbent solution, solving the LP relaxation with cutting planes, checking pruning conditions, branching, and determining optimality. It also includes an example of how to use the `BnB.jl` framework to solve a specific instance of the MISP.

```julia
using BnB    # BnB Framework
using JuMP   # Modeling language
using HiGHS  # Solver
using Graphs # Graphs package
using Random # For random shuffling in the heuristic

# Structure to represent the MISP data
struct MISPData <: BnB.BnBData
    graph::Graphs.SimpleGraph # Graph structure
    n::Int64                  # Number of vertices
    m::Int64                  # Number of edges
```

```
    # Constructor
    function MISPData(graph::Graphs.SimpleGraph)
        n = Graphs.nv(graph)
        m = Graphs.ne(graph)
        return new(graph, n, m)
    end
end

# Structure to represent the branch constraint
struct FixVariable <: BnB.BnBBranchConstraint
    v::Int64     # Variable index
    value::Int64 # Value to fix
end

# Function to find an initial incumbent solution using a greedy
heuristic
function BnB.custom_incumbent(data::MISPData)
    graph = data.graph
    # Initialize the independent set and a set to track selected
vertices
    independent_set = Set{Int}()
    selected = falses(data.n)
    # Iterate over vertices in order of degree (lowest degree first)
    for v in sort(1:data.n, by=v -> Graphs.degree(graph, v))
        if !selected[v]
            push!(independent_set, v)
            selected[v] = true
            # Mark neighbors as selected to prevent their inclusion
            for neighbor in Graphs.neighbors(graph, v)
                selected[neighbor] = true
            end
        end
    end
    return BnB.BnBSolution(solution=independent_set,
objective=length(independent_set))
end

# A simple heuristic to find a violated clique inequalities
function find_violated_cliques_multi(g, x; K=10)
    n = nv(g)
    cliques = Set{Set{Int}}()
```

```
    base = filter(v -> x[v] > 1e-4, 1:n)
    # Try multiple random orders to find different violated cliques
    for seed in 1:K
        # Random order of candidates to get different cliques in each
iteration
        candidates = copy(base)
        shuffle!(candidates)
        sort!(candidates, by=v->x[v], rev=true)
        clique = Set{Int}()
        weight = 0.0
        for v in candidates
            if all(u -> has_edge(g,u,v), clique)
                push!(clique, v)
                weight += x[v]
            end
        end
        # Only consider cliques of size >= 3 that violate the
inequality
        if length(clique) >= 3 && weight > 1 + 1e-4
            push!(cliques, clique)
        end
    end
    return cliques
end

# Function to solve the relaxation of the MISP for a given node using
Cutting Planes
function BnB.custom_relaxation(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Create the model
    model = JuMP.Model(HiGHS.Optimizer)
    # Silent mode (solver output is not printed)
    JuMP.set_silent(model)
    # Define the decision variables
    @variable(model, 0 <= x[v in 1:bnb.data.n] <= 1)
    # Objective function
    @objective(model, Max, sum(x[v] for v in 1:bnb.data.n))
    # Base Independence constraint (Edge Constraints)
    for e in Graphs.edges(bnb.data.graph)
        @constraint(model, x[e.src] + x[e.dst] <= 1)
    end
    # Add constraints for fixed variables from the BnB node
```

```
    for branch in node.branch_constraints
        @constraint(model, x[branch.v] == branch.value)
    end
    # Add cutting planes
    for clique in bnb.global_cuts
        @constraint(model, sum(x[v] for v in clique) <= 1)
    end
    # Run the solver
    JuMP.optimize!(model)
    # Check for violated clique constraints and add them to the global
cut pool
    violated_cliques = find_violated_cliques_multi(bnb.data.graph,
JuMP.value.(x), K=20)
    for violated_clique in violated_cliques
        if !isempty(violated_clique) && !(violated_clique in
bnb.global_cuts)
            push!(bnb.global_cuts, violated_clique)
        end
    end
    if JuMP.termination_status(model) == JuMP.OPTIMAL
        return BnB.BnBSolution(solution=JuMP.value.(x),
objective=JuMP.objective_value(model))
    end
end

# Function to check pruning conditions for a given node and update its
status accordingly
function BnB.custom_prune(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # 1. Prune by Infeasibility
    if isnothing(node.relaxation.solution)
        return BnB.PrunedByInfeasibility
    end
    # 2. Prune by Integrality
    if !any(x -> 1e-5 < x < (1 - 1e-5), node.relaxation.solution)
        return BnB.PrunedByIntegrality
    end
    # 3. Prune by Bound
    if node.relaxation.objective <= bnb.incumbent.objective
        return BnB.PrunedByBound
    end
    return BnB.Active
end
```

```
# Function to select the branches and the values to fix for a given
node
function BnB.custom_branch(node::BnB.BnBNode, bnb::BnB.BnBCore)
    # Create two branches: one excluding the edge and one including the
edge
    branch_constraints = Vector{BnB.BnBBranchConstraint}()
    # Find all fractional variables in the solution
    fractional_variables = findall(x -> 1e-5 < x < (1 - 1e-5),
node.relaxation.solution)
    # Create branches
    for var in fractional_variables
        left_branch = FixVariable(var, 0)
        right_branch = FixVariable(var, 1)
        # Check if the variable is already fixed in the current node
        if left_branch in node.branch_constraints || right_branch in
node.branch_constraints
            continue
        end
        push!(branch_constraints, left_branch)  # Left branch: exclude
the vertice
        push!(branch_constraints, right_branch) # Right branch: include
the vertice
        return branch_constraints
    end
end

# Function to determine if a node contains an optimal solution
function BnB.custom_is_optimal_solution(node::BnB.BnBNode,
bnb::BnB.BnBCore)
    # Check if the node has a valid solution
    if isnothing(node.relaxation.solution)
        return false
    end
    # Check if the solution is integral (i.e., all variables are either
0 or 1)
    is_integral = !any(x -> 1e-5 < x < (1 - 1e-5),
node.relaxation.solution)
    # Check if the relaxation value matches the best incumbent
objective value within a tolerance
    is_optimal = isapprox(node.relaxation.objective,
bnb.incumbent.objective; atol=1e-6)
```

```
    # It is optimal if it has an integral solution and its relaxation
value matches the incumbent objective value
    if is_integral && is_optimal
        return true
    end
    return false
end

# Example
graph = Graphs.smallgraph("icosahedral")
data = MISPData(graph)

# Define plot options for this example
custom_plot_options = BnB.BnBPlotOptions(
    figure_size = (750, 650),
    node_size = (95, 90),
    node_label_size = 14,
    edge_label_size = 14,
    show_legend = true,
    legend_position = :bottom,
    branch_label = (branch::FixVariable) -> "x[$(branch.v)] =
$(branch.value)"
)

# Solve the MISP using the BnB framework
Random.seed!(42) # Set a random seed for reproducibility
solution = BnB.solve(data, is_maximization = true, search_strategy
= :best_bound, custom_plot_options = custom_plot_options)
```

