5DV149 LP4 — Assignment 4

Graphs
Submission v1.0

Isak Mikaelsson (tfy20imn@cs.umu.se)
Henrik Linder (tfy18hlr@cs.umu.se)

June 19, 2022

Contents

0 Version history

v1.0 June 19, 2022 First submission.[1]

1 Introduction

The graph is a data type consisting of nodes and edges connecting these nodes. Graphs have many applications as for example models for electric circuits, or describing networks of different kinds. These networks could for example be describing street maps, communication networks, or travel routes. The last kind is the one we have worked with in this lab. In this assignment, we set out to read in information about available flight routes between different airports in Sweden, stored in text files, and convert this information to a graph of available flights. Then we wrote a program that used this graph to determine whether there is a path to get from one airport to another. An example of a graph to be can be seen in 1 below.

In order to determine whether there is a path between two nodes, a breadth-first algorithm was implemented, traversing the graph from the start node until the destination node is found, or until all nodes connected to the start node has been checked.

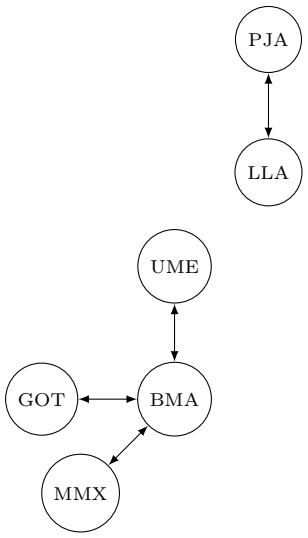

Figure 1: Graph illustration corresponding to the file `airmap1.map`.

2 User guide

2.1 Compilation

```
project root
├─ datastructures-v1.0.9
└─ code-folder
   ├─ graph.c
   └─ is_connected.c
```

Figure 2: The file tree for the example compilation.

[1] If this is a resubmission, include a list of changes with respect to the previous submission.

Assuming the user has the code base in the project's root folder and our code in a separate sub-folder, as shown in Fig.(2) above, the project can be compiled on a Unix-like system by running the following commands.

```
gcc graph.c -o graph.o -std=c99 -Wall -g -I ../datastructures-v1.0.9/include/

gcc is_connected.c -o is_connected -std=c99 -Wall -g -I ../datastructures-v1.0.9/include/
../datastructures-v1.0.9/src/queue/queue.o ../datastructures-v1.0.9/src/dlist/dlist.o
../datastructures-v1.0.9/src/list/list.o graph.o
```

2.2 File format

The file containing the graph description is formatted as follows.

- The file may contain blank lines or comment lines, starting with `#`.
- The first line that isn't blank or a comment contains an integer denoting the number of edges in the graph.
- All other lines each contain one edge:
- The lines contain two node names separated by whitespaces.
- There may be whitespaces at the start or end of the lines, in which case they should be ignored.
- The lines may have comments at the end which should be ignored.
- The node names are alphanumerical and are a maximum of 40 characters each.
- The final line of the file may be missing a line break.
- The text file may be in unix- or dos-format.
- There are no duplicate edges in the file.

An example of a file can be seen below. This file was given as an example in the assignment, and is the one shown in Fig.(1) and the one used in the test run in Fig.(3)

```
# 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
```

2.3 Test runs

Below is a description of a test run of the finished program. `is_connected` is run with the file containing the graph as argument. Then several pairs of start and destination nodes are tested. First, three pairs of nodes where there are paths, including one where the start and destination nodes are the same.

Then, a few examples of paths and nodes that do not exist. Then the user writes `quit` for a normal exit.

