Route Finding

Programming Project 2 Report

Luke Hutton [sc17ljth]

Contents

[Project Summary 2](#_Toc511636739)

[Design Plan 2](#_Toc511636740)

[Large scale design plan 2](#_Toc511636741)

[Aim of the project 2](#_Toc511636742)

[Program input 2](#_Toc511636743)

[Program output 2](#_Toc511636744)

[Medium scale design plan 3](#_Toc511636745)

[Code modules 3](#_Toc511636746)

[Dijkstra’s algorithm 3](#_Toc511636747)

[Priority queue implementation 4](#_Toc511636748)

[Hashing algorithm to efficiently find nodes 4](#_Toc511636749)

[Test Plan 4](#_Toc511636750)

[Testing the file data input functions 4](#_Toc511636751)

[Testing the program with Dijkstra’s algorithm 6](#_Toc511636752)

[Testing the programs output to use with gnuplot 10](#_Toc511636753)

[Schedule 10](#_Toc511636754)

[References 10](#_Toc511636755)

# Project Summary

A graph or network is collection of points (nodes) connected by a series of lines (edges). Graphs are used in an area of maths called discrete maths to model optimisation problems that involves finding an efficient solution to a problem. Solving an optimisation problem means finding the best solution out of many feasible solutions.

This project will involve creating an application that will find the shortest path between two nodes on a network. The network will be given in the form of a file; the application will then read this and output the shortest path. In order to create such an application, the following things need to be taken into consideration: importing the data, the data structure used to store the network and the algorithm used to determine the shortest path.

# Design Plan

## Large scale design plan

### Aim of the project

The ultimate aim of the project is to find the shortest path between two input nodes. This can be done using Dijkstra’s algorithm in O((E + N) log N) time using a priority queue with a binary heap (where E is the number of edges and N is the number of nodes). The program must be able to accept an XML input from a file, read the nodes and edges from such file and then output the shortest path in a graphical way to the user.

### Program input

The programs input will consist of an XML file containing the data used to represent the network. Some of this data will be used and other parts won’t. A plugin called libxml2 can be used to read and parse XML files I will therefore use this in my project to read the data file.

Upon further inspection I decided against using libxml2 and instead using the built in sscanf() and fgets() functions instead. It was simpler to read the file this way as libxml2 required a complex setup and contained many features the program wouldn’t need. However, this alternative method has limitations. The structure of the XML file must remain consistent and use the same <link> and <node> tags to denote an edge and node.

Another input from the user will the 2 points to find the shortest path between. The user will enter these once prompted through the command line.

### Program output

The program will output a graphical network consisting of nodes and edges. I will do this using GNUplot as we have already used it before and it is an easy to use application. With gnuplot you can change the line colour of elements. Therefore, it will be easy to display the shortest path calculated by the program.

## Medium scale design plan

### Code modules

The program will consist of the following files:

* main.c – the main file where the program is run
* tests.c – various tests implemented here which can be run from main
* readFile.c – takes the input XML file and reads it, adding the nodes and edges
* buildNetwork.c – contains functions to ‘build’ the network such as addNodes() and createNetwork()
* networkUtils.c – contains standard functions to use on the network such as getNode()
* binaryHeap.c – implements a binary min heap to use with the priority queue
* priorityQueue.c – implements a priority queue to use with djikstras algorithm
* djikstrasAlgorithm.c – implements dkikstra’s algorithm to find the shortest path between two nodes
* outFile.c – takes the resulting graph and outputs it to a file so it can be read by gnu plot

And the following header files:

* networkStructure.h – defines the structure of the network, nodes and edges
* Header files for each of the ‘.c’ files above

### Dijkstra’s algorithm

Djikstra’s algorithm can be used to calculate the shortest path between two nodes on a network. This algorithm originally had a runtime of O(N2) where N is the number of nodes in the network. However once implemented with a min-priority queue with a binary heap, the efficiency is increased giving it a run time of O((E + N)log(N)) where E is the number of edges and N is the number of nodes in the network.[1] This is the pseudo code I will base my implementation off of in c:

"Find shortest path"

FOR x IN numberOfNodes

distance(x) ← infinity

processed(x) ← false

parent (x) ← null

ENDFOR

distance(SourceNode) ← 0

WHILE (there are still nodes left to process)

Let 'sNode' be a node which hasn't been processed that has the smallest distance(node)

