Mongoose User Guide, Version 2.0.0

Scott Kolodziej, Nuri Yeralan, Tim Davis, William W. Hager

July 1, 2018

Contents

1 Overview

Mongoose is a graph partitioning library that can quickly compute edge cuts in arbitrary graphs [2]. 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 [5, 6]). 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.

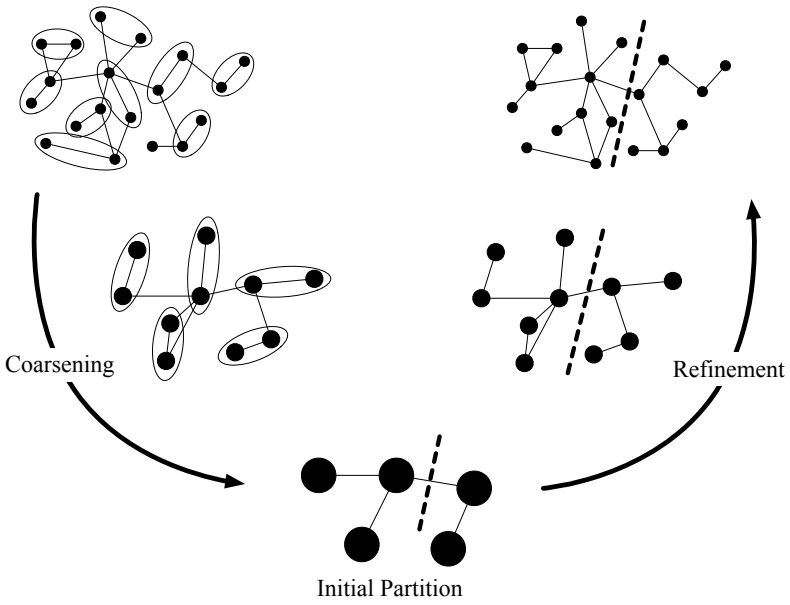

Figure 1: Coarsening and Refinement

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.

