COMPUTATIONAL OPTIMIZATION TECHNIQUES FOR GRAPH PARTITIONING

A Dissertation

by

SCOTT PARKER KOLODZIEJ

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Timothy A. Davis
Committee Members,	Dezhen Song
	Shaoming (Jeff) Huang
	Erick Moreno-Centeno
	William W. Hager
Head of Department,	Dilma Da Silva

August 2019

Major Subject: Computer Science

ABSTRACT

Partitioning graphs into two or more subgraphs is a fundamental operation in computer science, with applications in large-scale graph analytics, distributed and parallel data processing, and fill-reducing orderings in sparse matrix algorithms. Computing balanced and minimally connected subgraphs is a common pre-processing step in these areas, and must therefore be done quickly and efficiently. Since graph partitioning is NP-hard, heuristics must be used. These heuristics must balance the need to produce high quality partitions with that of providing practical performance. Traditional methods of partitioning graphs rely heavily on combinatorics, but recent developments in continuous optimization formulations have led to the development of hybrid methods that combine the best of both approaches.

This work describes numerical optimization formulations for two classes of graph partitioning problems, edge cuts and vertex separators. Optimization-based formulations for each of these problems are described, and hybrid algorithms combining these optimization-based approaches with traditional combinatoric methods are presented. Efficient implementations and computational results for these algorithms are presented in a C++ graph partitioning library competitive with the state of the art. Additionally, an optimization-based approach to hypergraph partitioning is proposed.

DEDICATION

To Claire

ACKNOWLEDGMENTS

I sincerely and especially wish to thank my wife, Elizabeth, and our daughter, Claire, for their support, patience, and sacrifice.

I also thank my parents, Saundra and Edwin Kolodziej, for their lifelong support and encouragement.

I would like to thank my doctoral research advisor, Dr. Tim Davis, for all of his help and support. Without his guidance, this research would never have happened.

I would also like to thank my doctoral research committee: Dr. Dezhen Song, Dr. Jeff Huang, Dr. Erick Moreno-Centeno, and Dr. Bill Hager. I have appreciated their feedback and suggestions.

I am also grateful to Dr. Dilma Da Silva for her career mentoring and support.

Through coursework and other interactions, I also owe a great deal of thanks to a number of other faculty in the department: Dr. Ricardo Bettati, Dr. Thomas Ioerger, Dr. Andrew Jiang, Dr. John Keyser, Dr. J. Michael Moore, Dr. Daniel Ragsdale, Dr. Scott Schaefer, Dr. Dylan Shell, Dr. Aakash Tyagi, Dr. Hank Walker, and Dr. Tiffani Williams.

And to all of the many friends I made along the way, a few of whom I list here: Mohsen Aznaveh, Hsin-min "Jasmine" Cheng, Chieh "Jay" Chou, Jory Denny, Raniero Lara-Garduno, Cassandra Oduola, Seth Polsley, Read Sandström, Traci Sarmiento, Wissam Sid-Lakhdar, Timmie Smith, Nathan Thomas, Pulakesh Upadhyaya, Diane Uwacu, and Stephanie Valentine.

I am also grateful for the assistance and support of the following department and university staff: Taffie Behringer, Karrie Bourquin, Dave Cote, Sheila Dotson, Brad Goodman, Sarah Morgan, David Ramirez, Elena Rodriguez, Kelly Ford Sandström, Krista Simmons, Valerie Sorenson, Bruce Veals, Carly Veytia, and, of course, Kathy Waskom.

And thank you to Reva Power, my high school computer science teacher. Her introduction to computer science has carried me further than I could have imagined as a high school freshman.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Timothy A. Davis, Dezhen Song, and Jeff Huang of the Department of Computer Science and Engineering at Texas A&M University, Professor Erick Centeno-Moreno of the Department of Industrial and Systems Engineering at Texas A&M University, and Professor William Hager of the Department of Mathematics at the University of Florida.

Chapter 2 is largely based on a jointly-authored paper by Dr. Timothy A. Davis, Dr. William W. Hager, Dr. Nuri S. Yeralan, and me [1]. The initial algorithm development was done by Dr. Yeralan, and completing the software and preparing it for publication and release was my starting point in my doctoral studies.

All other work conducted for this dissertation was completed independently.

Funding Sources

Graduate study was supported by the following funding sources:

- Research startup funding for Dr. Timothy Davis through the Department of Computer Science and Engineering at Texas A&M University.

- A Graduate Teaching Fellowship from the College of Engineering.

- National Science Foundation grants 1115297 and 1514406.

- Corporate gift funding from Intel Corporation and Nvidia Corporation.

NOMENCLATURE

AMD	Approximate Minimum Degree
CPU	Central Processing Unit
FM	Fiduccia-Mattheyses (Algorithm)
G_x	Gain metric for moving to part X
G_y	Gain metric for moving to part Y
γ	Penalty parameter
GPU	Graphics Processing Unit
∇f	The gradient of the function f
$\nabla_x f$	The gradient of the function f with respect to the variable x
$\nabla \tilde{f}$	An approximation of the gradient of the function f
$\boldsymbol{I}$	Identity matrix
KKT	Karush-Kuhn-Tucker (Conditions)
K-L	Kernighan-Lin (Algorithm)
LP	Linear Program (or Programming)
MCA	Mountain Climbing Algorithm
MILP	Mixed-Integer Linear Program (or Programming)
ψ	Imbalance metric
QCQP	Quadratically-Constrained Quadratic Program (or Programming)
QP	Quadratic Program (or Programming)
RCM	Reverse Cuthill-McKee (Ordering)
VLSI	Very Large Scale Integration

w_i	Vertex weight for vertex i
$w_{i,j}$	Edge weight for the edge (i, j)
w_{max}	Maximum vertex weight
w_{min}	Minimum vertex weight
x_i	Affinity of vertex i for part X
y_I	Affinity of vertex i for part Y
$\mathbf{0}_{m,n}$	An all-zero matrix with m rows and n columns

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE Page

1. INTRODUCTION TO GRAPH PARTITIONING

1.1 Graph Partitioning

Graph partitioning is the subfield of computer science and mathematics concerned with partitioning graphs into two or more partitions, or subgraphs. The starting graph can be either a regular graph with vertices and edges (with each edge connecting exactly two vertices) or a hypergraph (where each hyperedge can connect two or more vertices).

Graph partitioning problems take many forms. The most common is the balanced edge cut, where a minimal number of edges are cut to partition the graph into two subgraphs with roughly the same number of vertices.

1.2 Preliminaries

We begin with an overview of various types of graphs and graph partitioning problems. Much of this section and associated definitions are adapted from Bichot and Siarry's *Graph Partitioning* [2].

1.2.1 Graphs

Within discrete mathematics, a graph is a discrete structure made up of a set of unique *vertices* (also referred to as *nodes*) and *edges* which each connect exactly two vertices, representing some kind of relationship between those two vertices. Examples include road networks (with cities or addresses as vertices and roads as edges), distributed systems (with compute nodes as vertices and network connections as edges), and social networks (with individuals as vertices and social connections as edges).

Here we define several different types of graphs:[2]

Definition 1.2.1. (Graph) – Given a set of vertices V and a set of edges $E \subseteq \{e = (u, v) \in V \times V\}$, the pair $G = (V, E)$ defines a graph.

Definition 1.2.2. (Undirected Graph) – A graph whose edges are non-directional, i.e. the edge $e = (u, v) \in E$ implies the existence of $e = (v, u) \in E$.

Definition 1.2.3. (Directed Graph) – A graph whose edges are directional, i.e. the edge $(u, v) \in E$ is unique from the edge $(v, u) \in E$. Such directed edges are often referred to as arcs.

Generally, loop- or self-edges (i.e. edges that originate and terminate at the same vertex, $(u, u) \in E$) are ignored in the context of graph partitioning. For the bulk of this work, a graph is assumed to have no self-edges and at most one undirected edge connecting each pair of vertices. This is known as a *simple graph*:

Definition 1.2.4. (Simple Graph) – An undirected graph with no self-edges and at most one edge connecting each pair of vertices.

Additionally, graphs can have numeric *weights* associated with their vertices and edges. These can be used to represent properties about the vertices and edges, such as distances in a road network. A graph with such weights is said to be a *weighted graph*. In this work, graphs are assumed to be simple graphs with weighted edges and vertices, making them *simple weighted graphs*.

To further define the connectivity of vertices via edges, we introduce the concept of *adjacency* and that of the *neighborhood* of a vertex:

Definition 1.2.5. (Adjacency) – Vertices connected by an edge are said to be adjacent. In $(u, v) \in E$, vertices u and v are adjacent to one another.

Definition 1.2.6. (Neighborhood) – The set of all vertices adjacent to a vertex v_i defines the neighborhood $N(i)$.

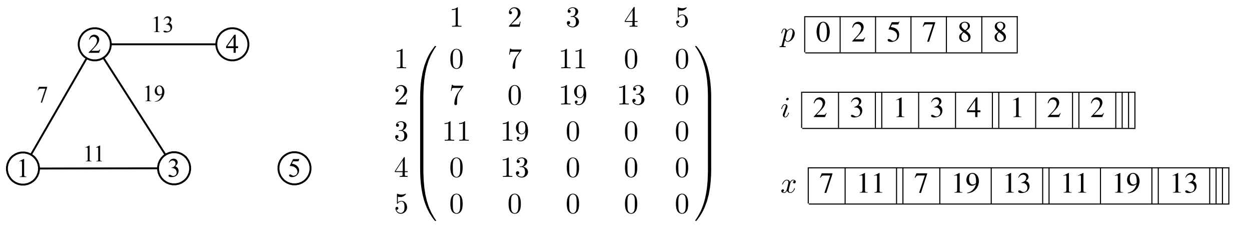
1.2.2 Hypergraphs

A related generalization of a graph where edges can connect more than two vertices is known as a *hypergraph*, with such edges being called *hyperedges* or *nets*.

Definition 1.2.7. (Hypergraph) – A set of vertices V and a set of hyperedges $E \subseteq \{e | e \in V^{|V|}\}$.

1.2.3 Graph Representations

Graphs are abstract data structures and can be represented in many different forms. We discuss here the two most relevant to graph partitioning: the adjacency matrix and compressed sparse column formats (see Figure 1.1).



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 7 & 11 & 0 & 0 \\ 7 & 0 & 19 & 13 & 0 \\ 11 & 19 & 0 & 0 & 0 \\ 0 & 13 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

p | 0 | 2 | 5 | 7 | 8 | 8 |

i | 2 | 3 || 1 | 3 | 4 || 1 | 2 || 2 |||

x | 7 | 11 || 7 | 19 | 13 || 11 | 19 || 13 |||

Figure 1.1: A graph, its adjacency matrix, and its compressed sparse column representation. Note that because this graph is so small, the compressed sparse column representation provides only a modest space savings (22 compared with 25 elements).

1.2.3.1 Adjacency Matrix

The adjacency matrix of a graph is a square matrix that represents each vertex as a row and column of the matrix, and each edge as an entry in the matrix. For a graph with $n = |V|$ vertices and $nz = |E|$ edges, its adjacency matrix is $n \times n$ with nz non-zero entries. An edge with weight w connecting vertices u and v is represented as $A_{u,v} = w$. Note that undirected graphs are represented as a symmetric matrix ($A_{u,v} = A_{v,u}$), while directed graphs are generally unsymmetric ($A_{u,v}$ need not equal $A_{v,u}$).

1.2.3.2 Compressed Sparse Column Format

For matrices with very few non-zero entries, an adjacency matrix is frequently an inefficient representation. In practice, most graphs have far more vertices than the average degree of a vertex, lead to very sparse adjacency matrices. Thus, sparse matrices (and by extension most graphs) require a compressed storage format. Here we discuss one such format: compressed sparse column (CSC) form.

In compressed sparse column form, three vectors are maintained: $p \in \mathbb{N}^{(n+1)}$ for column pointers, $i \in \mathbb{N}^{nz}$ for row indices, and $x \in \mathbb{R}^{nz}$ for matrix entry values. A separate vertex weight array $\boldsymbol{w} \in \mathbb{R}^{+n}$ is used if vertices have weights associated with them [3].

In short, the space required for storing a symmetric matrix in CSC form is $O(n+2nz)$ compared with $O(n^2)$ for an uncompressed adjacency matrix. For matrices where $nz \ll n^2$, such as in the case of sparse matrices, this format generally provides significant storage savings. Because of this,

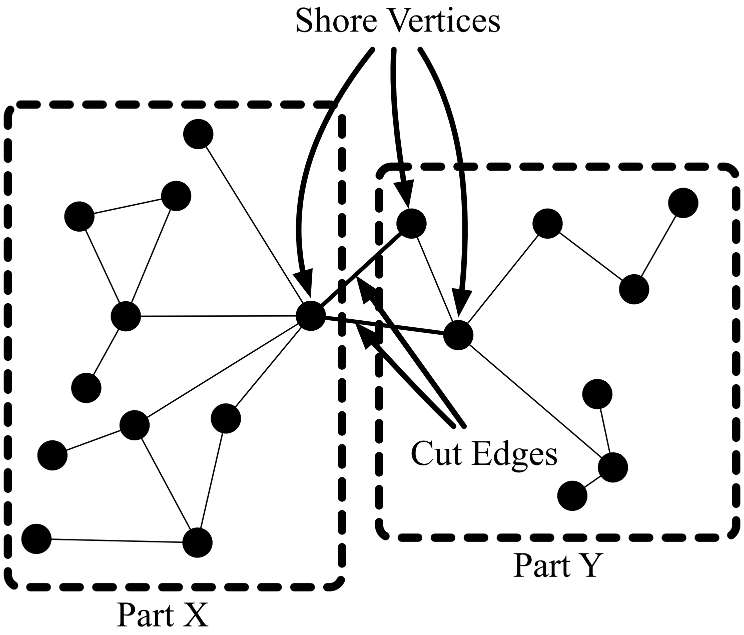


Figure 1.2: Edge cut

all graphs presented in this work are stored in CSC form and operated on in that format.

Several drawbacks exist for storing a graph in such a format. First, like most compressed formats, it is difficult to modify once created. Row-wise traversals are also significantly less efficient than column-wise traversals. Thus, graphs stored in this format are often treated as immutable and only operated on in a column-wise fashion.

1.3 Graph Partitioning Problems

There are many different ways graphs can be partitioned. This work focuses on two classes of partitioning problems in particular: edge cuts and vertex separators. In this section, we introduce these problems and discuss their challenges, applications, and prior work.

1.3.1 Edge Cuts

Given an undirected graph $G = (V, E)$ with vertices V and edges E, a graph can be said to have an edge cut (or edge separator) if the vertices are separated into two groups V_X and V_Y such that $V = V_X \cup V_Y$ and where V_X and V_Y are disjoint ($V_X \cap V_Y = \emptyset$). The naming of these

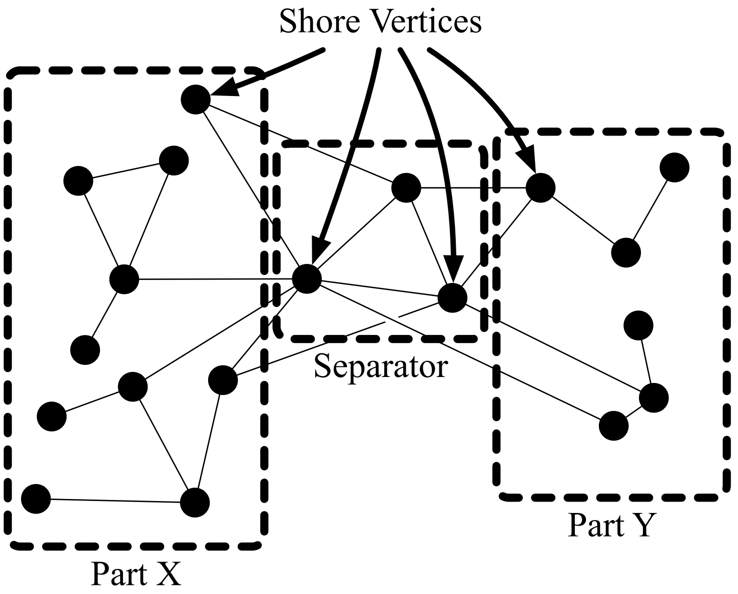


Figure 1.3: Vertex separator

groups is arbitrary, although in some formulations the groups are labeled in order of decreasing size (e.g. $|V_X| > |V_Y|$). These two groups (and in k-way partitioning, k groups) are also referred to as parts. The set of vertices V_X are said to belong to the X part in the partition, and likewise for V_Y belonging to the Y part. The edges that lie on the cut, $E_{cut} = \{(u, v)|u \in V_X, v \in V_Y\}$, exclusively connect the two groups of vertices – that is, if the cut edges were removed, V_X and V_Y would form two disjunct subgraphs.

The vertices located on either side of the cut, $E_{shores} = \{u \in V_X|\exists(u, v), v \in V_Y\}$, are said to be located on the shores of the cut or partition. Alternatively, vertices located on a shore have at least one edge on the cut.

A balanced edge cut is one where the sizes of the groups of vertices are approximately the same size, $|V_X| \approx |V_Y|$. This is usually implemented as a hard limit ($\max\{|V_X|, |V_Y|\} \leq 2 * \min\{|V_X|, |V_Y|\}$) or with some form of penalty function to allow for smaller cuts with some small amount of imbalance.

1.3.2 Vertex Separators

A different kind of partitioning can be created called a vertex separator, or vertex cut, where three groups of vertices are found: V_X and V_Y as in edge cuts, but also V_S, for vertices in the separator. The separator group represent vertices that must be visited in any traversal from part V_X to part V_Y. If the vertices in the separator $v \in V_S$ were removed, V_X and V_Y would from two disjunct subgraphs.

Vertex separator problems are generally not concerned with how many edges are cut, but rather attempt to minimize the number of vertices in the separator. Consequently, edge weights are often irrelevant in vertex separator problems.

1.3.3 Complexity

Both the vertex separator and edge cut problems are NP-hard [4]. While algorithms exist that can guarantee globally optimal cuts or separators, they take exponential time in the worst case. Practical existing methods rely almost exclusively on heuristics to compute minimal cuts and separators.

1.4 Applications

The applications of graph partitioning are as varied as the problem itself. However, listed below are three of the most common applications.

1.4.1 Graph Analytics and Data Partitioning

Analytics and data analysis over increasingly large data sets has driven the development of ever more efficient algorithms to aid in such analysis. However, as data sets continue to grow, outpacing such algorithmic gains, the need for subdividing and partitioning these data sets for analysis has arisen. While many facets of Big Data are unstructured, others such as social networks, genome graphs, brain networks, road networks, and internet graphs are all naturally structured as graphs [5, 6].

The development of GraphBLAS has also enabled the use of high-performance linear algebra in the domain of graph analytics [7, 8]. In this context, graph partitioning serves as a pre-processing step for enabling parallel sparse linear algebra such as sparse matrix-vector multiplication [9].

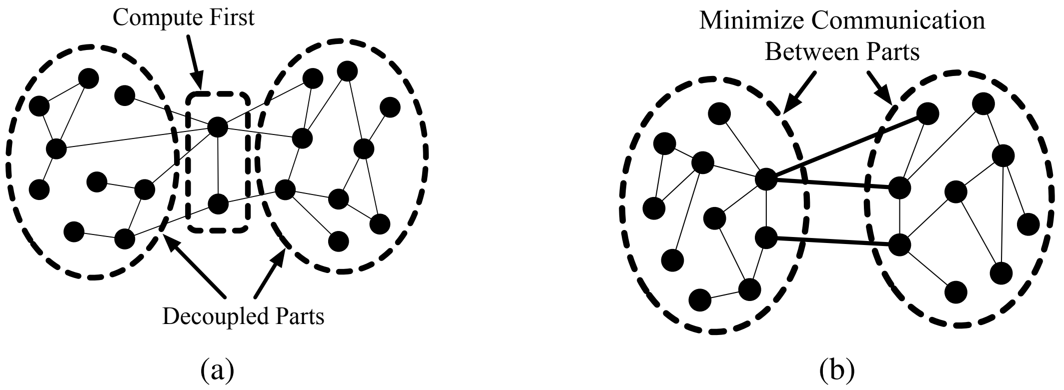


Figure 1.4: Graph partitioning in graph analytics applications.

Examples of how vertex separators (a) and edge cuts (b) are used in graph analytics and data partitioning applications. Vertex separators are used to decouple two large bodies of data, while edge cuts are used for communication minimization tasks.

More generally, vertex separators provide a boundary layer between two decoupled bodies of data in graph analytics tasks where each vertex represents a task that can be computed in any order. In this case, the separator vertices can be computed first, allowing a complete decoupling of the X and Y parts. Edge cuts are used for graph analytics problems that require partitioning to minimize communication between two coupled bodies of data, such as in the case of streaming graphs that update in real-time (see Figure 1.4).

1.4.2 Sparse Matrix Orderings and Nested Dissection

In the solution of sparse symmetric linear systems, the matrix can be reordered to minimize fill-in during a subsequent Cholesky factorization. One technique for computing such a fill-reducing ordering is called nested dissection, where a graph representing the matrix is recursively subdivided [10]. Figure 1.5 shows the impact that a fill-reducing ordering can have on a Cholesky factorization. For very large linear systems, an effective fill-reducing ordering can have a significant impact on the solution tractability.

