Project – Railwaysystem DV1490

1. To construct a railwaysystem

The primary task at hand is to find the optimal path by which to connect a number of railwaystations. It is required that this is accomplished with a minimum spanning tree (MST) algorithm, choices being either Prim's or Kruskal's algorithms. As this choice is a matter of preference, Prims algorithm is chosen early on to keep later information relevant.

The cost of each railway is specified in an input-file. Upon examination of said file it is evident that this data first must be translated and made easily accessible by the algorithm before it can be used to generate an MST. In other words, if the drivercode is excluded, code can be separated into 2 sections. Translation and processing. Considering the choice of Prim's algorithm, the input file can mainly be translated into a number of datastructures. Out of these, adjacencylists and adjacencymatrices we're primarily researched.

An adjacencylist is essentially just a linked list where the first element is the current vertex, and subsequent elements are vertices connected to it. This method is space efficient, and makes it quick to report neighboring vertices, which can be performed with constant time complexity. Alternatively one can use an adjacencymatrix. This can be described as a V by V grid where V is the total amount of vertices in the dataset. Each square of the grid can then be given a boolean value, which is determined by whether or not the vertex marking the row and the vertex marking the column are connected or not. Both an adjacency list and matrix seem to be viable options. There are some differences between these that are of note. An adjacency matrix is less space efficient than a list. A hashtable indexed by pairs of vertices can fix this issue, but implementing this is assumed not to affect run. time and will therefore not be done. An adjacencymatrix also has the ability to check for valid edges in a constant time.

The next problem is the implementation of the algorithm. There should not be too much flexibility how it calculates the MST, as to not loose the identity of Prim's algorithm. One should however consider how the calculated answers should be translated back to human readable form. One option is to make modifications to the existing adjacency list/matrix and then have another function do the reverse of the first translation. Alternatively information can be grabbed from the algorithm while it makes it's calculations, and output it as soon as it becomes available. Either option

2. Method

The first step is translation of the dataset. Every file starts by listing all vertices, each followed by a newline. This needs to be moved through before one can work on any adjacency list or matrix. Using a loop which iterates through the file line by line up until it contains nothing but a newline is used to solve this. There is some useful information which can be harvested along the way. Considering that this loop iterates as many times as there are total vertices, each iteration can be used to add the current line to a list containing every vertex. The current line as well as the iterator will also be added to a map. A map is like a vector, but instead of accessing elements by index they can be accessed by a specified key. By setting each line as the key and the current iterator as element, we'll be able to find the index of any element from without needing to search.

The given testdata always features an edgecount greater or equal to the amount of vertices. Taking this into account, an adjacency matrix will be used. It's constant time complexity when checking for valid edges makes this operation more common as edgecount increases. Some tweaks can be made to the standard matrix. Usually adjacency matrices only give information about whether 2 vertices at a given index are connected or not. This can be expanded upon. If the adjacency matrix is filled with integers rather than boolean values, the matrix can be used to show the cost of edges. This makes it possible to get the cost of every edge in the matrix if one knows the

index of the connected vertices. This means that edges which do not exist in the graph must be set as infinitely large, to avoid having the algorithm travel for free unless intended.

At this point another issue arises. The X and Y axis of the adjacency matrix are labeled by element name. This means that checking for for any connection would require the program to search for the matching elements on each axis and return their index. Once this is completed the index can be plugged into the matrix and a cost can finally be obtained. This process is lengthy increases overhead for finding valid edges, which makes the use of an adjacency matrix pointless. That's why the map was created. By feeding the map the current vertex, it returns it's index, all with constant timecomplexity. All of this amounts to about 4 lines of code. If using a map named index and ifstream called data, an entire matrix can be filled with the code seen in figure 1. One should note that every integer added to the adjacency matrix is added in 2 places, making it symmetrical. This is because none of the given edges are directional, and can thus be traveled both back and fourth.

At this point it is important to note that while traditional arrays make syntax more readable, they have some limitations when initialized to the size of a variable. On top of this, a 2-dimensional array would crash the program if run with too many elements. Therefore the program will instead use vectors anywhere possible.

This concludes the translation of data. It's time to deploy the algorithm. Prim's algorithm generally operates using a list of visited and unvisited vertices. It picks one at random, marks it as visited, and searches the cheapest edge to travel along next. If there are multiple one will be picked at random. Next the algorithm looks at every edge that leads somewhere unvisited, and so on. This process continues until there are as many visited vertices as total vertices. This will be the end condition. The algorithm will then loop through every element in the adjacency matrix. To give the algorithm a startingpoint, any edge can be marked as visited. To avoid potential segmentation issues with smaller graphs, index 0 is chosen. It is now time to find edges to connect to. Our adjacency matrix is represented as a 2 dimensional vector Each element can at this point be seen as an edge in the graph. It has start and end vertices in the form of coordinates, and a cost in the form of it's integer value. To check if any given edge should be connected the following tests must be run on it:

- The edge must be the cheapest available.
- The edge cannot connect to itself.
- The edge must have 1 visited vertex.

