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

Laboratory work 5:
Study and empirical analysis of greedy algorithms. Analysis of Prim’s and Kruskal’s algorithms.

Elaborated:
st. gr. FAF-222 Martiniuc Artiom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2024

TABLE OF CONTENTS

Objective

ALGORITHM ANALYSIS

Study and analyze different greedy algorithms.

Tasks:

1. To 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:

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:

In this laboratory exercise, we explore and analyze two different greedy algorithms: Prim and Kruskal. 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.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 2 naïve algorithms empirically.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)), The amount of memory used by the algorithm as the graph size increases.

Input Format:

The input for both algorithms graphs with different amount of nodes.

IMPLEMENTATION

Two algorithms will be implemented in their native form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending o memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Prim's Algorithm

Prim's algorithm starts with a single vertex and grows the spanning tree one edge at a time, adding the smallest edge that connects a vertex in the tree to a vertex outside the tree.

Typically uses a priority queue (or binary heap) to efficiently select the next edge with the minimum weight.

More efficient for dense graphs, where E is close to V^2, since the number of edges to be considered each step is larger.

Complexity: O(E + V log V)

Implementation:

```python
def prim(graph, start_vertex):
    visited = set([start_vertex])
    edges = [(cost, start_vertex, to) for to, cost in graph[start_vertex].items()]
    heapq.heapify(edges)
    mst_cost = 0
    mst_edges = []

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst_cost += cost
            mst_edges.append((frm, to, cost))

            for next_to, next_cost in graph[to].items():
                if next_to not in visited:
                    heapq.heappush( *args: edges, (next_cost, to, next_to))

    return mst_cost, mst_edges
```

Figure 1 Prim's in Python

Kruskal's Algorithm

Kruskal's algorithm treats every vertex as a separate tree and merges these trees, starting with the smallest edges, avoiding any edges that would create a cycle.

Utilizes a disjoint-set (or union-find) data structure to keep track of the sets of vertices that have been joined and to detect cycles efficiently.

Often more efficient for sparse graphs where E is much less than V^2, since the algorithm starts by considering all edges.

Complexity: O(E log E)

Implementation:

```
def kruskal(graph, vertices):
    result = []
    i, e = 0, 0
    graph = sorted(graph, key=lambda item: item[2])
    parent = [node for node in range(vertices)]
    rank = [0 for _ in range(vertices)]

    while e < vertices - 1 and i < len(graph):  # C
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
        y = find(parent, v)

        if x != y:
            e += 1
            result.append((u, v, w))
            union(parent, rank, x, y)

    return result
```

Figure 2 Kruskal's in Python

The `find` function is used to identify which subset a particular element is in. This can be used for determining if two elements are in the same subset.

The `union` function is used to merge two subsets into a single subset. It takes two elements, and if they are in different subsets, it merges the subsets.

```
def find(parent, i):
    if parent[i] == i:
        return i
    return find(parent, parent[i])


new *
def union(parent, rank, x, y):
    xroot = find(parent, x)
    yroot = find(parent, y)

    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1
```

Figure 3 Additional functions for Kruskal's in Python

RESULTS

The provided graphs display the execution time of Prim's and Kruskal's algorithms applied to sparse and dense graphs with an increasing number of nodes. The observations are as follows: As you can see, both algorithms exhibit a trend where the time increases with the number of nodes. However, the exact performance and efficiency depend on the structure of the graph and how nodes are connected.

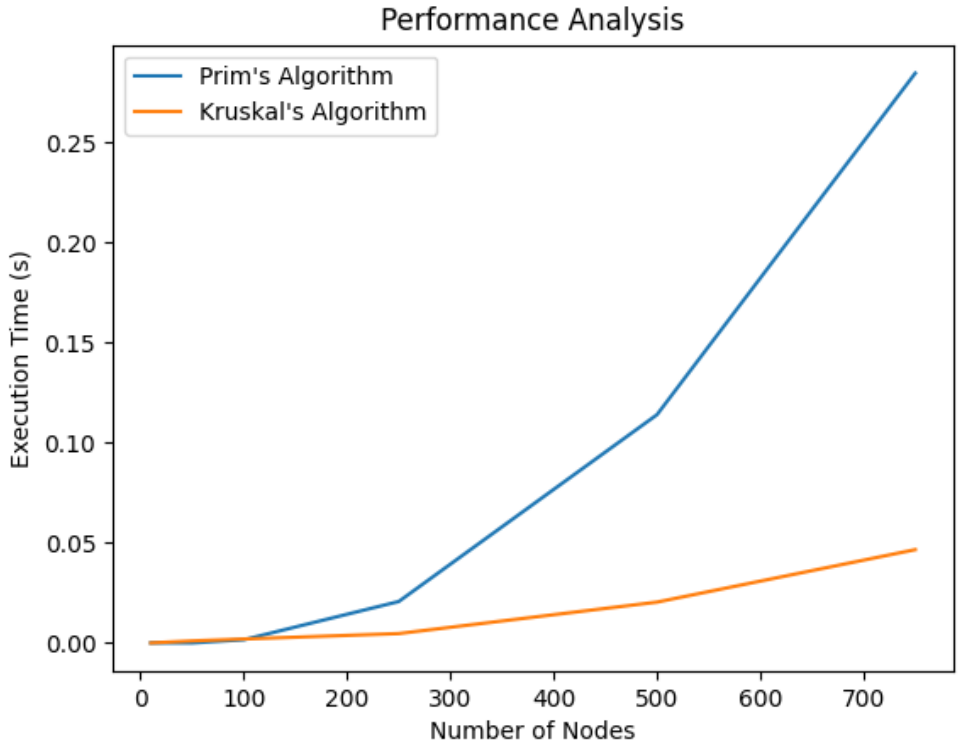

Figure 4 Graph of Algorithms Performance

The execution time for Kruskal's algorithm appears to increase at a slower rate than Prim's algorithm as the number of nodes increases. This suggests that for the graph sizes tested, Kruskal's algorithm is scaling better than Prim's algorithm.

Prim's algorithm shows a more pronounced increase in execution time as the number of nodes grows. This suggests that for the given implementation and the types of graphs generated, Prim's algorithm may be less efficient as the graph size increases, especially in the larger graph sizes.

There is no crossover point within the range tested; Kruskal's algorithm remains the faster one throughout this range. This indicates consistent performance advantages under the tested conditions.

CONCLUSION

In conclusion, while Kruskal's is outperforming Prim's in these tests, it would be essential to verify these results with multiple runs and consider whether the graph generation process or the chosen data structures could be optimized further. Additionally, testing on both sparse and dense graphs would provide a more complete picture of the performance characteristics of these algorithms.

On the one hand, Prim's algorithm should perform better on dense graphs due to its more favorable time complexity with the right data structures.

On the other hand, if the generated graphs tend to be sparse, Kruskal's algorithm could naturally have an advantage.