1.4.3 Parallel and Distributed Computation Load Balancing

Both the edge cut and vertex separator problems can be used to assign work to different cores or cluster nodes in a parallel computation framework. By minimizing the edges cut or separator size,

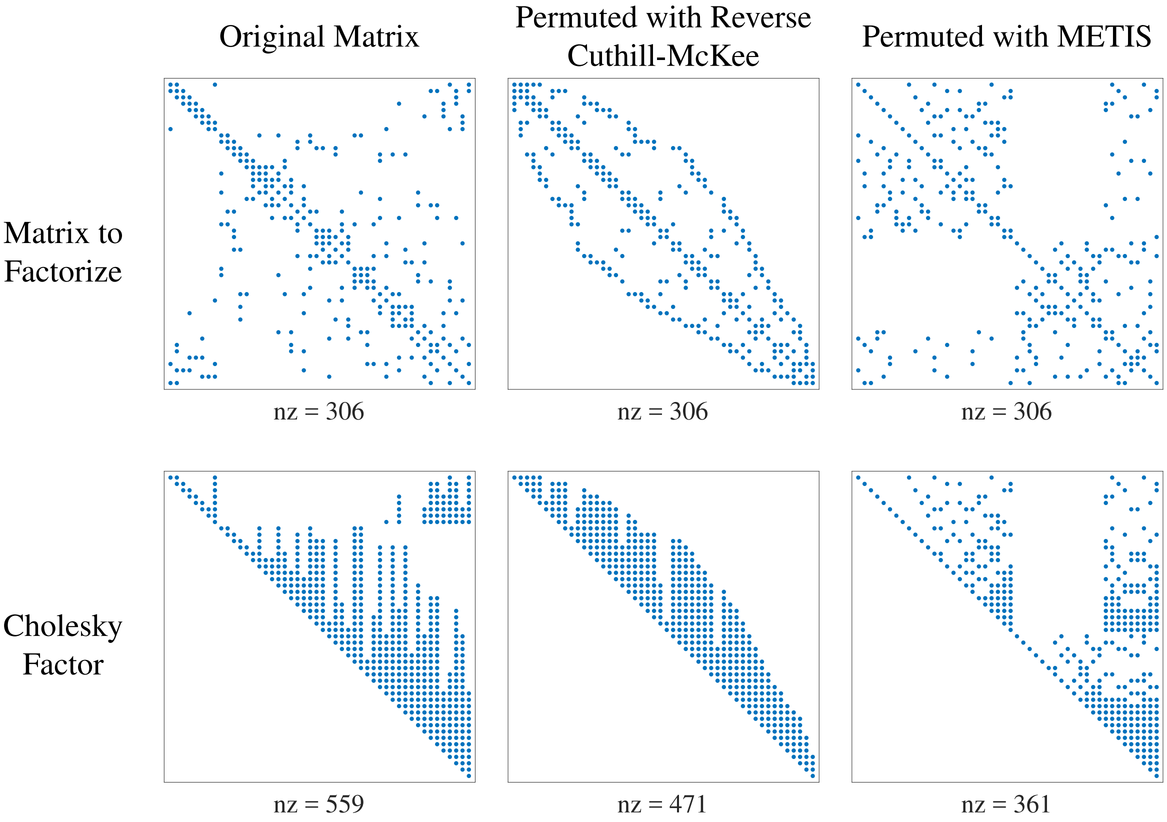


Figure 1.5: Effect of fill-reducing orderings on matrix Pothen/mesh1em1 [11]

The top row (in order) shows the original matrix, the matrix permuted using the reverse Cuthill-McKee ordering, and the same matrix permuted using an ordering derived from a graph partitioning library, METIS [12]. The second row shows each matrix after being factorized using Cholesky factorization. Note that the permuted matrices result in significantly less fill-in (35% less using reverse Cuthill-McKee and 78% less using METIS).

communication between each node can be minimized, and by balancing the sizes of the parts, each compute node can be assigned a roughly equally-sized amount of work. This application requires a specific type of load balancing, as a single node with less work than average is acceptable, but a single node with far more work than average results in a significant underutilization of all other nodes [13, 14].

1.5 Established Methods for Graph Partitioning

1.5.1 Combinatoric Local Search

Some of the first graph partitioning methods involved moving individual vertices to one side of the cut or the other. One such algorithm, proposed by Brian Kernighan and Shen Lin for VLSI circuit layout, computes the net benefit of moving a vertex across the cut. The algorithm then moves vertices accordingly, in such a way as to minimize the number of edges cut [15]. This algorithm runs in $O(n^2 \log n)$ time, where n is proportional to the size of the vertex set, $|V|$.

A refinement of the Kernighan-Lin algorithm, proposed by Charles M. Fiduccia and Robert M. Mattheyses, runs in linear time with respect to the number of vertices, or $O(|V|)$ time [16]. However, at its core, it is a very similar algorithm to Kernighan-Lin, utilizing computed gain values to determine which vertex to move to the other side of the cut. These algorithms could be characterized as greedy, shrinking the edge cut until a locally optimal solution is found. To find a better solution, a vertex move that increases the size of the cut may be required before being able to transition to a solution with a smaller cut. Although modern implementations address this by adding random perturbations to better explore the search space of partitions, this is a key weakness of this class of algorithm.

While both the Kernighan-Lin and Fiduccia-Mattheyses algorithms were developed for use in finding small, balanced edge cuts, they have been adapted for use in finding balanced vertex separators [17], [18]. The general idea is the same, computing gain values for vertices both in the separator and on the shores on each side of the separator and then moving vertices into and out of the separator accordingly.

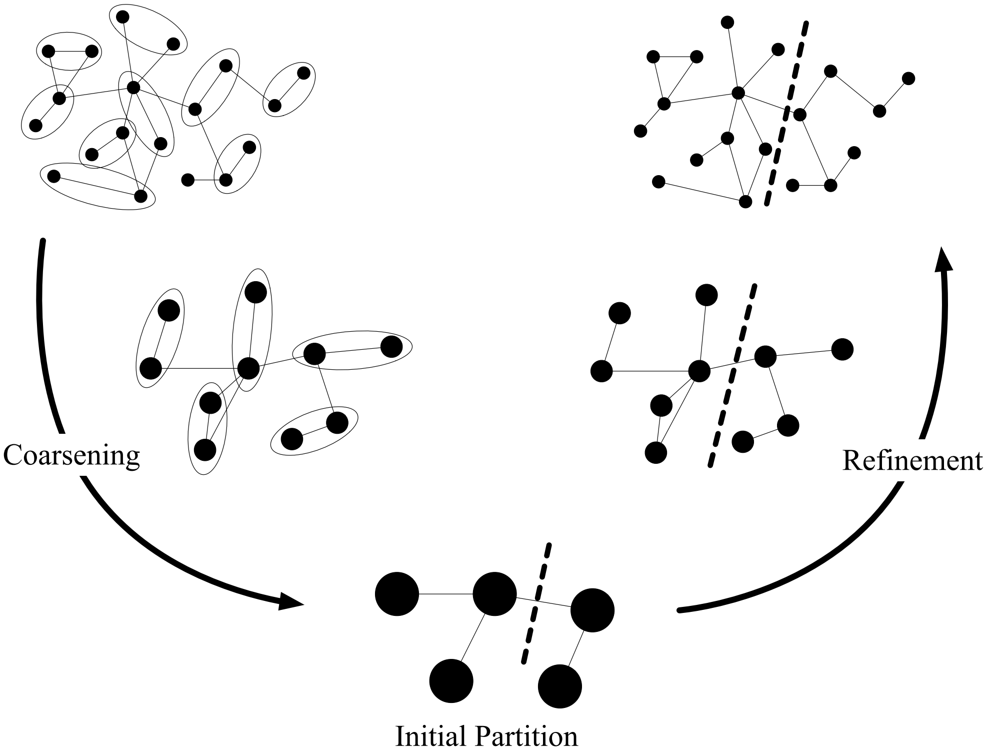


Figure 1.6: Graph coarsening and refinement

Multilevel algorithms first coarsen input graphs until they are of a reasonable size. An initial partition is computed, and that partition is then repeatedly projected back onto the larger graphs, often with minor readjustments.

1.5.2 Graph Coarsening and Refinement

Despite the effectiveness of combinatoric local search on finding high quality edge cuts and vertex separators, this approach becomes less effective for very large graphs. To address this, multilevel approaches were developed to reduce the graph to a manageable size [19, 20, 12]. Given a large graph, vertices are matched with other vertices using a variety of matching strategies, and then coarsened into super-vertices. This process is repeated until the graph is of a more tractable size, and an initial solution is computed on the coarsened graph. Then, the process is reversed, with the initial solution projected onto the less-coarsened (refined) graph, with periodic optimizations to increase the quality of the projected solution. Eventually, the solution is projected onto the original graph, generally yielding a high quality solution to the original problem (see Figure 1.6).

This entire process has a moderate computational complexity and is generally non-trivial to implement, but it is almost universally more efficient than attempting to solve the partitioning problem on the original, very large graph.

1.5.3 Spectral and Eigenvector Partitioning

Spectral methods have also been proposed for computing edge cuts and vertex separators [21]. The second smallest eigenvalue of the Laplacian of a graph can be used to provide a balanced partitioning, in addition to providing bounds on the size of the optimal separator. This approach has been shown to be related to the quadratic programming formulation proposed by Hager and Krylyuk [22].

1.5.4 Normalized Cut and Image Segmentation

One application area that has spawned a number of heuristics for graph partitioning is image segmentation. The well-known normalized cut was first proposed with this application in mind [23]. The normalized cut takes into account the degree of connectivity (or associativity) of vertices when determining the optimal cut. This helps to avoid edge cuts that are highly unbalanced. Graph clustering methods (generally with more than two parts) can be used to segment an image into relevant groups of pixels by brightness, color, or some other metric.

The definition of the normalized cut metric is shown below:

$$Ncut(X,Y) = \frac{cut(X,Y)}{assoc(X,V)} + \frac{cut(X,Y)}{assoc(Y,V)}$$

where $cut(X,Y) = \sum_{u \in X, v \in Y} w(u,v)$ and $assoc(X,V) = \sum_{u \in X, t \in V} w(u,t)$. Note that the normalized cut is a measure of cut quality, not an algorithm unto itself, but it can be paired with existing methods such as eigenvector decomposition to yield high quality cuts.

1.5.5 Network Flow Algorithms

Another established method for finding partitions of graphs is to use a max-flow min-cut algorithm [24]. While such algorithms allow for weighted edges, they have one principle disadvantage: they do not account for balanced partitions. For this reason, they are largely not relevant to this research, as the targeted application areas require some degree of balance in the computed partitions.

1.5.6 Existing Software Libraries

A number of existing graph partitioning libraries exist which employ numerous algorithms, heuristics, and other strategies. A subset is listed below with their relevant features:

1. **METIS** utilizes graph coarsening and refinement for scalability, and Fiduccia-Mattheyses-style combinatoric search techniques [12]. It also comes in several versions, including ParMETIS (for parallelized graph partitioning) [25, 26], hMETIS (for hypergraph partitioning) [27], and MT-METIS (for use with OpenMP) [28].

2. **Chaco**, like METIS, uses graph coarsening and refinement with FM-style combinatoric search. Chaco also utilizes spectral partitioning methods [20].

3. **SCOTCH** is a very diverse library, providing sequential and multi-threaded implementations of sparse matrix ordering and graph partitioning methods [29]. There is also a parallel distributed version using the MPI interface, PT-SCOTCH [30].

4. **Mondriaan** is a sequential graph and matrix partitioning library for use in sparse matrix-vector multiplication [9]. It uses a multilevel coarsening and refinement framework and can also be used to partition rectangular matrices.

5. **Zoltan** is a dynamic load-balancing library for parallel scientific computations. In addition to providing a number of simple partitioning methods for graphs and hypergraphs, it interfaces with other graph partitioning libraries to provide a large variety of available algorithms and heuristics [31].

6. **PaToH** is a hypergraph partitioning library that can provide k-way hyperedge partitions [32, 33]. Similar to METIS, it utilizes combinatoric local search heuristics within a coarsening and refinement framework to compute its partitioning.

Despite the great variety of graph partitioning libraries in existence, none explicitly utilize continuous optimization formulations nor fully exploit other traditional optimization algorithms.

1.6 New Approaches Using Optimization Formulations

Despite there being a large body of existing techniques for graph partitioning, a promising new research direction involves formulating graph partitioning problems as optimization problems. As many of these graph partitioning problems can be formulated as discrete or continuous optimization problems, existing algorithms can be applied, such as gradient projection or branch-and-bound search. However, this naive approach is generally not very efficient, and general purpose solvers either converge to poor solutions or take intractable lengths of time. New algorithms for solving these special classes of optimization problems are ongoing, with the principle advantage of being able to better explore a highly nonconvex search space. For example, continuous optimization methods do not require vertices to be in any single part at intermediate search steps, allowing vertices to be partially in and partially out of a given part. This can result in a more direct path to a better local solution.

1.7 Overview of Optimization Techniques

Because this work relies heavily on optimization techniques, we provide a short introduction of these techniques here.

1.7.1 Linear Programming

$$\begin{aligned} \text{maximize} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^{\mathsf{T}}\boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{LP}$$

Linear programming describes a class of optimization problems with a linear objective function and linear constraints. One general form for linear programming problems, or LPs, is shown in (LP). Note that the equality constraint can represent inequalities with the introduction of slack variables, and one can convert (LP) into a minimization problem by instead maximizing $-f(\boldsymbol{x})$. Example applications of linear programming include network flow problems (such as max-flow/min-cut) and crew scheduling problems [34].

Linear programs can be solved in polynomial time using either the simplex algorithm or an interior point method [35, 36].

1.7.2 Quadratic Programming

$$\begin{aligned} &\text{minimize} \quad && f(x) = \frac{1}{2}\boldsymbol{x}^{\mathsf{T}}\boldsymbol{G}\boldsymbol{x} + \boldsymbol{c}^{\mathsf{T}}\boldsymbol{x} \\ &\text{subject to} \quad && \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{A}^{\mathsf{T}}\boldsymbol{x} \leq \boldsymbol{b} \\ & && \boldsymbol{x} \in \mathbb{R}^n \end{aligned} \tag{QP}$$

Quadratic programming can be viewed as an extension of linear programming to a quadratic objective function, or a special case of nonlinear programming with linear constraints. Quadratic programming problems, or QPs, can often be solved by extensions to methods used to solve LPs, as well as by methods used to solve more complex nonlinear programming problems (NLPs).

The concept of convexity determines how difficult a given quadratic program is to solve. If the matrix $\boldsymbol{G}$ is positive semidefinite, the quadratic program is said to be *convex*, generally implying that a locally optimal solution is also globally optimal. If $\boldsymbol{G}$ is not positive semidefinite, the quadratic program is said to be *nonconvex*, implying the possibility of multiple local optima [36].

Methods exist for solving convex QPs efficiently and in polynomial time, but nonconvex QPs are NP-hard and generally require some form of spatial subdivision framework to guarantee global optimality. Because of this, a common approach is to apply a gradient- or Hessian-based method normally used on convex QPs on a nonconvex QP to converge to a local optimum. For example, if gradient descent is applied to a nonconvex QP and converges to a point satisfying first- and second-order optimality conditions, this solution is said to be *locally optimal*. Locating locally optimal solutions to a nonconvex QP can generally be done quickly, but no guarantees regarding the quality of such a solution (e.g. an optimality gap, or difference in objective function from the current solution and the global optimum) are provided.

1.7.3 Mixed-Integer Linear Programming

Graph partitioning involves making discrete decisions (e.g. a vertex being in a part or not), and is perhaps more naturally expressed as a discrete optimization problem. As such, we introduce mixed-integer linear programming, an extension of linear programming to include integer and/or binary variables.

Integer linear programming is a subset of mixed-integer linear programming where no continuous variables are used. Both ILP and MILP are NP-hard.

$$\begin{aligned} \text{maximize} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^{\mathsf{T}}(\boldsymbol{x} + \boldsymbol{y}) \\ \text{subject to} \quad & \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \\ & \boldsymbol{y} \in \mathbb{Z} \end{aligned} \tag{MILP}$$

MILPs are generally solved using a branch-and-bound tree, which involves progressively fixing and relaxing the integer variables to obtain bounds regarding the optimal solution. While branch-and-bound tree-based approaches suffer from poor worst-case asymptotic performance, they are also capable of superlinear speedup when parallelized [37, 38].

2. COMPUTATIONAL OPTIMIZATION APPROACHES TO COMPUTING EDGE CUTS*

2.1 Introduction

In this chapter, we present a multilevel graph partitioning library and algorithm incorporating novel coarsening and optimization approaches for computing edge cuts. We outline the algorithm used and its associated novel elements, its implementation details, and compare its performance using several graph partitioning metrics. We also apply this library to partition a large collection of graphs and compare our results to METIS, another graph partitioning library [12].

A brief discussion of multilevel graph partitioning appears in Section 2.3. Related and prior work in graph partitioning is discussed in Sections 2.2. The main components of the proposed algorithm and their relationship with one another are given in Sections 2.4, 2.5, and 2.6; computational results and comparisons are provided in Section 2.7. We conclude with a summary of this work and highlight future research directions in Section 2.8.

2.1.1 Problem Definition

Computing an edge cut is an NP-complete problem defined as taking an undirected input graph, $G = (V, E)$, and removing (i.e. cutting) edges until the graph breaks into two disjoint subgraphs. The set of edges deleted in this manner is known as the *cut set*, with the vertices on either side of these edges referred to as the *shores* of the cut. When partitioning a graph, we seek to minimize the number (or, more generally, the total weight) of edges in the cut set while maintaining balance between the size or weight of the vertices in each component.

2.1.2 Applications

Graph partitioning arises in a variety of contexts including VLSI circuit design, dynamic scheduling algorithms, computational fluid dynamics (CFD), and fill-reducing orderings for sparse direct methods for solving linear systems [39].

In VLSI circuit design, integrated circuit components must be arranged to allow uniform power

*The work described in this chapter is adapted from “Mongoose, A Graph Coarsening and Partitioning Library” by Timothy Davis, William Ward Hager, Scott Parker Kolodziej, and Nuri Sencer Yeralan, 2019, *ACM Transactions on Mathematical Software*, Copyright 2019 ACM [1]. Copyright of the work is retained by the authors.

demands across each silicon layer while simultaneously reducing the manufacturing costs by minimizing the required number of layers. Graph partitioning is used to determine when conductive material needs to be cut through to the next layer.

In the dynamic scheduling domain, task-based parallelism models dependencies using directed acyclic graphs. Graph partitioning is used to extract the maximum amount of parallelism for a set of vertices while maintaining uniform workload, maximizing high system utilization, and promoting high throughput.

Sparse matrix algorithms utilize graph partitioning when computing parallel sparse matrix-vector multiplication, as well as when computing fill-reducing orderings for sparse matrix factorizations.

2.2 Related Work

2.2.1 Combinatoric Methods

Kernighan and Lin at Bell Labs developed the first graph partitioning package for use at Bell Systems [40]. Their algorithm considers all pairs of vertices and swaps vertices from one part to the other when a net gain in edge weights is detected.

Fiduccia and Mattheyses improved upon the Kernighan-Lin swapping strategy by ranking vertices by using a metric called the *gain* of a vertex [16]. The Fiduccia-Mattheyses algorithm constrains edge weights to integers and computes gains in linear time. The algorithm swaps the partitions of vertices in order from greatest to least gain while updating the gains of its neighbors. Vertices are allowed to swap partitions once per application of the algorithm.

Many variations of these two algorithms exist, but their fundamental strategy of swapping discrete vertices has remained largely intact. As an example of more recent extensions, Karypis and Kumar considered constraining swap candidates to those vertices lying in the partition boundary [12], a strategy we have also adopted in Mongoose.

2.2.2 Coarsening, Matchings, and Multilevel Frameworks

Most graph partitioning heuristics scale at least quadratically with respect to either edges or vertices, and therefore become intractable for large graphs. However, many heuristics can perform well if given a sufficiently good initial partition to start from. Multilevel frameworks were introduced to address this issue by coarsening (or contracting) large graphs into a hierarchy of smaller graphs [20].

During coarsening, vertex matchings are computed that ideally retain the topology of the original graph. These matchings can be computed in a variety of ways. Karypis and Kumar considered *Heavy Edge Matching* (HEM), *Sorted Heavy Edge Matching* (SHEM), and *Heavy Clique Matching* (HCM) [12], as well as *Light Edge Matching* (LEM) and *Heavy Clique Matching* (HCM) [41]. Gupta considered *Heavy Triangle Matching* (HTM) [42]. Generally, some consideration is given to edge weights, and recently methods have been proposed to avoid stalling during coarsening, where matchings result in far fewer than the ideal $n/2$ vertex matches in each iteration, such as matching vertices with similar neighbors [11] and 2-hop matching [43].

Our extensions to these coarsening and matching methods are explained in detail in Section 2.4.

2.2.3 Recent Optimization Approaches

While the traditional approaches to graph partitioning are combinatorial in nature, swapping discrete vertices from one part to another, a variety of novel optimization formulations for partitioning problems have been introduced recently in the literature. While using optimization in graph partitioning is not uncommon using strategies such as simulated annealing [44] and mixed-integer programming [45], these discrete methods have many of the same scaling issues as the combinatorial methods that do not (explicitly) use optimization. As such, continuous optimization formulations have been proposed by Hager and Krylyuk [22], who showed that the discrete graph partitioning problem is equivalent to the continuous quadratic programming problem

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & (\mathbf{1}-\boldsymbol{x})^{\mathsf{T}}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{x} \\ \text{subject to} \quad & \mathbf{0}\le\boldsymbol{x}\le\mathbf{1}, \quad \ell\le\mathbf{1}^{\mathsf{T}}\boldsymbol{x}\le u, \end{aligned} \tag{2.1}$$

where ℓ and u are lower and upper bounds on the desired size of one partition, and $\boldsymbol{A}$ is the adjacency matrix of the graph. They show that this continuous quadratic programming problem has a binary solution; moreover, the partitions

$$\{i : x_i = 0\} \quad \text{and} \quad \{i : x_i = 1\}$$

are optimal solutions of the graph partitioning problem if the quadratic program is solved to optimality. This is the formulation that we utilize in Mongoose to form one part of our hybrid algorithm, described in more detail in Sections 2.5 and 2.6.

