Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

REPORT

Laboratory work no.7
Prim and Kruskal algorithms

Elaborated:
st. gr. FAF-213 Gutu Dinu

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS 3
- Objective 3
- Tasks 3
- Theoretical notes 3
- Introduction 3
- Comparison Metric 4
- Input Format 4

IMPLEMENTATION 7
- Prim algorithm 7
- Kruskal algorithm 10
- All Algorithms 14

CONCLUSION 15

BIBLIOGRAPHY 15

ALGORITHM ANALYSIS

Objective

1. Analysis and study of the algorithms.
2. Empirical analysis of the aforementioned algorithms.

Tasks

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical notes

Empirical analysis is an alternative to mathematical analysis in evaluating algorithm complexity. It can be used to gain insight into the complexity class of an algorithm, compare the efficiency of different algorithms for solving similar problems, evaluate the performance of different implementations of the same algorithm, or examine the efficiency of an algorithm on a specific computer. The typical steps in empirical analysis include defining the purpose of the analysis, choosing a metric for efficiency such as number of operations or execution time, determining the properties of the input data, implementing the algorithm in a programming language, generating test data, running the program on the test data, and analyzing the results. The choice of efficiency metric depends on the purpose of the analysis, with number of operations being appropriate for complexity class evaluation and execution time being more relevant for implementation performance. After the program is run, the results are recorded and synthesized through calculations of statistics or by plotting a graph of problem size against efficiency measure.

Introduction

Graphs are widely used to model a variety of systems and relationships in real-world problems, such as computer networks, transportation systems, social networks, and many others. One important problem in graph theory is finding the minimum spanning tree (MST) of a weighted undirected graph, which is a tree that spans all the vertices of the graph with the minimum possible total edge weight. The MST has many applications, such as optimizing network design, minimizing communication costs, and clustering data.

Prim's and Kruskal's algorithms are two popular and efficient algorithms for finding the MST of a graph. Prim's algorithm is a greedy algorithm that starts with a single vertex and adds the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST, until all vertices are included in the MST. Kruskal's algorithm is also a greedy algorithm that starts with a forest of isolated vertices and repeatedly adds the minimum weight edge that connects two different trees in the forest, until all vertices are in the same tree.

Both algorithms have a time complexity of O(E log V), where E is the number of edges and V is the number of vertices in the graph, making them suitable for large-scale graphs. However, they have different advantages and disadvantages depending on the characteristics of the graph, such as sparsity, connectivity, and edge weights.

In this report, we will compare and analyze the performance of Prim's and Kruskal's algorithms on different types of graphs, and discuss their strengths, weaknesses, and applications. We will also present some variations and extensions of the algorithms, and discuss some open research problems in this area.

Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format

The algorithms will take as input a graph which will be having increasing number of nodes ranging from 10 up to 200000 and will output the time it takes to create the minimal spanning tree.

Here is an example:

