MATGRAPH BY EXAMPLE

EDWARD SCHEINERMAN

This document illustrates the use of MATGRAPH through the use of specific examples. Some of the concepts (such as the notion of *declaring* `graph` objects) are explained in the accompanying users' guide *Matgraph: A MATLAB Toolbox for Graph Theory* that you should read in conjunction with this document. A description of all the MATGRAPH functions can be found in the accompanying web pages in the `html` directory. We assume that you have a reasonable command of MATLAB.

1. GETTING STARTED

1.1. **Download MATGRAPH.** To use MATGRAPH, you need to download the MATGRAPH compressed tar archive file from the website

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

Click on the the words "clicking here" in the paragraph that begins "You can download Matgraph by clicking here." This places a file named `matgraph-X.Y.tgz` on your computer (where `X.Y` is the version number). Double clicking this file or issuing the Unix command

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

(replace `X.Y`) should extract a directory (folder) named `matgraph` that you can place anywhere you wish on your computer.

1.2. **Design Principles.** MATGRAPH is designed to make interactive graph theory computation simple by building on the power of MATLAB. Before we begin in earnest, there are important principles behind the design of MATGRAPH that you must understand.

(1) All graphs in MATGRAPH are simple and undirected; there are no loops, multiple edges, or directed edges.

(2) The vertex set of graphs in MATGRAPH is always of the form $\{1,2,\ldots,n\}$ for some integer n. One implication of this principle is that when a vertex is deleted, all vertices with larger index are renumbered. (It is possible to attach a label to a vertex that is distinct from its vertex number).

(3) Graph variable must be declared prior to use (see §1.3). If a graph variable is declared within a `.m` function file, then it must be "released" before the function exits. Declaration and release are accomplished with the commands

```
g = graph;
```

and

```
free(g)
```

(4) MATGRAPH functions are capable of changing their arguments. For example, the command `delete(g,1,2)` deletes the edge $\{1,2\}$ from the graph `g`; the variable `g` is modified by this command. (This is unusual for MATLAB.) If a MATGRAPH function takes two

Date: 2006:10:11:10:00.

(or more) graph arguments then only the first argument to the function might be changed; all subsequent arguments are left unmodified.

1.3. **A first session.** Launch MATLAB and issue a command that looks like this:

```
>> addpath /home/ralph/Programming/matgraph/
```

This tells MATLAB where to find the MATGRAPH toolbox. Of course, replace the pathname with the location of the `matgraph` folder that you downloaded as described in §1.1.

For the rest of this document, we tacitly assume that you have given this command before using MATGRAPH. If you like, you may add this command to your `startup.m` file (see the MATLAB documentation for more detail).

Next, we *declare* a graph variable `g`:

```
>> g = graph
Graph system initialized. Number of slots = 500.
Graph with 0 vertices and 0 edges (full)
```

Unlike most MATLAB variables, graph variables *must* be declared; see the users' guide for more detail.

Next set `g` to be the Petersen graph:

```
>> petersen(g)
>> g
Graph with 10 vertices and 15 edges (full)
```

The command `petersen(g)` overwrites `g` with the Petersen graph.

Now we draw the graph in a figure window:

```
>> ndraw(g)
```

The command `ndraw` draws the graph and writes each vertex's number inside its circle. See Figure 1. Note that the embedding of the graph is imparted to `g` by the command `petersen`.

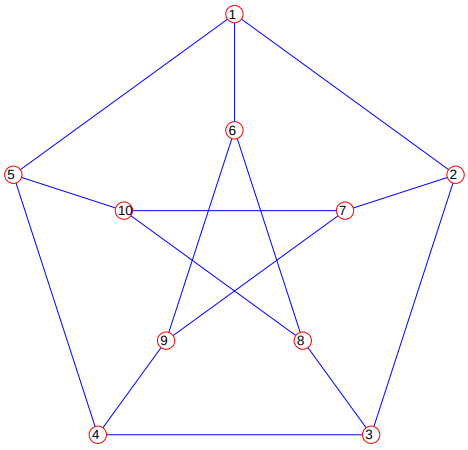

FIGURE 1. Petersen's graph.

In addition to `ndraw`, MATGRAPH provides the following variations: `draw` (draw the graph with vertices drawn as hollow circles), `ldraw` (draw the graph with the vertices inscribed with their

labels—different from their vertex numbers), and `cdraw` (draw the graph with colored vertices). Type, for example, `help cdraw` for more information.

It is well known that Petersen's graph is not Hamiltonian. To verify this type:

```
>> hamiltonian_cycle(g)
ans =
     []
```

The empty matrix indicates that no Hamiltonian cycle was found. However, if we delete any vertex from `g`, the graph is Hamiltonian:

```
>> delete(g,1)
>> hamiltonian_cycle(g)
ans =
     1
     2
     3
     4
     9
     7
     5
     8
     6
```

The command `delete(g,1)` deletes vertex number 1 from the graph.

Notice that the `delete` command changes the graph. By a bit of fancy footwork, MATGRAPH functions are able to modify arguments of type `graph`—this is counter to the usual MATLAB call-by-value semantics. Many of the MATLAB commands modify their graph arguments, but the following convention is observed: If a command is capable of modifying a graph, *only the first argument to the command can by modified*.

Notice that the vertices have been renumbered. The output of `hamiltonian_cycle` reports that

$$1 \to 2 \to 3 \to 4 \to 9 \to 7 \to 5 \to 8 \to 6 \to 1$$

is a Hamiltonian cycle in `g`. At first this may be confusing since we had previous deleted vertex 1 from the graph. MATGRAPH follows the convention that the vertex set *always* consists of consecutive integers beginning with 1. When a vertex is deleted from a graph, all vertices with higher numbers are renumbered accordingly. To see this, type this:

```
>> clf
>> ndraw(g)
```

The result is shown in Figure 2.

Notice that we issued the MATLAB command `clf` before drawing `g`. The `clf` command clears the current figure window. This is necessary because MATGRAPH's drawing commands draw their graphs on top of whatever is already in the figure window (without erasing the figure window first).

If we are done using the graph variable `g`, we should *not* simply give the usual MATLAB command `clear g`. Rather, we do this:

```
>> free(g)
>> clear g
```

The command `free(g)` releases the graph `g`'s "slot" in a hidden data structure. When MATGRAPH starts up (with the first `g=graph` command or by an explicit invocation of `graph_init`)

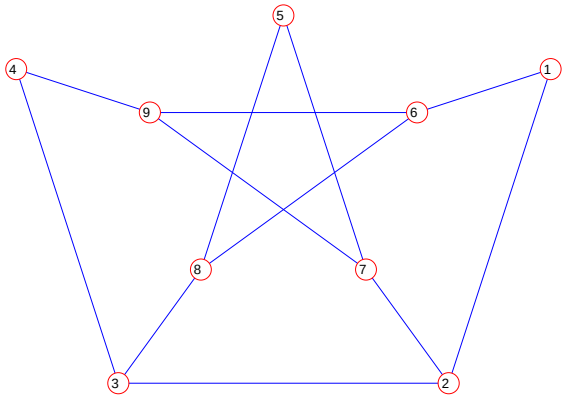

FIGURE 2. Petersen's graph with a vertex deleted is Hamiltonian.

a specific number of slots are allocated for graphs (at this writing, 500 slots). Each time a graph variable is declared (by typing `g=graph`), one of these slots is used; the command `free(g)` releases the slot held by the graph. See the users' guide for more detail. To wipe out the entire hidden data structure (and all the graphs contained therein) you can use `graph_destroy`.

