Mongoose User Guide

Scott Kolodziej

February 9, 2018

Contents

1 Overview **3**
1.1 Coarsening and Refinement Framework . . . 3
1.2 Quadratic Programming and Optimization . . . 4
1.3 Fiduccia-Mattheyses Algorithm . . . 4

2 Availability **5**
2.1 Getting the Code . . . 5
2.2 Prerequisites . . . 5
2.3 Compilation . . . 5

3 Using Mongoose as an Executable **6**
3.1 License . . . 7

4 Using Mongoose in C++ **7**
4.1 Sample C++ Program . . . 7
4.2 C++ API . . . 8
4.3 A Note on Memory Management . . . 9

5 Using Mongoose in MATLAB **9**
5.1 Sample MATLAB Program . . . 9
5.2 MATLAB API . . . 11

6 Options **12**
6.1 Coarsening Options . . . 12
6.2 Initial Guess/Partitioning Options . . . 13
6.3 Waterdance Options . . . 14
6.4 Fiduccia-Mattheyes Options . . . 14
6.5 Quadratic Programming Options . . . 15
6.6 Final Partition Target Options . . . 15
6.7 Other Options . . . 16

7 References **16**

1 Overview

Mongoose is a graph partitioning library that can quickly compute edge cuts in arbitrary graphs. Given a graph with a vertex set V and edge set E, an edge cut is a partitioning of the graph into two subgraphs that are balanced (contain the same number of vertices) and the connectivity between the subgraphs is minimized (few edges are in the cut).

Finding high quality edge cuts quickly is an important part of circuit simulation, parallel and distributed computing, and sparse matrix algorithms.

1.1 Coarsening and Refinement Framework

Mongoose uses a coarsening and refinement framework (sometimes referred to as a multilevel framework [4, 5]). Rather than attempt to compute an edge cut on the input graph directly, Mongoose first coarsens the graph by computing a vertex matching and contracting the graph to form a smaller, but structurally similar, graph.

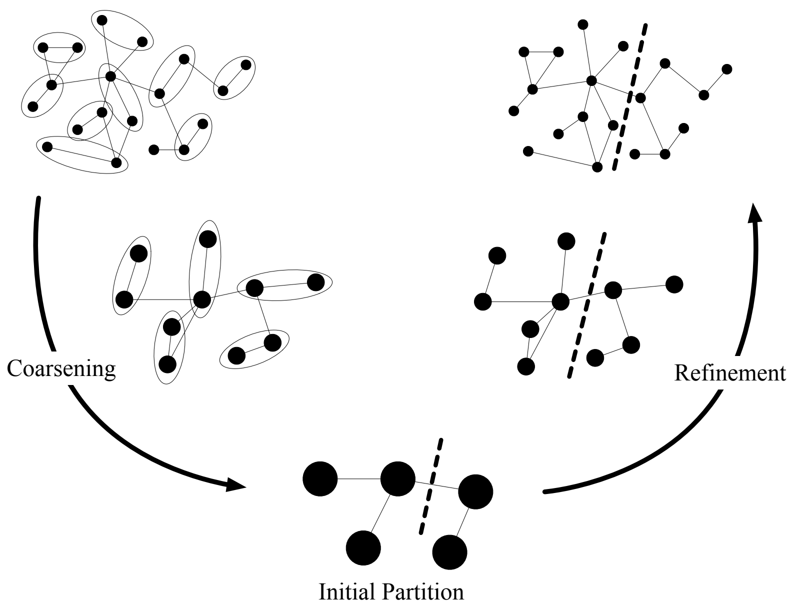

Figure 1: Coarsening and Refinement

Mongoose uses a variety of methods to coarsen the input graph, including random matching and heavy-edge matching. Additionally, Mongoose offers stall-reducing vertex matching strategies called Brotherly (or two-hop) matching and Community matching. Brotherly matching allows vertices who share a neighbor to be matched, even if they have no edge directly connecting them, and community matching allows two vertices whose neighbors are matched together to be matched together. These methods are advantageous in efficiently coarsening certain classes of graphs, notably social networking graphs, where the vertex degree can vary greatly.