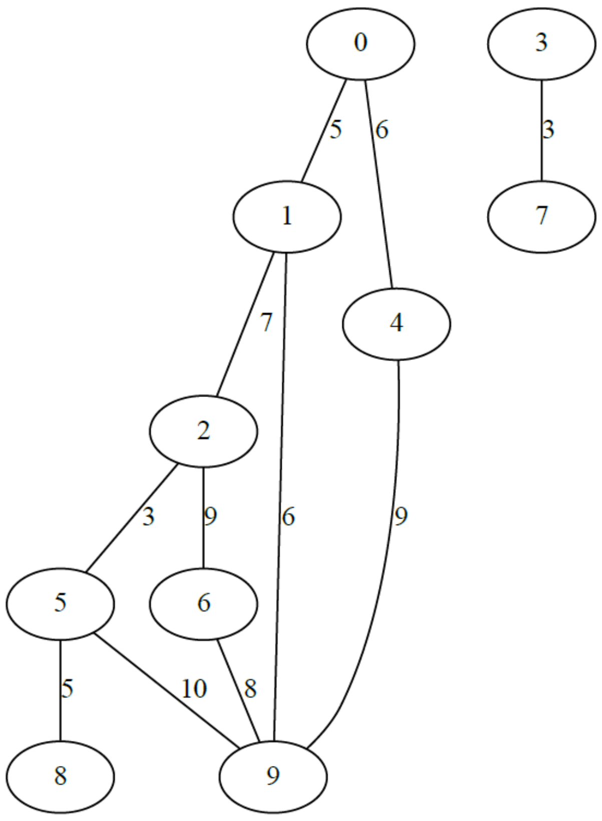


Figure 1. Example graph

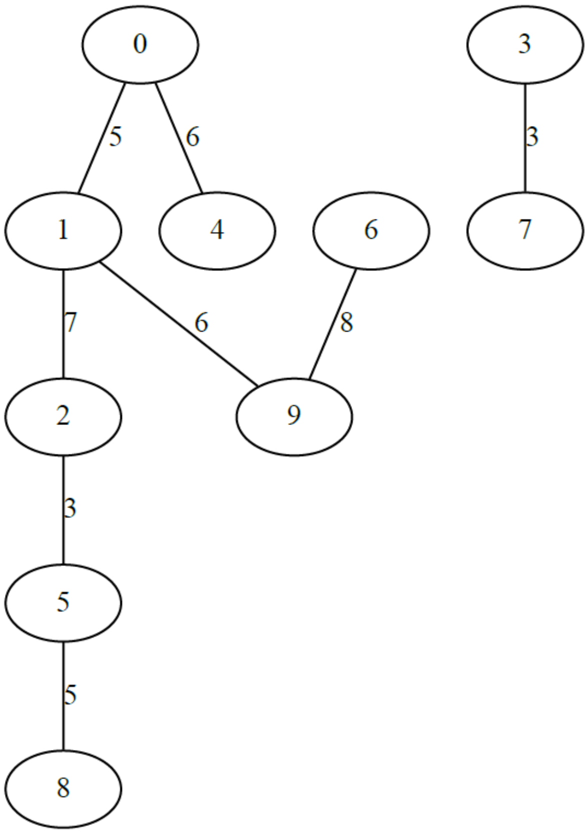


Figure 2. Kruskal minimal spanning tree

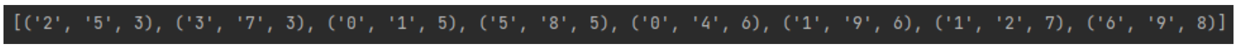

```
[('2', '5', 3), ('3', '7', 3), ('0', '1', 5), ('5', '8', 5), ('0', '4', 6), ('1', '9', 6), ('1', '2', 7), ('6', '9', 8)]
```

Figure 3. The mst for Kruskal algorithm

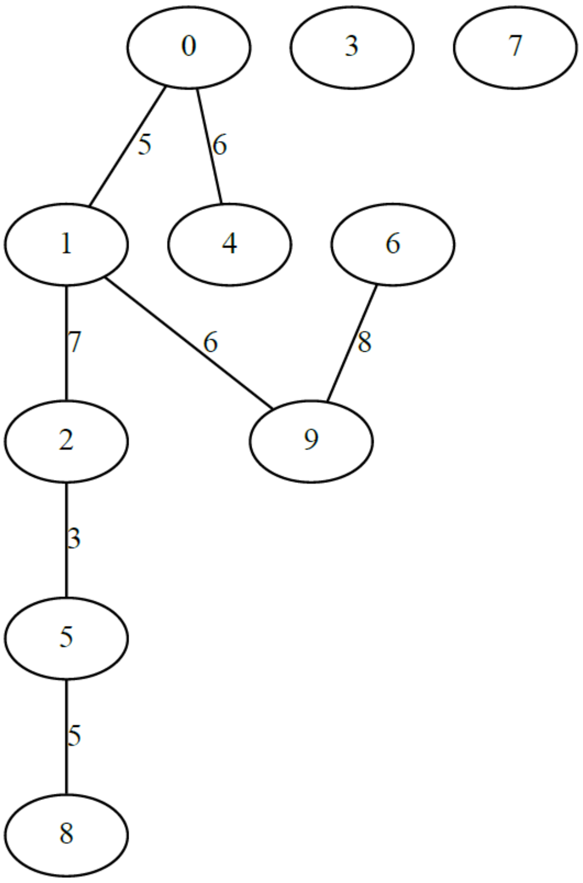
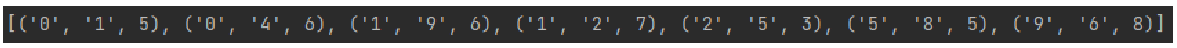