One more important point. The typical behavior of MATLAB's assignment operator is to make a copy. So if `A` is a matrix, `B=A` sets `B` to be an independent copy of `A`. Changes to `B` do not affect `A`. However, MATGRAPH graph objects behave differently. If `g` is a graph, then the command `h=g` does *not* make a separate copy of `g`, and any modification to `h` also modifies `g`. It is nearly certain this is not the behavior you desire. Instead, do this:

```
>> g = graph
Graph with 0 vertices and 0 edges (full)
>> petersen(g)
>> h = graph
Graph with 0 vertices and 0 edges (full)
>> copy(h,g)
>> delete(h,1,2)
>> h
Graph with 10 vertices and 14 edges (full)
>> g
Graph with 10 vertices and 15 edges (full)
```

The `copy(h,g)` overwrites `h` with an independent copy of `g`.

2. BASICS

2.1. **A path.**

One of the simplest graphs is a path on n vertices, P_n. Here we create such a graph in MATGRAPH.

```
>> g = graph
Graph system initialized. Number of slots = 500.
Graph with 0 vertices and 0 edges (full)
```

```
>> for k=1:9, add(g,k,k+1), end
>> ndraw(g)
```

This creates the graph P_{10} and draws it in a figure window. Notice that the command `add(g,u,v)` adds the edge uv to the graph. The variables u and v must be distinct positive integers; otherwise the command has no effect. If vertices u and v are already in the graph, the edge uv is simply added. However, if a graph has n vertices and either u or v is greater than n, then the graph's vertex set is first expanded to $\max\{u,v\}$ vertices and then the edge is added. [Remember, the vertex set of a graph in MATGRAPH is *always* of the form $\{1,2,\ldots,n\}$.]

Notice that the drawing of `g` places the vertices around a circle. If a graph does not have an embedding (e.g., the `petersen` command imparts an embedding to its argument), then then the drawing commands (such as `ndraw`) give the graph a default embedding by placing the vertices around a circle.

Now here is a simpler way to create (and view) P_{10}:

```
>> path(g,10)
>> clf
>> ndraw(g)
```

The command `path(g,10)` overwrites `g` with a path on 10 vertices together with a sensible embedding—the vertices are arranged in a straight line.

2.2. **Adding and deleting.** Let's create the graph formed by deleting a perfect matching from K_{10}. Here are the commands:

```
>> complete(g,10)
>> for k=1:5, delete(g,k,k+5), end
>> clf
>> ndraw(g)
```

The command `complete(g,10)` overwrites `g` with K_{10}. (We assume that we are simply continuing from the previous section so the graph `g` has already been declared.) The `delete(g,u,v)` command deletes the edge uv from the graph (assuming it exists). This 3-argument version of `delete` does not remove any vertices from the graph.

To delete vertex u from a graph (and all its incident edges) give the command `delete(g,u)`. Type `help graph/delete` to see all the various ways `delete` can remove vertices and edges from a graph.

To delete all vertices (and hence, all edges) from a graph, type `resize(g,0)`. To delete all edges (but no vertices) from a graph, type `clear_edges(g)`.

Creating the graph formed from $K_{5,5}$ by deleting a perfect matching is similar:

```
>> complete(g,5,5)
>> for k=1:5, delete(g,k,k+5), end
>> clf
>> ndraw(g)
```

The command `complete(g,m,n)` overwrites `g` with the complete bipartite graph $K_{m,n}$. Type `help graph/complete` to see what else `complete` can do.

Now try this:

```
>> resize(g,0)
>> add(g,3,6)
>> clf
```

```
>> ndraw(g)
```

The command `resize(g,0)` converts `g` to an empty (vertexless) graph. `add(g,3,6)` asks to add an edge between vertices 3 and 6, but since these vertices are not (yet) in the graph, the vertex set of `g` is expanded to $\{1,2,3,4,5,6\}$ and then the edge is added. Consequently, there are four isolated vertices in `g` as the picture reveals.

Next we create the Möbius ladder on 12 vertices (a 12-cycle plus edges between diametrically opposite vertices).

```
>> cycle(g,12)
>> elist = [1:6;7:12]'
elist =
     1     7
     2     8
     3     9
     4    10
     5    11
     6    12
>> add(g,elist)
>> clf; ndraw(g)
```

`cycle(g,12)` overwrites `g` with C_{12}. Next, we prepare a 6×2 matrix `elist` that specifies the extra edges we plan to add to `g`. The line `elist = [1:6;7:12]'` is standard MATLAB to create this matrix. Then `add(g,elist)` adds all the edges in `elist` to `g`.

2.3. **Neighbors, degrees, etc.** Create a grid graph like this:

```
>> grid(g,3,4)
>> clf;ndraw(g)
>> nv(g)
ans =
    12
>> ne(g)
ans =
    17
```

The grid is drawn nicely as shown in Figure 3. Notice that `nv` and `ne` report the number of vertices and edges, respectively, of the graph. Also try `size(g)`, `disp(g)`, or simply typing `g` on a line by itself.

We can learn the degree of a vertex, or the entire degree sequence of the graph, like this:

```
>> deg(g,1)
ans =
     2
>> deg(g,2)
ans =
     3
>> deg(g)
ans =
     2     3     2     3     4     3     3     4     3     2     3     2
```

To learn the neighbors of a vertex, we have two choices:

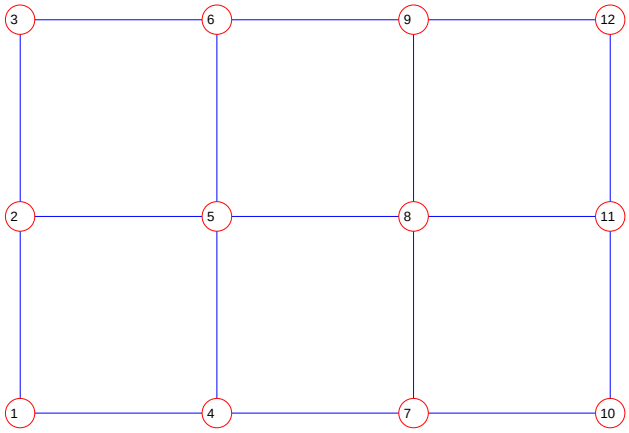

FIGURE 3. The 3×4 grid graph.

```
>> neighbors(g,2)
ans =
     1     3     5
>> g(2)
ans =
     1     3     5
```

[Note, the syntax `g(v)` does not seem to work inside `.m` file functions.]

