Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 7

Discipline: Algorithms’ Analysis

Topic: Greedy Algorithms

Elaborated: Tudor Sclifos,

student group FAF-211

Verified: Fiștic Cristofor,

university assistant

Chișinău – 2023

Algorithm analysis

**Objective:**

Implement algorithms Prim and Kruskal using greedy programming**.**

**Tasks:**

1 Study the greedy algorithm design technique.

2 To implement in a programming language algorithms Prim and Kruskal.

3 Empirical analyses of the Kruskal and Prim

4 Increase the number of nodes in graph and analyze how this influences the algorithms. Make a graphical presentation of the data obtained

5 To make a report.

**Theoretical Notes:**

The Prim's algorithm, also known as Prim's minimum spanning tree algorithm, is a greedy algorithm used to find the minimum spanning tree (MST) of a connected weighted graph.

Prim's algorithm starts with an arbitrary vertex and grows the minimum spanning tree one edge at a time. At each step, it selects the edge with the smallest weight that connects a vertex from the existing tree to a vertex outside the tree. This process continues until all the vertices are included in the tree, resulting in a minimum spanning tree.

One of the advantages of Prim's algorithm is that it works well for dense graphs, where the number of edges is close to the maximum possible number of edges. It has a time complexity of O(V^2) using an adjacency matrix representation, where V is the number of vertices. However, with the use of a binary heap or a Fibonacci heap as a priority queue, the time complexity can be reduced to O(E log V), where E is the number of edges.

Kruskal's algorithm is another widely used algorithm for finding the minimum spanning tree (MST) of a weighted undirected graph. Similar to Prim's algorithm, Kruskal's algorithm seeks to connect all vertices of the graph while minimizing the total weight of the edges.

One of the advantages of Kruskal's algorithm is its simplicity and efficiency. It has a time complexity of O(E log E), where E is the number of edges in the graph. This makes it suitable for large-scale problems. Additionally, Kruskal's algorithm can handle disconnected graphs and graphs with weighted edges that are not necessarily positive.

**Introduction:**

In this laboratory work I have to implement prim and kruskal algorithms and apply them on a sparse and a dense graph. Also I need to analyse the outputs.

**Comparison metric:**

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

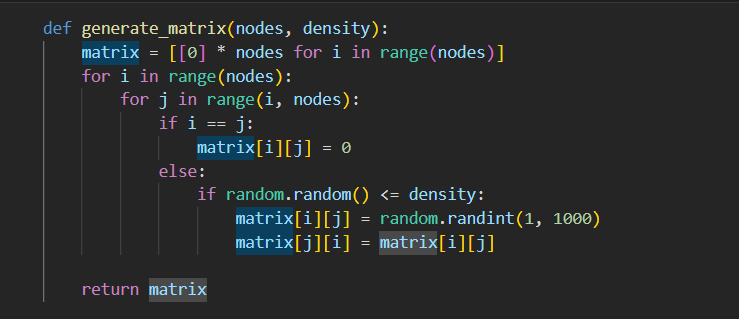
**Input format:**

The input for this laboratory work are the list of nodes for graphs, one dense and another sparse.

**IMPLEMENTATION**

**Graph Generator**

In summary, this function generates a matrix representation of a weighted graph by randomly assigning edge weights based on a given density. The higher the density value, the more likely it is to have edges between vertices, resulting in a denser graph.



