Lab - Honors Option – Dijkstra’s Algorithm

### Cayden Koweck – hd6247

## Functionality:

This program takes a CSV file input of a graph (directed, or undirected, and can accept weighted graphs with all positive edges). And converts it into a 2D array, the user is then able to insert a start and end node to calculate the shortest path for, the program calculates the shortest path, and displays its cost, the path taken, as well as the time taken to calculate both with a pairing heap, and a priority queue..

## Methodology:

I split this project into three major sections: File processing and interfacing, the algorithm itself, and then coding separate data structures for the minimum distance function which is where optimization for this algorithm is focused in this lab. In addition, while most implementations of this algorithm end up calculating the entire distance matrix as a part of the algorithm, I wanted to give the user the option to process the nodes individually (saving time when processing the entire table was unnecessary). My original implementation (to get the algorithm working) used a simple array that was searched through to find the minimum value.

It’s important now to determine what data structure is a good fit for this use. The two processes that are utilized n times are Accessing the minimum and modifying a value, an array gives us O(N) and O(1) respectively since we know the index we are modifying ahead of time. This is a step away from the traditional basic implementation of the algorithm which uses a priority queue which has O{logn) for getMin and O(1) for decrease Key, my basic structure was a simple array that mapped the index of the node to the index of an array of structs that kept track of information on that node, this flipped the amortized complexity of getting and decreaseKey to O(N) and O(1) respectively.

My method for implementing a pairing heap was to first make the algorithm modular (as long as a class has the members: nodeProcessed), decreaseKey, and dequeueMin it can be swapped in via changing the type of minStructure. Then I wrote a generic pairing heap class and used that as a base to implement the above three functions. To do this in a decent time I added a secondary array of pointers to the (for which the index element could be used as the key) and I added a parent field for the node struct allowing me to access any node based on it’s index. Once the array structure was implemented dequeueMin and nodeProcessed were very easy to implement, the parent tracking mechanism was significantly more challenging.

## Issues:

While the pairing heap works properly for 99% of calculations, there seems to be a few edge cases that break the tree structure of the heap. There we’re many bugs I was able to squash, this was not one of them. I did however allow the program to catch the underflow error without terminating the program.

## Results and conclusion:

When compiling the program, I expected the pairing heap to operate much less efficiently than the priority queue considering the comparatively high constant costs of the pairing heap. This was reflected in the test, as while the pairing heap occasionally outperformed the priority queue, a much larger graph is needed to reap the benefits of the reduced complexity of extracting the minimum value. The Fibonacci heap runs into this problem on a much larger magnitude.

## Program output

Text

Description automatically generatedText

Description automatically generated\