2.2.4 Graph Partitioning Libraries

A variety of graph partitioning libraries and algorithms have been developed over the past several decades. Perhaps most well-known is METIS [12], an early multi-level framework partitioner that has since been refined and expanded to include parallel [26], hypergraph [27], and multi-threaded [28] versions. We use METIS as our primary comparison in Section 2.7, and build on their work on coarsening and refinement. Other graph partitioning libraries are listed in Section 1.5.6.

2.3 Multi-Level Graph Partitioning

Multilevel graph partitioners seek to simplify the input graph in an effort to recursively apply expensive partitioning techniques on a smaller problem. The motivation for such a strategy is due to limited memory and computational resources to apply a variety of combinatoric techniques on large input problems. By reducing the size of the input, more advanced techniques can be applied and carried back up to the original input problem.

2.3.1 Graph Coarsening

The process whereby an input graph is simplified is known as *graph coarsening*. In graph coarsening, the original input graph is reduced through a series of vertex matching operations to an acceptable size [20]. Vertices are merged together using strategies that exploit geometric and topological features of the problem.

High degree vertices that arise in irregular graphs, particularly social networks, impede graph coarsening by reducing the maximum number of matches that can be made per coarsening phase.

When the number of coarsening phases becomes proportional to the degree of a vertex, we say that coarsening has *stalled*.

2.3.2 Initial Guess Partitioning

Once the input graph is coarsened to a size suitable for more aggressive algorithms, an initial guess partitioning algorithm is used. Initial partitioning strategies accumulate a number of vertices into one partition such that the desired partition balance is satisfied. Karypis and Kumar demonstrated that region-growing techniques, such as applying a breadth-first search from random start vertices, tend to find higher quality initial partitions than random guesses or first/last half guesses [12].

Note that for edge cuts even a random initial guess provides a valid cut. Also, when partitioning graphs with more than a single connected component, some initial partitioning schemes may fail or perform poorly. As such, random initial partitions and bin-packing approaches can offer a safe fallback method.

2.3.3 Uncoarsening

Once a satisfactory guess partition is computed at the coarsest level, projecting the partition back to the original input graph requires the inverse operation of graph coarsening, known as graph uncoarsening. During uncoarsening, vertices expand back into their original representations at the finer level. The part choice (X or Y) for each coarse vertex is applied to all of the vertices that participated in the matching used during coarsening.

Because a partition at a coarse level is not generally guaranteed to be optimal when projected to the finer level, traditional graph partitioning strategies (e.g. methods described in Section 2.6.4.2) are used to improve the projected partition as the graph is refined back to its original size.

2.4 Coarsening and Matching Strategies

To coarsen a graph as described in Section 2.3.1, a *matching* of which vertices are merged together must be computed. More precisely, a mapping of vertices to super-vertices (i.e. fine to coarse) must be created. A variety of matching strategies exist, including heavy edge matching, where vertices are matched with the neighbor with the incident edge of largest weight [12]. One disadvantage of heavy edge matching is that it can be prone to stalling. If matching is limited to neighbors, a high-degree vertex may prevent matching more than two vertices at a time. We present a variety of additional strategies to avoid such stalling, including an approach that can guarantee that the number of vertices in each phase of coarsening decreases by at least half.

2.4.1 Brotherly Matching

In brotherly matching, two vertices that are not neighbors can be matched if they share a neighbor (see Figure 2.1). This can help prevent stalling in cases such has star graphs, where a single high-degree vertex can only be matched with a single neighbor in each pass. Using brotherly matching, many vertices can be matched together because they share a neighbor (the central high-degree vertex).

Note that brotherly matching is already implemented in METIS 5 as *2-hop* matching [12]. The next two methods, adoption and community matching, are novel.

2.4.2 Adoption Matching

Related to brotherly matching is adoption matching, which allows a three-way matching between adjacent vertices. Given an even number of unmatched neighbors of a given unmatched vertex, a matching can be computed using heavy-edge and brotherly matching that leaves only one vertex unmatched. The remaining vertex that is not matched pairwise with any other vertex is added to (i.e. adopted by) an existing two-way match (see Adoption Matching in Figure 2.1).

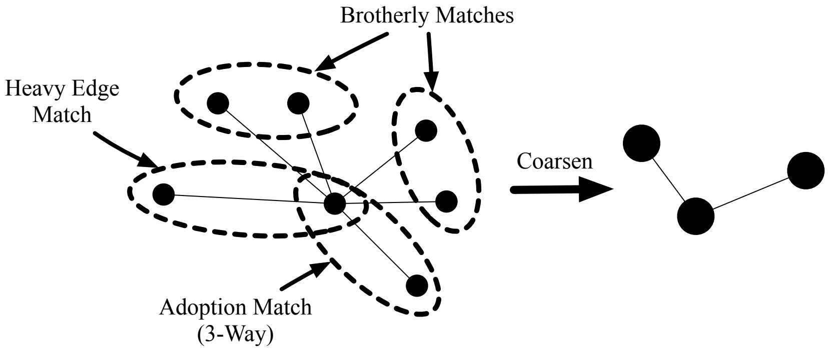


Figure 2.1: Brotherly and adoption matching

2.4.3 Community Matching

Community matching occurs when two neighboring vertices are both in a 3-way match formed by adoption matching. Since two neighboring vertices each have a vertex matched via adoption, those adopted vertices can instead be matched with each other (see Figure 2.2).

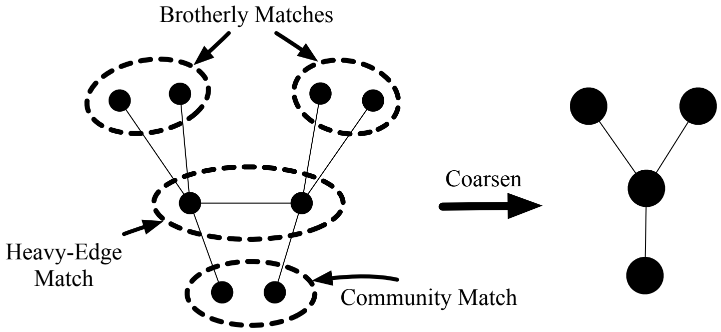


Figure 2.2: Community matching

2.5 Quadratic Programming Refinement

As mentioned earlier, Hager and Krylyuk [22] introduced a continuous quadratic programming formulation of the edge cut graph partitioning problem:

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \quad (\mathbf{1}-\boldsymbol{x})^{\mathsf{T}}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{x} \qquad \text{subject to} \quad \mathbf{0}\le\boldsymbol{x}\le\mathbf{1}, \quad \ell\le\mathbf{1}^{\mathsf{T}}\boldsymbol{x}\le u, \tag{1}$$

where ℓ and u are lower and upper bounds on the desired size of one partition, and $\boldsymbol{A}$ is the adjacency matrix of the graph. They show that this continuous quadratic programming problem has a binary solution; moreover, the partitions

$$\{i : x_i = 0\} \quad \text{and} \quad \{i : x_i = 1\}$$

are optimal solutions of the graph partitioning problem.

During refinement, we complement our implementation of the Fiduccia-Mattheyses algorithm with the quadratic programming approach. Because we use both traditional combinatoric methods as well as quadratic programming at the refinement stage in an effort to yield better quality results, we call this a *hybrid graph partitioning method*.

To actually solve the quadratic programming formulation, we first use the discrete partition choices for each vertex as a starting guess for a solution to the quadratic programming problem (2.1). We then perform iterations of gradient projection until reaching a stationary point of (2.1); often convergence takes just a few iterations. Although the stationary point may not be binary, the analysis in [22] shows how to move to a binary feasible point while possibly further improving the objective value.

Note that each iteration of the gradient projection algorithm takes a step along the negative gradient followed by projection onto the feasible set of (2.1). Since the constraints of (2.1) consist of a single linear constraint coupled with bound constraints, computing this projection amounts to solving a quadratic knapsack problem. An efficient algorithm for computing this projection, called NAPHEAP, is given in [46]. Because the quadratic programming formulation ignores the discrete notion of boundary, it is capable of identifying vertices to swap which do not lie on the boundary of the cut. Gradient projection also adheres to strict bounds on part size as a way of enforcing balance, and its local minimizers result in cuts with better balance than our Fiduccia-Mattheyses implementation.

2.6 Algorithm Description

In this section, we describe a novel hybrid graph partitioning heuristic algorithm based on both the quadratic programming formulation and existing combinatoric methods. We have implemented this algorithm in a production-ready C++ library available to the public.

2.6.1 Input and Pre-Processing

First, Mongoose accepts an input graph and ensures that it is undirected (i.e. the adjacency matrix is symmetric) and has no self-edges (an all-zero diagonal). It also ensures that all edge and vertex weights are strictly positive.

2.6.2 Coarsening

Once the graph is found to be acceptable, Mongoose coarsens the graph down to a user-specified number of vertices (64 by default). The stall-reducing matching methods (Section 2.4) are used during coarsening to avoid stalling on irregular and power-law graphs. However, community matching is disabled by default, as it can be computationally expensive on all but very irregular graphs.

2.6.3 Initial Partitioning

At the coarsest (and smallest) level, an initial partition is computed for the graph. By default, a random partitioning is computed, but partitions computed from the quadratic programming formulation as well as the natural order of the vertices are available.

The algorithm then performs a round of the Fiduccia-Mattheyses algorithm, but only considers shore vertices. This acts to minimize the size of the cut to a minimal (although not necessarily globally optimal) size.

These partition choices for vertices are then used as an initial starting point for gradient projection. Because gradient projection is a continuous method, it computes the affinity of a vertex as a floating point value between 0 and 1. Our algorithm discretizes this result and interprets values of $x \leq 0.5$ as the first partition and values $x > 0.5$ as the second partition.

2.6.4 Uncoarsening and Refinement

The graph is then repeatedly uncoarsened and the edge cut refined. First, the partition for the coarsened graph is projected upward to the uncoarsened parent graph, and the primary refinement loop begins.

During refinement of the cut, the following quadratic programming-based approach and discrete Fiduccia-Mattheyses algorithm are used alternately to search for a smaller (and/or more balanced) cut.

2.6.4.1 Quadratic Programming-Based Refinement

We solve the quadratic programming formulation (2.1) using gradient projection, which can be summarized as follows:

1. With an initial starting point, take a step in the direction of the negative gradient to improve the solution.

2. Project the step back onto the feasible space defined by the constraints.

Because the constraints are box constraints (upper and lower bounds only), the projection step is straightforward to compute, capping the solution at the bounds. Convergence using gradient projection to a locally optimal solution usually only requires a few iterations. We utilize a modified version of NAPHEAP [46] to solve the gradient projection subproblem.

2.6.4.2 Fiduccia-Mattheyses Algorithm Refinement

We implement a version of the Fiduccia-Mattheyses algorithm (see Sections 1.5.1 and) for improving edge cuts to use in tandem with the QP-based refinement already described. Rather than impose hard bounds on part size, we introduce a heuristic cost to penalize imbalance without strictly forbidding it:

$$f_{heuristic} = \begin{cases} \displaystyle\sum_{(i,j)\in \text{ Cut Set}} w_{i,j} + \psi H & \text{if } \psi > \text{imbalance tolerance} \\ \displaystyle\sum_{(i,j)\in \text{ Cut Set}} w_{i,j} & \text{otherwise} \end{cases} \tag{2.2}$$

In this expression, H is a heuristic penalty equal to twice the sum of all edge weights in the graph ($H = 2\sum\limits_{(i,j)\in E} w_{i,j}$). The imbalance metric ψ is defined as follows:

$$\psi = (\text{Target Ratio}) - \frac{\min\{\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}, \boldsymbol{w}^{\mathsf{T}}(\mathbf{1}-\boldsymbol{x})\}}{\mathbf{1}^{\mathsf{T}}\boldsymbol{w}} \tag{2.3}$$

As is the hallmark of the Fiduccia-Mattheyses algorithm, we use two heaps, one for each part, as priority queues keyed by the gain metric G for each vertex i:

$$G_i = W_{\text{External}} - W_{\text{Internal}} \tag{2.4}$$

In this gain metric, W_{External} is the sum of edge weights connecting vertex i to a vertex in another part, and W_{Internal} is the sum of edge weights connecting vertex i to a vertex in the same part as vertex i. Thus, for vertices where it would be advantageous to move to the opposite part, G_i should be positive, removing W_{External} from the cut and replacing it with edges with edge weights equal to W_{Internal}.

Our implementation of the Fiduccia-Mattheyses algorithm continues to make non-advantageous moves (swapping vertices with non-positive gains) in an effort to better explore the search space of edge cuts. Once a user-defined number of moves has been made with no improvement, the cut is restored to the best cut found to that point.

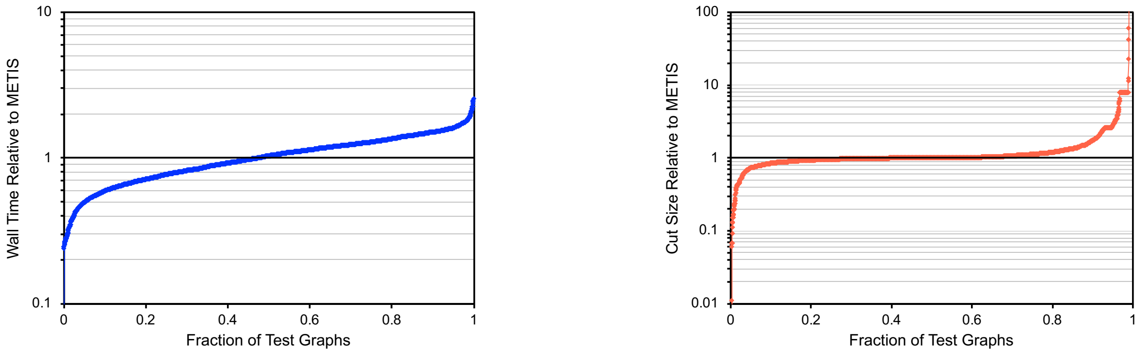
2.7 Results

In this section, we explore the computational performance of Mongoose compared to METIS, a popular graph partitioning library, on a variety of graph sizes and types. All experiments were run on a 24-core dual-socket 2.40 GHz Intel Xeon E5-2695 v2 system with 768 GB of memory. Note that only one thread was utilized, as both libraries are serial in nature. All comparisons were conducted with METIS 5.1.0 and compiled with GCC 4.8.5 on CentOS 7.

For consistency, each partitioner was run five times for each problem. The highest and lowest times are removed, and the remaining three are averaged (i.e. a 40% trimmed mean). Default options were used, and a target split of $50\%/50\%$ was used with a tolerance of $\pm 0.1\%$. All results shown satisfy this balance tolerance.

2.7.1 Overall Performance

Mongoose and METIS were run on the entire SuiteSparse Matrix Collection [11] with only modest filtering. First, complex matrices were removed. Of the remaining matrices, any unsymmetric matrices $\boldsymbol{A}$ were treated as the biadjacency matrix of a bipartite graph adjacency matrix $\boldsymbol{B} = \begin{bmatrix} \mathbf{0}_{m,m} & \boldsymbol{A} \\ \boldsymbol{A}^T & \mathbf{0}_{n,n} \end{bmatrix}$; symmetric matrices were unmodified and treated as undirected graphs. A final preprocessing step removed any nonzero diagonal elements (i.e. ignoring/eliminating self edges) and reduced the matrix to a binary pattern (i.e. nonzero elements were replaced with 1).

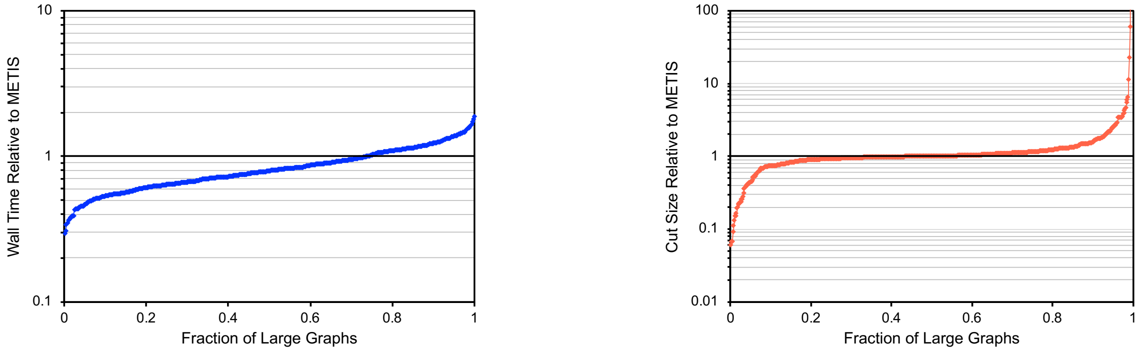


(a) (b)

Figure 2.3: Overall timing (a) and overall cut quality (b) performance of Mongoose relative to METIS 5. Note the logarithmic vertical scale. Points below the center line represent cases where Mongoose outperforms METIS (relative performance less than one), while points above the center line indicate cases where METIS outperforms Mongoose (relative performance greater than one). In general, Mongoose performs competitively with METIS 5.

Table 2.1: Performance comparison between Mongoose and METIS on all 2,685 graphs from (or formed from) the SuiteSparse Collection.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	759	527	1286
	Tie	113	173	286
	Mongoose	542	571	1113
		1414	1271	2685



(a)

(b)

Figure 2.4: Timing (a) and cut quality (b) performance profiles [47] of Mongoose on large graphs (1,000,000+ edges) relative to METIS 5. Note the logarithmic vertical scale. Points below the center line represent cases where Mongoose outperforms METIS (relative performance less than one), while points above the center line indicate cases where METIS outperforms Mongoose (relative performance greater than one).

This yielded 2,685 symmetric matrices at the time of comparison, which were then treated as undirected graphs to be partitioned.

The relative timing (a) and relative cut quality (b) performance are shown in Figure 2.3, and a tabular comparison is shown in Table 2.1. Of the 2,685 graphs, Mongoose found a smaller cut on 1,113 ($\sim$41%), and took less time to compute its cut on 1,271 ($\sim$47%). Mongoose outperformed METIS in both time and cut quality on 571 graphs ($\sim$21% of cases), while METIS outperformed Mongoose in both time and cut quality on 759 graphs ($\sim$28%). Thus, Mongoose is generally competitive with METIS, with METIS having a slight edge.

Figures 2.3, 2.4, and 2.7 are created by first computing the computational metrics (either cut quality or wall time) relative to METIS: less than 1 being better than METIS, and greater than 1 being worse. The results are then ordered from smaller (better than METIS) to larger (worse than METIS) and plotted on a logarithmic scale. Along the horizontal axis, graph numbers are normalized to the interval $[0, 1]$, with the first graph corresponding to 0, and the last graph corresponding to 1.

Table 2.2: Performance comparison between Mongoose and METIS on the 601 largest graphs (1,000,000+ edges) in the SuiteSparse Collection.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	99	215	314
	Tie	2	11	13
	Mongoose	57	217	274
		158	443	601

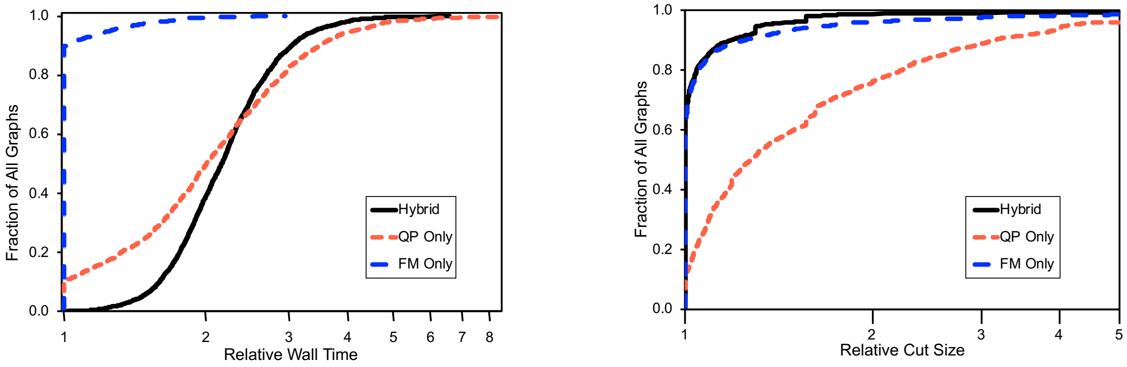
2.7.2 Performance on Large Graphs

When limited to graphs with at least 1,000,000 edges, Mongoose performs significantly better. Of the 601 graphs that meet this size criterion, Mongoose computed smaller edge cuts in 274 cases (∼46%), and terminated faster in 443 cases (∼74%). Mongoose outperformed METIS in both time and cut quality on 217 graphs (∼36%), while METIS outperformed Mongoose in both time and cut quality on only 99 of the large graphs (∼16%). Thus, Mongoose provides comparable cut quality, but much faster execution when partitioning large graphs compared to METIS. The relative timing (a) and relative cut quality (b) performance are shown in Figure 2.4, and a tabular comparison is shown in Table 2.2.

2.7.3 Hybrid Performance

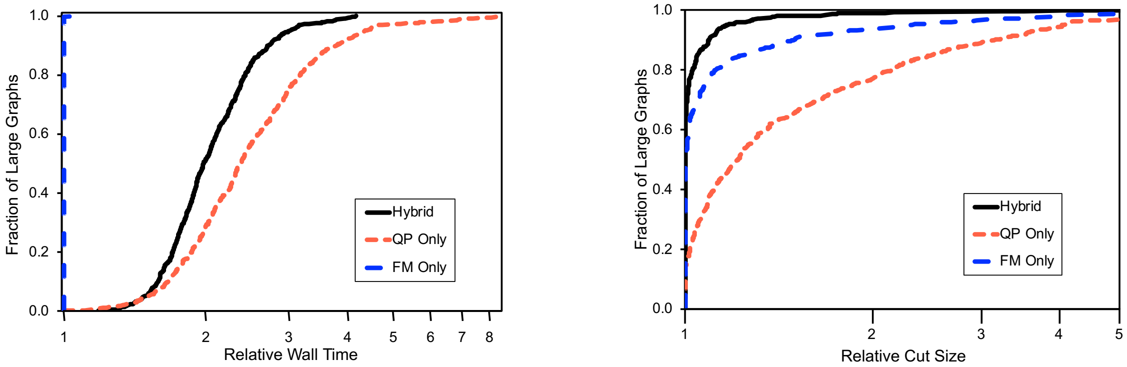
Figures 2.5 and 2.6 compare the hybrid graph partitioning method to the combinatorial method and quadratic programming methods in isolation. While the combinatorial Fiduccia-Mattheyses algorithm is very fast, its resulting cut quality is inferior to that of the hybrid approach (markedly so with large graphs). In isolation, the quadratic programming approach is less performant in both speed and cut quality when compared to the Fiduccia-Mattheyses and hybrid methods, highlighting the algorithmic cooperation of the two approaches that make the hybrid approach so effective.

