AI PRINCIPLES & TECHNIQUES

Minimum Spanning Tree Algorithm

MARIO TSATSEV *s1028415*
JONA TE LINTELO *s1021036*
Radboud University *02-10-20*

---

1 Introduction

Many search problems can be represented as graphs. To reason about these problems in their graph representation and find solutions, techniques such as finding the minimum spanning tree(MST) can be performed. MST's have a wide application, as they can help optimizing networks. It is therefore relevant to see how a famous algorithm for creating these MST's behaves under certain circumstances. A minimum spanning tree is defined as a path through a graph that connects all vertices with the smallest possible total weight of the edges. Finding a MST is a well known graph problem that is NP-complete, although there exist algorithms that have near-linear time complexity. In order to get more familiar with programming P and NP problems the task at hand was to program an implementation of Prim's algorithm and experiment with the runtime complexity. Prim's algorithm finds a MST by greedily adding vertices, it adds the currently best safe edges one-by-one whilst not creating a cycle.

2 Method

The output for the code should be a text representation of the MST that the algorithm finds in the graph. To properly represent a graph in a way that avoids cluttering, an object-oriented coding approach was used in this project. The implementation of the algorithm itself contains four core classes. The class structure is given in *figure 1.*

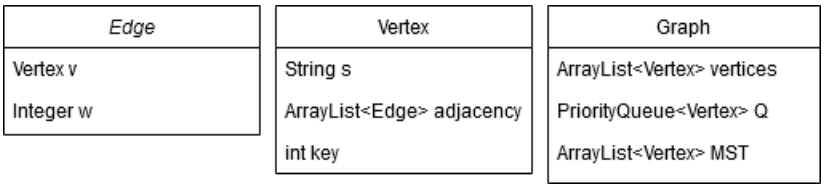

Figure 1: Class structure for the code implementation.

There are two known ways to represent a graph in programming languages, namely as adjacency lists or an adjacency matrix. For an undirected graph, adjacency matrices are simpler to work with because the indices in the matrix become the edge connections, and the values at positions *i,j* the weight. This simplicity of representation however is compensated by a higher run time because all operations on matrices like searching are in most cases $\mathcal{O}(n^2)$. For an undirected graph the runtime improves using adjacency lists at the cost of memory and more typing. Each edge from v to u must be declared twice. Once from v as a root vertex and once as u. The implementation used for this report is further explained below.

An edge in the graph is defined as a vertex(v) and weight(w). Where vertex v is the vertex to which vertex u is connected and w is the weight of the edge. An edge only has one vertex as attribute as otherwise redundancy would be created when all vertices are implemented. Regardless of the implementation of the Edge class declaring an edge twice is necessary when using lists. Were edges represented with two connected vertices a uniform list of all edges had to be separately maintained. Because the graph is undirected, two vertex representation adds no additional information at the cost of more storage.

A vertex is represented as a String(s), a list of edges(*adjacency*) and a cost(*key*). Where s is the label of the vertex, *adjacency* is a list of all connected edges to the vertex and *key* is the current key value of the vertex, initially set to infinity.

Finally, the graph is represented as a list of all vertices(*vertices*) that are contained in the graph, a priority queue(Q) that is sorted based on vertex key value and a list that will contain the vertices included in the MST(*MST*). In order for PriorityQueue to work in Java with a custom class the class has to become an instance of *comparable*. However even after the method *compereTo* is implemented the PriorityQueue in java does not work optimally for this algorithm. This is because the queue does not automatically re-sort when the key value of a vertex is changed. This can be solved by clearing and re-filling the queue after every iteration.

Prim's algorithm itself is also contained in the *Graph* class. For this implementation the pseudo code given in the slides has been the main guideline for the implementation. The pseudo code and implementation for Prim's algorithm is given directly below.

```
MST-PRIM(G, w, r)
    for each u in G.V          // including root r
        u.key = INFINITE
        u.parent = NILL
    r.key = 0
    Q = G.V
    while Q != EMPTY
        u = EXTRACT-MIN(Q)     // in 1st iteration, u = r
        for each v in G.adjacent[u]
            if v in Q and w(u, v) < v.key
                v.parent = u
                v.key = w(u, v)
```

```
public void Prim(Vertex r){
        for (Vertex v: vertices){
            if(v == r){
                v.setKey(0);
            }
        }
        Q.addAll(vertices);

        while (!Q.isEmpty()){
            Vertex u = Q.poll();
            vertices.remove(u);
            MST.add(u);
            for (Edge e: u.adjacency){
                int i = findByName(vertices,e.v);
                if ((Q.contains(e.v)) && weight(e) < vertices.get(i).key){
                    vertices.get(i).setKey(weight(e));
                    e.v.setKey(weight(e));
                    Q.clear();
                    Q.addAll(vertices);
                }
            }
        }
    }
```

