**Ministerul Educației și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică și Microelectronică**

**Raport**

**Lucrarea de laborator nr.3:**

**Metoda Greedy.**

**Elaborat: st. gr. TI-232, Galiț Ștefan**

**Verificat: asist. univ. Coșeru Cătălin**

**Chișinău – 2024**

# Scopul lucrării:

* 1. Studierea tehnicii greedy.
  2. Analiza şi implementarea algoritmilor greedy.

**Note teoretice:**

**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.

.

**1. Arbori parţiali de cost minim**

Fie *G* = <*V*, *M>* un graf neorientat conex, unde *V* este mulţimea vârfurilor şi *M* este mulţimea muchiilor. Fiecare muchie are un *cost* nenegativ (sau o *lungime* nenegativă). Problema este să găsim o submulţime *A* Í *M*, astfel încât toate vârfurile din *V* să rămână conectate atunci când sunt folosite doar muchii din *A*, iar suma lungimilor muchiilor din *A* să fie cat mai mică. Căutăm deci o submulţime *A* de cost total minim. Această problemă se mai numeşte şi *problema conectării oraşelor cu cost minim*, având numeroase aplicaţii.

Graful parţial <*V*, *A*> este un arbore şi este *numit arborele parţial de cost minim* al grafului *G* (*minimal spanning tree*). Un graf poate avea mai mulţi arbori parţiali de cost minim. Vom prezenta doi algoritmi greedy care determină arborele parţial de cost minim al unui graf. În terminologia metodei greedy, vom spune că o mulţime de muchii este o *soluţie*, dacă constituie un arbore parţial al grafului *G*, şi este *fezabila*, dacă nu conţine cicluri. O mulţime fezabilă de muchii este *promiţătoare*, dacă poate fi completată pentru a forma soluţia optimă. O muchie *atinge* o mulţime dată de vârfuri, dacă exact un capăt al muchiei este în mulţime.

Mulţimea iniţiala a candidaţilor este *M*. Cei doi algoritmi greedy aleg muchiile una cate una intr-o anumita ordine, această ordine fiind specifică fiecărui algoritm.

**1.1. Algoritmul lui Kruskal**

Arborele parţial de cost minim poate fi construit muchie, cu muchie, după următoarea metoda a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior şi care nu formează cu precedentele un ciclu. Alegem astfel #*V–*1 muchii. Este uşor de dedus că obţinem în final un arbore. Este însa acesta chiar arborele parţial de cost minim căutat?

**Proprietatea 1.** În algoritmul lui Kruskal, la fiecare pas, graful parţial <*V*, *A*> formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei un arbore parţial de cost minim pentru vârfurile pe care le conectează. În final, se obţine arborele parţial de cost minim al grafului *G*.

Pentru a implementa algoritmul, trebuie să putem manipula submulţimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structura de mulţimi disjuncte şi procedurile de tip *find* şi *merge* (Secţiunea 3.5). În acest caz, este preferabil să reprezentăm graful că o lista de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcţie de cost. Iată algoritmul:

**function** *Kruskal*(*G* = <*V*, *M*>)

{iniţializare}

sortează *M* crescător în funcţie de cost

*n* ¬ #*V*

*A* ¬ Æ {va conţine muchiile arborelui parţial de cost minim} iniţializează *n* mulţimi disjuncte conţinând fiecare cate un element din *V*

{bucla greedy}

**repeat**

{*u*, *v*} ¬ muchia de cost minim care încă nu a fost considerată

*ucomp* ¬ *find*(*u*) *vcomp* ¬ *find*(*v*)

**if** *ucomp* ¹ *vcomp* **then** *merge*(*ucomp*, *vcomp*)

*A* ¬ *A* È {{*u*, *v*}}

**until** #*A* = *n-*1

**return** *A*

Pentru un graf cu *n* vârfuri şi *m* muchii, presupunând că se folosesc procedurile *find*3 şi *merge*3, numărul de operaţii pentru cazul cel mai nefavorabil este în:

* *O*(*m* log *m*) pentru a sorta muchiile. Deoarece *m* £ *n*(*n*–1)/2, rezulta *O*(*m* log *m*) Í *O*(*m* log *n*). Mai mult, graful fiind conex, din *n*-1 £ *m* rezulta şi *O*(*m* log *n*) Í *O*(*m* log *m*), deci *O*(*m* log *m*) = *O*(*m* log *n*).
* *O*(*n*) pentru a iniţializa cele *n* mulţimi disjuncte.
* Cele cel mult 2*m* operaţii *find*3 şi *n*–1 operaţii *merge*3 necesita un timp în *O*((2*m*+*n*-1) lg\* *n*). Deoarece

*O*(lg\* *n*) Í *O*(log *n*) şi *n*-1 £ *m*, acest timp este şi în *O*(*m* log *n*).

