Knowledge-based AI

Assignment 0: Minimum Spanning Trees

Alice Johnson *s1234567*
Bob Smith *s7654321*
Radboud University *September 1, 2025*

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 be used in the stock market, distribution systems, complex brain networks and more (Nagarajan & Ayyanar, 2014; Rešovskỳ et al., 2013; Stam et al., 2014). 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 (Prim, 1957). 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 avoiding creating a cycle (Prim, 1957).

2 Methods

Prim's algorithm calculates the MST of a graph. Its pseudocode is depicted in Algorithm 1. As input, the algorithm takes a graph consisting of a set of vertices and edges $G = (V, E)$, a root vertex $v_0 \in V$ and a function c that assigns a costs to each edge $e_{uv} \in E$. When the input graph is unweighted, this function is a constant. The algorithm starts by initialising the MST as a single vertex (lines 1-2). After that, all vertices from the input graph are iteratively added to the MST (lines 3-6). This is done by selecting the edge with the lowest cost that connects a vertex in the MST to a vertex that is not yet in the MST (line 4). Then this new vertex and edge are added to the MST (lines 5-6). This results in a MST, which is returned (line 7) (Bennett, 2015).

Algorithm 1: Pseudocode of Prim's algorithm (Bennett, 2015)

Input: $G = (V, E), v_0, c$
Output: $G_{MST} = (V_{MST}, E_{MST})$
1 $E_{MST} \leftarrow \emptyset$
2 $V_{MST} \leftarrow \{v_0\}$
3 **while** $V \neq V_{MST}$ **do**
4 $\quad e_{uv} \leftarrow \arg\min \{c(e_{uv}) \mid u \notin V_{MST}, v \in V_{MST}, e_{uv} \in E\}$
5 $\quad V_{MST} \leftarrow V_{MST} \cup u$
6 $\quad E_{MST} \leftarrow E_{MST} \cup e_{uv}$
7 **return** (V_{MST}, E_{MST})

The implementation of this algorithm had been made using Python3. The code of this can be seen in Figure 1. For the representation of the graph, custom graph and edge classes have been created. It is a basic implementation where the vertices and edges are stored in lists. The exact implementation can be found in Figure 4.

The entire algorithm is contained in one function called `prim_mst`. Lines 1-4 of the implementation correspond to lines 1-2 of the pseudocode, where the MST with one vertex gets initialised. In line 5, the edges of the graph get sorted based on their weight. This is done such that it is easier to find the edges with the lowest weight. Note that the weight is the same as its costs, as mentioned in the pseudocode.

```
1  def prim_mst(graph):
2      mst = Graph()
3      root = graph.vertices[0]
4      mst.add_vertex(root)
5      graph.edges.sort(key=lambda edge: edge.weight)
6
7      while len(mst.vertices) < len(graph.vertices):
8          min_edge = None
9          for edge in graph.edges:
10             if edge.vertex1 in mst.vertices and edge.vertex2 not in mst.vertices
                   or \
11                 edge.vertex1 not in mst.vertices and edge.vertex2 in mst.vertices:
12                 min_edge = edge
13                 break
14
15         mst.add_vertex(min_edge.vertex1)
16         mst.add_vertex(min_edge.vertex2)
17         mst.add_edge(min_edge.vertex1, min_edge.vertex2, min_edge.weight)
18
19     return mst
```

Figure 1: Implementation of Prim's algorithm in Python.

To test this algorithm the three simple graphs in Figure 2 be used. The algorithm will be tested for correctness and its runtime complexity will be analysed.

