MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work 5:

Greedy Algorithms

Checked by:

Fiștic Cristof, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

TABLE OF CONTENTS:

ALGORITHM ANALYSIS

Objective

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

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 the graph and analyze how this influences the algorithms. Make a graphical presentation of the data obtained

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer. In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.

2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.

3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).

4. The algorithm is implemented in a programming language.

5. Generating multiple sets of input data.

6. Run the program for each input data set.

7. The obtained data are analyzed. The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach. This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result. However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure AD If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure. Advantages of Greedy Approach The algorithm is easier to describe. This algorithm can perform better than other algorithms (but, not in all cases).

Comparison Metric:

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

Input Format:

As input, each algorithm will receive different size graphs. We will generate different graphs (sparse and dense). Each algorithm will be called with the generated graph and the time execution will be saved in an array to later be plotted.

IMPLEMENTATION

Prim's algorithm:

In computer science, Prim'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.

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the Jarník's algorithm, Prim–Jarník algorithm, Prim–Dijkstra algorithm or the DJP algorithm.

Other well-known algorithms for this problem include Kruskal's algorithm and Borůvka's algorithm. These algorithms find the minimum spanning forest in a possibly disconnected graph; in contrast, the most basic form of Prim's algorithm only finds minimum spanning trees in connected graphs. However, running Prim's algorithm separately for each connected component of the graph, it can also be used to find the minimum spanning forest. In terms of their asymptotic time complexity, these three algorithms are equally fast for sparse graphs, but slower than other more sophisticated algorithms. However, for graphs that are sufficiently dense, Prim's algorithm can be made to run in linear time, meeting or improving the time bounds for other algorithms.

Implementation:

```
# Prim's Algorithm
def prim(graph):
    n = len(graph)
    mst = [None] * n  # MST to store the constructed MST
    key = [float('inf')] * n  # Key values used to pick minimum weight edge
in cut
    visited = [False] * n  # To represent set of vertices not yet included
in MST

    # Always include first 0th vertex in MST.
    key[0] = 0
    mst[0] = -1  # First node is always root of MST
```

```
    for _ in range(n-1):
        # Pick the minimum key vertex from the set of vertices not yet
included in MST
        u = min_key_vertex(key, visited)
        visited[u] = True

        # Update key and mst for adjacent vertices of the picked vertex
        for v in range(n):
            if 0 < graph[u][v] < key[v] and not visited[v]:
                mst[v] = u
                key[v] = graph[u][v]

    return mst

def min_key_vertex(key, visited):
    min_val = float('inf')
    min_index = -1

    for v in range(len(key)):
        if key[v] < min_val and not visited[v]:
            min_val = key[v]
            min_index = v

    return min_index
```

Figure 1 Prim's algorithm python implementation

Kruskal's 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. It is a greedy algorithm that in each step adds to the forest the lowest-weight edge that will not form a cycle. The key steps of the algorithm are sorting and the use of a disjoint-set data structure to detect cycles. Its running time is dominated by the time to sort all of the graph edges by their weight.

A minimum spanning tree of a connected weighted graph is a connected subgraph, without cycles, for which the sum of the weights of all the edges in the subgraph is minimal. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.

This algorithm was first published by Joseph Kruskal in 1956, and was rediscovered soon afterward by Loberman & Weinberger (1957). Other algorithms for this problem include Borůvka's algorithm, Jarník's algorithm, and the reverse-delete algorithm.

Implementation:

```
# Kruskal's Algorithm
class DisjointSet:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

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

    def union(self, u, v):
        u_root = self.find(u)
        v_root = self.find(v)

        if u_root == v_root:
            return False

        if self.rank[u_root] < self.rank[v_root]:
            self.parent[u_root] = v_root
        elif self.rank[u_root] > self.rank[v_root]:
            self.parent[v_root] = u_root
        else:
            self.parent[v_root] = u_root
            self.rank[u_root] += 1

        return True

def kruskal(graph):
    n = len(graph)
    result = []
    ds = DisjointSet(n)
```

```
    edges = []
    for i in range(n):
        for j in range(i+1, n):
            if graph[i][j] > 0:
                edges.append((i, j, graph[i][j]))

    edges.sort(key=lambda x: x[2])

    for edge in edges:
        u, v, weight = edge
        if ds.union(u, v):
            result.append(edge)

    return result
```

Figure 2 Kruskal's algorithm python implementation

Results:

Algorithmic Complexity:

Prim's Algorithm has a time complexity of O(V2), which can be improved to O(ElogV) (V is the number of vertices, and E is the number of edges.) Meanwhile, Kruskal's algorithm consistently has a time complexity of O(ElogV), leveraging sorting of edges and the disjoint-set (union-find) data structure for efficiency. Prim's algorithm may traverse nodes multiple times to find the minimum distance edge that connects a new vertex to the growing MST. In contrast, Kruskal's algorithm traverses each node only once, focusing on adding the smallest weighted edge available that does not form a cycle, regardless of its connection to a current tree.

Analysis of Results:

The graph compares the execution time of Prim's and Kruskal's algorithms for finding the minimum spanning tree (MST) on a graph with an increasing number of nodes. The y-axis represents the execution time in seconds and the x-axis represents the number of nodes in the graph.