To test if two vertices are adjacent we can use the `has` command or the syntax `g(u,v)` [which does not seem to work inside `.m` file functions].

```
>> has(g,1,2)
ans =
     1
>> has(g,1,5)
ans =
     0
>> g(1,2)
ans =
     1
>> g(1,5)
ans =
     0
```

We can verify that this graph is connected

```
>> isconnected(g)
ans =
     1
```

and find a shortest path between vertices 1 and 12:

```
>> find_path(g,1,12)
ans =
     1     2     3     6     9    12
```

Try `dist(g,1,12)` to see that the distance between these vertices is 5.

2.4. **Matrices.** To get the adjacency matrix of, say, the Petersen graph, do this:

```
>> petersen(g)
>> A = matrix(g)
A =
     0     1     0     0     1     1     0     0     0     0
     1     0     1     0     0     0     1     0     0     0
     0     1     0     1     0     0     0     1     0     0
     0     0     1     0     1     0     0     0     1     0
     1     0     0     1     0     0     0     0     0     1
     1     0     0     0     0     0     0     1     1     0
     0     1     0     0     0     0     0     0     1     1
     0     0     1     0     0     1     0     0     0     1
     0     0     0     1     0     1     1     0     0     0
     0     0     0     0     1     0     1     1     0     0
```

Next, we attempt to find the eigenvalues of this matrix, but run into trouble:

```
>> eig(A)
??? Function 'eig' is not defined for values of class 'logical'.
```

The problem is that MATGRAPH's `matrix` command returns a Boolean matrix (entries represent `true` and `false`), but it is simple to convert this to a numerical matrix and get the eigenvalues:

```
>> A = double(A);
>> eig(A)
ans =
   -2.0000
   -2.0000
   -2.0000
   -2.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    3.0000
```

Use `laplacian` to get the Laplacian matrix of a graph.

The command `spy(g)` is equivalent to `spy(matrix(g))`; this creates a square image with a dot in position i, j exactly when ij is an edge of g.

It is also possible to define a graph by specifying its adjacency matrix:

```
>> A = ones(6)-eye(6)
A =
     0     1     1     1     1     1
     1     0     1     1     1     1
```

```
     1     1     0     1     1     1
     1     1     1     0     1     1
     1     1     1     1     0     1
     1     1     1     1     1     0
>> set_matrix(g,A)
>> g
Graph with 6 vertices and 15 edges (full)
```

See also: `incidence_matrix`.

2.5. **Standard graph constructors.** MATGRAPH includes many functions for creating specific, standard graphs. We have encountered a few already: `path`, `cycle`, `complete`, `petersen`, and `grid`.

In addition to these, there are built-in methods for creating the Platonic solid graphs (for example, `dodecahdron`), wheels, Paley graphs, and so forth. See the on-line documentation (in the `matgraph/html` directory) for a complete list of all graph methods.

Graphs can also be built up from other graphs using graph operations; these are explored in §6.

Worthy of special mention are various methods to generate random graphs including `random`, `random_bipartite`, `random_regular`, and `random_tree`.

3. EMBEDDINGS

3.1. **Basics.** As we have seen, graphs created in `matgraph` can be drawn on the screen. A graph may have an embedding that is simply a specification of x,y-coordinates for all of the vertices. Edges are always drawn as line segments.

Some graph constructors (e.g., `petersen`) imbue their graphs with a prespecified embedding. However, if we start with a new graph and simply add vertices and edges, no embedding is created for the graph:

```
>> g = graph
Graph system initialized. Number of slots = 500.
Graph with 0 vertices and 0 edges (full)
>> for k=1:5, add(g,k,k+1), end
>> g
Graph with 6 vertices and 5 edges (full)
>> hasxy(g)
ans =
     0
```

This creates the path P_6 but no embedding is associated with the graph; this is observed with the `hasxy` command.

If we try to draw a graph that lacks an embedding, MATGRAPH gives the graph a default embedding in which the vertices are placed around a circle.

```
>> draw(g)
>> hasxy(g)
ans =
     1
```

We can specify the embedding for a graph by giving specific x,y-coordinates. Suppose we want to site the vertices of P_6 at $(1,0)$, $(2,0)$, …, $(6,0)$; we can do this:

```
>> xy = [ 1:6 ; zeros(1,6) ]'
xy =
     1     0
     2     0
     3     0
     4     0
     5     0
     6     0
>> embed(g,xy)
>> clf;draw(g)
```

If a graph possesses an embedding, `rmxy(g)` removes the embedding. To see the embedding of a graph, use `getxy`.

A random embedding can be given to a graph with `randxy(g)`. See also the `scale` function.

3.2. **Automatic graph layout.** MATGRAPH's default embedding—vertices uniformly around a circle—is usually unaesthetic and difficult to read. Fortunately, MATGRAPH provides a way to create embeddings automatically. Unfortunately, the one viable method we provide—`distxy`—is slow and requires[1] the Optimization Toolbox. Nevertheless, `distxy` gives reasonable results for moderately sized graphs. We invite readers who are expert in graph drawing algorithms to submit alternatives for inclusion in future releases of MATGRAPH.

The `distxy` embedding attempts to place vertices in a graph in the plane so that their graph theoretic distance equals the embedded vertices Euclidean distance. This is possible for path graphs, but otherwise is unattainable. Instead, we create a score function that measures how closely we achieve this goal and then use the services of the Optimization Toolbox to find a (local) minimum solution. Here is an example.

```
>> resize(g,0)
>> random_tree(g,10)
>> clf;draw(g)
```

This creates a random tree with 10 vertices and displays the tree in its default embedding. See the left portion of Figure 4. Now we compute an embedding using `distxy`.

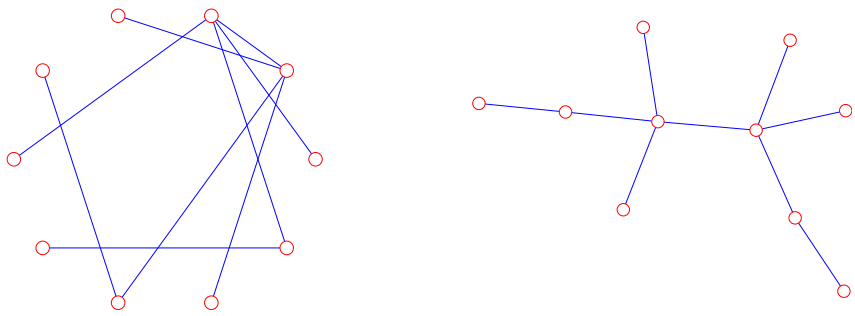

FIGURE 4. A random tree with its default embedding (left) and with a nice embedding found by `distxy` (right).

[1]If the Optimization Toolbox is not included with your version of MATLAB, it is available (for a fee) from The MathWorks.

```
>> distxy(g)
Optimization terminated: relative function value
 changing by less than OPTIONS.TolFun.
Embedding score = 2.7511
Elapsed time is 0.941816 seconds.
ans =
    2.7511
>> clf;draw(g)
```

The result is show in the right portion of Figure 4.