**Figure 1. Implementation Graph**

**Prim**

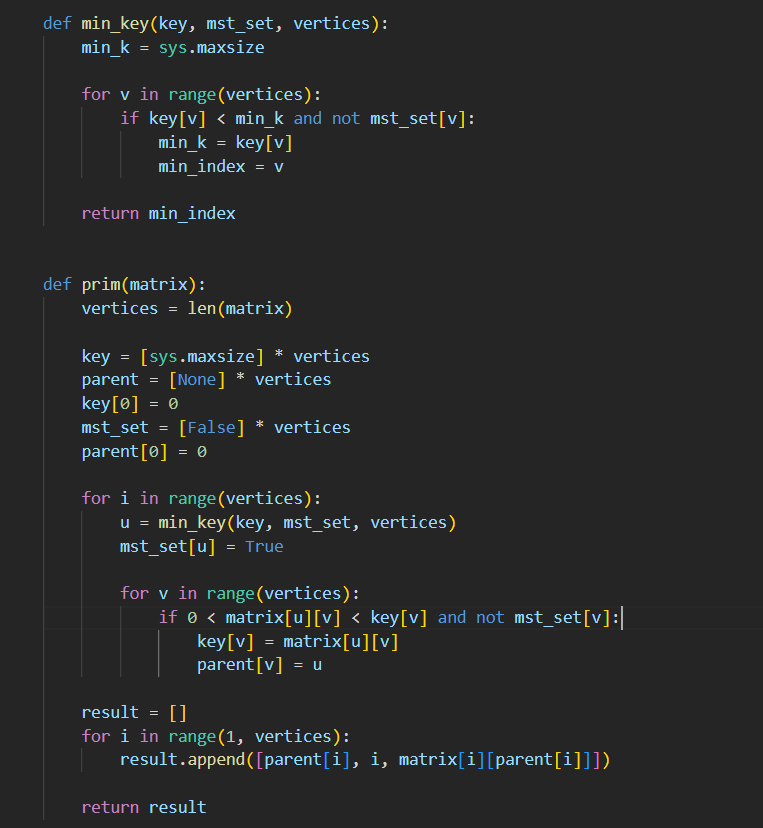
**Implementation:**

The function min\_key is defined to find the vertex with the minimum key value from the set of vertices not yet included in the MST. It takes three parameters: key, mst\_set, and vertices. The key list contains the current key values of all vertices, mst\_set is a boolean list indicating whether a vertex is included in the MST, and vertices represents the total number of vertices in the graph. The function iterates through each vertex, checks if its key value is less than the current minimum key (min\_k), and is not part of the MST. If it satisfies these conditions, the minimum key and its index (min\_index) are updated accordingly. Finally, the function returns the index of the vertex with the minimum key.

The prim function takes a matrix representing the weighted graph as input. It starts by initializing some data structures: key is a list that stores the key values for each vertex, initially set to a large value (sys.maxsize); parent is a list that stores the parent vertex for each vertex in the MST, initially set to None; mst\_set is a boolean list that tracks whether a vertex is included in the MST, initially set to False. The first vertex (0) is considered the starting vertex, so its key is set to 0, and its parent is also set to 0.

The function then iterates vertices times, selecting the vertex with the minimum key value using the min\_key function. The selected vertex u is marked as included in the MST by setting mst\_set[u] to True. Next, the function iterates over all the vertices and updates the key values and parents of the adjacent vertices if the edge weight (matrix[u][v]) is smaller than the current key value (key[v]). This step effectively updates the key values and determines the next vertex to be included in the MST.

Finally, the function constructs the MST by creating a list result that stores the edges of the MST. It iterates from vertex 1 to vertices and appends the parent vertex, current vertex, and the weight of the edge between them to the result list.



**Figure 2. Implementation Prim**

**Kruskal**

**Implementation:**

The function **find** is a helper function used to find the parent of a given node in the disjoint set data structure. It takes two parameters: **parent** and **i**, where **parent** is a list representing the parent of each node, and **i** is the index of the node being searched. The function recursively finds the root parent of the node **i** and assigns it as the direct parent of **i**. This process also optimizes the disjoint set structure by collapsing the path, which helps in subsequent find operations.

The function **union** is another helper function used to perform the union operation on two sets in the disjoint set data structure. It takes four parameters: **parent**, **rank**, **x**, and **y**. The function compares the ranks of the root parents (**x** and **y**) of the given nodes and performs the union accordingly. If the rank of **x** is less than the rank of **y**, **x** becomes the parent of **y**. If the rank of **x** is greater than the rank of **y**, **y** becomes the parent of **x**. If the ranks are equal, either choice can be made, and the rank of the resulting parent is increased by 1.

The **kruskal** function takes an input graph represented as an edge list. It starts by extracting the number of vertices from the last element of the graph and initializing an empty **result** list. Variables **i** and **e** are initialized to 0, representing the current edge being considered and the number of edges included in the MST so far, respectively.

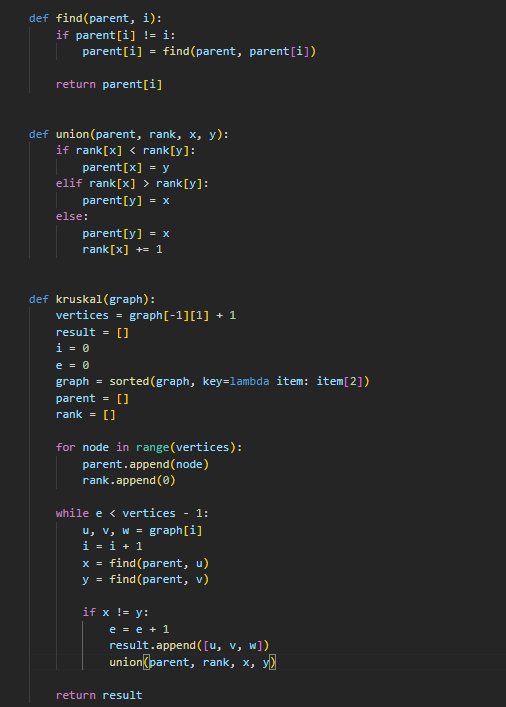
The graph is then sorted in non-decreasing order based on the edge weights using the **sorted** function and a lambda function as the key. This ensures that the algorithm processes the edges in ascending order of their weights.

