ASSESSED INDIVIDUAL WORK

2021 PROGRAMMING PORTFOLIO
COMBINED EXERCISE

Steven R. Bagley and Jamie P. Twycross

Submission Deadline:
15/01/2022
15:00

Version: 1.00

This coursework is based around three things:

- A graph library that can store and process graphs.
- An implementation of Dijkstra's algorithm as presented in COMP1006
- A network server that stores information about a network of networks, and that can be queried to find the next-hop for a packet being sent to a specified network.

For COMP1006, you will implement Dijkstra's algorithm in C using the graph library to store the network of networks. Over in COMP1007, you will develop a C program which implements the network server using TCP/IP, the graph library to store the network of networks, and the implementation of Dijkstra's algorithm to find the next-hop. And finally, in COMP1005 you'll implement the graph library itself, using linked lists.

Precompiled and ready to use implementations of both Dijkstra's algorithm and the graph library are provided in the git repository enabling you to complete each module's section without implementing the other parts. Details of how to use the functions in the Dijkstra and Graph libraries can be found in the Appendix.

To start this coursework, you will need to fork and clone the git repository below in the usual fashion:

`https://projects.cs.nott.ac.uk/2021-COMP1006/2021_pp_combined`

Good Luck!

Assessment Notes

Details of how each section is assessed are given at the end of each section. The pipeline will mark your work and give you a separate mark *for each of the three modules.*

COMP1005: Graph Library

This part of the coursework focuses on implementing a graph library using linked lists. There is one task to complete, details of which are given below. Links to external pages are provided in the details below which you will need to follow and read to complete the task.

Overview

In this task you will implement a number of library functions. An executable program is provided which will include and use the library functions which you implement. You will not need to add any files to your forked repository for this task, as all the required files are present. You will add function implementations to certain source files in your repository. A `Makefile` is also provided to build and test your library functions. All code will be compiled with the `gcc -ansi -pedantic-errors` switches.

Assessment

This coursework is worth 20% of your final COMP1005 grade. The points awarded for Task 1 are as follows:

	Compilation	Implementation	Execution	Bonus	Total
Task 1	2	3	10	5	**20**

For Task 1, you are awarded two points if your program compiles correctly: one point if it compiles without errors, and a further one point if it compiles without errors and warnings. If your program follows the implementation instructions given in the task details below, you are awarded three points.

If your program is implemented so that no memory leaks occur when it is executed, you are awarded *five* points. If your program produces the correct output, you are awarded a further five points. Additionally, if you implement COMP1006 Task 2 (Dijkstra's algorithm) using your graph library implemented in Task 1, you will be awarded an additional five bonus points. Bonus points will not be shown in the pipeline.

The contribution towards your final COMP1005 grade is calculated as follows:

```
floor(20 * task1_points_awarded / 20)
```

Your provisional score for each task can be viewed on GitLab after every push to `projects.cs.nott.ac.uk`. Your final score will be based on the code in the last commit pushed to `projects.cs.nott.ac.uk` before the coursework deadline. Commits pushed after the coursework deadline will be disregarded. After the coursework deadline, your code will undergo further review and, based on this review, your provisional score may go up or down. This further review will include checks for code plagiarism and for trivial implementations e.g. implementations just containing an empty main function or clearly not written following the task

guidelines. Final scores will be published on the COMP1005 moodle pages around a week after the coursework deadline.

Your repository contains a file called `.gitlab-ci.yml`. This file is used during the assessment process and *must not* be removed or edited in any way. Any tampering with this file will result in a score of zero for this coursework.

This coursework is individual work i.e. *must be your own work* and follow the University Academic integrity and misconduct guidance[1].

Overview

Your task is to implement a graph[2] library.

A test program task1_test.c and graph library header file graph.h are provided for you. You must not remove or edit these two files in any way.

Details

Graphs are fundamental data structures in Computer Science. They are commonly implemented using a linked list using an adjacency list representation. You will use the implementation of a linked list provided to implement a directed graph library using adjacency lists.

You do not need to implement a linked list library, as a full implementation is provided in the `linked_list.h` and `linked_list.c` files. You must not edit these two files in any way. You should use this implementation when implementing your graph library.

The header file which contains the interface for an implementation of a graph library in C is given in the `graph.h` file. This file contains the structures you will use to represent a graph, vertices, and edges. You will not need to implement any further structures.

