

## 15-122: Principles of Imperative Computation

### Lab E: Para, Holo, Oscillo, Tele, Odo Tom Cortina, Rob Simmons

**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 your neighbors as you work through this lab!

**Setup:** Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-graphs .
% cd lab-graphs
```

You should add your code to the existing files `graph.c`, `graph-search.c`, `graph-search.h`, and `graph-test.c` in the directory `lab-graph`.

**Grading:** You should finish (1.a), (1.b), (1.c), and (1.d) for 2 points, and additionally finish (1.e), (1.f), and (1.g) for 3 points.

### Representing undirected graphs with an adjacency matrix

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`):

```
1 typedef unsigned int vertex;
2 typedef _____* graph_t;
3
4 graph_t graph_new(unsigned int numvert);
5 //@ensures \result != NULL;
6
7 unsigned int graph_size(graph_t G);
8 //@requires G != NULL;
9
10 bool graph_hasedge(graph_t G, vertex v, vertex w);
11 //@requires G != NULL;
12 //@requires v < graph_size(G) && w < graph_size(G);
13
14 void graph_addedge(graph_t G, vertex v, vertex w);
15 //@requires G != NULL;
16 //@requires v != w && v < graph_size(G) && w < graph_size(G);
17 //@requires !graph_hasedge(G, v, w);
18
19 void graph_free(graph_t G);
20 //@requires G != NULL;
```

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 `adj` is an  $n \times n$  array of booleans such that `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 `adj[i][j]` is true, then `adj[j][i]` should also be true, and if `adj[i][j]` is false, then `adj[j][i]` should also be false. The graph should not have any self-loops (i.e. a vertex with an edge to itself).

(1.a) Complete the data structure invariant function `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.

(1.b) Complete the `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 `bool*`, then for each array element, have it point to a new 1D array of type `bool`. You can then access the array using the 2D notation (e.g. `G->adj[0][1] = 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.

(1.c) Complete the `graph_hasedge` and `graph_addedge` functions, which should both run in constant time.

(1.d) Complete the `graph_free` function that frees any dynamically-allocated memory for the given graph `G`.

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

```
% make graphtest
% ./graphtest
```

All tests should pass. (Look at the graphs in `graph-test2.c` to see why.) Be sure to use `valgrind` also to make sure you have freed all memory you allocated!

(1.e) Write a function `fully_connected(G)` in `graph-search.c` that returns true if a graph `G` is fully connected (i.e. there is a path from any vertex to any other vertex), false otherwise.

**Hint:** Perform a BFS and count the number of vertices visited. For a fully connected graph, the total should be a specific value. Test your function on several graphs, fully connected and not fully connected.

(1.f) Update the `graph-search.h` so that the interface includes your new function `fully_connected`.

(1.g) Write at least two test cases in `graph-test.c`: one where `fully_connected` returns true, and one where it returns false.