CMPE 492

Partitioning Undirected Graphs

Aral Dörtoğul

Advisor:

Can Özturan

January 14, 2024

TABLE OF CONTENTS

1. INTRODUCTION

Graph partitioning, a fundamental problem in computer science, involves dividing a graph into smaller, more manageable components. While heuristic algorithms have historically been employed to tackle this challenge, the aim of this research is to go beyond approximations. By leveraging modern computing capabilities, this study seeks to develop an exact algorithm for solving the NP-hard graph partitioning problem, even for smaller graphs that were once considered computationally intensive.

To address the NP-hard nature of graph partitioning, an integer linear programming approach is adopted in this research. This involves formulating the partitioning problem as a mathematical model, which is then solved using the Gurobi optimization engine in conjunction with Python. This theoretical framework not only enhances the understanding of the problem but also facilitates the translation of mathematical precision into practical, computer-executable solutions.

1.1. Broad Impact

Undirected graph partitioning, while posing a computational challenge, holds immense potential for influencing various domains. This research can extend to applications such as parallel processing, image processing, and VLSI design. In the realm of parallel processing, efficient graph partitioning can significantly enhance the distribution of computing tasks across multiple processors, leading to improved performance and speed. Likewise, in image processing, the ability to accurately partition graphs can streamline the analysis of complex visual data. Moreover, in VLSI design, optimized graph partitioning contributes to the creation of more efficient and compact electronic circuits. As computers continue to play an essential role in diverse fields, the impact of exact graph partitioning solutions becomes increasingly pronounced.

1.2. Ethical Considerations

The pursuit of advanced computational solutions brings with it ethical responsibilities. In the context of graph partitioning, ethical considerations revolve around reliability, fairness, transparency, and accountability in the use of partitioning algorithms. As these algorithms influence decision-making processes and resource allocations, it is crucial to ensure that their deployment is unbiased and just. This research is committed to navigating these ethical dimensions by prioritizing responsible algorithmic deployment and promoting transparency in its application.

2. PROJECT DEFINITION AND PLANNING

2.1. Project Definition

As memory become cheaper and the processing power of CPUs increase with technological advancements, the complexity of the problems solved by computers also increase. The classical methods of solving problems with a single threaded fashion is sometimes ineffective in large problems. For such cases, it is much more convenient to divide the data into several partitions that have equal load so that algorithms on data can be operated in a parallel fashion by multiple CPUs. Of course, parallelism brings the problem of interprocessor communication, which is time consuming. In order to overcome this, it is crucial to divide the data in a way that the communication between processors is minimized. For this context, graphs are a great abstraction to model this problem.

Graph theory is a branch of mathematics that focuses on the examination of graphs —mathematical structures composed of "vertices" or nodes connected by "edges" or arcs, as depicted in Figure 2.1. This field finds widespread applications in diverse domains, including computer science, engineering, physics, social sciences, and operations research.

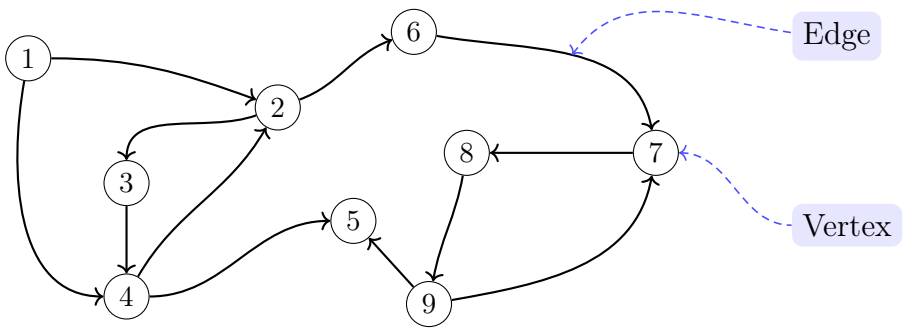
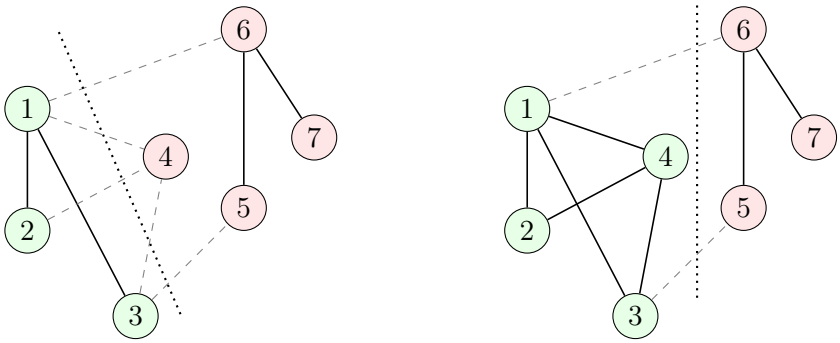


Figure 2.1. A directed graph

Graph partitioning algorithms partition a graph into equal sized subgraphs (in

terms of nodes) in such a way that the number of edges cut is minimized. For example, Figure 2.2 illustrates different ways of partitioning a graph, but not all alternatives provide an optimal solution.



Bad partition Good partition

Figure 2.2. Different partitions of a graph

In the more general problem, edges can also have weights. Moreover, since this problem is a minimization problem, it can be modeled as an integer programming model, which is used commonly in operations research.

2.2. Project Planning

The following is planned as follows:

(i) Construction of an integer programming model for the weightless, undirected graph partitioning problem.
(ii) Development of an algorithm that solves the model using a integer programming solver like Gurobi.
(iii) Construction of another model for the weighted version of the problem.
(iv) Development of an algorithm for the weighted version of the problem.
(v) Evaluation of the software's performance in both cases (weighted and weightless) on a range of graph sizes and complexities.

2.2.1. Project Time and Resource Estimation

The gantt chart for the project plan is in Figure 2.3.

