MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 7 : Greedy Algorithms

Elaborated:
st.gr. FAF-211 Gazea Sandu
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Greedy algorithms are problem-solving techniques that prioritize making the best immediate choices in the hopes of achieving the best overall solution. They are commonly used to solve optimization problems and are part of the broader field of algorithm design and analysis.

The underlying philosophy of greedy algorithms is to focus on the current step and select the option that appears to be the most beneficial without considering the long-term consequences. This approach may seem short-sighted, but it often leads to efficient and effective solutions, especially when applied in the right context. Greedy algorithms have a simple and intuitive nature, which makes them popular for solving a wide range of problems.

One of the main advantages of greedy algorithms is their computational efficiency. Unlike complex algorithms that require extensive computations and backtracking, greedy algorithms typically have a linear or near-linear runtime complexity. This makes them suitable for handling large input sizes where speed and scalability are important.

However, it's important to note that greedy algorithms do not always guarantee optimal solutions. Their focus on immediate gains may cause them to overlook certain possibilities or make choices that result in suboptimal outcomes. The correctness of a greedy algorithm depends heavily on the specific problem and its characteristics.

In this introduction, we will explore various aspects of greedy algorithms, including their principles, common applications, and limitations. We will examine the strategies used in designing greedy algorithms, such as the greedy choice property and the optimal substructure property. Additionally, we will discuss famous examples of greedy algorithms and techniques to evaluate and analyze their efficiency.

By understanding the fundamentals of greedy algorithms, you will gain a valuable set of tools to tackle optimization problems in various fields, including computer science, operations research, and economics. So let's embark on this journey to uncover the strengths and weaknesses of greedy algorithms and discover how they can help us find nearly optimal solutions in many real-world situations.

Prim’s algorithm

Prim’s algorithm is a well-known method used to find the minimum spanning tree (MST) of a weighted undirected graph. The minimum spanning tree is like a subset of the original graph that connects all the vertices while keeping the total weight of the edges as low as possible.The algorithm starts by picking an arbitrary vertex and progressively adds edges to create the MST. At each step, it chooses the edge with the smallest weight that connects a vertex in the MST to a vertex outside of it. This process continues until all the vertices are included in the MST.To begin, we start with an empty MST and a set of vertices that haven’t been added to the MST yet. We then choose any vertex as the starting point for the MST.Next, while there are vertices not yet included in the MST, we follow these steps:we select the edge with the minimum weight that connects a vertex in the MST to a vertex outside of it, we add the selected edge to the MST,we include the newly reached vertex in the set of vertices belonging to the MST.We repeat these steps until all the vertices are included in the MST, and at that point, our MST is complete.

The efficiency of Prim’s algorithm depends on the data structure used to store the edges and vertices. Commonly, a priority queue is used, which allows us to quickly retrieve the edge with the smallest weight during each iteration.

Prim’s algorithm guarantees that the resulting MST is always connected and has the minimum possible weight among all the spanning trees of the original graph. This optimality is achieved because, at each step, the algorithm selects the edge that seems best locally. The correctness of the algorithm is backed by the ”cut property,” which states that the edge with the minimum weight crossing any cut in the graph is always part of the MST.

The applications of Prim’s algorithm are widespread across various fields. For example, it can be used in network design to establish communication links between different locations while minimizing the overall cost. Prim’s algorithm is also applicable to problems like clustering analysis, image segmentation, and solving the traveling salesman problem.

However, it’s important to note that Prim’s algorithm assumes that the input graph is connected. If the graph has multiple disconnected components, the algorithm needs to be modified to handle them.

