UMEÅ UNIVERSITY
Department of Computing Science
Assignment 5 report

June 5, 2020

5DV149 - Assignment 5, v1.0

Data Structures and Algorithms (C) Spring 2020, 7.5 Credits

Graph Implementation and Algorithm

Name	Elias Olofsson
user@cs.umu.se	`tfy17eon@cs.umu.se`

Teacher
Anna Jonsson

Graders
Anna Jonsson, Lena Kallin Westin, Hannah Devinney

Contents

1 Introduction

In many fields of study, it can be beneficial to represent certain types of structures and processes as a *graph*. Whenever there is a set of objects and there exists pairwise relations between each or some of the objects, it can most likely be mathematically seen as a graph. A network of roads connecting cities, the communication and exchange of information over the internet and certain interactions among molecules creating proteins in your body are all examples of structures and processes that can be modeled as graphs. In short, being able to represent large and complex problems from the real world as a mathematical graphs and having algorithms to analyze and extract information out of mentioned graphs is an important addition to the toolbox of any aspiring scientist or engineer.

In this lab we take a look at the abstract datatype *graph* and a way to implement it in the programming language C. Furthermore, a program had to be designed around the implemented datatype that should be able to read a file containing the structure of a directed graph and to interactively answer whether or not there is a path from one specified place in the graph to another. Using the proper mathematical terms, a graph is a set of vertices or *nodes* connected by *edges*. Each node has a unique name or label, and its relationship to a neighbouring node is characterized by an edge. In this lab we will only deal with unweighted directed graphs, which means that all edges are considered equally and without bias, and that the relationship between any two nodes might be asymmetric and thus directional.

The aim of this lab is to broaden our ability to use existing datatype implementations while implementing other, higher level data structures. To implement the datatype *graph* and some corresponding graph algorithms, we will gain practice in handling large problems through sectioning work into small and manageable pieces, while continuously testing and ensuring that the overall design has a high quality and functions as per specification. Furthermore, the ability to correctly document a program in an informative way is tested and enhanced.

2 User's manual

This section aims to give an uninformed user the necessary information required to correctly use the program and achieve satisfying results. No knowledge of the underlying structures and algorithms used within the implementation are needed to sufficiently operate the program.

2.1 General program description

The program `is_connected.c` is an interactive program to find out if two arbitrary nodes within a given directed graph are connected or not. In an arbitrary graph, two nodes are defined as connected if one is able to move from one node to the other by traversing the graph, thus by successively moving from node to node along the existing edges. Since the graph is of a directed type, where the edges within the graph are unidirectional and thus can only be traversed in one way, the user will need to specify both the starting node and the desired end node. Once two valid nodes are entered by the user, the program will traverse the graph from the starting node looking for the end node. If the program could find the end node through traversal of the graph, the two nodes are connected and the successful result will be prompted to the user. If a complete traversal has been concluded without reaching the desired end node, the two nodes are not connected and the unsuccessful result is communicated to the user.

At program execution, an additional file containing information about the graph layout needs to be passed along to the program. Requirements on how this map file should be organized and formatted are detailed below, see sec.(2.3). If the map file is valid and could successfully be read by the program, the user will be prompted to insert two nodes. When two valid nodes have been entered by the user, the program searches the graph and delivers the result to the user, whereupon the program again prompts the user two enter two nodes. The user can either choose to continue indefinitely by conducting more searches between any other set of two nodes, or exit the program.

2.2 Compilation

In order to correctly build an executable, we need to include the right supporting data structures when compiling the program `is_connected.c`. In both the main program and the underlying implementation of the abstract datatype *graph* within `graph.c`, other data structures have been used from a code base supplied by the Department of Computing Science at Umeå University. The most current and up-to-date version of this code base at the time of writing was *datastructures-v1.0.8.2*.

To correctly build the program, one needs to not only include `graph.c` which was created within this lab, but also `queue.c`, `list.c`, `dlist.c` and `array_1d.c` from the code base. Furthermore, all the corresponding header files for each data structure implementation need to be included as well, all of which can be found within the code base, with the only exception

being `graph.h` which was supplied within this lab. Using the GNU Compiler Collection on Linux and assuming that all the required files are in the current working directory and that the header files exists in an `include` directory one level up in the folder structure, one example of the terminal command to compile the program is shown below. After such a call or one similar to this, the compiler will build an executable program named `is_connected` that the user can run later at will.

```
gcc -std=c99 -Wall -I../include/ is_connected.c graph.c
queue.c list.c dlist.c array_1d.c -o is_connected
```

2.3 Data input

At program execution, the executable `is_connected` needs to be supplied with a complimentary file containing the layout and structure of a directed graph. One example of a valid map file, originally conceived by the Department of Computing Science at Umeå University, is the file `1-airmap1.map` shown below.

