Assignment 3 Documentation (assignment by Andrew Crowell - 0545826)

Graph ADT:

Definition:
A list of vertices, each with their own list of edges representing adjacency between the vertices (adjacency list), where each edge can have a weight, and the graph may permit edges to be directed or undirected.

Operations:

* GraphNew
  Creates a new empty graph.
  PRE: none
  POST: the new graph is allocated and returned.
  ERRORS: creation failed
* GraphFree
  Disposes of a graph when no longer needed by the user, freeing all its vertices, which in turn free their edges.
  PRE: Graph was not already freed, pointer is valid graph reference.
  POST: the graph reference is destroyed. Any further graph operations are undefined.
  ERRORS: none
* GraphInsertVertex
  Inserts a new vertex into a graph with a specificied name label for the vertex.
  PRE: name must be unique and should be a single non-empty word (that is, no spaces or special character).
  POST: the new vertex is placed into the graph
  ERRORS: the graph already has a vertex by the same name,
  vertex name is empty
* GraphRemoveVertex
  Given a name, removes a vertex by that name from the graph, and with it, all associated edges.
  PRE: vertex by given name exists
  POST: the vertex is gone from the graph, and the edges between it and other vertices disappear.
  ERRORS: the vertex is not in the graph
* GraphInsertEdge
  Given the names of two vertices, creates an edge between the two vertices in the graph.
  PRE: vertices by both given names exist
  POST: If it's a directed, the edge goes from the first vertex to the second vertex.
  If it's undirected, the edge goes both ways between the first vertex and the second vertex
  ERRORS: the graph already has an edge between the two given vertices
* GraphRemoveEdge
  Given the names of two vertices, removes any edge between the two vertices in the graph.
  PRE: vertices by both given names exist
  POST: The edge is destroyed, meaning the two nodes are no longer adjacent.
  ERRORS: the graph doesn't have an edge between the two given vertices
* GraphGetVertexCount
  Counts the number of vertices in the graph
  PRE: none
  POST: the count of vertices is returned

ERRORS: none

* GraphWrite
Represents the graph as a string and writes it to a given stream
PRE: none
POST: representation of the graph is output on the steam.
ERRORS: none

* GraphRead
Parses the graph as a string from a given stream and converts the representation into a graph.
PRE: none
POST: representation of the graph is output on the steam.
ERRORS: none

* GraphClearEdges
Removes all edges from the graph.
PRE: none
POST: no edges exist in the graph, only vertices.
ERRORS: none

* GraphToAdjacencyMatrix
Converts the graph into an AdjacencyMatrix type, which is a two-dimensional matrix of the edges between this graph's vertices. This is useful for solving some graph problems.
PRE: none
POST: AdjacencyMatrix representation of the graph is created.
Graph MUST not be changed between converting to and from an adjacency matrix unless done with the matrix
ERRORS: none

* GraphFromAdjacencyMatrix
Parses an AdjacencyMatrix to define the edges between this graph's vertices in adjacency list form.
PRE: graph MUST not be changed between converting to and from an adjacency matrix unless done with the matrix
POST: graph has the edges that were stored in the adjaency matrix.
ERRORS: none

* GraphFindMinimumSpanningTree
Finds the minimal spanning tree for the graph using Kruskal's algorithm.
PRE: none
POST: graph is converted into its minimal spanning tree.
ERRORS: graph is directed -- MUST be undirected for algorithm to succeed.

* GraphFindTransitiveClosure
Finds the transitive closure of the graph
PRE: none
POST: graph is converted into its transitive closure
ERRORS: graph is undirected -- MUST be directed for algorithm to succeed.

* GraphFindShortestPath
Given the vertices names, finds the shortest path between two vertices on the graph.
PRE: vertices by both given names exist
POST: generates a list of moves between the source and the destination
ERRORS: path not found
vertices do not exist in graph

AdjacencyMatrix ADT:

Definition:
A two dimensional matrix of edges betwen a graph's vertices. Useful

for solving some graph problems. Cannot be created directly. Instead, they must be converted from an existing graph. The user must be cautious to not change the graph this matrix represents while using the adjacency matrix. Otherwise, converting this data back into adjacency list form will have unexpected side-effects.

Operations:

* AdjacencyMatrixFree
  Frees the adjacency matrix
  PRE: none
  POST: The adjacent matrix is freed
  ERRORS: none
* AdjacencyMatrixGetSize
  Gets the adjacency matrix's vertex count. The matrix is NxN, so it is the same size in both directions.
  PRE: none
  POST: the size of the matrix N is returned
  ERRORS: none
* AdjacencyMatrixClearEdges
  Wipes all edges from the matrix
  PRE: none
  POST: matrix is completely populated with no-edge markers.
  ERRORS: none
* AdjacencyMatrixGetEdge
  Gets the weight of the edge between two numerically indexed vertices, or 0 if there is no edge.
  PRE: none
  POST: the weight is given between the vertices
  ERRORS: none
* AdjacencyMatrixSetEdge
  Sets the weight of the edge between two numerically indexed vertices.
  Setting a weight of 0 is considered as a removal of an edge.
  PRE: none
  POST: the weight is changed between the vertices
  ERRORS: none
* AdjacencyMatrixGetEdgeCount
  Gets the number of edges that are associated with a given numerically indexed vertex
  PRE: none
  POST: gives the edge count for the vertex
  ERRORS: none

How I Designed My Graph

My graph implementation stores a List* of Vertex*, which in turn store a List* of Edge*. Each edge has a start Vertex* and end Vertex*. On an undirected graph, the edges are actually stored as directed internally, it's just that it creates an edge in both directions upon insertion. The nature of my implementation uses an adjacency list. This made it much easier to dynamically insert vertices and edges, unlike a matrix which needs to resize itself each iteration or have a fixed max vertex count. However, there were instances where it was beneficial to use adjacency matrices instead, which is why I also defined operations that allow you to convert to and from a matrix. These were especially useful in pulling off my implementation of Kruskall's algorithm.