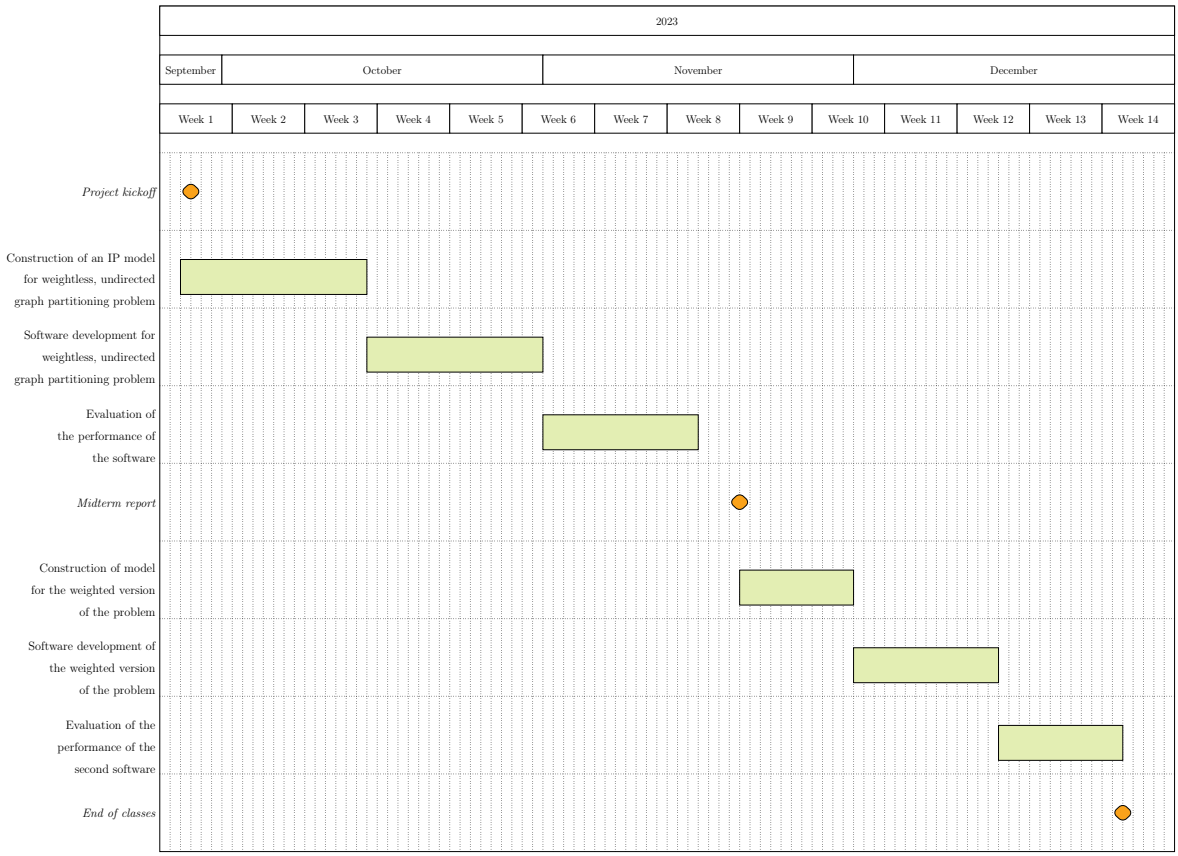


Figure 2.3. Project Plan

2.2.2. Success Criteria

The study is considered to be successful if

(i) weightless and weighted graphs can be partitioned by software,

(ii) the software is able to partition large (but sparse) graphs.

2.2.3. Risk Analysis

(i) Technical Risks

  (a) Algorithm Complexity

Risk The complexity of the algorithm may result in longer computation times, especially for larger graphs.

Impact Delays in delivering results and potential resource strain.

Likelihood High

Mitigation Implement algorithm optimizations, explore parallel computing options, and conduct thorough testing on a range of graph sizes and densities.

(b) Model Formulation Challenges

Risk Difficulty in accurately formulating the graph partitioning problem as an IP model.

Impact Incorrect solutions or failure to converge to an optimal solution.

Likelihood Low

Mitigation Perform rigorous testing on representative graph instances, conduct a comprehensive literature review.

(ii) Operational Risks

(a) Resource Constraints

Risk Inadequate computational resources for executing the algorithm efficiently.

Impact Slower execution and potential limitations on the size of the graphs that can be processed.

Likelihood Low to Moderate

Mitigation Assess and secure sufficient computational resources, optimize code for resource efficiency, and maybe explore cloud computing options.

(b) Data Security Concerns

Risk Potential vulnerabilities in handling sensitive graph data.

Impact Data breaches and/or unauthorized access.

Likelihood Low

Mitigation Implement secure data handling practices, use encryption where necessary.

3. RELATED WORK

The study of graph partitioning has a long history and can be traced back to various researchers and fields. While it's challenging to pinpoint a single origin, the problem has been explored by mathematicians, computer scientists, and researchers in related disciplines for many decades.

One early mention of graph partitioning can be found in the field of spectral graph theory. In the 1970s, researchers like F. R. K. Chung [1] began investigating the relationship between graph properties and eigenvalues of certain matrices associated with graphs. This work laid the foundation for spectral graph partitioning algorithms.

In the context of computer science and algorithms, the graph partitioning problem gained attention for parallel computing applications. During the 1980s and 1990s, researchers explored methods to efficiently partition graphs for parallel processing to optimize computational tasks. The proven NP-hard nature of the problem [2] led researchers to develop heuristics as well as exact algorithms.

3.1. Exact Algorithms

A lot of research has been done on ways to solve the Generalized Partitioning Problem (GPP). Some methods are made specifically the bipartitioning case, while others work for the general GPP. Most of these methods use a common approach called the branch and bound framework.

Researchers use different strategies to set limits on the solutions. According to Buluç [3], some use a method called semi-definite programming, like Karisch [4], or Armbruster [5]. Sellman and Sensen [6,7] use something called multi-commodity flows. Linear programming is another technique used by Brunetta [8], Ferreira [9], Lisser [10], and Armbruster [5], and others. Hager and others [11,12] take a continuous quadratic

approach and then use the branch-and-bound method on it.

3.2. Heuristics

There are several heuristics proposed to solve graph partitioning problem. Buluç [3] lists many heuristics, including:

- Spectral Partitioning
- Graph Growing
- Flows
- Geometric Partitioning
- Streaming Graph Partitioning (SGP)
- Node-swapping Local Search
- Multilevel Graph Partitioning

3.3. Graph Partitioning Software

Several software packages implement various graph partitioning algorithms. Notable ones include: [3]

Metis (and its variants kMetis and hMetis) Known for its speed and flexibility, Metis is widely used for graph partitioning, with kMetis focusing on partitioning speed and hMetis emphasizing partition quality, especially in hypergraph scenarios.

Scotch Developed by Pellegrini, Scotch offers a comprehensive graph partitioning framework with support for both sequential and parallel techniques. It utilizes recursive multilevel bisection for effective partitioning.

KaHIP (Karlsruhe High-Quality Partitioning) Released by Sanders and Schulz, KaHIP implements various advanced methods, including flow-based techniques, more-localized local searches, and diverse parallel and sequential meta-heuristics. It has demonstrated high performance in challenges and benchmarks.

