MATGRAPH: A MATLAB TOOLBOX FOR GRAPH THEORY

EDWARD R. SCHEINERMAN

OVERVIEW

MATGRAPH is a toolbox for working with simple[1] graphs in MATLAB. The goal is to make interactive graph theory exploration simple and efficient.

In order to use this toolbox, you need a copy of MATLAB, available from The MathWorks. A few of the MATGRAPH functions require the additional Optimization Toolbox, also available from The MathWorks.

This document gives an overview of the MATGRAPH toolbox. More information can be found in the accompanying web pages (see §3).

In addition to providing a `graph` class, MATGRAPH also defines helper classes for working with permutations (see §4) and partitions (see §5).

1. GETTING MATGRAPH

1.1. **Download and install.** MATGRAPH is available for free from my web site. Go to

`http://www.ams.jhu.edu/~ers/matgraph`

and select the link in the sentence "You can download Matgraph by clicking here."

This should cause the file `matgraph-X.X.tgz` to be downloaded to your computer. (The `X.X` is the version number.) This is a compressed archive. To unpack on a UNIX system, give the command

```
tar xfz matgraph-X.X.tgz
```

(where `X.X` is replaced by the correct version number). On other computers, double clicking on the file's icon should serve the same purpose.

This should create a directory named `matgraph`. You may move this directory to any convenient location on your hard drive.

1.2. **License.** This software is copyrighted by Edward R. Scheinerman and released free of charge under the GNU General Purpose Licenses. A copy of this license can be found at

`http://www.gnu.org/licenses/gpl.html`

I would like to make this tool as useful to as many people as possible. I invite you to send me improvements for the current `.m` files as well as new `.m` files for added functionality.

2. USING MATGRAPH

2.1. **Basic principles.** We assume the reader is familiar with MATLAB. Before we discuss how to use MATGRAPH, it is crucial that the following basic principles be understood.

- All graphs handled by MATGRAPH are simple and undirected. That is, these graphs do not have loops or multiple edges. Each pair of distinct vertices either is not adjacent or else is joined by a single edge.
- The vertex set of all graphs in MATGRAPH is always of the form $\{1,2,\ldots,n\}$ where $n \geq 0$ is the number of vertices in the graph.

  An important consequence of this is that when a vertex is deleted from a graph, all vertices with higher value are renumbered.
- All graphs in MATGRAPH are specially defined `graph` objects. They do not behave in the same manner as, say, matrices in MATLAB. For the sake of efficiency, MATGRAPH functions are capable of modifying their arguments.

  Most MATLAB functions use *call by value* semantics. That is, a *copy* of the argument is sent to the function. For example, suppose we have a function named `func` defined in a file `func.m`. In MATLAB, issuing the command `func(A)` does not affect the value held in the ordinary variable `A`.

  MATGRAPH, however, is designed differently. Function arguments of type `graph` use *call by reference* semantics. This means that a command such as `add(g,2,4)` can modify the graph `g` (in this case, by adding an edge joining vertices `2` and `4`).

  Not all MATGRAPH functions modify their arguments, but they all use call by reference for `graph` arguments. For functions that are capable of modifying graphs, the graph that is modified is always the *first*

Date: 2008:03:04:09:47.

[1] Simple graphs are undirected graphs without loops or multiple edges.

argument to the function. For example, MATGRAPH provides a `line_graph` function. The command

```
line_graph(g,h)
```

overwrites `g` with the line graph of `h`. The graph `h` is not affected. For more detail, see §2.3.

- As a consequence of how graphs are stored in `matgraph`, the only time a `graph` variable should appear on the left hand side of an assignment statement is when the variable is initialized like this:

  ```
  g = graph
  ```

  At no other time should a `graph` variable appear to the left of an equal sign.

2.2. **Starting MATGRAPH.** Launch MATLAB and be sure that the MATGRAPH directory is visible on MATLAB's path. This can be done with MATLAB's `addpath` function. For example:

```
addpath('/home/betty/programming/matgraph/')
```

assuming Betty placed the `matgraph` directory inside a folder named `programming` on her computer.