A skeleton implementation file `graph.c` which implements each of the functions declared the interface file is also provided. Your task is to complete the implementation of the functions in graph.c. You should only edit the `graph.c` file, and must not edit the `graph.h` file. You are free to implement the internals of each function in any way you want, but you must not change the function definitions in the provided `graph.c` file i.e. you must not change the function names, return types or parameters. All warnings and error messages in your implementation should be printed to stderr. Further implementation guidelines are given in the comments in `graph.c`.

A test program, `task1_test.c`, is provided which will include your implementation and use your implementation to manipulate a graph. You must not edit the file `task1_test.c` or `utility.h`. To compile your implementation and the test program, type:

```
$ make clean task1_test
```

---

[1] https://www.nottingham.ac.uk/studyingeffectively/studying/integrity/index.aspx

[2] https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

To compile your implementation and the test program and then run the test program, type:

```
$ make clean task1_test_run
```

A typical output (your warning messages may vary) from a correctly working implementation is:

```
$ make clean task1_test_run
rm -f task1_test *.o
gcc -c linked_list.c -Wall -ansi -pedantic-errors -g
gcc -c graph.c -Wall -ansi -pedantic-errors -g
gcc  -o  task1_test  task1_test.c  linked_list.o  graph.o  -Wall  -ansi
-pedantic-errors -g
./task1_test 1:3:1 1:5:2 2:3:10 2:4:2 3:4:2
Performing Test 1...
warning: unable to remove vertex
warning: unable to find vertex
warning: unable to remove edge
warning: unable to find vertex
warning: unable to remove edge
warning: unable to add vertex
warning: unable to add edge
warning: unable to add undirected edge
warning: unable to find vertex
warning: unable to print graph
warning: unable to free edge
warning: unable to free vertex
warning: unable to free graph
Completed Test 1.
Performing Test 2...
initialising graph...
adding edges...
removing graph...
Completed Test 2.
Performing Test 3...
initialising graph...
adding edges...
removing edges...
removing graph...
Completed Test 3.
Performing Test 4...
initialising graph...
adding edges...
removing vertices...
warning: unable to remove vertex
warning: unable to remove vertex
removing graph...
Completed Test 4.
Performing Test 5...
initialising graph...
adding edges...
printing graph...
1: 3[1.00] 5[2.00]
```

```
3: 1[1.00] 2[10.00] 4[2.00]
5: 1[2.00]
2: 3[10.00] 4[2.00]
4: 2[2.00] 3[2.00]
removing graph...
Completed Test 5.
Performing Test 6...
initialising graph...
adding edges...
removing edges...
adding edges...
printing graph...
1: 3[1.00] 5[2.00]
3: 1[1.00] 2[10.00] 4[2.00]
5: 1[2.00]
2: 3[10.00] 4[2.00]
4: 2[2.00] 3[2.00]
removing graph...
Completed Test 6.
Performing Test 7...
initialising graph...
adding edges...
printing graph...
1: 3[1.00] 5[2.00]
3: 1[1.00] 2[10.00] 4[2.00]
5: 1[2.00]
2: 3[10.00] 4[2.00]
4: 2[2.00] 3[2.00]
removing graph...
Completed Test 7.
```

Your implementation should handle dynamically allocated memory correctly i.e. free all dynamically allocated memory. To check your implementation using valgrind you can type:

```
$ make task1_test_memcheck
```

If your program has correctly handled dynamic memory allocation, the last line of output should read:

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

COMP1006: Dijkstra's algorithm

In Week 05 of COMP1006, we saw how Dijkstra's algorithm could be used to calculate the routing table for a network. The network of networks was represented as a graph, and a slightly modified version of Dijkstra's algorithm was then run over the graph to calculate the next-hop. The routing table contains the next-hop (i.e. which network should a packet be forwarded too to reach that destination) for each network in a network of networks.

For this section, we will represent each network by an integer ID that is greater than zero (i.e. `1` is the lowest valid network ID), and will create vertices in the graph for each network (again labelled by the integer ID). We will represent the connections between the networks by edges between the vertices, the edges being labelled with the weight of the connection (as a `double`).

You will implement Dijkstra's algorithm using the Graph Library detailed above (*if you have not yet completed your graph library implementation, then you can use the precompiled* `.o` *file supplied to complete this task*). The algorithm is implemented as a single C function:

```
Path *dijkstras(Graph *graph, int id, int *pnEntries)
```

This function takes a pointer (`graph`) to the `Graph` describing the network of networks, this will be created (see the test program supplied) by calling the various graph library functions to create the required vertices and edges. The parameter `id` specifies which network is being used as the starting point, if there is no vertex in graph with an `id` of then the function should return `NULL`.

