Algorithm 1003: Mongoose, a Graph Coarsening and Partitioning Library

TIMOTHY A. DAVIS, Texas A&M University
WILLIAM W. HAGER, University of Florida
SCOTT P. KOLODZIEJ, Texas A&M University
S. NURI YERALAN, University of Florida

Partitioning graphs is a common and useful operation in many areas, from parallel computing to VLSI design to sparse matrix algorithms. In this article, we introduce Mongoose, a multilevel hybrid graph partitioning algorithm and library. Building on previous work in multilevel partitioning frameworks and combinatoric approaches, we introduce novel stall-reducing and stall-free coarsening strategies, as well as an efficient hybrid algorithm leveraging (1) traditional combinatoric methods and (2) continuous quadratic programming formulations. We demonstrate how this new hybrid algorithm outperforms either strategy in isolation, and we also compare Mongoose to METIS and demonstrate its effectiveness on large and social networking (power law) graphs.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; *Quadratic programming*; *Nonconvex optimization*; *Mathematical software performance*;

Additional Key Words and Phrases: Graph partitioning, vertex matching, graph coarsening

ACM Reference format:
Timothy A. Davis, William W. Hager, Scott P. Kolodziej, and S. Nuri Yeralan. 2020. Algorithm 1003: Mongoose, a Graph Coarsening and Partitioning Library. *ACM Trans. Math. Softw.* 46, 1, Article 7 (March 2020), 18 pages.
https://doi.org/10.1145/3337792

1 INTRODUCTION

In this article, we present a multilevel graph partitioning library and algorithm incorporating novel coarsening and optimization approaches. 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 sparse matrices and compare our results to METIS, another graph partitioning library [Karypis and Kumar 1998a].

A brief discussion of multilevel graph partitioning appears in Section 2. Related and prior work in graph partitioning is discussed in Sections 3. The main components of the proposed algorithm

This work is supported by the Office of Naval Research under grants N00014-11-1-0068, N00014-15-1-2048, and N00014-18-1-2100, and by the National Science Foundation under grants CMMI-0620286, DMS-1522629, DMS-1819002, and CNS-1514406.

Authors' addresses: T. A. Davis and S. P. Kolodziej, Department of Computer Science and Engineering, 3112 TAMU, Texas A&M University, College Station, TX 77843-3112; emails: {davis, scottk}@tamu.edu; W. W. Hager, Department of Mathematics, University of Florida, Gainesville, FL 32611-8105; email: hager@ufl.edu; S. N. Yeralan (Current Address), 1 Microsoft Way, Redmond, WA 98052; email: nuri@microsoft.com.

0098-3500/2020/03-ART7 $15.00
https://doi.org/10.1145/3337792

and their relationship with one another are given in Sections 4, 5, and 6; computational results and comparisons are provided in Section 7. We conclude with a summary of this work and highlight future research directions in Section 8.

1.1 Definition

The Binary Graph Partitioning Problem is an NP-complete problem defined as taking an undirected input graph, $G(V,E)$, and removing edges until the graph breaks into two disjoint subgraphs. The set of edges deleted in this manner is known as the "cut set." When partitioning a graph, we seek to minimize the number of edges in the cut set while maintaining a target partitioning balance in the ratio of vertices in each component.

When the input graph has weighted vertices and edges, we generalize the problem definition by seeking to minimize the sum of edge weights for edges in the cut set rather than simply the number of edges. Further, the partition balance ratio is determined by considering the sum of vertex weights in each partition rather than the number of vertices in each partition. If weights are absent from vertices or edges, we assume a weight of 1.

In the k-way Graph Partitioning Problem, edges are deleted until there are k disjoint subgraphs. When k is a power of two, the k-way Graph Partitioning Problem can be solved recursively by solving the graph partitioning problem on each of the resulting disjoint components, although this method is not always the best heuristic.

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 [Pothen 1997].

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

Multilevel graph partitioners seek to simplify the input graph in an effort to recursively apply expensive (and more effective) 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 combinatorial analysis 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.

2.1 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 [Hendrickson and Leland 1995]. Vertices are merged together using strategies that exploit geometric and topological features of the problem. In Figure 1, the initial graph in the upper left is successively coarsened by matching pairs of vertices to obtain the smaller graphs appearing beneath it.

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."

2.2 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, as indicated by the dashed line in the bottom graph of Figure 1. 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 [Karypis and Kumar 1998b].

2.3 Graph Refinement

Once a satisfactory guess partition is achieved at the coarsest level, projecting the partition back to the original input graph requires the inverse operation of graph coarsening, known as graph refinement. In graph refinement, vertices expand back into their original representations at the finer level, as depicted by the two graph on the right side of Figure 1. The partition choice for each coarse vertex is applied to all of the vertices that participated in the matching used during graph coarsening.