The next step is to initialize the MATGRAPH system. This is done by giving the following command:

```
graph_init
```

This sets up hidden data structures (see §6) used by MATGRAPH. MATLAB responds:

```
Graph system initialized. Number of slots = 500.
```

MATGRAPH is now ready to work with graphs. By default, the system can handle 500 different graphs. This should be adequate for most purposes. However, if you need an array of, say, 1000 graphs, then this is not sufficient. Alternatively, MATGRAPH may be initialized with an explicit argument specifying the number of "slots" (place holders for graphs) like this:

```
graph_init(20000)
```

If you ever wish to delete all of the hidden data structures (and thereby erasing all graphs held therein), use the function `graph_destroy`.

See also[2] the `free_all`, `num_available`, and `max_available` functions.

2.3. **Declaring `graph` objects and memory management.** Before working with graphs, it is necessary to declare variable(s) to be of type *graph*. This runs against MATLAB's philosophy that variables do not need to be declared, but is necessary for the sake of efficiency.

To declare a variable, say `g`, to be of type *graph*, we give the following command:

```
g = graph;
```

It is helpful to read this as "let *g* be a graph." **This is the only circumstance in which a graph object may appear to the left of an equal sign.**

Behind the scenes, one of the "slots" allocated for graphs (by `graph_init`) is set aside for the graph `g`.

Every time a `graph` variable is declared, one these slots is taken to hold the data for the graph. If a `graph` variable is no longer needed, then its slot can be released like this:

```
free(g)
```

If, inadvertently, a `graph` variable is cleared from MATLAB's workspace, there is no convenient way to free the slot it occupied.

Freeing a graph's slot is not generally necessary when working at MATLAB's command prompt. Graphs can be modified repeatedly and one need not declare more graph variables than the number of graphs one is currently considering.

However, releasing the slot held by a graph is *vital* in `.m` files. It is often useful for a function to declare a `graph` variable temporarily. **Every graph that is created using the `g=graph` constructor must be released by a matching call to `free(g)`.** Otherwise, each time the function is invoked, another slot in MATGRAPH's hidden data structure is consumed until no slots remain available.

To make a copy of a graph, we must *not* use the statement `h=g`. Rather, use `copy(h,g)`.

2.4. **Basic graph operations.** The statement `g=graph` creates a new graph with no vertices or edges. It is now possible to add and to delete vertices and edges using the `resize`, `add`, and `delete` functions. For each of these, the graph to be modified is the first argument (see the discussion of basic principles in §2.1).

- `resize(g,n)` changes the number of vertices in `g` to the value held in `n`. If `n` is greater than the number of vertices in `g`, then additional, isolated vertices are added. On the other hand, if `n` is less than the number of vertices in `g`, then the highest numbered vertices are deleted leaving a graph with `n` vertices. For example, if `g` has 10 vertices and we invoke the command `resize(g,7)`, then vertices 8, 9, and 10 are deleted (and all edges incident on these vertices are also deleted).
- `add(g,u,v)` adds an edge between `u` and `v` to the graph. If either of these values is nonpositive, or if they are equal, nothing happens. If either `u` or `v` is greater than the number of vertices currently in `g`, the graph is expanded to have $\max\{$`u`,`v`$\}$ vertices.

  Another way to use `add` is like this: `add(g,elist)` where `elist` is a $k \times 2$ array of positive integers. Each row of `elist` is considered an edge, and all of these edges are added to the graph. If any end point of any edge is larger than the number of vertices currently in the graph, the graph is resized to accommodate.

  The automatic resizing has the potential side effect of adding isolated vertices. For example, if `g` has 5 vertices, and we then give the command `add(g,3,7)` the graph is resized to have 7 vertices. Vertex 6 is added as an isolated vertex.

[2] See §3 for a description of the web-based information on all MATGRAPH functions. When we direct the reader to see a particular function, we typically mean to view that function's documentation with a web browser or by using MATLAB's `help` command.