4. Helper Classes: Partitions and Permutations

MATGRAPH includes two classes that are useful for supporting work: partitions and permutations.

4.1. **Partitions.**

A *partition* is a set of pairwise disjoint, nonempty subsets of a set A whose union is A. In MATGRAPH, all partitions must be of a set of the form $[n] = \{1, 2, \ldots, n\}$. `partition` variables do not need to be declared (only `graph` objects require that special treatment).

Partitions are useful in graph theory. In MATGRAPH the functions to find the connected components of a graph or to find a coloring of a graph return `partition` objects.

There are a few ways to create a partition. The most basic is this:

```
>> p = partition(8)
{ {1} {2} {3} {4} {5} {6} {7} {8} }
```

The command `partition(n)` creates a default partition of $[n]$ in which each element is in a part by itself.

Alternatively, we can form a `partition` from a MATLAB cell array. Each cell in the cell array is a list (vector) of integers; taken together, these cells should contain all of the numbers from 1 to n (for some n) exactly once. Here is an example.

```
>> c = cell(3,1);
>> c{1} = [1 3 5];
>> c{2} = [4 6 7];
>> c{3} = [2 8];
>> p = partition(c)
{ {1,3,5} {2,8} {4,6,7} }
```

The statement `c = cell(3,1);` builds a 3×1 cell array (a fundamental MATLAB data structure). The next three lines populate the array with three lists of numbers. Finally, the statement `p = partition(c)` assigns to `p` a partition with the expected blocks.

The `merge` command is used to combine parts in a partition. Continuing with the example above, we type this:

```
>> merge(p,1,2)
{ {1,2,3,5,8} {4,6,7} }
>> p
{ {1,3,5} {2,8} {4,6,7} }
```

`merge(p,1,2)` forms a new partition in which the parts containing elements 1 and 2 are combined into a single part. Note that `merge` does not alter `p`; this is normal MATLAB behavior.

Now try this:

```
>> p(1)
ans =
     1     3     5
>> p(1,2)
ans =
     0
```

The command `p(v)` returns (as a list) the elements in `v`'s block. The command `p(v,w)` returns 1 (true) if `v` and `w` are in the same block, and returns 0 (false) otherwise.

There are other ways to extract the parts of a partition.

```
>> pts = parts(p);
>> pts{1}
ans =
     1     3     5
>> pts{2}
ans =
     2     8
>> pts{3}
ans =
     4     6     7
```

The function `parts` returns the parts of a partition as a cell array.

```
>> array(p)
ans =
     1     2     1     3     1     3     3     2
```

The `array` function returns an index number for each element; an element has index number i if it is in the i^{th} part of the partition.

The binary operators == and != can be used to test if two partitions are equal or unequal.

The binary operators + and $\star$ can be used to compute the join and meet of two partitions.

`nv(p)` returns the size of the ground set of the partition and `np(p)` returns the number of blocks in the partition. See also `size(p)`.

4.2. **Permutations.** A *permutation* is a bijection of a set A to itself. In MATGRAPH, the set A is always of the form $[n] = \{1, 2, \ldots, n\}$.

A new permutation is created with the `permutation` function:

```
>> permutation(9)
(1)(2)(3)(4)(5)(6)(7)(8)(9)
```

The `permutation(n)` command creates the identity permutation of $[n]$.

The `permutation` function can also be used to create a permutation from a list of numbers.

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

Here, the permutation is given by the matrix

$$\pi = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 5 & 2 & 4 & 6 \end{bmatrix}$$

The list `vec` gives the bottom row. This notation means that $\pi(1) = 1$, $\pi(2) = 3$, $\pi(3) = 5$, $\pi(4) = 2$, $\pi(5) = 4$, and $\pi(6) = 6$.

A random permutation can be created like this:

```
>> p = permutation(9);
>> p = random(p)
(1,7,6,2,4,9)(3,8)(5)
```

MATGRAPH defines $\star$ to denote permutation composition. The notation `p(j)` applies the permutation `p` to element `j`:

```
>> p(2)
ans =
     4
```

The inverse of a permutation can be calculated like this:

```
>> inv(p)
(1,9,4,2,6,7)(3,8)(5)
>> p^-1
(1,9,4,2,6,7)(3,8)(5)
```

In general, `p^m` is the *m*-fold composition of `p` with itself; `m` may be negative.

```
>> matrix(p)
ans =
     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     0     1     0
     0     1     0     0     0     0     0     0     0
     0     0     0     0     1     0     0     0     0
     0     0     0     0     0     0     1     0     0
     1     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0
     0     0     0     1     0     0     0     0     0
>> array(p)
ans =
     7     4     8     9     5     2     6     3     1
```

`matrix(p)` creates a permutation matrix and `array(p)` gives the lower row of the permutation when written in $2 \times n$-matrix notation.

```
>> c = cycles(p);
>> c{1}
ans =
     1     7     6     2     4     9
>> c{2}
ans =
     3     8
>> c{3}
ans =
     5
```

The `cycles` function creates a cell array containing the permutation's cycles.

5. Vertex Numbers and Labels

MATGRAPH rigidly enforces the rule that the vertex set of any graph must be of the form $\{1,2,\ldots,n\}$. If we delete some vertices of the graph, other vertices are renumbered and this can make associating a vertex's original number with its new number difficult. Also, one may wish to give a vertex an alphanumeric name.

To deal with these issues, MATGRAPH provides a mechanism for labeling the vertices of a graph with arbitrary text strings.

Try this:

```
>> g = graph
Graph with 0 vertices and 0 edges (full)
>> cycle(g,8)
>> label(g)
>> label(g,3,'X')
>> delete(g,4)
>> ldraw(g)
```

When a graph is first created, there are no labels associated with its vertices. The command `label(g)` causes `g`'s vertices to be given default labels. The default label assigned to a vertex is simply a string containing the digits of its vertex number (e.g., vertex 23 would be labeled `'23'`).

The command `label(g,3,'X')` labels vertex number 3 with the string `'X'`. The label need not be a single character, we could have labeled this vertex like this: `label(g,3,'three')`.

Next we delete vertex number 4. This renumbers vertices 5 through 8 to new numbers 4 through 7. However, the label associated with a vertex remains the same. That is, the vertex now numbered 4 (and formally numbered 5) has the label `'5'`.

Finally, the `ldraw` command draws the graph with each vertex's label written in the middle of its circle. See Figure 5.

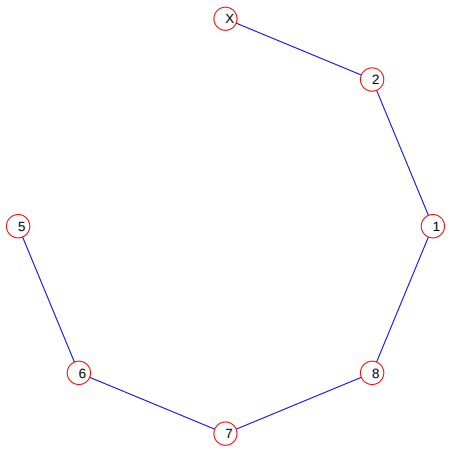

FIGURE 5. A drawing of a labeled graph.

