15-122: Principles of Imperative Computation — Spring 2024

Lab 14: Spend Some Cycles Thinking — Tuesday April 16th

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:

```
% 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:

```
% 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`):

```
typedef unsigned int vertex;
// typedef ______* graph_t;
// typedef ______* neighbors_t;

// New graph with v vertices
graph graph_new(unsigned int v);
//@ensures \result != NULL;

void graph_free(graph G);
//@requires G != NULL;

unsigned int graph_size(graph G);
//@requires G != NULL;

bool graph_hasedge(graph G,
                   vertex v,
                   vertex w);
//@requires G != NULL;
//@requires v < graph_size(G);
//@requires w < graph_size(G);
```

```
void graph_addedge(graph G, vertex v, vertex w);
//@requires G != NULL;
//@requires v != w;
//@requires v < graph_size(G);
//@ensures w < graph_size(G);
//@requires !graph_hasedge(G, v, w);

neighbors_t graph_get_neighbors(graph_t G, vertex v);
//@requires G != NULL && v < graph_size(G);
//@ensures \result != NULL;

bool graph_hasmore_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;

vertex graph_next_neighbor(neighbors_t nbors);
//@requires nbors  != NULL;
//@requires graph_hasmore_neighbors(nbors);

void graph_free_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;
```

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 `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).

(2.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.

1.5pt

(2.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.

(2.c) Complete the functions `graph_hasedge` that checks if an edge is in the graph and `graph_addedge` that adds a new edge to the graph.

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

The functions `graph_get_neighbors`, `graph_hasmore_neighbors`, `graph_next_neighbor` and `graph_free_neighbors` have been pre-implemented for you at the very bottom of file `graph.c`, but for an extra challenge write them yourself.

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`:

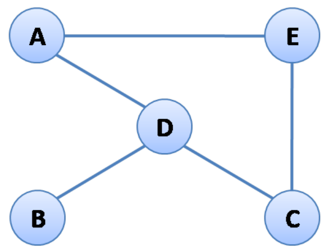
```
% make graphtest
% ./graphtest
```

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

3pt

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:



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.

4pt

(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.