1-airmap1.map

```
# Some airline network
8
UME BMA # Umea-Bromma
BMA UME # Bromma-Umea
BMA MMX # Bromma-Malmo
MMX BMA # Malmo-Bromma
BMA GOT # Bromma-Goteborg
GOT BMA # Goteborg-Bromma
LLA PJA # Lulea-Pajala
PJA LLA # Pajala-Lulea
```

The requirements of any map file is that it should on the first non-comment and non-blank line have an integer representing the number of edges existing within the graph. On the following lines, the edges may be declared individually by stating the label for the source node followed by the label for the destination node, specifically in that order. Only a single edge per line is allowed and there should not be any duplicate edges within the file. Each node name must be of alphanumeric type with a max length of 40 characters and each pair of node labels must be separated by white-space on every line. Any line may also contain additional white-space at the beginning and/or

at the end which should not be considered. Furthermore, any line where the first non-white-space character is a number sign (#) is a comment line and should not be considered. Comments may also occur after the node labels on any line and should similarly also be ignored. The file may be either of DOS or UNIX type and contain the corresponding newline control characters. The very last line of the file may or may not have a newline control character.

Assuming that the program on execution is given a valid map file containing the structure of a directed graph, the user will then get prompted to input two nodes. The user should type in two valid node labels with the keyboard, separated by white-space, where the first node is the starting node and the second is the desired end node. After pressing enter, the program will conduct its search and present the result, whereupon the program repeats and prompts the user once more. Should the user desire to exit the program, then it can be achieved by typing `quit` and pressing enter.

2.4 Expected output

Once an executable has been correctly compiled and a valid map file constructed, the program is ready to be run. Assuming execution from a Linux terminal and where both the executable `is_connected` and the map file `1-airmap1.map` are existing within the current working directory, one example on how the terminal output could look like at runtime may be seen below.

```
user@hostname:~$ ./is_connected 1-airmap1.map
Enter origin and destination (quit to exit):  BMA GOT
There is a path from BMA to GOT.

Enter origin and destination (quit to exit):  PJA UME
There is no path from PJA to UME.

Enter origin and destination (quit to exit):  MMX UME
There is a path from MMX to UME.

Enter origin and destination (quit to exit):  quit
Normal exit.
```

Here the map file is passed along directly behind the executable as one additional argument, which thus makes it really easy to quickly change and apply the same program to other graphs, without having to modify the

code of the program. The user gets repeatedly prompted to enter two nodes within the graph and thereafter answered whether or not there is a path from the starting node to the destination node. The user finally concludes by exiting the program through the keyword `quit`.

3 Description of the system

This section details design choices and representations made during the implementation of the program and the underlying data structures within this lab.

3.1 Implementation of the graph datatype

Central to the program `is_connected.c` is the implementation of the abstract datatype *graph* within `graph.c`. Under the surface of the datatype interface, the graph has been represented as an array of nodes. More accurately, the array actually stores pointers to `struct`:s, where each `struct` contains the information corresponding to a single node. For each node, the `struct` members are; a pointer to a unique label by which the node can be identified, a pointer to a directed list containing all the neighbouring nodes to which the node is connecting to, and lastly a flag to allow for the correct implementation of various graph algorithms. The flag is an identifier for whether or not the node has been "seen" before, thus whether it has been visited already during a traversal of the graph. This is an important addition to avoid unforeseen loops during traversal.

The complete and exhaustive interface on how the abstract datatype *graph* can be implemented can be seen in the list of declarations within the header file `graph.h`. Not all of these functions have actually been implemented within `graph.c`, instead only the functions deemed necessary to fulfill the expected functionality of the program `is_connected.c` have been realized. The handful of functions in the interface which were selected to be implemented in `graph.c` are listed below.

implemented datatype graph.c

```
graph_empty           (int max_nodes)                    → graph *
graph_insert_node     (graph *g, const char *s)          → graph *
graph_find_node       (const graph *g, const char *s)    → node *
graph_node_is_seen    (const graph *g, const node *n)    → bool
graph_node_set_seen   (graph *g, node *n, bool seen)     → graph *
graph_reset_seen      (graph *g)                         → graph *
graph_insert_edge     (graph *g, node *n1, node *n2)     → graph *
graph_neighbours      (const graph *g, const node *n)    → dlist *
graph_kill            (graph *g)                         → void
nodes_are_equal       (const node *n1, const node *n2)   → bool
```