Because a partition at a coarse level is not generally guaranteed to be optimal when projected to the refined level, traditional graph partitioning strategies (e.g., methods described in Section 3.1) are used to improve the projected partition as the graph is refined back to its original size.

3 RELATED WORK

3.1 Combinatorial Methods

Kernighan and Lin at Bell Labs developed the first graph partitioning package for use at Bell Systems [Kernighan and Lin 1970]. 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 [Fiduccia and Mattheyses 1982]. 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 [Karypis and Kumar 1998a], a strategy we have also adopted in Mongoose.

3.2 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 [Hendrickson and Leland 1995].

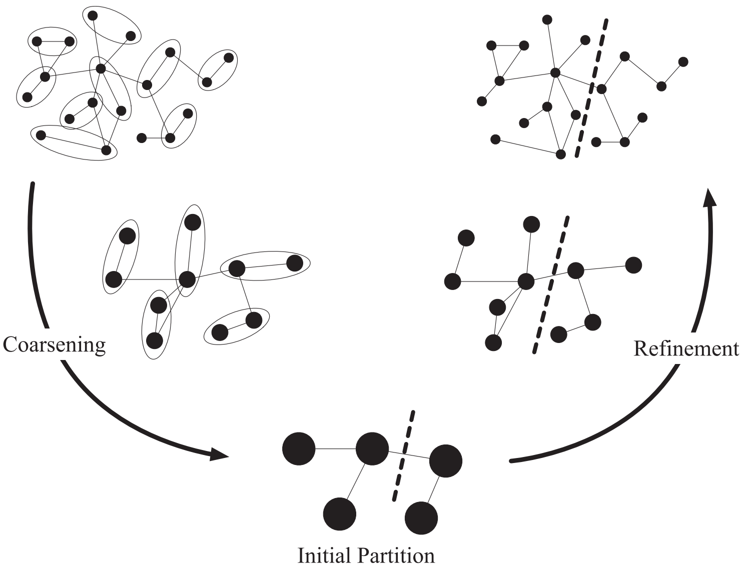


Fig. 1. Multilevel graph partitioning.

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) [Karypis and Kumar 1998a], as well as “Light Edge Matching” (LEM) and “Heavy Clique Matching” (HCM) [Karypis and Kumar 1995]. Gupta considered “Heavy Triangle Matching” (HTM) [Gupta 1997]. 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 [Davis and Hu 2011] and 2-hop matching [LaSalle et al. 2015].

Our extensions to these coarsening and matching methods are explained in detail in Section 4.

3.3 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 [Johnson et al. 1989] and mixed-integer programming [Johnson et al. 1993], 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 [1999], who showed that the discrete graph partitioning problem is equivalent to the continuous quadratic programming problem:

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

where ℓ and u are lower and upper bounds on the desired size of one partition, and $\mathbf{A}$ 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 5 and 6.

3.4 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 [Karypis and Kumar 1998a], an early multilevel framework partitioner that has since been refined and expanded to include parallel [Karypis et al. 1997], hypergraph [Karypis et al. 1999], and multi-threaded [LaSalle and Karypis 2013] versions. We use METIS as our primary comparison in Section 7 and build on their work on coarsening and refinement.

Other graph partitioning libraries include the following:

- **Chaco**, a multilevel partitioner that uses combinatorial methods (e.g., Kernighan-Lin and Fiduccia-Mattheyses), as well as spectral methods [Hendrickson and Leland 1993].
- **KaHIP**, short for Karlsruhe High Quality Partitioning, which employs flow-based and evolutionary partitioning methods [Sanders and Schulz 2013].
- **PARTY**, which uses a combination of local and global partitioning methods, as well as meta-heuristics using interfaces to other partitioners [Preis and Diekmann 1997].
- **SCOTCH**, a partitioning library with a variety of functions and methods, including a multilevel framework, combinatorial and greedy refinement methods, and a diffusion optimization. SCOTCH is also capable of static mapping, clustering, and hypergraph partitioning, and is available in both sequential and parallel versions [Pellegrini and Roman 1996].
- **Zoltan**, a diverse toolkit of parallel graph partitioning and other algorithms, is also available as a serial build [Boman et al. 2012].

4 COARSENING AND MATCHING STRATEGIES

To coarsen a graph as described in Section 2.1, 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 [Karypis and Kumar 1998a]. 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.

4.1 Brotherly Matching

In brotherly matching, two vertices that are not neighbors can be matched if they share a neighbor (see Figure 2). 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 [Karypis and Kumar 1998a]. The next two methods, adoption and community matching, are new.