To learn the label of a vertex, or to extract a cell array containing all the labels of a graph, use `get_label`.

It is possible to assign two different vertices the same label, so there need not be a one-to-one correspondence between vertices and label.

6. GRAPH OPERATIONS

MATGRAPH provides various operations to perform on graphs. Here we present some examples.

```
>> g = graph
Graph system initialized. Number of slots = 500.
Graph with 0 vertices and 0 edges (full)
>> complete(g,[2,3,4])
>> deg(g)
ans =
     7     7     6     6     6     5     5     5     5
>> complement(g)
>> deg(g)
ans =
     1     1     2     2     2     3     3     3     3
>>
```

This sets `g` to be the complete multipartite graph $K_{2,3,4}$, and then overwrites `g` with its own complement, $\overline{K_{2,3,4}}$. This is equivalent to the disjoint union $K_2 \oplus K_3 \oplus K_4$. MATGRAPH can compute disjoint unions of graphs like this:

```
>> cycle(g,5)
>> h = graph;
>> cycle(h,6)
>> k = graph;
>> disjoint_union(k,g,h)
>> k
Graph with 11 vertices and 11 edges (full)
```

This code resets `g` to be the 5-cycle, defines a new graph variable `h` to be a 6-cycle, and then places the disjoint union of these graphs in a third graph `k`. See also the `union` command.

The complement of $C_5 \oplus C_6$ can now be computed using `complement(k)`. Alternatively, we can do this:

```
>> complement(g);   % g is now the complement of C_5 (which is C_5)
>> complement(h);   % h is now the complement of C_6
>> join(k,g,h)
>> h
Graph with 6 vertices and 9 edges (full)
```

The `join` commands overwrites its first argument with a graph formed from the disjoint union of its second and third arguments, plus all possible edges between these latter two graphs.

The *Cartesian product* of graphs G and H is a new graph $G \times H$ defined as follows:

$$V(G \times H) = V(G) \times V(H) = \{(v,w) : v \in V(G), w \in V(H)\}$$

$$E(G \times H) = \big\{\{(v_1,w_1),(v_2,w_2)\} : [v_1v_2 \in E(G) \text{ and } w_1 = w_2] \text{ or } [v_1 = v_2 \text{ and } w_1w_2 \in E(H)]\big\}$$

We illustrate how to calculate Cartesian product in MATGRAPH:

```
>> clf;draw(k)
>> cycle(g,10)
>> cycle(h,3)
>> cartesian(k,g,h)
>> k
Graph with 30 vertices and 60 edges (full)
>> distxy(k)
Optimization terminated: relative function value
 changing by less than OPTIONS.TolFun.
Embedding score = 51.6601
Elapsed time is 5.263166 seconds.
ans =
   51.6601
>> clf;draw(k)
```

The resulting drawing is shown in Figure 6. The hypercube Q_n is defined to be the n-fold product

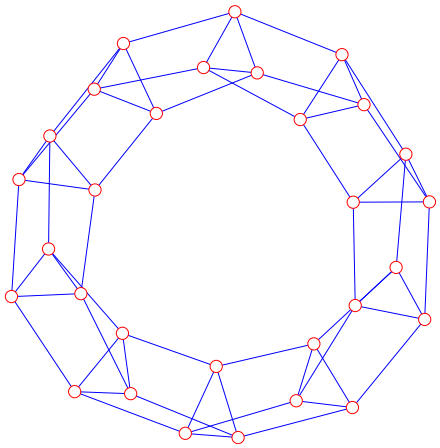

FIGURE 6. The Cartesian product $C_{10} \times C_3$.

$K_2 \times K_2 \times \cdots \times K_2$. The command `cube(g,n)` overwrites `g` with the graph Q_n.

MATGRAPH can find spanning trees in (connected) graphs. The two commands `bfstree` and `dfstree` find breadth-first and depth-first spanning trees of their respective graphs.

```
>> dodecahedron(g)
>> bfstree(h,g)
>> clf; draw(g,':')
>> draw(h)
```

The command `bfstree(h,g)` overwrites `h` with a breadth-first spanning tree of `g` rooted at vertex 1. (To start from another vertex, use `bfstree(h,g,v)`.) We then draw the original graph `g` using dotted lines (the extra argument to `draw`) and then draw the spanning tree (using the default solid lines) without erasing the first drawing. The result is in Figure 7.

MATGRAPH can also find Hamiltonian cycles (but only in small graphs). Here's an example.

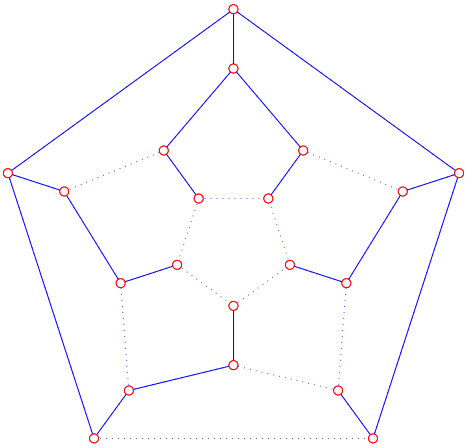

FIGURE 7. A breadth-first spanning tree of the dodecahedron graph.

```
>> dodecahedron(g)
>> hamiltonian_cycle(h,g);
>> clf;draw(g,':')
>> draw(h)
```

The result is in Figure 8.

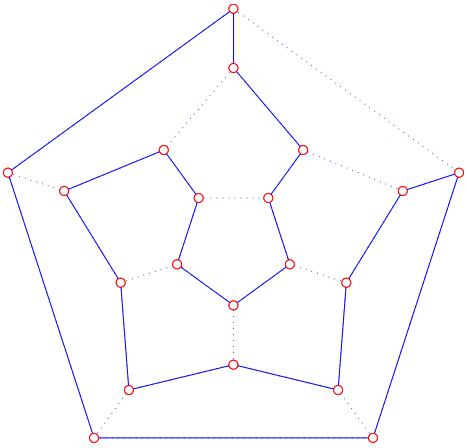

FIGURE 8. A Hamiltonian cycle in the dodecahedron graph.

MATGRAPH can form induced subgraphs.

```
>> cycle(g,10)
>> induce(h,g,[1 2 3 4 5 9])
>> h
Graph with 6 vertices and 4 edges (full)
```

The `induce` command above overwrites `h` with the induced subgraph of `g` generated by the vertex set $\{1,2,3,4,5,9\}$. This makes `h` the graph consisting of a 5-path (on vertices 1 through 5) plus an isolated vertex (now numbered 6 in `h`). The new vertex 6 in `h` inherits the label of vertex 9 in `g` (assuming `g` was labeled).