Chaco One of the early publicly available packages, Chaco, developed by Hendrickson and Leland, implements multilevel approaches, basic local search algorithms, and spectral partitioning techniques.

ParMetis A parallel implementation of the Metis GP algorithm by Karypis and Kumar, ParMetis is widely used for efficient graph partitioning in parallel computing environments.

4. METHODOLOGY

4.1. Integer Programming

Integer programming (IP) is a mathematical optimization technique used to find the optimal solution to a problem, where the decision variables are required to take integer values. It is a special case of linear programming (LP), where the decision variables can be any real number. In the context of graph partitioning, integer programming becomes a powerful tool for formulating and solving the problems. In our case, the goal is to find an optimal way to partition the graph's vertices while considering specific constraints, such as minimizing the number of edges between partitions.

4.1.1. Methods of Solving Integer Programming Models

4.1.1.1. Branch and Bound. Branch and bound is a common way of solving integer programming models. Before solving the problem, the integer constraints on decision variables are relaxed and treated as continuous variables. Then, a basic, feasible solution is found using regular linear programming solving techniques like the simplex method. If the basic solution contains originally integer variables that are fractional, then branch and bound algorithm starts.

For the variables that got fractional values, the problem is split into two branches by adding additional constraints to omit the fractional value from the set of feasible solutions: For example, if x_i is originally an integer variable but $x_i = 5.3$ in the relaxed LP formulation, one branch adds the constraint $x_i \leq 5$ and the other adds $x_i \geq 6$. For each branch, the problem is solved again with the hope of finding an integer solution. This procedure is repeated until an integer solution is found and it is made sure that no better unknown solutions exist. [13] Figure 4.1 summarizes the branch and bound method on a sample maximization problem with 2 integer variables (x_1, x_2).

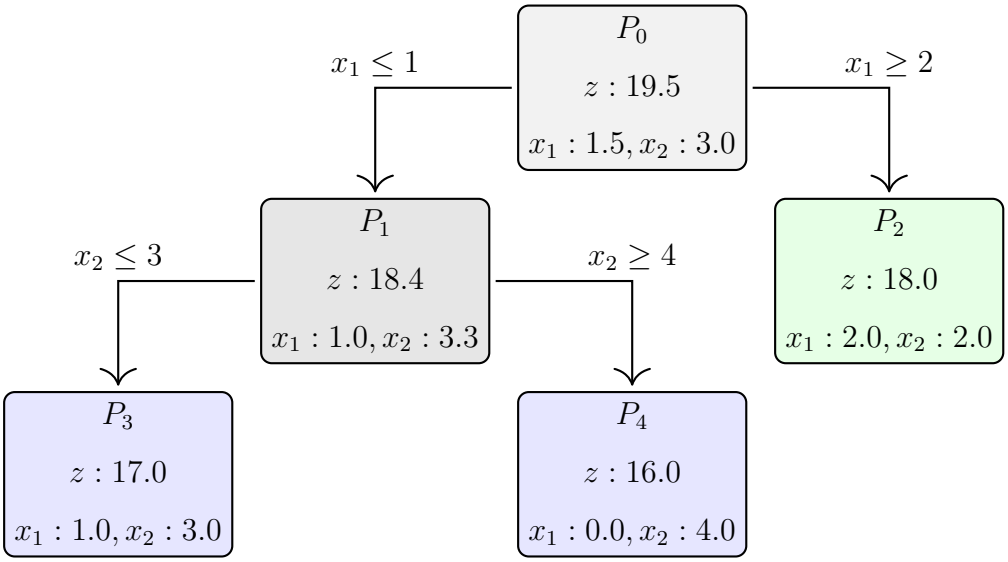


Figure 4.1. Branch and bound method

4.1.1.2. Cutting Plane. Cutting plane method, is similar to branch and bound. Initially, the relaxed version of the problem is solved using simplex. (Integer constraints are discarded.) Then, if the solution includes variables violating the integer constraint, additional cutting constraints called "Gomory cuts" are added to tighten the relaxed problem. The Gomory cuts are special kinds of cuts in the feasible region of the problem that omit the previously found optimal (but violating integrality constraint) solution without eliminating any other integer solutions of the problem. Cuts are added iteratively to achieve an integer solution. [14] An overview of the cutting plane algorithm can be seen in Figure 4.2

4.2. Gurobi

Gurobi is a cutting-edge optimization engine designed for solving complex mathematical optimization problems, including integer programming. [15] Gurobi uses branch and bound and cutting plane methods together while solving an integer model, and the joint use of these methods is called "branch and cut". One of Gurobi's notable features is its ability to perform parallel optimization, harnessing the power of modern computing architectures to accelerate problem-solving. By efficiently distributing computation across multiple processors, Gurobi enhances the speed and scalability of

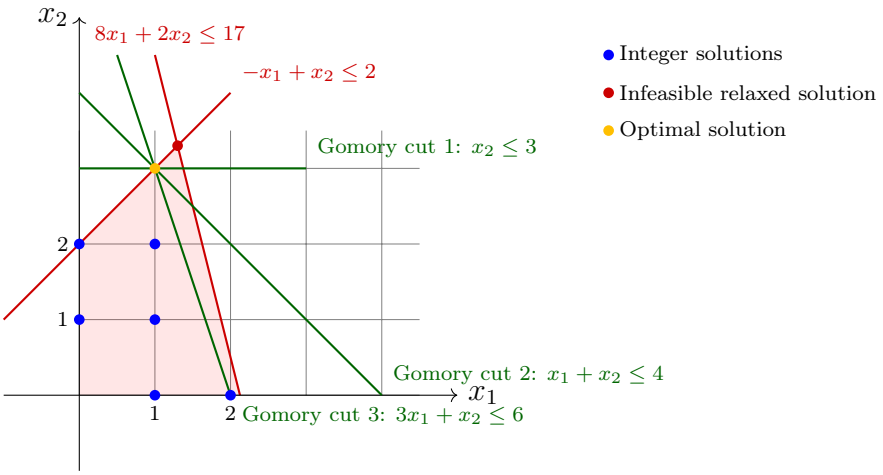


Figure 4.2. Cutting plane method

integer programming solutions. Because of these reasons, Gurobi is utilized in the software.

5. REQUIREMENTS SPECIFICATION

(i) Functional Requirements

(a) Algorithm

i. The algorithm **MUST** partition the graph into two equal subgraphs with minimum edge cuts.

ii. The algorithm **MUST** be able to receive a graph from the user as input.

iii. The algorithm **MUST** be able to give the output of edges cut and the partition information back to the user.