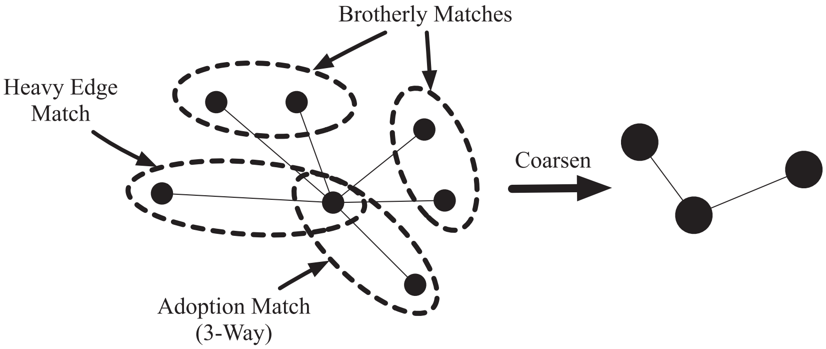


Fig. 2. Brotherly and adoption matching.

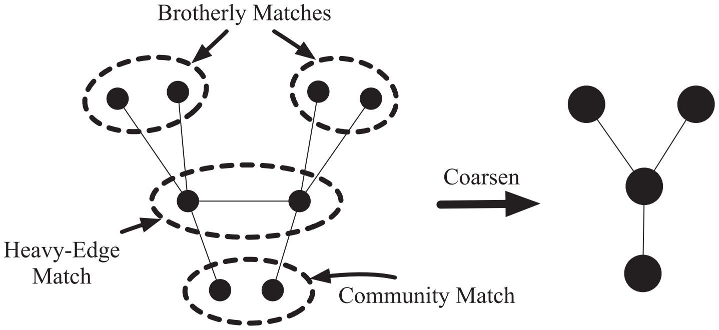


Fig. 3. Community matching.

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).

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 3).

5 QUADRATIC PROGRAMMING REFINEMENT

As mentioned earlier, Hager and Krylyuk [1999] introduced a continuous quadratic programming formulation of the binary graph partitioning problem:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \quad (\mathbf{1}-\mathbf{x})^{\mathsf{T}}(\mathbf{A}+\mathbf{I})\mathbf{x} \quad \text{subject to} \quad \mathbf{0} \le \mathbf{x} \le \mathbf{1}, \quad \ell \le \mathbf{1}^{\mathsf{T}}\mathbf{x} \le u, \tag{2}$$

where ℓ and u are lower and upper bounds on the desired size of one partition, and $\mathbf{A}$ 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 (1). We then perform iterations of gradient projection until reaching a stationary point of Equation (1); often convergence takes just a few iterations. Although the stationary point may not be binary, the analysis in Hager and Krylyuk [1999] 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 Equation (1). Since the constraints of Equation (1) consist of a single linear constraint coupled with bound constraints, computing this projection amounts to solving a quadratic knapsack problem. An extremely efficient algorithm for computing this projection is given in Davis et al. [2016]. Because the quadratic programming formulation ignores the combinatorial notion of boundary, it is capable of identifying vertices to swap that do not lie on the boundary of the cut. Gradient projection also adheres to strict balancing, and its local minimizers result in cuts with better balance than our Fiduccia-Mattheyses implementation.

6 ALGORITHM DESCRIPTION

In this section, we describe in detail our implementation of our Hybrid Combinatorial-Quadratic Programming Approach to graph partitioning. We use a multilevel approach that blends combinatorial methods with continuous graph partitioning strategies. Our algorithm for the graph partitioning problem is as follows:

(1) **Preprocessing.** The algorithm verifies that the input graph is undirected, free from self-edges, and that vertex and edge weights (if provided) are positive. The algorithm then traverses the graph to compute the sum of edge weights, sum of vertex weights, and average degree of the vertices.

(2) **Coarsening.** Mongoose's coarsening phase uses a novel matching algorithm to prevent the coarsening operation from stalling. The details of this operation follow:

   (a) **Heavy Edge Matching**. To quickly compute an initial matching, a standard iteration of heavy edge matching is computed, pairing vertices with their neighbor with whom they share an edge with largest weight (i.e., the heaviest edge). In rare cases, this matches all vertices in the graph, but in most cases, at least some vertices are left over.