In conclusion, Prim’s algorithm is a powerful tool for finding the minimum spanning tree of a weighted undirected graph. Its simplicity, efficiency, and optimality make it a popular choice in many applications involving network optimization, cost minimization, and constructing spanning trees. Code:

```
Prim'sAlgorithm(Graph G):
    Select an arbitrary starting vertex s
    Create an empty set MST to store the minimum spanning tree
    Create a priority queue Q to keep track of candidate edges
```

```
Mark the starting vertex s as visited
Add all edges incident to s into the priority queue Q

while Q is not empty:
    Select the edge with the minimum weight from Q (u, v)
    Remove (u, v) from Q

    if v is not visited:
        Mark v as visited
        Add (u, v) to MST
        Add all edges incident to v into the priority queue Q

Return MST
```

Kruskal's Algorithm

Kruskal's algorithm is a technique used to find the best way to connect all the locations in a weighted undirected graph while minimizing the overall cost. The goal is to create a network that includes all the vertices while keeping the total weight of the edges as low as possible.

Instead of starting from a specific point, Kruskal's algorithm begins with an empty network and gradually adds edges to it. The algorithm works by sorting all the edges based on their weights, from smallest to largest.

Next, it goes through each edge in this sorted list. If adding the edge to the network doesn't create a loop, meaning it doesn't connect vertices that are already connected, it is included in the network. Otherwise, if adding the edge would create a loop, it is discarded.

This process continues until either all the vertices are connected or all the edges have been considered.

Kruskal's algorithm ensures that the resulting network is always connected and has the minimum possible total weight among all the possible networks. It achieves this optimality by progressively adding edges with the smallest weights, as long as they don't create loops or cycles in the network.

The applications of Kruskal's algorithm are widespread across various fields. It can be used in designing communication networks, where the aim is to establish links between different locations while minimizing the overall cost. Kruskal's algorithm also finds applications in clustering analysis, image segmentation, and optimizing transportation networks.

In conclusion, Kruskal's algorithm is a powerful tool for finding the best way to connect locations in a weighted undirected graph. By considering the weights of the edges and avoiding loops, it efficiently constructs a network that achieves the minimum possible total weight.

Code:

```
Kruskal'sAlgorithm(Graph G):
    Create an empty MST to store the minimum spanning tree
    Create an empty set of disjoint sets (one for each vertex)
    Sort all the edges in G in ascending order based on their weights

    for each vertex v in G:
        Create a new disjoint set containing v

    for each edge (u, v) in the sorted list of edges:
        if the vertices u and v belong to different disjoint sets:
            Add the edge (u, v) to the MST
            Merge the disjoint sets containing u and v

    Return MST
```