(b) Implementation

i. The algorithm **MUST** use the methodologies for solving integer programming models.

ii. The software **SHALL** utilize integer programming model solvers like Gurobi, CPLEX when solving the IP model.

(c) Performance

i. The software **MUST** be able to partition graphs of different sizes and densities.

6. DESIGN

6.1. Information Structure

The software is not linked to any database management system. The information flow occurs with basic input and output files and the terminal.

6.1.1. Input

6.1.1.1. Graph Data. The input graph is taken from the user and must be in the form of a Matrix Market file (.mtx file). The Matrix Market (MM) exchange formats is an ASCII based, human readable format designed by NIST[1] as a convenient way of exchanging matrix data. [16] The input matrix provided by the user has to be a symmetric matrix, as the software can only parititon undirected graphs.

6.1.2. Output

6.1.2.1. Gurobi's Log. The user can provide a log file name as an option when running the program in the terminal to tell Gurobi to write logs. The log file is a human readable, simple, text file that includes details about the optimization process of Gurobi.

6.1.2.2. Graph Plots. The user can provide a flag when running the program in the terminal to tell the software to draw a plot of the graph with the partitions.

6.1.2.3. Graph Partitioning Data. After a graph is partitioned, the user receives the graph partitioning data (edges cut, clusters etc.) from the terminal.

[1]National Institute of Standards and Technology

6.2. Information Flow

The flow of the software is summarized in Figure 6.1.

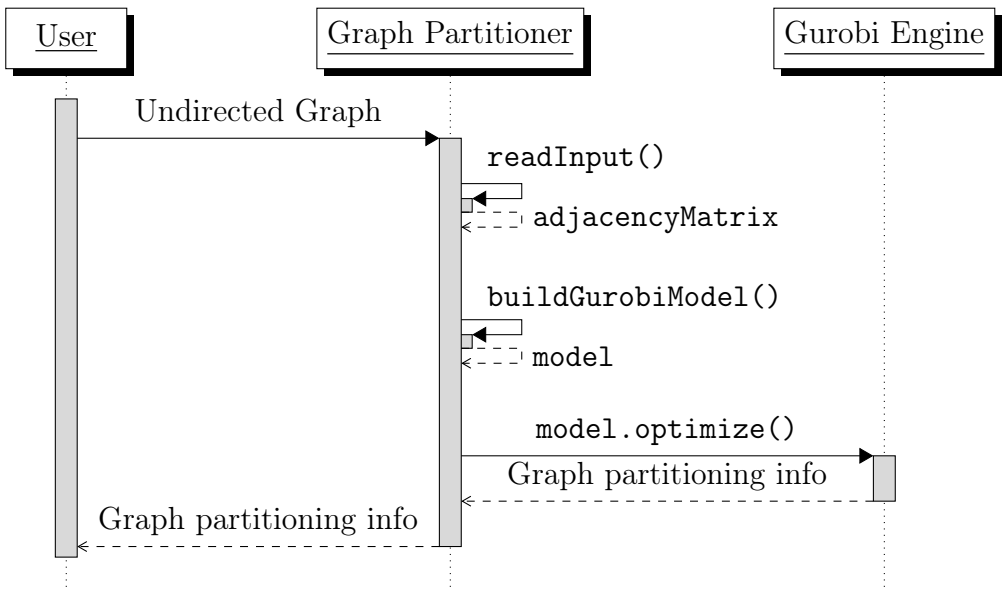


Figure 6.1. Sequence Diagram

6.3. System Design

6.3.1. Integer Programming Model

Let's define $V = \{1, \ldots, n\}$ as the set of vertices in the graph, and W as the adjacency matrix of the graph, which can be both weighted and weightless. $w_{ij} \neq 0$ if there is an edge from vertex i to j, and 0 otherwise. For each vertex $i \in V$, we need to determine the node's partition. We name the partitions as partition #0 and partition #1 for convenience.

Since the graph partitioning problem is a minimization problem, we can model it as an Integer Programming model. Our goal is to minimize the sum of the penalties attained for each edge cut. When the graph is weightless, penalties associated with cutting each edge is equal and equal to 1. On the other hand, if the graph is weighted, the each penalty for cutting an edge is the weight of that edge.

The first thing that should be done when building an Integer Programming model is to decide on the decision variables. In this problem, for each edge from vertex i to j, we need to decide whether to cut it or not. For each $i, j \in V$ and $w_{ij} \neq 0$, we define x_{ij} as a decision variable that is 1 if an edge will be cut between i and j, and 0 otherwise. The variable is defined only when $w_{ij} \neq 0$ because the edge i to j must be present before actually cutting it. Moreover, we need to decide on the partition of each vertex. We can denote the partition of vertex i as p_i. p_i can be either 0 or 1

On top of these, it is worth mentioning that we will use only the lower triangle of the adjacency matrix because the adjacency matrix is symmetric, and using only one half of the matrix is enough to build the model.

The objective function of this problem will be the sum of all $w_{ij} \times x_{ij}$'s.

6.3.1.1. Sets.

- $V = \{1, \ldots, n\}$: the set of vertices.

6.3.1.2. Parameters.

- $w_{ij} = \begin{cases} > 0, & \text{if vertex } i \text{ and } j \text{ are connected} \\ 0, & \text{if vertex } i \text{ and } j \text{ are not connected} \end{cases}, \forall i, j \in V, i < j$

6.3.1.3. Decision Variables.

- $x_{ij} = \begin{cases} 1, & \text{if the edge between vertex } i \text{ and } j \text{ is "cut"} \\ 0, & \text{the edge is not touched, vertex } i \text{ and } j \text{ are connected} \end{cases}, \forall i, j \in V, w_{ij} \neq 0$
- $p_i = \begin{cases} 1, & \text{if vertex } i \text{ is in partition } \#1 \\ 0, & \text{if vertex } i \text{ is in partition } \#0 \end{cases}, \forall i \in V$

6.3.1.4. Objective Function.

- minimize $\sum_{i=2}^{n} \sum_{j \in V | w_{ij} \neq 0} w_{ij} \times x_{ij}$

In order for this model to work, we have to define the constraints of a model:

(i) If nodes i and j are connected (if $x_{ij} = 0$), they have to be in the same partition. ($p_i = p_j$)

(ii) Partition sizes must be equal.