   (b) **Stall-Free Matching**. After an initial heavy edge matching iteration, the algorithm considers vertices that remain unmatched. When the algorithm finds an unmatched vertex, $v_{unmatched}$, it does the following:

      i. **Find a suitable pivot neighbor**. The unmatched vertex scans its adjacency list to find the neighboring matched vertex with maximum edge weight. We call this neighbor v_{pivot}.

      ii. **Resolve unmatched neighbors of v_{pivot} pairwise**. Since v_{pivot} has at least one unmatched neighbor, namely $v_{unmatched}$, the algorithm shifts its focus to resolve all the unmatched neighbors of v_{pivot} with the hopes that $v_{unmatched}$ is not its only unmatched neighbor. The algorithm matches the unmatched neighbors of v_{pivot} pairwise. Although the vertices matched in this manner do not share an edge, they are topologically close in the graph. This is a brotherly matching.

      iii. **Adopt any remaining unmatched neighbor**. If there was an odd number of unmatched neighbors of v_{pivot}, then the pairwise matching strategy leaves one neighbor unmatched. Instead, v_{pivot} includes this unmatched neighbor to its matching in a 3-way match.

iv. **Community matching to prevent 4-way matches**. The first time v_{pivot}'s matching adopts a leftover unmatched vertex, it creates a 3-way adoption matching. However, v_{pivot} is not the only participant in its matching. Its match partner may have already adopted a vertex from an earlier stall-free matching. In this case, by performing the adoption, v_{pivot} would create a 4-way matching. Instead, v_{pivot} creates a new **community** match consisting of its would-be adoptee and the vertex its match previously adopted. This strategy prevents 4-way matches from occurring while guaranteeing coarsening progress.

We call this matching strategy *Stall-free Matching*, because it guarantees that in a single pass over the unmatched vertices after an initial matching that every vertex in the graph participates in a topologically relevant matching of at most 3 vertices. Stall-free matching also guarantees that the coarse graph has at most half the number of vertices as its predecessor.

Note that computing a stall-free matching is computationally more expensive than heavy edge matching alone. As such, the algorithm includes an option to limit these additional matching strategies to vertices with degree at least some multiple of the average degree of the graph. As high-degree vertices are generally responsible for stalls during coarsening, this focuses these additional strategies on only areas of the graph that are likely to benefit from them.

This strategy is guaranteed to be stall-free only if no degree threshold is used and all three matching strategies (brotherly, adoption, and community) are used. The options in the code allow for one or all of these strategies to be disabled. If just brotherly and adoption matchings are used, for example, we obtain a *stall-reducing* method, which typically avoids a stall but is not guaranteed to do so.

(c) **Singleton matching**. During coarsening, if a graph has multiple connected components, these components may be contracted into a singleton vertex. At this point, singleton vertices are matched with each other preferentially with at most one singleton remaining unmatched per iteration. This is a simple and efficient method for handling multiple connected components without computationally expensive methods such as bin-packing.

(3) **Initial Partitioning.** Mongoose contains several methods for computing an initial partitioning after coarsening has completed. 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 Fiduccia-Mattheyses but considers only those vertices in the boundary set. As swaps are made, vertices enter and leave the boundary set when swaps place them near or far from the boundary, respectively.

These partition choices for vertices are then used as an initializer 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.

(4) **Refinement.** Partition refinement for our algorithm consists of two parts working in tandem:

(a) **Boundary Fiduccia-Mattheyses**. Our algorithm uses a variation of the Fiduccia-Mattheyses algorithm that exploits the boundary optimization first used with the Boundary Kernighan-Lin strategy. This implementation of Fiduccia-Mattheyses maintains one heap per partition for boundary vertices. Vertices contained in these heaps represent swap candidates. Vertices in the heap are keyed by a sequential vertex identifier and are backed by the Fiduccia-Mattheyses gain value. The top three entries of the heap are considered, and their heuristic values are computed. The vertex with the maximum heuristic gain swaps partitions.

We also introduce a balance-aware heuristic that considers the effect of performing a swap with respect to the target balance constraints. This heuristic value is derived from the following formulation:

We define the gain of a vertex V_a as the sum of edge weights in the adjacency of V_a that lie in the cut set subtracted by the sum of the edge weights in the adjacency of V_a that does not lie in the cut set. These quantities are effectively the sum of edge weights from V_a to vertices in its adjacency that lie in the other partition minus those in the same partition.

Without loss of generality, suppose V_a currently lies in P_0 and a flip operation would transfer V_a to P_1. The definitions follow from this construction:

$$gain_a = \sum_{i \in P_1} E_{a,i} - \sum_{j \in P_0} E_{a,j}.$$

We then define a scalar imbalance resulting in a swap of vertex V_a as the ratio of sum of vertex weights in the left partition after the move divided by the total sum of vertex weights W subtracted by the user's desired target balance ratio:

$$Imbalance_a = \sum_{i \in P_{left} \bigcup V_a} V_i / W.$$

We finally define a heuristic value that compares the derived values considering the vertex gains as well as the impact to partition balance ratio should the vertex be swapped. In a way, this heuristic value considers both edge weights and vertex weights with one value:

$$Heuristic_a = Gain_a + \begin{cases} 2W * Imbalance_a & : Imbalance_a \geq \tau, \\ 0 & : otherwise. \end{cases}$$

If the current cut is balanced, the penalty term contributes nothing to the heuristic value. If the current cut is imbalanced, then we impose a balance penalty of the measure of imbalance times twice the sum of vertex weights.

