Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Catedra ATI

**RAPORT**

Lucrarea de laborator nr.3

*Analiza si proiectarea algoritmilor*

A efectuat:

st. gr. TI-15X Suruceanu Valentin

A verificat:

lect., univ. V.Bagrin

Chisinau 2016

**LUCRAREA DE LABORATOR NR.3**

**Tema:** Tehnica greedy de proiectare a algoritmilor

**Scopul lucrării:**

* Studierea tehnicii greedy.
* Analiza și implementarea algoritmilor greedy.

**Note de curs**

Algoritmii *greedy* (greedy = lacom) sunt în general simpli şi sunt folosiţi la rezolvarea problemelor 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:

* 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;

1. o *funcţie de selecţie* care indică la orice moment care este cel mai promiţător dintre candidaţii încă nefolosiţi;
2. o *funcţie obiectiv* care dă valoarea unei soluţii (timpul necesar executării tuturor lucrărilor într-o anumită ordine , lungimea drumului pe care lam gasit) aceasta este fuctia pe care urmarim sa o optimizam (minimizam)

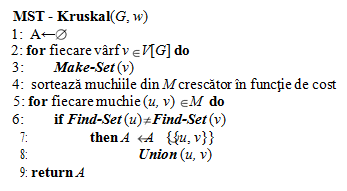


**Algoritmul lui Kruskal**

Arborele de acoperire 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.

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 de acoperire minim pentru vârfurile pe care le conectează. În final, se obţine arborele parţial de cost minim al grafului *G*.

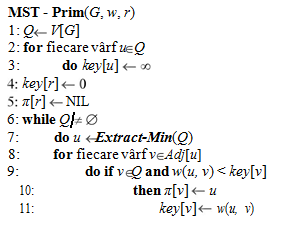
Pentru a implementa algoritmul, trebuie să putem manipula submulţimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structura de date pentru mulţimi disjuncte pentru prezentarea mai multor mulţimi de elemente disjuncte [Cormen]. Fiecare mulţime conţine vârfurile unui arbore din pădurea curentă. Funcţia ***Find-Set*** (*u*) returnează un element reprezentativ din mulţimea care îl conţine pe*u*. Astfel, putem determina dacă douăvârfuri *u* şi *v* aparţin aceluiaţi arbore testând dacă ***Find-Set*** (*u*) este egal cu ***Find-Set*** (*v*). Combinarea arborilor este realizată de procedura ***Union***. În acest caz, este preferabil să reprezentăm graful ca o lista de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcţie de cost. În continuare este prezentat algoritmul:



**Algoritmul lui Prim**

Cel de-al doilea algoritm greedy pentru determinarea arborelui de acoperire minimal 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 firădăcina, iar mulţimea *A* a muchiiloreste 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*.

Cheia implementării eficiente a algoritmului lui Prim este să procedăm în aşa fel încât să fie uşor să selectăm o nouă muchie pentru a fi adăugată la arborele format de muchiile din *A*. În pseudocodul de mai jos, graful conex *G* şi rădăcina *r* a arborelui minim de acoperire, care urmează a fi dezvoltat, sunt privite ca date de intrare pentru algoritm. În timpul execuţiei algoritmului, toate vârfurile care *nu* sunt în arbore se află într-o coadă de prioritate *Q* bazată pe un câmp *key*. Pentru fiecare vârf *v*, *key*[*v*] este costul minim al oricărei muchii care îl uneşte pe *v* cu un vărf din arbore. Prin convenţie, *key*[*v*] = dacă nu există o astfel de muchie. Câmpul *π*[*v*] reţine „părintele” lui *v* din arbore. *Adj*[*u*] este lista de adiacenţă cu vârful *u*.



**SARCINA DE BAZĂ:**

1. De studiat tehnica greedy de proiectare a algoritmilor.

2. De implementat într-un limbaj de programare algoritmii Prim şi Kruskal.

3. De făcut analiza empirică a algoritmilor Kruskal şi Prim.

4. De alcătuit un raport.

**Codul programului in JAVA**

import java.util.\*;

import java.lang.\*;

class Graph {

static int noEdges, noConx;

static int[][] adiacentMatrix, weightMatrix, kruscalTree, adiacentWeightMatrix;

static private ArrayList<Integer>[] list;

static long startTime,stopTime,duration;

static long iteratii=0;

public static void makeList(int[][] aM) {

ArrayList<Integer>[] lst = new ArrayList[noEdges];

for (int i=0; i<noEdges;i++) {

lst[i] = new ArrayList<Integer>();