The Road Design Program

In a file named P1, I placed the following data:

```
6 0
0 Rockville 1:121 2:98 3:204 4:186 5:60
1 Swamptown 2:149 3:78 4:216 5:80
2 Lakeland 3:156 4:74 5:197
3 Treeville 4:250 5:68
4 Forestlake 5:236
5 Swampyrock
```

I then ran ./Road, which output the minimum spanning tree for P1:

```
overkill@overkill:~/CIS/CIS2520/A4$ ./Road
Please enter the file to read from: P1
6 0
0 Rockville 5:60
1 Swamptown 3:78
2 Lakeland 4:74
3 Treeville 1:78 5:68
4 Forestlake 2:74
5 Swampyrock 0:60 3:68
```

Compiler Analysis Program

In a file named P2, I placed the following data:

```
6 1
0 if_i_greater_than_zero 1:1 4:1
1 set_a_sub_i_equal_to_zero 2:1
2 decrement_i 3:1
3 jmp 0:1
4 param_a 5:1
5 call_foo
```

I then ran ./Analysis, which output the transitive closure for P2:

```
overkill@overkill:~/CIS/CIS2520/A4$ ./Analysis
Please enter the file to read from: P2
6 1
0 if_i_greater_than_zero 0:1 1:1 2:1 3:1 4:1 5:1
1 set_a_sub_i_equal_to_zero 0:1 1:1 2:1 3:1 4:1 5:1
2 decrement_i 0:1 1:1 2:1 3:1 4:1 5:1
3 jmp 0:1 1:1 2:1 3:1 4:1 5:1
4 param_a 5:1
5 call_foo
```

Travel Planner Program

In a file named P3, I placed the following data:

```
14 0
0 YYZ 1:505 3:687 5:307 8:306
1 YEG 0:505 2:402 3:722 7:654
2 YYC 1:402 3:520 7:191
3 YVR 0:687 1:722 2:520 4:322
4 YXS 3:322
5 YWG 0:307 6:243 7:156
6 YXE 5:243 8:176
7 YQR 1:654 2:191 5:156
8 YUL 0:306 6:176 9:323 10:426
9 YQB 8:323 11:736 13:519
10 YHZ 8:426 11:239
11 YYT 9:736 10:239 12:572
12 YYG 11:572 13:400
13 YSJ 9:519 12:400
```

I then ran ./Travel, which after prompting me for the source and destination, found the shortest path between YYZ and YSJ:

```
overkill@overkill:~/CIS/CIS2520/A4$ ./Travel
Please enter the file to read from: P3
14 0
0 YYZ 1:505 3:687 5:307 8:306
1 YEG 0:505 2:402 3:722 7:654
2 YYC 1:402 3:520 7:191
3 YVR 0:687 1:722 2:520 4:322
4 YXS 3:322
5 YWG 0:307 6:243 7:156
6 YXE 5:243 8:176
7 YQR 1:654 2:191 5:156
8 YUL 0:306 6:176 9:323 10:426
9 YQB 8:323 11:736 13:519
10 YHZ 8:426 11:239
11 YYT 9:736 10:239 12:572
12 YYG 11:572 13:400
13 YSJ 9:519 12:400
Name of source airport: YYZ
Name of destination airport: YSJ
Route found:
YYZ
YUL
YQB
YSJ
```

How I Approached The Problems

I decided that as I was approaching each of the three graph problems assigned, it was best if the operations were defined on the ADT itself. Some of the algorithms seemed to be far easier to wrap
my head around when I had an adjacency matrix, so I made conversion mechanisms for my graph ADT and used them in such a way that the adjacency matrix conversion is unapparent to the caller of the
problem operations. The main reason why I chose ADT operations, is it saves the end-user of the interface from having to define their own algorithms. So if more programs needed this ADT, it

would be trivial to call the builtin Graph library functions rather than create a new shortest path finding algorithm each time.

Using Kruskall's algorithm given in the lecture slides, I developed an operation to find the minimum spanning tree of a graph ADT. I pulled off an implementation not too different from the pseudo-code given. Using the transitive closure pseudocode given in class, I came up with the graph ADT operation in C. Using the Dijkstra's algorithm pseudocode given in class, I came up with a way to calculate distances between all the nodes in the graph ADT, and I stored the closest neighbours between each node so that a shortest path could be discovered and transformed into a list of movements.

Testing Methodology

My ineffably awesome testing methodology consisted of trying out every operation that the ADT implemented. If it had a pre condition, I tested the "working" pre cases, and then I tried to manipulate function inputs to "break" the code by not obeying the preconditions. I examined the ADT before the operation and after the operation, and if it did not meet the expected postconditions, I knew that the ADT needed some error returns. These error returns were then checked for in my operation tests with GraphMenu, I appropriately handled the error codes.

I used printf() to debug code blocks to ensure that all conditional blocks were being properly executed. I was relentlessly trying to break my code with a variety of vicious data. It tests for pretty much ALL forms of idiocy. Some certain pre conditions are made to be smart assumptions, though. For instance, I assume you're using this for fairly small storage, such that your machine will not explode and there's some things that just CAN'T be tested in C, like ensuring all pointers are valid. So these tests assume that the end user is slighly brain-dead, but at least is smart enough to use an interface by mostly following its preconditions.