

University of Colorado
Introduction to Engineering Computing - GEEN 1300
Spring 2014
MATLAB Challenge Project: Bracing a Structure

Readings: posted pdf scan from Baglivo and Graver, Incidence and Symmetry in Design and Architecture, Cambridge University Press, Cambridge, 1983.

Basic assignment:

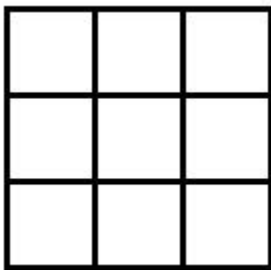
1. Read in a braced grid of any size from a file.
2. Convert the grid to a structure called a Graph, which represents rows and columns of the grid.
3. Search the Graph to determine if the braced grid is (a) rigid in the plane and (b) minimal, in the sense that no extra bracings are needed.

Extra credit:

1. Extend this approach to a tension-braced grid.

Background.

Consider a grid of steel beams, like this.



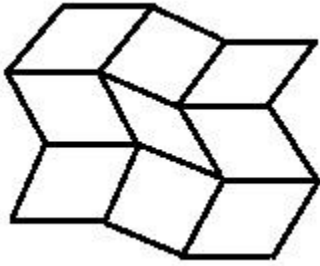
Assume that the beams are only one square long, and that they are connected by pin joints. This means that each individual square in the grid



can be squashed into a trapezoid. Note that the side lengths don't change (because steel beams don't really stretch), but the angles between the sides do. You can also assume that all the squashing happens in the plane of the page here—don't worry about the grid folding out of plane.



It's possible for the whole grid to get squashed in different directions, as well; see the unfortunate 3 x 3 grid below.



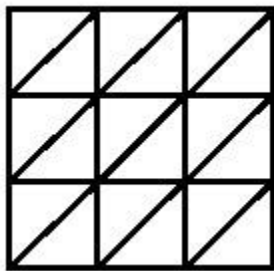
From a structural engineering standpoint, the above squashing is bad news. If that grid is the structural support for your walls, then your house has probably fallen down already.

We can brace a single square crosswise, in either direction, to stop the square from squashing.

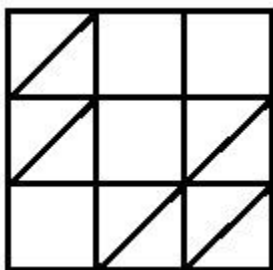


For this problem (excluding the extra credit), it does not matter which of the two diagonal directions we brace.

By extension, we could brace all the squares in the grid to keep it from squashing, as below. That's got to be rigid.

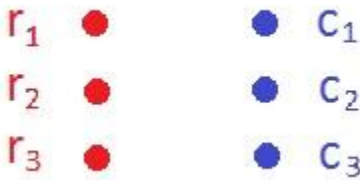


But we don't actually need to brace all of the squares for this. Here's a smaller bracing, which involves only 5 squares, and still keeps the whole grid rigid in the plane.

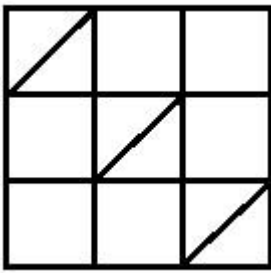


Suppose we want to find the minimum number of squares that brace the grid completely, so that we can save big bucks on construction and the buildings we design still won't embarrass us by falling down. Finding which squares to brace for a minimum grid is tricky and not very intuitive: see the readings page from Baglivo and Graver (marked p.79) for an example of different 3x3 grid bracings that are squashable.

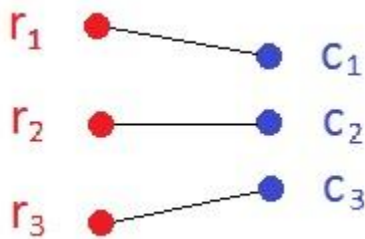
Assuming that our grid is an $n \times m$ matrix, we can convert it into a graph with a set of n 'row vertices' ($r_1, r_2, r_3, \dots, r_n$) on one side, and a set of m 'column vertices' ($c_1, c_2, c_3, \dots, c_m$) on the other side. Here you see the graph for a 3x3 grid (3 row vertices and 3 column vertices result from a 3x3 matrix).



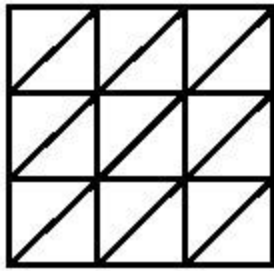
When our grid contains a brace at grid square (i,j) , we make an edge in our graph between row vertex r_i and row vertex c_j . For this grid, with $n = 3$ rows and $m = 3$ columns, and 3 braces,



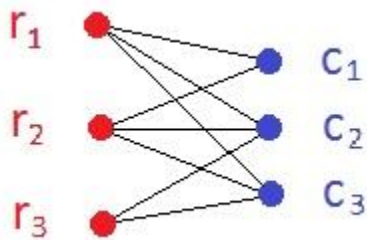
the resulting graph looks like this. It has 3 edges, one for each brace.