}

for (int i=0; i<noEdges; i++) {

for (int j=0; j<noEdges;j++) {

if(adiacentMatrix[i][j] == 1) {

lst[i].add(j);

lst[j].add(i);

}

}

}

list = lst;

}

public static void showList() {

for(int i=0;i<noEdges; i++) {

System.out.printf("[%d]\t",i);

for (int j=0;j<list[i].size();j++) {

System.out.printf("%d\t", list[i].get(j));

}

System.out.printf("\n\n");

}

}

//caz mediu

public static void initGraph(int v,int n) {

int i,j;

int[][] init = new int[v][v];

Random rand = new Random();

int edgeFrom,edgeTo,stop = 0;

while(stop != n) {

edgeFrom = rand.nextInt(v);

edgeTo = rand.nextInt(v);

if (init[edgeFrom][edgeTo] == 1) continue;

init[edgeFrom][edgeTo] = 1;

stop++;

}

adiacentMatrix = init;

}

//caz favorabil

public static void initGraph(int v) {

noConx = 1;

int i,j=0;;

int[][] init = new int[v][v];

for(i = 0; i<v; i++) {

if (i+1 < v) j=i+1;

if (i == v-1) break;

init[i][j] = 1;

}

noConx=v-1;

adiacentMatrix = init;

}

public static void showGraph() {

for (int i = 0; i<noEdges; i++) {

for (int j = 0; j<noEdges; j++) {

System.out.printf(" %d",adiacentMatrix[i][j]);

}

System.out.printf("\n");

}

}

public static void setWeightMatrix() {

weightMatrix = new int [noConx][3];

int k = 0;

Scanner input = new Scanner(System.in);

Random rand = new Random();

for (int i = 0; i<noEdges; i++) {

for (int j = 0; j <noEdges; j++) {

if (adiacentMatrix[i][j] == 1) {

weightMatrix[k][0]=i;

weightMatrix[k][1]=j;

weightMatrix[k][2]= rand.nextInt(10000);

k++;

}

}

}

}

public static void showWeightMatrix() {

for (int i = 0; i<noConx; i++) {

for (int j = 0; j<3; j++) {

System.out.printf("\t%d",weightMatrix[i][j]);

}

System.out.println();

}

}

int findWeight(int u,int v) {

for (int i=0; i<noConx;i++) {

if ((weightMatrix[i][0] == u)&&(weightMatrix[i][1] == v) || (weightMatrix[i][1] == u)&&(weightMatrix[i][0] == v)) {

return weightMatrix[i][2];

}

}

return 0;

}

void setAdiacentWeightM (int[][] aM,int[][] wM) {

adiacentWeightMatrix = new int[noEdges][noEdges];

for (int i=0; i<noEdges;i++) {

for(int j=0; j<noEdges;j++) {

if (adiacentMatrix[i][j] == 1) {

adiacentWeightMatrix[i][j] = findWeight(i,j);

adiacentWeightMatrix[j][i] = findWeight(i,j);

}

}

}

}

void printMST(int parent[], int n, int graph[][])

{

System.out.println("Edge Weight");

for (int i = 1; i < noEdges; i++)

System.out.println(parent[i]+" - "+ i+" "+

graph[i][parent[i]]);

}

public static int minKey(int key[], Boolean mstSet[])

{

// Initialize min value

int min = Integer.MAX\_VALUE, min\_index=-1;

for (int v = 0; v < noEdges; v++) {

if (mstSet[v] == false && key[v] < min)

{

min = key[v];

min\_index = v;

}

}

return min\_index;

}

void primMST(int graph[][])

{

// Array to store constructed MST

int parent[] = new int[noEdges];

// Key values used to pick minimum weight edge in cut

int key[] = new int [noEdges];

// To represent set of vertices not yet included in MST

Boolean mstSet[] = new Boolean[noEdges];

// Initialize all keys as INFINITE

for (int i = 0; i < noEdges; i++)

{

key[i] = Integer.MAX\_VALUE;

mstSet[i] = false;

}

// Always include first 1st vertex in MST.

for (int x = 0; x<noEdges; x++) {

if(!list[x].isEmpty()) {

key[x] = 0;

break;

}

} // Make key 0 so that this vertex is

// picked as first vertex

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < noEdges-1; count++)

{

// Pick thd minimum key vertex from the set of vertices

// not yet included in MST

int u = minKey(key, mstSet);

if(u == -1) break;