The `trim` command is useful for removing vertices of degree 0. More generally, `trim(g,d)` removes all vertices of degree at most d from `g`, and then repeats this operation on the resulting graph until `g` has minimum degree at least $d+1$ (or all vertices have been deleted).

7. Graph Computations

7.1. **Basic invariants.** As discussed earlier, `nv(g)` and `ne(g)` returns the number of vertices and edges in `g`, respectively. `size(g)` reports the same information as a list.

The independence number, clique, and domination number can be computed by MATGRAPH; note that the computation of these invariants requires MATLAB's Optimization Toolbox.

```
>> g = graph
Graph system initialized. Number of slots = 500.
Graph with 0 vertices and 0 edges (full)
>> icosahedron(g)
>> alpha(g)  % compute the independence number
Optimization terminated.
ans =
     3
>> omega(g)  % compute the clique number
Optimization terminated.
ans =
     3
>> dom(g)    % compute the domination number
Optimization terminated.
ans =
     2
```

In each case, we can find the realizing set (independent, clique, or dominating) with an extra output argument:

```
>> [d,S] = dom(g)
Optimization terminated.
d =
     2
S =
     4
     7
>> sort([g(4),g(7),4,7])
ans =
    1    2    3    4    5    6    7    8    9   10   11   12
```

7.2. **Connection.** MATGRAPH can determine if a graph is connected, find paths between vertices, and determine distances. We illustrate this on the "Bucky ball" graph: the molecular graph of Buckminsterfullerene C_{60} (a ball comprised of 60 carbon atoms) or, equivalently, the graph implicitly drawn on a soccer ball.

```
>> bucky(g)
>> isconnected(g)
ans =
     1
>> find_path(g,1,60)
ans =
     1     5     4    21    22    23    52    51    55    60
>> diam(g)
ans =
     9
>> dist(g,1,60)
ans =
     9
>>
```

MATGRAPH can find the connected components of a graph; these are returned as a partition object:

```
>> complete(g,[2,3,4])
>> complement(g)
>> components(g)
{ {1,2} {3,4,5} {6,7,8,9} }
>> component(g,3)
ans =
     3
     4
     5
```

The last function, `component(g,v)`, returns a list of the vertices in `v`'s component of `g`.

The `split` command finds a reasonable partition of the vertices of a graph into two sets that are more tightly clustered among themselves than between the two sets. For example, consider a graph formed by combining two disjoint copies of K_8 linked by a single edge. This is a connected graph, but clearly divides into two natural clusters. Here we show how this works in MATGRAPH:

```
>> h = graph
Graph with 0 vertices and 0 edges (full)
>> complete(g,8)
>> disjoint_union(h,g,g)
>> add(h,1,16)
>> components(h)
{ {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16} }
>> split(h)
{ {1,2,3,4,5,6,7,8} {9,10,11,12,13,14,15,16} }
```

7.3. **Coloring.** Perhaps the most celebrated invariant in graph theory is the chromatic number of a graph, $\chi(G)$. This is the minimum number of colors needed so that we can color the vertices of G such that adjacent vertices have different colors. Equivalently, this is the minimum number of blocks in a partition of $V(G)$ into independent sets.

The `color` command can be used to find such a partition. The default `color(g)` performs a greedy coloring of the graph (which might not be optimal). Other algorithms can be specified; for

example, to get a true optimal coloring, use `color(g,'optimal')`. This, of course, may take a long time for large graphs.

```
>> icosahedron(g)
>> color(g)
{ {1,6,8} {2,4,11} {3,5,12} {7,9,10} }
>> bucky(g)
>> c1 = color(g);
>> size(c1)
ans =
    60     4
>> c2 = color(g,'optimal');
>> size(c2)
ans =
    60     3
>> cdraw(g,c2)
```

Notice that the greedy coloring produces a proper 4-coloring of the graph, but the best-possible coloring is with three colors. See Figure 9 produced by this code. The `cdraw` command draws

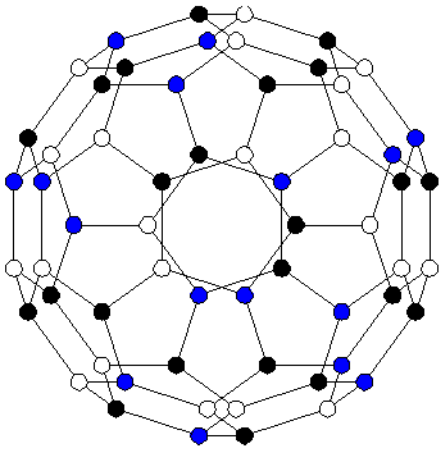

FIGURE 9. An optimal coloring of the Bucky ball.

a graph with a given coloring. Note that the coloring need not be a proper coloring. Here is an example:

```
>> grid(g,5,5)
>> c = split(g);
>> clf; cdraw(g,c)
```

The result is in Figure 10.

MATGRAPH can find the chromatic polynomial of small graphs.

```
>> cube(g,3)
>> chromatic_poly(g)
ans =
```

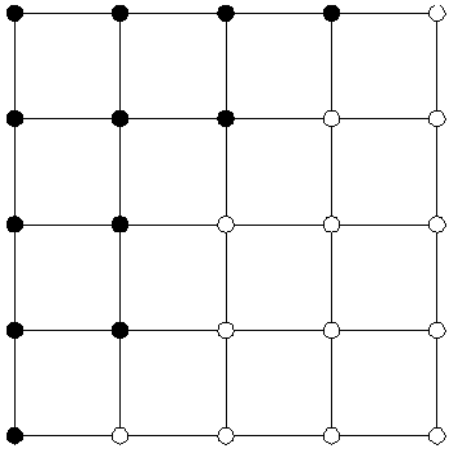

FIGURE 10. A 5×5 grid partitioned into two sets of vertices by `split`.

```
     1   -12    66  -214   441  -572   423  -133     0
```

This tells us that

$$\chi(Q_3;x) = x^8 - 12x^7 + 66x^6 - 215x^5 + 441x^4 - 572x^3 + 423x^2 - 133x.$$

If a graph has a two-coloring (i.e., if the graph is bipartite) then we can use `bipartition` to find the two color classes.

```
>> cycle(g,8)
>> bipartition(g)
{ {1,3,5,7} {2,4,6,8} }
```

Given a partition of a graph into two sets, we can find a maximum matching between those sets with `bipmatch`.

```
>> random_bipartite(g,6,6,.5)
>> bipartition(g)
{ {1,2,3,4,5,6} {7,8,9,10,11,12} }
>> bipmatch(g,ans)
ans =
     1     7
     2     8
     3     9
     4    11
     5    10
     6    12
```

8. SPARSE GRAPHS

Graphs in MATGRAPH are housed in symmetric matrices. MATLAB can hold matrices either as *full* or *sparse* arrays. The amount of memory used by a full array is proportional to the number of entries in the matrix, while the memory used by a sparse array is proportional to the number of nonzero entries in the matrix.

