UMEÅ UNIVERSITY
Department of Computing Science
Assignment report

April 2, 2020

5DV149 - Assignment 5

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

Writing a program for searching paths between nodes in a graph

Name	Rasmus Lyxell
user@cs.umu.se	`c19rll@cs.umu.se`
Version	`2.0`

Teacher
Niclas Börlin

Graders
Niclas Börlin, Ola Ringdahl, Anna Jonsson, Fredrik Peteri, Lennart Steinvall

Contents

1 Introduction

This assignment focuses on three things being the construction of a directed unweighted graph as a datatype, reading information from a file, and using breadth first search in a program. To encompass all of this a program and an implementation of a graph that accomplishes the following will be written.

- The program will read a file containing edged between two nodes and represent this file as a graph. This type of file will be called an "airmap".
  - This representation will require a graph implementation to be written
- The user will then be able to input two nodes in the console and see if a path exists from the first to the second.
  - This will be accomplished using breadth-first search in the graph representation of the read file.
- Wrong inputs like invalid paths, invalid formats in files, and invalid inputs from the user during a program run should a be caught and should all be handled appropriately

2 User manual

To use the program, run the compiled file "is_connected" in the terminal followed by the path to a file containing an airmap. After running the program you will be prompted to enter two nodes from the airmap separated by a space like this: "Node1 Node2". After that the program will display if there exists a path from Node1 to Node2. A few examples of user inputs are shown in Figure 1 including invalid inputs.

```
rasmus@rasmus-ideapad ~/d/o/kod> ./is_connected ../maps/airmap1.map
Enter origin and destination (quit to exit): UME UME
There is a path from UME to UME.

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

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

Enter origin and destination (quit to exit): UME
Please enter exactly two nodes in the format 'n1 n2' (separated with one space).

Enter origin and destination (quit to exit): UME STH
The node 'STH' does not exist in the given map.

Enter origin and destination (quit to exit): quit
Normal exit.
rasmus@rasmus-ideapad ~/d/o/kod>
```

Figure 1: Some user inputs

2.1 Airmap criteria

Airmaps need to be structured in a certain way to be read by the program. To allowing commenting in the airmap file the program also ignores any blank lines and characters following a #sign (including the #sign) which will be called comments from now on. For an airmap to be considered valid it needs to meet the following criteria:

- The first line (excluding blank lines and comments) has to contain the number of edges stated in the file.
- The rest of the file should (excluding blank lines and comments) contain edges written as two strings separated by any number of spaces with two strings per line.
  - These strings should each be under 41 characters and only consist of alphanumerical characters
  - No duplicate edges should exist in the file.
- Each line should be at most 300 characters long.

Two examples of valid airmaps can be seen in Figure 2.

```
# Some airline network
7
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
```

(a) Normal example

```
#Airmap
        #Comment

6
Node1 Node2    #Test
Node2    Node3
Node12345678912367891236784123786123                    Node4
a b
c d#Test
d  c

#End comment
```

(b) Extreme example

Figure 2

If an invalid airmap is specified the program will exit and print the reason for the error. Examples of this can be seen in Figure 3

```
rasmus@rasmus-ideapad ~/d/o/kod> ./is_connected ../maps/x-bad-map-format.map
ERROR: Line:
 Nod1 Nod3 Nod4
 contains more than two nodes⏎
rasmus@rasmus-ideapad ~/d/o/kod> ./is_connected ../maps/x-bad-map-format.map
ERROR: Separator other than space in Line:
Nod3    Nod4####
rasmus@rasmus-ideapad ~/d/o/kod> ./is_connected ../maps/x-bad-map-format.map
ERROR: Line:
 Nod2
 only contains one node⏎
rasmus@rasmus-ideapad ~/d/o/kod> ./is_connected ../maps/x-bad-map-format.map
ERROR: First node in line:
Nod5123412341234123412341234123412341 Nod4
contains more than 40 characters⏎
rasmus@rasmus-ideapad ~/d/o/kod>
```

Figure 3: Some invalid airmap messages

3 System description

The program is constructed using three different implemented datatypes; graphs, queues, and directed lists. The interfaces for each datatype and the implementations of queue and directional list come from the provided code base for the course. The implementation for graphs was partly written as a part of this assignment where only the relevant parts that are used in the program were implemented and the implementations require some assumption to function.