With this selection of interface operations, an external program can create an empty graph structure with the function `graph_empty()` and insert all required nodes with `graph_insert_node()`. The function `graph_find_node()` will obtain the node pointers corresponding to any of the inserted nodes, which then can be used in conjunction with functions `graph_node_is_seen()`, `graph_node_set_seen()` and `graph_insert_edge()` to either inspect or set the seen status of any node, or place an edge in between any two nodes. For more efficient setting of the seen status to the "unseen" mode, the function `graph_reset_seen()` can be used to reset all nodes in the graph in a single function call. Furthermore, having acquired a node pointer to any of the nodes within the graph, the function `graph_neighbours()` can be used to get a directed list with pointers to all the node neighbours that the current node is connected to. When the life of the graph structure has come to an end and the user has chosen to exit the program, a call to the function `graph_kill()` is necessary in order to free any memory that was dynamically allocated to hold the graph structure. The last, but very important function within the interface of `graph.c` is the node comparison function `nodes_are_equal()`, whose responsibility it is to determine whether two nodes are in fact equal or not. Such a function is essential, since comparing the pointers to nodes in a graph is not the same as comparing nodes within the graph, and as such external comparison of pointers might bring undefined or incorrect behaviour to the program. Since the node labels for each node are stored in plain-text as null-terminated strings, we have implemented the node comparison function `nodes_are_equal()` as a string comparison function, utilizing the `strcmp()` function defined within the standard library `string.h`.

With this set of interface operations we are set for efficiently implementing the program `is_connected.c` as per specifications. Since we trust that the graph implementation has been done correctly, so a newly created graph will not contain any nodes or edges, and the graph structure, once built from the file, is going to remain static during runtime until a program exit, we will not need any functions to check if the graph is empty or has edges, or to remove single nodes or edges. These requirements eliminates the need for functions `graph_is_empty()`, `graph_has_edges()`, `graph_delete_node()` and `graph_delete_edge()` seen in the header file `graph.h`. Since we always rely on the user to select a node within the graph, we likewise do not need the function `graph_choose_node()`, also declared in the aforementioned header file, which picks out one arbitrary node existing in the graph. Finally, I chose not to implement the last function in the header file `graph_print()`, which

prints the contents of the graph structure, simply because it was not required and since there is other more efficient ways of debugging the code during creation than implementing this function.

3.2 Supplementary data structures

In the creation of the main program `is_connected.c` and the graph datatype implementation `graph.c`, other supporting datatypes have been used in the underlying structure. All external data structures used have been from the same aforementioned code base *datastructures-v1.0.8.2*, supplied by the Department of Computing Science, Umeå University.

Central for the graph datatype implementation was the datatypes `array_1d.c` and `dlist.c`, both given from the code base. The one-dimensional array data structure was used to store multiple pointers to `struct`:s containing the relevant information for each node. Following any of the pointers in the array elements, one could access the flag of the current node's seen status, a pointer to the label of the node in question and a pointer to a directed list containing the node's neighbours. The labels of all the nodes were simply stored fully in plaintext, as a null-terminated strings of characters, which could be reached via the corresponding pointer in the corresponding node `struct`. The operations in the interface of `array_1d.c` necessary to get the required functionality of the main program is the selection of functions seen below.

implemented datatype array_1d.c

```
array_1d_create (int lo, int hi, free_function free_func) → array_1d *
array_1d_inspect_value (const array_1d *a, int i) → void *
array_1d_has_value (const array_1d *a, int i) → bool
array_1d_set_value (array_1d *a, void *v, int i) → void
array_1d_kill (array_1d *a) → void
```

With this set of operations, one can create an empty array and insert element values in a controllable way. Existence of an element value can be verified such as not to overwrite values or traverse an array past the region of interest. Once the life of the array is up and the user has chosen to exit the program, appropriate freeing of memory dynamically allocated for the array can be done in a controlled way.

The next part of the implementation of the graph data structure is the usage of directed lists through `dlist.c` to host the neighbouring nodes for a given node. In the aforementioned `struct`:s pointed to from the array, there is one additional pointer stored that directs to a list of `dlist` type. This directed list contains the names of the node's neighbours, stored as null-terminated strings. When an external program wants to find out to which nodes an arbitrary node is connecting to, a call to the function `graph_neighbours()` in the interface of the graph datatype is made. Even though the directed list of neighbours we have stored for each node is of the right format for the expected return from the function `graph_neighbours()`,

it would not be appropriate to directly return a pointer to any of the lists stored within the graph structure itself. This is because it may bring undefined behaviour when a user might feel tempted to modify the directed list directly through the interface of `dlist.c`, and not through the sanctioned interface of `graph.c`. Thus the structure of the graph might change without any calls to the interface of the graph datatype, which is not desirable. The solution is to make a separate `dlist` copy of the neighbour list whenever the function `graph_neighbours()` is called, which is achieved through the internal function `clone_dlist()` within `graph.c` in conjunction with `copy_string()`, which makes a new dynamical copy of the input list and all strings within it. As such, all the required operations from the interface of `dlist.c` necessary to get the specified functionality of the main program are listed below.