So, with everything set, here is the summary of our integer programming model:

$$\text{minimize} \quad \sum_{i=2}^{n} \sum_{j \in V | w_{ij} \neq 0} w_{ij} \times x_{ij}$$

subject to

$$p_i - p_j \leq x_{ij} \qquad \forall i, j \in V, w_{ij} \neq 0, \text{ (Adjacent nodes are in the same partition)},$$

$$p_i - p_j \geq -x_{ij} \qquad \forall i, j \in V, w_{ij} \neq 0, \text{ (Adjacent nodes are in the same partition)},$$

$$\sum_{i=1}^{n} p_i = \left\lfloor \frac{n}{2} \right\rfloor \qquad \text{(Equal partition sizes)},$$

$$x_{ij} = \{0, 1\} \qquad \forall i, j \in V, w_{ij} \neq 0, \text{ (Binary variable constraint)},$$

$$p_i = \{0, 1\} \qquad \forall i \in V, \text{ (Binary variable constraint)}$$

7. IMPLEMENTATION AND TESTING

7.1. Implementation

The integer programming model for the weightless graph partitioning is programmed in Python using Gurobi's Python API (`gurobipy` library). Python is preferred in this project because of its ease of use, support for Matrix Market File (.mtx file) I/O using `scipy` library, and the availability of great graph visualization tools with the help of `igraph` and `matplotlib` libraries.
The repository of the software can be obtained from [17].

7.2. Testing

The performance of the program is tested with various graphs on a MacBook Pro with the following hardware:

Processor Name Quad-Core Intel Core i5
Processor Speed 1.4GHz
Number of Processors : 1
Total Number of Cores : 4
L2 Cache (per Core) 256 KB
L3 Cache 6 MB
Hyper-Threading Technology Enabled
Memory 16 GB

The results of the test is discussed in Section 8.

7.3. Deployment

To be able to run the program, the user should obtain a license from Gurobi's website, which is free for academic use. After installing the license in a computer, the program's repository can be cloned with the following command:

```
git clone
https://github.com/araldortogul/CMPE492_Partitioning_Undirected_Graphs.git
```

The required Python libraries are listed in `requirements.txt`, and they can be downloaded by typing the command:

```
pip install -r requirements.txt
```

inside the project repository.

After cloning, the program can be run by opening a terminal in the repository and typing

```
./graph_partitioner.py
```

Here is the usage of `graph_partitioner`:

```
usage: ./graph_partitioner.py [-h] [-i INPUT FILE DIRECTORY] [-r]
                              [-n NUMBER OF RANDOM GRAPHS TO PARTITION]
                              [-s SIZE OF THE RANDOM GRAPHS TO PARTITION]
                              [-d DENSITY OF THE RANDOM GRAPHS TO PARTITION]
                              [-l LOG FILE DIRECTORY] [-w] [-p]
                              [-pDir PLOTDIR] [--version]

Bipartition undirected graphs with Gurobi.

options:
  -h, --help            show this help message and exit
  -i , --input INPUT FILE DIRECTORY
                        Specify the input file directory for your graph. It MUST be a
    Matrix Market File (.mtx), and MUST NOT include any negative numbers in the
```

```
    adjacency matrix. Graph partitioner works both with a weighted adjacency matrix
    and an unweighted adjacency matrix.
14
15  -r, --random          Provide this flag if you want to experiment with random Erdos-
    Renyi graphs. If you also provide -i argument,the program will first partition the
     graph in the input file, then partition the random graphs.
16
17  -n, --number NUMBER OF RANDOM GRAPHS TO PARTITION (Default: 1)
18                        Number of random graphs to partition. Need to be specified if
    -r is given.
19
20  -s, --size SIZE OF THE NUMBER OF RANDOM GRAPHS TO PARTITION (Default: 10)
21                        Size (number of vertices) of the random graphs. Need to be
    specified if -r is given.
22
23  -d, --density DENSITY OF THE RANDOM GRAPHS TO PARTITION (Default: 0.1)
24                        Density of the random graphs. Need to be specified if -r is
    given.
25
26  -l, --log LOG FILE DIRECTORY
27                        Provide a log file name to log Gurobi's optimization process.
28
29  -w, --weighted        Provide this flag if you want your random graphs to be
    weighted. You don't need this flag for your input graph (-i).
30
31  -p, --plot            Provide this flag if you want your graphs to be plotted.
32
33  -pDir, --plotDir PLOTDIR (Default: The directory of the program.)
34                        Provide a directory to store the plots. Needs -p flag
35
36  --version             show program's version number and exit
37
38 All rights reserved. For more info about the partitioning algorithm, please refer to
    the project report found in
39    https://github.com/araldortogul/CMPE492_Partitioning_Undirected_Graphs
```

8. RESULTS

8.1. Small, Weighted Graphs

The software is able to partition weighted, small graphs (size between 20 and 30) very quickly. According to the average runtimes in Figure 8.1, higher density results in higher computation time. This is expected because the IP model has $|V|+|E|$ decision variables and $1+|V|+3\times|E|$ constraints.

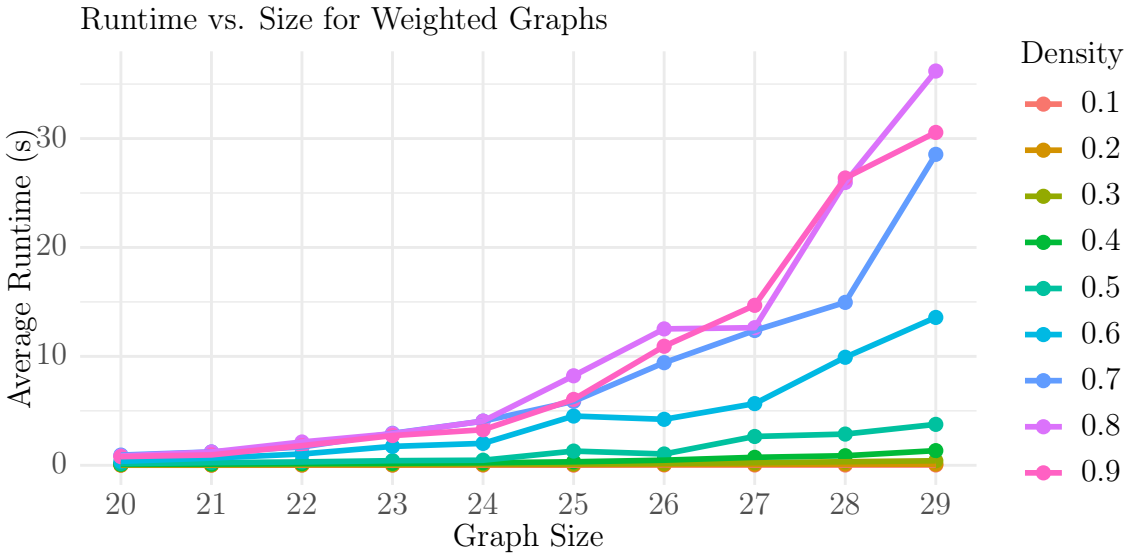


Figure 8.1. Runtimes of Small, Weighted Graphs with Different Densities

8.2. Small, Weightless Graphs

The software is also able to partition weightless, small graphs of all densities in reasonable time. However, unlike the weighted version, high density does not always lead to high runtime. According to the runtimes in Figure 8.2, densities 0.6 and 0.7 took the most time to find the best solution. This may be due to the fact that very dense graphs (density $>$ 0.8) might be predictable in terms of the number of cuts. On the contrary, in the weighted version of the problem, the weights of the individual edges have a significant impact on the outcome, so the edges should be handled more carefully in the weighted version.

