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 Grama Alexandru
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Greedy algorithms are a class of problem-solving techniques that prioritize making locally optimal choices at each step with the hope of finding a globally optimal solution. These algorithms belong to the broader field of algorithm design and analysis, and they are particularly useful in solving optimization problems. The underlying philosophy of greedy algorithms is to make the best choice available at each decision point, without considering the overall consequences of that choice.

The key characteristic of greedy algorithms is their myopic approach, focusing on immediate gains rather than considering the long-term ramifications. Although this may seem shortsighted, greedy algorithms can often yield efficient and effective solutions, especially when used in the right context. The simplicity and intuitive nature of greedy algorithms make them an appealing choice for solving a wide range of problems.

One of the primary advantages of greedy algorithms is their computational efficiency. Unlike other complex algorithms that may require extensive computations and backtracking, greedy algorithms often have a linear or near-linear runtime complexity. This efficiency makes them well-suited for solving problems with large input sizes, where speed and scalability are crucial.

However, it's important to note that greedy algorithms do not guarantee optimal solutions in all scenarios. Due to their greedy nature, they may overlook certain possibilities or make choices that lead to suboptimal outcomes. Consequently, the correctness of a greedy algorithm relies heavily on the specific problem and the characteristics of the problem instance.

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

By understanding the fundamentals of greedy algorithms, you will gain a valuable toolset to approach optimization problems in diverse fields such as computer science, operations research, and economics. So let's embark on this journey to uncover the power and limitations of greedy algorithms and discover how they can help us find near-optimal solutions in many real-world scenarios.

Prim's algorithm

Prim's algorithm is a well-known greedy algorithm used for finding the minimum spanning tree (MST) of a weighted undirected graph. The minimum spanning tree is a subgraph that connects all the vertices of the original graph while minimizing the total weight or cost of the edges.

The algorithm starts with an arbitrary vertex and progressively adds edges to the MST. At each step, the algorithm selects the edge with the minimum weight that connects a vertex in the MST to a vertex outside of it. This process continues until all vertices are included in the MST. Here is a step-by-step description of Prim's algorithm:

1. Initialize an empty MST and a set of vertices that are not yet included in the MST.
2. Choose an arbitrary vertex to start the MST.
3. hile there are vertices not yet included in the MST:
   - Select the edge with the minimum weight that connects a vertex in the MST to a vertex outside of it.
   - Add the selected edge to the MST.
   - Add the newly reached vertex to the set of vertices in the MST.
4. Once all vertices are included, the MST is complete.

The efficiency of Prim's algorithm depends on the data structure used to store the edges and vertices. A common approach is to use a priority queue, which allows efficient retrieval of the edge with the minimum weight in each iteration.

Prim's algorithm guarantees that the resulting MST is always connected and has the minimum possible weight among all spanning trees of the original graph. This optimality is a consequence of the greedy choice property, as at each step, the algorithm selects the locally optimal edge. The proof of correctness relies on the cut property, which states that for any cut in the graph, the minimum-weight edge crossing the cut is always part of the MST.

The applications of Prim's algorithm are diverse and span various fields. For example, it can be used in network design, where the goal is to establish communication links between different locations while minimizing the overall cost. Prim's algorithm can also be applied to problems such as clustering analysis, image segmentation, and solving the traveling salesman problem.

However, it's important to note that Prim's algorithm assumes the input graph is connected. If the graph is not connected, the algorithm needs to be modified to handle multiple components.

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 that involve network optimization, cost minimization, and spanning tree construction.

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print
    # the constructed MST stored in parent[]
    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph
    # represented using adjacency matrix representation
    def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
```

```
parent = [None] * self.V  # Array to store constructed MST
# Make key 0 so that this vertex is picked as first vertex
key[0] = 0
mstSet = [False] * self.V

parent[0] = -1  # First node is always the root of

for cout in range(self.V):

    # Pick the minimum distance vertex from
    # the set of vertices not yet processed.
    # u is always equal to src in first iteration
    u = self.minKey(key, mstSet)

    # Put the minimum distance vertex in
    # the shortest path tree
    mstSet[u] = True

    # Update dist value of the adjacent vertices
    # of the picked vertex only if the current
    # distance is greater than new distance and
    # the vertex in not in the shortest path tree
    for v in range(self.V):