Graphs in MATGRAPH are held, behind the scenes, in either full or sparse matrices. To find out which, use the functions `isfull` or `issparse`. Alternatively, simply typing the graph variable's name reveals its storage type.

```
>> petersen(g)
>> g
Graph with 10 vertices and 15 edges (full)
```

For large graphs with relatively few edges, sparse storage is preferable; indeed, full storage may not be feasible because the computer might not have enough RAM to hold the matrix. To convert a graph to sparse storage, simply type `sparse(g)`.

```
>> sparse(g)
>> cycle(g,1000)
>> g
Graph with 1000 vertices and 1000 edges (sparse)
```

When declaring a new graph variable, one may specify the number of vertices in the constructor: `h = graph(n)`. If `n` is large, then sparse storage is used.

```
>> k = graph(10000)
Graph with 10000 vertices and 0 edges (sparse)
```

How large is "large"? This is controlled by the function `set_large`.

9. INPUT AND OUTPUT

9.1. **Saving graphs to disk with `save` and `load`.** The usual mechanisms for saving variables to disk do not work for `graph` variables in MATGRAPH. Were you to attempt to save a `graph` variable, or the entire MATLAB workspace, the graphs you have created will be lost when you try to load them back in. This is one of the prices we pay for creating a fast call-by-reference system.

Instead, MATGRAPH provides its own `save` and `load` commands. `save(g,filename)` saves the graph `g` to a file in the current directory on your hard drive. A subsequent call to `load(g,filename)` overwrites the graph `g` with the graph saved in the file. Here is an example:

```
>> g = graph
Graph system initialized. Number of slots = 500.
Graph with 0 vertices and 0 edges (full)
>> petersen(g)
>> save(g,'pete')
>> free(g)
>> g
Invalid graph object (index 1)
>> clear g
>> g = graph
Graph with 0 vertices and 0 edges (full)
>> g
Graph with 0 vertices and 0 edges (full)
>> load(g,'pete')
>> g
Graph with 10 vertices and 15 edges (full)
```

9.2. **SGF: Simple Graph Format.** The MATGRAPH function `sgf` is a mechanism to convert `graph` objects to and from a two-column matrix format called Simple Graph Format. For a graph with n vertices and m edges, the Simple Graph Format matrix has either $m+1$ or $n+m+1$ rows. The first row of the matrix gives the number of vertices and the number of edges in the graph. The following m rows specify the edges of the graph. Optionally, an additional n rows specify the x,y-coordinates of the embedding of the graph. Here is an example.

```
>> complete(g,4)
>> sgf(g)
ans =
     4     6
     1     2
     1     3
     2     3
     1     4
     2     4
     3     4
>> distxy(g)
Optimization terminated: relative function value
 changing by less than OPTIONS.TolFun.
Embedding score = 0.34315
Elapsed time is 0.079532 seconds.
ans =
    0.3431
>> sgf(g)
ans =
    4.0000    6.0000
    1.0000    2.0000
    1.0000    3.0000
    2.0000    3.0000
    1.0000    4.0000
    2.0000    4.0000
    3.0000    4.0000
    1.3651    1.3939
    1.2374    2.5943
    0.7011    1.9303
    1.9014    2.0580
```

Not only can `sgf` be used to create a Simple Graph Format matrix from a graph, it can also be used to specify a graph. For example, here we create the SGF matrix for the graph $K_{1,5}$ and an embedding using MATLAB commands, and then build a graph based on that matrix.

```
>> edges = [ ones(5,1), [2:6]' ]
edges =
     1     2
     1     3
     1     4
     1     5
     1     6
```

```
>> xy = [ 0 0 ; -2 1 ; -1 1 ; 0 1 ; 1 1 ; 2 1 ];
>> S = [ 6 5 ; edges ; xy ]
S =
     6     5
     1     2
     1     3
     1     4
     1     5
     1     6
     0     0
    -2     1
    -1     1
     0     1
     1     1
     2     1
>> sgf(g,S)
>> clf;draw(g)
```

The result is show in Figure 11.

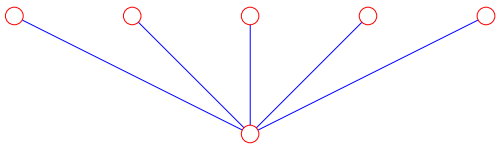

FIGURE 11. A star graph created using a Simple Graph Format matrix.

The Simple Graph Format is useful for working with other computing environments. You may have, say, a C++ program that you use to create graphs. You can have that program write the graph to disk in simple graph format. Then, using the usual MATLAB `load` command, the two-column matrix can be read from disk and converted into a graph.

9.3. **A C++ graph parser.** Inside the main MATGRAPH directory, you can find a subdirectory named `tools` that contains a further subdirectory named `graph_parser`. This directory contains a C++ program to build a command-line tool that reads textual graph data from the standard input and writes its output to a file named `parsed_graph.m`. This can then be converted into a graph in MATGRAPH by giving the command `parsed_graph(g)`. Here are the steps you need to take to make this work.

Compile the program. We assume basic knowledge of the Unix shell (Linux, Mac OS X, Cygwin on Windows, etc.) and that your computer has a C++ compiler installed. (This has been tested using the GNU compiler `g++`.)

To build the program, simply change directory to the `graph_parser` directory and type `make`:

```
$ cd /home/username/matgraph/tools/graph_parser/
$ make
```

```
g++ -ansi -O   -c -o main.o main.cc
g++ -ansi -O   -c -o LineParser.o LineParser.cc
g++ main.o LineParser.o -o graph_parser
$
```

The program `graph_parser` is created. This can be moved to any convenient location.

Graph data file. The `graph_parser` program reads a specific type of data file. Vertices are named as character strings (henceforth, "words") such as `head-node` or `city` or `123`. No white space may appear in the name of a vertex and vertex names are case sensitive (the word `hello` is not the same as `Hello`).

A typical line in the data file contains the name of exactly two words; such a line indicates that there is an edge between the named vertices. If there are more than two words on a line, only the first two words are processed; the rest of the line is ignored.

In order to accommodate isolated vertices, a line in the data file may contain just a single word. This tells `graph_parser` that the given word is the name of a vertex. If this word has not been previously encountered (e.g., on a previous line as part of an edge), then this names a new vertex in the graph.

If a line begins with the same word twice, the second instance of the word is ignored and this line is treated as if it contained only one word.

Finally, if a line is blank or if a line begins with the sharp character `#`, then the line is ignored (this is useful for annotating the data file).

