Algorithms Assignment Report

Exam Number: Y3843317

Dijkstra's Algorithm

C Implementation

Table of Contents

Abstract

This report outlines the development, design choices and testing of an implementation of Dijkstra's Algorithm. The program is written in standard ANSI C using the minGW (gcc) C compiler. The program is modular and with heavy emphasis on Abstract Data Types (ABT). This report covers the chosen ADT's and justifications for those choices. It also describes the limitations of the program and suggested improvements that could be made in future updates. It explains the testing methodology used and shows the results of those test. A small section on how to use the program is provided at the end of the document.

The Problem

To write and verify a C program that implements a Graph Abstract Data Type (ADT). It should then read in a tab-delimited file, "ukcities.txt", of city names and distances between the cities and use this to fill the Graph. This should then be used to find the shortest distance between two cities provided in the "cityparis.txt" file. It is then to print the distance between the cities and route taken to another text file "output.txt". It must be written in standard ANSI C using the minGW (gcc) C compiler.

Solution Overview

This program uses two ADT to implement Dijkstra's Algorithm. These are a Graph and Minimum Heap. The Graph ADT is implemented using an Adjacency List. The graph consists of an Array of pointers to its Vertices all of which contain an array of pointers to their Edges. Each Edge then has a pointer to its start and end Vertices. This was chosen to minimise the data requirements of the Graph, when compared to implementations such as an Adjacency Matrix. Each Vertex contains the city name string, which must be a max of 250 Chars and other information about its location within the Heap structure. The use of pointers and dynamic memory allocation and reallocation is heavy throughout the program. Getters and Setters are therefore a common function within the implementation of the ADT's to enable the changing of a pointers value from outside the .c file it was declared within. Dijkstra's shortest path is then used alongside a minimum heap to calculate the shortest distance and path between to vertices. A minimum heap was chosen due to the $O(E_T * LogV)$ efficiency it provides. Where E_T is the total number of Edges and V the number of Graph Vertices. All dynamically allocated memory is freed once it is no longer needed. This is mainly done in specific destructor functions within each data structure file and is called at the end of the program.

Detailed Design

Reading Files

The files that are used as inputs to the program must follow a certain format. Though the program has the error handling to warn if this is not the case for most situations. The inputs must be tab-delimited and each line must be terminated in both a carriage return '\r' and line feed '\n' characters.

To read the files the `fscanf()` function is used where the number of items `fscanf()` reads per line is used to determine if it has reached the end of the file or has encountered an error. For the "ukcities.txt" file the number of items should be three. Two strings and an integer. Therefore the function looks like:

```
while(3 == fscanf(file, "%[^\t]\t%[^\t]\t%d\r\n", start, end, &distance))
```

Here, `%[^\t]` , means read everything that is not a tab symbol and it stores this into the variable start. These method of reading the file is used in graph.c. Error checking for input files is explained in the Error Handling section.

Graph

The Graph ADT is implemented using three structures. These are the Vertex and Edge structures and the overall Graph structure. The implementation of the Graph is done using an Adjacency List. The Graph structure contains the number of overall cities as in integer value and then a pointer to an array of Vertex pointers called `adjLists`. Each Vertex structure contains a pointer to a list of Edge pointers which forms the adjacency list. Each Vertex also includes some information needed for the Heap implementation of Dijkstra's Algorithm, such as, the previous vertex, distance from the source vertex, index position in the Heap and a visited state used for Boolean true or false. The cityName variable within the Vertex is a dynamically allocated 250 Bytes of memory enough to fit 250 chars in. This is one of the limitations of the Graph ADT as if a input string of over 250 characters is read in the program will terminate. This could be fixed by dynamically reallocating the size of the char and reading the file character by character. However due to time constraints and simplicity this has not yet been implemented.

An Adjacency List implementation was chosen over an Adjacency Matrix implementation due to considerations on memory efficiency. This was based off the assumption that the Graph would be sparse, with most cities not connected to one another. This would have resulted in many of the values within an Adjacency Matrix being zeroes or wasted space in memory.

To populate the graph the edges are read from the "ukcities.txt" file. Each city on each edge is then checked to see if it is already known and a vertexNumber assigned to it. After assigning or retrieving the vertexNumber the Edges are then constructed connecting the Vertices together. Each new Edge reallocates enough memory within the associated source and end vertices structures, this means the minimum memory required is used. Due to reading through all the edges, E_T, and checking if the cities are already known vertices, V, the total efficiency of pre-processing the data and populating the Graph is therefore O($E_T V$).

Heap

The Heap ADT is a Minimum Heap implementation, the smaller the value the higher its priority within the Heap. This particular Heap is addressable from Index 0. Meaning that the equations for child and parent Nodes are as follows:

```
parentPosition = (int)floor(((currentPosition - 1)/2));

childPositionLeft = 2*currentPosition + 1;
childPositionRight = 2*currentPosition + 2;
```

The Heap is used to contain all the Vertices distances from the source Vertex. This means as Dijkstra's Algorithm runs through updating the distances, smaller values rise to the top and the smallest value is easily accessible as it will be at the zero index.