Figures 2.5, 2.6, and 2.8 are generated by calculating performance for each option relative to the fastest time or smallest cut size (with the best result being 1, and all other results being greater than or equal to 1). The graphs are ordered from best to worst along the vertical axis and



(a) (b)

Figure 2.5: Relative timing (a) and relative cut size (b) performance profiles [47] on the 2,685 graphs formed from the SuiteSparse Matrix Collection. In the figure, the following methods appear: Hybrid (black), Quadratic Programming only (red), and Fiduccia-Mattheyses only (blue). Note that the horizontal axis is logarithmic.



(a) (b)

Figure 2.6: Relative timing (a) and relative cut size (b) performance profiles [47] on the largest 601 graphs in the SuiteSparse Matrix Collection (1,000,000+ edges). In the figure, the following methods appear: Hybrid (black), Quadratic Programming only (red), and Fiduccia-Mattheyses only (blue). Note that the horizontal axis is logarithmic. The hybrid approach provides better cuts than either standalone approach while taking less time than the quadratic programming method alone.

normalized on the interval $[0, 1]$, with the first graph (best result) at 0 and the last (worst result) at 1. These plots are generally known as performance profiles [47].

2.7.4 Power Law and Social Networking Graphs

We examined our hybrid combinatorial quadratic programming algorithm on power law graphs that arise in social networking and Internet hyperlink networks. The problem set of 41 social networking graphs was formed by filtering the SuiteSparse Matrix Collection using the words “wiki," “email," “soc-*," and all matrices in the Laboratory for Web Algorithmics (LAW) collection [48] [49].

Figure 2.7 and Table 2.3 suggest that the hybrid approach is both significantly faster and nearly always computes a higher quality cut than METIS for this class of graph. We speculate that this is due to the following factors:

- **Our Coarsening Strategy** is able to prevent stalling during coarsening while preserving topological features. In mesh-like and regular graphs, stalling is generally not a problem, but in social networking graphs, high-degree (or “celebrity") vertices can lead to time-consuming coarsening phases. With our brotherly/adoption matching methods, Mongoose is able to efficiently coarsen these social networking graphs.

- **Algorithmic Cooperation.** The combinatorial algorithm provides the quadratic programming formulation a guess partition that gradient projection can improve on. Conversely, the quadratic programming formulation exchanges vertices that are not necessarily on the partition boundary, overcoming a limitation of our combinatorial partitioning method.

Table 2.4, which contains the largest 15 social networking graphs from the problem set of 41, further suggests that our hybrid approach may result in significant improvement in cut quality for large social networks. Of these largest 15 such networks, Mongoose found a better cut in all but one case when compared to METIS, and did so faster in 8 out of the 14 cases.

One social networking graph of particular note is SNAP/email-EuAll, as it highlights Mongoose’s singleton handling during coarsening. This graph has a single connected component that

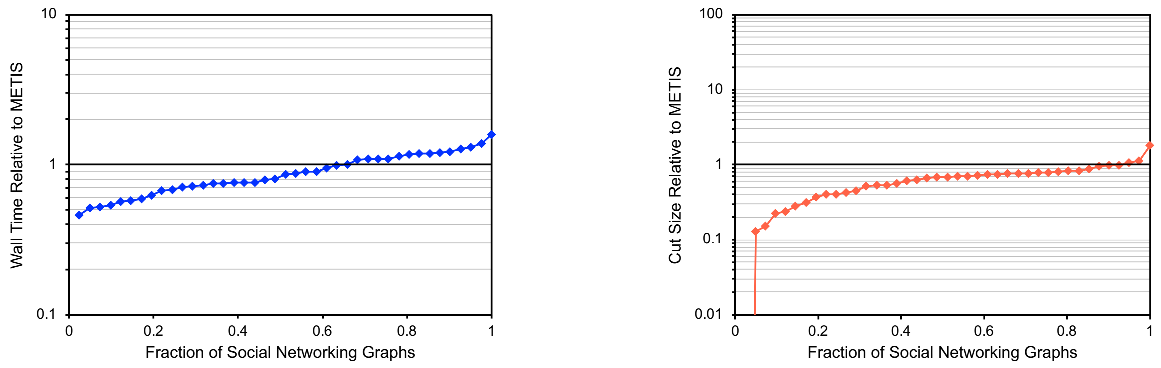


Figure 2.7: Performance of Mongoose on social networking graphs relative to METIS 5. Note the logarithmic vertical scale. Points below the center line represent cases where Mongoose outperforms METIS (relative performance less than one), while points above the center line indicate cases where METIS outperforms Mongoose (relative performance greater than one).

Table 2.3: Performance comparison between Mongoose and METIS on 41 social networking (power law) graphs in the SuiteSparse Collection. There were no ties in cut quality.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	1	2	3
	Mongoose	14	24	38
		15	26	41

Table 2.4: Performance comparison between Mongoose and METIS on the 15 largest (by edges) social networking graphs in the SuiteSparse Collection. Note that the bipartite graph is formed for unsymmetric (directed) graphs, which is all graphs listed except LAW/hollywood-2009. All results had zero imbalance (i.e. the target balance of 50% was achieved in all cases).

Problem			Wall Time (s)			Cut Size (# of Edges)		
Graph Name	Vertices	Edges	METIS	Mongoose	Speedup	METIS	Mongoose	Relative Cut Size
LAW/sk-2005	101,272,308	3,898,825,202	554.1	**255.2**	**2.17**	7,380,768	**4,518,734**	**0.61**
LAW/it-2004	82,583,188	2,301,450,872	226.1	**128.6**	**1.76**	2,486,866	**693,068**	**0.28**
LAW/webbase-2001	236,284,310	2,039,806,380	488.0	**253.1**	**1.93**	2,709,752	**616,101**	**0.23**
LAW/uk-2005	78,919,850	1,872,728,564	194.7	**121.2**	**1.61**	1,810,378	**821,430**	**0.45**
LAW/arabic-2005	45,488,160	1,279,998,916	109.6	**64.6**	**1.70**	805,443	**189,641**	**0.24**
LAW/uk-2002	37,040,972	596,227,524	82.3	**44.3**	**1.86**	613,916	**192,917**	**0.31**
LAW/indochina-2004	14,829,732	388,218,622	26.9	**18.0**	**1.49**	46,350	**18,522**	**0.40**
LAW/ljournal-2008	10,726,520	158,046,284	**61.6**	66.4	0.93	3,962,147	**3,015,059**	**0.76**
SNAP/soc-LiveJournal1	9,695,142	137,987,546	**58.8**	69.4	0.85	3,740,193	**3,093,681**	**0.83**
LAW/hollywood-2009	1,139,905	112,751,422	**10.8**	11.8	0.92	2,388,505	**1,872,190**	**0.78**
Gleich/wikipedia-20070206	7,133,814	90,060,778	**43.2**	59.7	0.72	5,536,148	**2,833,749**	**0.51**
Gleich/wikipedia-20061104	6,296,880	78,766,470	**41.9**	50.2	0.84	4,763,514	**2,544,141**	**0.53**
Gleich/wikipedia-20060925	5,966,988	74,538,192	**35.4**	56.1	0.63	4,653,238	**2,455,991**	**0.53**
Gleich/wikipedia-20051105	3,269,978	39,506,156	20.2	**19.9**	**1.02**	1,780,359	**1,352,360**	**0.76**
LAW/eu-2005	1,725,328	38,470,280	2.9	**2.3**	**1.26**	**40,188**	42,670	1.06

makes up nearly 50% of the vertices in the graph. Since Mongoose preferentially matches singletons with other singletons during coarsening, vertices in the largest components are internally matched with one another while the components making up the other half of the graph are matched with each other. This results in at least two large connected components at the coarsest level, leading to an edge cut of size zero.

2.7.5 Sensitivity Analysis of Options

Mongoose has a variety of options that can significantly impact performance (both time and cut quality). To investigate the tradeoffs of each set of options, four options were varied as described below, and each combination was used to compute an edge cut.

- **Matching Strategy.** During the coarsening phase, vertices are matched with other vertices to be contracted together to form a smaller (but structurally similar) graph. Mongoose contains four such methods of computing this matching:
  - **Random** matching matches a given unmatched vertex to a randomly selected unmatched neighbor.
  - **Heavy Edge Matching (HEM)** matches an unmatched vertex with a neighboring unmatched vertex with whom it shares the edge with the largest weight.
  - **Heavy Edge Matching with Stall-Free or Stall-Reducing Matching (HEMSR)** first conducts a heavy edge matching pass, but follows with a second matching pass to further match leftover unmatched vertices in brotherly, adoption, and community matches.
  - **Heavy Edge Matching with Stall-Reducing Matching, subject to a degree threshold (HEMSRdeg)**. Like HEMSR above, but the second matching pass is only conducted on unmatched vertices with degree above a certain threshold (in these experiments, twice the average degree).
- **Initial Cut Strategy.** After coarsening is complete, an initial partition is computed using one of three approaches:

  - **Random.** Randomly assigns vertices into an initial part.
  - **QP (Quadratic Programming).** Runs a single iteration of the quadratic programming formulation of the edge cut problem, with an initial guess of $x = 0.5$ for all vertices.
  - **Natural Order.** Assigns the first $\lfloor n/2 \rfloor$ vertices to one part, and the next $\lceil n/2 \rceil$ vertices to the other.

- **Coarsening Limit.** Coarsening terminates when a specified threshold number of coarsened vertices is reached. In these experiments, values of 1024, 256, and 64 were tested.
- **Community Matching.** When using stall-reducing matching, vertices can be optionally aggressively matched in community matches (two vertices are matched if their neighbors are matched together). This can be enabled to further maximize the number of matched vertices, or disabled to potentially save time.

The results of this sensitivity analysis in both time and cut quality are shown in Figure 2.8. For each option, the best result (in both time and cut quality) is chosen, and relative metrics are computed relative to this best result. The relative metrics are then sorted and plotted as a performance profile, with the best results being the ones that stay at or near 1.0 for the largest percentage of problems.

2.7.5.1 Matching Strategy

Heavy edge matching and random matching are competitive only with small graphs, but quickly become intractable for large problems. Of the two options that use stall-reducing matching, the one that is not subject to the degree threshold appears to perform slightly faster with no noticeable decrease in cut quality.

2.7.5.2 Initial Cut Strategy

While the natural ordering approach can sometimes be effective for meshes and other regular graphs, it is generally outperformed by both the QP and random initial cuts. Interestingly, the random initial cut yields a comparable final cut weight despite being more efficient to compute.

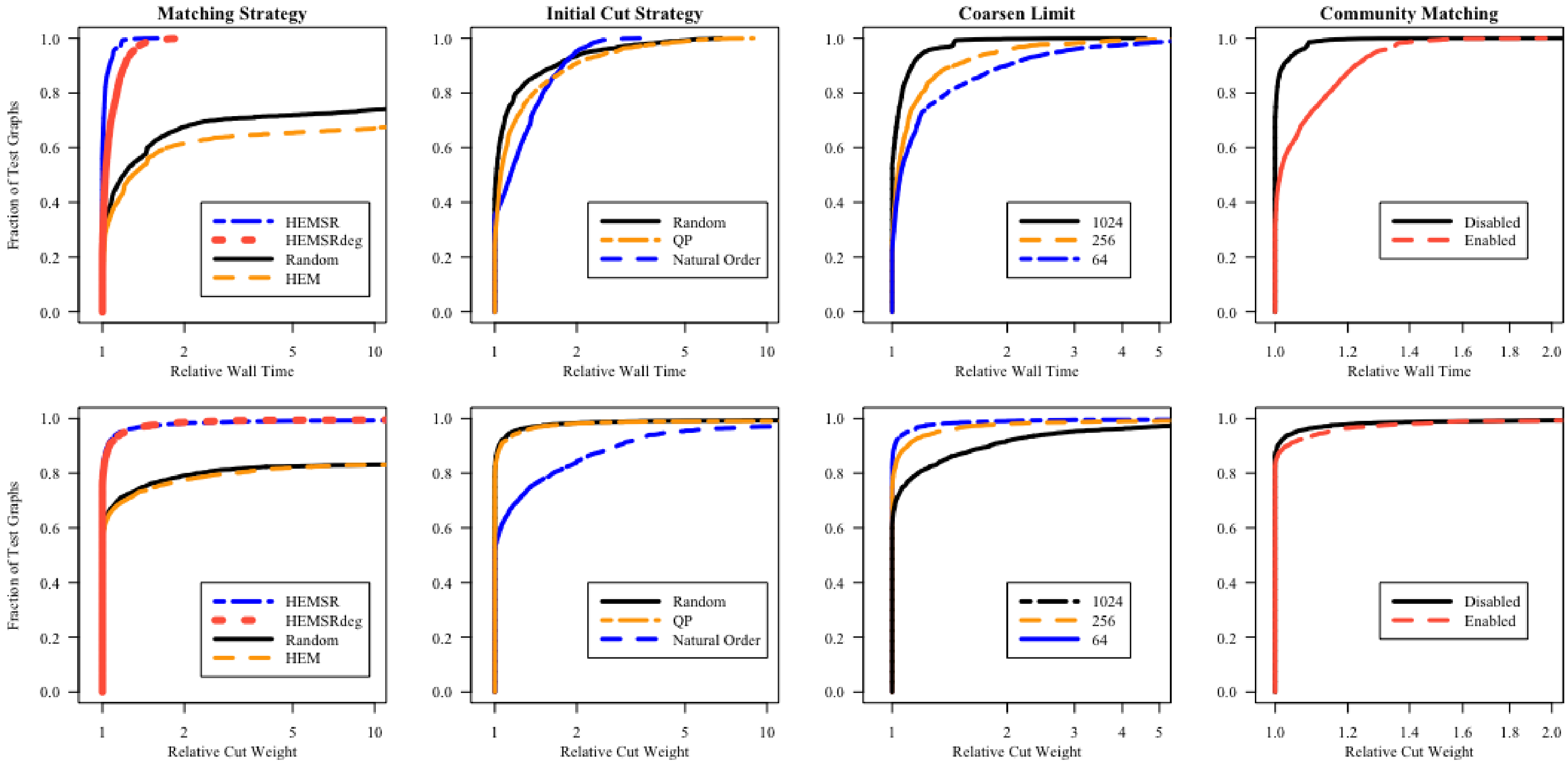


Figure 2.8: Relative timing (top row) and cut quality (bottom row) performance profiles of each set of options. Each column corresponds to an available option in Mongoose. Note that the horizontal axis is logarithmic, and the vertical axis corresponds to the fraction of the 2,685 graphs used for testing. Runs that exceeded 7200 seconds were terminated (as was the case for much of the HEM and Random matching strategy data).

2.7.5.3 Coarsening Limit

There is a tradeoff between speed and cut quality in determining the coarsening limit. If coarsening is terminated early (1024 vertices), less computational time is spent on coarsening, but the final cut weight is generally worse. Inversely, if coarsening continues to 64 vertices, more time is spent on the coarsening phases, but the resulting cut quality is generally better. This is unsurprising, as the heuristics used to find the initial cut and to progressively refine the cut are generally more effective with smaller graphs.

2.7.5.4 Community Matching

In the majority of cases, community matching has no significant effect on cut quality. However, for a sizable minority of graphs, community matching does have a detrimental effect on timing. In short, community matching does not appear to offer a significant improvement, but can be mildly helpful in coarsening graphs that are prone to stalling. For most graphs, the reduced stalling during coarsening does not justify the computational cost of computing the matching.

2.8 Summary

In this chapter, we have presented a novel hybrid graph partitioning library, Mongoose, for efficiently computing edge cuts for arbitrary graphs. By combining the optimization-based approach with the combinatoric Fiduccia-Mattheyses algorithm and utilizing novel graph coarsening methods, we can outperform existing graph partitioning software, especially for large social network graphs. In the next chapter, we will extend this approach to vertex separators.

3. COMPUTATIONAL OPTIMIZATION APPROACHES TO COMPUTING VERTEX SEPARATORS

3.1 Introduction

Now that we have developed a multilevel graph partitioning library capable of computing edge cuts, we now move on to the development of an algorithm for computing vertex separators. We can build on the work from Chapter 2 to create an efficient, scalable approach capable of computing high-quality vertex separators in arbitrary graphs. First, we review the current state of the art for vertex separators (Sections 3.3 and 3.4), then derive generalized gain metrics capable of combining multiple existing vertex separator algorithms (Section 3.5), before exploring the computational results of our new graph partitioning algorithm and library implementation (Sections 3.6 through 3.8).

3.2 Overview of the Vertex Separator Problem

As discussed in Section 1.3.2, the vertex separator problem is another graph partitioning problem that requires finding three parts: part X and part Y, which should be approximately balanced in size or weight, and a separator part S, whose size is minimized. The separators part should separate the other two parts such that no path exists from a vertex in part X to a vertex in part Y without traversing a vertex in part S.

While similar to an edge cut, the vertex separator problem is fundamentally a more difficult problem. First, in an edge cut, any possible assignment of vertices to parts is a (perhaps unbalanced) feasible edge cut. This is not the case in the vertex separator problem, where even the existence of a feasible solution is not guaranteed. For example, if the graph being partitioned is a clique (i.e. all vertices are connected to all other vertices), no vertex separator exists. Second, the concept of a balanced partition is less clear and more difficult to satisfy compared to edge cuts. Due to the introduction of a third part, balance can be evaluated a number of different ways. Do we account for the size of the separator when evaluating balance ($\frac{|X|}{|V|}$) or not ($\frac{|X|}{|X \cup Y|}$)? While both the edge cut problem and vertex separator problem are essentially mutli-objective optimization prob-

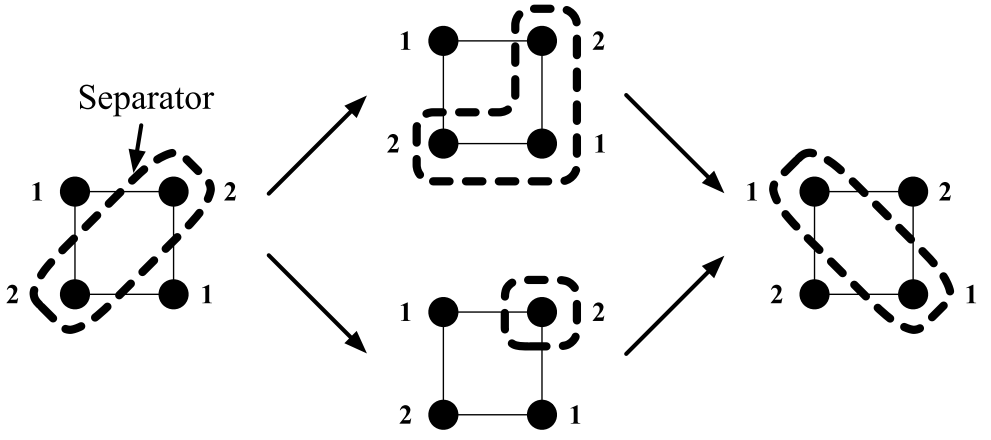


Figure 3.1: Difficulties in transitioning between valid vertex separator states. There may be no feasible transition states between one partition and another better partition. In the top transition, the graph is reduced to only two parts, and in the bottom, the separation constraint is violated.

lems and must address the tradeoff between balance and cut (or separator) size, ensuring balance in the vertex separator problem, especially during coarsening and uncoarsening of the graph, is non-trivial.

During coarsening, the number of vertices is reduced, but vertex weights are aggregated and increase. Thus, at very coarsened levels of the multilevel framework, balance may be difficult or impossible to satisfy. Worse, the balance constraint may even prevent moving through an unbalanced intermediate stage between one vertex separator to a better one (see Figure 3.1).

3.2.1 Complexity

Both the vertex separator and edge cut problems are NP-hard [4]. While algorithms exist that can guarantee globally optimal cuts or separators, they take exponential time in the worst case. Practical existing methods rely almost exclusively on heuristics to compute minimal cuts and separators.

3.3 Traditional Approaches

While both the Kernighan-Lin [40] and Fiduccia-Mattheyses [16] algorithms were developed for use in finding small, balanced edge cuts, they have been adapted for use in finding balanced vertex separators [17, 18]. The general approach is the same, computing gain values for vertices both in the separator and on the shores on each side of the separator and then moving vertices into and out of the separator accordingly. While these combinatoric methods are generally the most prevalent, other methods do exist, usually as extensions to edge cut algorithms (see Section 1.5).

3.4 Optimization Formulations and Approaches

Despite there being a large body of existing techniques for graph partitioning, a promising new research direction involves formulating graph partitioning problems as optimization problems. As many of these graph partitioning problems can be formulated as discrete or continuous optimization problems, existing algorithms can be applied, such as gradient projection or branch-and-bound search. However, this naive approach is generally not very efficient, and general purpose solvers either converge to poor solutions or take intractable lengths of time. New algorithms for solving these special classes of optimization problems are ongoing, with the principle advantage of being able to better explore a highly nonconvex search space. For example, continuous optimization methods do not require vertices to be in any single part at intermediate search steps, allowing vertices to be partially in and partially out of a given part. This can result in a more direct path to a better local solution.