3.1 Queue and directional list

Queues were used to perform the breadth-first search in the graph and directional lists were used in the construction of the graph implementation. The relevant parts of their interfaces are shown in section and section .

3.1.1 Queue interface

queue_empty: Takes a function for freeing memory and returns a pointer to new dynamically allocated queue that uses said function for freeing memory.

queue_is_empty: Takes a queue and returns true if the queue is empty and false otherwise.

queue_enqueue: Takes a queue and a void pointer and puts the void pointer at the back of the queue. Returns a pointer to the changed queue.

queue_dequeue: Takes a queue and removes the front-most element in the queue. Frees the removed pointer the the specified free function. Returns a pointer to the changed queue.

queue_front: Takes a queue and returns the void pointer at the front of the queue.

queue_kill: Takes a queue and returns all memory allocated by the implementation and if a free function was specified it also frees each pointer in the queue.

3.1.2 Directional list interface

The directional list used in this program uses "dlist_pos" to represent positions in a directional list.

dlist_empty: Takes a function for freeing memory and returns a pointer to new dynamically allocated directional list that uses said function for freeing memory.

dlist_first: Takes a directional list and returns the first position in the list.

dlist_next: Takes a directional list and a position in said list and returns the next position in the list relative to the input position.

dlist_is_end: Takes a directional list and a position in the list and returns true if said position is the end of the list (meaning the position after the last element) or false otherwise.

dlist_inspect: Takes a directional list and a position in the list and returns the void pointer at said position in the list.

dlist_insert: Takes a directional list, a position in the list, and a void pointer.

Places the void pointer in the position specified. Returns the position of the elemement just inserted.

dlist_kill: Takes a directional list and returns all memory allocated by the implementation and if a free function was specified it also frees each inserted pointer in the list.

3.2 Graph

The graph implementation was constructed using an array to store all the nodes in the graph with each node containing a directional list of all the nodes that it is connected to, excluding itself. Each node has a bool stating if it is "seen" or not and a string representing the nodes name or label. The graph itself contains the array of all the nodes and two coutners for the number of nodes in the graph and the number of edges. The assumption that no nodes will be removed from the graph during use allows the graph to use the number of nodes as the position to insert the next node into the array of nodes and makes looping through all nodes easier by guaranteeing all indices upto the number of nodes has an allocated pointer in it. The following functions from the interface for a graph were inplemented:

nodes_are_equal: Takes two nodes and compares the labels using strcmp() and returns the resulting bool.

graph_empty: Takes a max number of nodes and creates a new graph with the corresponding array node-array size and returns the pointer to the newly dynamically allocated memory.

graph_has_edges: Takes a graph and checks if the number of nodes is over 0 and returns the resulting bool.

graph_insert_node: Takes a graph and a string and creates a new node using the string as a label. The string is copied to the node to fulfill the interfaces promise of the user handling their allocated memory. Returns a pointer to the changed graph.

graph_find_node: Takes a graph and a string and searches the graph for a node with the corresponding label. Returns a pointer to the node found if one is found otherwise returns NULL. Uses strcmp() on each node in the graphs label with the input string to find the right node.

graph_node_is_seen: Takes a graph and a node and returns the bool tracking if the node is seen.

graph_node_set_seen: Takes a graph, a node, and a bool and sets the node in the graph's "seen" bool the the value of the bool inputted.

graph_reset_seen: Takes a graph and loops through all nodes in the given graph and sets the "seen" bool the false for all nodes.

graph_insert_edge: Takes a graph and two nodes. Adds the second node to the second nodes list of neighbours. Returns a pointer to the changed graph.

graph_neighbours: Takes a graph and a node and creates a copy of the list of neighbours for the inputted node. This list has to be killed using dlist_kill to avoid memory leaks.

graph_kill: Frees all memory used by the nodes, the labels, the graph itself.