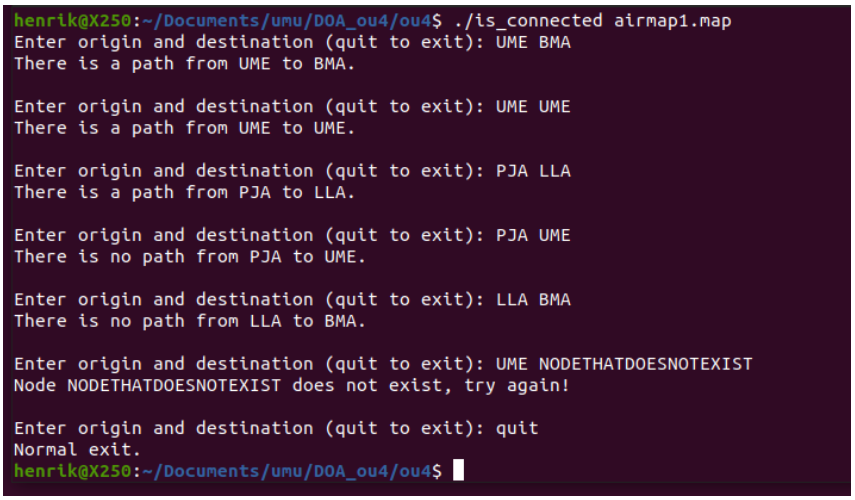

Figure 3: A test run of the program with the graph file airmap1.map. The user has tested several pairs of airports with existing paths, non-existing paths, and invalid inputs.

3 System description

3.1 Data structures

3.1.1 Graph

The graph is implemented as a list, and the interface of the implementation is described below.

The function `graph_empty()` allocates memory for the graph and creates an empty list for the nodes.

The function `graph_is_empty()` calls the `dlist`-function `dlist_is_empty` with the list of nodes.

The function `graph_has_edges(g)` iterates through the list of nodes for the graph g and reads a node at each iteration. The read node's neighbours are sent to the `dlist`-function `dlist_is_empty` to determine whether the list is empty. If the list is not empty, that means that there is an edge, and the function returns 1. If no edge is found in any of the nodes, the function returns 0.

The function `graph_insert_node(s, g)` allocates memory for a node. The string s is then written to the node's `identifier`, its `seen_status` is set to false, and memory is allocated for a potential list of neighbours. Then, a node is inserted into the graph g using `dlist_insert` and the graph is returned.

The function `graph_find_node(s, g)` iterates through the list of nodes and reads a node at eaach iteration. The read node's `identifier` is compared to s using the built-in C-function `strcmp`. If `strcmp` returns `true`, the inspected node is returned. If no s matches the identifier of any node, the function returns `NULL`.

The function `graph_node_is_seen(v, g)` returns `seen_status` for the node v.

The function `graph_node_set_seen(v, g, seen)` changes `seen_status` for the node v to `seen` and returns the graph g.

The function `graph_reset_seen(g)` iterates through the entire graph g, uses `dlist_is_empty` to read each node, and sets its `seen_status` to 0. It finally returns the graph g.

The function `graph_insert_edge(n1, n2, g)` starts by reading the list of neighbours for *n1* and the node name *identifier* for *n2*. Then the node name is inserted into the list of neighbours and the updated

graph g is returned.

The function `graph_delete_node(v, g)` iterates through the graph g and reads nodes that are then compared to v through the function `nodes_are_equal`. If the nodes are equal, the node is removed from the list of nodes and the memory from both the node and its list of neighbours is returned. The graph g is then returned.

The function `graph_choose_node(g)` reads and then returns the node tat is at the top of the list of nodes for g.

The function `graph_neighbours(n, g)` starts by creating an empty list with `dist_empty` and reads the list of neighbours for n. Then the function iterates over the list of neighbours and fetches the node for each neighbour with `graph_find_node`. It then rewrites the the empty list with this list. When the nodes for each neighbour have been written to the new list, the list is returned. The memory is not freed by the function but must be freed by the user.

The function `graph_kill(g)` iterates through the list of nodes and at each iteration reads the top node with `graph_choose_node`. This node is then deleted with `graph_delete_node`. Finally, the list is deleted with `dlist_kill` and the memory for g is freed.

3.1.2 Other data structures

Beyond graph, we also used the data type doubly linked linked list, `dlist`, the interface of which can be seen in the table below.

Table 1: A table showing the user interface for the `dlist` data type.