Listing B.88: Julia code to solve the MISP using Branch-and-Cut technique.

Listing B.89 presents the output of the Maximum Independent Set Problem (MISP) solved with the Branch-and-Cut technique. The output includes the total time taken, the best solution found, the objective value, the number of nodes explored and pruned, pruning statistics, and a visual representation of the search tree with nodes labeled by their objective values and pruning status.

Output

```
========== Branch-and-Bound Completed ==========

Total time: 0.5160820484161377 seconds
```

```
Best solution: Set([5, 3, 1])
Objective value: 3.0

Nodes explored: 13
Nodes pruned: 0
Pruning statistics:
  - Pruned by infeasibility: 0
  - Pruned by integrality: 0
  - Pruned by bound: 0

Cuts generated: 14

================================================
Search Tree
================================================

└── Node 1 (6.0)
    ├── Node 2 (3.83)
    │   ├── Node 4 (3.5)
    │   │   ├── Node 10 (3.5)
    │   │   │   ├── Node 12 (3.0) ⭐
    │   │   │   └── Node 13 (3.0) ⭐
    │   │   └── Node 11 (3.0) ⭐
    │   └── Node 5 (3.5)
    │       ├── Node 8 (3.0) ⭐
    │       └── Node 9 (3.0) ⭐
    └── Node 3 (3.5)
        ├── Node 6 (3.0) ⭐
        └── Node 7 (3.0) ⭐

Legend:

  🔒 : Pruned by Integrality
  🚫 : Pruned by Infeasibility
  ✂ : Pruned by Bound
  ⭐ : Optimal

================================================
```

Listing B.89: Branch-and-Cut Output for MISP.

Figure B.49 presents a visual representation of the search tree for the Maximum Independent Set Problem (MISP) solved with the Branch-and-Cut technique. The figure shows the nodes labeled by their objective values and pruning status.