Figure 4. Prim minimal spanning tree



```
[('0', '1', 5), ('0', '4', 6), ('1', '9', 6), ('1', '2', 7), ('2', '5', 3), ('5', '8', 5), ('9', '6', 8)]
```

Figure 5. The mst for Prim algorithm

IMPLEMENTATION

Prim algorithm

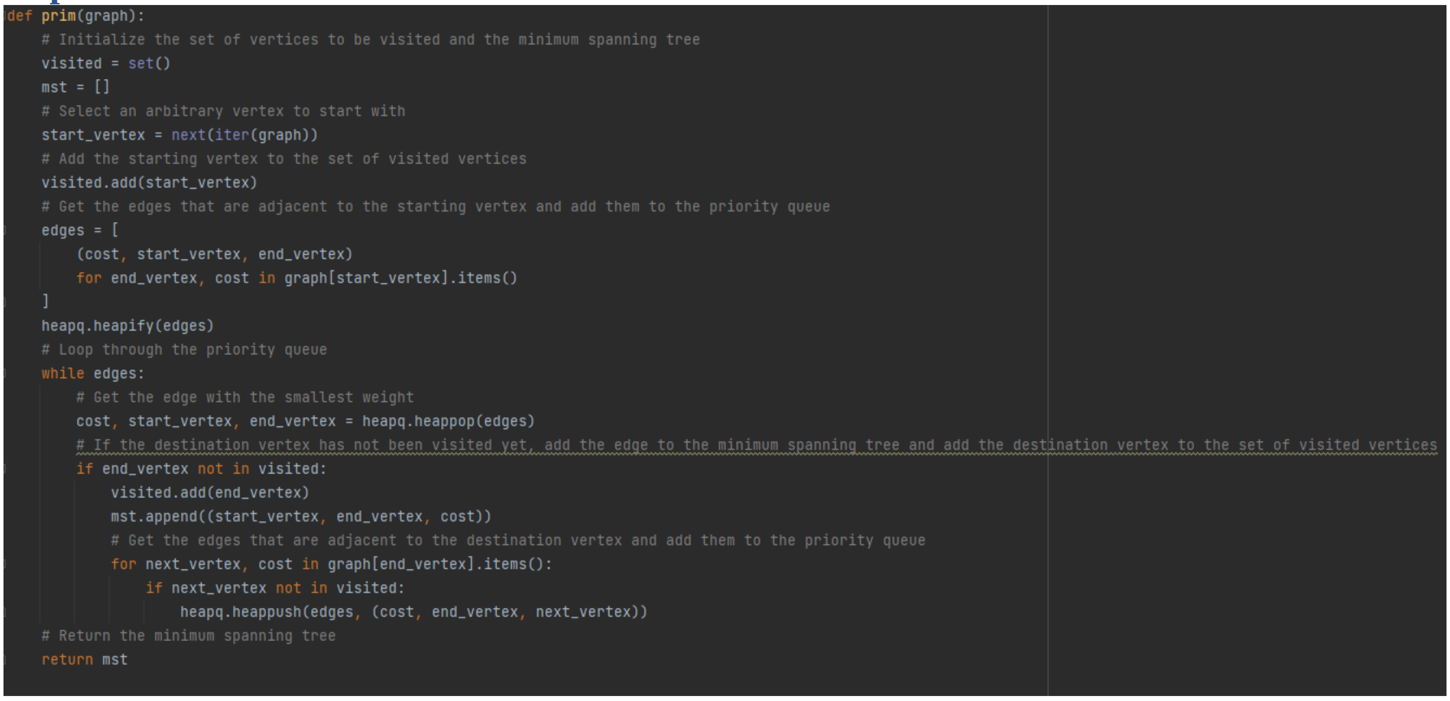
In computer science, **Prim's algorithm** (also known as Jarník's algorithm) is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Algorithm Description:

Pseudocode:

```
1. Associate with each vertex v of the graph a number C[v]
   (the cheapest cost of a connection to v) and an edge E[v]
   (the edge providing that cheapest connection). To
   initialize these values, set all values of C[v] to +∞ (or
   to any number larger than the maximum edge weight) and set
   each E[v] to a special flag value indicating that there is
   no edge connecting v to earlier vertices.
2. Initialize an empty forest F and a set Q of vertices that
   have not yet been included in F (initially, all vertices).
3. Repeat the following steps until Q is empty:
     a. Find and remove a vertex v from Q having the minimum
        possible value of C[v]
     b. Add v to F
     c. Loop over the edges vw connecting v to other vertices
        w. For each such edge, if w still belongs to Q and vw
        has smaller weight than C[w], perform the following
        steps:
          i. Set C[w] to the cost of edge vw
         ii. Set E[w] to point to edge vw.
4. Return F
```