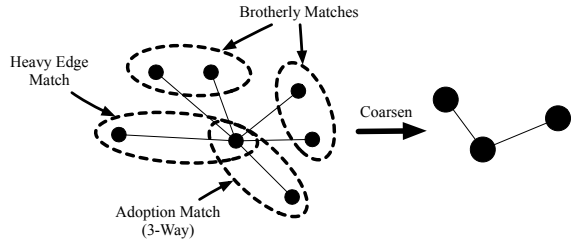

Figure 2: Brotherly Matching

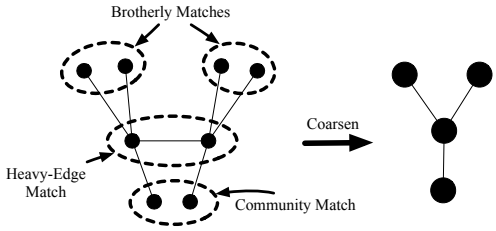

Figure 3: Community Matching

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 [4]. 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 [3] 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
```

Mongoose also appears as a component package of SuiteSparse, `http://suitesparse.com`.

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
********************************************************************************
Mongoose Graph Partitioning Library, Version 2.0.0 June 30, 2018
Copyright (C) 2017–2018
Scott P. Kolodziej, Nuri S. Yeralan, Timothy A. Davis, William W. Hager
Mongoose is licensed under Version 3 of the GNU General Public License.
Mongoose is also available under other licenses; contact authors for details.
********************************************************************************
Computing an edge cut for bcspwr01.mtx...
Partitioning Complete!
Cut Cost:       3
Cut Imbalance:  1.3%
Trial Time:     0.61ms
```

```
********************************************************************************
Computing an edge cut for bcspwr02.mtx...
Partitioning Complete!
Cut Cost:       5
Cut Imbalance:  1%
Trial Time:     0.37ms
********************************************************************************
Computing an edge cut for bcspwr03.mtx...
Partitioning Complete!
Cut Cost:       11
Cut Imbalance:  0%
Trial Time:     0.55ms
********************************************************************************
Computing an edge cut for bcspwr04.mtx...
Partitioning Complete!
Cut Cost:       25
Cut Imbalance:  2.6e-16%
Trial Time:     1.5ms
********************************************************************************
Computing an edge cut for bcspwr05.mtx...
Partitioning Complete!
Cut Cost:       12
Cut Imbalance:  0.11%
Trial Time:     1.1ms
********************************************************************************
Computing an edge cut for bcspwr06.mtx...
Partitioning Complete!
Cut Cost:       7
Cut Imbalance:  1.5e-15%
Trial Time:     2.8ms
********************************************************************************
Computing an edge cut for bcspwr07.mtx...
Partitioning Complete!
Cut Cost:       7
Cut Imbalance:  0%
Trial Time:     2.7ms
********************************************************************************
Computing an edge cut for bcspwr08.mtx...
Partitioning Complete!
Cut Cost:       25
Cut Imbalance:  3.3e-16%
Trial Time:     2.5ms
********************************************************************************
Computing an edge cut for bcspwr09.mtx...
Partitioning Complete!
Cut Cost:       11
Cut Imbalance:  0.029%
Trial Time:     2.7ms
********************************************************************************
Computing an edge cut for bcspwr10.mtx...
Partitioning Complete!
Cut Cost:       25
```

```
Cut Imbalance:  1.1e-15%
Trial Time:     8.9ms
********************************************************************************
Computing an edge cut for jagmesh7.mtx...
Partitioning Complete!
Cut Cost:       27
Cut Imbalance:  0%
Trial Time:     2.2ms
********************************************************************************
Computing an edge cut for troll.mtx...
Partitioning Complete!
Cut Cost:       3.7e+04
Cut Imbalance:  0.00023%
Trial Time:     2.5e+03ms
********************************************************************************
Total Demo Time:  2.5s


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

To run the complete test suite, the command `make test` can be used. Note that Python 2.7+ must be installed. Additionally, this user guide can be generated from source with the command `make userguide`. XeLaTeX (commonly included in LaTeX distributions) 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
********************************************************************************
Mongoose Graph Partitioning Library, Version 2.0.0 June 30, 2018
Copyright (C) 2017-2018
Scott P. Kolodziej, Nuri S. Yeralan, Timothy A. Davis, William W. Hager
Mongoose is licensed under Version 3 of the GNU General Public License.
Mongoose is also available under other licenses; contact authors for details.
********************************************************************************
Total Edge Separator Time: 0.403932s
```

```
 Matching:   0.03894s
 Coarsening: 0.1145s
 Refinement: 0.04114s
 FM:         0.00397s
 QP:         0.1837s
 IO:         2.039s
Cut Properties:
 Cut Size:       40517
 Cut Cost:       4.052e+04
 Imbalance:      2.342e-06

cat troll_out.txt
{
  "InputFile": "../Matrix/troll.mtx",
  "Timing": {
    "Total": 0.403932,
    "Matching": 0.03894,
    "Coarsening": 0.114478,
    "Refinement": 0.041145,
    "FM": 0.00397,
    "QP": 0.183679,
    "IO": 2.03914
  },
  "CutSize": 40517,
  "CutCost": 40517,
  "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/Doc/License.txt`. For a commercial license, please contact Dr. Timothy A. Davis at davis@tamu.edu.

4 Using Mongoose in C++

4.1 Sample C++ Program

```
#include "Mongoose.hpp"
#include <iostream>
#include <iomanip>
#include <math.h>

using namespace Mongoose;
using namespace std;

int main(int argn, const char **argv)
{
    EdgeCut_Options *options = EdgeCut_Options::create();
    if (!options) return EXIT_FAILURE; // Return an error if we failed.

    options->matching_strategy = HEMSRdeg;
    options->initial_cut_type = InitialEdgeCut_QP;

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

    // Call Mongoose to compute an edge separator
    EdgeCut *result = edge_cut(graph, options);

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

    options->~EdgeCut_Options();
    graph->~Graph();
    result->~EdgeCut();

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

4.2 Creating a Graph

There are several different ways to create a `Graph` class instance for use in Mongoose.
The input graph to Mongoose is undirected, and can optionally be a weighted graph. The graph is held in compressed-sparse column form, or equivalently compressed-sparse row form since the adjacency matrix is symmetric. The graph is represented by the following components:

- `Int n`: the number of vertices in the graph.
- `Int nz`: the number of nonzero entries in the adjacency matrix, which is twice the number edges.
- `Int p [n+1]`: the column pointer vector of size `n+1`.
- `Int i [nz]`: the adjacency lists held in a single array. The adjacency list of vertex `j` is held in `i [p [j] ... p [j+1]-1]`. Self-edges must not appear. The graph must be undirected, and so the adjacency matrix must be symmetric.
- `double x [nz]`: an optional array of edge weights, where `x [p [j] ... p [j+1]-1]` are the edge weights of the corresponding edges in the adjacency list of vertex `j`. If `x` is `NULL`, then the edges all have weight 1.
- `double w [n]`: an optional array of vertex weights, where vertex `j` has weight `w [j]`. If `w` is `NULL`, then the vertices all have weight 1.

Note that the `Int` type is generally a 64-bit (long) integer type. It is defined as `typedef SuiteSparse_long Int;` which is further defined as `#define SuiteSparse_long long` in SuiteSparse_config.