3.4.1 Mixed-Integer Linear Programming Approaches

The first optimization formulation described for solving the vertex separator problem uses mixed-integer linear programming. This formulation, first introduced by Balas and de Souza, guarantees global optimality when used with a branch-and-bound MILP solver [50]. Given the computational complexity of branch-and-bound, this approach was very inefficient.

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{V} w_i(x_i + y_i) & & \\
\text{subject to} \quad & x_i + y_i \leq 1 & & \forall\, i \in V \\
& x_i + y_j \leq 1 & & \forall\, (i,j) \in E \\
& x_j + y_i \leq 1 & & \forall\, (i,j) \in E \qquad \text{(VSP-MILP)} \\
& \ell_x \leq \sum_{i \in V} w_i x_i \leq u_x & & \\
& \ell_y \leq \sum_{i \in V} w_i y_i \leq u_y & & \\
& x_i, y_i \in \{0,1\} & & \forall\, i \in V
\end{aligned}$$

The objective function maximizes the size of the X and Y parts (which is equivalent to minimizing the size of the separator). The first constraint enforces that each vertex is either in part X, part Y, or the separator S. The second and third constraints enforce the separator requirement, ensuring that no edge can have one endpoint in part X and the other endpoint in part Y (only $X-X$, $Y-Y$, $X-S$, and $S-Y$ edges are allowed).

Both the objective function and the constraints in this formulation are linear, but the x and y variables are strictly binary.

3.4.1.1 Solution Methods

As discussed in Section 1.7.3, solving mixed-integer linear programming problems to optimality is NP-hard, being an extension to 0-1 integer programming, one of the original NP-complete problems [51]. Solution strategies to find a good (but not necessarily optimal) solution to MILPs include simulated annealing [52, 44] and ant colony search and optimization [53], which has also been directly applied to graph partitioning [54, 55]. Due to their computational complexity, they have generally not been used in the graph partitioning space with the exception of some proofs of concept.

For guarantees on global optimality (or ϵ-optimality), the standard approach to solving MILPs is to use the branch-and-bound algorithm, solving linear programming subproblems until the difference between upper and lower bounds on the solution are reduced to within some optimality gap ϵ [56, 57, 58]. However, this algorithm is generally even more computationally complex than the previously discussed methods, taking even more time to arrive at a solution. Solving the MILP formulation (VSP-MILP) efficiently and in time practicable for use in graph partitioning is still an open problem.

3.4.2 Quadratic Programming Approaches

Both the edge separator and vertex separator problems can be formulated as bilinear (nonconvex) quadratic programming problems. The QP formulation for the vertex separator problem is shown below (extended from [59]):

$$\begin{array}{lll} \text{maximize} & \boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y}) & \\ \text{subject to} & \boldsymbol{x}^{\mathsf{T}}\boldsymbol{y} = 0 & \text{(Exclusivity Constraint)} \\ & \boldsymbol{x}^{\mathsf{T}}\boldsymbol{A}\boldsymbol{y} = 0 & \text{(Separation Constraint)} \\ & \boldsymbol{0} \leq \boldsymbol{x} \leq \boldsymbol{1} \quad \boldsymbol{0} \leq \boldsymbol{y} \leq \boldsymbol{1} & \\ & \ell_x \leq \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} \leq u_x \quad \ell_y \leq \ \boldsymbol{w}^{\mathsf{T}}\boldsymbol{y} \leq u_y & \end{array} \qquad \text{(VSP-QPQC)}$$

$\boldsymbol{w}$ is an array of vertex weights, and $\boldsymbol{A}$ is the binary adjacency matrix for the graph. ℓ_x and ℓ_y are the lower bounds on the size of each part, usually $\ell_x = \ell_y = 1$. u_x and u_y are the upper bounds on the size of each part, usually $u_x = u_y = \frac{2}{3}|V|$. x is a binary array such that $x_i = 1$ if vertex i is in part X, and zero otherwise. $\boldsymbol{y}$ is the analog to $\boldsymbol{x}$ with respect to part Y.

The objective function serves to maximize the size of the parts (thus minimizing the size of the separator). The first constraint enforces that each vertex is in either one part or the separator, but never both parts simultaneously. For reference, we will call this the *exclusivity constraint*. The second constraint enforces that there does not exist an edge between one part and the other. As this constraint enforces the separation of the parts, we will call this the *separation constraint*. The last constraints limit the size of the parts and also bound $\boldsymbol{x}$ and $\boldsymbol{y}$ between 0 and 1. To see how these constraints correspond to the vertex separator problem, see Figure 3.2.

This quadratically constrained quadratic programming problem (or QCQP) can be reformulated

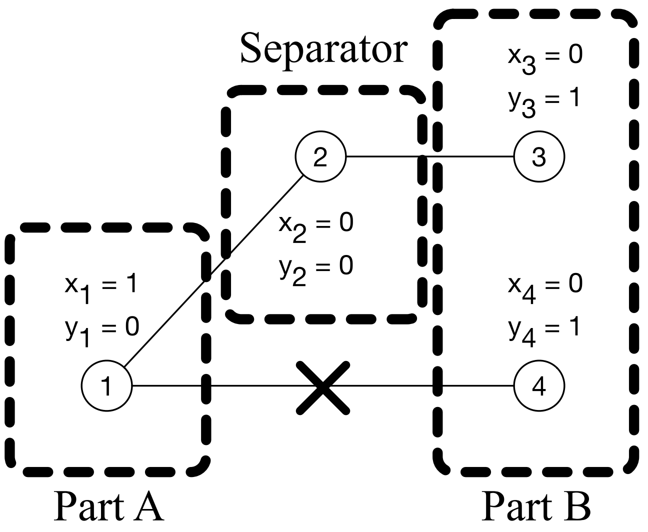


Figure 3.2: Mapping of graph connectivity to optimization formulation values

To represent the graph as an optimization problem formulation, we introduce an $\boldsymbol{x}$ and a $\boldsymbol{y}$ vector with an element for each vertex. Note that because vertices 1 and 2 are connected by an edge in the graph, $A_{1,2} = 1$, and so $x_1 A_{1,2} y_2 = 0$. Alternatively, note how this constraint is violated in the case of vertices 1 and 4 ($x_1 A_{1,4} y_4 = 1 * 1 * 1 = 1 \neq 0$).

as a linearly constrained quadratic programming problem using a penalty function:

$$\begin{array}{ll} \text{maximize} & \boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y}) - \gamma \boldsymbol{x}^{\mathsf{T}}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{y} \\ \text{subject to} & \boldsymbol{0} \leq \boldsymbol{x} \leq \boldsymbol{1} \quad \boldsymbol{0} \leq \boldsymbol{y} \leq \boldsymbol{1} \\ & \ell_x \leq \boldsymbol{1}^{\mathsf{T}}\boldsymbol{x} \leq u_x \quad \ell_y \leq \boldsymbol{1}^{\mathsf{T}}\boldsymbol{y} \quad \leq u_y \end{array} \tag{VSP-QPLC}$$

Note that for large enough γ, the penalty function is equivalent to the first and second constraints in Problem VSP-QPQC. Unfortunately, in both cases the QP is nonconvex, and finding a global optimum is NP-hard. Using a nonlinear programming (NLP) solver on either formulation generally results in poor solutions. However, several techniques have been developed to find better binary solutions via perturbation [59, 60].

3.4.2.1 Solution Methods

As discussed in Section 1.7.2, quadratic programming is generally easy to solve if the problem is convex, but NP-hard when nonconvex. These quadratic programming problems (VSP-QPQC and VSP-QPLC) are both nonconvex and a special case of quadratic programming called *bilinear programming*. In bilinear programming, the variables can be divided into two disjoint subsets such that the quadratic term $\frac{1}{2}\boldsymbol{x}^\mathsf{T}\boldsymbol{G}\boldsymbol{x}$ becomes linear when either subset is held constant.

Methods for solving bilinear programming problems include spatial subdivision methods, an extension of branch-and-bound algorithms that solve to ϵ-optimality [61, 62], and bilevel methods, which work by decomposing the bilinear program into linear programs by fixing a subset of variables [63, 64, 65]. As simply alternating between linear subproblems does not guarantee convergence to an optimum (global or otherwise), bilevel methods often use a variety of techniques to improve their solutions, such as cutting planes [66] and solving the dual of the problem [63]. However, as the underlying problem is NP-hard, these algorithms either do not guarantee optimality or do not run in polynomial time in the worst case. Some bilevel methods, specifically Konno's algorithm [66], can be used to find an improved solution quickly by running for only a small number of iterations, a fact which we exploit later in Section 3.6.6.

3.5 Generalized Gains

Despite these recent developments in optimization formulations of the vertex separator problem, implementations generally scale poorly [60]. One difficulty that we address here is the conversion between a continuous partition ($\boldsymbol{x} \in [0,1]^n$, $\boldsymbol{y} \in [0,1]^n$) and a discrete one ($\boldsymbol{x} \in \{0,1\}^n$, $\boldsymbol{y} \in \{0,1\}^n$). This conversion process is often costly to compute, as are the associated properties for solving the quadratic programming formulation VSP-QPLC such as objective function values. Ideally, we would like to have a set of common metrics from which we can efficiently compute properties about the partition regardless of the continuous or discrete state of the partition. Specifically, we aim to avoid recomputing the objective function of the quadratic programming formulation and the gain values used in the discrete Fiduccia-Mattheyses algorithm, as both require $O(|E|)$ time to compute with no prior information.

Our goal is to derive a set of general gains that should be equivalent to the Fiduccia-Mattheyses gains for vertices in the separator, but should also be applicable to vertices with continuous values of $\boldsymbol{x}$ and $\boldsymbol{y}$.

First, both the quadratic programming objective function and Fiduccia-Mattheyses gains address the fact that a vertex i moving out of the separator will decrease the weight of the separator by its weight w_i. Thus, we take the first term of the quadratic programming objective function $\boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y})$ as our starting point.

$$f = \boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y}) \tag{3.1}$$

Next, we consider that a vertex cannot be in both the X part and the Y part simultaneously. In the Fiduccia-Mattheyses algorithm, this is assumed: vertices have discrete states of being in one part or the other. However, in the quadratic programming formulation, this prohibition is addressed with a penalty term in the objective function: $-\gamma\boldsymbol{x}^{\mathsf{T}}\boldsymbol{I}\boldsymbol{y}$, where γ is a sufficiently large penalty parameter, $\boldsymbol{x}$ and $\boldsymbol{y}$ are $n \times 1$ vectors, and $\boldsymbol{I}$ is the $n \times n$ identity matrix. Thus, this penalty is zero only when $\boldsymbol{x}$ and $\boldsymbol{y}$ are orthogonal ($\boldsymbol{x} \perp \boldsymbol{y}$).

$$f = \boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y}) - \gamma\boldsymbol{x}^{\mathsf{T}}\boldsymbol{I}\boldsymbol{y} \tag{3.2}$$

Lastly, we incur a penalty for moving a vertex out of the separator into a part (e.g. X) who has neighbors in the opposite part (e.g. Y). The Fiduccia-Mattheyses gains assume that a vertex moving in this way will immediately draw the violating neighbors into the separator. For example, if a vertex moves from the separator to X, then all of its neighbors in Y will move to the separator S. Once again, the quadratic programming formulation must address this in continuous terms, so another penalty parameter is introduced: $-\gamma\boldsymbol{x}^{\mathsf{T}}\boldsymbol{A}\boldsymbol{y}$, where γ is the same sufficiently large penalty parameter from before, $\boldsymbol{x}$ and $\boldsymbol{y}$ are $n \times 1$ vectors, and $\boldsymbol{A}$ is the $n \times n$ graph adjacency matrix.

$$f = \boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y}) - \gamma\boldsymbol{x}^{\mathsf{T}}\boldsymbol{I}\boldsymbol{y} - \gamma\boldsymbol{x}^{\mathsf{T}}\boldsymbol{A}\boldsymbol{y} \tag{3.3}$$

$$f = \boldsymbol{w}^{\mathsf{T}}(\boldsymbol{x}+\boldsymbol{y}) - \gamma\boldsymbol{x}^{\mathsf{T}}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{y} \tag{3.4}$$

This is the quadratic programming formulation objective function. Note that this function is defined globally over all vertices in the graph.

To derive the generalized gains, we take the expanded version of the objective function and take partial derivatives with respect to $\boldsymbol{x}$ and $\boldsymbol{y}$ to yield the gradient of the objective function:

$$\nabla f_{obj} = \begin{bmatrix} \dfrac{\partial f_{obj}}{\partial \boldsymbol{x}} \\ \dfrac{\partial f_{obj}}{\partial \boldsymbol{y}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}^{\mathsf{T}} - \gamma \boldsymbol{I} \boldsymbol{y} - \gamma \boldsymbol{A} \boldsymbol{y} \\ \boldsymbol{w}^{\mathsf{T}} - \gamma \boldsymbol{I} \boldsymbol{x} - \gamma \boldsymbol{A} \boldsymbol{x} \end{bmatrix} \tag{3.5}$$

We can now compare these expressions with the Fiduccia-Mattheyses gains:

$$G_x = w_i - \sum_{j \in N_y(i)} w_j \tag{3.6}$$

$$G_y = w_i - \sum_{j \in N_x(i)} w_j \tag{3.7}$$

where w_i is the weight for vertex i, and $N_x(i)$ and $N_y(i)$ are the sets of neighbor vertices adjacent to vertex i in parts X and Y respectively. First, we note that unlike the gradient of the quadratic programming objective function, the Fiduccia-Mattheyses gains are only defined for individual vertices that are located in the separator. We can determine the contribution of a vertex i toward the objective function by looking at the i'th element of the gradient:

$$\nabla f_i = \begin{bmatrix} w_i - \gamma y_i - \gamma \boldsymbol{A}_{*,i} \boldsymbol{y} \\ w_i - \gamma x_i - \gamma \boldsymbol{A}_{*,i} \boldsymbol{x} \end{bmatrix} \tag{3.8}$$

The term $\gamma \boldsymbol{A}_{*,i} \boldsymbol{y}$ can also be expressed as a summation, $\gamma \sum_{j \in N(i)} y_j$, where $N(i)$ is the set of vertices adjacent to vertex i, also known as the closed neighborhood of i. The notation $\boldsymbol{A}_{*,i}$ refers to the i'th column of the binary adjacency matrix $\boldsymbol{A}$.

$$\nabla f_i = \begin{bmatrix} w_i - \gamma y_i - \gamma \sum\limits_{j \in N(i)} y_j \\ w_i - \gamma x_i - \gamma \sum\limits_{j \in N(i)} x_j \end{bmatrix} \tag{3.9}$$

We now address the penalty parameter, γ. The choice of γ is arbitrary so long as it is sufficiently large, so that in solving the quadratic programming formulation, solutions that represent invalid separators are penalized disproportionately compared to valid separators (ones that do not violate the exclusivity and separation constraints). In the Fiduccia-Mattheyses gains, these violations cannot occur, and so the penalty is directly proportional to the gain of each vertex.

If we relax this penalty to resemble the Fiduccia-Mattheyses gains by replacing γ with vertex weights, the gains will no longer strictly enforce the exclusivity and separation constraints. However, as γ need only be sufficiently large, we can easily re-introduce γ to once again enforce these constraints. For now, we will make this substitution, yielding *generalized gains*:

$$G_{x,i} = w_i - w_i y_i - \sum_{j \in N(i)} w_j y_j \tag{3.10}$$

$$G_{y,i} = w_i - w_i x_i - \sum_{j \in N(i)} w_j x_j \tag{3.11}$$

Similar to the Fiduccia-Mattheyses gains, G_x and G_y can be said to describe the effect of moving a vertex to the X part or Y part, respectively, but are applicable to continuous and invalid partitions as well.

3.5.1 Fiduccia-Mattheyses Gains as a Special Case

Note that the Fiduccia-Mattheyses gains are now a special case of the generalized gains. If we impose the same restrictions on $\boldsymbol{x}$ and $\boldsymbol{y}$ that the Fiduccia-Mattheyses algorithm imposes (i.e. $\boldsymbol{x} \in \{0,1\}^n$, $\boldsymbol{y} \in \{0,1\}^n$, and $\boldsymbol{x} \perp \boldsymbol{y}$), we can remove the exclusivity term, and the generalized gains become

$$G_{x,i} = w_i - \sum_{j \in N(i)} w_j y_j \tag{3.12}$$

$$G_{y,i} = w_i - \sum_{j \in N(i)} w_j x_j \tag{3.13}$$

We can simplify further by noting that $x_j = 1$ for all vertices in $N_x(i)$, and likewise that $y_j = 1$

for all vertices in $N_y(i)$, yielding the Fiduccia-Mattheyses gains exactly:

$$G_{x,i} = w_i - \sum_{j \in N_y(i)} w_j \tag{3.14}$$

$$G_{y,i} = w_i - \sum_{j \in N_x(i)} w_j \tag{3.15}$$

3.5.2 Determining Separation and Exclusivity Violations

While the generalized gains are useful for continuous x and y, they can also be used to detect which vertices are in violation of the separation and exclusivity constraints. When considering such vertices, we note that valid, discretely partitioned vertices fall into three cases:

- **Case 1**. A vertex in part X ($x_i = 1, y_i = 0$) with no neighbors in Y ($y_j = 0 \; \forall \; j \in N(i)$).
- **Case 2**. A vertex in part Y ($x_i = 0, y_i = 1$) with no neighbors in X ($x_j = 0 \; \forall \; j \in N(i)$).
- **Case 3**. A vertex in part S ($x_i = 0, y_i = 0$).

For case 1, the generalized X gain is equal to w_i:

$$G_{x,i} = w_i - w_i y_i - \sum_{j \in N(i)} w_j y_j = w_i \tag{3.16}$$

For case 2, the generalized Y gain is also equal to w_i:

$$G_{y,i} = w_i - w_i x_i - \sum_{j \in N(i)} w_j x_j = w_i \tag{3.17}$$

For case 3, we can only say that $G_x \leq w_i$ and $G_y \leq w_i$.

If a vertex is not in the separator ($x_i > 0$ and/or $y_i > 0$), we would expect that at least one of the generalized gains $G_{x,i}$ or $G_{y,i}$ would be equal to w_i. If this is not the case, one or both of the separation and exclusivity constraints is in violation.

For example, if a vertex is violating the exclusivity constraint with $x_i = 0.5$ and $y_i = 1$, the generalized gains will be

$$G_{x,i} = w_i - w_i(1) - \sum_{j \in N(i)} w_j y_j < w_i \tag{3.18}$$

$$G_{y,i} = w_i - w_i(0.5) - \sum_{j \in N(i)} w_j x_j < w_i \tag{3.19}$$

Alternatively, perhaps a vertex in the X part ($x_i = 1, y_i = 0$) has a neighbor v_k partly in the Y part ($x_k = 0, y_k = 0.3$). This violation of the separation constraint is reflected in the generalized gains:

$$G_x = w_i - w_i(0) - \sum_{j \in N(i) \backslash k} w_j y_j - w_k(0.3) < w_i \tag{3.20}$$

$$G_y = w_i - w_i(1) - \sum_{j \in N(i) \backslash k} w_j x_j - w_k(0) < w_i \tag{3.21}$$

Thus, we can quickly (in $O(1)$ time) identify which vertices are not in a discrete part by examining the current values of the generalized gains. Such violations can be resolved by moving the invalid vertex into the separator ($x_i = y_i = 0$), and so a valid partitioning can be generated in linear ($O(|V|)$) time, assuming at least one vertex is left in each of the X and Y parts.

3.5.3 Computing the Quadratic Programming Objective Function

As previously discussed, the generalized gains do not strictly enforce the separation and exclusivity constraints. However, we can compute an approximation of the ith element of the objective function gradient in constant time. We start with the generalized x gain for a vertex i:

$$G_{x,i} = w_i - w_i y_i - \sum_{j \in N(i)} w_j y_j \tag{3.22}$$

Recalling that the penalty parameter need only be sufficiently large, we multiply through by γ:

$$\gamma G_{x,i} = \gamma w_i - \gamma w_i y_i - \sum_{j \in N(i)} \gamma w_j y_j \tag{3.23}$$

Note that for $w_i \geq 1$, $\gamma w_i \geq \gamma$, implying that if γ is sufficiently large, so is $\gamma'_i = \gamma w_i \; \forall \; i \in V$. Thus, if we assume $w_i \geq 1 \; \forall \; i \in V$, we can therefore simplify the above expression to the following:

$$\gamma G_{x,i} = \gamma w_i - \gamma'_i y_i - \sum_{j \in N(i)} \gamma'_j y_j \tag{3.24}$$

We now correct for the first term containing w_i to yield the objective function gradient:

$$\nabla_x \tilde{f}_i = w_i(1-\gamma) + \gamma w_i - \gamma'_i y_i - \sum_{j \in N(i)} \gamma'_j y_j \tag{3.25}$$

$$\nabla_x \tilde{f}_i = w_i(1-\gamma) + \gamma G_{x,i} \tag{3.26}$$

The same process can be used to derive the analogous formula for the gradient approximation with respect to y:

$$\nabla_y \tilde{f}_i = w_i(1-\gamma) + \gamma G_{y,i} \tag{3.27}$$

We can now compute an approximation of the gradient to the quadratic programming objective function in constant time from the generalized gains. Note that this approximation penalizes violations between heavily weighted vertices more than violations between lighter weighted vertices, unlike the original formulation, which penalizes all violations equally regardless of vertex weight.

3.5.4 Update Formulas for Generalized Gains

To compute the generalized gains for a vertex requires data from all vertices neighboring that vertex. Thus, computing gains for each vertex requires $O(|N(i)|)$ time, and computing gains for all vertices in the graph requires $O(|E|)$ time. This can quickly become intractable for very large graphs, especially as vertices are frequently moved from one part to another in search of better solutions.