        # graph[u][v] is non zero only for adjacent vertices of m
        # mstSet[v] is false for vertices not yet included in MST
        # Update the key only if graph[u][v] is smaller than key[v]
        if self.graph[u][v] > 0 and mstSet[v] == False \
        and key[v] > self.graph[u][v]:
            key[v] = self.graph[u][v]
            parent[v] = u

self.printMST(parent)
```

```
if __name__ == '__main__':
    g = Graph(5)
    g.graph = [[0, 2, 0, 6, 0],
               [2, 0, 3, 8, 5],
               [0, 3, 0, 0, 7],
               [6, 8, 0, 0, 9],
               [0, 5, 7, 9, 0]]

    g.primMST()
```

Kruskal's Algorithm

Kruskal's algorithm is a widely-used greedy algorithm for finding the minimum spanning tree (MST) of a weighted undirected graph. Like other MST algorithms, Kruskal's algorithm aims to construct a subgraph that connects all the vertices while minimizing the total weight or cost of the edges.

The algorithm starts by sorting all the edges of the graph in non-decreasing order of their weights. It then systematically considers each edge in the sorted order, adding it to the MST if it does not create a cycle. This process continues until all vertices are included in the MST or until all edges have been examined.

Here is a step-by-step description of Kruskal's algorithm:

1. Sort all the edges of the graph in non-decreasing order of their weights.
2. Initialize an empty MST.
3. For each edge (u, v) in the sorted order:
    - If adding the edge (u, v) to the MST does not create a cycle, add it to the MST.
    - Otherwise, discard the edge and move to the next edge.
4. Once all vertices are included or all edges have been examined, the MST is complete.

To check if adding an edge creates a cycle, Kruskal's algorithm employs a disjoint-set data structure (also known as a union-find data structure). This data structure keeps track of disjoint sets of vertices and allows efficient checking of connectivity and merging of sets.

Kruskal's algorithm guarantees that the resulting MST is always connected and has the minimum possible weight among all spanning trees of the original graph. The optimality of Kruskal's algorithm stems from the fact that at each step, it selects the next lightest edge that does not create a cycle, adhering to the greedy choice property. The correctness of the algorithm can be proved using the cut property, similar to other MST algorithms.

Kruskal's algorithm is often favored for its simplicity and ease of implementation. It does not require the graph to be connected initially, making it suitable for disconnected graphs as well. Furthermore, Kruskal's algorithm can handle graphs with both weighted and unweighted edges, although it is most commonly used for weighted graphs.

Applications of Kruskal's algorithm can be found in various domains, such as network design, transportation planning, circuit design, and resource allocation. It is particularly useful in scenarios where the emphasis is on constructing a cost-efficient spanning tree that connects all the vertices.

In conclusion, Kruskal's algorithm provides an efficient and effective approach to finding the minimum spanning tree of a weighted undirected graph. By leveraging the greedy choice property and the disjoint-set data structure, Kruskal's algorithm ensures that the resulting MST is optimal and satisfies the connectivity requirements.

```
class Graph:


    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    # Function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i
    # (truly uses path compression technique)
    def find(self, parent, i):
        if parent[i] != i:

            # Reassignment of node's parent
            # to root node as
            # path compression requires
            parent[i] = self.find(parent, parent[i])
        return parent[i]

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[y] = x
            rank[x] += 1

    # The main function to construct MST
    # using Kruskal's algorithm
    def KruskalMST(self):

        # This will store the resultant MST
        result = []

        # An index variable, used for sorted edges
        i = 0

        # An index variable, used for result[]
        e = 0

        # Sort all the edges in
        # non-decreasing order of their
        # weight
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])
```

```
parent = []
rank = []

# Create V subsets with single elements
for node in range(self.V):
    parent.append(node)
    rank.append(0)

# Number of edges to be taken is less than to V-1
while e < self.V - 1:

    # Pick the smallest edge and increment
    # the index for next iteration
    u, v, w = self.graph[i]
    i = i + 1
    x = self.find(parent, u)
    y = self.find(parent, v)