The algorithm explores suboptimal moves after all obvious moves have been made. Our implementation of Fiduccia-Mattheyses may make several net-zero moves, which shift the partition boundary in an attempt to locate vertices with positive heuristic gains. Because the algorithm is balance-aware, such exploratory moves do not significantly disrupt the target balance partition.

By combining the boundary optimization with a balance-aware heuristic, the algorithm is able to consider moves that imbalance the problem. However, it only commits to such moves if it discovers that doing so results in an extraordinary reduction in the weight of the cut set.

(b) **Gradient Projection**. Following an application of our Balance-aware Boundary Fiduccia-Mattheyses, discussed above, we use the discrete partition choices for each vertex as starting guess for a solution of the quadratic programming problem (1). We then perform iterations of the gradient projection algorithm until reaching a stationary point of Equation (1); often convergence takes just a few iterations. Since the quadratic program is an exact reformulation of the graph partitioning problem, each iteration strictly improves the partition. Although the limit point may not be binary, the analysis in Hager and Krylyuk [1999] 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 Equation (1). Since the constraints of Equation (1) consist of a single linear constraint coupled with bound constraints, computing this projection amounts to solving a quadratic knapsack problem. An extremely efficient algorithm for computing this projection is given in Davis et al. [2016]. Because the quadratic programming formulation ignores the combinatorial notion of boundary, it is capable of identifying vertices to swap that do not lie on the boundary. Gradient projection also adheres to strict balancing, and its local minimizers result in cuts with better balance than our Fiduccia-Mattheysesxbrk implementation.

Note that this algorithm is applicable for partitioning any simple undirected graph with positive vertex and edge weights.

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 ±0.1%. All results shown satisfy this balance tolerance.

7.1 Overall Performance

Mongoose and METIS were run on the entire SuiteSparse Matrix Collection [Davis and Hu 2011] with only modest filtering. First, complex matrices were removed. Of the remaining matrices, any unsymmetric matrices $\mathbf{A}$ were treated as the biadjacency matrix of a bipartite graph adjacency matrix $\mathbf{B} = \begin{bmatrix} \mathbf{0}_{m,m} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0}_{n,n} \end{bmatrix}$; 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). This yielded 2,685 symmetric matrices that were then treated as undirected graphs to be partitioned.

The relative timing (a) and relative cut quality (b) performance are shown in Figure 4, and a tabular comparison is shown in Table 1. Of the 2,685 graphs, Mongoose found a smaller cut on 1,113 (~41%), and took less time to compute its cut on 1,271 (~47%). Mongoose outperformed METIS in both time and cut quality on 571 graphs (~21% of cases), while METIS outperformed Mongoose in both time and cut quality on 759 graphs (~28%). Thus, Mongoose is generally competitive with METIS, with METIS having a slight edge.

Figures 4, 5, and 8 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.

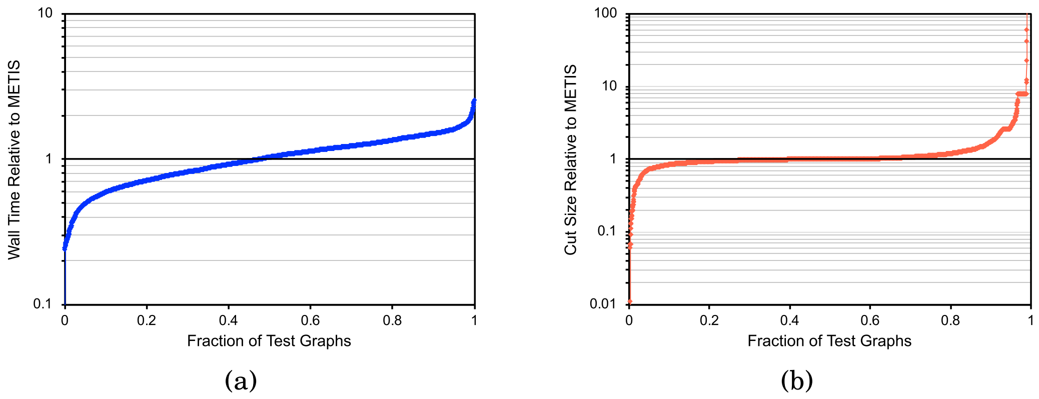


Fig. 4. (a) Overall timing and (b) overall cut quality performance of Mongoose relative to METIS 5. Note the logarithmic vertical scale. Points below the center line represent cases where Mongoose outperforms METIS (relative performance less than one), while points above the center line indicate cases where METIS outperforms Mongoose (relative performance greater than one). In general, Mongoose performs competitively with METIS 5.