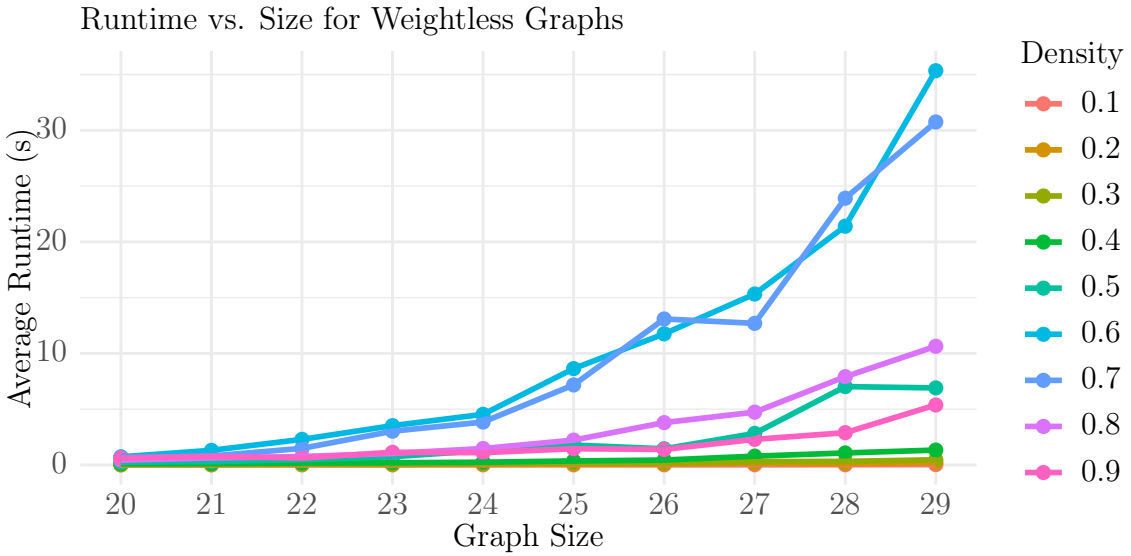


Figure 8.2. Runtimes of Small, Weightless Graphs with Different Densities

8.3. Graph between 50 and 100 Vertices

Dense graphs with more than 50 vertices become too complex for the software to find a solution quickly. The software is tested on sparse graphs between 75 and 100 vertices, and Figure 8.3 and Figure 8.4 display the runtimes for the weighted and weightless versions of the problem.

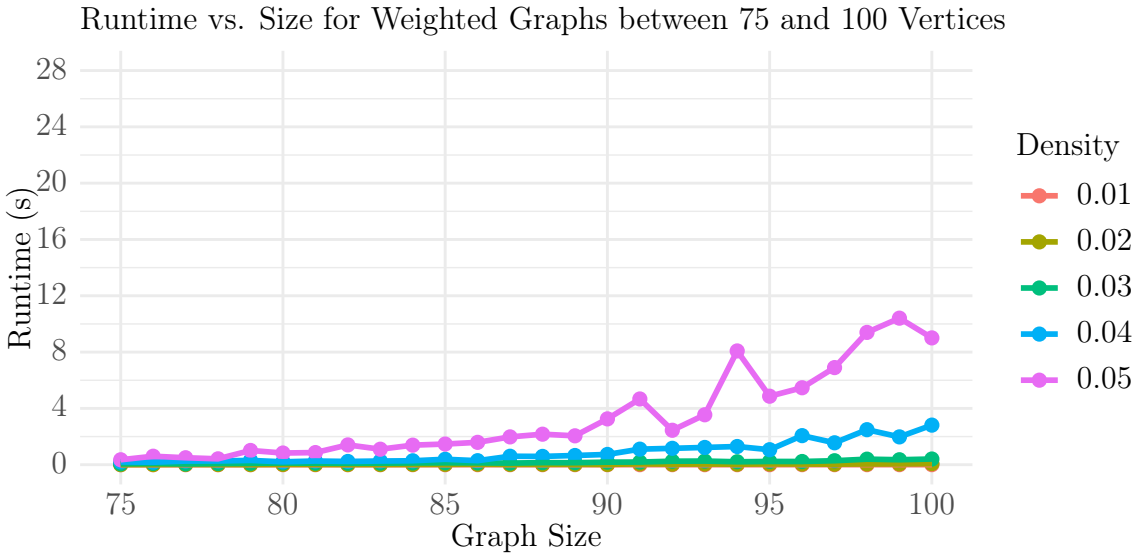


Figure 8.3. Runtimes of Weighted, Sparse Graphs between 75 and 100 Vertices

One major finding from the data in Figure 8.3 and Figure8.4 is that the weightless