    # If including this edge doesn't
    # cause cycle, then include it in result
    # and increment the index of result
    # for next edge
    if x != y:
        e = e + 1
        result.append([u, v, w])
        self.union(parent, rank, x, y)
    # Else discard the edge

minimumCost = 0
print("Edges in the constructed MST")
for u, v, weight in result:
    minimumCost += weight
    print("%d -- %d == %d" % (u, v, weight))
```

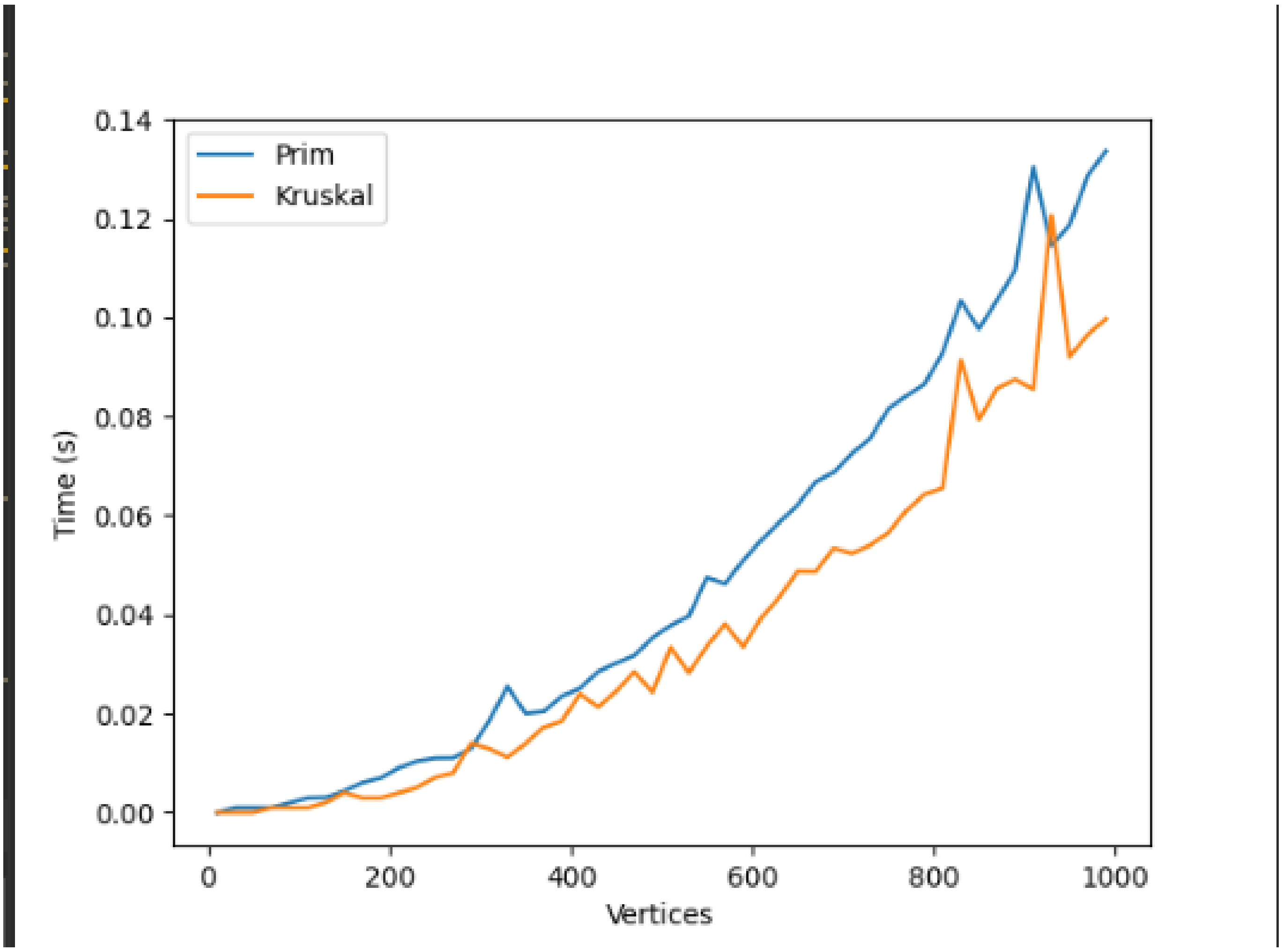
```
        print("Minimum Spanning Tree", minimumCost)