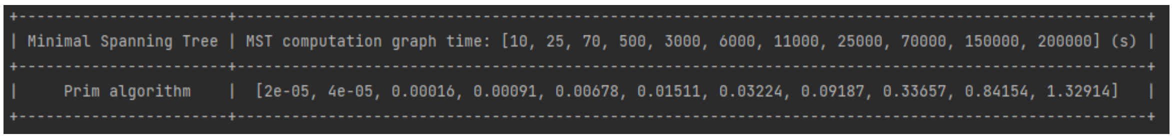
Implementation



```python
def prim(graph):
    # Initialize the set of vertices to be visited and the minimum spanning tree
    visited = set()
    mst = []
    # Select an arbitrary vertex to start with
    start_vertex = next(iter(graph))
    # Add the starting vertex to the set of visited vertices
    visited.add(start_vertex)
    # Get the edges that are adjacent to the starting vertex and add them to the priority queue
    edges = [
        (cost, start_vertex, end_vertex)
        for end_vertex, cost in graph[start_vertex].items()
    ]
    heapq.heapify(edges)
    # Loop through the priority queue
    while edges:
        # Get the edge with the smallest weight
        cost, start_vertex, end_vertex = heapq.heappop(edges)
        # If the destination vertex has not been visited yet, add the edge to the minimum spanning tree and add the destination vertex to the set of visited vertices
        if end_vertex not in visited:
            visited.add(end_vertex)
            mst.append((start_vertex, end_vertex, cost))
            # Get the edges that are adjacent to the destination vertex and add them to the priority queue
            for next_vertex, cost in graph[end_vertex].items():
                if next_vertex not in visited:
                    heapq.heappush(edges, (cost, end_vertex, next_vertex))
    # Return the minimum spanning tree
    return mst
```

Figure 6. Prim algorithm

Results



```
+----------------------+-------------------------------------------------------------------------------------+
| Minimal Spanning Tree | MST computation graph time: [10, 25, 70, 500, 3000, 6000, 11000, 25000, 70000, 150000, 200000] (s) |
+----------------------+-------------------------------------------------------------------------------------+
|     Prim algorithm    |  [2e-05, 4e-05, 0.00016, 0.00091, 0.00678, 0.01511, 0.03224, 0.09187, 0.33657, 0.84154, 1.32914]   |
+----------------------+-------------------------------------------------------------------------------------+
```

Figure 7. Prim results cases

The time complexity of Prim's algorithm is O(E log V), where E is the number of edges and V is the number of vertices in the graph. The space complexity is also O(E+V), which is the space required to store the visited vertices, the minimum spanning tree, and the priority queue used in the algorithm.

In the algorithm, the vertices are visited one by one, and for each visited vertex, its adjacent edges are checked. The adjacent edges are then pushed to the priority queue, and the edge with the smallest weight is popped out. If the destination vertex of the edge has not been visited, it is added to the minimum spanning tree and visited set, and its adjacent edges are pushed to the priority queue. This process is repeated until all the vertices have been visited.

The time complexity of the algorithm depends on the implementation of the priority queue. In the above implementation, a heap is used as the priority queue, which gives us a time complexity of O(E log V). If a simple array is used instead of a heap, the time complexity would increase to O(V^2).