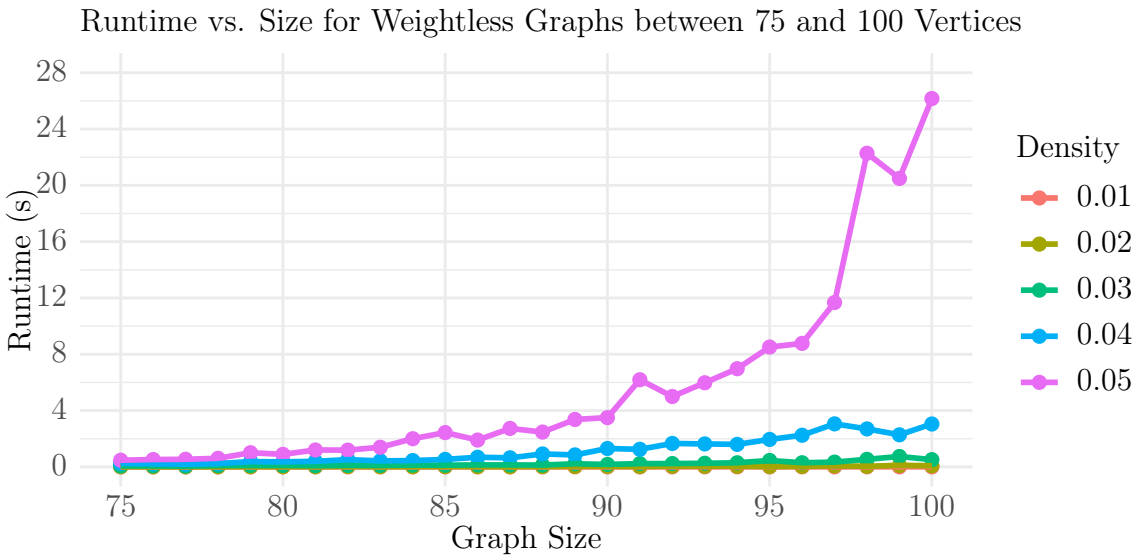


Figure 8.4. Runtimes of Weightless, Sparse Graphs between 75 and 100 Vertices

version of the problem takes slighlty more time to find a solution than the weighted version.

8.4. Large, Sparse Graphs

For large graphs (size $>$ 100), the software is tested on very low densities. Figure 8.5 shows how the average runtime of the algorithm changes when the graph size increases for densities 0.01, 0.02 and 0.03. Graphs with 0.03 density can be solved under 1 minute if the size of the graph is less than 150. Graphs with 0.02 density are slightly easier to solve, and the computation takes less than 1 minute for sizes less than 180. Finally, graphs with 0.01 density can be solved up to size 250.

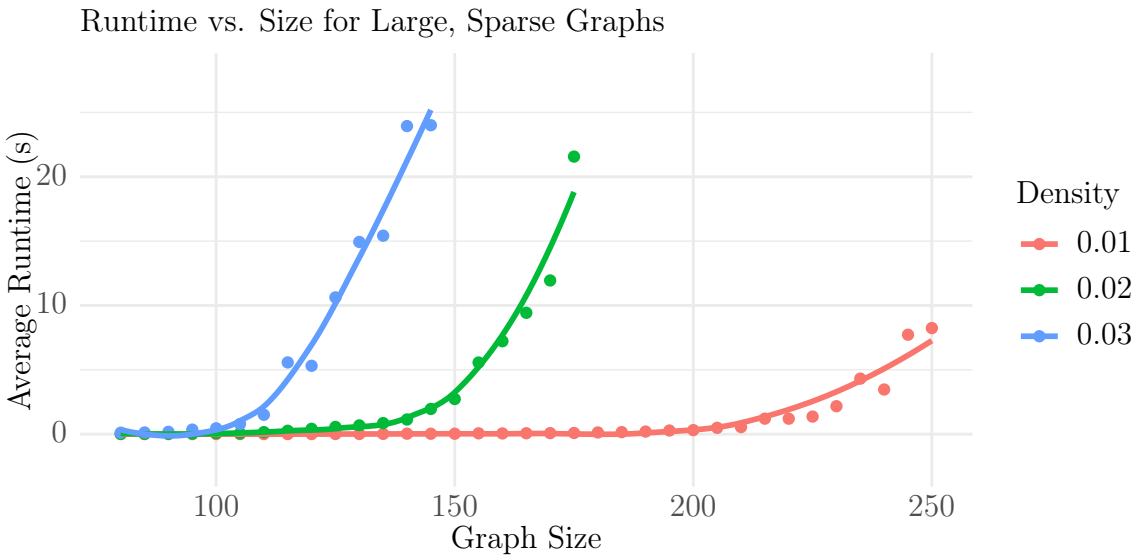


Figure 8.5. Runtimes of Large, Sparse Graphs with Different Densities

8.5. Sample Graphs and their Partitions

8.5.1. Graphs from SuitSparse Matrix Collection

In this section, sample sparse matrices are taken from SuiteSparse Matrix Collection [18].

Here are some examples of solved cases:

8.5.1.1. Graph 1.

- Name: Pajek/GD96_c
- Description: Pajek network: Graph Drawing contest 1996
- Size: 65
- Edges: 250

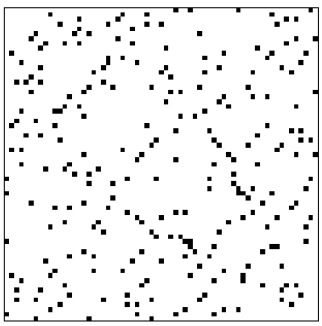


Figure 8.6. Adjacency Matrix of GD96_c

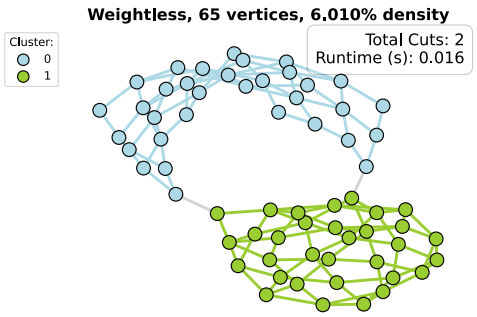


Figure 8.7. Solution for GD96_c

8.5.1.2. Graph 2.

- Name: HB/dwt_59
- Description: Symmetric Connection Table from Dtnsrdc, Washington
- Size: 59
- Edges: 267

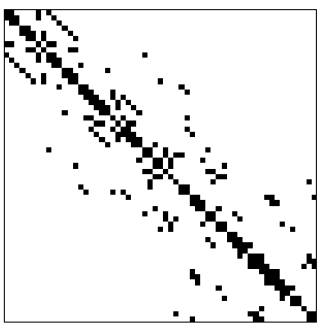


Figure 8.8. Adjacency Matrix of dwt_59

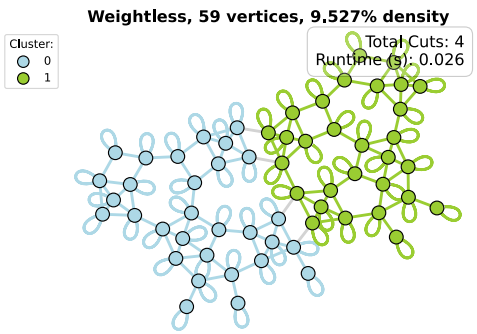


Figure 8.9. Solution for dwt_59

8.5.2. Weightless Erdős–Rényi Graphs

Here are some example solutions for weightless random graphs generated by the Erdős–Rényi model.