Remember: The vertex set of all graphs in MATGRAPH are always of the form $\{1,2,\ldots,n\}$ where $n \geq 0$.

- `delete` is used to delete vertices or edges from a graph.
  - `delete(g,v)` deletes the vertex `v` from the graph. (If `v` is not a vertex of the graph, there is no effect.)
    All vertices with number greater than `v` have their values decreased by 1. For example, suppose `g` is a path graph with edges $1 \sim 2 \sim 3 \sim 4 \sim 5$. When we give the command `delete(g,3)` vertex 3 is deleted from the graph, and vertices 4 and 5 get renamed 3 and 4, respectively. Thus, after `delete(g,3)` the vertex set of `g` is $\{1,2,3,4\}$ and the only edges are $1 \sim 2$ and $3 \sim 4$.
  - `delete(g,vlist)` deletes an entire set of vertices. In this form, `vlist` is a $k \times 1$ array of positive integers. All vertices in `vlist` are deleted from the graph, and then vertices are renamed so the vertex set remains of the form $\{1,2,\ldots,n\}$.
    For example, if `g` is a cycle on 5 vertices with edges $1 \sim 2 \sim 3 \sim 4 \sim 5 \sim 1$, then `delete(g,[2;3])` deletes vertices 2 and 3 from the graph (leaving edges $4 \sim 5 \sim 1$) and then renumbers vertices 4 and 5 with the new names 2 and 3, so the final result is the path $2 \sim 3 \sim 1$.
  - `delete(g,u,v)` deletes the edge between `u` and `v` from the graph. If this edge is not present in the graph, nothing happens.
  - `delete(g,elist)` deletes a list of edges from the graph. The variable `elist` must be a $k \times 2$ array of positive integers.

  Point to notice: `delete(g,[3;4])` deletes vertices 3 and 4 from the graph (second argument is a column vector) whereas `delete(g,[3,4])` deletes the edge $3 \sim 4$ from the graph (second argument is a $k \times 2$ array where k happens to equal 1).

  See also the `clear_edges` function that deletes all edges from a graph.

See also `set_matrix` (described in §2.8). See also `contract`.

2.5. **Standard graphs.** MATGRAPH provides several functions for forming standard graphs. One of the more versatile is the `complete` function for creating complete graphs, complete bipartite graphs, and complete multipartite graphs:

- `complete(g)` adds all possible edges to `g` without changing its vertex set.
- `complete(g,n)` sets `g` to be the complete graph K_n.
- `complete(g,n,m)` sets `g` to be the complete bipartite graph $K_{n,m}$.
- `complete(g,list)` sets `g` to be the complete multipartite graph $K(a_1,a_2,\ldots,a_t)$ where the indices are the entries in `list`.

Other general graph builders include `path`, `cycle`, `grid`, `wheel`, `cube`, `circulant`, and `paley`. Specific graphs can be formed using these: `bucky`, `dodecahedron`, `icosahedron`, `octahedron`, and `petersen`. Various random graphs can be built using these functions: `random`, `sprandom`, `random_bipartite`, and `random_regular`.

2.6. **Graph operations.** MATGRAPH provides a variety of operations to form new graphs from old. For example, `line_graph(h,g)` sets `h` to be the line graph[3] of `g`. Other operations include `cartesian`, `complement`, `mycielski`, `induce`, `intersect`, `union`, and `trim`.

Breadth-first and depth-first spanning trees can be found using `bfstree` and `dfstree`. See also `nsptrees`.

2.7. **Graph inspectors.** The functions `nv(g)` and `ne(g)` give the number of vertices and edges of `g`. These two values are returned by `size(g)` in a 1×2 array.

There are two ways to see if an edge is present in a graph `g`. The command `has(g,u,v)` returns 1 (for true) if the edge between `u` and `v` is present in `g`, and 0 otherwise. Alternatively, the same result is produced by `g(u,v)`.