According to the graph, Prim's algorithm appears to be faster than Kruskal's algorithm for finding the MST for all the data points shown in the range of 0 to 2000 nodes.

It's important to note that this is a single data point, and the execution time can vary depending on the specific implementation of the algorithms and the characteristics of the graph. ᵕ

Here are some additional factors that can affect the performance of Prim's and Kruskal's algorithms:

- Graph density: Generally, Prim's algorithm performs better for dense graphs, while Kruskal's algorithm may perform better for sparse graphs. The graph in the benchmark doesn't specify the density of the graphs used.
- Implementation: Different implementations of the algorithms may have different time complexities.

Overall, while the graph suggests that Prim's algorithm might be faster in this case, it's important to consider the factors mentioned above when choosing an algorithm for a specific application.

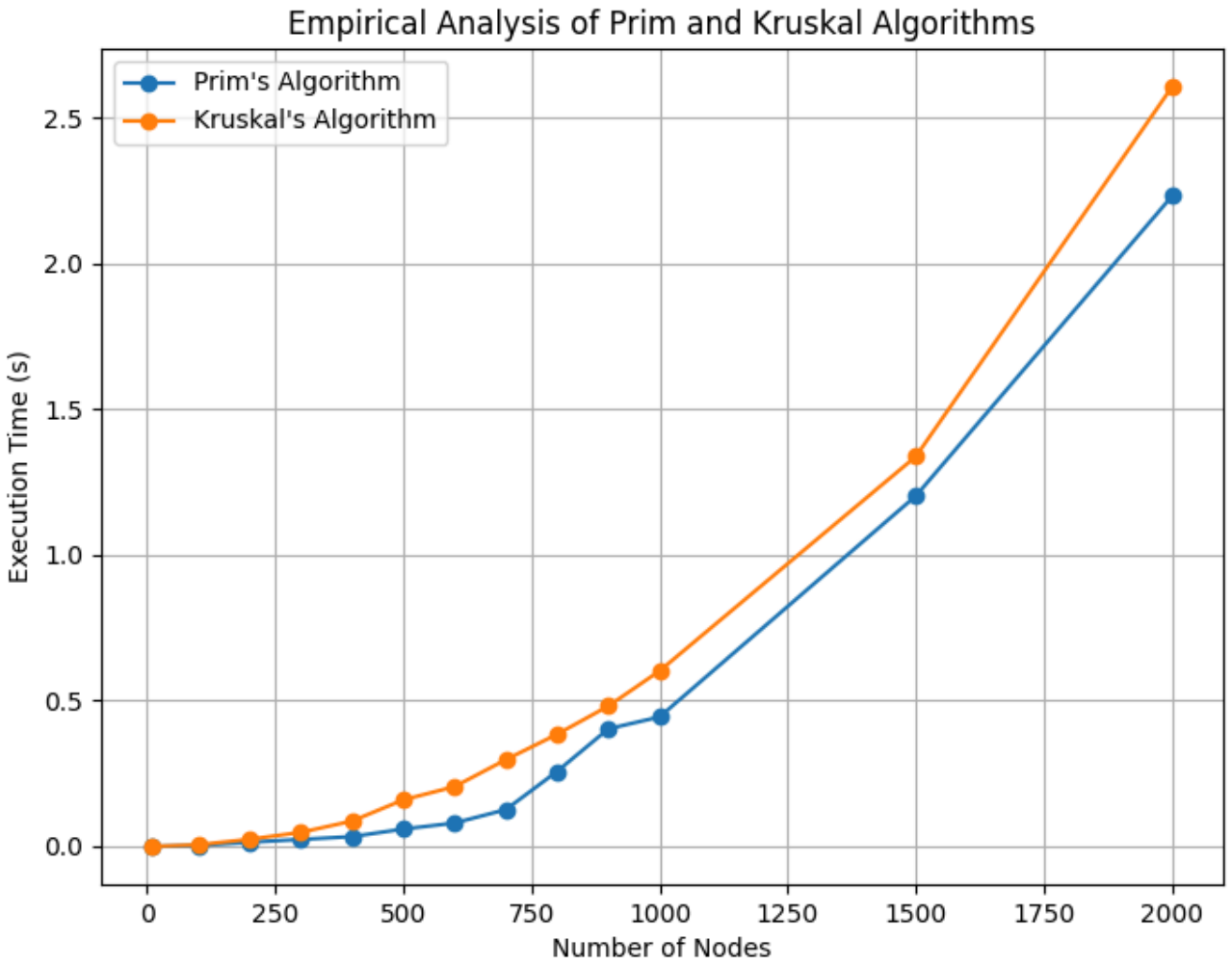

Figure 3 benchmark of Prim's and Kruskal's Algorithm

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.
It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.
Prim's algorithm prefer list data structures.	Kruskal's algorithm prefer heap data structures.

Figure 4 comparison table

CONCLUSION:

In conclusion, this laboratory work compared Prim's and Kruskal's algorithms for finding the minimum spanning tree (MST) of a graph. The results suggest that Prim's algorithm might be faster for the given test cases, particularly for denser graphs. However, it's important to acknowledge that Kruskal's algorithm offers guaranteed efficiency with a time complexity of O(E log V) and can handle disconnected graphs more effectively. When choosing an algorithm for real-world applications, the specific characteristics of the graph, such as its density and connectivity, along with the importance of implementation simplicity, should be considered.