implemented datatype dlist.c

```
dlist_empty     (free_function free_func)                → dlist *
dlist_first     (const dlist *l)                         → dlist_pos
dlist_next      (const dlist *l, const dlist_pos p)      → dlist_pos
dlist_is_end    (const dlist *l, const dlist_pos p)      → bool
dlist_inspect   (const dlist *l, const dlist_pos p)      → void *
dlist_insert    (dlist *l, void *v, const dlist_pos p)   → dlist_pos
dlist_remove    (dlist *l, const dlist_pos p)            → dlist_pos
dlist_kill      (dlist *l)                               → void
```

With this set of functions, directed lists can be created and controllably built according to specification. The lists can be traversed efficiently, new elements can be inserted and consequently inspected. When the life of the list is up and the program is exiting, memory dynamically allocated for the lists can be freed appropriately.

In the main program `is_connected.c`, other supporting data structures have been used to get the required functionality. At the center of the solution, the implementation of the datatype queue in `queue.c`, given from the code base, is the essential component for searching the graph structure. When two nodes have been entered by the user at runtime, the program performs a "breadth-first" traversal of the graph from the starting node in search for the destination node. During such a traversal, a queue is the vital component to keep track of and order the nodes which should be visited next. The algorithm for the breadth-first search is described in more details in sec.(4). The selected operations in the interface of `queue.c` necessary to fulfill the specifications of the program are listed below.

implemented datatype queue.c

```
queue_empty     (free_function free_func)   → queue *
queue_is_empty  (const queue *q)            → bool
queue_enqueue   (queue *q, void *v)         → queue *
queue_dequeue   (queue *q)                  → queue *
queue_front     (const queue *q)            → void *
queue_kill      (queue *q)                  → void
```

Using this set of functions, the main program can create a new queue and verify that it is in fact empty. New elements can be placed at the back of the queue, and the element at the front can be inspected and dequeued. When a breadth-first traversal as been fully completed, the queue used during the traversal is empty and is no longer needed. Using the interface function, the queue can then safely be destroyed and any allocated memory freed accordingly.

A side-note is that the queue implementation `queue.c` does in fact also build on one additional supporting data structure, the list implementation `list.c` given in the code base. But since these implementation details are hidden beneath the interface of `queue.c`, we will not go into detail on what is more than necessary to know for our program implementation in `is_connected.c` done within this lab. Although one should note that it is necessary to include the additional source and header files for this list implementation during the compilation of the program, as detailed in the previous section, sec.(2.2).

3.3 Implementation of main program

Within the main program `is_connected.c`, a couple additional internal functions have been implemented in order to give the required functionality and make the overall solution more modular and easier to overview.

Firstly, the function `populate_graph()` has been implemented to help create and fill a new graph structure according to a map file formatted as per the specifications of sec.(2.3). The function takes the filename of the map file as an argument and returns a pointer to the new and populated graph. During the process of building the graph structure, various verifications and interpretations of the file contents need to be performed. In order to filter out comment lines and blank lines in the map file, the functions `line_is_comment()` and `line_is_blank()`, in conjunction with `first_non_white_space()`, have been implemented. All three of these functions are based on the lecture notes from the "Data Structures and Algorithms" course at the Department of Computing Science, Umeå University, and were originally authored by Niclas Börlin.

Furthermore, the function `find_path()` has been implemented to simplify the call to perform a breadth-first search in the the graph structure. During program execution, this function will be called for each time the user has entered two valid nodes and has not yet chosen to exit the program. As arguments, the function takes a pointer to the graph structure to be traversed, a pointer to the starting node and a pointer to the desired destination node. At completion, the function `find_path()` returns true if there is in fact a path from the start to the destination, and false otherwise. This function is the essential piece to find out the answer to the question at the core of this program, whether or not two chosen arbitrary nodes are connected within a given directed graph structure.

When the user of the program has chosen to exit the program, all dynamically allocated memory for the entire graph structure must be freed. This is performed by the simple call to the function `graph_kill()` which accordingly deallocates all

memory safely before the return of the `main` function.