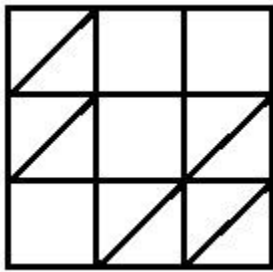
If the grid instead has all the possible braces,



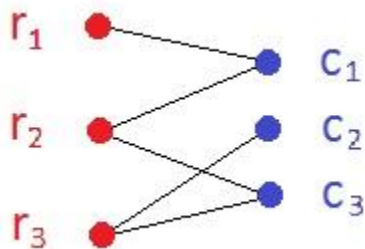
then its graph looks like this. All the edges that can exist between a row vertex r_i and a column vertex c_j are there.



If the grid has a minimal bracing,



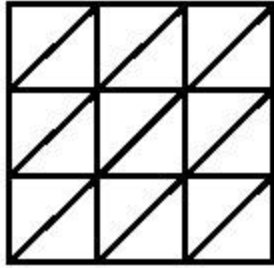
then its graph looks like this.



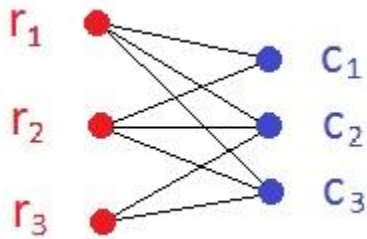
Surprisingly, you can solve the bracing problem over a grid quickly by transforming it to a graph problem. There are 2 rules about minimum rigid bracings and graphs:

1. The bracing is rigid if the graph's edges let us trace out a path that touches every vertex.

In this fully braced grid,



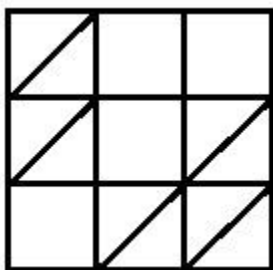
the graph is



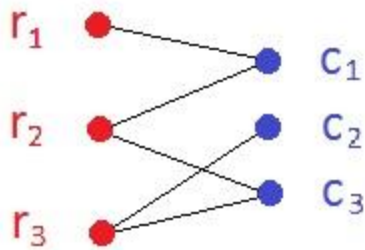
and we can follow the edges from c_1 to r_1 to c_2 to r_2 to c_3 to r_3 —thus there is a path, via the edges of the graph, that connects every vertex in the graph. In this example, other paths that also touch all 6 vertices are possible. Any time we have a set of edges that give us a path touching all the row and column vertices, we have a rigid bracing. (In this particular example, we'd better, since we braced all the squares!) Notice that we don't have a minimal bracing, in this case. But we definitely have a rigid one.

2. The bracing is minimal if the graph's edges never make a cycle, which is a closed loop in the graph.

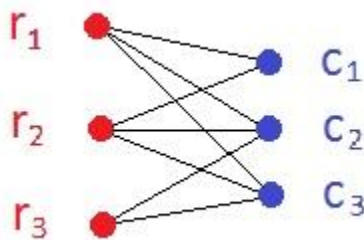
The minimal bracing



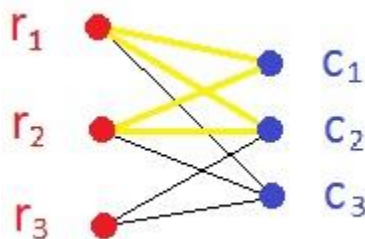
has the graph



and here the path r_1 to c_1 to r_2 to c_3 to r_3 to c_2 touches every vertex (so it's rigid), and the graph contains no cycles—there is no way to follow a series of edges that double back on any vertex in the path. By contrast, the over-braced graph



lets you go from r_1 to c_1 to r_2 to c_2 and then back to r_1 , which is a closed edge loop, or a cycle, with a bow-tie shape (see the yellow cycle below). Several other cycles exist in this graph, of course.



If I give you a rectangular grid with n rows and m columns, and some of the grid squares are braced, can you tell me if that grid is

(1) rigid, meaning that a path of edges connects all the row and column vertices in the graph, and

(2) minimal, meaning that no cycles exist in the graph's edges?

If both of those things are true simultaneously, then you have a minimal rigid bracing.

Project checkpoint #1: On the Moodle Project Quiz, sign up for this project.