A typical input file (named `test`) is included in the `graph_parser` directory; we show the contents of that file here:

```
one two
one              three
four
five two
two six      and the rest of this line is ignored
seven
eight nine
nine two
one one   <-- a loop is not created
two nine

eight seven
eight one  fifty
four six
seven six
four five
   three five
# this line should be skipped
nine seven
```

Convert the data file into a `.m` file. Once the data file is prepared, we use `graph_parser` to convert the data file into a `.m` file that can be run in MATLAB. In the shell, give the following command:

```
./graph_parser < filename
```

where `filename` is the name of the file containing the graph data. The result of running the program `graph_parser` is the creation of a file named `parsed_graph.m` in the same directory in which `graph_parser` was run. You need to have write permission for that directory or `graph_parser` will complain:

```
Unable to open file parsed_graph.m for output
```

The file `parsed_graph.m` can be moved to any convenient location. You should not change the name of this file because it is a MATLAB function. If you wish to process two (or more) graph files, run `graph_parser` on the first data file and then read the graph into MATLAB (explained next) before processing subsequent data files.

Run the `.m` file in MATLAB. The final step is to execute the function `parsed_graph` inside MATLAB.

```
>> parsed_graph(g)
>> g
Graph with 9 vertices and 13 edges (full)
>> distxy(g)
Optimization terminated: relative function value
 changing by less than OPTIONS.TolFun.
Embedding score = 2.5448
Elapsed time is 0.210241 seconds.
ans =
    2.5448
>> ldraw(g)
```

The graph defined textually in `test` is now saved as `graph` object in MATGRAPH and can be handled like any other such graph. The drawing of this graph is shown in Figure 12.

9.4. **Connecting with other programs.** It is possible to create MATLAB programs to write graphs to files in other formats. Included with MATGRAPH are ways to do this for Graphviz and OmniGraffle.

Saving graphs for Graphviz. Graphviz is a graph visualization tool available from the website

```
http://www.graphviz.org/
```

One of the Graphviz tools is named `dot`, and MATGRAPH includes a function also named `dot` to convert `graph` objects into a format that can be read by Graphviz's `dot`. The MATGRAPH command has the form `dot(g,'filename.dot')`. This writes a file to the computer's disk that can then be used by Graphviz. Here is an example of how to do this:

```
>> cube(g,4)
>> dot(g,'four-cube.dot')
Wrote "four-cube.dot"
```

The file `four-cube.dot` is now read into a Graphviz tool to produce attractive drawings such as the one shown in Figure 13. See the Graphviz website for more information.

Saving graphs for OmniGraffle. OmniGraffle is a graph drawing program for Macintosh available from this website:

```
http://www.omnigroup.com/
```

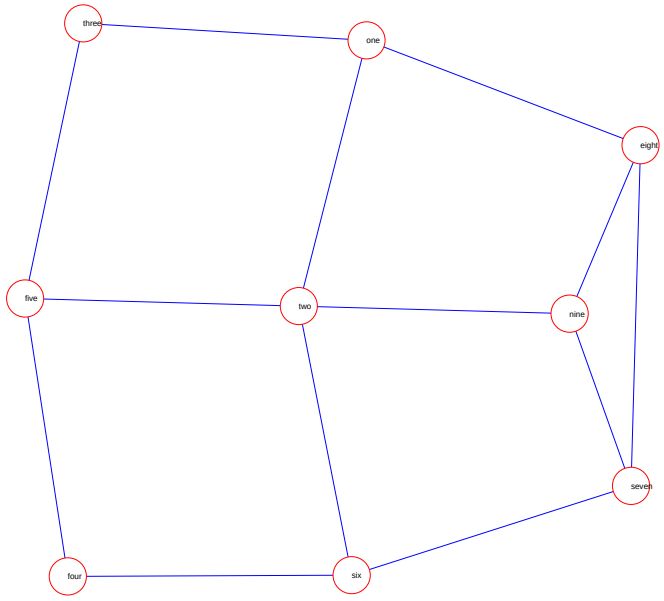

FIGURE 12. A drawing of a graph read into MATGRAPH via the `graph_parser` program.

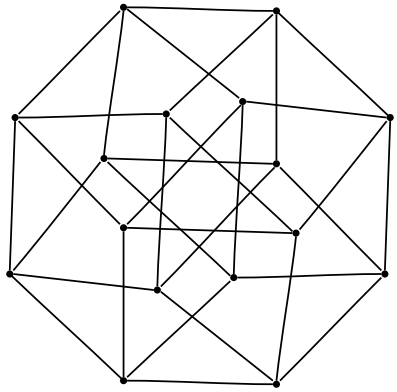

FIGURE 13. A picture of Q_4 produced by exporting a graph from MATGRAPH and then laid out using GraphViz.

MATGRAPH can save graphs in a format that can be read by OmniGraffle. The MATGRAPH command `graffle(g,'filename.graffle')` writes the graph to disk. Double clicking the created file launches OmniGraffle. Here's an example:

```
>> cube(g,3)
>> graffle(g,'cube.graffle')
```

Using OmniGraffle's layout tool, we can produce a nice embedding of the graph as shown in Figure 14.

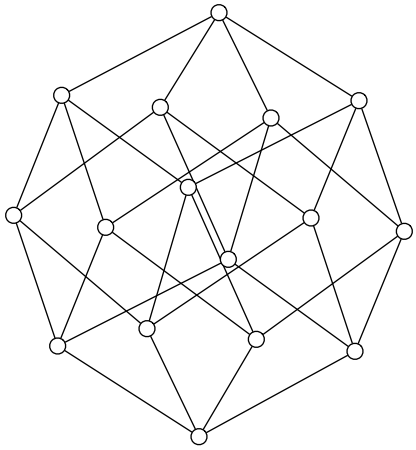

FIGURE 14. A picture of Q_4 produced by exporting a graph from MATGRAPH and then laid out using OmniGraffle.

To a limited extent, it is possible to convert graphs prepared in OmniGraffle for import into MATGRAPH. Inside the `matgraph/tools` directory resides a program named `graffle2sgf.py`. This is a Python language program so, in order to run it you must have Python installed on your computer. This program takes as input a graph saved by OmniGraffle and returns as output a matrix specifying the graph in Simple Graph Format (see §9.2).

Suppose you have created a graph using OmniGraffle and saved it on your hard disk in a file called `mygraph.graffle`. Issue the following command in the Unix shell:

```
./graffle2sgf.py < mygraph.graffle > mygraph
```

This reads the graph saved by OmniGraffle and writes the relevant data into the file `mygraph`.

Now, inside MATLAB, do the following:

```
>> load mygraph
>> sgf(g,mygraph)
>> g
Graph with 5 vertices and 6 edges (full)
```

The MATLAB command `load mygraph` reads the file `mygraph` and saves the matrix contained therein into a variable that is also named `mygraph`. The command `sgf(g,mygraph)` overwrites `g` with the graph specified by the SGF matrix `mygraph`.

The `graffle2sgf.py` tool is not completely reliable. It works well on diagrams that contain only nodes and edges. If there are other extraneous lines or text in the diagram (which an OmniGraffle diagram certainly may have), then the program can get confused and give poor performance. Readers are invited to submit a better version.

DEPARTMENT OF APPLIED MATHEMATICS AND STATISTICS, THE JOHNS HOPKINS UNIVERSITY, BALTIMORE, MARYLAND 21218-2682 USA

E-mail address: ers@jhu.edu