The `dijkstra(…)` function returns the routing table as an array of `Path` structures (defined in `dijkstra.h`). This array is indexed using the network id and so you will need to allocate enough memory to allow for all valid indexes (i.e. if the highest vertex is 13, then you should be able to use 13 as valid index into the array). The integer pointer `*pnEntries` should be filled with the number of entries in the array.

Each entry in the returned array either contains the next hop (equivalent to the value from the table `R[]` in the COMP1006 notes), and weight (equivalent to the table `D[]`) for a network in `graph` (the destination network id is used as an index into this array), or contains a value to indicate that the network does not exist — in which case, the weight should be set to DBL_MAX (a constant defined in the `#included` `float.h`), and next hop to `-1`.

You should consult the aforementioned lecture notes for COMP1006 for a full discussion of how you should implement Dijkstra's algorithm, but in brief, your implementation should:

- Check that the Graph `graph`, and starting node, `id`, are valid
- Create the set ***S*** containing all the networks (vertices) expect the source node (you might want to use an array for this.
- Create an array to represent the table **D** and initialise it with the weights of the edges from the source node, or infinity if no edge exists. You should use the constant `DBL_MAX` to represent infinity.

- Create an array to represent the table **R** and initialise it with the next hops if an edge exists from the source, or 0 otherwise.
- Then repeatedly the remaining rules of Dijkstra's algorithm, updating the values in ***D*** and ***R*** until ***S*** is empty.
- Each of the values required to complete the above can be found by calling the various functions (`get_vertices()`, `get_edge()`, `edge_destination()`, `edge_weight()`, etc.)
- Once Dijkstra's algorithm has run, you will need to create the routing table to be returned by allocating enough memory for the required number of `Path` structures and filling in all entries of the array.
- Finally, you should free any unused memory you have allocated, set `*pnEntries` to the correct value, and return the pointer to the routing table you've just filled in (obviously, you shouldn't free that!)

You will not need to add any files to your forked repository for this coursework, as all the required files are present. For each task, you will add function implementations to certain source files in your repository. A `Makefile` is also provided to build and test your `dijkstra()` function.

A test program, `comp1006_test.c`, is provided which includes your implementation of the `dijkstra()` function and uses your implementation to find the shortest paths in an example graph. You must *not* edit the file `comp1006_test.c`. To compile your implementation and the test program, type:

```
$ make clean comp1006_test
```

To compile your implementation, and the test program and then run the test program, type:

```
$ make clean comp1006_test_run
```

A typical output from a correctly working implementation is:

```
$ make comp1006_test_run
./comp1006_test 3 1:3:1 1:5:2 2:3:10 2:4:2 3:4:2
Net   Weight Next Hop
1     1.00        1
3      INF       -1
5     3.00        1
2     4.00        4
4     2.00        4
```

As discussed in COMP1005, a good tool for assessing if a program has handled dynamic memory allocation correctly is `valgrind`. To check your implementation using `valgrind` you can type:

```
$ make comp1006_test_memcheck
```

If your program has correctly handled dynamic memory allocation, the last line of output should read:

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Assessment Criteria

This section is worth 33% of your final COMP1006 mark. The gitlab pipeline available for this coursework will mark your program using the following criteria. Specifically, the 33 marks will be split into 25 marks for the functionality of your Dijkstra algorithm, and 8 marks are awarded for the C implementation itself. The marks will be broken awarded as follow:

- Program correctly calculates shortest next hop for a network of two networks **[5 marks]**
- Program correctly calculates shortest next hop for each node of a fully connected network of networks (comprising more than two networks) **[5 marks]**
- Program correctly calculates shortest next hop for an arbitrary network of networks **[6 marks]**
- Program correctly shows that certain networks are unreachable if there is no route between two networks **[5 marks]**
- Program calculates the distance for each route **[4 marks]**
- Program correctly allocates and frees memory **[4 marks]**
- Program returns NULL if asked to process non-existent network **[4 marks]**

COMP1007: Route Network Server

For COMP1007, you are to write a TCP/IP server that can build up a graph of a network of networks (using the supplied graph library and implementation of Dijkstra's algorithm) and then query that graph to find out which link between two networks should be used as the next hop to send a packet of data from one network to another within the network of networks (using the supplied implementation of Dijkstra's algorithm). The next hop can be found by using Dijkstra's algorithm, as discussed in Systems and Architecture.[3] The mark scheme for this exercise (see below) has been designed so that it is still possible to obtain some marks for this section even if you have not completed the previous section.