To test this implementation the example graph from the slides is used. The graph from the slides is created by initializing all the vertices. The example graph is given in *figure 2*. A graph can be represented as a trio of numbers, one for each edge, contained within a *.in* file. An example of the way vertices of a graph are initialized is given in directly below.

```
1 2 4 # vertex 1(A) is connected with vertex 2(B) with weight 4
1 8 8 # vertex 1(A) is connected with vertex 8(H) with weight 8
2 1 4 # vertex 2(B) is connected with vertex 1(A) with weight 4
```

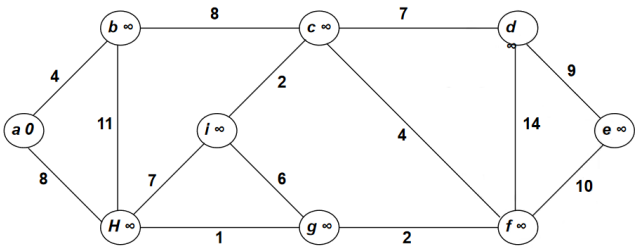

Figure 2: Graph used to test the implementation.

In order to document results it is needed to measure the run time of the algorithm. The runtime of the algorithm can be measured by incrementing an integer every time a comparison or operation is made that is related to the input of the algorithm. In this implementation of the algorithm the integer is incremented every time an edge is considered. This is a better approach than an actual timer due to the fact that the timer is very hardware specific. Different runs will lead to different results due to CPU load, address allocation in memory or language of use. In the following two sections the actual complexity of the algorithm's implementation and results are discussed.

3 Complexity

As mentioned before the algorithm is based on adjacency list graph representation with ArrayLists. ArrayLists hold indices of their elements like arrays, which makes searching linear and indexing constant, while retaining their list like properties. This makes inserting an element constant[1].

- ArrayList.get() = $\mathcal{O}(1)$
- ArrayList.add() = $\mathcal{O}(1)$
- ArrayList.remove(obj o) = $\mathcal{O}(n)$
- ArrayList.indexof() = $\mathcal{O}(n)$ where n is the length of the list.
- ArrayList.contains() = $\mathcal{O}(n)$ where n is the length of the list.

PriorityQueue is used with the following complexity on its operation:

- PriorityQueue.addAll() = $\mathcal{O}(n \log n)$
- PriorityQueue.poll() = $\mathcal{O}(\log n)$
- PriorityQueue.clear() = $\mathcal{O}(n \log n)$

The helper function *findByName* takes $\mathcal{O}(n)$ steps.

Let n be the number of elements, m be the number of adjacent vertices of each vertex, then:

$$\begin{aligned} Total &= \mathcal{O}(n * (n + \log n) + (m * (n + n \log n * n \log n))) \\ &= \mathcal{O}(n^2 + n \log n + mn + mn^2 \log n^2) \\ &= \mathcal{O}(mn^2 \log n^2) \end{aligned}$$

This complexity exceeds the typical runtime complexity for Prim's of $\mathcal{O}(|E| + |V| \log |V|)$. The complexity achieved in this implementation is technically $\mathcal{O}(|E| * |V|^2 \log |V|^2)$, however the extra complexity was accumulated in the implementation here in Java using helper functions because otherwise it was not possible to read a graph from a file and make the algorithm work at the same time.

4 Results

The results are documented as a set of answers to the given questions about the runtime of our implementation.

1. How does the runtime change for increasing numbers of vertices |V| in G?

   For an increasing number of vertices |V| in G, the runtime also increases. The algorithm has to perform more sorting and potentially a greater amount of searching in the adjacency lists for every added vertex. A correct binary heap implementation of Prim's algorithm has a runtime complexity of O(|E|log |V|), thus increasing V also increases the runtime complexity. In execution of the files *'graphslides.in'* and *'graph_increasedvertices.in'* the runtime in terms of compared edges increases from 56 to 64, when adding one extra vertex with 2 edges.

2. How does the runtime change if the number of edges |E| increases from |V| − 1 to |V| × (|V| − 1)/2 for different |V|?

   If the number of edges |E| increases from |V| − 1 to |V| × (|V| − 1)/2, the runtime also increases. When the amount of edges are increased the algorithm also has to perform more searches in the list of connected edges to find the lowest possible key. Increasing E in O(|E|log |V|) results in a higher runtime. If there are only |V| − 1 edges there is only one path to be found for the algorithm. Increasing this number means that now there are more edges to be compared, thus increasing runtime.

3. Does the runtime change in relation to the amount of equal edge weights?

   The runtime does not change in relation to the amount of equal edge weights. The algorithm will still have to make an equal amount of comparisons as it would have for a smaller amount of equal edge weights. In execution of the files *'graphslides.in'* and *'graph_equalweights.in'* both returned a runtime of 56 in terms of compared edges.

4. Does the runtime change in relation to the (minimum, maximum, average) magnitude of the edge weights?

   The runtime does not change in relation to the (minimum, maximum, average) magnitude of the edge weights. This is because the only operation done with edge weights is a comparison. Comparing large numbers with each other does not have a higher runtime complexity than comparing small numbers with each other. Only the total amount of comparisons that have to be made would increase the runtime complexity. In execution of the files *'graphslides.in'* and *'graph_largeweights.in'* both returned a runtime of 56 in terms of compared edges.