Two lists, **parent** and **rank**, are initialized to keep track of the parent nodes and ranks of each node in the disjoint set data structure.

The algorithm then enters a while loop that continues until **e** (the number of edges in the MST) reaches **vertices - 1** (the number of vertices minus one, which is the maximum number of edges in a spanning tree).

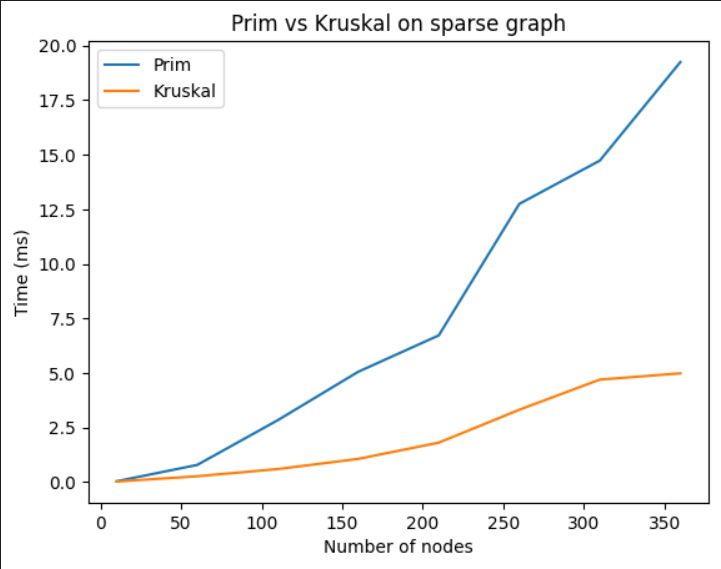
Within the loop, the algorithm extracts the next edge from the sorted graph and its corresponding vertices and weight. It then performs the **find** operation on the parent nodes of the two vertices to determine if they belong to the same disjoint set. If they have different parent roots (**x** and **y**), it means adding the edge (**u**, **v**, **w**) to the MST will not create a cycle. Thus, the edge is considered safe, and it is added to the **result** list. The **union** operation is then performed to merge the two sets by updating the parent and rank accordingly.

Finally, the **result** list, containing the edges of the minimum spanning tree, is returned.

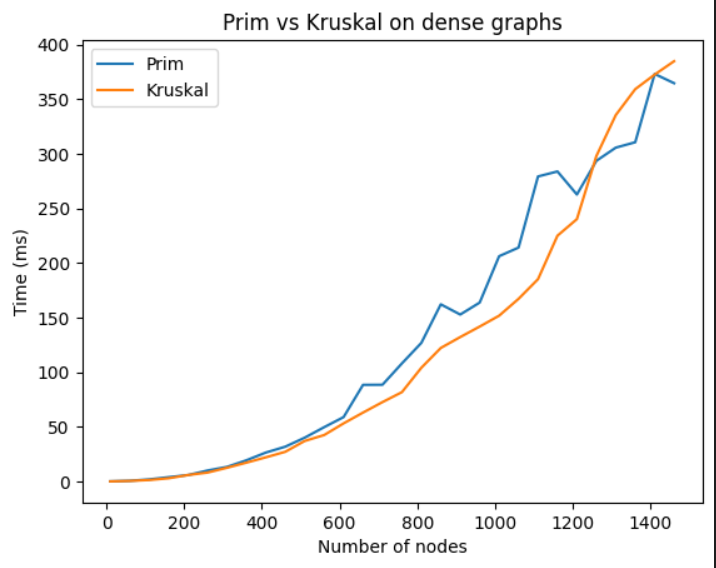


**Figure 3. Implementation Kruskal**

**Results for the sparse tree:**



**Figure 4. Plot with results for the sparse tree.**



**Figure 5. Plot with results for the dense tree.**

**Conclusion:**

In summary, Kruskal's algorithm and Prim's algorithm are effective approaches for finding the minimum spanning tree of a weighted graph. They differ in their strategies and implementation details, making them suitable for different graph characteristics and application scenarios. The choice between the two algorithms depends on the specific requirements of the problem and the characteristics of the graph being considered. Both algorithms have time complexities that depend on the number of edges and vertices in the graph.

Kruskal's algorithm has a time complexity of O(E log E), where E is the number of edges. Prim's algorithm has a time complexity of O(E log V), where V is the number of vertices. Generally, Kruskal's algorithm performs better in graphs with a large number of edges, while Prim's algorithm is more efficient for sparse graphs.

**Link to GitHub:** <https://github.com/SexomQ/AlgorithmsAnalysis-labs>