A skeleton file, `server.c` has been provided for you and you should implement your solution to this part of the coursework within that file. This file can be built using the supplied Makefile by typing:

```
$ make server
```

This will automatically compile your server and link it against your implementations of the graph library above, provided in graph.c.

The graph is created and queried by clients connecting to the server and issuing commands to the server following the protocol outlined below. The graph created should persist between different connections to the server, but does not need to persist if the server is stopped and then restarted. The server should listen for and accept incoming connections on the TCP/IP port specified as the first (and only) argument on the command line, so if your program was called:

```
$ ./server 4242
```

where `server` is the name of your program, then your program should listen on port 4242.

To simplify things, we will represent the networks by an integer number between 1 and 255, and specify the connections between two networks by specifying the source and destination network ID numbers along with the weight (cost) of using that connection. You can assume that there is at most one link between two networks. You should represent the networks as Vertices in your graph, and the links between networks as edges.

Protocol

Your server should implement the following protocol[4], in most cases this can be done by writing C code to parse the supplied command and arguments and then calling the appropriate function in the graph library, or the implementation of Dijkstra's algorithm. Programs that do not implement all

---

[3] The implementation of Dijkstra's algorithm provided conforms to the algorithm described in `https://moodle.nottingham.ac.uk/course/view.php?id=123012#section-11`

[4] This protocol has been created solely for the purposes of this coursework, in reality it would not be useful for calculating network routes.

the functionality specified will still be able to obtain marks (see the assessment criteria below) for the functionality implemented.

The client and server communicate in ASCII (i.e readable text) with data being sent as a series of lines of text, terminated by both a carriage return (`CR`) and a line feed (`LF`).[5] You can assume that no line of text will be greater than 512 bytes long. All commands sent by the client to the server should be a single line long, while the responses from the server will be at least one line long.

The first line of any response sent out by the server should indicate whether the action has been a success or failure. In the case of success, the response should start with '`+OK`' — in the case of an error, the response should start with '`-ERR`'. Additional information may follow the '`+OK`', '`-ERR`' but this is optional **except where otherwise stated**.

When a client connects to the server, the server will respond by sending a greeting. This is a simple line of text that identifies the server. The greeting should begin with '`+OK`' since the connection has been made.

The connection between the client and the server should be closed on receipt of the QUIT command from the client. A single line response should be sent from the server to the client to indicate reception of the command before the connection is terminated by calling `close()`.

Your server should implement the following nine commands. As has been previously stated, most of these commands can be implemented by making the appropriate calls to the graph library created in part one. If your server receives any other command, it should respond with an error indicating that the command is not implemented.

Protocol Commands

`NET-ADD <integer>`
Receipt of this command from the client should cause the server to add a new network to the graph. The ID for the network is specified by the integer parameter. By default, the network is not connected to any other network.

The server should respond appropriately to indicate success after adding the network to the graph. If a network already exists with that ID, then the server should respond with an error and the graph should not be altered.

`NET-DELETE <integer>`
Receipt of this command from the client should cause the server to delete an existing network from the graph. The ID for the network is specified by the integer parameter. In addition, all edges (connections to other networks) associated with this network should also be deleted.

The server should respond appropriately to indicate success after deleting the network from the graph. If no network already exists with that ID, then the server should respond with an error and the graph should not be altered.

---

[5] CRLF can be generated in C using `\r\n`

NET-LIST

Receipt of this command should cause the server to list the IDs of all the networks that have been added to it (using `NET-ADD`). The server will always respond with a success response containing the '+OK' followed by the number of networks to be listed, e.g.:

```
+OK 6
```

would indicate that details of six networks will follow. Following this line, the network IDs for the known networks should be sent to the client, one per line. This command should never return an error response.

ROUTE-ADD <src network> <dest network> <weight>

Receipt of this command informs the server of a connection between two networks specified by their network IDs as integers. Although the two networks are specified as source and destination the link should be interpreted as being bidirectional. The weight is specified as an integer number indicating the cost of using that link.

The specified link should be added to the graph, and an appropriate success response sent, unless one of the network does not exist in which case an error should be sent. If the link already exists, then the weight should be updated to the newly specified value.

ROUTE-DELETE <src network> <dest network>

Receipt of this command from the client should cause the server to delete the specified link between two networks from the graph. The IDs for the networks are specified by the integer parameter.