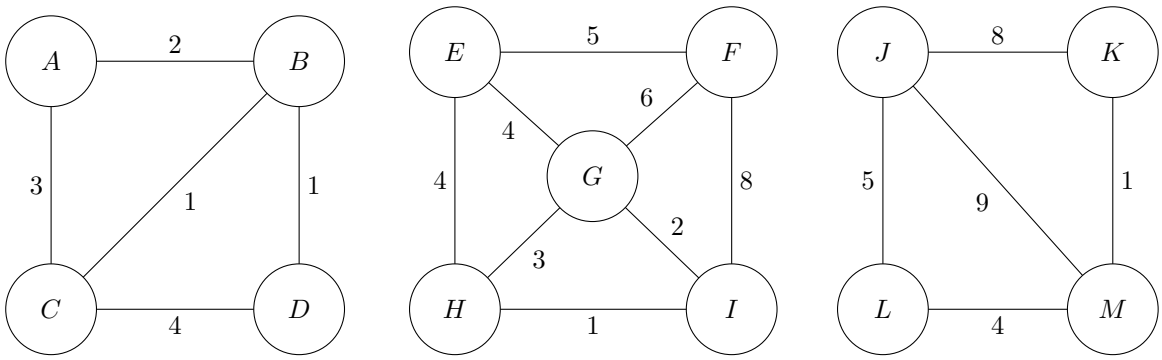

Figure 2: Three simple test graphs.

3 Results

3.1 Test graph

All the three test were completed without issues. The resulting graphs of the tests are depicted in Figure 3. For each graph, a spanning tree has been found. This can be seen immediately as the amount of vertices in the graphs are one more than the amount of edges and the graphs are fully connected. Next to that, all spanning trees are also MST's. In the second graph, there were two possible MST's but only one has been found. The other MST would has the edge (E, H) instead of (E, G). In the first and second graph, the only MST has been found.

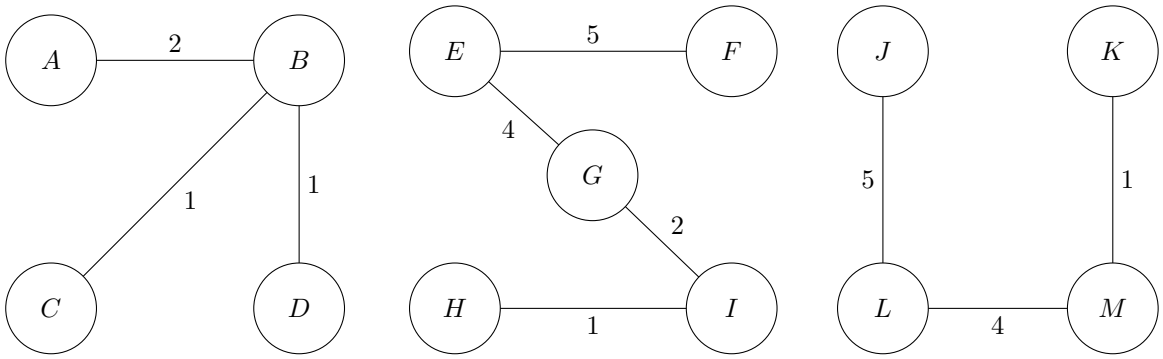

Figure 3: Results of running Prim's algorithm on the test graphs of Figure 2

3.2 Runtime complexity

In theory, the runtime complexity of Prim's algorithm can reach $\mathcal{O}(|E| \log |V|)$, where $|E|$ is the amount of edges and $|V|$ is the amount of vertices in the original graph. However, that is only if a binary heap is used to search for the next vertex to add together with an adjacency list to store the graph (Tarjan, 1983).

The runtime complexity of our implementation is $\mathcal{O}(|E||V|^2)$. This is calculated as follows. Lines 1-5 in Figure 4 have a runtime complexity of $\mathcal{O}(|E| \log |E|)$, because the list of edges gets sorted and the other lines run in constant time. The main loop (lines 7-17), goes over all remaining vertices, thus completing $|V| - 1$ iterations. The for-loop inside of the main while-loop (lines 9-13) goes over all edges in the original graph, thus having $|E|$ iterations. Within each iteration, there is an if-statement that checks whether the edge's vertices are already contained in the MST's list of vertices. In the worst case scenario, this has a runtime complexity of $\mathcal{O}(|V|)$. Combining this gives a runtime complexity of $\mathcal{O}(|E||V|^2)$ for the main loop. This leads tot he total runtime complexity being $\mathcal{O}(|E| \log |E| + |E||V|^2) = \mathcal{O}(|E||V|^2)$. This approach is also worse than using a graph with an adjacency matrix, as the runtime complexity for such an implementation would be $\mathcal{O}(|V|^2)$ (Tarjan, 1983).

