**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.3

*la Analiza și Proiectarea Algoritmilor*

A efectuat:

st. gr. SI-221 Ceclea Victor

A verificat:

asist. univ. Andrievschi-Bagrin Veronica

Chişinău - 2023

**Lucrare de laborator nr 3**

**Tema:** Algoritmi greedy.

**Scopul lucrării:**

1. Studierea tehnicii greedy.

2. Analiza şi implementarea algoritmilor greedy.

**Rezumat succint la tema lucrării de laborator:**

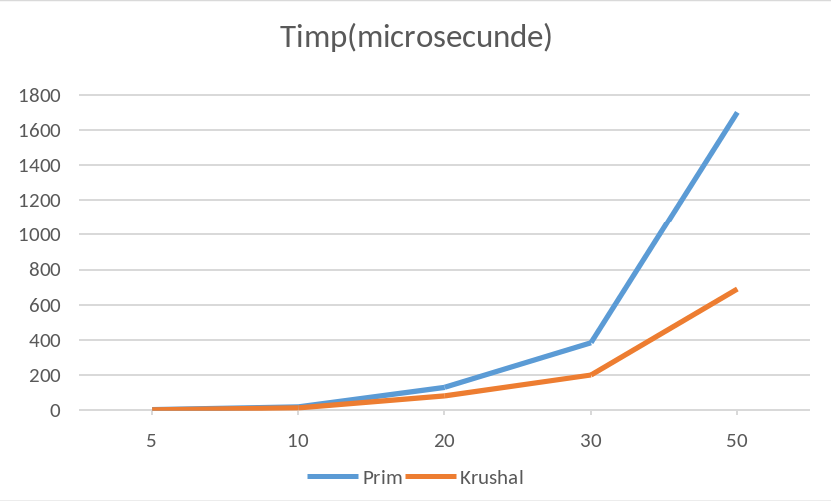
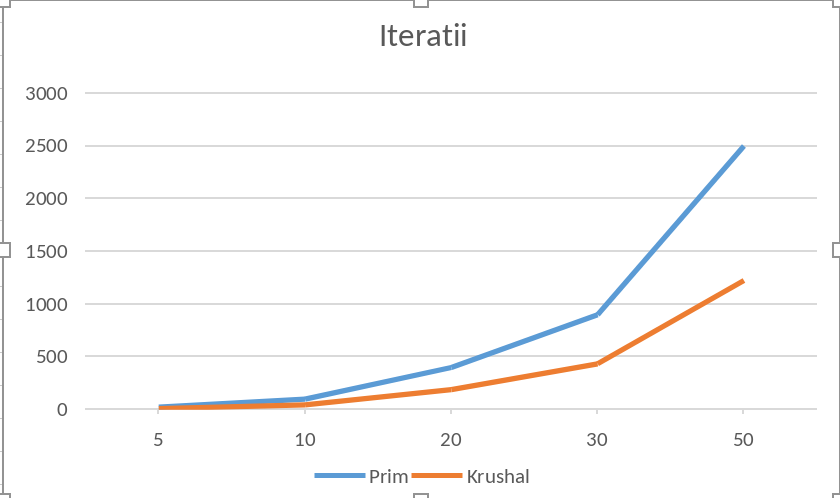
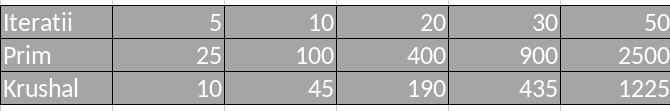
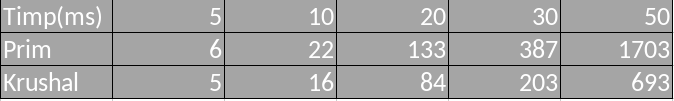
Note de curs: Tehnica greedy