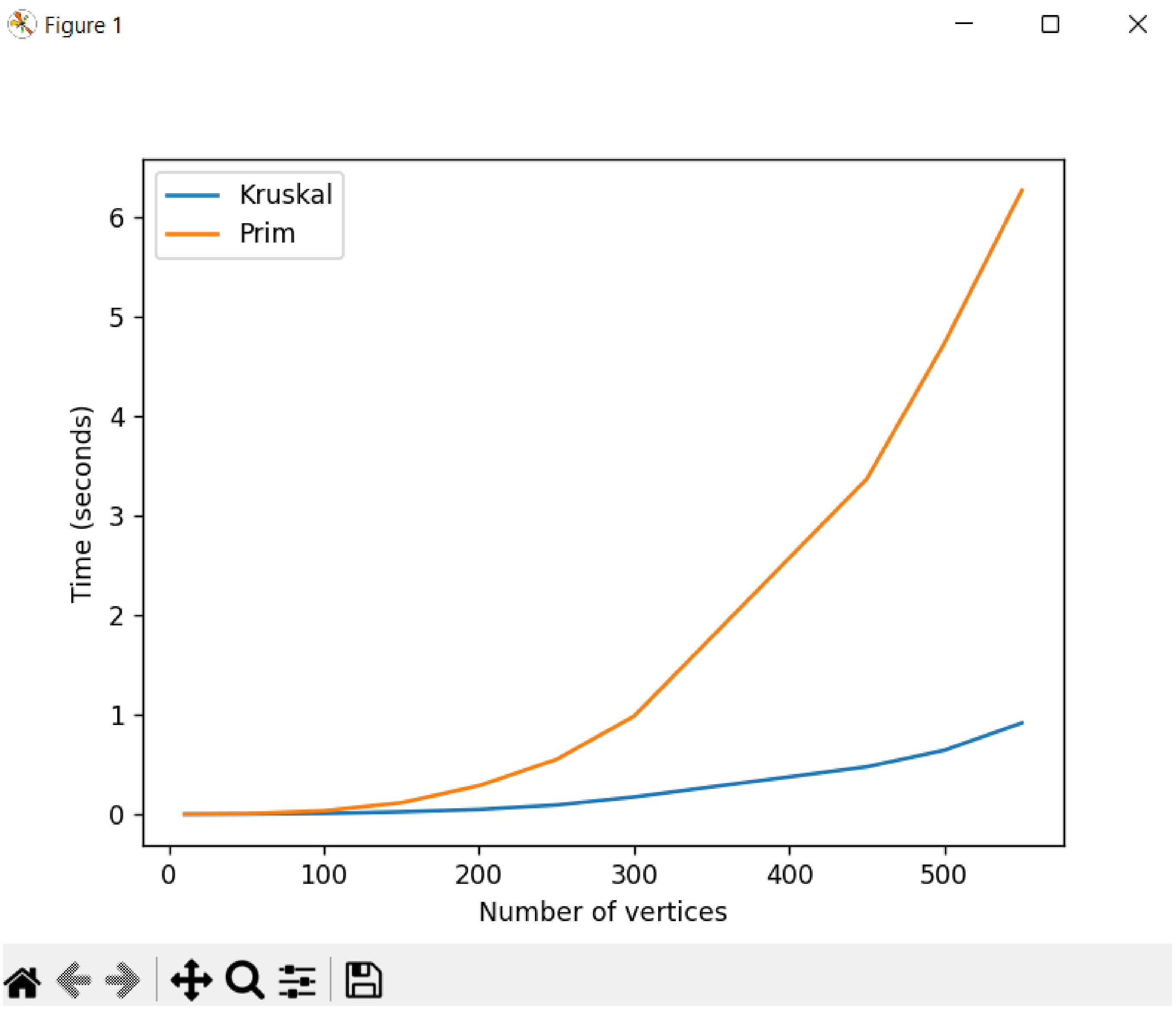
Implementation

```
import matplotlib.pyplot as plt

import random

import time


# Node class, represents a single node in the network.

class Node:

     # Each node has an id and a dictionary to store connected nodes (edges) and their wei

    def __init__(self, id):

        self.id = id

        self.edges = {}

# Method to add a connection between this node and another

    def add_edge(self, node, weight):

        self.edges[node] = weight


    def get_edges(self):

        return self.edges



class Network:

    def __init__(self, size):

        self.nodes = [Node(i) for i in range(size)]


    def add_edge(self, u, v, weight):

        self.nodes[u].add_edge(v, weight)

        self.nodes[v].add_edge(u, weight)


    def get_size(self):

        return len(self.nodes)


# Kruskal's algorithm for finding the minimum spanning tree of a network.
```

```
def kruskal(network):
    # A set to store the minimum spanning tree.
    mst = set()
    edges = [(u, v, w) for u in range(network.get_size()) for v, w in network.nodes[u].get
    edges.sort(key=lambda x: x[2])
    # Initialize disjoint sets for each node.
    disjoint_sets = [{i} for i in range(network.get_size())]
    # Initialize disjoint sets for each node.
    for u, v, w in edges:
        for s in disjoint_sets:
            if u in s:
                set_u = s
            if v in s:
                set_v = s
        if set_u != set_v:
            mst.add((u, v))
            disjoint_sets.remove(set_u)
            disjoint_sets.remove(set_v)
            disjoint_sets.append(set_u.union(set_v))
    return mst

# Prim's algorithm for finding the minimum spanning tree of a network.
def prim(network):
    # A set to store the minimum spanning tree.
    mst = set()
    # Set of visited nodes, initially containing just the first node.
    visited = {0}
    # While there are unvisited nodes.
    while len(visited) < network.get_size():
        # Get all edges that connect a visited node with an unvisited node.
        edges = [(u, v, network.nodes[u].get_edges()[v]) for u in visited for v in network
        # Choose the edge with the smallest weight.
        u, v, w = min(edges, key=lambda x: x[2])
        # Add the chosen edge to the MST and the new node to the set of visited nodes.
        mst.add((u, v))
        visited.add(v)
    return mst


# Function to generate networks and benchmark the Kruskal and Prim algorithms.
def benchmark(num_vertices_list):
    # Generate a list of networks of different sizes.
    network_list = [Network(num_vertices) for num_vertices in num_vertices_list]
    for network in network_list:
        # Add random edges between the nodes of the network.
        for u in range(network.get_size()):
            for v in range(u + 1, network.get_size()):
                network.add_edge(u, v, random.randint(1, 100))

    kruskal_times = benchmark_algorithm(kruskal, network_list)
    prim_times = benchmark_algorithm(prim, network_list)

    plt.plot(num_vertices_list, kruskal_times, label='Kruskal')
    plt.plot(num_vertices_list, prim_times, label='Prim')
    plt.xlabel('Number of vertices')
    plt.ylabel('Time (seconds)')
    plt.legend()
    plt.show()


def benchmark_algorithm(algorithm, network_list):
    times = []
    for network in network_list:
        start_time = time.time()
        algorithm(network)
        end_time = time.time()
        times.append(end_time - start_time)
    return times
```

```
num_vertices_list = [10, 50, 100, 150, 200, 250, 300, 450, 500, 550,]
benchmark(num_vertices_list)
```