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of the adjacent

// vertices of the picked vertex. Consider only those

// vertices which are not yet included in MST

for (int v = 0; v < noEdges; 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 (graph[u][v]!=0 && mstSet[v] == false &&

graph[u][v] < key[v])

{

parent[v] = u;

key[v] = graph[u][v];

}

}

// print the constructed MST

//printMST(parent, noEdges, graph);

}

class Edge implements Comparable<Edge>

{

int src, dest, weight;

// Comparator function used for sorting edges based on

// their weight

public int compareTo(Edge compareEdge)

{

return this.weight-compareEdge.weight;

}

};

// A class to represent a subset for union-find

class subset

{

int parent, rank;

};

int V=noEdges, E=noConx; // V-> no. of vertices & E->no.of edges

Edge edge[]; // collection of all edges

// A utility function to find set of an element i

// (uses path compression technique)

int find(subset subsets[], int i)

{

// find root and make root as parent of i (path compression)

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

void Union(subset subsets[], int x, int y)

{

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of high rank tree

// (Union by Rank)

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and increment

// its rank by one

else

{

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// The main function to construct MST using Kruskal's algorithm

void KruskalMST()

{

Edge result[] = new Edge[V]; // Tnis will store the resultant MST

int e = 0; // An index variable, used for result[]

int i = 0; // An index variable, used for sorted edges

for (i=0; i<V; ++i)

result[i] = new Edge();

// Step 1: Sort all the edges in non-decreasing order of their

// weight. If we are not allowed to change the given graph, we

// can create a copy of array of edges

Arrays.sort(edge);

// Allocate memory for creating V ssubsets

subset subsets[] = new subset[V];

for(i=0; i<V; ++i)

subsets[i]=new subset();

// Create V subsets with single elements

for (int v = 0; v < V; ++v)

{

subsets[v].parent = v;

subsets[v].rank = 0;

}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1

while (e < V - 1)

{

// Step 2: Pick the smallest edge. And increment the index

// for next iteration

Edge next\_edge = new Edge();

next\_edge = edge[i++];

if(i>=V) break;

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge does't cause cycle, include it

// in result and increment the index of result for next edge

if (x != y)

{

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// print the contents of result[] to display the built MST

/\*System.out.println("Following are the edges in the constructed MST");

for (i = 0; i < e; ++i)

System.out.println(result[i].src+" -- "+result[i].dest+" == "+

result[i].weight);

\*/

}

Graph (int v,int n) {

noEdges=v;

noConx = n;

initGraph(v,n);

setWeightMatrix();

makeList(adiacentMatrix);

setAdiacentWeightM(adiacentMatrix,weightMatrix);

V = noEdges;

E = noConx;

edge = new Edge[E];

for (int i=0; i<E; i++){

edge[i] = new Edge();

edge[i].src = weightMatrix[i][0];

edge[i].dest = weightMatrix[i][1];

edge[i].weight = weightMatrix[i][2];

}

System.out.println("initialized!");

}

Graph (int v) {

noEdges=v;

noConx=v;

initGraph(v);

setWeightMatrix();

makeList(adiacentMatrix);

setAdiacentWeightM (adiacentMatrix,weightMatrix);

V = noEdges;

E = noConx;

edge = new Edge[E];

for (int i=0; i<E; i++){

edge[i] = new Edge();

edge[i].src = weightMatrix[i][0];

edge[i].dest = weightMatrix[i][1];

edge[i].weight = weightMatrix[i][2];

}

System.out.println("initialized!");

}

public static void main(String[] args) {

int n,v;

Scanner input = new Scanner(System.in);

System.out.printf("Number of edges: ");

n = input.nextInt();

System.out.printf("Number of conexions: ");

v = input.nextInt();

Graph g = new Graph(n,v);

g.showList();

System.out.printf("PRIM: \n");

startTime = System.nanoTime();

g.primMST(g.adiacentWeightMatrix);

stopTime = System.nanoTime();

duration = stopTime - startTime;

System.out.printf("Time: %d nS\n",duration);

startTime = System.nanoTime();

System.out.printf("Kruskal: \n");

g.KruskalMST();

stopTime = System.nanoTime();

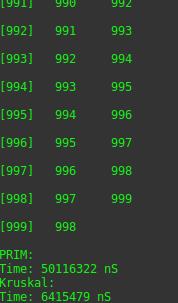
duration = stopTime - startTime;

System.out.printf("Time: %d nS\n",duration);