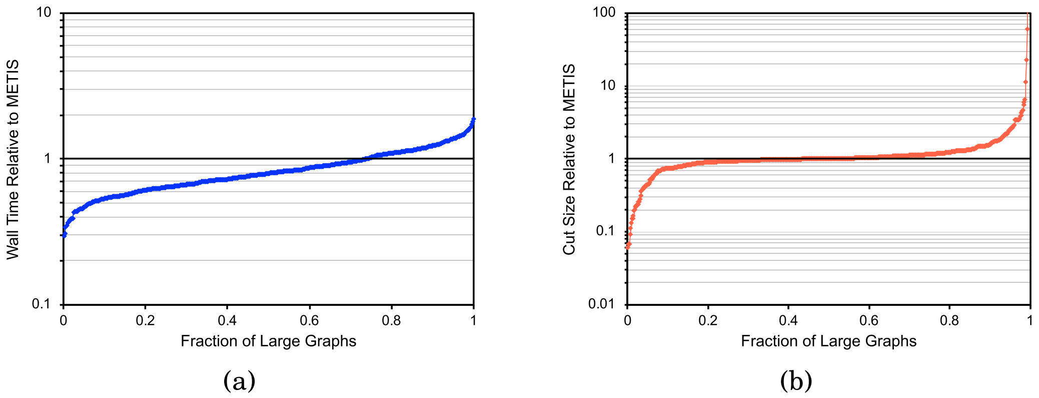


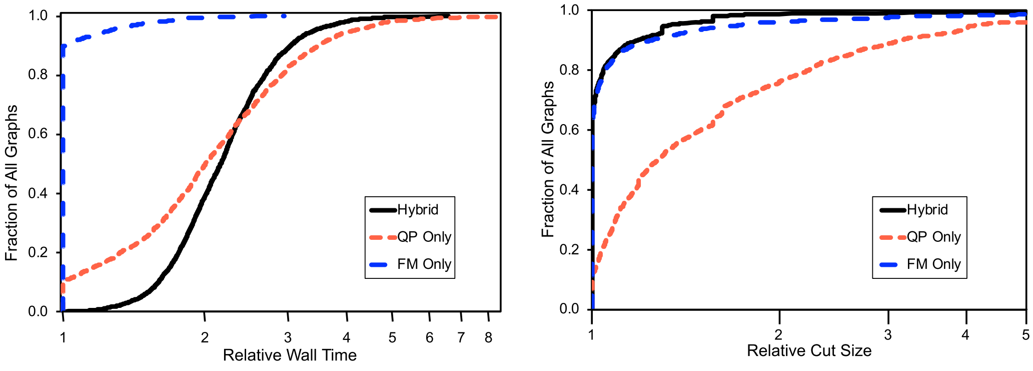
Fig. 5. (a) Timing and (b) cut quality performance profiles [Dolan and Moré 2002] of Mongoose on large graphs (1M+ edges) relative to METIS 5. Note the logarithmic vertical scale. Points below the center line represent cases where Mongoose outperforms METIS (relative performance less than one), while points above the center line indicate cases where METIS outperforms Mongoose (relative performance greater than one).

Table 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	1,286
	Tie	113	173	286
	Mongoose	542	571	1,113
		1,414	1,271	2,685

Table 2. Performance Comparison between Mongoose and METIS on the 601 Largest Graphs (1M+ 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



(a) (b)

Fig. 6. (a) Relative timing and (b) Relative cut size performance profiles [Dolan and Moré 2002] 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.

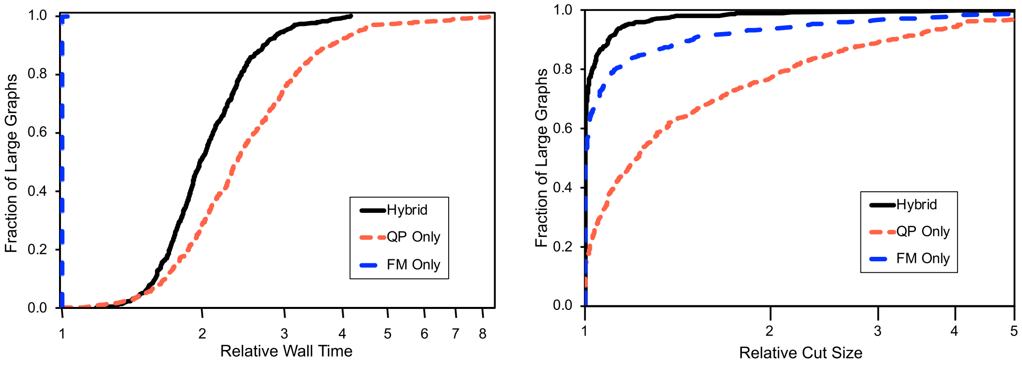
7.2 Performance on Large Graphs

When limited to graphs with at least 1M 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 5, and a tabular comparison is shown in Table 2.