Screenshot:



Conclusion

Both Prim's algorithm and Kruskal's algorithm are commonly used methods to find the minimum spanning tree (MST) of a graph. The MST is a subgraph that connects all the vertices while minimizing the total weight of the edges.

Prim's algorithm takes a greedy approach. It starts with an arbitrary vertex and adds edges to the MST one by one. Each edge chosen has the smallest weight and connects a vertex in the MST to a vertex outside of it. This process continues until all vertices are included in the MST. Prim's algorithm guarantees a connected MST with the minimum weight among all possible spanning trees. It works well for dense graphs and can be implemented using a priority queue or heap data structure.

Kruskal's algorithm also follows a greedy strategy but with a different approach. It sorts all the edges in non-decreasing order based on their weights and considers each edge systematically. The algorithm adds an edge to the MST if it doesn't create a cycle, using the disjoint-set data structure to efficiently check connectivity and merge sets. Kruskal's algorithm also guarantees a connected MST with the minimum weight. It performs well for sparse graphs and has a runtime complexity of O(E log V), where E is the number of edges and V is the number of vertices.

When choosing between Prim's algorithm and Kruskal's algorithm, several factors come into play. Prim's algorithm works efficiently for dense graphs and situations where we start with a specific vertex. It often performs faster in practice for such cases. On the other hand, Kruskal's algorithm can handle disconnected graphs and works well for sparse graphs. It is often simpler to implement and has a more intuitive edge-by-edge selection process.

The choice between the two algorithms depends on the specific problem and the characteristics of the graph. Factors such as graph density, connectivity requirements, and edge weights influence the selection. Additionally, the efficient implementation of data structures like priority queues and disjoint-sets can impact overall performance.

In conclusion, both Prim's algorithm and Kruskal's algorithm are powerful techniques for finding the minimum spanning tree in a graph. They offer different strategies and considerations, allowing flexibility in selecting the most suitable approach based on the problem's requirements and the characteristics of the input graph.

$https://github.com/GSandu1/LabsAA.git$