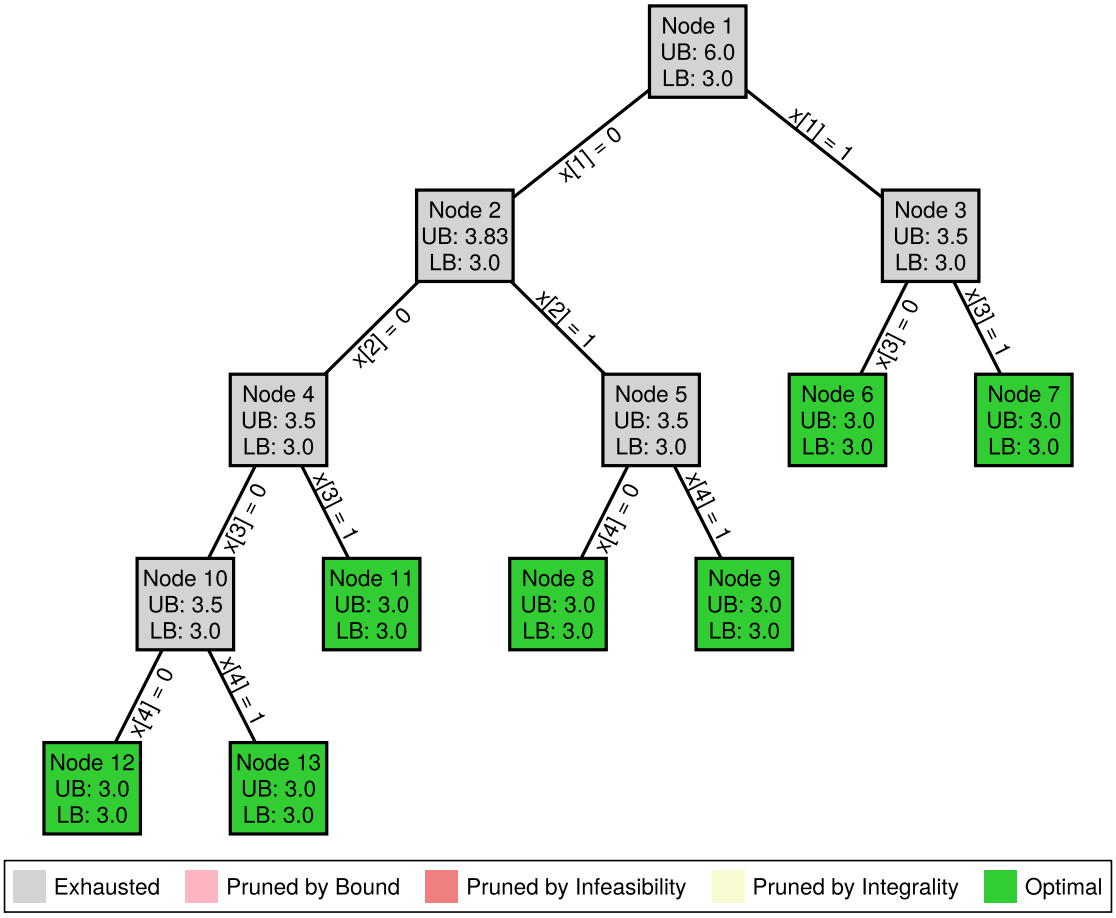


Figure B.49: Branch-and-Cut Search Tree for MISP.

B.8.3.3 c) Compare the performance of the Branch-and-Bound and Branch-and-Cut methods for solving the MISP

Table B.7 presents a comparison of the performance metrics between the Branch-and-Bound and Branch-and-Cut methods for solving the Maximum Independent Set Problem (MISP). The table includes the objective value, time taken, number of nodes explored, number of nodes pruned, and the number of cuts generated.

Metric	Branch-and-Bound	Branch-and-Cut
Objective value	3.0	3.0
Time	0.384 s	0.551 s
Nodes explored	23	13

Metric	Branch-and-Bound	Branch-and-Cut
Nodes pruned	3	0
Cuts generated	0	14

Table B.7: Difference between Branch-and-Bound and Branch-and-Cut strategies for solving the MISP.

The generated cuts significantly strengthened the LP relaxation, reducing the number of explored nodes from 23 to 13. However, the computational overhead associated with generating and separating 14 cuts outweighed the reduction in tree size, resulting in a longer overall runtime. Therefore, while Branch-and-Cut produced a smaller search tree, it did not provide a runtime advantage on this small instance. The benefits of Branch-and-Cut would be expected to become more pronounced on larger and more difficult MISP instances, where the stronger relaxation can substantially reduce the size of the branch-and-bound tree.