The heap Nodes all contain pointers to the Vertex within the Graph that they are associated with. This enables the updating of their index position in the heap to be updated and stored within the Graph Structure. Used to speed up access when a shorter distance is found within Dijkstra's to O(1).

Like the Graph, the minHeap was also designed with memory efficiency in mind. Each new Node reallocates only enough space within the Heap to store one extra Node. This resulted in some buffer overflow errors when sorting the Heap. When checking if the furthest right Node of the Heap has any children it would be result in a call to an index of the minHeap array that was out of its range as

memory hadn't been allocated there yet. To fix this, two variables `hasChildLeft` and `hasChildRight` were added to the Node structure and are used as flags to determine if it is necessary to check their values. This prevented the addressing of unallocated memory locations.

The Enqueue and Dequeue functions are both, at worse, O(LogN) where N is the number of Nodes. A more detailed explanation of this is in the Calculating Big-O of Program section. This results in a much greater efficiency compared to storing the shortest distances in an unsorted array or using alternative sorting methods such as bubble sort or quicksort to have the smallest distance at index zero.

Testing and Validation

Error Handling

When reading in the file and parsing the data there is a high potential for error if the data from the file doesn't match the expected input. To that end, some safeguards have been implemented to reduce the likelihood of error.

When reading the "ukcities.txt" file for the first time, the `populateGraph()` function will initially count the number of line within the file, and then compare the number it reaches when trying to read two strings and an integer from each line. If it encounters a line that isn't the correct format it exits out the loop and the line number it reached is displayed as having caused the error. This same technique is used for reading the "citypairs.txt" file also. The program then exits with a status of -1.

Whilst building the Graph representation of the input data the distance between each city must be greater than zero. Having a zero distance wouldn't cause any errors in the calculation of Dijkstra's Algorithm. However, it does result in the addition of a useless Edge from a city to the same city or is signifies an error in the distance that the user has assigned for that Edge. An error message is displayed to the user with the source and end city names displayed and the program exits with a status of -1.

Prior to running Dijkstra's Algorithm the Vertex number associated with the input source and destination city names is retrieved in the function `graphGetVertexNumber()`. If the city name has not been seen before within the "ukcities.txt" file the function returns an error message stating that the city is not known and the program exits with a status of -1.

Catching errors of input. Must have 3 inputs on each line of the "ukcities.txt" file and 2 on each line of the "citypairs.txt" file. City must be one that is known in the "citypairs.txt" file. Distances can't be zero or less. Files must open correctly and return error when they don't.

Some other inputs variations that have been tested and work correctly include: City names with spaces, such as "Great Yarmouth". Using just a single Edge as an input. The Source and End inputs in "citypairs.txt" being the same city. Multiple Edges of different lengths between two cities, the shortest Edge will be returned as the distance between the two.

The program has been tested to run correctly with a "ukcities.txt" file with up to 1000 unique city names and up to 100000 Edges. However, an input string of over 250 chars does cause the program to crash and this is not yet error handled and just results in segmentation fault.

Output

For an input text file “citypairs.txt” which contained the following source and destination cities for the program to calculate:

Leicester & Moffat, Hull & Oxford and Lincoln & Bristol.

The resulting output written to the “output.txt” file is shown in Figure 1

```
Leicester to Moffat is 474km

Route:
Leicester ---> Sheffield ---> Leeds ---> Blackpool ---> Carlisle ---> Moffat



Hull to Oxford is 325km

Route:
Hull ---> Nottingham ---> Birmingham ---> Oxford



Lincoln to Bristol is 284km

Route:
Lincoln ---> Leicester ---> Birmingham ---> Bristol
```

Figure 1 The contents of the output file showing calculated total values and journey for three different routes.

Calculating Big-O of Program

Heap Dequeue and Decrease Node Value Functions

In the worst case scenario of both of these functions the efficiency is O(Log_2V). This is due to the structure of the heap and the number of possible comparison operations to restore the heap condition. The number of comparisons in the worst case would be from the top node down to the bottom most node or vice versa. As shown in Figure 2, for 7 nodes the number of comparisons is the floor of $log_2(7) = 2$.

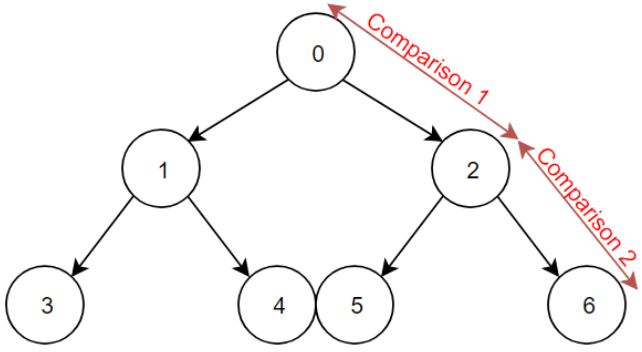

Figure 2 in the worst case, the number of comparisons for a Heap with seven Nodes is two.