Graphs

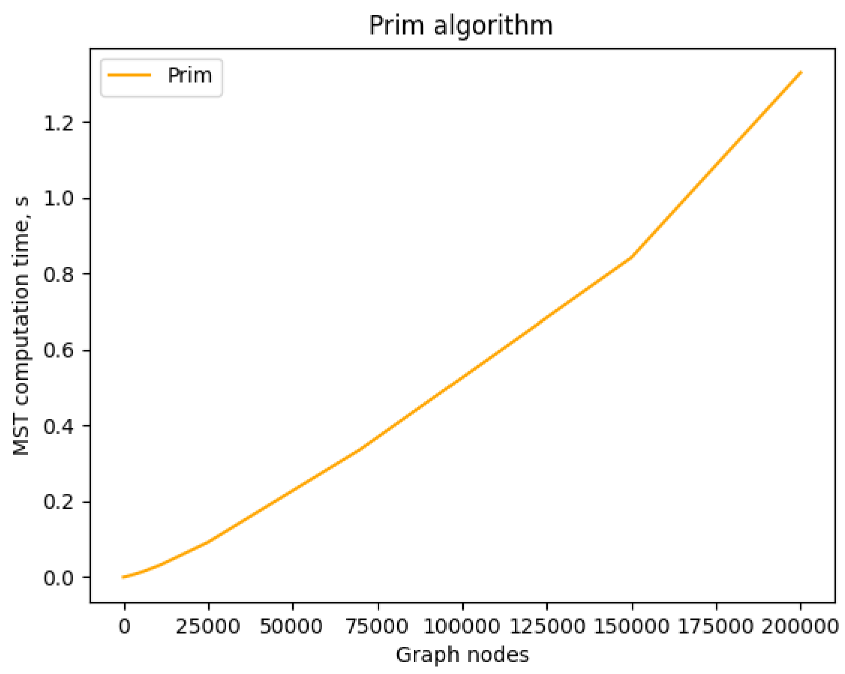


Figure 8. Prim results graph

Kruskal algorithm

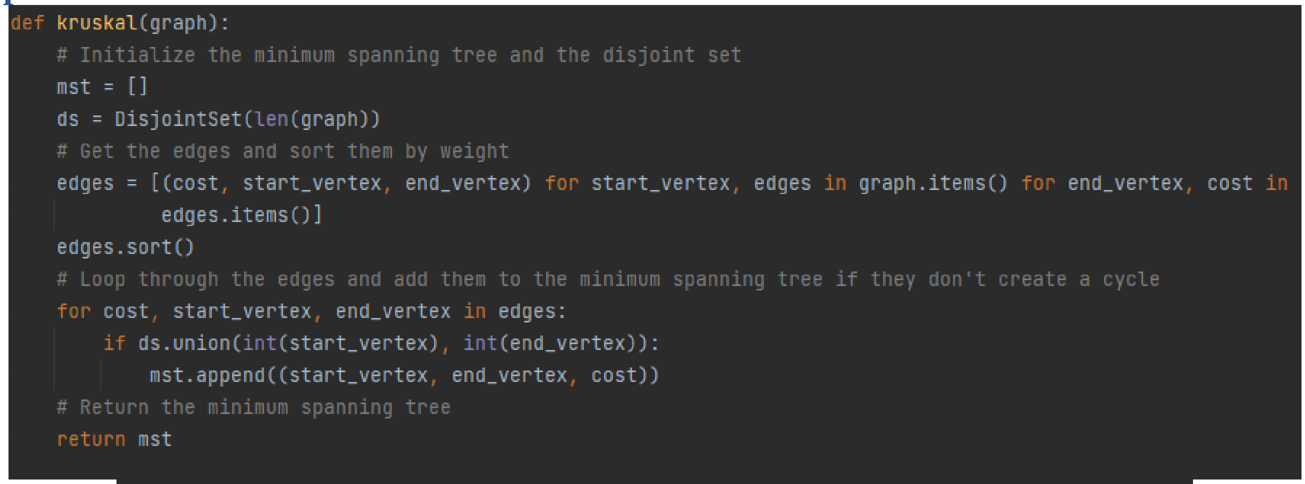
Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest

Algorithm Description:

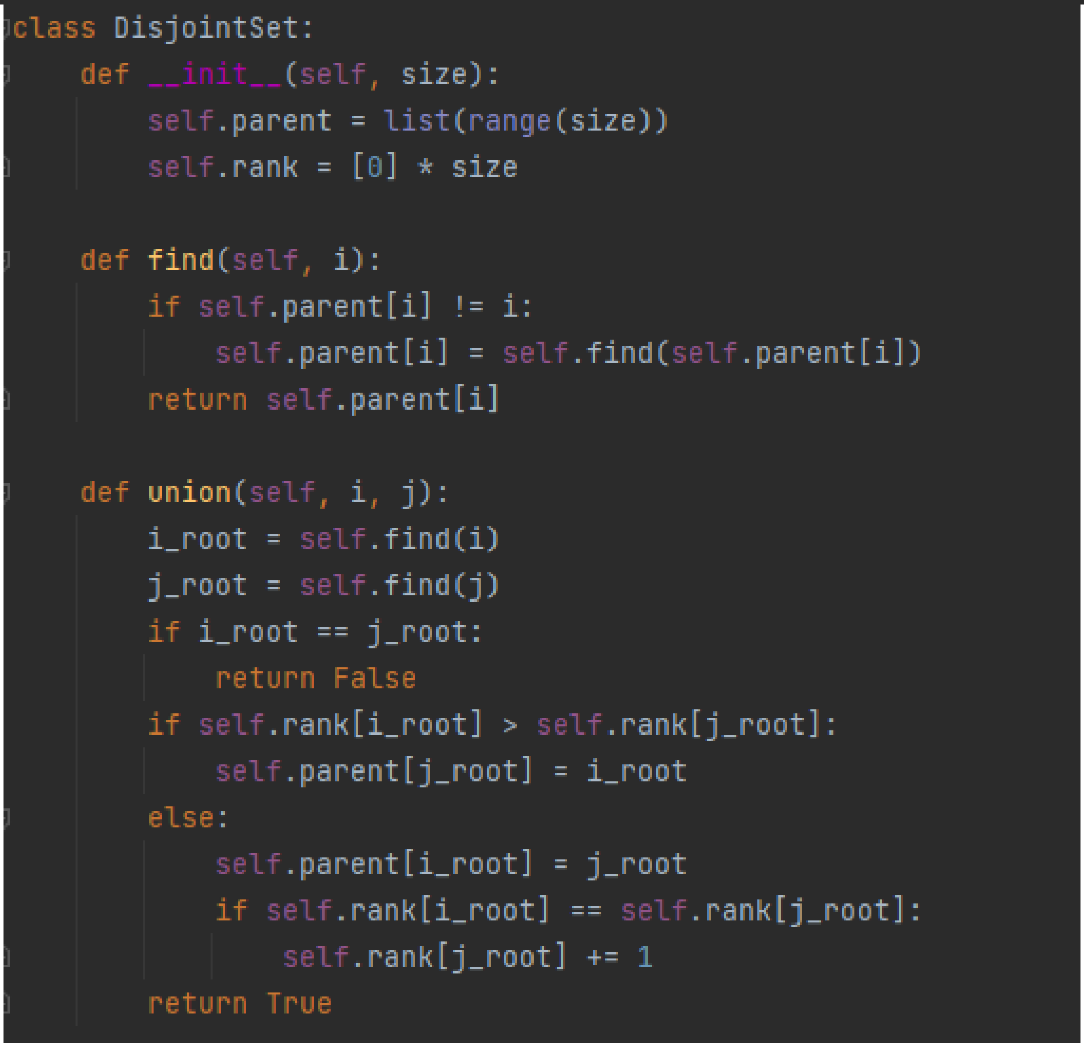
Pseudocode:

```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing
do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)} ∪ {(v, u)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Implementation



```python
def kruskal(graph):
    # Initialize the minimum spanning tree and the disjoint set
    mst = []
    ds = DisjointSet(len(graph))
    # Get the edges and sort them by weight
    edges = [(cost, start_vertex, end_vertex) for start_vertex, edges in graph.items() for end_vertex, cost in
             edges.items()]
    edges.sort()
    # Loop through the edges and add them to the minimum spanning tree if they don't create a cycle
    for cost, start_vertex, end_vertex in edges:
        if ds.union(int(start_vertex), int(end_vertex)):
            mst.append((start_vertex, end_vertex, cost))
    # Return the minimum spanning tree
    return mst
```



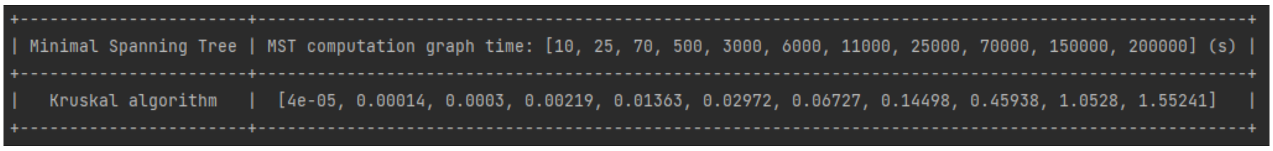
```python
class DisjointSet:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size

    def find(self, i):
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        i_root = self.find(i)
        j_root = self.find(j)
        if i_root == j_root:
            return False
        if self.rank[i_root] > self.rank[j_root]:
            self.parent[j_root] = i_root
        else:
            self.parent[i_root] = j_root
            if self.rank[i_root] == self.rank[j_root]:
                self.rank[j_root] += 1
        return True
```

Figure 9. Kruskal algorithm

Results