4.2.1 Creating a Graph Manually

To create a graph manually, the following constructor is used:

```
static Graph *create(const Int _n,
                     const Int _nz,
                     Int *_p = NULL,
                     Int *_i = NULL,
                     double *_x = NULL,
                     double *_w = NULL);
```

Using the manual constructor, the number of vertices (or dimension of the matrix, `Int _n`) and the number of edges (or nonzero entries in the matrix, `Int _nz`) must both be specified. The column pointer vector `Int *_p` and row index vector `Int *_i` must either be specified by the user, or, if left `NULL`, they will be allocated such that `_p = (Int *)SuiteSparse_calloc(n + 1, sizeof(Int));` and `_i = (Int *)SuiteSparse_malloc(nz, sizeof(Int));`. The edge weights `double *_x` and the vertex weights `double *_w` can either be specified, or, if left `NULL`, will be assumed to be one for all edges and vertices,

respectively.

Here are some examples:

- `Graph::create(20, 50);` creates a `Graph` with 20 vertices and 50 edges, but no data. `Graph->p` and `Graph->i` are allocated by Mongoose to store exactly 20 columns (vertices) and 50 nonzero elements (edges). This allows the user to populate `Graph->p` and `Graph->i` manually. All edge and vertex weights are assumed to be one.
- `Graph::create(20, 50, _p, _i);` creates a `Graph` with 20 vertices and 50 edges with the pattern specified. `Graph->p` and `Graph->i` are shallow copies of the arguments `_p` and `_i` and will not be freed upon calling the destructor. All edge and vertex weights are assumed to be one.
- `Graph::create(20, 50, _p, _i, _x);` creates a `Graph` with 20 vertices and 50 edges with the pattern and edge weights specified. `Graph->p`, `Graph->i`, and `Graph->x` are shallow copies of the arguments `_p`, `_i`, and `_x`, and will not be freed upon calling the destructor. Edge weights are specified by `_x`, but vertex weights are assumed to be one.
- `Graph::create(20, 50, _p, _i, _x, _w);` creates a `Graph` with 20 vertices and 50 edges with edge and vertex weights specified. `Graph->p`, `Graph->i`, `Graph->x`, and `Graph->w` are shallow copies of the arguments `_p`, `_i`, `_x`, and `_w`, and will not be freed upon calling the destructor. Edge weights are specified by `_x`, and vertex weights are specified by `_w`.

4.2.2 Creating a Graph from a Sparse Matrix

Another way to create a `Mongoose::Graph` is from a pre-existing `CSparse` matrix struct.

```
static Graph *Create(cs *matrix);
```

Using this constructor, a `CSparse` matrix struct can be passed directly, with shallow copies made of `cs->p`, `cs->i`, and `cs->x`. Note that this constructor is equivalent to calling the following for a `CSparse` matrix `cs *A`:

```
Graph::create(const Int A->n, A->p[A->n], A->p, A->i, A->x, NULL);
```

4.2.3 Creating a Graph from a Matrix Market File

Perhaps the easiest way to create a `Graph` instance is from a file. Mongoose provides easy file input helpers to read, sanitize, and format a Matrix Market file. 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.

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

For example, to read in the Matrix Market file `jagmesh7.mtx` located in the Mongoose sample matrix directory, the following code can be used:

```
Mongoose::read_graph("../Matrix/jagmesh7.mtx");
```

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

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

- **`int edge_cut(const Graph *);`**
- **`int edge_cut(const Graph *, const EdgeCut_Options *);`**

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

- **`static EdgeCut_Options *create();`**

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

- **`static Graph *create(const Int _n,`**
  `const Int _nz,`
  `Int *_p = NULL,`
  `Int *_i = NULL,`
  `double *_x = NULL,`
  `double *_w = NULL);`
- **`static Graph *create(cs *matrix);`**

`Mongoose::Graph::create` is the primary constructor for the `Graph` class. There are two versions: one to manually specify attributes of the `Graph`, and one to form a `Graph` from a `CSparse` struct.

Using the manual constructor, the number of vertices (or dimension of the matrix, `Int _n`) and the number of edges (or nonzero entries in the matrix, `Int _nz`) must both be specified. The column pointer vector `Int *_p` and row index vector `Int *_i` must either be specified by the user, or, if left `NULL`, they will be allocated such that `_p = (Int *)SuiteSparse_calloc(n + 1, sizeof(Int));` and `_i = (Int *)SuiteSparse_malloc(nz, sizeof(Int));`. The edge weights `double *_x` and the vertex weights `double *_w` can either be specified, or, if left `NULL`, will be assumed to be one for all edges and vertices, respectively.

Note that Mongoose will NOT free pointers passed to it, and that all pointers are shallow copies (i.e. Mongoose does not make a copy of any data passed into it).