The neighborhood of a vertex is returned by the command `neighbors(g,v)`; the result is a list (one-dimensional array) of the vertices adjacent to `v`. The same result is returned by `g(v)`.

The degree of a vertex is given by `deg(g,v)`. With only a single argument, `deg(g)` returns the degree sequence of the graph.

`find_path(g,u,v)` finds a shortest path from `u` to `v` (returned as a list of vertices on the path); if no such path exists, an empty array is returned.

`isconnected(g)` returns 1 (true) if `g` is connected and 0 (false) otherwise.

The components of a graph can be found using `components(g)`. This returns a partition object (see §5) each of whose blocks is the vertex set of a component of `g`.

The distance between vertices can be found with `dist(g,u,v)`. The form `dist(g,u)` returns an array giving the distances from `u` to all the vertices in the graph. Calling `dist(g)` returns a square matrix giving the distances between all pairs of vertices. See `diam`.

2.8. **Graph–matrix conversions.** The adjacency matrix of a graph is returned by `matrix(g)`. This returns a square *logical* matrix. To use this matrix arithmetically, convert it to class `double`; for example, the following command returns the eigenvalues of (the adjacency matrix of) a graph:

```
eig(double(matrix(g)))
```

Conversely, given a square, symmetric, zero-one, zero-diagonal matrix `A`, we can set `g` to have this matrix as its adjacency matrix like this: `set_matrix(g,A)`.

See also `spy`, `laplacian`, and `incidence_matrix`.

[3]The line graph of G is a graph $L(G)$ whose vertex set is $E(G)$. Two vertices e_1 and e_2 of $L(G)$ are adjacent in $L(G)$ provided, when considered as edges of G, they are incident with a common vertex.

2.9. **Graph invariants and partitions.** In addition to basic information functions described in §2.7, MATGRAPH can calculate other invariants and features of graphs of interest to graph theorists. These include `alpha` (independence number), `omega` (clique number), and `dom` (domination number). (All three of these use the integer programming facilities in MATLAB's Optimization Toolbox.) See also `diam`.

The `bipartition` function determines whether a graph is bipartite; if it is, it returns the bipartition as a `partition` object (see §5). Otherwise (the graph is not bipartite) `bipartition` returns an empty partition.

A graph coloring algorithm is available in the `color` function. This returns a partition of the vertex set of a graph into independent sets by a greedy coloring algorithm (step through the vertices in decreasing degree order and give the first available color to each vertex in turn). Alas, this generally does not find a coloring with $\chi(G)$ colors.

The chromatic polynomial of a graph can be found for small graphs. For example:

```
>> g = graph;
>> cycle(g,5)
>> chromatic_poly(g)
ans =
     1    -5    10   -10     4     0
```

shows that the chromatic polynomial of C_5 is

$$x^5 - 5x^4 + 10x^3 - 10x^2 + 4x.$$

2.10. **Input-output.** MATGRAPH can read and write graphs in files on the user's hard disk.

Suppose the user wishes to build a graph using some other software (such as a C program written by the user) and then read that graph into MATGRAPH. To do this, the data should be saved in a file as a list of edges. Each line of the file should contain exactly two integers separated by white space. These integers should range from 1 to the number of vertices in the graph. Let's say that this data is saved in a file named `mygraph`.

The MATLAB command `load mygraph` reads the file `mygraph` and saves the contents of that file in a variable that is also named `mygraph`. (MATLAB's current working directory must be the same as the directory that contains the file `mygraph`.)

The variable `mygraph` is an $m \times 2$ array of edges. This can be converted into a graph like this:

```
g = graph(mygraph)
```

or if the graph `g` already exists, like this:

```
resize(g,0)
add(g,mygraph)
```

(The `resize(g,0)` clears all data from `g`.)

MATGRAPH also provides its own `save` and `load` commands.