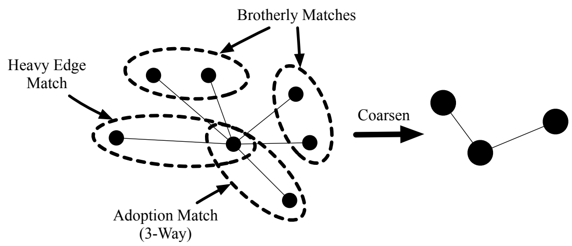

Figure 2: Brotherly Matching

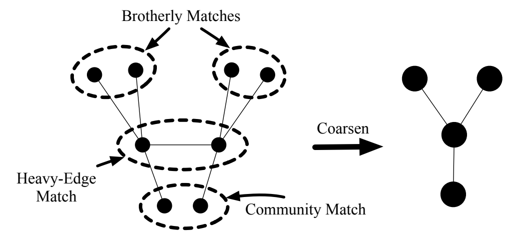

Figure 3: Community Matching

Another matching strategy used in Mongoose is known as Adoption matching. If an unmatched vertex has no unmatched neighbors, it can be grouped into a 3-way matching with a neighboring matched vertex. These strategies allow the graph to be coarsened quickly even when the graph is highly irregular, which in turn decreases memory requirements and overall computational time.

1.2 Quadratic Programming and Optimization

Mongoose is known as a hybrid graph partitioner, as it uses multiple methods in tandem to find higher quality cuts efficiently. The first such method Mongoose employs is quadratic programming (QP). The edge cut problem was formatted as a continuous quadratic programming problem by Hager and Krylyuk [3]. This formulation is solved (rather, improved) using a gradient projection algorithm and a modified version of NAPHEAP, a quadratic knapsack solver [1].

The quadratic program used is shown below. Hager and Krylyuk have proven that the global optimum to this quadratic program yields the solution to the graph partitioning problem (but note that both are NP-hard problems to solve).

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

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.

1.3 Fiduccia-Mattheyses Algorithm

In addition to the quadratic programming approach for refining an edge cut, a standard implementation of the Fiduccia-Mattheyses algorithm [2] is also provided. This involves swapping vertices from one part to the other in an effort to improve the edge cut quality. Some vertices are swapped even if no immediate improvement is found in an attempt to escape a locally optimal solution. However, if no improvement is found after a number of swaps, the change is reverted.

The Fiduccia-Mattheyses (FM) implementation in Mongoose utilizes heaps for high efficiency.

2 Availability

2.1 Getting the Code

Mongoose is available on GitHub at https://github.com/ScottKolo/Mongoose. The code can be downloaded using git using the following command:

```
git clone https://github.com/ScottKolo/Mongoose
```

Alternatively, Mongoose can be downloaded as a zip archive from the following URL:

https://github.com/ScottKolo/Mongoose/archive/edgesep.zip

2.2 Prerequisites

Mongoose requires CMake 2.8 and any ISO/IEC 14882:1998 compliant C++ compiler. Mongoose has been tested to work with GNU GCC 4.4+ and LLVM Clang 3.5+ on Linux, and Apple Xcode 6.4+ on macOS.

2.3 Compilation

Once downloaded, Mongoose can be compiled using the following commands:

```
cd Mongoose
mkdir _build # Create a build directory
cd _build
cmake ..     # Use CMake to create the Makefiles
make         # Build Mongoose
```

After compilation, the Mongoose demo can be run using ./bin/Demo:

```
./bin/Demo
****************************************************
Computing an edge cut for bcspwr01.mtx...
Partitioning Complete!
Cut Cost:       3
Cut Imbalance:  1.3%
Normalized Cut: 0.13
Trial Time:     1ms
****************************************************

...

****************************************************
Computing an edge cut for troll.mtx...
Partitioning Complete!
Cut Cost:       4.1e+04
```

```
Cut Imbalance:  0.00023%
Normalized Cut: 0.014
Trial Time:     3.28e+03ms
**************************************************
Total Demo Time:  3.4s

**************************************************
***************** Demo Complete! *****************
**************************************************
```

To run the complete test suite, the command `make test` can be used. Note that Python 2.7+ must be installed.

3 Using Mongoose as an Executable

In addition to the Demo executable, the `mongoose` executable is built at `./bin/mongoose`. This executable can be used to partition a graph given a Matrix Market file:

```
mongoose <MM-input-file.mtx> [output-file]
```

The `mongoose` executable generates a text file with two blocks: a JSON-formatted information block with timing and cut quality metrics, and the partitioning information itself. The partitioning information is listed with one vertex per line, with the vertex number followed by the part (0 for part A, 1 for part B).