Dijkstra's

For the Dijkstra's algorithms using the implementation of a minimum heap very basic pseudo code is shown below with the calculated time efficiency of each step shown on the right.

```
while(Min_Heap_Is_Not_Empty){  ------------------------> O(V)
    Vertex_A = minHeapDequeue(minHeap);  ---------------> O(Log(V))
    for (For_All_Edges_Of_Vertex_A){  ------------------> O(Ev)
        Vertex_B = Vertex_At_End_Of_Edge  --------------> O(1)
        if(Vertex_B_Is_Not_Visted){  -------------------> O(1)
            Calculate_Alternate_Route  -----------------> O(1)
            if(Alternate_Route_Shorter){  --------------> O(1)
                minHeapDecreaseNodeValue(minHeap, v); -> O(Log(V))
            }
        }
    }
    Vertex_A_Is_Now_Visted  ----------------------------> O(1)
}
```

V = the total number of Vertices in the Graph
E_V = Edges on each individual Vertex

This would result in an efficiency of:

O(V*(LogV + E_V*LogV))

Simplified to:

O(V*E_V*LogV)

And since V*E_V is the total number of Edges, E_T , this could be more tightly written as:

O(E_T*LogV)

Total Efficiency

Graph Pre-processing	Dijkstra's
E_TV	E_TLogV

Taking a worst-case scenario, the highest order of BigO of the entire program is E_TV

Verifying Predicted Results

The time efficiency of the program was tested to confirm that it matched the expected efficiency of O(E_TV). This was done using another small C program that generated N random Edges at a time, choosing cities at random from 1000 unique city names that were downloaded from the website (Name Generator, 2019). These Edges were stored in the correct, tab-delimited, format in a .txt file ready to be read in by the Dijkstra's program. This was then used to generate bigger and bigger Graphs for the program to solve.

For each input data set the program was run 10000 times and the total CPU_TIME_USED taken was then divided by 10000 so give the average time taken for each run of the algorithm. CPU_TIME_USED as this prevents background applications affecting the result. The data collected is shown below in Table 1 and Figure 3.

Dijkstras Efficiency		
Edges	Vertices	Time/s
100	169	0.009709
200	318	0.01159
300	439	0.01431
400	547	0.01618
500	650	0.01784
600	693	0.0185
700	754	0.01943
800	805	0.02051
900	846	0.02241
1000	874	0.02481

Table 1 Time efficiency measurements of

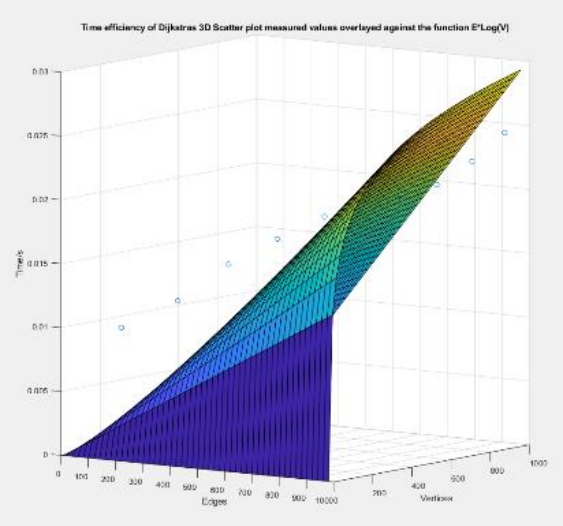

Figure 3 Dijkstras function of program plotted

The time efficiency of the Dijkstras function of the program seems to match somewhat the BigO estimation of its efficiency. Though a much larger sample of data and more rigorous testing would be needed to confirm this to be the case for certain.

The method of generating new test input files was also used to test the program could handle huge input graphs with one-thousand Vertices and one-hundred-thousand Edges. Which it did without error.

User Manual

To change the output city routes use edit the “citypairs.txt” file. Make sure that the city names used are also within the “ukcities.txt” file. Each city must be separated with a tab and each line terminated with a carriage return and line feed characters. For checking input files it is recommended that Notepad++ is used and the setting “Show all Characters” selected so as to check the input file matches the required formatting. With “Show all Characters” selected the input file, “ukcities.txt”, on Notepad++ should look like 4.

```
1 York→Hull→60CRLF
2 Leeds→Doncaster→47CRLF
3 Liverpool→Nottingham→161CRLF
4 Manchester→Sheffield→61CRLF
5 Reading→Oxford→43CRLF
6 Oxford→Birmingham→103CRLF
```

Figure 4 Notepad++ with all characters shown, displaying necessary format of ukcities file.

Once the both files are setup, open and run the main.c file. The output will then be saved to “output.txt” and can be viewed in any text editor.

Bibliography

Name Generator. (2019). Retrieved from https://www.name-generator.org.uk/town/