Rather than compute the generalized gains after each modification, we can derive update formulas to more efficiently compute the new gains G'_x and G'_y given previous gains G_x and G_y. We begin with the x gain G_x and a modification $y'_i = y_i + \Delta y_i$:

$$G_x = w_i - w_i(y_i + \Delta y_i) - \sum_{j \in N(i)} w_j y_j \tag{3.28}$$

$$G'_x = w_i - w_i y'_i - \sum_{j \in N(i)} w_j y_j \tag{3.29}$$

$$G'_x = w_i - w_i y_i - w_i \Delta y_i - \sum_{j \in N(i)} w_j y_j \tag{3.30}$$

$$G'_x = G_x - w_i \Delta y_i \tag{3.31}$$

We can derive a similar formula for updating a vertex whose neighbor has changed with a modification $y'_j = y_j + \Delta y_j$:

$$G_x = w_i - w_i(y_i + \Delta y_i) - \sum_{j \in N(i)} w_j y_j \tag{3.32}$$

$$G'_x = w_i - w_i y_i - \sum_{j \in N(i)} w_j y'_j \tag{3.33}$$

$$G'_x = w_i - w_i y_i - \sum_{j \in N(i)} w_j (y_j + \Delta y_j) \tag{3.34}$$

$$G'_x = w_i - w_i y_i - \sum_{j \in N(i)} w_j y_j + w_j \Delta y_j \tag{3.35}$$

$$G'_x = G_x + w_j \Delta y_j \tag{3.36}$$

Analogous formulas can be derived for the y gain G_y:

$$G'_y = G_y - w_i \Delta x_i \tag{3.37}$$

$$G'_y = G_y + w_j \Delta x_j \tag{3.38}$$

These updates can all be computed in constant ($O(1)$) time. Therefore, for each vertex move, the gains for that vertex and all of its neighbors can be computed in $O(|N(i)|)$ time.

3.6 Algorithmic Description

The aforementioned generalized gains were used as a foundation for a new heuristic for solving the vertex separator problem over arbitrary graphs. The only restrictions placed on the graphs are that they must have positive vertex weights (or no vertex weights, in which case the weights are assumed to be 1). The following sections describe the primary portions of the algorithm, followed by an explanation of how they were combined and implemented as an extension to Mongoose, our graph partitioning library written in C++.

3.6.1 Heuristic Cost Metric

In practice, graph partitioning is a multiobjective optimization problem, attempting to compute both a small separator and a balanced partition. To simplify our algorithm and address this issue, we introduce a single *heuristic cost metric*:

$$f_{heuristic} = \begin{cases} \sum_{i \in V} w_i \left(\max\{1 - x_i - y_i, 0\} \right) + \psi H & \text{if } \psi > \text{imbalance tolerance} \\ \sum_{i \in V} w_i \left(\max\{1 - x_i - y_i, 0\} \right) + \psi w_{min} & \text{otherwise} \end{cases} \tag{3.39}$$

In this metric, $\sum_{i \in V} w_i \max(1 - x_i - y_i, 0)$ represents the weight of all vertices in the separator, H is a large penalty of $3w_{max}$, and ψ is a measure of imbalance defined as follows:

$$\psi = \left| p_{target} - \frac{\min(|X|, |Y|)}{|X| + |Y|)} \right| \tag{3.40}$$

By default, $p_{target} = 0.5$, representing an ideal 50% split between X and Y, although p_{target} can be specified in the interval $(0, 0.5]$. We use the minimum of $|X|$ and $|Y|$ so $\frac{\min(|X|,|Y|)}{|X|+|Y|)}$ is guaranteed to be in the same range. Thus, if $p_{target} = 0.5$ and $X = Y$, then $\psi = 0$. The default imbalance tolerance is 0.05.

This imbalance penalty has the following properties (see Figure 3.3):

- With $\psi \leq$ tolerance, the imbalance penalty is between zero and w_{min}. In this range, the penalty will never be large enough to cause the heuristic cost to choose a larger separator over a smaller one, but does penalize less balanced partitions slightly more than more balanced ones.

- With $\psi >$ tolerance, the imbalance penalty is very large. While there are cases where a partition will be found whose imbalance exceeds the tolerance, the relative improvement must be very significant. This is particularly useful during intermediate steps of the algorithm when the graph is very coarse and the balance constraint may be difficult to satisfy.

With this heuristic cost metric, we can compare any two partitions and assess their quality. We aim to minimize this cost metric at all steps in the algorithm, saving better partitions and rejecting new partitions that evaluate to a larger heuristic cost.

3.6.2 Pre-Processing and Coarsening

A simple undirected input graph is accepted in the form of a symmetric sparse matrix in compressed sparse column format. Edge weights are assumed to be binary for the vertex separator problem, and vertex weights are assumed to be 1 if not present; if present, vertex weights are made positive using an absolute value transformation. Starting statistics for the graph are computed, including minimum and maximum vertex weight. Memory for the $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors is allocated, along with space for the the X and Y heaps (discussed in Section 3.6.4).

The graph is then coarsened until the coarsened graph is reduced to a user-defined number of vertices (128 vertices by default).

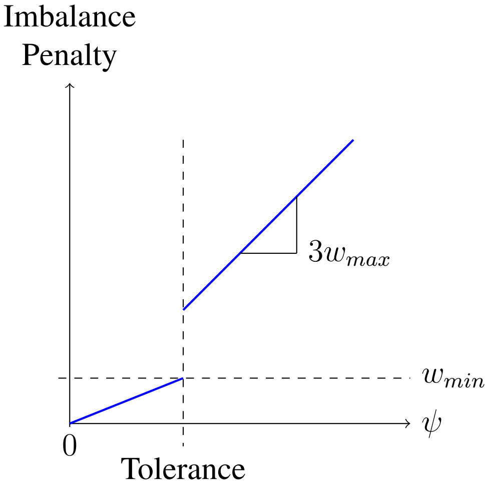
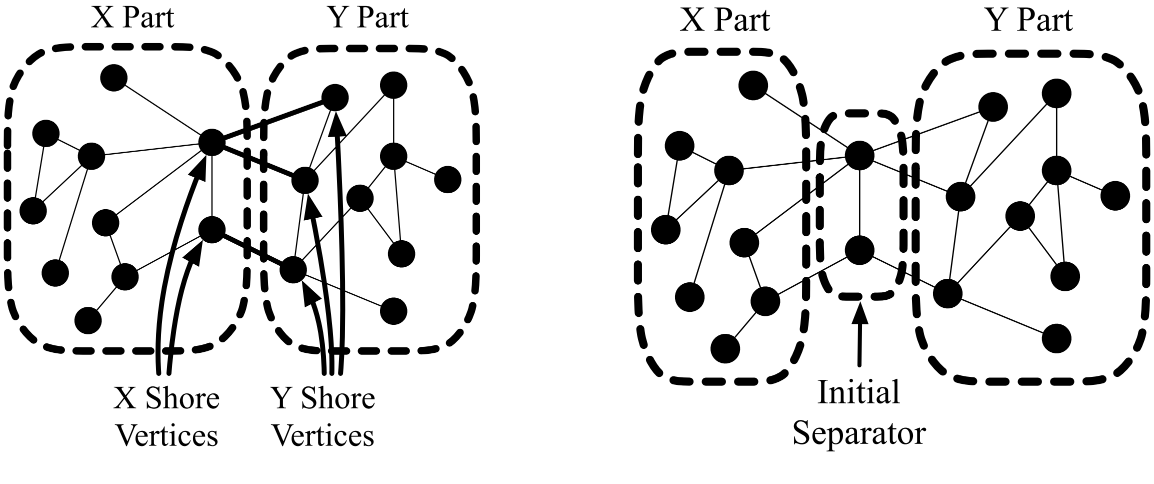


Figure 3.3: Plot of imbalance penalty as a function of imbalance



(a)

(b)

Figure 3.4: Deriving an initial vertex separator from an edge cut.

Using the methods described in Chapter 2, an edge cut is computed for the graph (a). The vertex weights of each set of shore vertices are summed, and the set with the least weight is chosen as the initial separator part (b).

3.6.3 Initial Separator Selection

Once the graph has been coarsened to a sufficiently small size, an initial vertex separator is computed. First, an edge cut is computed for the graph using the methods described in Chapter 2. The vertex weights of the vertices in the shores of the edge cut are summed, and the shore with the least weight is chosen as the initial separator part (see Figure 3.4).

It is possible, especially for initially large graphs, for this coarsened graph to be a clique. Thus, a vertex separator does not exist. After the initial separator is computed, if there are zero vertices in the X or Y parts, the initial partitioning fails. The graph is then uncoarsened once and the initial partitioning is attempted again. This process is repeated until the partitioning succeeds, producing three parts (X, Y, and S) with at least one vertex in each.

3.6.4 Uncoarsening and Refinement Loop

Once an initial separator is computed, the graph is repeatedly uncoarsened, with the coarse solution (which is always a valid, discrete vertex separator) being projected onto the uncoarse (or parent) graph. After this solution is projected, the vertex separator is refined using several techniques. This refinement loop is done twice by default, but can be done as many times as needed to obtain a desired quality (at the cost of runtime). Because of the combination of continuous optimization methods and combinatoric approaches, this set of heuristics is termed a hybrid algorithm.

At the beginning of each refinement loop, two max-heaps are built: an X heap and a Y heap. All vertices *not* in the X part are added to the X heap, and likewise all vertices *not* in the Y part are added to the Y heap. The heaps are keyed by the generalized X and Y gains, respectively, such that the root of the X heap contains the vertex with the largest X gain that is also not in the X part.

3.6.5 Greedy Knapsack Algorithm

Immediately following uncoarsening and projection, it is common for the initial solution to be trivially suboptimal, meaning vertices that had been in a minimal separator in the coarse graph are easily moved to the X or Y parts in the uncoarse graph. To address this, we first treat each part as an instance of the knapsack problem. We can greedily move vertices at the top of the X heap into the X part from the separator, and likewise with the Y heap and Y part, until either the best gain becomes negative or such a move would violate the upper or lower bounds on the X and Y parts. This is a variation of George Dantzig's greedy algorithm for the knapsack problem [67].

Algorithm 1 Greedy Knapsack Packing Algorithm

```
Input: $\boldsymbol{x} \in [0,1]^n$; $\boldsymbol{y} \in [0,1]^n$; valid $X$ and $Y$ heaps
Output: $f_{heuristic,out} \leq f_{heuristic,in}$
  improvement ← true
  while improvement do
    improvement ← false
    for each part $P$ do
      select first unmarked vertex $i$ from top of corresponding heap
      if moving vertex $i$ to part $P$ decreases heuristic cost $f_{heuristic}$ then
        improvement ← true
        mark vertex $i$
        move vertex $i$ to part $P$
        update gains for vertex $i$ and $N(i)$
      end if
    end for
  end while
```

3.6.6 Quadratic Programming with Gamma Reduction

We now use the quadratic programming formulation of the vertex separator problem to further optimize the partition. Following the general approach described in Hager and Hungerford (2015) and Hager et al. (2018) [59, 60], we solve the quadratic programming problem using the mountain climbing algorithm, a version of successive linear programming (SLP).

As described in Section 3.4.2, the quadratic programming formulation is nonconvex, despite having relatively straightforward linear box constraints. We can solve this QP (to local optimality) using what is known as a mountain climbing algorithm. If we fix $\boldsymbol{y}$ to $\boldsymbol{y}_{fixed}$, the quadratic programming problem simplifies to a linear programming problem (LP):

$$\begin{aligned} \text{maximize} \quad & \left[\boldsymbol{w} - \gamma(\boldsymbol{A} + \boldsymbol{I})\boldsymbol{y}_{fixed}\right]^{\mathsf{T}} \boldsymbol{x} \\ \text{subject to} \quad & \mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1} \\ & \ell_x \leq \mathbf{1}^{\mathsf{T}}\boldsymbol{x} \leq u_x \end{aligned} \tag{VSP-LP}$$

An analogous LP for a fixed $\boldsymbol{x}$ vector is solved in succession with VSP-LP until the first order optimality conditions are satisfied or an iteration limit is reached. This is the mountain (or hill) climbing algorithm (or MCA) described previously [66, 60]; see Algorithm 2. While more sophisticated optimization techniques exist for optimizing nonconvex QPs, this method is both simple to implement and fast to execute.

However, we are still left with the question of how to solve the LPs. Thankfully, we can use the same solver used in Chapter 2 when designing the hybrid algorithm for edge cuts. NAPHEAP is capable of solving separable convex quadratic knapsack problems, of which the LPs described are a subset. Thus, we chose to use NAPHEAP to solve the successive LPs.

NAPHEAP requires the objective function in the form of a cost vector; in other words, the gradient of the objective function with respect to the variables being optimized. As discussed in Section 3.5.3, we can compute an approximation of the objective function gradient for the quadratic programming formulation in linear time:

$$\nabla_x \tilde{f}_i = w_i(1-\gamma) + \gamma G_{x,i} \tag{3.26}$$

$$\nabla_y \tilde{f}_i = w_i(1-\gamma) + \gamma G_{y,i} \tag{3.27}$$

3.6.6.1 Gamma Reduction

Solving the quadratic programming problem with a sufficiently large penalty γ often results in poor solutions, forcing variables to their bounds to avoid constraint violations. To combat this, and to help better explore the nonconvex search space, the process of gamma reduction was proposed [60]. The QP is first solved (via the mountain climbing algorithm) with a reduced penalty

Algorithm 2 Mountain Climbing Algorithm (MCA)

```
Input: $\gamma > 0$; $\boldsymbol{x} \in [0,1]^n$; $\boldsymbol{y} \in [0,1]^n$
  while optimality conditions are not satisfied and maximum iterations not reached do
    solve VSP-LP for fixed $y$ and variable $x$
    solve VSP-LP for fixed $x$ and variable $y$
  end while
```

parameter γ, allowing for violations of the separation and exclusivity constraints. Then, the QP is solved again with a sufficiently large γ, usually w_{max}, using the previous solution as a starting point, resulting in a solution with few or no constraint violations. This acts as a perturbation with the potential to discover better separators that are not near the current separator.

The overall algorithm described in this section is outlined in Algorithm 3.

Algorithm 3 Quadratic Programming Algorithm with Gamma Reduction

```
  $\gamma_{reduced} \leftarrow w_{max}$
  for a specified number of iterations do
    $(x', y') \leftarrow$ MCA($\gamma_{reduced}$, $x$, $y$)
    $(x^*, y^*) \leftarrow$ MCA($w_{max}$, $x'$, $y'$)
    if heuristic cost is improved then
      save current partition and gains
      $\gamma_{reduced} \leftarrow w_{max}$
    else {heuristic cost is worse}
      restore previous partition and gains
      $\gamma_{reduced} \leftarrow \frac{1}{2}\gamma_{reduced}$
    end if
  end for
```

3.6.7 Continuous Fiduccia-Mattheyses Algorithm

After using the quadratic programming formulation to optimize the current partition, there may be many elements of the $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors that are not at their binary bounds. To address this, we have developed a version of the Fiduccia-Mattheyses algorithm that works with the generalized gains and continuous $\boldsymbol{x}$ and $\boldsymbol{y}$.

First, the vertices with the greatest generalized X and Y gains (at the top of the X and Y heaps) are considered for a move to the X or Y parts, respectively. For simplicity, we only consider moves that will strictly improve the partition, although one could extend this algorithm to make moves that do not improve the partition to better explore the solution search space. Each of these moves forces a vertex to either $x_i = 0, y_i = 1$ or $x_i = 1, y_i = 0$, decreasing the size of the separator and removing any exclusivity violations. As in the traditional Fiduccia-Mattheyses algorithm, once a vertex is moved, it cannot be moved again during that iteration.

The iteration terminates when no more such moves exist (i.e. all potential vertices have either been moved that iteration, or moving them would make the heuristic cost metric worse). The algorithm is repeated until no improvements are made during an iteration (see Algorithm 4). While similar, this algorithm is notably different from the previously discussed greedy knapsack packing algorithm (Section 3.6.5). Instead of alternating between the X and Y parts, this continuous FM algorithm moves whichever vertex has the highest gain in *either* of the X or Y heaps. Thus, in this algorithm, it is possible for several vertices to be successively moved to the same part (if $\max G_x > \max G_y$), while in the greedy knapsack packing, this is generally not the case.

3.6.8 Rectification

Up to this point in the refinement loop, we have assumed continuous values of $\boldsymbol{x}$ and $\boldsymbol{y}$, but we ultimately wish to apply the traditional Fiduccia-Mattheyses algorithm and return a valid (discrete) vertex separator. Using the relationships described in Section 3.5.2, we can quickly iterate through the vertices in $O(n)$ time and move any vertices with violations into the separator, eliminating such violations (see Algorithm 5 and Figure 3.5).

For all vertices not strictly in the separator, we check that at least one of the generalized gains $G_{x,i}$ or $G_{y,i}$ is equal to w_i. If this is not the case, we move the given vertex to the separator ($x_i = y_i = 0$). Because we check that at least one of the gains must be *equal* to w_i, this also moves vertices that are partially in a part to the separator (e.g. $x_i = 0$ and $y_i = 0.5$).

Algorithm 4 Continuous Fiduccia-Mattheyses Algorithm

```
Input: $\boldsymbol{x} \in [0,1]^n$; $\boldsymbol{y} \in [0,1]^n$; valid $X$ and $Y$ heaps
Output: $f_{heuristic,out} \leq f_{heuristic,in}$
  improvement ← true
  while improvement do
    improvement ← false
    unmark all vertices
    select first unmarked vertices $i$ and $j$ from top of $X$ and $Y$ heaps, respectively
    if $G_{x,i} > G_{y,j}$ and moving vertex $i$ to part $X$ decreases heuristic cost $f_{heuristic}$ then
      improvement ← true
      mark vertex $i$
      move vertex $i$ to part $X$
      update gains for vertex $i$ and $N(i)$
    else if moving vertex $j$ to part $Y$ decreases heuristic cost $f_{heuristic}$ then
      improvement ← true
      mark vertex $j$
      move vertex $j$ to part $Y$
      update gains for vertex $j$ and $N(j)$
    end if
  end while
```

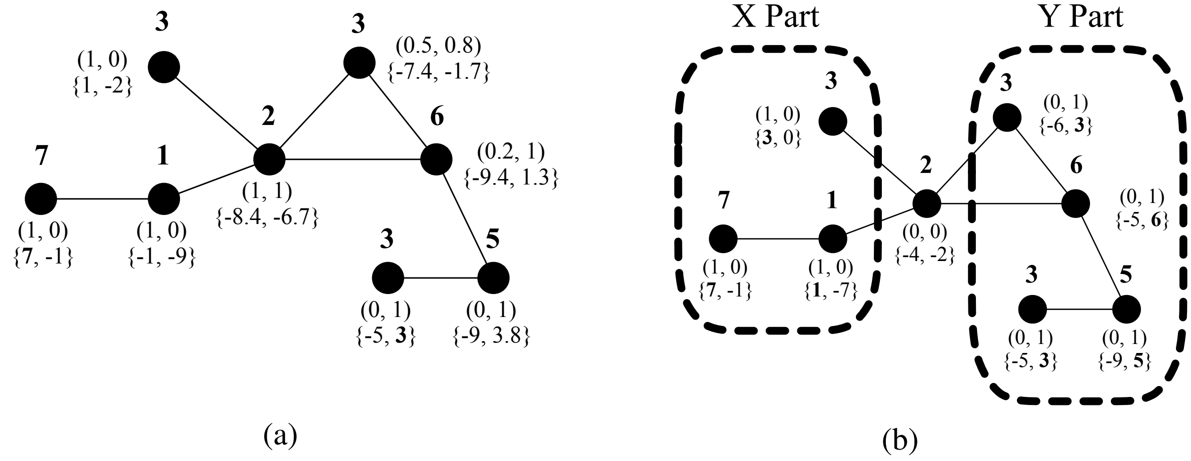


Figure 3.5: Rectification of an invalid vertex separator using generalized gains.

(a) An initial graph with continuous $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors, and (b) one possible rectification to eliminate non-binary and violations to the separation and exclusivity constraints. Vertex weights are shown in bold above each vertex, with the (x, y) values for each vertex given in parentheses and generalized gains $\{G_x, G_y\}$ given in brackets. Note that vertices with no violations (and not in the separator), one of its generalized gains is equal to the weight of the vertex (shown in bold).

Algorithm 5 Rectification

```
Input: $\boldsymbol{x} \in [0,1]^n$; $\boldsymbol{y} \in [0,1]^n$; valid $X$ and $Y$ heaps
Output: $(x_i, y_i) \in \{(0,0),(1,0),(0,1)\} \; \forall \; i \in V$
  for each vertex $i \in V$ do
    if $G_{x,i} < w_i$ and $G_{y,i} < w_i$ then
      move vertex $i$ to separator part $S$
      update gains for vertex $i$ and $N(i)$
    end if
  end for
```

3.6.9 Discrete Fiduccia-Mattheyses Algorithm

Now that we are guaranteed to have a discrete and valid vertex separator, we can apply the Fiduccia-Mattheyses algorithm [16]. The generalized gains now equate to the gains used in the Fiduccia-Mattheyses algorithm. The maximum gains from each of the X and Y heaps are compared, and the vertex corresponding to the greatest gain (subject to balance constraints) is moved to either the X or Y part. This is repeated for a user-defined number of moves, referred to as the search depth. Once the search depth is reached, the solution is rolled back to the best solution found during the search (see Algorithm 6).

It should be noted that this version of the Fiduccia-Mattheyses algorithm also acts as a perturbation on the partition by making moves that are not always immediately beneficial. Many moves may be made that increase the separator or increase imbalance before potentially locating an improvement.