- ∼**`EdgeCut_Options();`** is the destructor for the `EdgeCut_Options` struct. If the user creates an `EdgeCut_Options` struct using `EdgeCut_Options::create`, the user is also responsible for destructing it.
- ∼**`Graph();`** is the destructor for the `Graph` class. If the user creates a `Graph` class instance using `Graph::create`, the user is also responsible for destructing it.

4.4 A Note on Memory Management

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

- For each `Graph::create`, there should be a matching `Graph::∼Graph()`.
- For each `EdgeCut_Options::create`, there should be a matching `EdgeCut_Options::∼EdgeCut_Options();`

Also note that `edge_cut()` returns an `EdgeCut` struct that is dynamically allocated. After use, it should be freed with `EdgeCut::∼EdgeCut()`.

Lastly, Mongoose will NOT free pointers passed to it, and that all pointers are shallow copies (i.e. Mongoose does not make a copy of any data passed into it). Freeing memory referenced by Mongoose prior to Mongoose completing will result in a segmentation fault.

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

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

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

fprintf('\n\nPartitioning graph with edge and vertex weights\n\n');
fprintf('=== Cut Info ===\n');
fprintf('Cut Size:   %d\n', full(sum(partVert .* sum(sign(A)))));
fprintf('Cut Weight: %d\n\n', full(sum(partVert .* sum(A))));
fprintf('=== Balance Info ===\n');
fprintf('Target Split:     0.3\n');
fprintf('Actual Split:     %1.4f\n', sum(partVert .* V) / sum(V));
fprintf('Unweighted Split: %1.4f\n', sum(partVert) / 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, Opts, A)`**
  `coarsen` is used to coarsen an adjacency matrix (`G`) one level (one round of matching). An optional edgecut_options struct (`Opts`) can be specified, as well as vertex weights (`A`).

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

- **`function partition = edgecut (G, Opts, A)`**
  `edgecut` computes an edge cut of the graph `G` with edgecut_options `Opts` 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, *Opts*, *A*)**
  safe_coarsen attempts to coarsen a graph G with edgecut_options Opts 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, *Opts*, *A*)**
  safe_edgecut attempts to compute and edge cut for a graph G with edgecut_options Opts and vertex weights A. Note that both Opts 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 ($w =$ sign(w)) if make_binary = true.

6 Options

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

6.1 Coarsening Options

Name	coarsen_limit
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 coarsen_limit. 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	matching_strategy
Type	MatchingStrategy (enum)
Default	HEMSR

During coarsening, a matching of vertices is computed using one of several strategies determined by the matching_strategy 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.
- HEMSR, heavy edge matching with stall-reducing matching. A pass of heavy edge matching is followed by a brotherly, adoption, and community (if enabled) 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.
- HEMSRdeg, heavy edge matching with stall-reducing matching subject to a degree threshold. Same as HEMSR, but the stall-reducing step is only attempted on unmatched vertices whose degree is above a threshold, described by EdgeCut_Options::high_degree_threshold $*$ (average degree of graph). high_degree_threshold 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 stall-reducing matching.

Name	do_community_matching
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	high_degree_threshold
Type	double
Default	2.0

When using the HEMSRdeg matching strategy, only vertices satisfying the following inequality are considered for brotherly, adoption, and community matching:

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

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

6.2 Initial Guess/Partitioning Options

Name	initial_cut_type
Type	InitialEdgeCutType (enum)
Default	InitialEdgeCut_QP

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

- InitialEdgeCut_QP. This method uses the quadratic programming solver to compute an initial partitioning.
- InitialEdgeCut_Random. This method randomly assigns vertices to a part.
- InitialEdgeCut_NaturalOrder. 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	num_dances
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. num_dances is used to specify the number of waterdances.

For example, if num_dances = 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	use_FM
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	FM_search_depth
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 FM_search_depth limits the number of these unfavorable moves before the algorithm stops.

Name	FM_consider_count
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. FM_consider_count 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 FM_search_depth vertices, that iteration terminates, and the next vertex in the heap is considered.

Name	FM_max_num_refinements
Type	Int
Default	20

FM_max_num_refinements 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	use_QP_gradproj
Type	bool
Default	true

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

Name	gradproj_tolerance
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 gradproj_iteration_limit, as a decreased tolerance may require additional iterations to converge.

Name	gradproj_iteration_limit
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	target_split
Type	double
Default	0.5

target_split 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	soft_split_tolerance
Type	double
Default	0

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

6.7 Other Options

Name	random_seed
Type	Int
Default	0

Random number generation is used primarily in random matching strategies (matching_strategy = Random) and random initial guesses (initial_cut_type = InitialEdgeCut_Random). random_seed 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] Davis, T. A., Hager, W. W., Kolodziej, S. P., and Yeralan, N. S. Algorithm XXX: Mongoose, a graph coarsening and partitioning library. *ACM Trans. Math. Softw.*. Submitted.

[3] 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.

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

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

[6] 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.