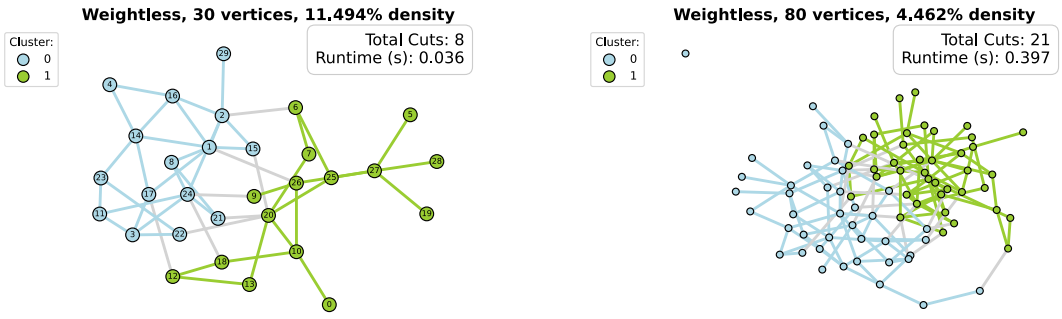


Figure 8.10. Solutions of Randomly Generated Weightless Graphs

8.5.3. Weighted Erdős–Rényi Graphs

Here are some example weighted random graphs generated by the Erdős–Rényi model. Weights of the edges are uniformly distributed between 1 and 5.

In the plots, edges with higher weghts are drawn thicker.

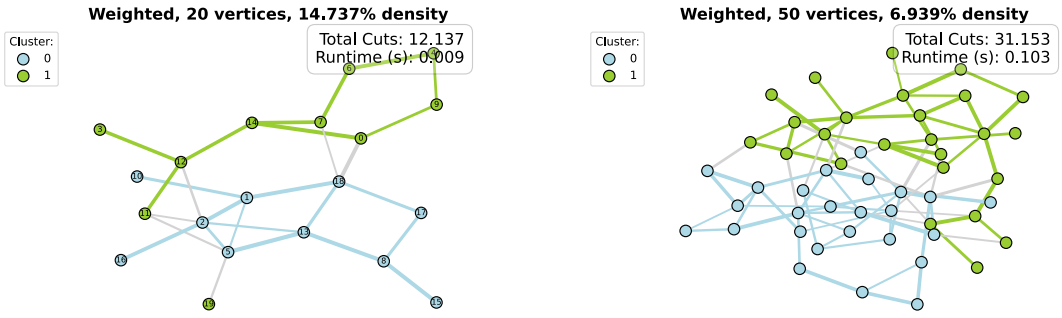


Figure 8.11. Solutions of Randomly Generated Weighted Graphs

9. CONCLUSION

The integer programming model developed for the graph partitioning problem can handle the bipartitioning of small graphs and sparse, large graphs. It gives promising results and can be further improved by generalizing the problem to partitioning the graph into more than 2 subgraphs. The program may be used for partitioning data that is sparsely connected.

This project helped me refine my operations research knowledge as well as my software developing skills. Moreover, it shows how approaching in an interdisciplinary fashion helps finding unique solutions to engineering problems.

REFERENCES

1. Chung, F., *Spectral Graph Theory*, Conference Board of Mathematical Sciences, American Mathematical Society, 1997, https://books.google.com.tr/books?id=4IK8DgAAQBAJ.

2. Feldmann, A., "Fast Balanced Partitioning Is Hard Even on Grids and Trees", *Theoretical Computer Science*, Vol. 485, 11 2011.

3. Buluç, A., H. Meyerhenke, I. Safro, P. Sanders and C. Schulz, *Recent Advances in Graph Partitioning*, pp. 117–158, Springer International Publishing, 2 2016, https://doi.org/10.1007/978-3-319-49487-6_4.

4. Karisch, S., F. Rendl and J. Clausen, "Solving Graph Bisection Problems with Semidefinite Programming", *INFORMS Journal on Computing*, Vol. 12, pp. 177–191, 08 2000.

5. Armbruster, M., "Branch-and-Cut for a Semidefinite Relaxation of Large-scale Minimum Bisection Problems", *Fakultät für Mathematik der Technischen Universität Chemnitz*, 06 2007.

6. Sellmann, M., N. Sensen and L. Timajev, "Multicommodity Flow Approximation Used for Exact Graph Partitioning", G. Di Battista and U. Zwick (Editors), *Algorithms - ESA 2003*, pp. 752–764, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

7. Sensen, N., "Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows", F. M. auf der Heide (Editor), *Algorithms — ESA 2001*, pp. 391–403, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

8. Brunetta, L., M. Conforti and G. Rinaldi, "A branch-and-cut algorithm for the equicut problem", *Mathematical Programming*, Vol. 77, pp. 243–263, 08 1997.

9. Ferreira, C., A. Martin, C. Souza, R. Weismantel and L. Wolsey, "The node capacitated graph partitioning problem: A computational study", *Mathematical Programming*, Vol. 81, pp. 229–256, 01 1998.

10. Lisser, A. and F. Rendl, "Rendl, F.: Graph partitioning using linear and semidefinite programming. Math. Program. Ser. B 95(1), 91-101", *Mathematical Programming*, Vol. 95, pp. 91–101, 01 2003.

11. Hager, W. W., D. T. Phan and H. Zhang, "An exact algorithm for graph partitioning", *Mathematical Programming*, Vol. 137, No. 1, pp. 531–556, 2013, https://doi.org/10.1007/s10107-011-0503-x.

12. Hager, W. W. and Y. S. Krylyuk, "Graph Partitioning and Continuous Quadratic Programming", *SIAM J. Discret. Math.*, Vol. 12, pp. 500–523, 1999, https://api.semanticscholar.org/CorpusID:14431056.

13. Wolsey, L. A., *Branch and Bound*, chap. 7, pp. 113–138, John Wiley & Sons, Ltd, 2020, https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119606475.ch7.

14. Wolsey, L. A., *Cutting Plane Algorithms*, chap. 8, pp. 139–166, John Wiley & Sons, Ltd, 2020, https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119606475.ch8.

15. Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual", , 2023, https://www.gurobi.com.

16. "Matrix Market: File Formats", , 8 2013, https://math.nist.gov/MatrixMarket/formats.html.

17. Dörtoğul, A., "Graph Partitioner", https://github.com/araldortogul/CMPE492_Partitioning_Undirected_Graphs.

18. Davis, T. A. and Y. Hu, "The University of Florida Sparse Matrix Collection", *ACM Trans. Math. Softw.*, Vol. 38, No. 1, dec 2011, https://doi.org/10.1145/2049662.2049663.