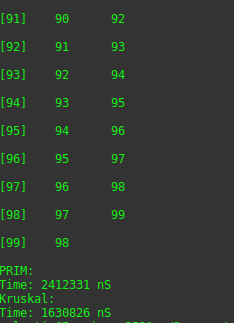
}

}

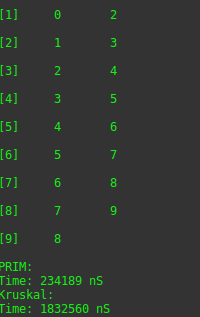
**Cazul favorabil**



***Fig 1.*** Alg.Prim & Alg.Kruskal 1000 noduri

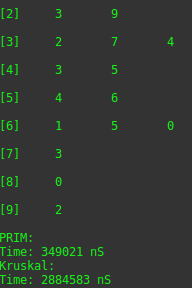
****

***Fig 2.*** Alg.Prim & Alg.Kruskal 100 noduri

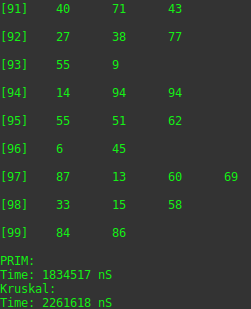
****

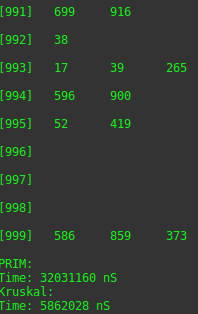
***Fig 3.*** Alg.Prim & Alg. Kruskal 10 noduri

**Cazul mediu**

****

***Fig 4.*** Alg.Prim & Alg. Kruskal 10 noduriri

****

***Fig 5.*** Alg.Prim & Alg. Kruskal 100 noduri

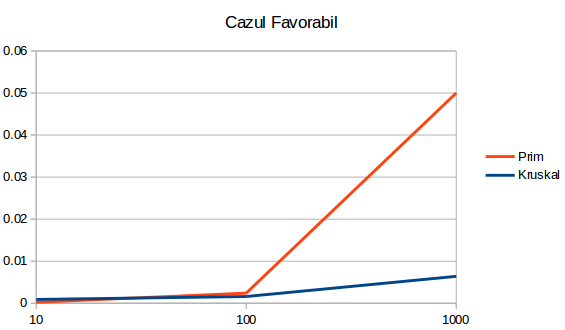
***Fig 6.*** Alg.Prim & Alg. Kruskal 1000 noduri

**Cazul Favorabil**

|  |  |  |
| --- | --- | --- |
| **Algoritmul** | **Nr.de virfuri** | **Timpul** |
| Kruskal | 1000 | 0.0064 |
| Kruskal | 100 | 0.0016 |
| Kruskal | 10 | 0.0018 |
| Prim | 1000 | 0.0500 |
| Prim | 100 | 0.0024 |
| Prim | 10 | 0.00023 |

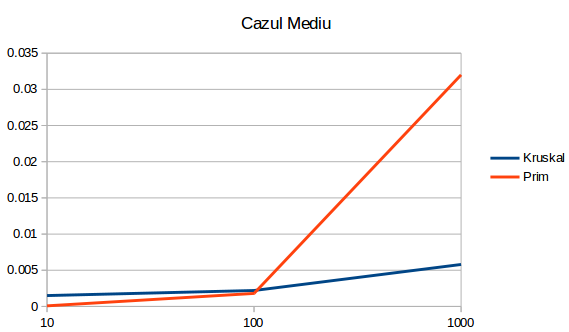
**Cazul mediu**

|  |  |  |
| --- | --- | --- |
| **Algoritmul** | **Nr.de virfuri** | **Timpul** |
| Kruskal | 1000 | 0.0058 |
| Kruskal | 100 | 0.0022 |
| Kruskal | 10 | 0.0015 |
| Prim | 1000 | 0.0320 |
| Prim | 100 | 0.0018 |
| Prim | 10 | 0.00008 |

****

Kruskal

Prim

****

**Concluzie**

Realizînd laboratorul dat am obținut deprinderi noi în programarea algoritmelor Greedy. După rezultatele obținute se observa ca algoritmul Kruscal este considerabil mai efectiv din punct de vedere al timpului de executie decât algoritmul Prim deoarece programînd algoritmul Prim am folosit tabloul bidimensional dar calculînd complexitatea după pseudocod se observa ca algoritmul Kruskal are un timp mai efectiv decât algoritmul Prim.