For example, the following can be used to partition `troll.mtx`:

```
./bin/mongoose ../Matrix/troll.mtx troll_out.mtx
Total Edge Separator Time: 1.00848s
 Matching:   0.08622s
 Coarsening: 0.2118s
 Refinement: 0.05528s
 FM:         0.003129s
 QP:         0.5684s
 IO:         2.522s
Cut Properties:
 Cut Size:       50837
 Cut Cost:       5.084e+04
 Normalized Cut: 0.01727
 Imbalance:      2.342e-06

cat jagmesh7_out.txt
{
  "InputFile": "../Matrix/troll.mtx",
  "Timing": {
    "Total": 1.00848,
    "Matching": 0.086223,
    "Coarsening": 0.211779,
    "Refinement": 0.055275,
    "FM": 0.003129,
```

```
    "QP": 0.568422,
    "IO": 2.52204
  },
  "CutSize": 50837,
  "CutCost": 50837,
  "NormCut": 0.0172744,
  "Imbalance": 2.34244e-06
}

0 0
1 0
2 0

...

204829 1
204830 1
204831 0
204832 0

...

213450 0
213451 0
213452 0
```

The output file name is optional. If omitted, the default is `mongoose_out.txt`.

3.1 License

Mongoose is licensed under the GNU Public License version 3 (GPLv3). Full text of the license can be found int `Mongoose/LICENSE`. For a commercial license, please contact Dr. Timothy A. Davis at davis@tamu.edu.

4 Using Mongoose in C++

4.1 Sample C++ Program

```cpp
#include "Mongoose.hpp"
#include <iostream>
#include <iomanip>

using namespace Mongoose;
using namespace std;

int main(int argn, const char **argv)
{
    Options *options = Options::Create();
```

```
    if (!options) return EXIT_FAILURE; // Return an error if we failed.

    options->matchingStrategy = HEMDavisPA;
    options->guessCutType = GuessQP;

    Graph *graph = readGraph(argv[1]);
    if (!graph)
    {
      options->~Options();
      return EXIT_FAILURE;
    }

    // Call Mongoose to compute an edge separator
    ComputeEdgeSeparator (graph, options);

    cout << "Partitioning Complete!" << endl;
    cout << "Cut Cost:       " << setprecision(2) << graph->cutCost << endl;
    cout << "Cut Imbalance: " << setprecision(2) << fabs(100*graph->imbalance) << "%" << endl;

  options->~Options();
    graph->~Graph();

    /* Return success */
    return EXIT_SUCCESS;
}
```

4.2 C++ API

The following functions are available in the C++ API. After Mongoose is compiled, a static library version of Mongoose is built at `Mongoose/_build/lib/libmongoose.a`. Include the `Mongoose.hpp` header file located in `Mongoose/Include` and link with the static library to enable the following API functions.

- **`Graph *readGraph(const std::string &filename);`**
- **`Graph *readGraph(const char *filename);`**

  `Mongoose::readGraph` will attempt to read a Matrix Market file with the given filename and convert it to a Mongoose Graph instance. The matrix contained in the file must be sparse, real, and square. If the matrix is not symmetric, it will be made symmetric by computing $\frac{1}{2}(A + A^T)$. If a diagonal is present, it will be removed.

  `Mongoose::readGraph(const std::string &filename)` accepts a C++-style std::string, while `Mongoose::readGraph(const char *filename)` accepts a C-style null-terminated string.

- **`int ComputeEdgeSeparator(Graph *);`**
- **`int ComputeEdgeSeparator(Graph *, const Options *);`**

Mongoose::ComputeEdgeSeparator will attempt to compute an edge cut of the provided Mongoose::Graph object. An Options struct can also be supplied to modify how the edge cut is computed – otherwise, the default options are used (see Section 6).

- **static Options *Create();**

  Mongoose::Options::Create will return an Options struct with default state (see Section 6 for details about option fields and defaults). To run Mongoose with specific options, call Options::Create and modify the struct as needed.

- **static Graph *Create(Int _n, Int _nz);**
- **static Graph *Create(Graph *_parent);**
- **~Options();**
- **~Graph();**

4.3 A Note on Memory Management

Mongoose uses two primary data structures to pass information: the Graph class and the Options struct. Both are dynamically allocated and must be destructed.