# Driver code
if __name__ == ’__main__’:
    g = Graph(4)
    g.addEdge(0, 1, 10)
    g.addEdge(0, 2, 6)
    g.addEdge(0, 3, 5)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)

    # Function call
    g.KruskalMST()
```

Implementation

Code in python:

```
import sys

import time

import matplotlib.pyplot as plt

import random


class Graph:

    def __init__(self, vertices):

        self.V = vertices

        self.graph = [[0 for column in range(vertices)]

                      for row in range(vertices)]


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

        self.graph[u][v] = w

        self.graph[v][u] = w




class Prim(Graph):

    def __init__(self, vertices):

        super().__init__(vertices)


    def primMST(self):

        key = [sys.maxsize] * self.V

        parent = [None] * self.V

        key[0] = 0

        mstSet = [False] * self.V


        parent[0] = -1


        for cout in range(self.V):

            u = self.minKey(key, mstSet)
            mstSet[u] = True

            for v in range(self.V):
                if (self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u]
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)

    def minKey(self, key, mstSet):
        min = sys.maxsize
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])


class Kruskal:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

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

    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.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

    def kruskalMST(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

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

        self.printMST(result)

    def printMST(self, result):
        print("Edges in the constructed MST")
        for u, v, weight in result:
            print("%d -- %d == %d" % (u, v, weight))



def analyze_algorithms():
    vertices_input = input("Enter the number of vertices: ")
    try:
        v = int(vertices_input)
    except ValueError:
        print("Invalid input! Please enter an integer.")
        return

    vertices = range(10, v, 20)
    times_prim = []
    times_kruskal = []

    for v in vertices:
        graph_prim = Prim(v)
        graph_kruskal = Kruskal(v)
        for i in range(v):
            for j in range(i+1, v):
                weight = random.randint(1, 100)
                graph_prim.add_edge(i, j, weight)
                graph_kruskal.add_edge(i, j, weight)

        start_time = time.time()
        graph_prim.primMST()
        times_prim.append(time.time() - start_time)

        start_time = time.time()
        graph_kruskal.kruskalMST()
        times_kruskal.append(time.time() - start_time)

    plt.plot(vertices, times_prim, label='Prim')
    plt.plot(vertices, times_kruskal, label='Kruskal')
    plt.xlabel('Vertices')
    plt.ylabel('Time (s)')
    plt.legend()
    plt.show()


if __name__ == '__main__':
    analyze_algorithms()
```

Screenshot:



0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Time (s)
Prim
Kruskal
0
200
400
600
800
1000
Vertices

Conclusion

Prim's algorithm and Kruskal's algorithm are two widely-used approaches for finding the minimum spanning tree (MST) of a graph. Both algorithms aim to construct a subgraph that connects all the vertices while minimizing the total weight or cost of the edges. However, they differ in their strategies and implementation.

Prim's algorithm follows a greedy approach by starting with an arbitrary vertex and progressively adding edges that have the minimum weight and connect 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 that the resulting MST is connected and has the minimum weight among all spanning trees of the graph. It is efficient for dense graphs and can be implemented using a priority queue or heap data structure.

Kruskal's algorithm, on the other hand, also employs a greedy strategy but takes a different approach. It sorts all the edges in non-decreasing order of their weights and systematically considers each edge. The algorithm adds the edge to the MST if it does not create a cycle, using the disjoint-set data structure to efficiently check for connectivity and merge sets. Kruskal's algorithm guarantees a connected and minimum-weight MST as well. 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.

Both Prim's algorithm and Kruskal's algorithm have their advantages and considerations. Prim's algorithm works well for dense graphs and situations where we start with a particular vertex. It is often faster in practice for such cases. Kruskal's algorithm, on the other hand, can handle disconnected graphs and performs efficiently for sparse graphs. It is often simpler to implement and has a more intuitive edge-by-edge selection process.

The choice between Prim's algorithm and Kruskal's algorithm depends on the specific characteristics of the problem and the graph being analyzed. Factors such as graph density, connectivity requirements, and edge weights play a role in determining the most suitable algorithm. Additionally, the efficient implementation of data structures like priority queues and disjoint-sets can impact the overall performance.

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