```
+-----------------------+----------------------------------------------------------------------------------------------------+
| Minimal Spanning Tree | MST computation graph time: [10, 25, 70, 500, 3000, 6000, 11000, 25000, 70000, 150000, 200000] (s) |
+-----------------------+----------------------------------------------------------------------------------------------------+
|   Kruskal algorithm   |  [4e-05, 0.00014, 0.0003, 0.00219, 0.01363, 0.02972, 0.06727, 0.14498, 0.45938, 1.0528, 1.55241]   |
+-----------------------+----------------------------------------------------------------------------------------------------+
```

Figure 10. Kruskal results

For a graph with *E* edges and *V* vertices, Kruskal's algorithm can be shown to run in *O*(*E* log *E*) time, or equivalently, *O*(*E* log *V*) time, all with simple data structures. These running times are equivalent because:

- *E* is at most V^2 and $\log V^2 = 2\log V \in O(\log V)$.
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain *V* ≤ 2*E*, so log *V* is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in *O*(*E* log *E*) time; this allows the step "remove an edge with minimum weight from *S*" to operate in constant time. Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We place each vertex into its own disjoint set, which takes O(*V*) operations. Finally, in worst case, we need to iterate through all edges, and for each edge we need to do two 'find' operations and possibly one union. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(*E*) operations in *O*(*E* log *V*) time. Thus the total time is *O*(*E* log *E*) = *O*(*E* log *V*).

Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use a more sophisticated disjoint-set data structure to run in *O*(*E* α(*V*)) time, where α is the extremely slowly growing inverse of the single-valued Ackermann function.

Graphs

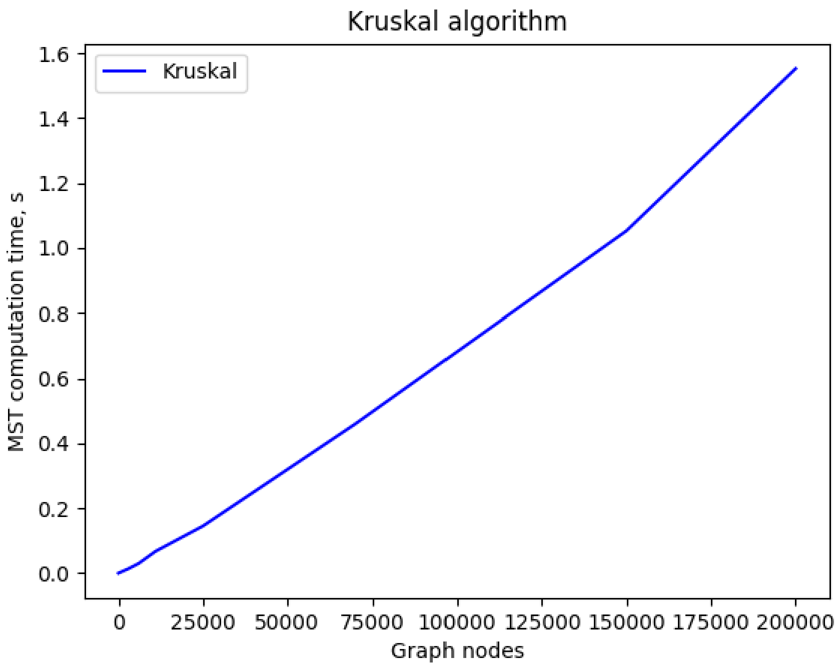
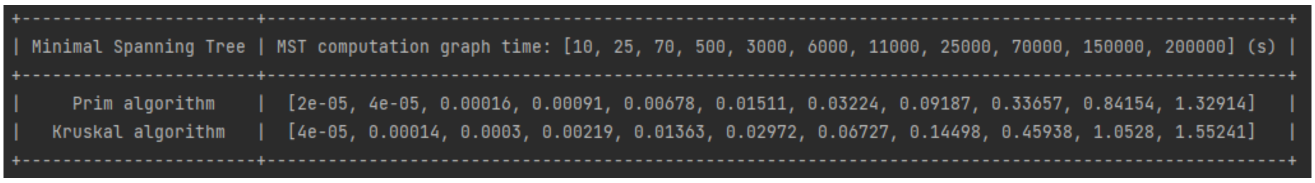


Figure 11. Kruskal results graph

All Algorithms

All results