- `save(g,filename)` saves the graph `g` to the user's hard drive in a file named in `filename`. (For example, `save(g,'mygraph')`.) This saves all the information about the graph and not just a list of edges.
- `load(g,filename)` reads the graph data in the file named in `filename` and sets `g` to be that graph. The file named in `filename` must be one created by MATGRAPH's `save` command and *not* just a list of edges.

MATGRAPH provides a function named `sgf` which stands for **s**imple **g**raph **f**ormat. This function converts graphs to and from a 2-column matrix whose rows have the following meanings:

- The first row is $[n\ m]$ where n is the number of vertices and m is the number of edges in the graph.
- The next m rows give the edges of the graph; each row is of the form $[u\ v]$ where $1 \le u \ne v \le n$.
- Optionally, an additional n rows give the locations of the vertices. Row number $m+1+i$ is $[x_i\ y_i]$ and specifies the coordinates of vertex i.

Matrices of this form are easy to read or to write on disk, and this format is easy for other programs to produce.

In addition, it is possible to create `.m` files for saving graphs to be used by other programs. For example, MATGRAPH's `dot` command writes graphs to disk in a format that can be processed by GraphViz's `dot` program. See also `graffle` and `nauty`.

2.11. **Handling large graphs.** Behind the scenes, graphs in MATGRAPH are saved as square matrices. A graph with, say, ten thousand vertices would occupy an array with 100 million entries. MATLAB provides the ability to handle matrices large matrices with few nonzero entries efficiently. This ability is embedded into MATGRAPH.

Small graphs are best handled using full storage. By default, a graph created by `g = graph` uses full storage. It is easy, however, to convert a graph to use sparse storage.

- `sparse(g)` converts a graph's storage method to be sparse. This is useful for extremely large graphs with relatively few edges (i.e., small average degree).
- `full(g)` converts a graph's storage to full mode. This is the preferred method for small graphs and graphs with many edges.

One can check the type of storage in use with the `issparse` and `isfull` functions.

The graph constructor `g = graph` takes an optional argument; one can write `g = graph(n)` where `n` is a nonnegative integer. This creates a new graph with `n` vertices. If `n` is small, full storage is used for `g`. If `n` is large, sparse storage is automatically provided. How large is "large"? See `set_large`.

2.12. **Labels.** Naming vertices as consecutive integers from 1 to n can be inconvenient. MATGRAPH provides a means to assign text labels to vertices. The command `label` can be used to assign such labels to vertices: `label(g,v,string)` assigns the characters in `string` to be a label for vertex `v`. If a vertex of a graph is deleted, all higher numbered vertices are renumbered, but their labels are retained. The command `get_label` is used to learn the label of an individual vertex or to return a list of all labels on all vertices in a graph. See also `clear_labels`.

2.13. **Visualization.** It is often useful to be able to see pictures of graphs. MATGRAPH provides a basic means to do this.

One may associate an embedding with a graph; this is a mapping from the vertex set to points in the plane.

The command `draw` draws a picture of the graph in the plane by placing each vertex at its x,y-coordinates and joining adjacent vertices by line segments. See also `ndraw` and `ldraw`. Note that `draw` simply draws the graph in the current figure window without erasing the contents of that figure; to see only the graph, first give the MATLAB command `clf`. See also the functions `ndraw` and `cdraw`.

When `draw` is invoked for graphs without embeddings, a default, circular embedding is automatically constructed. Some graph building operations attach an embedding to the graphs they form; for example, `petersen(g)` sets `g` to be the Petersen graph with vertices located at classic coordinates.

It is possible to set a graph's embedding to coordinates of your choosing with the `embed` command. If `g` has n vertices and `xy` is an $n \times 2$ matrix of real numbers, then `embed(g,xy)` sets the coordinates of the vertices to the corresponding rows of `xy`. The command `randxy(g)` sets the coordinates of the vertices to random locations.

To learn the current embedding of a graph, use `getxy(g)`. To erase a graph's embedding, use `rmxy(g)`. See also `hasxy`.