Algoritmii greedy (greedy = lacom) sunt în general simpli şi sunt folosiţi la probleme de optimizare, cum ar fi: să se găsească cea mai bună ordine de executare a unor lucrări pe calculator, să se găsească cel mai scurt drum într-un graf etc. În cele mai multe situaţii de acest fel avem: • o mulţime de candidaţi (lucrări de executat, vârfuri ale grafului etc); • o funcţie care verifică dacă o anumită mulţime de candidaţi constituie o soluţie posibilă, nu neapărat optimă, a problemei; • o funcţie care verifică dacă o mulţime de candidaţi este fezabilă, adică dacă este posibil să completăm această mulţime astfel încât să obţinem o soluţie posibilă, nu neapărat optimă, a problemei; • o funcţie de selecţie care indică la orice moment care este cel mai promiţător dintre candidaţii încă nefolosiţi; • o funcţie obiectiv care dă valoarea unei soluţii (timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit etc); aceasta este funcţia pe care urmărim să o optimizăm (minimizăm/maximizăm). Pentru a rezolva problema de optimizare, se caută o soluţie posibilă care să optimizeze valoarea funcţiei obiectiv. Un algoritm greedy construieşte soluţia pas cu pas. Iniţial, mulţimea candidaţilor selectaţi este vidă. La fiecare pas, se adaugă acestei mulţimi cel mai promiţător candidat, conform funcţiei de selecţie. Dacă, după o astfel de adăugare, mulţimea de candidaţi selectaţi nu mai este fezabilă, se elimină ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulţimea de candidaţi selectaţi este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când se lărgeşte mulţimea candidaţilor selectaţi, se verifică dacă această mulţime nu constituie o soluţie posibilă a problemei. Dacă algoritmul greedy funcţionează corect, prima soluţie găsită va fi totodată o soluţie optimă a problemei. Soluţia optimă nu este în mod necesar unică: se poate că funcţia obiectiv să aibă aceeaşi valoare optimă pentru mai multe soluţii posibile. Funcţia de selecţie este de obicei derivată din funcţia obiectiv; uneori aceste două funcţii sunt chiar identice.

**Codul programului in C++**

#include <iostream>  
#include <vector>  
#include <ctime>  
#include <chrono>  
#include <climits>  
#include <algorithm>  
#include <cstring>  
using namespace std;  
  
// Definim o structura pentru a reprezenta muchi  
struct Edge {  
 int src, dest, weight;  
};  
  
// Clasa pentru reprezentarea unui graf neorientat ponderat  
class Graph {  
public:  
 vector<Edge> edges;  
 int V, E; // V = numarul de varfuri, E = numarul de muchii  
 Graph(int V, int E) {  
 this->V = V;  
 this->E = E;  
 }  
  
 void addEdge(int src, int dest, int weight) {  
 Edge edge = {src, dest, weight};  
 edges.push\_back(edge);  
 }  
};  
  
// Algoritmul Prim pentru arborele de cost minim  
int primMST(Graph& graph) {  
 int V = graph.V;  
 vector<int> key(V, INT\_MAX);  
 vector<bool> mstSet(V, false);  
  
 key[0] = 0;  
  
 for (int count = 0; count < V - 1; count++) {  
 int u = -1;  
 for (int v = 0; v < V; v++)  
 if (!mstSet[v] && (u == -1 || key[v] < key[u]))  
 u = v;  
  
 mstSet[u] = true;  
  
 for (Edge& edge : graph.edges)  
 if (edge.src == u && !mstSet[edge.dest] && edge.weight < key[edge.dest])  
 key[edge.dest] = edge.weight;  
 }  
  
 int totalWeight = 0;  
 for (int i = 0; i < V; i++)  
 totalWeight += key[i];  
  
 return totalWeight;  
}  
  
// Functie pentru compararea muchiilor in algoritmul Kruskal  
bool compareEdges(Edge& a, Edge& b) {  
 return a.weight < b.weight;  
}  
  
// Gasirea setului al carui reprezentant este v  
int find(int parent[], int v) {  
 if (parent[v] == -1)  
 return v;  
 return find(parent, parent[v]);  
}  
  
// Unirea a doua seturi  
void Union(int parent[], int x, int y) {  
 int xset = find(parent, x);  
 int yset = find(parent, y);  
 parent[xset] = yset;  
}  
  