ENDWHILE

IF (sNode ← destinationNode) THEN

Exit While Loop

ENDIF

processed(sNode) ← true

FOR Each unprocessed parent node, dNode

IF (distance(node) + weight(sNode, dNode) < distance(dNode)) THEN

distance(dNode) ← distance(sNode) + weight(sNode, dNode)

parent(dNode) ← sNode

ENDIF

ENDFOR

# Find path

node ← destinationNode

WHILE node != sourceNode

Append node to the beginning of the path list

node ← parent(dNode)

ENDWHILE

Append sourceNode to the beginning of the path list

### Priority queue implementation

To implement the min-priority queue for use with djikstra’s algorithm, we will use a min-binary heap. This is a data structure that guarantees the item with the lowest priority will be at the top of the structure. This means that finding the item with the minimum priority can be found in O(1) time and found and removed in a worst case run time of O(log n).[2]

Whilst a more efficient implementation with better time complexity is possible using a Fibonacci heap, this is often inefficient in practice due to the complexity of the algorithm. They are only typically efficient on sparse networks with less edges between nodes.

A min-binary tree is ordered in such a way so that the first item on the tree has the lowest priority. For any node N, if P is the parent of N, then the priority of P is less than or equal to the priority of N[4].

### Hashing algorithm to efficiently find nodes

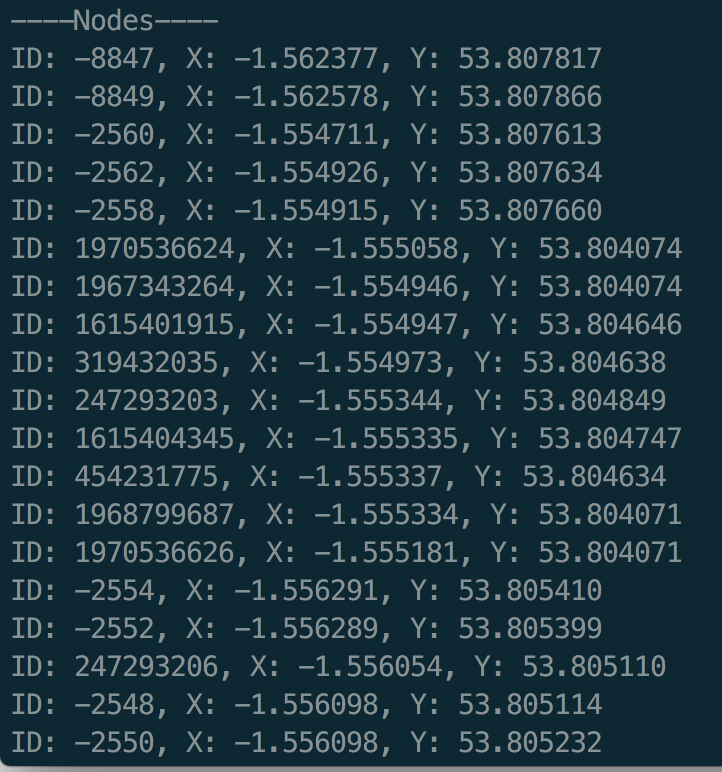
Dijkstra’s algorithm requires nodes to be ‘found’ amongst all nodes that were added. Instead of searching through each node and checking its ID (which would take at most O(n) time), a technique called hashing can be used. This has an average time complexity of O(1) meaning it is much more efficient, especially on a large dataset like the one we have.

In order to form a hash of the data, I will use a library called uthash. This provides a way to make a hash table and handle hash clashes that occur.

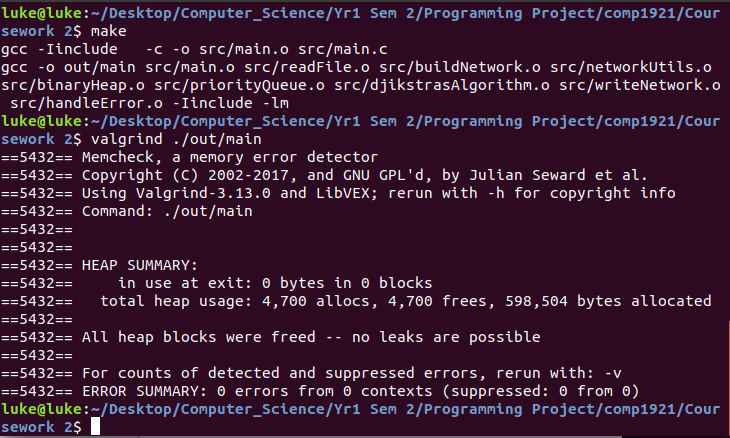
# Test Plan

## Testing the file data input functions

The program first needed to be able to take a file as an input and obtain node and edge data to be used throughout the rest of the program. In order to test this, I started out with a small trial dataset of 3 nodes and 3 edges to see if the data was obtained correctly and added to the struts correctly. Then I used the large dataset to test for a much larger set of nodes and edges.