7.3 Hybrid Performance

Figures 6 and 7 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 6, 7, and 9 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



(a) (b)

Fig. 7. (a) Relative timing and (b) relative cut size performance profiles [Dolan and Moré 2002] on the largest 601 graphs in the SuiteSparse Matrix Collection (1M+ 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.

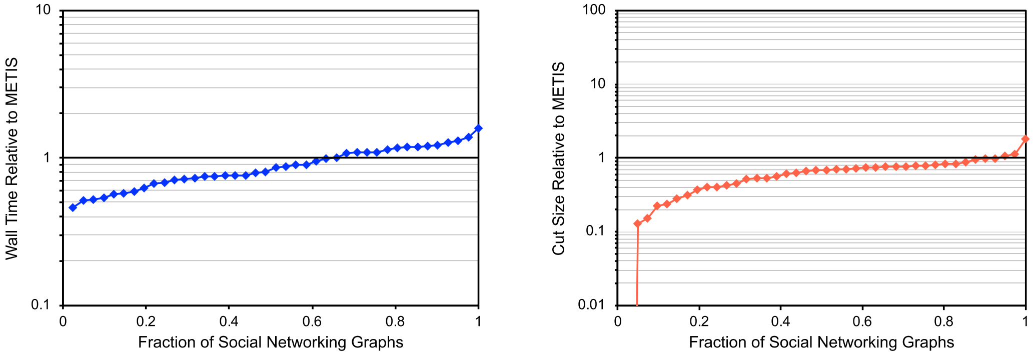


Fig. 8. Performance of Mongoose on social networking graphs relative to METIS 5. Note the logarithmic vertical scale. Points below the center line represent cases where Mongoose outperforms METIS (relative performance less than one), while points above the center line indicate cases where METIS outperforms Mongoose (relative performance greater than one).

equal to 1). The graphs are ordered from best to worst along the vertical axis and 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 [Dolan and Moré 2002].

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 [Boldi and Vigna 2004] [Boldi et al. 2011].

Figure 8 and Table 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:

Table 3. Performance Comparison between Mongoose and METIS on 41 Social Networking (ower law) Graphs in the SuiteSparse Collection

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

There were no ties in cut quality.

Table 4. Performance Comparison between Mongoose and METIS on the 15 Largest (by edges) Social Networking Graphs in the SuiteSparse Collection

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

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).

- **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 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 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.

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 1,024, 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 9. 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.

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

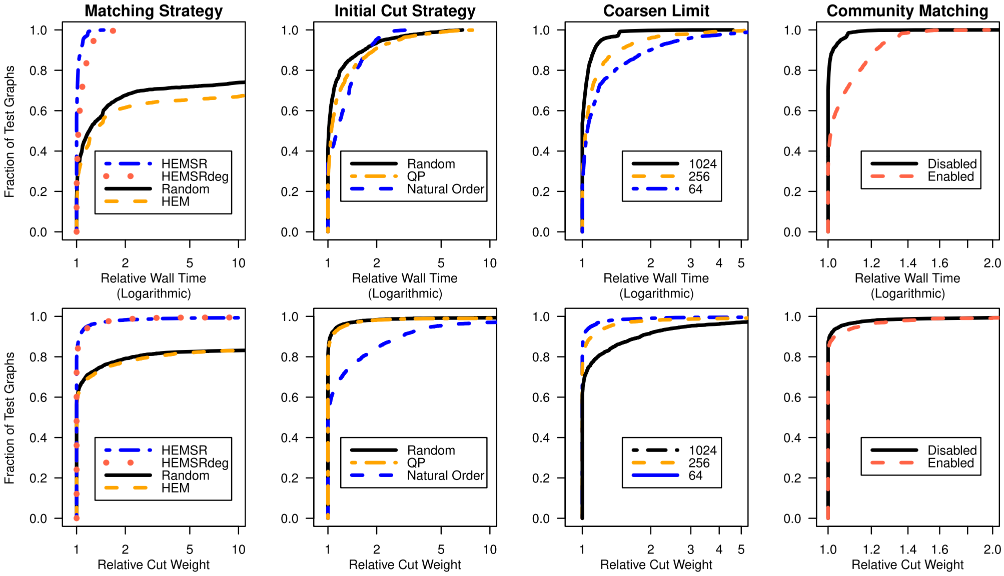


Fig. 9. Relative timing (top row) and cut quality (bottom row) performance profiles of each set of options. Each column corresponds to an available option in Mongoose. Note that the horizontal axis is logarithmic, and the vertical axis corresponds to the fraction of the 2,685 graphs used for testing. Runs that exceeded 7,200 seconds were terminated (as was the case for much of the HEM and Random matching strategy data).