// Algoritmul Kruskal pentru arborele de cost minim  
int kruskalMST(Graph& graph) {  
 int V = graph.V;  
 int E = graph.E;  
  
 vector<Edge> result(V - 1);  
  
 sort(graph.edges.begin(), graph.edges.end(), compareEdges);  
  
 int parent[V];  
 memset(parent, -1, sizeof(parent));  
  
 int e = 0, i = 0;  
 while (e < V - 1 && i < E) {  
 Edge next\_edge = graph.edges[i++];  
  
 int x = find(parent, next\_edge.src);  
 int y = find(parent, next\_edge.dest);  
  
 if (x != y) {  
 result[e++] = next\_edge;  
 Union(parent, x, y);  
 }  
 }  
  
 int totalWeight = 0;  
 for (i = 0; i < V - 1; ++i)  
 totalWeight += result[i].weight;  
  
 return totalWeight;  
}  
  
void generateGraph(Graph& graph, int numNodes) {  
 // Generarea unui graf complet (fiecare nod este conectat la fiecare alt nod)  
 for (int i = 0; i < numNodes; i++) {  
 for (int j = i + 1; j < numNodes; j++) {  
 int weight = rand() % 100 + 1; // Generam un cost random intre 1 si 100  
 graph.addEdge(i, j, weight);  
 graph.addEdge(j, i, weight); // Daca graful este neorientat, adaugam si muchia inversa  
 }  
 }  
}  
  
int main() {  
 int numNodes = 50; // Numarul de noduri  
 Graph graph(numNodes, numNodes \* (numNodes - 1) / 2); // Numarul maxim de muchii intr-un graf complet  
 generateGraph(graph, numNodes);  
 // Algoritmul Prim  
 auto start = chrono::steady\_clock::now();  
 int primWeight = primMST(graph);  
 auto end = chrono::steady\_clock::now();  
 auto primTime = chrono::duration\_cast<chrono::microseconds>(end - start).count();  
  
 // Algoritmul Kruskal  
 start = chrono::steady\_clock::now();  
 int kruskalWeight = kruskalMST(graph);  
 end = chrono::steady\_clock::now();  
 auto kruskalTime = chrono::duration\_cast<chrono::microseconds>(end - start).count();  
  
 // Afiseaza rezultatele  
 cout << "Algoritmul Prim: " << primWeight << " (Timp: " << primTime << " microsecunde)\n";  
 cout << "Algoritmul Kruskal: " << kruskalWeight << " (Timp: " << kruskalTime << " microsecunde)\n";  
  
 return 0;  
}

**Analiza empirică a algoritmilor**



**Concluzie:**

În cadrul acestei lucrări de laborator, am explorat și implementat algoritmii lui Prim și Kruskal pentru găsirea arborelui de cost minim într-un graf ponderat. Am observat că ambele algoritme oferă soluții corecte pentru această problemă, dar au abordări diferite în ceea ce privește strategia de selectare a muchiilor.

Algoritmul lui Prim începe cu un nod arbitrar și extinde arborele prin adăugarea celei mai mici muchii care conectează un nod din arbore cu un nod din afara arborelui. Acest lucru se repetă până când toate nodurile sunt incluse în arbore. Astfel, Prim este un algoritm de tip "growth", care extinde soluția pas cu pas.

Pe de altă parte, algoritmul Kruskal construiește un arbore de cost minim prin alegerea iterativă a celei mai mici muchii disponibile, fără a lua în considerare conectivitatea. Dacă muchia selectată nu formează un ciclu cu muchiile deja selectate, ea este adăugată în arbore.

Am observat că algoritmul lui Prim este mai eficient în cazul grafurilor dense, unde numărul de muchii este apropiat de numărul total de muchii posibile, deoarece acesta are o complexitate de timp mai mică. În schimb, algoritmul Kruskal este potrivit pentru grafuri rare, unde numărul de muchii este semnificativ mai mic decât numărul total de muchii posibile.

În concluzie, am învățat să implementăm și să comparăm acești doi algoritmi esențiali pentru rezolvarea problemei arborelui de cost minim. În funcție de specificul problemei și de caracteristicile grafului, putem alege algoritmul potrivit pentru a obține o soluție eficientă.