The best MATGRAPH method to generate an embedding is `distxy`. This function uses MATLAB's Optimization Toolbox. It works reasonably well on small graphs. It attempts to place vertices in the plane so that the Euclidean distance matches their graph-theoretic distance, but the penalty is lessened the further apart the nodes.

We also provide `mdsxy` which creates an embedding based on multidimensional scaling. It's fast, but produces mediocre results.

Readers are warmly encouraged to submit other embedding algorithms for incorporation into MATGRAPH.

3. DOCUMENTATION

This introduction to MATGRAPH does not list every function available to the user. However, all functions (in `.m` files) are documented in the accompanying web pages. These web pages are housed in the `html` subdirectory of the main `matgraph` folder. Double clicking on the file `index.html` opens the main documentation page in a web browser. From here, all the `.m` files can be found including descriptions, cross references, and source code. This includes the supporting classes `partition` and `permutation`.

In addition, the user may get help on any MATGRAPH command with MATLAB's usual `help` command. Some command names are overloaded. For example, the name `delete` is both a built-in MATLAB command and a MATGRAPH function. Type `help graph/delete` to access the MATGRAPH version.

For a list of all methods available for `graph` objects, type `methods graph`.

See also *Matgraph By Example* in the `doc` directory.

The web pages in the `html` directory were generated by by the `m2html` package created by Guillaume Flandin; this utility can be found on the MathWorks' web site.

4. THE PERMUTATION CLASS

Included in this toolbox is a class called `permutation`. These objects represent permutations of the integers $\{1,2,\ldots,n\}$. Unlike `graph`s, these object behave according to the usual MATLAB conventions and do not need to be specially declared. `permutation` objects are used by MATGRAPH's `renumber` command.

The standard way to build a permutation p is to specify its action with a vector of the form $[a_1,a_2,\ldots,a_n]$ where $a_i = p(i)$. For example:

```
>> p = permutation( [2 1 3 5 6 4] )
(1,2)(3)(4,5,6)
```

This assigns to p a permutation in which $p(1) = 2$, $p(2) = 1$, $p(3) = 3$, $p(4) = 5$, $p(5) = 6$, and $p(6) = 4$. Note that p is displayed on the console using the standard disjoint cycle notation.

The basic operations of applying a permutation to an element and composition of permutations are implemented. Typing `p(k)` returns the action of the permutation p on the element k. If k is not in scope (outside the range 1 to n), then 0 is returned. Composition of permutations is denoted by multiplication, `*`. Repeated composition can be achieved using the `^` operator: `p^3` is equivalent to `p*p*p`. The power may be zero or negative.

Equality and inequality of permutations can be checked with `==` and `~=`, respectively.

Here are the functions defined for the class `permutation`. Their `.m` files are found in the `@permutation` folder.

- `array`: convert a permutation to an array. The syntax is `array(p)`. This returns a $1 \times n$ array whose i^{th} entry if $p(i)$. For example, if $p = (1,2)(3)(4,5,6)$, then `array(p)` returns the array $[2,1,3,5,6,4]$.
- `cycles`: determine the cycle structure of a permutation. The syntax is `cycles(p)`. This returns a cell array. Each member of the cell array contains the elements (in order) of a cycle of p. For example, if $p = (1,2)(3)(4,5,6)$, then `c=cycles(p)` sets `c{1}` to $[1,2]$, `c{2}` to $[3]$, and `c{3}` to $[4,5,6]$.
- `inv`: permutation inverse. The syntax is `inv(p)`. This returns the inverse permutation, p^{-1}.