Function	Description
`dlist_empty(free_func)`	Constructs an empty list.
`dlist_is_empty(l)`	Returns `true` if list l is empty.
`dlist(l)`	Returns the first position in the list l
`dlist_next(l, p)`	Takes the position p and returns the next value in the list l.
`dlist_is_end(l, p)`	Checks if position p is the last element of the list l and returns `true` if it is.
`dlist_inspect(l, p)`	Returns the value at position p in the list l.
`dlist_insert(l, v, p)`	Inserts the value v at position p in the list l.
`dlist_remove(l, p)`	Removes the value at position p in the list l.
`dlist_kill(l)`	Destroys the list l and returns all dynamic memory from the list and its elements.
`dlist_print(l, print_func)`	Iterates over the elements in the list l and prints their values.

We also used the data type `queue`, the interface of which can be seen in the table below.

Table 2: A table showing the user interface for the `queue` data type.

Function	Description
`queue_empty(free_func)`	Constructs an empty queue.
`dlist_is_empty(q)`	Returns `true` if queue q is empty.
`queue_enqueue(q, v)`	Inserts a value v into the queue q.
`queue_dequeue(q)`	Removes the first value from the queue q.
`queue_front(q)`	Returns the first value in the queue q.
`queue_kill(q)`	Destroys the queue q and returns the memory from the queue and its elements.
`queue_print(q, print_func)`	Prints the values of all elements in the queue q.

3.2 Algorithms

3.2.1 Parsing file

The algorithm we used for parsing the text files containing the map and generating a graph from the content can be seen in the list below.

1. For each line
    1.1 If line is blank or comment
        1.1.1 Skip to next line
    1.2 If first non-commented line is not an integer
        1.2.1 Exit with error
    1.3 Read number of edges from first line
    1.4 Remove trailing comments
    1.5 Remove trailing and preceding whitespaces around the string
    1.6 Get position of whitespace in string
    1.7 Save string up to whitespace in array of start node names
    1.8 Save string after whitespace in array of destination node names
2. Create empty graph
3. For number of edges
    3.1 If start or end node for this edge is not in the graph already
        3.1.1 Insert that node into the graph
    3.2 Get start and destination node for this edge from the graph
    3.3 If neither of the nodes are `NULL`
        3.3.1 Insert an edge between the nodes

3.2.2 `find_path`

The function `find_path` is our implementation of a breadth-first search in the graph to find the destination node. The algorithm we used for this is shown below.

1. While queue of neighbours is not empty
    1.1 Read the top node in the queue
    1.2 If inspected node is equal to destination node
        1.2.1 Return 1
    1.3 Remove inspected node from queue
    1.4 Read list of neighbours from inspected node
    1.5 For each entry in list of neighbours
        1.5.1 If inspected node is not seen
            1.5.1.1 Mark the node as seen and add to queue
    1.6 Free memory for list of neighbours
2. Free memory for queue
3. Reset seen-status for all nodes
4. Return 0

4 Reflections

4.1 Work distribution

When working on this project, we wanted to make sure that both of us understand the whole of the work process. This includes writing the code and writing the report. Therefore we sat together and did most of the development and design of the algorithm. When we worked individually, we made sure to sit down afterwards and step through the work to make sure that we were both on board with how everything worked. This had the added benefit of making sure that we both had a good enough understanding of the code to be able to help each other when we had issues.

4.2 Reflections

When doing this assignment we found it very interesting to build the graph and particularly to design the algorithms and uses of the code. We spent quite a bit of time at the start of the project sitting down at a whiteboard and sketching different solutions, trying to figure out ways to get everything to behave according to the specifications. In this phase we collaborated a lot with other groups and people that have taken the course previously, which helped in getting different perspectives and fairly quick solutions to the issues that came up along the way.

This was a surprisingly fun way to get a grip of the assignment and probably saved us some time later since we had a (somewhat) working algorithm from the start and so didn't have to do any major reworkings afterwards. Instead we could just do minor tweaks to account for edge cases that we hadn't considered in the brainstorming phase.

Overall, the work was fairly painless. Of course, there was the usual amount of swearing over memory leaks and pointers behaving unexpectedly, but no huge bumps on the road. The assignment was still time consuming and far from trivially simple, but we agree that it turned out fairly well considering the circumstances.

We both agree that this has been very rewarding work assignment that helped us develop our skills in software development and problem solving much more than the previous assignments. The fact that we had to sit down and plan the structure of the code beforehand was very helpful and forced us to think more in detail about the algorithms, which turned out to be a great experience.