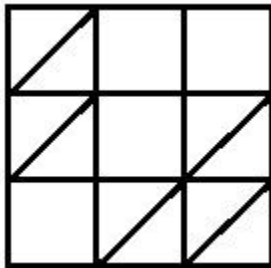
Project checkpoint #2 (4/4): To start, open an m-file called `bracing.m` in a directory of your usb drive and put my `GEENGraph` and `GEENStack` m-files in the same directory as `bracing.m`. (You don't need those `GEEN` m-files—sorry—only `bracing.m`)

Inside the file, begin the function

```
function bracing()
```

Inside this `bracing()` function, you create a graph by adding vertices to a 1D array and then adding edges between these vertices to an $n \times n$ sized array. See the notes from 3/13 for a graph example.

Suppose that someone (like me) gives you a data file for the rectangular grid, with n rows and m columns; if the grid square at that row and column is unbraced, the data file has a 0, and if it is braced, the file has a 1. If the file tells us to make this braced grid,



then the file looks like this:

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

This 3 x 3 matrix defines the grid input. Start by making a small txt file like the one above and learning to read it in; the load command from Lecture 15 may be useful here. Eventually, you will need to read in ANY file with a rectangular matrix (not just the one in this 3 x 3 example) as one of the starting tasks in the project. See lectures 14 and 15 for slides on this. In the `bracing.m` file, after the `bracing()` function you started, add the function to read in a grid. Here is the first line of that function:

```
function [ok, grid_bracing] = read_grid_file(filename)
```

This function, `read_grid_file`, gives back a 2D matrix of zeros and ones, and the value `ok` (which is true if nothing bad happened, and false otherwise). Writing this function is your next task. Test this new function by going back to the end of the `bracing()` function, and adding the line:

```
[ok, bracing] = read_grid_file('sample_grid01.txt')
```

where `'sample_grid01.txt'` is the name of that text file you made with the 3x3 grid. Later, you will prompt the user for a filename, and open whatever grid file he tells you to.

You will then need to write MATLAB code that uses the rectangular grid matrix you read in to build a graph for this problem. A grid of q rows and r columns gives you a graph with $(q + r)$ vertices. Let's simplify $q + r$ to p for now. A graph with p vertices has to keep track of a $1 \times p$ (or $p \times 1$) list of vertex names, which you should encode as a cell array; in addition, it also maintains a regular matrix of size $p \times p$ for the edges. You will define a cell array and add as many row vertices as the grid has rows, all with distinct names; call them `r1`, `r2`, ..., etc., up to the maximum row number in your grid. (You will need to

create the right name as a string; the MATLAB functions `strcat` and `num2str` may help here.) Then you will do the same to add the column vertices to the cell array; call them `c1`, `c2`, ..., etc., up to the maximum column number.

For the 3x3 grid example, you'd have 6 vertices in all.

```
vertices{1,1} = 'r1'
```


```
vertices{1,2} = 'r2'
```

```
vertices{1,3} = 'r3'
```

```
vertices{1,4} = 'c1'
```

```
vertices{1,5} = 'c2'
```

```
vertices{1,6} = 'c3'
```

Once you have the vertices for all the rows and columns added to your cell array, define edges, a square matrix of zeros. If your bracing has q rows and r columns, edges is a $(q+r) \times (q+r)$ matrix. In the above example, this is a 6 x 6 matrix. Write a nested for-loop that goes over the row and columns of your braced grid; if the inner loop code finds a brace at row 2 and column 3 of the grid, for instance, then it adds an edge between the `r2` (index 2) and `c3` (index 6) vertices of the graph; this means that `edges(2, 6)` turns from 0 to 1. Add the reverse edge as well by setting `edges(6, 2) = 1`. Your edge graph at the end should be symmetric around the upper left to lower right diagonal. 

Project checkpoint #3 (4/11): On moodle, turn in a printout of your `bracing.m` file so far, with the ability to read in a grid file and construct the vertices and edges of the graph. Display the vertices in your cell array and display the edges matrix.

You can tell if the graph is connected using an algorithm called **depth first traversal**, which checks the graph for paths between vertices. In depth first traversal, you begin from any vertex you like (this starting point is completely arbitrary). Part 1 of your mission is to find every vertex that you can reach by following edges starting from this vertex; if all of the row and column vertices are reachable via some path from the starting vertex, then the graph is connected, and therefore the corresponding grid is a rigid bracing. Part 2, described later, checks if your graph of edges between these vertices is cyclical. If the graph is connected and has no cycles, then the bracing is a minimal rigid bracing, with no redundant braces.