* *O*(*m*) pentru restul operaţiilor.

Deci, pentru cazul cel mai nefavorabil, algoritmul lui Kruskal necesită un timp în *O*(*m* log *n*).

O altă variantă este să păstram muchiile într-un min-heap. Obţinem astfel un nou algoritm, în care iniţializarea se face într-un timp în *O*(*m*), iar fiecare din cele *n–*1 extrageri ale unei muchii minime se face într-un timp în *O*(log *m*) = *O*(log *n*). Pentru cazul cel mai nefavorabil, ordinul timpului rămâne acelaşi cu cel al vechiului algoritm. Avantajul folosirii min-heap-ului apare atunci când arborele parţial de cost minim este găsit destul de repede şi un număr considerabil de muchii rămân netestate. În astfel de situaţii, algoritmul vechi pierde timp, sortând în mod inutil şi aceste muchii.

**1.2. Algoritmul lui Prim**

Cel de-al doilea algoritm greedy pentru determinarea arborelui parţial de cost minim al unui graf se datorează lui Prim (1957). În acest algoritm, la fiecare pas, mulţimea *A* de muchii alese împreună cu mulţimea *U* a vârfurilor pe care le conectează formează un arbore parţial de cost minim pentru subgraful

<*U*, *A*> al lui *G*. Iniţial, mulţimea *U* a vârfurilor acestui arbore conţine un singur vârf oarecare din *V*, care

va fi rădăcina, iar mulţimea *A* a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naştere unui nou arbore parţial de cost minim. Arborele parţial de cost minim creste “natural”, cu cate o ramură, până când va atinge toate vârfurile din *V*, adică până când *U* = *V*.

**Proprietatea 2.** În algoritmul lui Prim, la fiecare pas, <*U*, *A*> formează un arbore parţial de cost minim pentru subgraful <*U*, *A*> al lui *G*. În final, se obţine arborele parţial de cost minim al grafului *G*.

Presupunem că vârfurile din *V* sunt numerotate de la 1 la *n*, *V* = {1, 2, ..., *n*}, matricea simetrică *C* dă costul fiecărei muchii, cu *C*[*i*, *j*] = +¥, dacă muchia {*i*, *j*} nu există. Folosim două tablouri paralele. Pentru fiecare *i* Î *V* \ *U*, *vecin*[*i*] conţine vârful din *U*, care este conectat la *i* printr-o muchie de cost minim, *mincost*[*i*] dă acest cost. Pentru *i* Î *U*, punem *mincost*[*i*] = –1. Mulţimea *U*, în mod arbitrar iniţializată cu {1}, nu este reprezentată explicit. Elementele *vecin*[1] şi *mincost*[1] nu se folosesc.

**function** *Prim*(*C*[1 .. *n*, 1 .. *n*])

{iniţializare, numai vârful 1 este în *U*}

*A* ¬ Æ

**for** *i* ¬ 2 **to** *n* **do** *vecin*[*i*] ¬ 1

*mincost*[*i*] ¬ *C*[*i*, 1]

{bucla greedy}

**repeat** *n–*1 **times**

*min* ¬ +¥

**for** *j* ¬ 2 **to** *n* **do**

**if** 0 < *mincost*[ *j*] < *min* **then** *min* ¬ *mincost*[ *j*]

*k* ¬ *j*

*A* ¬ *A* È {{*k*, *vecin*[*k*]}}

*mincost*[*k*] ¬ –1 {adaugă vârful *k* la *U*}

**for** *j* ¬ 2 **to** *n* **do**

**if** *C*[*k, j*] < *mincost*[ *j*] **then** *mincost*[ *j*] ¬ *C*[*k*, *j*]

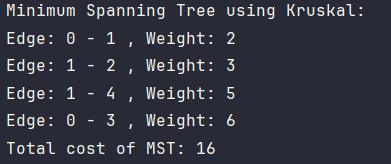
*vecin*[ *j*] ¬ *k*

**return** *A*

Bucla principală se execută de *n–*1 ori şi, la fiecare iteraţie, buclele **for** din interior necesită un timp în *O*(*n*). Deci, algoritmul *Prim* necesită un timp în *O*(*n*2). Am văzut că timpul pentru algoritmul lui Kruskal este în *O*(*m* log *n*), unde *m =* #*M*. Pentru un graf *dens* se deduce că *m* se apropie de *n*(*n–*1)/2. În acest caz, algoritmul *Kruskal* necesită un timp în *O*(*n*2 log *n*) şi algoritmul *Prim* este probabil mai bun. Pentru un graf *rar m* se apropie de *n* şi algoritmul *Kruskal* necesită un timp în *O*(*n* log *n*), fiind probabil mai eficient decât algoritmul *Prim*.

