TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

DEPARTMENT: **SOFTWARE ENGINEERING AND AUTOMATION**

CC - LABORATORY WORK NR 6

# Greedy Algorithms

Ciuş Iurie    FAF-203

Chişinău 2022

# Contents

# 1 Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

## 1.1 Introduction

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time. For example, a greedy strategy for the travelling salesman problem (which is of high computational complexity) is the following heuristic: "At each step of the journey, visit the nearest unvisited city." This heuristic does not intend to find the best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically requires unreasonably many steps. In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids and give constant-factor approximations to optimization problems with the submodular structure.

Greedy algorithms produce good solutions on some mathematical problems, but not on others. Most problems for which they work will have two properties:

**Greedy choice property**

We can make whatever choice seems best at the moment and then solve the subproblems that arise later. The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage and may reconsider the previous stage's algorithmic path to the solution.

**Optimal substructure**

"A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems."

## 1.2 Objectives

- To study the greedy technique of designing algorithms.
- To be implemented in a programming language algorithms Prim and Kruskal.
- To make the empirical analysis of the Kruskal and Prim algorithms.
- To make a report.

## 1.3 Control questions:

- Describe the greedy method.
- When is the Kruskal algorithm applied and when is the Prim algorithm?
- How algorithm performance can be improved Right?
- What type of data is convenient to use when developing the program of a greedy algorithm?
- What are the advantages and disadvantages of Prim and Algorithms Kruskal?

## 1.4 Theoretical Notes

The Greedy Algorithm solves problems by making choices that seem best fitting during a particular moment. The use of this algorithm often appears throughout many optimization problems. Greedy algorithms provide efficient solutions that is close to optimal under two properties: one of them being the “Greedy Choice Property” which makes locally optimal decisions based on its current situation and the second being: “Optimal Structure” which contains solutions to sub problems of an optimal solution. However, Greedy algorithms at times fail to come to a global optimal solution because they don’t exhaustively work on all of the data at once. This is an issue because making certain decisions early on in the algorithm can prevent it from finding the best overall solution later on. This being said, if a Greedy algorithm can be proven to generate the global optimum of a given problem, it becomes the method of choice because it is faster than any other optimization methods such as Dynamic Programming etc.

Greedy algorithms are simple instinctive algorithms used for optimization (either maximized or minimized) problems. This algorithm makes the best choice at every step and attempts to find the optimal way to solve the whole problem.

**Kruskal’s Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph.

**Prim’s Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim’s algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

An improved **Dijkstra shortest path algorithm** might look like this. The improved algorithm introduces a constraint function with weighted value to solve the defects of the data structure storage, such as lots of redundancy of space and time. The number of search nodes is reduced by ignoring reversed nodes and the weighted value is flexibly changed to adapt to different network complexity.

## 1.5 Description of The Algorithms

The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

# 2 Code

## 2.1 Implementation

### Prim Algorithm

```java
import java.util.Scanner;

public class PrimMST extends Graph{
  int cost;
  int[] key;
  int[] included;

  public PrimMST(int v) {
    super(v);
    cost = 0;
    key = new int[v];
    for(int i = 0; i<v; i++)  key[i] = Integer.MAX_VALUE;
    included = new int[v];
  }

  int extractMin(){
    int min=Integer.MAX_VALUE;
    int index=-1;
    for(int i=0;i<v;i++){
      if(included[i]==0 && key[i]<min){
        min = key[i];
        index=i;
      }
    }
    return index;
  }

  void Prim(int[][] weight, int start){
    key[start]=0;
    vertex[start].parent=null;
    int u = extractMin();
    while(u!=-1){
      cost+=key[u];
      for(Vertex ver:vertex[u].adj){
        if(included[ver.data]==0 && weight[u][ver.data]<key[ver.
  data]){
          ver.parent=vertex[u];
          key[ver.data]=weight[u][ver.data];
        }
      }
      included[u]=1;
      u=extractMin();
    }
    System.out.println("Cost: "+cost);
  }

  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int v = sc.nextInt();
    int e = sc.nextInt();
    int[][] weight = new int[v][v];
    PrimMST g = new PrimMST(v);
    for(int i = 0; i<e; i++){
      int a = sc.nextInt(), b = sc.nextInt(), w = sc.nextInt();
      weight[a][b]=w;
      weight[b][a]=w;
      g.addEdge(a, b);
    }
    int start = 0;
    g.Prim(weight,start);
    sc.close();
  }
}
```

script.java

## Kruskal Algorithm

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

class Checker implements Comparator<int[]>{
  public int compare(int[] o1, int[] o2) {
    return Integer.compare(o1[2], o2[2]);
  }
}

public class KruskalMST extends Graph{
  int cost;
  boolean cycle;

  public KruskalMST(int v) {
    super(v);
    cost = 0;
    cycle = false;
  }

  void Kruskal(int[][] edges){
    for(int i = 0; i<edges.length; i++){
      int a = edges[i][0], b = edges[i][1], w = edges[i][2];
      if(a==b)  continue;//loop
      addEdge(a, b);
```