- For each Graph::Create, there should be a matching Graph::~Graph().
- for each Options::Create, there should be a matching Options::~Options();

5 Using Mongoose in MATLAB

5.1 Sample MATLAB Program

Below is a sample MATLAB program using the Mongoose MATLAB API. First, it loads in a matrix, sanitizes it, and then partitions it using edge and vertex weights, then only edge weights, and the no weights.

```
% A simple demo to demonstrate Mongoose. Reads in a matrix, sanitizes it,
% and partitions it several different ways.
function mongoose_demo

% Obtain the adjacency matrix
matfile_data = matfile('494_bus.mat');
Prob = matfile_data.Problem;
A = Prob.A;
[m ~] = size(A);

% Sanitize the adjacency matrix: remove diagonal elements, make edge weights
% positive, and make sure it is symmetric. If the matrix is not symmetric
% or square, a symmetric matrix (A+A')/2 is built.
```

```
A = sanitize(A);

% Create a node weight vector and create a heavy node
V = ones(1,m);
V(10) = 300;

% Create a set of default options and modify the target balance
O = defaultoptions();
O.targetSplit = 0.3;

% Run Mongoose to partition the graph with edge and node weights.
partNode = edgecut(A, O, V);

fprintf('\n\nPartitioning graph with edge and node weights\n\n');
fprintf('=== Cut Info ===\n');
fprintf('Cut Size:   %d\n', full(sum(partNode .* sum(sign(A)))));
fprintf('Cut Weight: %d\n\n', full(sum(partNode .* sum(A))));
fprintf('=== Balance Info ===\n');
fprintf('Target Split:     0.3\n');
fprintf('Actual Split:     %1.4f\n', sum(partNode .* V) / sum(V));
fprintf('Unweighted Split: %1.4f\n', sum(partNode) / m);

% Run Mongoose to partition the graph with no vertex weights.
partEdge = edgecut(A, O);

fprintf('\n\nPartitioning graph with only edge weights\n\n');
fprintf('=== Cut Info ===\n');
fprintf('Cut Size:   %d\n', full(sum(partEdge .* sum(sign(A)))));
fprintf('Cut Weight: %d\n\n', full(sum(partEdge .* sum(A))));
fprintf('=== Balance Info ===\n');
fprintf('Target Split: 0.5\n');
fprintf('Actual Split: %1.4f\n', sum(partEdge) / m);

% Remove edge weights
A = sanitize(A, 1);

% Run Mongoose to partition the graph with no edge weights.
% Note that only the graph is passed as an argument, so default
% options are assumed.
partPattern = edgecut(A);

fprintf('\n\nPartitioning graph with only edge weights\n\n');
fprintf('=== Cut Info ===\n');
fprintf('Cut Size:   %d\n', full(sum(partPattern .* sum(sign(A)))));
fprintf('Cut Weight: %d\n\n', full(sum(partPattern .* sum(A))));
fprintf('=== Balance Info ===\n');
fprintf('Target Split: 0.5\n');
fprintf('Actual Split: %1.4f\n', sum(partPattern) / m);

figure('Position', [100, 100, 1000, 400]);

% Plot the original matrix before permutation
```

```
subplot(1, 2, 1);
spy(A)
title('Before Partitioning')

% Plot the matrix after the permutation
subplot(1, 2, 2);
perm = [find(partEdge) find(1-partEdge)];
A_perm = A(perm, perm); % Permute the matrix
spy(A_perm)
title('After Partitioning')

% Set overall title
suptitle('HB/494\_bus')

end
```

5.2 MATLAB API

- **`function [G_coarse, A_coarse, map] = coarsen (G, O, A)`**
  `coarsen` is used to coarsen an adjacency matrix (G) one level (one round of matching). An optional options struct (O) can be specified, as well as node weights (A).

- **`function options = defaultoptions()`**
  `defaultoptions()` returns an options struct with defaults set. If modifications to the default options are needed, call `defaultoptions()` and modify the struct as needed. See section 6 for details on available option fields.