Depth first search requires you to remember which vertices you have already visited. We do this with a vector called **visited**. At the start, the visited vector should contain all zeros. If we know the number of vertices in the graph, then we can make that **visited vector** in one line of MATLAB using the `zeros` command. Figure out this size and build this vector in your `bracing()` function. Also, build a **predecessor array** of the same size, also filled with zeros, in a second line of `bracing()`.

In the same m-file, but **after** your `bracing()` function, add a function for depth first search, called **DFS**. Here is the first line of my version of this function.

```
function [vertices, edges, v, visited, predecessor] = ...  
    DFS(vertices, edges, v, visited, predecessor)
```


In your `bracing()` function, you can choose any vertex v to start the search; you specify it by its index in your vertex array. Then this code should call the `DFS()` function once, and it will explore the rest of the graph until it's checked out every connection. Here is one way to call the DFS function you're about to write in the `bracing()` code:

```
[vertices, edges, v, visited_verts, predecessors] = ...
    DFS(vertices, edges, 1, visited_verts, predecessors)
```

So the next task is to finish the function for depth first search,

```
function [vertices, edges, v, visited, predecessor] = ...
    DFS2(vertices, edges, v, visited, predecessor)
```

Here's the pseudocode logic behind DFS from http://en.wikipedia.org/wiki/Depth-first_search; I modified this to be simpler by taking out the code involving visited edges, since that information is not required for this problem. This code is recursive, which means that it calls itself (!); see line 6. You have not seen a function do that before.

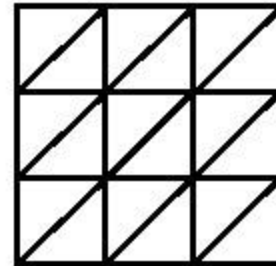
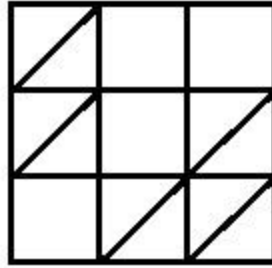
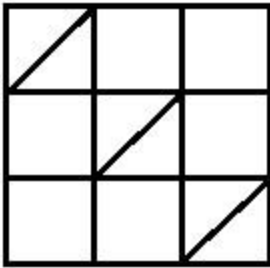
```
1  procedure DFS( $G, v$ ) :
2      label  $v$  as visited
3      for all edges  $e$  from  $v$ 
4           $w \leftarrow$  adjacent vertex reachable from  $v$  via  $e$ 
5          if vertex  $w$  is unvisited
6              DFS( $G, w$ )
7          end
8      end
```

Assume that v is still that starting vertex (try vertex 1) at the start. That G in the algorithm above is really just your Graph's vertices, edges, visited, and predecessor arrays all put together. Now you see why this function's input and outputs are so long—you are mailing all those vectors and matrices back and forth.

```
function [vertices, edges, v, visited, predecessor] = ...
    DFS2(vertices, edges, v, visited, predecessor)
```

Your job is to convert as much of the DFS pseudocode above to MATLAB code as you can. Refer to Lecture 16 (this upcoming Monday) for a little background.

Project checkpoint #4 (4/18): Email your finalized m-file, with the DFS code added, to the project TA. Upload it to Moodle as well. Also, trace out how the depth first search algorithm will explore the 3×3 grids below, once they are in graph form:



Email a PDF file or a scan of this exploration to the project TA (email TBA) and arrange an appointment with the project TA to check you off on this during the week of 4/14-4/18. Be prepared to explain your reasoning and your next steps.

Some hints for finishing after checkpoint 3:

Your DFS code tells you if a path connects all the vertices when you inspect the visited array. What must be true of the visited array after DFS runs to completion if the graph is connected? What's true if it's not connected?

If the graph is connected through all the vertices, and it has no cycles, what relation must hold between the number of edges and the number of vertices in the graph? (This sounds hard but is really a surprisingly simple answer, if you sketch it out.)

Using your 2 cases above, it is straightforward to determine whether the input grid has a rigid bracing, and if so, whether it is minimal.

Extra Credit:

(3 pts) Write another MATLAB function in your m-file that draws a MATLAB plot of the graph that you built from your grid (a plot like the graphs in the handout above), with different colored vertices for rows and columns, vertex labels, and appropriate edges.

(2 pts) Write a MATLAB function that takes a non-minimal rigid bracing and tries to make it minimal by taking out redundant braces until it can confirm that the bracing is still rigid and now minimal. Have it write the old bracing and the new bracing to 2 separate text files.

(5 pts) In a separate m-file called tensing.m, extend this analysis to a tension model. See the PDF file by Baglivo and Graver for more information. This depends on

- (1) Using a directed edge model of the graph
- (2) Taking into account how many wires duplicate one brace
- (3) Recognizing what will be true of a fully connected directed graph after the DFS function above runs to completion