If each of these conditions are true the current coordinates and cost are saved. If cheaper options are found on the same run these will overwrite the previous ones. Once the entire adjacency matrix has been explored, the previously unvisited vertex has it's status changed to visited. At this point it was also decided to start writing to a file containing all selected edges. This approach might be seen is as strange, as it makes little sense for the algorithm to also print its calculations. Alternatively one could add every edge to a vector, or just alter the adjacency matrix, but either of these approaches are seen as an unnecessary step.

3. Analysis

The expected time complexity of Prim's algorithm with adjacency matrix is $O(v^2)$ where v is the total amount of vertices. This only this is only for finding a valid edge however, and the current implementation has yet another loop, bringing complexity up to $O(v^3)$ This code analysis is done in figure 2, where the nested loop is the clear bottleneck. Unfortunately there are few optimizations that can be done without migrating from an adjacency matrix to either an adjacency list, binary heap or fibonacci heap which have superior timecomplexities of either $O(e \log_2 n)$ or $O(e+n \log(n))$, where e is the amount of edges. By getting the average of several runs with differing amounts of vertices, the following results are recorded:

Table 1: Runtime test

Data: Vertex – Edge	100 – 1000	1000 – 10000	10000 - 10000
Time: (Seconds)	0.047	9.944	3276.0276

These numbers roughly match the time complexity of $O(n^3)$. It should be noted that these tests were done using the "time" command, and therefore include the amount of time it would take to translate the data.

The adjacency matrix approach was not chosen for it's timecomplexity, rather for it's ability to go unaffected by the amount of edges. The hypothesis was that with the use of an an adjacency matrix, generating an MST for a graph with 10^3 vertices and as many edges, should take as much time as doing the same for a graph with 10^3 vertices and 10^5 edges.

This can be confirmed by the following data:

Table 2: Edgecount test

Data: Vertex – Edge	1000 – 1000	1000 – 10000	1000 - 1000
Time: (Seconds)	10.647	9.944	10.192

The hypothesis that edges do not matter with an adjacency matrix is correct, but the reason for this is not in any way beneficial to the algorithm. With this implementation one can see that finding edges in the matrix isn't more efficient in a matrix. Instead e ends up irrelevant as finding edges always means looking through every possible combination of connections which can be made – even if there is no edge there to begin with. In other words, e is not left out because a matrix makes finding edges constant, rather because the total amount of edges, existent or not, is tied to the amount of vertices. The current implementation functions correctly, even though the times are suboptimal at best, and unusable at worst.

In summary, the current implementation uses a standard adjacency matrix. This is problematic as searching for edges requires also searching parts of the matrix that do not contain any. For this reason any extra edges do not matter not because they can be found quicker than in that of an adjacencylist, but because stops where extra edges would be stored are searched anyway.

4. Reflections

While the problems brought up in the first part of this report have been resolved, saying that the wrong tool for the job was used is an understatement. The reason for choosing an adjacencymatrix over every other solution was to have it perform well if the amount of edges outweigh the amount of vertices. This was accomplished, but the total timecomplexity was overlooked up until testing with the largest datasets. The cases in which and adjacencymatrix might outperform an $O(e\log(n))$ solution are very few, if any. The amount of uniqe edges one can create will rarely outnumber to total amount of vertices to such a degree that a matrix solution will be worth using.

Total time spent on the project is estimated to be around 70 hours, where 20 were allocated to this report. As for programming, most time was spent working on translation, to make sure this part generates as little overhead as possible. Before this part could get started however, different ways of storing the graphs were researched. The most talked about approaches are adjacency matrices and adjacency lists. How this research was done is partly to blame for how the project ended up. A number of sources discuss the advantages of both adjacency matrices as well as lists, giving the impression that both approaches are comparable in performance once implemented. During testing it was discovered that matrices perform far worse than adjacency lists, something which is easily seen when actual timecomplexity is taken into account. Why this was overlooked is as simple as it not being mentioned in any of the sources.

If the project was to be done again the changes that would be made are obvious. Adjacency matrices are not how graphs should be represented if there are any more than 100 vertices. Instead, adjacencylists, priorityqueues or any other approach with a logarithmic growth for timecomplexity should be used.

5. Attachments

```
    while (std::getline(data, line)) {
        std::istringstream sstream(line);
        sstream >> source_vertex >> destination_vertex >> weight;
        adjacency_matrix[index[source_vertex]][index[destination_vertex]] = weight;
        adjacency_matrix[index[destination_vertex]][index[source_vertex]] = weight;
    }
llk
```

Figure 1: Matrix insertion

```
  while (visited_count < vertex_count) {            // O(n)
      int cheapest = int_max, x = -1, y = -1;       // O(1)
      for (int i = 0; i < vertex_count; i++) {            // O(n)
              for (int j = 0; j < vertex_count; j++) {      // O(n)
              if (matrix[i][j] < cheapest && validedge(i,j, visited)) {     // O(1)
                  cheapest = matrix[i][j];    // O(1)
                  x = i;    // O(1)
                  y = j;    // O(1)
              }
          }
      }
```

Figure 2: Complexity Analysis