3.6.10 Weight Perturbation

In an attempt to further explore the search space, we perturb the weight vector for vertices in the current separator to encourage the algorithm to seek other separators in the graph. One way to accomplish this is to simply add a penalty to the weights of some or all of the vertices in the current separator. However, it is possible that some of the vertices in the current separator are in the globally optimal separator, and should therefore not be penalized. Thus, the selection of vertices to penalize is purely heuristic.

We have chosen to penalize a randomly chosen subset of vertices in the separator with the

Algorithm 6 Discrete Fiduccia-Mattheyses Algorithm

```
Input: $(x_i, y_i) \in \{(0,0),(1,0),(0,1)\} \; \forall \, i \in V$; valid $X$ and $Y$ heaps
Output: $f_{heuristic,out} \leq f_{heuristic,in}$
  improvement ← true
  while improvement and iteration limit not reached do
    improvement ← false
    search_depth ← 0
    unmark all vertices
    while search_depth limit not reached do
      select first unmarked vertices $i$ and $j$ from top of $X$ and $Y$ heaps, respectively
      if $G_{x,i} > G_{y,j}$ then
        mark vertex $i$
        move vertex $i$ to part $X$
        update gains for vertex $i$ and $N(i)$
        for each vertex $k \in N(i) \cap Y$ do
          move vertex $k$ to separator part
          update gains for vertex $k$ and $N(k)$
        end for
      else if moving vertex $j$ to part $Y$ decreases heuristic cost $f_{heuristic}$ then
        mark vertex $j$
        move vertex $j$ to part $Y$
        update gains for vertex $j$ and $N(j)$
        for each vertex $k \in N(i) \cap X$ do
          move vertex $k$ to separator part
          update gains for vertex $k$ and $N(k)$
        end for
      end if
      if heuristic cost is improved then
        improvement ← true
        best_search_depth ← search_depth
      end if
    end while
    roll back to best_search_depth
  end while
```

following penalty:

$$w_{\text{penalty},i} = \frac{\max_w \boldsymbol{w}}{(\epsilon + |G_{x,i} - G_{y,i}|)} \tag{3.41}$$

where $\boldsymbol{w}$ is the vertex weight vector, $G_{x,i}$ and $G_{y,i}$ are the generalized X and Y gains for vertex i, and ϵ is some small positive value to avoid division by zero. Note that the gain term in the denominator is equivalent to the following expression:

$$\begin{align} |G_{x,i} - G_{y,i}| &= \left| \left(w_i - w_i y_i - \sum_{j \in N(i)} w_j y_j \right) - \left(w_i - w_i x_i - \sum_{j \in N(i)} w_j x_j \right) \right| \tag{3.42} \\ &= \left| \sum_{j \in N(i)} w_j x_j - \sum_{j \in N(i)} w_j y_j \right| \tag{3.43} \\ &= \left| \sum_{j \in N(i)} w_j (x_j - y_j) \right| \tag{3.44} \end{align}$$

For a vertex in the separator, this expression represents the imbalance in weight between its neighbors in each part. A large such imbalance may imply that the separator vertex is likely to be moved to another part, even without a weight penalty, while a vertex with a small imbalance may require a larger penalty to make it beneficial to remove from the separator. Hence, this imbalance term is placed in the denominator of the penalty.

This penalty function generally works well at perturbing the current solution, but many other heuristics could be used, such as using some function of vertex degree.

Algorithm 7 Weight Perturbation

```
for each vertex $i$ in the separator part do
    $w_i \leftarrow (w_i + w_{penalty,i})$
    update gains for vertex $i$ and $N(i)$
end for
```

3.7 Implementation

The multi-part algorithm described in the previous section was implemented in C++ as a significant extension to the graph partitioning library, Mongoose, described in Chapter 2. To ensure reliability and correctness of the codebase, the following software engineering tools and techniques were used in development:

- A thorough test suite written using Catch2 [68] resulting in near-total (97%+) code coverage.
- Continuous integration testing using Travis CI [69] to ensure compatibility with a variety of compilers and compiler versions on Linux and macOS.
- Static analysis using Cppcheck [70].
- Dynamic analysis using Valgrind [71] to detect and fix memory leaks and uninitialized variables.

Despite the increased efficiency of this implementation due to the generalized gains, the continuous portions of the algorithm described in Sections 3.6.5, 3.6.6, 3.6.7, and 3.6.8 are utilized only for graphs with no more than 10,000 vertices, mainly due to the computational complexity of solving the quadratic programming formulation using NAPHEAP and the mountain climbing algorithm. Using other techniques, such as other QP and LP solvers, may result in better performance and enable our algorithm to utilize the continuous portions of the algorithm on larger graphs.

Altogether, these techniques are combined into a multilevel hybrid vertex separator algorithm, described at a high level in Algorithm 8.

3.8 Computational Results

In this section, we examine the computational performance of Mongoose compared to METIS, the same graph partitioning library we compared our edge cut algorithm with. All experiments were run on a 24-core dual-socket 2.40 GHz Intel Xeon E5-2695 v2 system with 768 GB of memory. Only one thread was utilized, as both Mongoose and METIS are serial algorithms. All comparisons were conducted with METIS 5.1.0 and compiled with GCC 4.8.5 on CentOS 7.

Algorithm 8 Overall Multilevel Hybrid Vertex Separator Algorithm

```
Input: A simple, undirected, weighted graph $G = (V, E)$
Output: A vertex separator partition of $G$
  while $|V|$ > coarsen_limit do
    coarsen graph (Section 3.6.2)
  end while
  compute an initial vertex separator (Section 3.6.3)
  while graph can still be uncoarsened do
    for a set number of iterations do
      if $|V| < 1 \times 10^4$ then
        apply Greedy Knapsack Algorithm (Section 3.6.5)
        apply Quadratic Programming with Gamma Reduction (Section 3.6.6)
        apply Continuous Fiduccia-Mattheyses Algorithm (Section 3.6.7)
        apply Rectification (Section 3.6.8)
      end if
      apply Discrete Fiduccia-Mattheyses Algorithm (Section 3.6.9)
      if heuristic cost is improved then
        save current partition and gains
      else {heuristic cost is worse}
        restore previous partition and gains
      end if
      apply weight vector perturbation (Section 3.6.10)
    end for
  end while
```

Table 3.1: Performance comparison between Mongoose and METIS on all 2,778 graphs from (or formed from) the SuiteSparse Collection with imbalance tolerance of 20%.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	1414	101	1515
	Tie	683	8	691
	Mongoose	480	92	572
		2577	201	2778

For consistency, each partitioner was run five times for each problem. The highest and lowest times are removed, and the remaining three are averaged (i.e. a 40% trimmed mean). However, only a single trial is used if the partitioner took more than one hour to partition a given graph. Default options were used, and a target split of 50%/50% was used with tolerances of 1.5% and 20% to compare the effects of imbalance constraints.

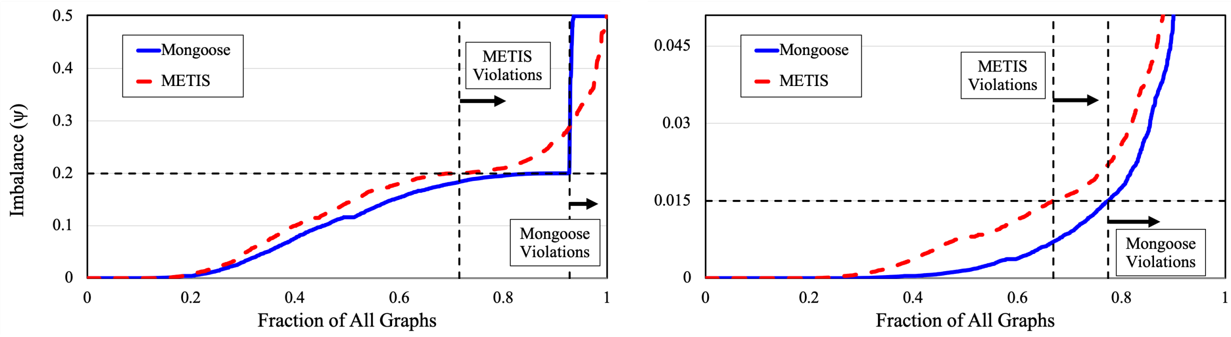
Our first comparison is on the entire SuiteSparse Matrix Collection [11]. As in Section 2.7, we filter the collection to remove complex matrices, and unsymmetric graphs are used to form an augmented system $\boldsymbol{B} = \begin{bmatrix} \mathbf{0}_{m,m} & \boldsymbol{A} \\ \boldsymbol{A}^T & \mathbf{0}_{n,n} \end{bmatrix}$. All diagonal values (self-edges) were removed, and the matrix was treated as a pattern matrix. All input weights were assumed to be 1. This yielded 2,778 graphs (93 matrices were added to the collection since the results in Chapter 2 were tabulated).

Tables 3.1 and 3.2 show the overall results comparing METIS and Mongoose with 20% and 1.5% imbalance tolerance, respectively. In the majority of cases ($\sim$51% for the 20% tolerance and $\sim$66% for the 1.5% tolerance), METIS provides a better quality cut in less time compared to Mongoose.

However, there are two flaws with this comparison. First, a number of results from both Mongoose and METIS violate the stated imbalance tolerances (see Figure 3.6). To address this point, we can choose to treat all solutions returned by either library in violation of the imbalance constraint as a failure. Revised tables taking into account such failures are shown in Tables 3.3 and

Table 3.2: Performance comparison between Mongoose and METIS on all 2,778 graphs from (or formed from) the SuiteSparse Collection with imbalance tolerance of 1.5%.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	1826	105	1931
	Tie	385	1	386
	Mongoose	385	76	461
		2596	182	2778



(a)

(b)

Figure 3.6: Imbalance comparison between Mongoose and METIS with different imbalance tolerances on all 2,778 graphs from the SuiteSparse Collection.

The imbalance of the reported vertex separator provided by both METIS and Mongoose with a specified imbalance tolerance of (a) 20% and (b) 1.5%. Note how both partitioners fail on a sizable subset of the graphs. METIS generally fails with smaller violations, but on more graphs overall (∼28–33%), while Mongoose treats the imbalance tolerance more strictly, failing on fewer graphs (∼7–23%) but with sometimes greater violations.

Table 3.3: Performance comparison between Mongoose and METIS on all 2,778 graphs from (or formed from) the SuiteSparse Collection with imbalance tolerance of 20%, treating imbalance violations as failures.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	973	51	1024
	Tie	674	5	679
	Mongoose	272	640	912
		1919	696	2615

3.4. In the 20% imbalance tolerance case, both METIS and Mongoose failed to return a separator within the balance constraint in 163 instances, with Mongoose failing on 201 ($\sim$7%) and METIS failing on 791 ($\sim$28%). In the 1.5% imbalance case, neither library returned a solution within the tolerance on 621 instances, with Mongoose failing on 628 ($\sim$23%) and METIS failing on 921 ($\sim$33%).

As METIS generally fails more often to return a separator with the specified balance, these results reflect slightly more favorably on Mongoose. METIS finds a better (balanced) separator in less time in only 37% of cases at 20% imbalance, and 58% at 1.5% imbalance, while Mongoose now finds a better (balanced) separator in less time in 24% and 14% of cases at 20% and 1.5% imbalance, respectively.

The other flaw with the original comparison involves the size of the graphs formed from the SuiteSparse Matrix Collection. While the Collection is an excellent cross-section of applications and matrix types, nearly 20% of the graphs have fewer than 1,000 vertices, and almost all of them ($\sim$94%) have fewer than 1,000,000 vertices. For small graphs, runtimes may be measured in milliseconds. We wish to focus our comparison on partitioner performance when partitioning very large graphs, where runtimes can take significantly longer (more than one hour in some cases) and scalability becomes more important. Thus, we limit the Collection to graphs with at least 10,000,000 vertices, which represent more modern computational loads such as applications in

Table 3.4: Performance comparison between Mongoose and METIS on all 2,778 graphs from (or formed from) the SuiteSparse Collection with imbalance tolerance of 1.5%, treating imbalance violations as failures.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	1252	53	1305
	Tie	337	0	337
	Mongoose	206	309	515
		1795	362	2157

Big Data and exascale computing.

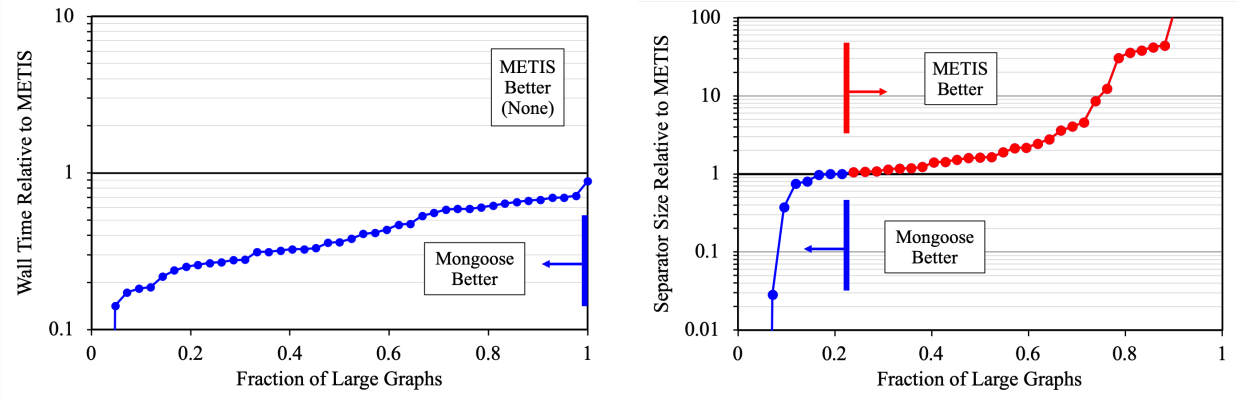
The Collection contains 42 such graphs with 10,000,000 vertices or more. If we repeat the previous analysis on this subset, again excluding failures, we find that Mongoose scales extremely well when compared to METIS (see Tables 3.5 and 3.6 and Figures 3.7, 3.8, 3.9, and 3.10). Mongoose yields faster execution in almost all cases with very few imbalance violations. However, despite this scaling, METIS generally still provides better cuts most of the time ($\sim$63% of cases for the 0.2 imbalance constraint, and $\sim$89% of cases for the 1.5% imbalance constraint).

The largest graph in the collection (Sybrandt/MOLIERE_2016) has more than 30 billion vertices and more than 6 trillion edges. Even on our computational server with 768GB of memory, METIS was unable to partition this graph due to memory constraints. However, Mongoose was able to partition this graph in approximately two hours with a separator of size 24,899,311 that satisfies the balance constraint of 1.5%, and a separator of size 21,688,654 that satisfies the 20% balance constraint.

If we examine the performance on the 15 largest graphs (by vertex count) in the Collection, we see that Mongoose has a significant edge in runtime, although METIS provides generally smaller separators (see Table 3.7). As has been the trend to this point, Mongoose more often satisfies the imbalance constraint, and scales better to larger graphs in run-time.

Table 3.5: Performance comparison between Mongoose and METIS on 42 large graphs (10,000,000+ vertices) from (or formed from) the SuiteSparse Collection with imbalance tolerance of 20%.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	1	25	26
	Tie	0	2	2
	Mongoose	0	13	13
		1	40	41

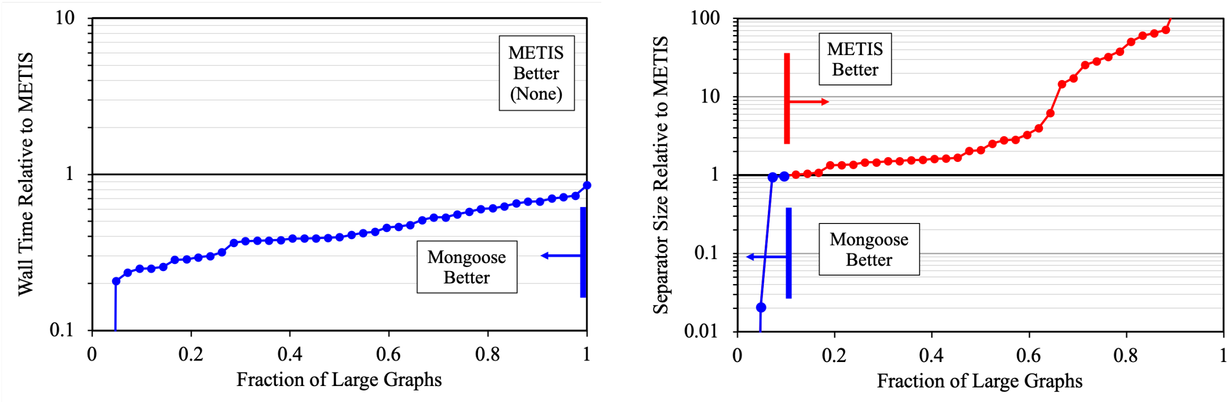


(a)

(b)

Figure 3.7: Wall time (a) and separator size (b) comparison between Mongoose and METIS with 20% imbalance tolerance on large graphs (10,000,000+ vertices) from the SuiteSparse Collection, including results with imbalance constraint violations.

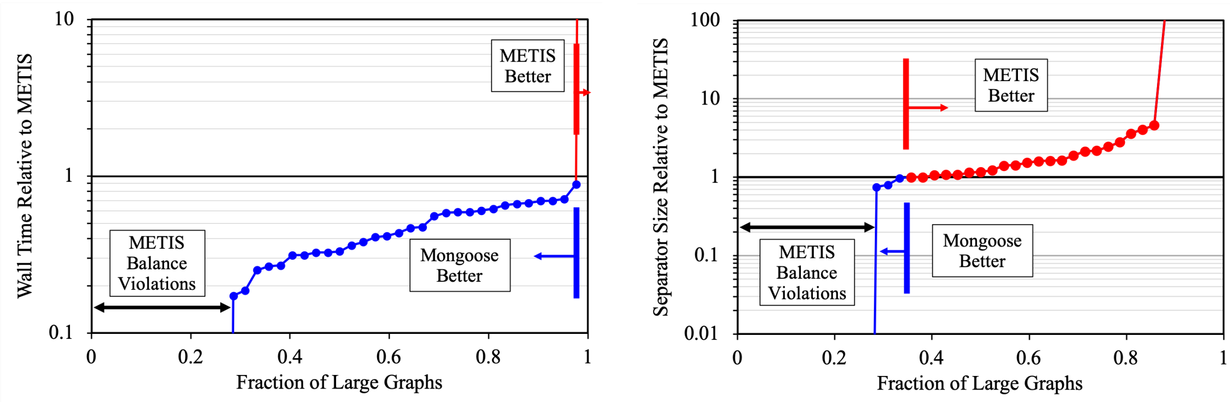
For all 42 large graphs, Mongoose completed execution faster, but provided a better separator on only 7 graphs (with an additional 2 ties). However, imbalance violations were not removed from this data. On 10 of these graphs, METIS returned a separator in violation of the imbalance constraint; Mongoose returned only 1 result that violated the same constraint. See Figure 3.9 for this same plot with violations removed.



(a) (b)

Figure 3.8: Wall time (a) and separator size (b) comparison between Mongoose and METIS with 1.5% imbalance tolerance on large graphs (10,000,000+ vertices) from the SuiteSparse Collection, including results with imbalance constraint violations.

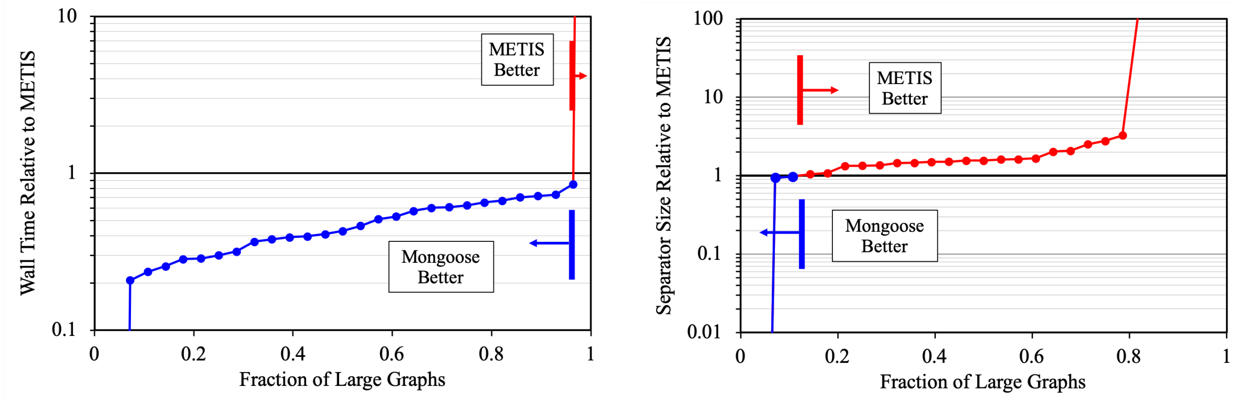
For all 42 large graphs, Mongoose completed execution faster, but provided a better separator on only 7 graphs (with an additional 2 ties). However, imbalance violations were not removed from this data. On 10 of these graphs, METIS returned a separator in violation of the imbalance constraint; Mongoose returned only 1 result that violated the same constraint. See Figure 3.10 for this same plot with violations removed.



(a) (b)

Figure 3.9: Wall time (a) and separator size (b) comparison between Mongoose and METIS with 20% imbalance tolerance on the large graphs (10,000,000+ vertices) from the SuiteSparse Collection with results that violate the imbalance constraint treated as failures.

For all except 1 of the 42 large graphs, Mongoose completed execution faster, with the exception being an imbalance constraint violation. METIS returned a separator in violation of the imbalance constraint for 10 graphs; these were treated as infinite time and infinite size cuts. Mongoose returned only 1 result that violated the same constraint and was treated the same as the ones returned in violation by METIS, with infinite time and an infinite size cut. See Figure 3.7 for this same plot with violations included and ignored.