This list of nodes continues, and the edges are then displayed. But it is clear to see that the nodes and edges are correctly added to the ‘adjacencyListArray’ and each node’s ‘linked adjacency list array’ respectively.

I then checked to make sure that this part of the program had no memory leaks. Including destroying the network after adding all the data:

## Testing the program with Dijkstra’s algorithm

The next part of the program involved implementing Dijkstra’s algorithm to perform on the data input from the text file. In order to test this, I came up with multiple pre-calculated paths and made sure the program returned these correctly:

*The tests used can be run using the ‘runTests.sh’ bash script*

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Expected output: | Actual output: | Pass / Fail |
| Test 1 – testing wrong filename input | An error stating that the file at the file path entered could not be found. | ‘Error: Error opening file, please check the file path provided’. | Pass |
| Test 2 – running the program with no parameters provided for the file path and start and end node ID’s | An error stating that too few parameters were provided. | ‘Error: Missing arguments 'filePath, start node ID, end node ID'’ | Pass |
| Test 3 – running Dijkstra’s algorithm on a disconnected graph | An error stating that a path could not be constructed because a node in the path is disconnected. | ‘Error: Path cannot be completed because a node is disconnected from the network’ | Pass |
| Test 4 – testing an input with node ID’s that don’t exist | An error stating that the node ID’s could not be found. | ‘Error: Node ID could not be found, please ensure the node exists’ | Pass |
| Test 5 – testing Dijkstra’s algorithm on a simple map | Expected path: 1-2-5-4  Expected distance: 7 | ‘Path distance: 7.000000  ---------PATH----------  ID:1, X:2.000000, Y:4.000000  ID:2, X:4.000000, Y:6.000000  ID:5, X:7.000000, Y:2.000000  ID:4, X:9.000000, Y:4.000000  -----------------------’ | Pass |
| Test 6 – testing Dijkstra’s algorithm on another simple graph. | Expected path: 6-1-2  Expected distance: 3 | ‘Path distance: 3.000000  ---------PATH----------  ID:6, X:4.000000, Y:2.000000  ID:1, X:2.000000, Y:4.000000  ID:2, X:4.000000, Y:6.000000  -----------------------’ | Pass |
| Test 7 – wrong type of node ID entered i.e. text instead of a number | An error stating that the node ID’s entered are invalid. | ‘Error: Node ID could not be found, please ensure the node exists’ | Pass |
| Test 8 – the same start and end nodes are entered | An error stating that the start node and end node cannot be the same. | ‘Error: Start node cannot be the same as the end node’ | Pass |
| Test 9 – a path that should contain a single edge between two adjacent nodes | Expected path: 1967343264-1615401915  Expected distance: 49.025177 | ‘Path distance: 49.025177  ---------PATH----------  ID:1967343264, X:-1.554946, Y:53.804074  ID:1615401915, X:-1.554947, Y:53.804646  -----------------------’ | Pass |
| Test 10 -a medium length path between two nodes | Expected Path: -2562,-2560, 1187324666, 1187324679, 54060643, 54060637, 985096817, 985096821, 985096825, -2416, -2380  Expected distance: 139.481734 | ‘Path distance: 139.481734  ---------PATH----------  ID:-2562, X:-1.554926, Y:53.807634  ID:-2560, X:-1.554711, Y:53.807613  ID:1187324666, X:-1.554948, Y:53.807527  ID:1187324679, X:-1.554977, Y:53.807365  ID:54060643, X:-1.555217, Y:53.807386  ID:54060637, X:-1.555378, Y:53.807402  ID:985096817, X:-1.555495, Y:53.807059  ID:985096821, X:-1.555548, Y:53.807069  ID:985096825, X:-1.555564, Y:53.807037  ID:-2416, X:-1.556363, Y:53.807154  ID:-2380, X:-1.556347, Y:53.807182  -----------------------’ | Pass |

Test 11 – A large path with many edges

Expected Result:

Expected path: -2524, -2540, 247958669, 247958668, 301673229, 247958643, 247958642, 247958641, 247958640, 247958645, 247958646, 247958647, 247958648, 247958649, 247958650, 247958651, 247958653, 247958654, 301673241, 247958598, 247958611, 247958610, 247958608, 247958607, 247958606, 247958605, 247958604, 301673248, 247958601, 247958600, 247958599, 247293219, -2502, 247293220, 247293215, 985096764, 985096813, 985096817, 54060637, 54060643, 1187324679, 1187324666, -2560, 1187324670, 1187324682, 1187324692, 1187324676, 1187324678, 1187324665, 1187324690, 984231425, 54060524, 984231603, 54060515, 54060547, 54060551, 1725582974, 54060555, 54060557, 1668111641, 1668111640

Expected distance: 464.077136

Actual Result:

‘Path distance: 464.077136

---------PATH----------

ID:-2524, X:-1.556024, Y:53.805705

ID:-2540, X:-1.556026, Y:53.805809

ID:247958669, X:-1.556134, Y:53.805909

ID:247958668, X:-1.555860, Y:53.805960

ID:301673229, X:-1.555653, Y:53.805973

ID:247958643, X:-1.555440, Y:53.805993

ID:247958642, X:-1.555420, Y:53.805998

ID:247958641, X:-1.555412, Y:53.805999

ID:247958640, X:-1.555397, Y:53.806004

ID:247958645, X:-1.555393, Y:53.806023

ID:247958646, X:-1.555395, Y:53.806042

ID:247958647, X:-1.555397, Y:53.806051

ID:247958648, X:-1.555404, Y:53.806069

ID:247958649, X:-1.555412, Y:53.806084

ID:247958650, X:-1.555418, Y:53.806096

ID:247958651, X:-1.555426, Y:53.806114

ID:247958653, X:-1.555429, Y:53.806124

ID:247958654, X:-1.555431, Y:53.806137

ID:301673241, X:-1.555433, Y:53.806157

ID:247958598, X:-1.555411, Y:53.806157

ID:247958611, X:-1.555411, Y:53.806227

ID:247958610, X:-1.555400, Y:53.806294

ID:247958608, X:-1.555371, Y:53.806336

ID:247958607, X:-1.555319, Y:53.806361

ID:247958606, X:-1.555260, Y:53.806407

ID:247958605, X:-1.555212, Y:53.806430

ID:247958604, X:-1.555167, Y:53.806458

ID:301673248, X:-1.555178, Y:53.806551

ID:247958601, X:-1.555190, Y:53.806568

ID:247958600, X:-1.555207, Y:53.806595

ID:247958599, X:-1.555217, Y:53.806633

ID:247293219, X:-1.555220, Y:53.806689

ID:-2502, X:-1.555212, Y:53.806754

ID:247293220, X:-1.555209, Y:53.806774

ID:247293215, X:-1.555187, Y:53.806972

ID:985096764, X:-1.555269, Y:53.806968

ID:985096813, X:-1.555247, Y:53.807015

ID:985096817, X:-1.555495, Y:53.807059

ID:54060637, X:-1.555378, Y:53.807402

ID:54060643, X:-1.555217, Y:53.807386

ID:1187324679, X:-1.554977, Y:53.807365

ID:1187324666, X:-1.554948, Y:53.807527

ID:-2560, X:-1.554711, Y:53.807613

ID:1187324670, X:-1.554686, Y:53.807622

ID:1187324682, X:-1.554646, Y:53.807846

ID:1187324692, X:-1.554574, Y:53.808077

ID:1187324676, X:-1.554412, Y:53.808180

ID:1187324678, X:-1.554382, Y:53.808301

ID:1187324665, X:-1.554195, Y:53.808311

ID:1187324690, X:-1.554048, Y:53.808346

ID:984231425, X:-1.554026, Y:53.808717

ID:54060524, X:-1.553937, Y:53.808757

ID:984231603, X:-1.554091, Y:53.808897

ID:54060515, X:-1.554290, Y:53.809044

ID:54060547, X:-1.554369, Y:53.809073

ID:54060551, X:-1.554168, Y:53.809208

ID:1725582974, X:-1.554076, Y:53.809261

ID:54060555, X:-1.554034, Y:53.809306

ID:54060557, X:-1.553964, Y:53.809354

ID:1668111641, X:-1.554272, Y:53.809558

ID:1668111640, X:-1.554399, Y:53.809494

-----------------------’

Then I checked that the program wasn’t leaking memory using valgrind:

## Testing the programs output to use with gnuplot

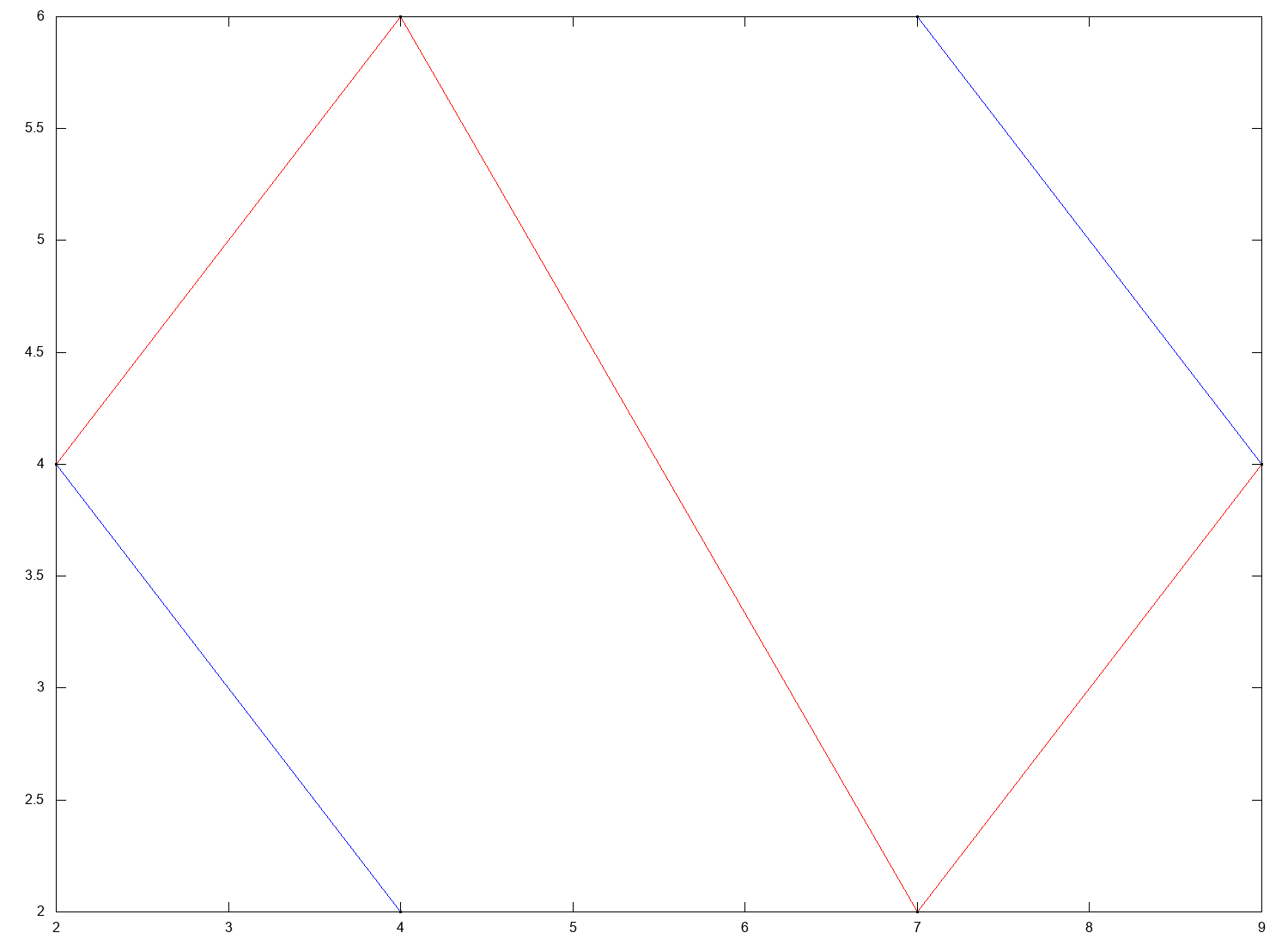
The last part of the program is creating a way to visualize the data collected. This uses an application called gnuplot which can be used to create graphs and charts. It was easy to test whether this part of the application worked correctly as the output should be a network with certain edges colours differently denoting a path. I tested this using the paths calculated above to make sure the program worked correctly.

I then checked to make sure the program contained no memory leaks for each path tested, this is an example of one such path:



The program has been designed to output a png file giving a visual representation of the network and path. In the tests implemented above here are the png files produced:

Test 5



# Schedule

# References

[1] https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm Retrieved 22/3/2018

[2] https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html Retrieved 22/3/2018

[3] [https://en.wikipedia.org/wiki/Fibonacci\_heap Retrieved 22/3/2018](https://en.wikipedia.org/wiki/Fibonacci_heap%20Retrieved%2022/3/2018)

[4] <https://en.wikipedia.org/wiki/Heap_(data_structure)> Retrieved 25/3/2018