- `length`: number of elements permuted. The syntax is `length(p)`. For example, if $p = (1,2)(3)(4,5,6)$, then `length(p)` is 6.
- `matrix`: return a permutation matrix that represents the permutation. The syntax is `matrix(p)`. For example, if $p = (1,2)(3)(4,5,6)$, then `matrix(p)` gives this:

```
0     1     0     0     0     0
1     0     0     0     0     0
0     0     1     0     0     0
0     0     0     0     0     1
0     0     0     1     0     0
0     0     0     0     1     0
```

- `permutation`: class constructor. This can be called two ways. If n is a positive integer, `permutation(n)` returns the identity permutation on $\{1,2,\ldots,n\}$. If x is an array containing the elements 1 through n, then `permutation(x)` creates the permutation specified by those elements. For example, if x is `[2 1 3 5 6 4]`, then `permutation(x)` gives the permutation $(1,2)(3)(4,5,6)$.
- `random`: shuffle a permutation. The syntax is `random(p)`. This returns a permutation on the same elements as p but in a random order. Typically, to generate a random permutation on n elements, one would type this: `random(permutation(n))`.
- `sign`: sign (parity) of a permutation. The syntax is `sign(p)`. This returns 1 is p is an even permutation and -1 is p is an odd permutation.
- `size`: give the number of elements and number of cycles in a permutation. The syntax is `size(p)`. This returns a two-element array. The first element is the number of objects permuted by the permutation and the second element is the number of cycles in the disjoint-cycle representation of p.

  MATLAB uses this when reporting on permutation variables in the workspace. The permutation $(1,2)(3)(4,5,6)$ is described as a `<6x3 permutation>`. The 6 refers to the fact that this is a permutation of the set $\{1,2,\ldots,6\}$ and the 3 refers to the fact that this permutation has 3 cycles.

5. The Partition Class

The `partition` class represents partitions of sets of the form $\{1,2,\ldots,n\}$. A partition can be created with the command `p=partition(n)`. This creates a partition in which all parts have size 1; that is, the partition $\big\{\{1\},\{2\},\ldots,\{n\}\big\}$. A partition can also be created from a cell array. Each cell in the array should list the elements of a block; the integers 1 through n should appear exactly once in each member of the cell array. For example, if we type

```
c = {[1 2 4],[3 5 6],[7:10]};
p = partition(c)
```

Then p is the partition $\big\{\{1,2,4\},\{3,5,6\},\{7,8,9,10\}\big\}$ and MATLAB types this:

```
{ {1,2,4} {3,5,6} {7,8,9,10} }
```

If p is a partition and k is an integer, the expression `p(k)` returns the elements in the same part as k. For the partition presented above, `p(2)` would return the array `[1 2 4]`. For integers j and k, the expression `p(j,k)` returns true if j and k are in the same part of p and false otherwise.

The meet and join of two partitions is computed using `p*q` and `p+q`, respectively. Equality and inequality can be checked with == and ~=.

Partitions can be converted into cell arrays with the `parts` function.

Here is a list of the various functions defined for the `partition` class; these can be found in the `@partition` folder.

- `merge`: combine two parts. The syntax is `merge(p,j,k)`. This returns a new partition in which the parts containing j and k have been merged into a single part.
- `np`: number of parts. The syntax is `np(p)`; the number of parts in the partition is returned.
- `nv`: size of the ground set. The syntax is `nv(p)`; the number of elements in the ground set of the partition is returned.
- `partition`: constructor for this type. The simple syntax is `partition(n)` (where n is a positive integer). This builds a partition with ground set $\{1,2,\ldots,n\}$ in which there are n parts (all of size one).

  Alternatively, if c is a cell array, then `partition(c)` creates a partition based on the arrays in c. Each of `c{1}`, `c{2}`, and so on, is a list of integers. Together, these lists should contain each of the integers in $[n]$ exactly once.
- `parts`: get the parts of the partition. The syntax is `parts(p)`. This returns a cell array. Each cell contains a list (vector) of positive integers in one of the parts of p.
- `size`: report the number of elements in the ground set and the number of parts. The syntax is `size(p)`. This returns a 1×2 array `[n m]` where n is the size of the ground set and m is the number of blocks.

  This is used by MATLAB when it reports the variables in a workspace. A partition is reported like this: `<10x3 partition>`. This means the partition's ground set is $[10]$ and there are three parts in the partition.