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.

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.

7.5.3 *Coarsening Limit.* There is a tradeoff between speed and cut quality in determining the coarsening limit. If coarsening is terminated early (1,024 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.

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 it 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.

8 CONCLUSION

We have demonstrated a novel graph partitioning library utilizing new stall-free coarsening strategies and a hybrid refinement strategy utilizing quadratic programming in tandem with combinatoric methods to construct edge cuts in arbitrary graphs.

8.1 Future Work

Moving forward, we plan to extend this edge partitioning library to compute vertex separators and to ultimately incorporate this work into a nested dissection framework for computing fill-reducing orderings. Hypergraph partitioning, k-way partitioning, and parallelization of the library are also planned extensions.

Further investigation is also warranted in other mathematical programming formulations of graph partitioning problems. While a variety of formulations of the edge cut and vertex separator problems exist, formulations of other related problems require further development.

REFERENCES

P. Boldi, M. Rosa, M. Santini, and S. Vigna. 2011. 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.

P. Boldi and S. Vigna. 2004. The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*. ACM Press, 595–601.

E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. 2012. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Sci. Prog.* 20, 2 (2012), 129–150.

T. A. Davis, W. W. Hager, and J. T. Hungerford. 2016. An efficient hybrid algorithm for the separable convex quadratic knapsack problem. *ACM Trans. Math. Softw.* 42, 3, Article 22 (May 2016), 25 pages.

T. A. Davis and Y. Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.

E. D. Dolan and J. J. Moré. 2002. Benchmarking optimization software with performance profiles. *Math. Prog.* 91 (2002), 201–213.

C. M. Fiduccia and R. M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Conference on Design Automation, 1982.*175–181. DOI: https://doi.org/10.1109/DAC.1982.1585498

A. Gupta. 1997. Fast and effective algorithms for graph partitioning and sparse-matrix ordering. *IBM J. Res. Dev.* 41, 1.2 (Jan. 1997), 171–183. DOI: https://doi.org/10.1147/rd.411.0171

W. W. Hager and Y. Krylyuk. 1999. Graph partitioning and continuous quadratic programming. *SIAM J. Disc. Math.* 12, 4 (1999), 500–523.

B. Hendrickson and R. Leland. 1993. *The Chaco Users Guide. Version 1.0.* Technical Report. Sandia National Labs., Albuquerque, NM.

B. Hendrickson and R. Leland. 1995. A multi-level algorithm for partitioning graphs. In *Proceedings of the SuperComputing Conference* (1995), 28. DOI: https://doi.org/10.1109/SUPERC.1995.3

D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. 1989. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Op. Res.* 37, 6 (1989), 865–892.

E. L. Johnson, A. Mehrotra, and G. L. Nemhauser. 1993. Min-cut clustering. *Math. Prog.* 62, 1–3 (1993), 133–151.

G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. 1999. Multilevel hypergraph partitioning: Applications in VLSI domain. *IEEE Trans. Very Large Scale Integ. Syst.* 7, 1 (1999), 69–79.

G. Karypis and V. Kumar. 1995. Multilevel graph partitioning schemes. In *Proceedings of the International Conference on Parallel Processing*. CRC Press, 113–122.

G. Karypis and V. Kumar. 1998a. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

George Karypis and Vipin Kumar. 1998b. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

G. Karypis, K. Schloegel, and V. Kumar. 1997. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota* (1997), 22.

B. W. Kernighan and S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* 49, 2 (1970), 291–307. DOI: https://doi.org/10.1002/j.1538-7305.1970.tb01770.x

D. LaSalle and G. Karypis. 2013. Multi-threaded graph partitioning. In *Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13)*. IEEE, 225–236.

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

F. Pellegrini and J. Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of the International Conference on High-Performance Computing and Networking*. Springer, 493–498.

A. Pothen. 1997. Graph partitioning algorithms with applications to scientific computing. In *Parallel Numerical Algorithms*. Springer, 323–368.

R. Preis and R. Diekmann. 1997. PARTY—A software library for graph partitioning. *Adv. Computat. Mech. Parallel Distrib. Proc.* (1997), 63–71.

P. Sanders and C. Schulz. 2013. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13) (LNCS)*, Vol. 7933. Springer, 164–175.

Received April 2018; revised January 2019; accepted May 2019