15-122: Principles of Imperative Computation, Fall 2014
Lab 14: Graphs in C

Tom Cortina(tcortina@cs) and Rob Simmons(rjsimmon@cs)

Monday, December 1, 2014

For this lab, you will show your TA your answers as you complete each activity. Autolab is not used for this lab. You should be able to complete exercise 4 (hopefully with no memory leaks) for full lab credit.

Make a directory lab14 in your private 15122 directory and copy the required files to your lab14 directory:

```
cd $HOME/private/15122
cp -R /afs/andrew.cmu.edu/usr9/tcortina/public/15122-f14/lab14 lab14
```

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 graph2.h):

```
typedef unsigned int vertex;
typedef struct graph_header* graph;

graph graph_new(unsigned int numvert); // New graph with numvert vertices
void graph_free(graph G);
unsigned int graph_size(graph G);      // Number of vertices in the graph
bool graph_hasedge(graph G, vertex v, vertex w);
  //@requires v < graph_size(G) && w < graph_size(G);
void graph_addedge(graph G, vertex v, vertex w);
  //@requires v < graph_size(G) && w < graph_size(G);
  //@requires !graph_hasedge(G, v, w);
```

1 The adjacency matrix implementation of undirected graphs

You are given an incomplete file graph2.c that should implement the graph interface in graph2.h using an adjacency matrix. Recall that if a graph has n vertices, then its adjacency matrix adj is an n X 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).

Exercise 1. 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.

Exercise 2. 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: 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.)

Also complete the `graph_free` function that frees any dynamically-allocated memory for the given graph G.

Exercise 3. Complete the `graph_hasedge` and `graph_addedge` functions given their specifications. Remember that the graph that we are implementing is an undirected graph.

Exercise 4. Once you are done implementing the functions above, you should have a complete `graph2.c`. Compile your code and test it with the given DFS and BFS searches in `graph-search2.c` and the given graphs in `graph-test2.c`:

```
gcc -Wall -Wextra -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c *.c
```

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!

Exercise 5. Write at least two additional tests for each of the given graphs that tests your graph implementation further. Compile and run again to make sure all tests pass as expected.

2 Connected graphs

Exercise 6. Write a function `fully_connected(G)` in `graph-search2.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.) Be sure to update `graph-search2.h` and write your test code in `graph-test2.c`.