Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we’ve learned, look at problems from a different perspective and allow you to ask questions about topics you don’t understand. We encourage discussing problems with other students in this lab!

Setup: Download the lab handout and code from the course website, and move it to your private directory in your unix.qatar.cmu.edu machine. Following that create a directory, move the handout to it, and unzip the handout file by executing the following commands:

```bash
% mkdir lab_14
% mv 14-handout.tgz lab_14
% cd lab_14
% tar -xvf 14-handout.tgz
```

Submission:
Create a tar file by executing the command below and submit it to autolab, under the lab name:

```bash
% tar cfzv handin.tgz graph-search.h graph-search.c graph-test.c graph.c
```

The graph interface

This lab involves implementing a graph using an adjacency matrix rather than an array of adjacency lists. Graphs will be specified by the following C interface (as in `graph.h`):
Representing undirected graphs with an adjacency matrix

In class, we discussed the adjacency list implementation of graphs. In this lab, we’ll work through the adjacency matrix implementation.

Recall that if a graph has \( n \) vertices, then its adjacency matrix \( \textbf{adj} \) is an \( n \times n \) array of booleans such that \( \textbf{adj}[i][j] \) is true if there is an edge from vertex \( i \) to vertex \( j \) (for valid \( i \) and \( j \)), false otherwise. Since the graph is undirected, if \( \textbf{adj}[i][j] \) is true, then \( \textbf{adj}[j][i] \) should also be true, and if \( \textbf{adj}[i][j] \) is false, then \( \textbf{adj}[j][i] \) should also be false. The graph should not have any self-loops (i.e., a vertex with an edge to itself).

2.a) Complete the data structure invariant function \textbf{is\_graph} that returns true if \( G \) points to a valid graph given the definition above, or false otherwise.

Make sure to capture the fact that the graph is undirected in your data structure invariant! Compare notes with a neighbor before you move on.

2.b) Complete the \textbf{graph\_new} function that creates a new graph using a dynamically-allocated 2D array of boolean for the adjacency matrix. Create the 2D array in two steps: first create a new 1D array of type \textbf{bool*}, then for each array element, have it point to a new 1D array of type \textbf{bool}. You can then access the array using the 2D notation (e.g., \( G->\textbf{adj}[0][1] = \text{true} \)).

Note: Don’t ever do this in practice! C has ways of supporting 2D arrays that don’t require an extra array of pointers; you’ll learn about this more efficient way of doing things in later classes, like 15-213.

2.c) Complete the functions \textbf{graph\_hasedge} that checks if an edge is in the graph and \textbf{graph\_addedge} that adds a new edge to the graph.

2.d) Complete the \textbf{graph\_free} function that frees any dynamically-allocated memory for the given graph \( G \).

The functions \textbf{graph\_get\_neighbors}, \textbf{graph\_hasmore\_neighbors}, \textbf{graph\_next\_neighbor} and \textbf{graph\_free\_neighbors} have been pre-implemented for you at the very bottom of file \textbf{graph.c}, but for an extra challenge write them yourself.

Once you are done implementing the functions above, you should have a complete \textbf{graph.c}. Compile your code and test it with the given DFS and BFS searches in \textbf{graph-search.c} and the given graphs in \textbf{graph-test.c}:

\begin{verbatim}
% make graphtest
% ./graphtest
\end{verbatim}

All tests should pass. (Look at the graphs in \textbf{graph-test.c} to see why.) Be sure to use \textbf{valgrind} also to make sure you have freed all memory you allocated!
**Testing for graph connectedness**

We say that a graph $G$ is *connected* if there is a path from any vertex to any other vertex in $G$.\(^1\)

For example the following graph is connected:

![Graph Diagram]

In an undirected graph, this definition is equivalent to saying that there is a path from a *single arbitrary vertex* to any other vertex. Can you see why?

(3.a) Write a function `connected(G)` in `graph-search.c` that returns `true` if a graph $G$ is connected, or `false` otherwise. Make sure your implementation is as efficient as possible.

**Hint:** Your function should work similarly to BFS, but it should count the number of vertices visited. For a connected graph, the total should be a specific value. Test your function on several graphs, connected and not connected.

(3.b) Write at least two test cases in `graph-test.c`: one where `connected` returns `true`, and one where it returns `false`.

---

\(^1\)A graph where there is an *edge* from any vertex to any other vertex is called *complete*. Complete graphs are a special case of connected graphs.