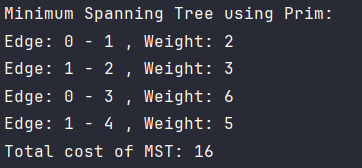
**Rezultatul rulării programului:**

În figura (1) este afișat rezultatul rulării programului, în consolă este afișat timpul de execuție a algoritmului.



Figura(1) rezultatul rulării.

În Anexa B este prezentat codul algoritmului și în figura (2) este prezentat rezultatul rulării programului.



Figura(2) rezultatul rulării.

# Concluzie

Pe parcursul lucrării, am explorat tehnica greedy și aplicabilitatea acesteia în rezolvarea problemelor de optimizare, analizând modul în care algoritmii greedy identifică soluții rapide, dar nu neapărat globale optime, prin selectarea la fiecare pas a candidatului cel mai promițător. Implementarea algoritmilor lui Kruskal și Prim pentru găsirea arborelui parțial de cost minim a demonstrat că fiecare dintre aceștia este potrivit în funcție de densitatea grafului, subliniind adaptabilitatea și eficiența tehnicii greedy în anumite contexte de optimizare. Această analiză evidențiază valoarea abordărilor greedy în obținerea unor soluții rapide și eficiente pentru probleme complexe.

**Anexa A**

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
public class KruskalAlgorithm {  
 private Graph graph;  
  
 public KruskalAlgorithm(Graph *graph*) {  
 this.graph = *graph*;  
 }  
  
 public void kruskalMST() {  
 Collections.**sort**(graph.edges);  
 int[] parent = new int[graph.V];  
 for (int i = 0; i < graph.V; i++)  
 parent[i] = i;  
  
 List<Edge> result = new ArrayList<>();  
 int totalWeight = 0; // Total cost of MST  
  
 for (Edge edge : graph.edges) {  
 int x = find(parent, edge.src);  
 int y = find(parent, edge.dest);  
  
 if (x != y) {  
 result.add(edge);  
 totalWeight += edge.weight;  
 union(parent, x, y);  
 }  
 }  
  
 System.out.println("Minimum Spanning Tree using Kruskal:");  
 for (Edge edge : result) {  
 System.out.println("Edge: " + edge.src + " - " + edge.dest + " , Weight: " + edge.weight);  
 }  
 System.out.println("Total cost of MST: " + totalWeight);  
 }  
  
 private int find(int[] *parent*, int *i*) {  
 if (*parent*[*i*] != *i*)  
 *parent*[*i*] = find(*parent*, *parent*[*i*]);  
 return *parent*[*i*];  
 }  
  
 private void union(int[] *parent*, int *x*, int *y*) {  
 int xroot = find(*parent*, *x*);  
 int yroot = find(*parent*, *y*);  
 *parent*[xroot] = yroot;  
 }  
}

*Anexa A* codul algoritmului Kruskal.

**Anexa B**

import java.util.Arrays;  
  
public class PrimAlgorithm {  
 private Graph graph;  
  
 public PrimAlgorithm(Graph *graph*) {  
 this.graph = *graph*;  
 }  
  
 public void primMST() {  
 boolean[] mstSet = new boolean[graph.V];  
 int[] key = new int[graph.V];  
 int[] parent = new int[graph.V];  
 int totalWeight = 0;  
  
 Arrays.**fill**(key, Integer.MAX\_VALUE);  
 key[0] = 0;  
 parent[0] = -1;  
  
 for (int count = 0; count < graph.V - 1; count++) {  
 int u = minKey(key, mstSet);  
 mstSet[u] = true;  
  
 for (Edge edge : graph.edges) {  
 if (edge.src == u || edge.dest == u) {  
 int v = (edge.src == u) ? edge.dest : edge.src;  
 if (!mstSet[v] && edge.weight < key[v]) {  
 parent[v] = u;  
 key[v] = edge.weight;  
 }  
 }  
 }  
 }  
  
 System.out.println("Minimum Spanning Tree using Prim:");  
 for (int i = 1; i < graph.V; i++) {  
 System.out.println("Edge: " + parent[i] + " - " + i + " , Weight: " + key[i]);  
 totalWeight += key[i];  
 }  
 System.out.println("Total cost of MST: " + totalWeight);  
 }  
  
 private int minKey(int[] *key*, boolean[] *mstSet*) {  
 int min = Integer.MAX\_VALUE, minIndex = -1;  
  
 for (int v = 0; v < graph.V; v++) {  
 if (!*mstSet*[v] && *key*[v] < min) {  
 min = *key*[v];  
 minIndex = v;  
 }  
 }  
 return minIndex;  
 }  
}

*Anexa B* codul algoritmului Prim.