The server should respond appropriately to indicate success after deleting the network from the graph. If the link does not exist, then the server should respond with an error and the graph should not be altered.

ROUTE-SHOW <network>

Receipt of this command should cause the server to list the IDs of all the networks for which a link has been added (by `ROUTE-ADD`) from the specified network. The server will respond with a success response containing the '`+OK`' followed by the number of networks to be listed, e.g.:

```
+OK 6
```

would indicate that details of six links will follow. Following this line, the network IDs for the other end of the link should be sent to the client, one per line.

This command should return an error response if the network does not exist.

ROUTE-HOP <src network> <dest network>

Receipt of this command should cause the server to query the graph and return the *next hop* where a packet should be sent to forward it from the source network to the destination network. The networks are specified using their integer IDs. Your server should run Dijkstra's algorithm on its graph and return the next-hop for the specified network, or -1 if there's no possible route between the two networks.

The program should respond indicating a success, **following** the '`+OK`' with the network ID of the next hop (i.e. the network to which this packet should be sent). An error should only be returned if either of the networks does not exist, or if the source and destination networks are identical.

`ROUTE-TABLE <network>`

Receipt of this command should cause the server to list the complete routing table for the specified network. The networks is specified using its integer IDs. Your server should run Dijkstra's algorithm on its graph and return the next-hop for every *other* network in the graph (i.e. no route is shown for the specified network).

The server will respond with a success response containing the '`+OK`' followed by the number of entries in the routing table, e.g.:

```
+OK 6
```

This should then be followed by a line for each entry of the routing table (excluding the network specified), in the following format:

`<current>` `->` **`<destination>,`** `next-hop` **`<hop>,`** `weight` **`<weight>`**

where `<current>` is the integer ID for the network being queried, and `<destination>` is the network ID of the destination network. `<hop>` is the network ID where packets should be sent to reach the `<destination>` network (this may, or may not, be the same network), while `<weight>` should be the cost of traversing that route (as calculated by Dijkstra's algorithm).

If there is no route to a network, `<hop>` should be -1 and `<weight>` should read `INF`.

Example Transaction

Outlined below is a sample transaction between the client and server. The **`C`**: and **`S`**: are not part of the transaction and have been added for clarity to show which program sent the data.

```
S: +OK 2020 Programming Portfolio Route Server
C: NET-ADD 1
S: +OK Added 1
C: NET-ADD 2
S: +OK Added 2
C: NET-ADD 3
S: +OK Added 3
C: NET-ADD 4
S: +OK Added 4
C: NET-ADD 5
S: +OK Added 5
C: ROUTE-ADD 1 5 2
S: +OK Route Added
C: ROUTE-ADD 1 3 1
S: +OK Route Added
C: ROUTE-ADD 3 2 10
S: +OK Route Added
C: ROUTE-ADD 3 4 2
S: +OK Route Added
C: ROUTE-ADD 4 2 2
```

```
S: +OK Route Added
C: THIS-COMMAND-DOES-NOT-EXIST
S: -ERR Not Implemented
C: ROUTE-TABLE 3
S: +OK 5
S: 3 -> 1, next-hop 1, weight 1
S: 3 -> 2, next-hop 4, weight 4
S: 3 -> 4, next-hop 4, weight 2
S: 3 -> 5, next-hop 1, weight 3
C: ROUTE-SHOW 5 3
S: +OK 1
S: 5 1 2
C: QUIT
S: +OK
```

Testing your server

You can test your server manually by connecting to it using the `nc` program on the school's Linux machines. `nc` lets you connect to a network server and see its output as well as type in data to be sent back to the server. Since the protocol you are implementing is ASCII based, `nc` is a natural way to test how it works. Assuming that your server is running on the school's Linux systems, open a second terminal window and at the prompt type the following command:

`nc -C localhost <port>`

where `<port>` is the number (between 1024 and 65536) that you specified as an argument when you ran your server. You should be connected to your server and (hopefully) see your server's greeting displayed on screen. You can then simply type sample commands listed in the protocol to test that they produce the output you expect.

a B

Assessment Criteria

This section is worth 33% of your final COMP1006 mark. The gitlab pipeline available will mark your program and give you feedback on your implementation, specifically, 33 marks will be awarded as follows:

- Program correctly opens a TCP socket on the port specified on the commands line that a client can connect to **[4 Marks]**
- Server responds with appropriate greeting when a client connects **[3 Marks]**
- Server responds to the `QUIT` command by sending appropriate response **[3 Marks]**
- Server responds to the `QUIT` command by closing its side of the TCP/IP connection **[4 marks]**
- Server responds to unimplemented commands as per protocol defined above **[2 marks]**
- Server implements some commands **[2 marks]**
- Server implements all commands **[3 marks]**

- Server implements the protocol above according to the specification, broken down as follows:
  - Server will correctly add, delete and list nodes **[3 marks]**
  - Server will correctly add, delete and list routes between nodes **[3 marks]**
  - Server returns plausible responses for next hop **[3 marks]**
  - Network of Networks persists in program between connections to the server **[3 marks]**

Appendix

Graph Library

Listed below are the functions that form part of the Graph Library, the file `graph.o` contained within the repository contains a full-working implementation of the functions below that you can use to implement Dijkstra's algorithm and the TCP/IP server. These are identical to functions as you need to implement for COMP1005.

The library is centred around three structures: Graph, Vertex and Edge. You do not need to know the internal format of these structures, instead you will pass pointers to and from the functions below.

`Graph *init_graph(void)`
This function creates a new graph returns a pointer to the `Graph` structure. The graph should be freed by calling free_graph() when it is no longer needed.

`void free_graph(Graph *graph)`
This function frees the memory allocated for `graph` along with any associated `Vertex`es and `Edge`s. This should be called when you have finished using the graph.

void print_graph(Graph *);

`Vertex *add_vertex(Graph *graph , int id)`

This function creates a new Vertex to the graph with the specified id and adds it to graph. Returns the pointer to the vertex or NULL if an error occurs. If a vertex with id already exists, it returns the pointer to the existing Vertex.

`Vertex *find_vertex(Graph *graph, int id)`

This function returns a pointer to the `Vertex` with the specified `id`, or `NULL` if the vertex does not exists.

`void remove_vertex(Graph *, int id)`
Deletes the vertex with the specified id from the graph (all edges connected to this vertex are also removed).

`int *get_vertices(Graph *graph, int *count)`
Returns a pointer to an array of integers containing the ids for the vertices added to the graph. The caller of this function is responsible for `free()`-ing this pointer.

`Edge *add_edge(Graph *graph, int from, int to, double weight)`
Add a directed edge with the specified weight between the vertex `from` to the vertex `to`. If no vertices with the specified id exists, then they are created. Returns a pointer to the created `Edge`.

`void add_edge_undirected(Graph *graph, int from, int to, double weight)`
Add an undirected edge with the specified weight between the vertex `from` to the vertex `to`. If no vertices with the specified id exists, then they are created. Returns a pointer to the created `Edge`.

`Edge *get_edge(Graph *graph, int from, int to)`
Returns a pointer to the Edge from the vertex with id from to the Vertex from id to, or NULL if no edge exists

`Edge **get_edges(Graph *graph, Vertex *vertex, int *count)`
Returns a pointer to an array of pointers containing the `Edge`s for a given vertex. The array of `Edge`s is dynamically allocated and the caller of this function is responsible for `free()`-ing this pointer. The integer pointed to by `count` will be filled in with the number of edges in the array.

`Edge *get_edge(Graph *graph, int from, int to)`
Returns a pointer to an Edge from `from` to `to`, or `NULL` if no edge is present within the graph.

`int edge_destination(Edge *edge)`
Returns the id of the destination in the `Edge` pointed to by `edge`

`double edge_weight(Edge *edge)`
Returns the weight associated with the `Edge` pointed to by `edge`

`void remove_edge(Graph *graph, int from, int to)`
Removes an edge (if present) from the graph between vertex `from` and vertex `to`.

Dijkstra's Library

Listed below is the dijkstra function that implements Dijkstra's algorithm using the above Graph Library.

`Path *dijkstras(Graph *graph, int id, int *pnEntries)`
This function takes a pointer (`graph`) to the `Graph` describing the network of networks, and returns the routing table (as an array of `Path` structures). The parameter `id` specifies which network is being used as the starting point, if there is no vertex in graph with an id of `id` then the function returns `NULL`. The integer pointer `*pnEntries` should be filled with the number of entries in the array, and the caller is responsible for freeing the array.

The returned array is indexed using the destination network id, and the Path structure is defined below, where weight contains the distance to the destination network and next-hop contains the `id` of the network should be sent to, should a route exist. If no route is available to a destination id then weight will be DBL_MAX and next-hop will be -1.