- **`function partition = edgecut (G, O, A)`**
  `edgecut` computes an edge cut of the graph G with options O and vertex weights A, such that `A(i)` $=$ weight(v_i). The returned array, `partition`, is a $1 \times n$ binary array such that

$$\texttt{partition(i)} = \begin{cases} 0 & \text{if } v_i \in \text{part A} \\ 1 & \text{if } v_i \in \text{part B} \end{cases}$$

- **`function [G_coarse, A_coarse, map] = safe_coarsen (G, O, A)`**
  `safe_coarsen` attempts to coarsen a graph G with options O and vertex weights A. Prior to coarsening, `safe_coarsen` first calls `sanitize(G)` to ensure that the graph is able to be coarsened.

- **`function partition = safe_edgecut(G, O, A)`**
  `safe_edgecut` attempts to compute and edge cut for a graph G with options O and vertex weights A. Note that both O and A are optional arguments. `safe_edgecut` first calls `sanitize(G)` to ensure that

the graph is formatted correctly.

- **`function A_safe = sanitize (A, `*`make_binary`*`)`**
  `sanitize` attempts to take an adjacency matrix A and convert it to one that Mongoose can read and convert to an undirected graph. Note that `make_binary` is optional, with the default being `false`. `sanitize` does the following as needed:

  - If the matrix is unsymmetric, it forms $\frac{1}{2}(A^T + A)$.
  - The diagonal is removed (set to zero).
  - Edge weights are forced to be positive ($w = |w|$) if `make_binary = false`.
  - Edge weights are forced to be binary ($w = \text{sign}(w)$) if `make_binary = true`.

6 Options

When calling Mongoose, an optional Options struct can be provided to specify how Mongoose should behave.

6.1 Coarsening Options

Name	`coarsenLimit`
Type	`Int`
Default	`50`

Prior to computing a cut, the input graph is repeatedly coarsened until a sufficiently small number of vertices exist in the graph. This limit is specified by `coarsenLimit`. Larger values will result in less time being spent on the coarsening process, but may yield poor initial cuts or may require more time in computing such an initial cut. Smaller values may result in more time spent coarsening, as well as a resulting coarsened graph which is a poor structural representation of the input graph.

Name	`matchingStrategy`
Type	`MatchingStrategy` (enum)
Default	`HEMDavisPA`

During coarsening, a matching of vertices is computed using one of several strategies determined by the `matchingStrategy` option field. The possible values for this field are described below:

- `Random`, random matching. Randomly matches unmatched vertices with each other until no more than one unmatched vertex exists.

- `HEM`, heavy edge matching. Matches a given vertex with an unmatched neighbor with the largest weighted edge between them.

- HEMPA, heavy edge matching with passive-aggressive matching. A pass of heavy edge matching is followed by a passive-aggressive matching where vertices that have been left unmatched by heavy edge matching are paired with vertices that share a neighbor, but may not be directly connected.
- HEMDavisPA, heavy edge matching with passive-aggressive matching subject to a brotherly threshold. Same as HEMPA, but the passive-aggressive step is only attempted on unmatched vertices whose degree is above a threshold, described by Options::davisBrotherlyThreshold $*$ (average degree of graph). davisBrotherlyThreshold is set to 2.0 by default, meaning only unmatched vertices with degree greater than or equal to two times the average degree of the graph are considered for passive-aggressive matching.

Name	doCommunityMatching
Type	bool
Default	false

Community matching is a matching option to aggressively match vertices whose neighbors have already been matched. This can help in cases where coarsening easily stalls (e.g. social networking graphs), but incurs a slight performance overhead during coarsening.

Name	davisBrotherlyThreshold
Type	double
Default	2.0

When using the HEMDavisPA matching strategy, only vertices satisfying the following inequality are considered for passive-aggressive matching:

$$\text{degree}(v) \geq \lfloor (\texttt{davisBrotherlyThreshold}) \cdot \left(\frac{nz}{n}\right) \rfloor$$

Note that $\frac{nz}{n}$ is the average degree of the vertices in the graph.

6.2 Initial Guess/Partitioning Options

Name	guessCutType
Type	GuessCutType (enum)
Default	GuessQP

After coarsening, an initial partitioning is computed. This initial guess can be computed several ways:

- GuessQP. This method uses the quadratic programming solver to compute an initial partitioning.
- GuessRandom. This method randomly assigns vertices to a part.
- GuessNaturalOrder. This method assigns the first $\lfloor n/2 \rfloor$ vertices listed to one part, and the remainder to the other part.