```java
      if(!isCycle()){
        System.out.println(a+" "+b+" "+w+" added");
        cost+=w;
      }else{
        vertex[a].adj.removeLast();
        vertex[b].adj.removeLast();
      }
    }
    System.out.println("Cost: "+cost);
  }

  void reset(){
    for(int i = 0; i<v; i++)
      vertex[i].color = 0;
    cycle = false;
  }

  boolean isCycle(){
    reset();
    for(int i=0;i<v && !cycle;i++){
        if(vertex[i].color==0)
          DFSVisit(i);
      }
    return cycle;
  }

  void DFSVisit(int u){
    vertex[u].color=1;
    for(Vertex v: vertex[u].adj){
      if(cycle) break;
      if(v.color==2){//cycle
        cycle = true;
        return;
      }
      if(v.color==0){
        DFSVisit(v.data);
      }
    }
    vertex[u].color=2;
  }

  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int v = sc.nextInt();
    int e = sc.nextInt();
    int[][] edges = new int[e][3];
    //edge from a[0] to a[1] with weight a[2]
    for(int i = 0; i<e; i++){
      int a = sc.nextInt(), b = sc.nextInt(), w = sc.nextInt();
```

```
            edges[i][0] = a;
            edges[i][1] = b;
            edges[i][2] = w;
        }
        Arrays.sort(edges,new Checker());
        KruskalMST g = new KruskalMST(v);
        g.Kruskal(edges);
        sc.close();
    }
}
```

kruskal.java

## 2.2 Complexity of the Algorithms

### Prim's Algorithm

The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight, which can be done using a priority queue. The following table shows the typical choices:

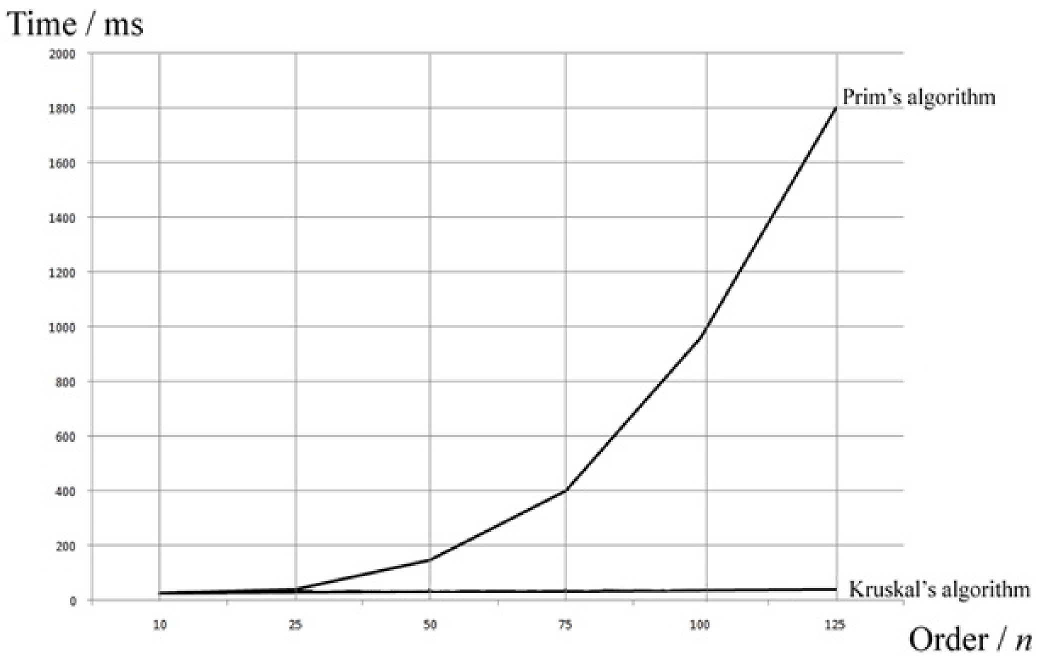
| Minimum edge weight data structure | Time complexity (total) |
|---|---|
| adjacency matrix, searching | $O(\|V\|^2)$ |
| binary heap and adjacency list | $O((\|V\| + \|E\|)log\|V\|) = O(\|E\|log\|V\|)$ |
| Fibonacci heap and adjacency list | $O(\|E\| + \|V\|log\|V\|)$ |

### Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of **O(V+E)** where V is the number of vertices, E is the number of edges.

### 2.3 Graphs



## 3 Conclusion

To conclude, the current piece of work represents my personal implementation of 2 greedy algorithms - Prim's and Kruskal. The programming language I chose is Java. The results show the time performance in seconds based on the N nodes. As you can see in the *Figure 1*, the Kruskal algorithms performed the best with a complexity of O(0), while the last one - $O(n^2)$.

### 3.1 References

- https://github.com/IuraCPersonal