6. Under the Hood (Stuff You Don't Need to Know)

All data about graphs are held in a hidden global data structure named `GRAPH_MAGIC`. Objects of type `graph` are simply indices into this structure. This enable us to simulate call-by-reference semantics for *graph* objects; that is, MATLAB functions can modify *graph* arguments.

It is possible to save the entire `GRAPH_MAGIC` structure into another variable; this would allow multiple "graph theory universes" to coexist. It's not clear this is needed.

6.1. The `GRAPH_MAGIC` structure.

The global data structure is named `GRAPH_MAGIC`. To access this structure directly, use the following line in your `.m` files:

```
global GRAPH_MAGIC
```

The `GRAPH_MAGIC` structure contains the following fields:

- `ngraphs`: the number of "slots" available in this structure (equal to the size of the arrays `GRAPH_MAGIC.graphs` and `GRAPH_MAGIC.in_use`).
- `graphs`: this is a cell array containing the graphs. (See §6.1.1.)
- `in_use`: an array that indicates which slots are taken. A 1 in position *i* of this array signals that slot *i* is taken; a 0 means the slot is available to hold a new graph.
- `Q`: a structure implementing a double-ended queue. (See §6.1.2.)
- `large_size`: a variable holding the cutoff between "large" and "small" graphs. If the graph constructor `graph` is fed a large argument, it creates a sparse graph.

6.1.1. *Inside `GRAPH_MAGIC.graphs`.* The cell array `GRAPH_MAGIC.graphs` holds the graphs. Each cell in this array is a structure with two fields: `array` and `xy`. The `array` field holds the adjacency matrix of the graph (a zero-one, symmetric matrix). The `xy` field holds the embedding for the graph; this is an $n \times 2$ array of real values giving the coordinates of the vertices.

6.1.2. *The private double-ended queue.* The `Q` field of `GRAPH_MAGIC` is a double-ended queue available for use by `graph` algorithms (e.g., `bfstree`). It is a structure that contains three fields:

- `array`: a one-dimensional array that holds the stack/queue values.
- `first`: an index pointing to the first (front most) element of the queue.
- `last`: an index pointing to the last (back most) element of the queue.

There is a small suite of tools for working with the queue in the directory `@graph/private`. These are visible to functions inside the `@graph` directory, but not generally available. Here they are (in alphabetical order):

- `q_capacity`: gives the maximum capacity of the queue.
- `q_get`: returns a list of the elements in the queue (that is, `array(first:last)`).
- `q_init`: called with one argument, this initializes the queue with a given capacity.
- `q_pop_back`: pops off (and returns) the last element of the queue. This is a stack-like operation.
- `q_pop_front`: pops off (and returns) the front most element in the queue. This is a queue-like operation.
- `q_push`: called with one argument, this adds an element to the back of the queue.
- `q_size`: returns the number of elements in the queue.

6.2. The `graph` type.

The *graph* type is simply a "wrapper" for an integer; that integer is an index into the `GRAPH_MAGIC.graphs` array. A *graph* object contains just one field, `idx`, which holds that integer.

In many *graph* functions (in the `@graph` directory) we see the following:

```
GRAPH_MAGIC.graphs{g.idx}.array
```

This is how we refer to the adjacency matrix of the graph `g`.

7. Future Projects

There are many additions I would like for this project. Here are few:

- Planarity testing and embedding. I would like an `is_planar` method to test if a graph is planar and a good `planar_embed` method for finding a crossing-free embedding.
- The `distxy` layout routine works reasonably well, but is hardly fast. It also relies on the Optimization Toolbox. I'd like a better layout engine.
- A GUI for creating and editing graphs. I'd like to see this invoked with a command such as `gui(g)` or `graph_edit(g)`.
- We need `.m` files for connectivity, edge connectivity, maximum matching (in general graphs), approximate isomorphism, better heuristic coloring algorithms, girth, and so forth.

Department of Applied Mathematics and Statistics, The Johns Hopkins University, Baltimore, Maryland 21218-2682 USA
E-mail address: ers@jhu.edu