4 Description of the algorithms

At the center for this solution is the algorithm for a breadth-first traversal of the graph structure. The universal algorithm describing the traversal need only to be modified slightly as to work as a breadth-first search for a specific node. As described in the previous sections, the algorithm uses the datatype queue as means of keeping track of the nodes to be visited.

Assuming a valid input from the user, two nodes are specified and verified to be existing within the graph. For the starting node, mark the node as visited and put it in the queue. Then for each node in the queue, inspect and dequeue the first node of the queue and fetch a list of its neighbours. For each neighbour in the list, check if the corresponding node has already been visited. If not, mark the node as seen and place it in the queue. Continue with this until the queue is empty and thus all nodes in the graph, which can be traversed to from the starting node, have been visited.

This is the universal algorithm for a standard breadth-first traversal. When modifying the algorithm to make it search breadth-first for a specific node, we only need to add one additional test of node equality and a corresponding test-flag. Right after any node gets marked as seen in the algorithm, perform a test if the current node are equal to the destination node. Practically, this is aided by calling the node comparison function `nodes_are_equal()` from the interface of `graph.c` and comparing the current node with the destination node. If we raise a flag once this node comparison test has a positive result, we can let the main algorithm loop terminate early and not wait until the queue is empty. This last additional test-flag is not necessary, but speeds up the execution of the program by not waiting any longer if a result already has been found.

5 Program test runs

Here, two test run examples to briefly verify the functionality of a correct implementation of the main program `is_connected.c` follow. The executable program `is_connected` is fed with two different map files and the corresponding terminal output is displayed. Using the same map file `1-airmap1.map` detailed in sec.(2.3) as an example of a correctly formatted map file, the following terminal output for an arbitrary test run can be seen in fig.(1). As a counterexample, by using an incorrectly formatted map file `x-bad-map-format.map` instead, also supplied by the Department of Computing Science at Umeå University, we can experience an early termination of the program as seen in fig.(2). In this specific map file, two nodes were not stated on each line and as a result, the program exited and gave the corresponding error message.

```
elias@DesktopUbuntu:~/Documents/doa/ou5$ ./is_connected maps/1-airmap1.map
Enter origin and destination (quit to exit): PJA UME
There is no path from PJA to UME.

Enter origin and destination (quit to exit): MMX GOT
There is a path from MMX to GOT.

Enter origin and destination (quit to exit): BMA BMA
There is a path from BMA to BMA.

Enter origin and destination (quit to exit): quit
Normal exit.
```

Figure 1 – Terminal output of the executable program `is_connected` for a correct map file `1-airmap1.map`.

```
elias@DesktopUbuntu:~/Documents/doa/ou5$ ./is_connected maps/x-bad-map-format.map
FAIL: Incorrect map format. Expected two alphanumeric node names on each line.
```

Figure 2 – Terminal output of the executable program `is_connected` for an incorrect map file `x-bad-map-format.map`.

6 Discussion

In conclusion, we have made an implementation of the abstract datatype *graph* and created a program which utilizes this implementation in order to read a file of a directed graph structure and interactively answer questions posed by the user of the program. Many underlying data structures have been used to achieve this, both in the main program and in the graph implementation. The graph implementation has been represented as an array of nodes, where in each element, a directed list is present which contains all the node's neighbours. In the main program, the breadth-first search for the destination node is helped by an implementation of the abstract datatype *queue* to keep track of which nodes to visit next. Overall, non of the underlying datatypes used are really advanced or intricate, but when combined with each other in larger implementations, they can create much more complex and elaborate behaviour.

However, the implementations could absolutely be done in an even more intelligent and efficient manner than what currently has been implemented. For example, having the node names always stored fully in plaintext can likely consume much more memory than necessary and increases the complexity for string comparison and cloning. This becomes especially noticeable if the graph has a large number of edges, i.e. long lists of neighbouring nodes, since even here the node names are stored in plaintext within the directed lists, which is very uneconomical. Furthermore, some other more intelligent way of traversing the array of nodes could be useful. Right now, the implementation linearly searches through the array in search for the right node label, which for a large graph quickly becomes very inefficient. One potential solution that could remedy this problem is perhaps some version of a hashing algorithm to code and decode the node labels into array indices. If such a solution were to be implemented, one could likely see large improvement of traversal times for large graphs.

All in all, this is a working implementation and fulfills the specifications of the lab. It may not be the most efficient and highest performing solution, but given enough memory and time, it will give a satisfying result. Personally, I found it to be interesting working on this lab. Both the design and coding stages were challenging and took quite some planning and mental juggling of all the small parts. In hindsight, I would have liked to try to implement it in a much more intricate way, but since time has been a factor, no such adventures could be had. Nevertheless, it might be a potential personal project for the coming summer.