4 Discussion

The algorithm works well, as it has been able to find the MST of each of the three test graphs. For the second graph, where there were multiple MST's, it did only return a single one. This is by design, but it might be more useful in certain cases to have access to all possible MST's. The implementation could be extended to incorporate this as well. However, that would likely

be at the cost of its efficiency as more combinations of possible MST's need to be considered. The runtime complexity of the implementation is also not ideal. This is partially due to the implementation of the graph structure. As all vertices and edges are stored in a list, instead of more useful structures like an adjacency matrix. Using a binary heap or priority queue can also lead to improvement of the runtime complexity. Especially if combines with a better graph implementation.

A possible improvement could also be implementing heuristics that could greatly reduce the search cost for the lowest cost edge. This is especially useful in larger and more dense graphs. As for now all edges are considered in the search, regardless whether they are already in the MST, connect two vertices within the MST or two vertices both not in the MST. The sorting of the edges based on their costs does already serve as heuristic and likely leads to a better average runtime complexity than the original algorithm, but this might not be enough in larger graphs.

5 Conclusion

The implementation of Prim's algorithm that is proposed in this report does work, however it is not efficient as it has a runtime complexity of $\mathcal{O}(|E||V|^2)$. That is due to being a very simple and non-optimised implementation that depends on a suboptimal graph implementation. It could benefit from improvements such as introducing adjacency matrices or lists and using a binary heap to help search for the lowest cost edges.

References

Bennett, N. (2015). *Introduction to algorithms and pseudocode.*

Nagarajan, A., & Ayyanar, R. (2014). Application of minimum spanning tree algorithm for network reduction of distribution systems. *2014 North American Power Symposium (NAPS)*, 1–5. https://doi.org/10.1109/naps.2014.6965353

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, *36*(6), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

Rešovskỳ, M., Horváth, D., Gazda, V., & Siničáková, M. (2013). Minimum spanning tree application in the currency market. *Biatec*, *21*(7), 21–23.

Stam, C., Tewarie, P., Van Dellen, E., van Straaten, E., Hillebrand, A., & Van Mieghem, P. (2014). The trees and the forest: Characterization of complex brain networks with minimum spanning trees. *International Journal of Psychophysiology*, *92*(3), 129–138. https://doi.org/10.1016/j.ijpsycho.2014.04.001

Tarjan, R. E. (1983). Data structures and network algorithms. Society for Industrial; Applied Mathematics.

A Graph implementation

```
1  class Edge:
2      def __init__(self, vertex1, vertex2, weight):
3          self.vertex1 = vertex1
4          self.vertex2 = vertex2
5          self.weight = weight
6
7  class Graph:
8      def __init__(self):
9          self.vertices = []
10         self.edges = []
11
12     def add_vertex(self, vertex):
13         if vertex not in self.vertices:
14             self.vertices.append(vertex)
15
16     def add_edge(self, vertex1, vertex2, weight):
17         assert vertex1 in self.vertices and vertex2 in self.vertices, "Both
               vertices must be in the graph"
18         self.edges.append(Edge(vertex1, vertex2, weight))
19
20     def __str__(self):
21         string = "Vertices:\n\t" + " ".join(self.vertices) + "\nEdges:\n"
22         for edge in self.edges:
23             string += f"\t{edge.vertex1} -- {edge.weight} -- {edge.vertex2}\n"
24         return string
```

Figure 4: The implementation of the graph and edge in python. The vertices are strings corresponding to their names.