5. How does the range in edge weights effect the runtime?

   When the range in edge weights is large the runtime complexity does not change. Comparing if 1 is smaller than 2 should not take significantly more time than comparing 1 with 2000000. In execution of the files *'graphslides.in'* and *'graph_largeweights.in'* both returned a runtime of 56 in terms of compared edges.

5 Discussion

As stated earlier, Java's PriorityQueue is not optimal. A correct binary heap implementation and a adjacency list result in a runtime complexity of O(|E|log |V|). But by using a more sophisticated Fibonacci heap the complexity can be brought down to O(|E| + |V|log |V|). Or even linear time when |E| is at least |V| log |V|.

6 Conclusion

From the results the conclusions can be made that for Prim's algorithm the number of edges and the number of vertices in the graph affect the runtime complexity. Whereas the results show that increased values for edge weights do/do not affect the runtime complexity. In order to improve the implementation a different queue should be used. Additionally performance can be improved by changing the way a graph is represented, stored and printed.

7 Bonus

An implementation for generating random sparse or none-sparse undirected graphs which follows the convention for graph definition[2] is implemented in the class Generator. The convention is that three numbers are given in a line where the first two represent the vertices and the third is the weight of the edge between them. This algorithm for graph generation adheres to an implementation of an adjacency matrix. Its definition is that at index *i,j* there is an edge with weight *matrix[i][j]* and there is no edge if *matrix[i][j]* = 0. The generator is given below.

```
public class Generator {
    private int[][] matrix;
    private int vertices;
    Random r = new Random();


    public Generator(int vertices) {
        if (vertices > 17576) throw new IllegalArgumentException("out of bounds");

        this.vertices = vertices;
        this.matrix = new int[this.vertices][this.vertices];
    }
    . . .
```

The matrix is populated with random numbers at each location preserving the property that there are no cycles and that row and position i and column at position i are transposed versions of each other. This is achieved by inserting a random number at positions i,j and j,i, positions i,i are 0 meaning that there is no connection between these two edges. The sparseness is achieved by implementing a support function which more often returns 0 if a none-sparse graph is desired.

```
public int randomNumber(int max, int min, boolean sparse){
    if (sparse && r.nextBoolean()){
        return 0;
    }
    return r.nextInt(max-min+1)+min;
}
```

```
public void createGraph(int maxWeight,int minWeight,Boolean sparse){
    for (int i =0; i < vertices; i++){
        for (int j=0; j <= i; j++){
            if (i == j){
                matrix[i][j] = 0;
            }
            else {
                int w = randomNumber(maxWeight,minWeight,sparse);
                matrix[i][j] = w;
                matrix[j][i] = w;
            }
        }
    }
}
```

Reading from a file and writing to a file is done with Java's IO library. These are standard procedures and are not explained in this report. Parting the input into the data structure was quite difficult. As discussed previously the data structure is optimal for applying the algorithm on a given graph but it is not optimal to parse input into it. Java does not support meta programming. For that reason the function given directly below was used.

```
    public ArrayList<Vertex>CombineAdj(ArrayList<Vertex> V){

    ArrayList<Vertex> newV = (ArrayList<Vertex>) V.stream()
            .collect(Collectors.groupingBy(x -> x.s))
            .values().stream()
            .map(g -> {
                ArrayList<Edge> list = new ArrayList<>();
                for (Vertex i : g) {
                    ArrayList<Edge> adjacency = i.adjacency;
                    for (Edge edge : adjacency) {
                        list.add(edge);
                    }
                }
                Vertex v = new Vertex(g.get(0).s);
                v.initAdjList(list);
                return v;
            }).collect(Collectors.toList());

    return newV;
}
```

After reading the file, many of the same vertex would be created, but each would have a different adjacency list for each of its neighbors. So it was necessary to concatenate the lists inside each vertex given that the vertex was "the same" as defined by the comparable interface function *compareTo*. The way to flatten a list of vertex adjacency lists is given below.

1. Group all vertices which are equal according to the *compareTo* method. This means that vertices with the same string and key are grouped together.

```
Vertex("1") with adj(Vertex("2"),Vertex("4"))
Vertex("1") with adj(Vertex("3"))
Vertex("1") with adj(Vertex("5"),Vertex("0"))
```

2. For each group, add adjacent vertices to a single list of edges and create a new vertex with the same name as the grouped vertices, but with the updated adjacency list.
3. Return a new list of all vertices.

The full algorithm can be found at [3]

Problems with the bonus

After such a list of new vertices is created, the ID of each edge is different. Therefore they are not updated by reference in the algorithm. It was needed to add extra functions to make it work which increased time complexity.

8 References

https://www.baeldung.com/java-collections-complexity[1]

https://csacademy.com/app/graph_editor/[2]

https://github.com/mtsatsev/Prim-s-Algorithm-PriorityQueue[3]