```
+-----------------------+----------------------------------------------------------------------------------------------------+
| Minimal Spanning Tree | MST computation graph time: [10, 25, 70, 500, 3000, 6000, 11000, 25000, 70000, 150000, 200000] (s) |
+-----------------------+----------------------------------------------------------------------------------------------------+
|     Prim algorithm    |  [2e-05, 4e-05, 0.00016, 0.00091, 0.00678, 0.01511, 0.03224, 0.09187, 0.33657, 0.84154, 1.32914]   |
|   Kruskal algorithm   |  [4e-05, 0.00014, 0.0003, 0.00219, 0.01363, 0.02972, 0.06727, 0.14498, 0.45938, 1.0528, 1.55241]   |
+-----------------------+----------------------------------------------------------------------------------------------------+
```

Figure 12. All algortihms' results

All graphs

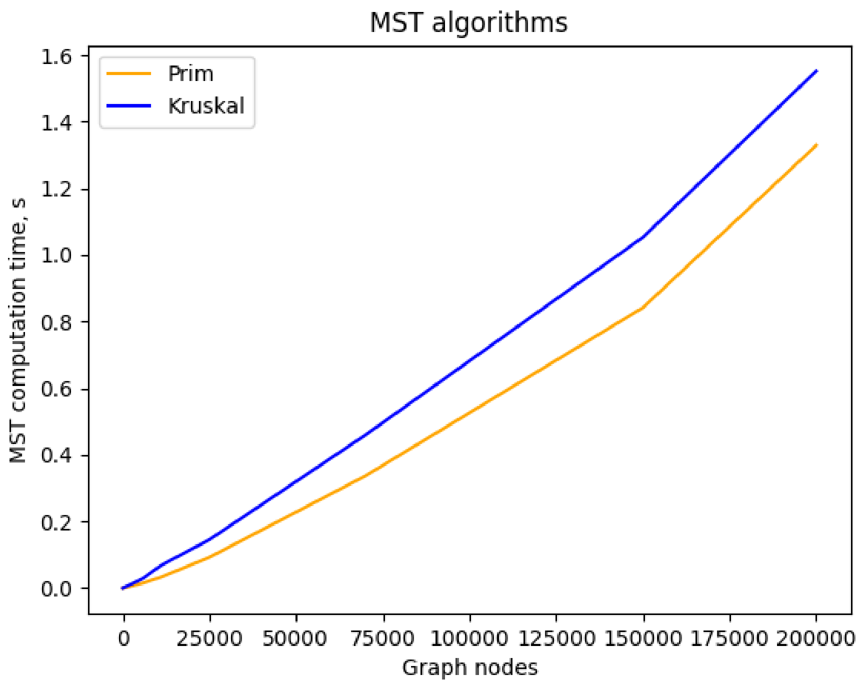


Figure 13. All algortihms graph

CONCLUSION

In conclusion, Prim and Kruskal algorithms are both efficient methods for finding the minimum spanning tree of a weighted undirected graph. Both algorithms have a time complexity of O(E log V), where E is the number of edges and V is the number of vertices in the graph.

However, there are some differences in the performance of the two algorithms. Prim's algorithm tends to be faster when the graph is dense, that is, when the number of edges is close to the maximum possible number of edges. This is because Prim's algorithm involves adding edges to a priority queue, which has a time complexity of O(log V), and a dense graph will have a larger number of edges.

On the other hand, Kruskal's algorithm tends to be faster when the graph is sparse, that is, when the number of edges is much smaller than the maximum possible number of edges. This is because Kruskal's algorithm involves sorting the edges, which has a time complexity of O(E log E), but a sparse graph will have a smaller number of edges.

In terms of implementation, both algorithms are relatively easy to understand and implement. Prim's algorithm involves adding edges to a priority queue and selecting the edge with the smallest weight, while Kruskal's algorithm involves sorting the edges and using a disjoint set to check for cycles.

In summary, the choice between Prim and Kruskal algorithms depends on the characteristics of the graph being analyzed. If the graph is dense, Prim's algorithm may be a better choice, while if the graph is sparse, Kruskal's algorithm may be more efficient.

BIBLIOGRAPHY

Github: https://github.com/Grena30/APA_Labs/tree/main/Lab7