(a)

(b)

Figure 3.10: Wall time (a) and separator size (b) comparison between Mongoose and METIS with 1.5% imbalance tolerance on the large graphs (10,000,000+ vertices) from the SuiteSparse Collection with results that violate the imbalance constraint treated as failures.

For all except 1 of the 42 large graphs, Mongoose completed execution faster, with the exception being an imbalance constraint violation. METIS returned a separator in violation of the imbalance constraint for 10 graphs; these were treated as infinite time and infinite size cuts. Mongoose returned only 1 result that violated the same constraint and was treated the same as the ones returned in violation by METIS, with infinite time and an infinite size cut. See Figure 3.8 for this same plot with violations included and ignored.

Table 3.6: Performance comparison between Mongoose and METIS on 42 large graphs (10,000,000+ vertices) from (or formed from) the SuiteSparse Collection with imbalance tolerance of 1.5%.

		Better Time		
		METIS	Mongoose	
Better Cut	METIS	1	24	25
	Tie	0	0	0
	Mongoose	0	3	3
		1	27	28

Table 3.7: Performance comparison between Mongoose and METIS on the 15 largest (by vertices) graphs in the SuiteSparse Collection. Note that the bipartite graph is formed for unsymmetric (directed) graphs. Imbalance violations are denoted with an asterisk, and results with clearly better performance (i.e. better run time and/or separator size with no imbalance violation) are denoted in bold.

Problem			Wall Time (s)			Separator Size ($\|S\|$)			Imbalance (ψ)	
Graph Name	Vertices	Edges	METIS	Mongoose	Speedup	METIS	Mongoose	Relative $\|S\|$ Size	METIS	Mongoose
LAW/webbase-2001	236,284,310	2,039,806,380	909.0	**129.3**	**7.03**	35385	0	0	0.19922	*0.20009
MAWI/mawi_201512020330	226,196,185	480,047,890	474.6	**82.1**	**5.78**	**1**	2115	2115	0.00029	0.18059
GenBank/kmer_V1r	214,005,017	465,410,904	918.5	219.8	4.18	1794891	**51279**	0.029	*0.49411	0.16310
GenBank/kmer_A2a	170,728,175	360,585,172	643.2	**209.9**	**3.06**	**1,038,180**	17,107,88	1.65	0.19428	0.16707
GenBank/kmer_P1a	139,353,211	297,829,984	505.6	**165.1**	**3.06**	**826,275**	1,563,902	1.89	0.19350	0.17789
MAWI/mawi_201512020130	128,568,730	270,234,840	221.9	**41.5**	**5.34**	**1**	825	825	0.00001	0.00000
LAW/sk-2005	101,272,308	3,898,825,202	907.9	**198.0**	**4.59**	25,681	1,077,219	41.9	0.19999	*0.20003
SNAP/twitter7	83,304,460	2,936,730,364	14,064.5	**3,909.5**	**3.60**	2,211,788	27,245,584	12.3	0.19980	*0.21912
LAW/it-2004	82,583,188	2,301,450,872	468.8	**121.8**	**3.85**	41,493	1,587,842	38.3	0.19999	*0.20022
LAW/uk-2005	78,919,850	1,872,728,564	378.5	**106.0**	**3.57**	74,939	2,299,813	30.7	0.19999	*0.20069
MAWI/mawi_201512020030	68,863,315	143,414,960	96.9	**25.8**	**3.75**	**1**	463	463	0.00001	0.19923
GenBank/kmer_U1a	67,716,231	138,778,562	143.7	**62.5**	**2.30**	**192,260**	270,836	1.41	0.19539	0.17142
SNAP/com-Friendster	65,608,366	3,612,134,270	14,737.2	**4,617.4**	**3.19**	**3,127,474**	11,273,576	3.60	0.00069	0.19957
GenBank/kmer_V2a	55,042,369	117,217,600	111.7	**52.2**	**2.14**	**119,572**	193,599	1.62	0.19654	0.16919
DIMACS10/europe_osm	50,912,018	108,109,320	56.7	**21.6**	**2.62**	**199**	433	2.18	0.05591	0.19247

3.9 Summary

In this chapter, we have described the use of computational optimization in a novel hybrid algorithm for computing vertex separators in arbitrary graphs. We have derived generalized gains that can be used to bridge the gap between combinatoric and continuous approaches to partitioning, and built an algorithm around these gains, and reported computational results for this new algorithm.

It should be noted that this algorithm, while demonstrating favorable scaling properties and being competitive with METIS on very large graphs, is only the starting point for future developments. With the generalized gains, additional sub-algorithms can be created that build off of Mongoose with the goal of yielding higher quality and better balanced partitions. In short, while Mongoose has been engineered to be high-quality scientific software (see Section 3.7), it should still be considered a prototype code with room for future optimizations and improvements.

In the next chapter, we conclude and describe future directions for the Mongoose library and hybrid algorithms for graph partitioning in general.

4. CONCLUSIONS AND FUTURE WORK

We have explored two types of graph partitioning, edge cuts and vertex separators, and applied a number of computational optimization approaches to develop novel heuristic partitioning algorithms.

Many possible future directions exist for further improving the algorithms proposed in this work, as well as for developing new optimization formulations for both the graph partitioning problems discussed here and for others, such as hypergraph partitioning.

4.1 Parallelization

While we have limited ourselves to serial computation for the proposed algorithms and for all comparisons to the state of the art, the natural progression of this work is the development of versions of these graph partitioning algorithms that exploit computational parallelism. There are many existing graph partitioning libraries that utilize parallelism, using either multiple CPU threads or external accelerators such as GPUs [26, 28, 31, 30].

Many of these parallel graph partitioning libraries focus on subdividing the the graph and processing these subgraphs in parallel. For example, one can use techniques from community detection and clustering, such as label propagation, to spatially subdivide the graph and process each part largely independently of the others. While these techniques are often effective, they are also highly dependent on the cluster size and the graph topology itself.

Hybrid techniques allow another avenue for exploiting parallelism: numerical optimization. Depending on how the optimization portion of the hybrid algorithm is solved, parallelism can be introduced in the following areas:

- Multi-threaded and GPU-accelerated sparse linear algebra in the solving of LPs, QPs, and LP subproblems of MILPs.

- Parallel sparse matrix-vector multiply in the updating of generalized gains in the hybrid vertex separator algorithm presented in Chapter 3.

- Parallel exploration of the branch-and-bound tree when solving MILPs.

Further exploration of the MILP formulation of the vertex separator problem, described in Section 3.4.1, may be especially worthwhile, given the potential for parallelism in that approach.

4.2 Further Algorithmic Optimizations

Currently, both graph partitioning algorithms (the edge cut algorithm from Chapter 2 and the vertex separator algorithm from Chapter 3) are generally competitive, but could be further optimized both in run-time performance and cut quality. For example, Intel MKL kernels [72] and the Sparse BLAS [73] could be used to exploit vectorization and sparse linear algebra operations. Also, a sensitivity analysis could be performed on the vertex separator algorithm's many user-specified options to determine optimal defaults.

The BLP algorithm proposed by Hager, Hungerford, and Safro [60] may also provide a starting point for further improvements to Mongoose. While BLP is not optimized for speed, it yields high-quality vertex separators, and utilizes much of the same hybrid approach of applying continuous optimization in tandem with traditional combinatoric approaches.

Another area for further development and optimization is the use of heaps in both algorithms described in Chapters 2 and 3. Currently, a simple binary heap is used, but other forms of heaps (or, more specifically, priority queues) may yield better performance. Asymptotically, Fibonacci heaps offer superior time complexity when compared to binary heaps, with $O(1)$ amortized insertion. However, they suffer from large constant factors that may prove too costly when operating on smaller graphs [34].

4.3 Extensions to k-way Partitioning

Another class of graph partitioning problems involve partitioning the graph into k subgraphs, usually with $k \geq 3$. While k-way partitioning can be approximated using the methods described in this work by applying the proposed algorithms recursively, oftentimes it is more effective to use an algorithm specifically designed for k-way partitioning. Still, the proposed algorithms could be used as a starting point for novel k-way partitioning algorithms and libraries.

4.4 Hypergraph Partitioning

A hypergraph is a graph whose edges (referred to as hyperedges or nets) may connect more than two vertices (see Section 1.2.1). Hypergraphs have wide applicability in areas such as VLSI design, as more than two electrical components can be connected to each other with a single wire or lead. They are also a generalization of traditional undirected graphs, and thus approaches to hypergraph partitioning can be used to partition traditional graphs.

Similar to edge cuts in graph (non-hypergraph) partitioning, the goal of hypergraph partitioning is to partition the vertices of a hypergraph such that the fewest hyperedges are cut. In this context, a cut hyperedge is one which contains at least one vertex in each part X and Y.

Despite optimization formulations existing for both edge cuts and vertex separators of traditional graphs, a canonical optimization formulation for hypergraph partitioning has not been put forth. Given hypergraph consisting of a set of n vertices V, a set of m hyperedges E, and an $n \times 1$ vertex weight vector w, we propose the following formulation and its derivation as a foundation for future development:

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{Ax} \circ \boldsymbol{A}(\mathbf{1} - \boldsymbol{x}) \\ \text{subject to} \quad & \\ & \ell \leq \boldsymbol{w}^{\mathsf{T}} \boldsymbol{x} \leq u \\ & \mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1} \\ & \boldsymbol{x} \in \mathbb{R}^n \end{aligned} \tag{HP-QP}$$

In this formulation, the matrix $\boldsymbol{A}$ is the hypergraph incidence matrix, the hypergraph equivalent of an adjacency matrix. Instead of A_{ij} representing the weight or existence of an edge between vertices i and j, $\boldsymbol{A}$ is an $m \times n$ binary matrix representing m hyperedges and n vertices. Each row in $\boldsymbol{A}$ represents a hyperedge e_j with the following relationship, given a row (hyperedge) $j \in 1, ..., m$ and column (vertex) $i \in 1, ..., n$:

$$A_{ij} = \begin{cases} 1 & \text{if vertex } v_i \in e_j \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

The $n \times 1$ vector $\boldsymbol{x}$ represents whether a vertex v_i is in one part or the other in the partitioning:

$$x_i = \begin{cases} 1 & \text{if vertex } v_i \text{ is contained in part } X \\ 0 & \text{if vertex } v_i \text{ is contained in part } Y \end{cases} \tag{4.2}$$

If all the vertices in a hyperedge e_j are located in one part, then either $x_i = 0$ or $x_i = 1 \ \forall \ v_i \in e_j$. It follows that $\boldsymbol{A}_{j,*}\boldsymbol{x} = 0$ or $\boldsymbol{A_{j,*}}(\boldsymbol{1} - \boldsymbol{x}) = 0$, so $[\boldsymbol{x}^{\mathsf{T}}\boldsymbol{A}_{j,*}] \cdot [(\boldsymbol{1} - \boldsymbol{A}_{j,*}\boldsymbol{x})] = 0$. Thus, for any edge that is not cut, it will contribute zero toward the objective function. However, if an edge is cut, then $[\boldsymbol{A}_{j,*}\boldsymbol{x}] \cdot [\boldsymbol{A}_{j,*}(\boldsymbol{1} - \boldsymbol{x})] \geq 1$.

The constraints for this optimization formulation are analogous to the constraints in the formulation for traditional edge cuts, with a balance constraint limiting the size of each part and limiting the elements of $\boldsymbol{x}$ to the closed, continuous range of $x_i \in [0, 1]$.

Unfortunately, this formulation is incomplete and requires further development. The objective function does not treat all possible cuts of a given hyperedge equally. For example, let us assume that $\boldsymbol{A} = [1\ 1\ 1\ 1]$, a single hyperedge in a hypergraph with four vertices. Consider the following two partitionings: $\boldsymbol{x}_1 = [0\ 0\ 1\ 1]^{\mathsf{T}}$ and $\boldsymbol{x}_2 = [0\ 0\ 0\ 1]^{\mathsf{T}}$.

$$\boldsymbol{A}\boldsymbol{x}_1 \circ \boldsymbol{A}(\boldsymbol{1} - \boldsymbol{x}_1) = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 2 \cdot 2 = 4 \tag{4.3}$$

$$\boldsymbol{A}\boldsymbol{x}_2 \circ \boldsymbol{A}(\boldsymbol{1} - \boldsymbol{x}_2) = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = 1 \cdot 3 = 3 \tag{4.4}$$

In both cases, a single hyperedge is cut, but the contribution to the objective function is different. Thus, this objective function not only penalizes cut hyperedges, it also penalizes more for cut hyperedges whose vertices are more symmetrically partitioned (2 and 2 versus 3 and 1). Nonetheless, this formulation roughly approximates the hyperedge partitioning problem, and can likely be developed further to more accurately model the problem.

REFERENCES

[1] T. A. Davis, W. W. Hager, S. P. Kolodziej, and S. N. Yeralan, “Algorithm XXX: Mongoose, a graph coarsening and partitioning library,” *ACM Trans. Math. Software*, 2019.

[2] C.-E. Bichot and P. Siarry, *Graph Partitioning*. Wiley Online Library, 2011.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*. The Johns Hopkins University Press, 4 ed., 2013.

[4] T. Bui and C. Jones, “Finding good approximate vertex and edge partitions is NP-hard,” *Information Processing Letters*, vol. 42, no. 3, pp. 153–159, 1992.

[5] M. U. Nisar, A. Fard, and J. A. Miller, “Techniques for graph analytics on big data,” in *2013 IEEE International Congress on Big Data*, pp. 255–262, IEEE, 2013.

[6] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, “Trends in big data analytics,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561–2573, 2014.

[7] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, *et al.*, “Mathematical foundations of the GraphBLAS,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, IEEE, 2016.

[8] T. Davis, “Algorithm 9xx: SuiteSparse: GraphBLAS: graph algorithms in the language of sparse linear algebra,” *ACM Trans. Math. Software*, 2019.

[9] B. Vastenhouw and R. H. Bisseling, “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication,” *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[10] A. George, “Nested dissection of a regular finite element mesh,” *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

[11] T. A. Davis and Y. Hu, “The University of Florida Sparse Matrix Collection,” *ACM Trans. Math. Software*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[12] 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.

[13] H. D. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, vol. 2, no. 2-3, pp. 135–148, 1991.

[14] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[15] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[16] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Conference on Design Automation, 1982.*, pp. 175–181, June 1982.

[17] C. Ashcraft and J. W. Liu, "A partition improvement algorithm for generalized nested dissection," *Boeing Computer Services, Seattle, WA, Tech. Rep. BCSTECH-94-020*, 1994.

[18] B. Hendrickson and E. Rothberg, "Improving the run time and quality of nested dissection ordering," *SIAM J. Sci. Comput.*, vol. 20, pp. 468–489, Dec. 1998.

[19] S. Barnard and H. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101–117, 1994.

[20] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," *SuperComputing Conference*, p. 28, 1995.

[21] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, pp. 430–452, 1990.

[22] W. W. Hager and Y. Krylyuk, "Graph partitioning and continuous quadratic programming," *SIAM Journal on Discrete Mathematics*, vol. 12, no. 4, pp. 500–523, 1999.

[23] J. Shi and J. Malik, “Normalized cuts and image segmentation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 888–905, Aug 2000.

[24] Bui, Chaudhuri, Leighton, and Sipser, “Graph bisection algorithms with good average case behavior,” *Combinatorica*, vol. 7, no. 2, pp. 171–191, 1987.

[25] G. Karypis and V. Kumar, “Parallel multilevel graph partitioning,” in *Proceedings of International Conference on Parallel Processing*, pp. 314–319, IEEE, 1996.

[26] G. Karypis and V. Kumar, “A parallel algorithm for multilevel graph partitioning and sparse matrix ordering,” *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[27] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: applications in VLSI domain,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[28] D. LaSalle and G. Karypis, “Multi-threaded graph partitioning,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 225–236, IEEE, 2013.

[29] F. Pellegrini and J. Roman, “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs,” in *International Conference on High-Performance Computing and Networking*, pp. 493–498, Springer, 1996.

[30] C. Chevalier and F. Pellegrini, “PT-Scotch: A tool for efficient parallel graph ordering,” *Parallel Computing*, vol. 34, no. 6-8, pp. 318–331, 2008. Parallel Matrix Algorithms and Applications.

[31] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine, “The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring,” *Scientific Programming*, vol. 20, no. 2, pp. 129–150, 2012.

[32] Ü. V. Çatalyürek and C. Aykanat, “PaToH: A multilevel hypergraph partitioning tool,” *Tech. Rep. BU-CE-9915*, 1999.

[33] Ü. V. Çatalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 673–693, July 1999.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[35] D. A. Spielman and S.-H. Teng, “Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time,” *J. ACM*, vol. 51, pp. 385–463, May 2004.

[36] J. Nocedal and S. Wright, *Numerical Optimization*. Springer Science & Business Media, 2006.

[37] M. J. Quinn and N. Deo, “An upper bound for the speedup of parallel best-bound branch-and-bound algorithms,” *BIT Numerical Mathematics*, vol. 26, no. 1, pp. 35–43, 1986.

[38] B. Gendron and T. Crainic, “Parallel branch-and-bound algorithms: Survey and synthesis,” *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 2011.

[39] A. Pothen, “Graph partitioning algorithms with applications to scientific computing,” in *Parallel Numerical Algorithms*, pp. 323–368, Springer, 1997.

[40] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[41] G. Karypis and V. Kumar, “Multilevel graph partitioning schemes,” in *Proc. 1995 Intl. Conf. Parallel Processing*, pp. 113–122, CRC Press, 1995.

[42] A. Gupta, “Fast and effective algorithms for graph partitioning and sparse-matrix ordering,” *IBM Journal of Research and Development*, vol. 41, pp. 171–183, Jan 1997.

[43] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis, “Improving graph partitioning for modern graphs and architectures,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, p. 14, ACM, 2015.

[44] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, “Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning,” *Operations Research*, vol. 37, no. 6, pp. 865–892, 1989.

[45] E. L. Johnson, A. Mehrotra, and G. L. Nemhauser, “Min-cut clustering,” *Mathematical Programming*, vol. 62, no. 1-3, pp. 133–151, 1993.

[46] T. A. Davis, W. W. Hager, and J. T. Hungerford, “An efficient hybrid algorithm for the separable convex quadratic knapsack problem,” *ACM Trans. Math. Software*, vol. 42, pp. 22:1–22:25, May 2016.

[47] E. D. Dolan and J. J. Moré, “Benchmarking optimization software with performance profiles,” *Math. Program.*, vol. 91, pp. 201–213, 2002.

[48] P. Boldi and S. Vigna, “The WebGraph framework I: Compression techniques,” in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pp. 595–601, ACM Press, 2004.

[49] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th International Conference on World Wide Web*, ACM Press, 2011.

[50] C. de Souza and E. Balas, “The vertex separator problem: algorithms and computations,” *Mathematical Programming*, vol. 103, no. 3, pp. 609–631, 2005.

[51] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations*, pp. 85–103, Springer, 1972.

[52] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[53] C. Blum, “Ant colony optimization: Introduction and recent trends,” *Physics of Life reviews*, vol. 2, no. 4, pp. 353–373, 2005.

[54] P. Kuntz, P. Layzell, and D. Snyers, “A colony of ant-like agents for partitioning in vlsi technology,” in *Proceedings of the Fourth European Conference on Artificial Life*, pp. 417–424, MIT Press, Cambridge, MA, 1997.

[55] P. Korošec, J. Šilc, and B. Robič, “Solving the mesh-partitioning problem with an ant-colony algorithm,” *Parallel Computing*, vol. 30, no. 5-6, pp. 785–801, 2004.

[56] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[57] E. Balas, “An additive algorithm for solving linear programs with zero-one variables,” *Operations Research*, vol. 13, no. 4, pp. 517–546, 1965.

[58] E. L. Lawler and D. E. Wood, “Branch-and-bound methods: A survey,” *Operations research*, vol. 14, no. 4, pp. 699–719, 1966.

[59] W. Hager and J. Hungerford, “Continuous quadratic programming formulations of optimization problems on graphs,” *European Journal of Operational Research*, vol. 240, no. 2, pp. 328–337, 2015.

[60] W. W. Hager, J. T. Hungerford, and I. Safro, “A multilevel bilinear programming algorithm for the vertex separator problem,” *Computational Optimization and Applications*, vol. 69, no. 1, pp. 189–223, 2018.

[61] J. E. Falk and R. M. Soland, “An algorithm for separable nonconvex programming problems,” *Management Science*, vol. 15, no. 9, pp. 550–569, 1969.

[62] H. S. Ryoo and N. V. Sahinidis, “A branch-and-reduce approach to global optimization,” *Journal of Global Optimization*, vol. 8, no. 2, pp. 107–138, 1996.

[63] G. Gallo and A. Ülkücü, “Bilinear programming: an exact algorithm,” *Mathematical Programming*, vol. 12, no. 1, pp. 173–194, 1977.

[64] L. N. Vicente and P. H. Calamai, “Bilevel and multilevel programming: A bibliography review,” *Journal of Global Optimization*, vol. 5, no. 3, pp. 291–306, 1994.

[65] S. Dempe, "Annotated bibliography on bilevel programming and mathematical programs with equilibrium constraints," *Optimization*, vol. 52, no. 3, pp. 333–359, 2003.

[66] H. Konno, "A cutting plane algorithm for solving bilinear programs," *Mathematical Programming*, vol. 11, no. 1, pp. 14–27, 1976.

[67] G. B. Dantzig, "Discrete-variable extremum problems," *Operations Research*, vol. 5, no. 2, pp. 266–288, 1957.

[68] P. Nash, "Catch2." https://github.com/catchorg/Catch2.

[69] Travis CI, GmbH, "Travis CI." https://travis-ci.com.

[70] D. Marjamäki, "Cppcheck: a tool for static C/C++ code analysis," 2013.

[71] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2005.

[72] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi^{TM}*, pp. 167–188, Springer, 2014.

[73] K. Remington and R. Pozo, "NIST Sparse BLAS: User's Guide," tech. rep., Citeseer, 1996.