6.3 Waterdance Options

Name	`numDances`
Type	`Int`
Default	`1`

At each level of graph refinement, both the Fiduccia-Mattheyses refinement algorithm and the quadratic programming algorithm are used to refine the graph. This combination of algorithms, run back-to-back, is informally referred to as a waterdance. `numDances` is used to specify the number of waterdances.

For example, if `numDances = 2`, at each refinement level, the FM refinement will be done, then QP refinement, then FM and QP again.

6.4 Fiduccia-Mattheyes Options

Name	`useFM`
Type	`bool`
Default	`true`

`useFM` can be used to enable or disable the use of the Fiduccia-Mattheyses refinement algorithm. If `useFM` is `false`, then the FM refinement is skipped.

Name	`fmSearchDepth`
Type	`Int`
Default	`50`

The Fiduccia-Mattheyses algorithm attempts to make positive gain moves whenever possible. However, to better explore the non-convex search space, the FM algorithm will make unfavorable moves in an attempt to locate another more favorable solution. The `fmSearchDepth` limits the number of these unfavorable moves before the algorithm stops.

Name	`fmConsiderCount`
Type	`Int`
Default	`3`

During the Fiduccia-Mattheyses algorithm, a heap is maintained of the vertices sorted by their gains. Vertices that have fewer neighbors in the same part relative to neighbors in the opposite part are prioritized higher in the heap (with higher gains), and are generally more likely to yield better quality cuts when swapped to the opposite part. `fmConsiderCount` defines the number of vertices at the top of the heap to consider swapping to the opposite part before terminating. When a vertex swap being considered does not yield a better cut after moving `fmSearchDepth` vertices, that iteration terminates, and the next vertex in the heap is considered.

Name	`fmMaxNumRefinements`
Type	`Int`
Default	`20`

`fmMaxNumRefinements` specifies the number of passes the Fiduccia-Mattheyses algorithm takes over the graph. During each pass, suboptimal moves may be attempted to escape local optima.

6.5 Quadratic Programming Options

Name	`useQPGradProj`
Type	`bool`
Default	`true`

`useQPGradProj` can be used to enable or disable the use of the quadratic programming refinement algorithm. If `useQPGradProj` is `false`, then the QP refinement is skipped. This may provide faster solutions at the cost of cut quality.

Name	`gradProjTolerance`
Type	`double`
Default	`0.001`

Convergence tolerance for the projected gradient algorithm in the quadratic programming refinement approach. Decreasing the tolerance may improve solution quality at the cost of additional computation time. It may also be advisable to increase `gradprojIterationLimit`, as a decreased tolerance may require additional iterations to converge.

Name	`gradprojIterationLimit`
Type	`Int`
Default	`50`

Maximum number of iterations for the gradient projection algorithm in the quadratic programming refinement approach. More iterations may allow the gradient projection algorithm to find a better solution at the cost of additional computation time.

6.6 Final Partition Target Options

Name	`targetSplit`
Type	`double`
Default	`0.5`

`targetSplit` specifies the desired balance of the edge cut. The default is a balanced cut (0.5). Note that the target split takes into account weighted vertices.

Name	`softSplitTolerance`
Type	`double`
Default	`0`

Cuts within `targetSplit` $\pm$ `softSplitTolerance` are treated equally. For example, if any cut within 0.4 and 0.6 balance is acceptable, the user may specify `targetSplit` $= 0.5$ and `softSplitTolerance` $= 0.1$.

6.7 Other Options

Name	`randomSeed`
Type	`Int`
Default	`0`

Random number generation is used primarily in random matching strategies (`matchingStrategy = Random`) and random initial guesses (`guessCutType = GuessRandom`). `randomSeed` can be used to seed the random number generator with a specific value.

7 References

References

[1] Davis, T. A., Hager, W. W., and Hungerford, J. T. An efficient hybrid algorithm for the separable convex quadratic knapsack problem. *ACM Trans. Math. Softw. 42*, 3 (May 2016), 22:1–22:25.

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

[3] Hager, W. W., and Krylyuk, Y. Graph partitioning and continuous quadratic programming. *SIAM Journal on Discrete Mathematics 12*, 4 (1999), 500–523.

[4] Hendrickson, B., and Leland, R. A multi-level algorithm for partitioning graphs. *SC Conference 0* (1995), 28.

[5] Karypis, G., and Kumar, V. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 1995), Supercomputing '95, ACM.