3.3 Flow of information

The program uses functions that come from some of the lectures by Niklas Börlin that gives indices in strings ignoring whitespace and the function that reads the file to a stream is also taken from a lecture. Other than these utility functions the program consists of a function that creates a graph representation using other functions called "interpretMap", a function that uses a breadth-first algorithm to search for a path between two nodes, and a main function that calls interpretMap and reads the user input to use in the breadth first search. InterpretMap uses several functions (not counting the utility ones) to achieve the representation. One function reads the number of edges, one function reads the labels from a single line from the file which uses another function that reads tokens from a file, and one function adds an edge to a graph while checking for duplicate edges.

3.4 Algorithms

```
Algorithm g=breadthFirst(Node n, Graph g)
  input: A node n in a graph g to be traversed
Queue q ← empty();
(n,g) ← seen(n,g) // Mark the node as seen.
q ← enqueue(n,q); // Put node in queue.
while not isempty(q) do
  p ← front(q); // Pick first node from queue
  q ← dequeue(q);
  neighbourSet ← neighbours(p,g);
  for each neighbour b in neighbourSet do
    if not isSeen(b,g) then
        (b,g) ← seen(b,g) // Mark node as seen.
        q ← enqueue(b,q); // Put node in queue.
```

Figure 4: Pseudo code for a breadth first traversal in a graph

The central algorithm for this program is the breadth first algorithm for traversing graph shown in pseudo code in Figure 4. It is implemented by comparing each node visited to the destination node and returning true if it was found during the traversal.

4 Testing

For testing my program i used airmaps found on the course site combined with the pre-written list of inputs found alongside them. The list of inputs all correspond to a map from the site and contain several different inputs to be tested. To run each input without typing all of them simply piping the result of the "cat" command into the executable allows the scanf function in the code to pick up each input as if they were typed in the console. The resulting outputs still need to be manually checked if they are correct. Valgrind was used to run the executable to also check for memory leaks. The result of such a test looks like in Figure 5

```
rasmus@rasmus-ideapad ~/d/o/kod> cat ../maps/4-standard-test-input.txt  | valgrind --leak-check=full
 --show-reachable=yes ./is_connected ../maps/4-standard-test.map
==17381== Memcheck, a memory error detector
==17381== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17381== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==17381== Command: ./is_connected ../maps/4-standard-test.map
==17381==
Enter origin and destination (quit to exit): There is a path from Nod1 to Nod4.

Enter origin and destination (quit to exit): There is no path from Nod4 to Nod1.

Enter origin and destination (quit to exit): There is a path from Nod1 to Nod2.

Enter origin and destination (quit to exit): There is no path from Nod1 to Nod5.

Enter origin and destination (quit to exit): There is no path from Nod1 to Nod9.

Enter origin and destination (quit to exit): There is a path from Nod9 to Nod4.

Enter origin and destination (quit to exit): There is a path from Nod8 to Nod6.

Enter origin and destination (quit to exit): There is a path from Nod8 to Nod4.

Enter origin and destination (quit to exit): Normal exit.
==17381==
==17381== HEAP SUMMARY:
==17381==     in use at exit: 0 bytes in 0 blocks
==17381==   total heap usage: 218 allocs, 218 frees, 14,561 bytes allocated
==17381==
==17381== All heap blocks were freed -- no leaks are possible
==17381==
==17381== For counts of detected and suppressed errors, rerun with: -v
==17381== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 5: Valgrind memory test using predefined inputs

5 Reflections

Overall I think this project served its purpose well. I've learned to read files and to implement a known algorithm into my program. Only thing I would have done differently if I would have done it again is to make sure to properly split my program into understandable functions before it became hard to understand.