AI: Principles & Techniques
Programming Task 1; MST Programming

Jordy Aaldering, s1004292

October 2 2020

1 Introduction

Prim's algorithm is an algorithm for finding a minimum spanning tree (MST) for weighted undirected graphs. This report shows an implementation of this algorithm in Python and evaluates the effects of different graphs on its runtime.

2 Methods

Prim's algorithm first chooses an arbitrary vertex as the root of the MST. Then for each step of the algorithm, of all edges connecting the MST to vertices not in the MST, the edge with the minimum weight is chosen. The destination vertex of this edge is than added to the MST. Since every vertex has to be added and the root is initially in the tree, there are always $|V| - 1$ steps.

Below is the implementation of this algorithm in Python. The function `prims_mst` takes a graph and a potential root. and returns the list of all parent vertices for the vertices of the corresponding indices in the list. A custom type `Graph` is made for improved readability. Graphs are represented in an adjacency matrix representation. This is because graphs can potentially be very dense, which would make lookup in an adjacency list representation very slow. It is also easier to represent edge weights in an adjacency matrix representation. A downside is that graphs are undirected, so a slightly more than half of the space in the adjacency matrix is wasted as it is mirrored against the diagonal, and with the diagonal consisting of only zeros.

```
Graph = List[List[int]]

def prims_mst(graph: Graph, root: int = 0) -> List[int]:
    size = len(graph)
    keys = [Inf] * size
    parents = [-1] * size
    in_mst = [False] * size

    keys[root] = 0

    for _ in range(size):
        u = extract_min(keys, in_mst)

        for v in range(size):
            if not in_mst[v] and graph[u][v] > 0 and keys[v] > graph[u][v]:
                keys[v] = graph[u][v]
                parents[v] = u
```

The minimum vertex is found by looking at all edges connecting the MST to vertices not in the MST, and choosing the edge with the lowest weight. The function `extract_min` returns the index of this vertex and is defined as:

```
def extract_min(keys: List[int], in_mst: List[bool]) -> int:
    min_weight = Inf
    min_index = -1

    for v in range(len(keys)):
        if not in_mst[v] and keys[v] < min_weight:
            min_weight = keys[v]
            min_index = v

    in_mst[min_index] = True
    return min_index
```

The source code can be found in the attachments. This code also contains functions for generating random graphs (`Graph.py`) and evaluating the results (`Eval.py`).

3 Results

Runtime is measures using Python's `time` package. We will first look at the effects of increasing the number of vertices in the graph. The result can be seen in figure 1. Here a density of 50% is chosen, meaning that every vertex has an edge to half of all other vertices. Every step is repeated twenty times to reduce error. As expected, the runtime grows exponentially with the number of vertices.

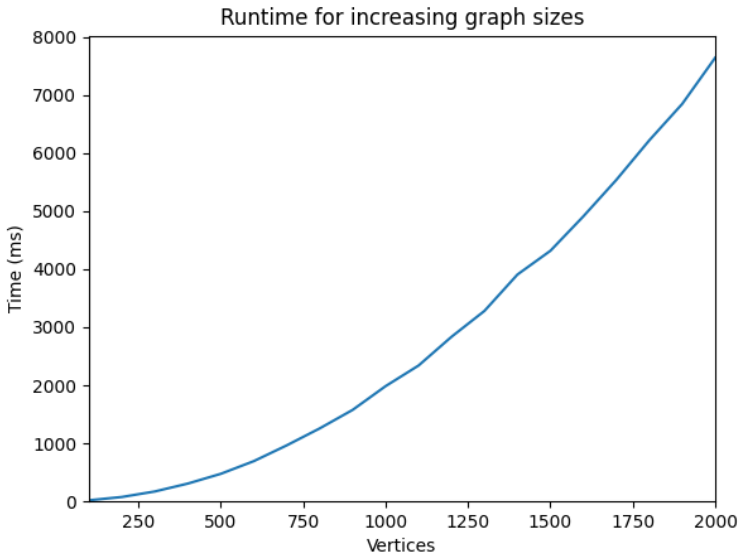

Figure 1: Runtime for increasing graph sizes

The effects of changing the density of the graph, the maximum edge weight, and the range and magnitude of the edge weights have also been evaluated. But as can be seen in figure 2 below, these did not impact the performance of the algorithm in a meaningful way.

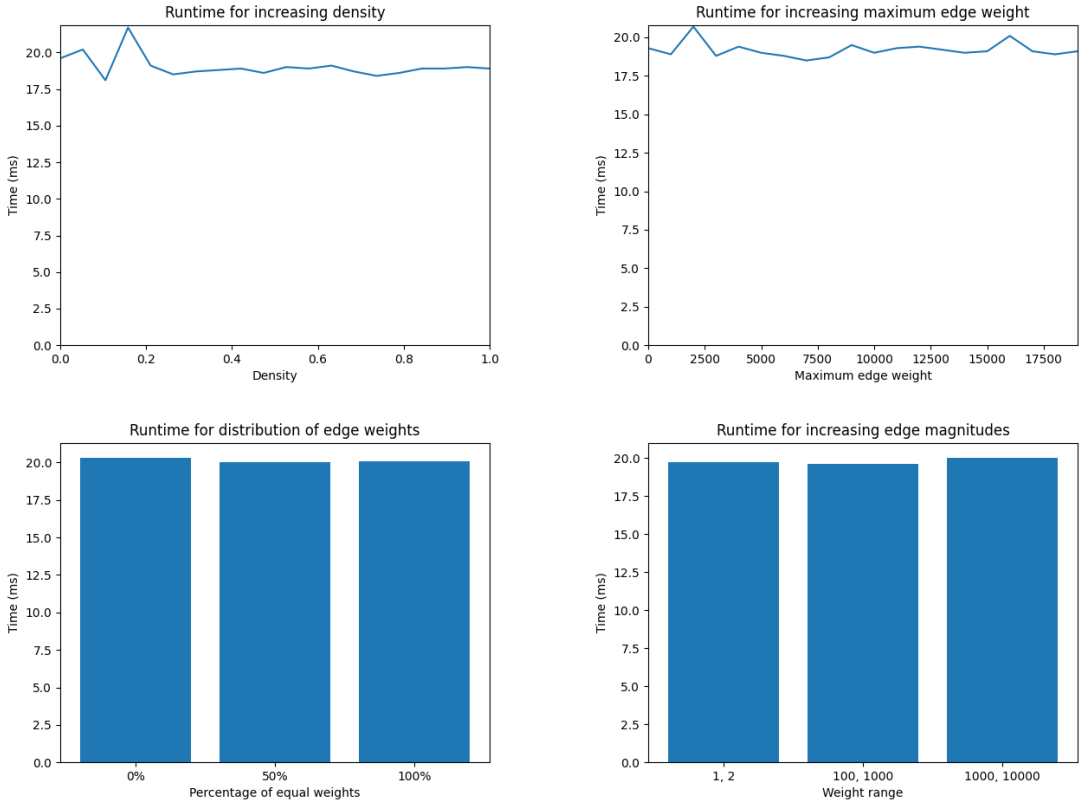

Figure 2: Effects of changing the density of the graph, the maximum edge weight, and the range and magnitude of the edge weights.

4 Discussion

This implementation of Prim's algorithm in Python is efficient and very easily readable. The implementation does not depend on factors other than the number of vertices, meaning that it is useful for varying graphs.