Ministerul Educaţiei, Culturii și Cercetării  
Universitatea Tehnică a Moldovei

Departamentul Informatică și Ingineria Sistemelor

**RAPORT**

Lucrarea de laborator nr.3  
la Analiza și Proiectarea Algoritmilor

A efectuat:   
st. gr. TI-206 Borș Nicoleta

A verificat: Ernest Bîtca

Chişinău – 2021

**Tema**: Algoritmi Greedy

**Scopul lucrării:** Studierea tehnicii greedy. Analiza și implementarea algoritmilor greedy

**Sarcina:**

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.

**Întrebări de control:**

1. Descrieţi metoda greedy.

Algoritmii greedy (greedy = lacom) 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. Un algoritm greedy construieşte soluţia pas cu pas.

Un astfel de algoritm se numește “lacom” (am putea să-i spunem și “nechibzuit”). La fiecare pas, procedura alege cel mai bun candidat la momentul respectiv, fără să-i pese de viitor și fără să se răzgândească. Dacă un candidat este inclus în soluție, el rămâne acolo; dacă un candidat este exclus din soluție, el nu va mai fi niciodată reconsiderat. Un algoritm greedy acționează simplist, totuși, ca și în afaceri, o astfel de metodă poate da rezultate foarte bune tocmai datorită simplității ei.

1. Când se aplică algoritmul Kruskal şi când algoritmul Prim ?

Pentru un graf dens (adică, cu foarte multe muchii), se deduce că m se apropie de n(n–1)/2. În acest caz, algoritmul Kruskal necesită un timp în O(n2 log n) și algoritmul Prim este mai bun. Pentru un graf rar (adică, cu un număr foarte mic de muchii), m se apropie de n și algoritmul Kruskal necesită un timp în O(n log n), fiind mai eficient decât algoritmul Prim. (n – nr de vârfuri, m – nr de muchii)

1. Ce tip de date este comod de folosit la elaborarea programului al unui algoritm de tip greedy?

Pentru algoritmul Kruskal 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. Putem să folosim și tablouri bidimensionale pentru ambii algoritmi.

1. Care sunt avantajele şi dezavantajele algoritmilor Prim si Kruskal?

Pentru un graf dens m se apropie de n(n–1)/2 iar în acest caz, algoritmul Kruskal necesită un timp în O(n2 log n). Pentru un graf rar, m se apropie de n și algoritmul Kruskal necesită un timp în O(n log n), fiind mai eficient decât algoritmul Prim. Astfel dezavantajul algoritmului Prim fiind grafurile cu multe vârfuri iar a lui Kruskal grafurile cu multe muchii. Kruskal însă este mai ușor de implementat decât Prim.

**Teorie:**

**Algoritmul lui Kruskal**

Arborele parțial de cost minim poate fi construit muchie, cu muchie, după următoarea metodă a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Alegem astfel V–1 muchii.

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

#### 

#### **Figura 1** Un graf și arborele său parțial de cost minim

#### 

#### **Figura 2** Algoritmul lui Kruskal aplicat grafului din Figura 1

#### **Pseudocodul:**

#### 

#### Pentru un graf cu n vârfuri și m muchii, presupunând că se folosesc procedurile find și merge, numărul de operații pentru cazul cel mai nefavorabil este în:

#### • O(m log m) pentru a sorta muchiile.

#### • O(n) pentru a inițializa cele n mulțimi disjuncte.

#### • Cele cel mult 2m operații find și n–1 operații merge necesită un timp în O((2m+n−1) lg\* n)

#### • O(m) pentru restul operațiilor.

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

#### O alta variantă este să păstrăm muchiile într-un min-heap. Obținem astfel un nou algoritm, în care inițializarea 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 același cu cel al vechiului algoritm. Avantajul folosirii min-heap-ului apare atunci când arborele parțial de cost minim este găsit destul de repede și un număr considerabil de muchii rămân netestate. În astfel de situații, algoritmul vechi pierde timp, sortând în mod inutil și aceste muchii.

#### **Algoritmul lui Prim**

#### În algoritmul lui Prim, la fiecare pas, <U, A> formează un arbore parțial de cost minim pentru subgraful <U, A> al lui G. În final, se obține arborele parțial de cost minim al grafului G.

#### Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf oarecare din V, care va fi rădăcina, iar mulțimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naștere unui nou arbore parțial de cost minim.

#### 

#### **Figura 3** Algoritmul lui Prim aplicat grafului din Figura 1a

#### Bucla principală se execută de n–1 ori și, la fiecare iterație, buclele for din interior necesită un timp în O(n). Algoritmul Prim necesită, deci, un timp în **O(n2).** Am văzut că timpul pentru algoritmul lui Kruskal este în O(m log n).

#### **Pseudocodul:**

#### 

#### **Rezultatele programului:**

#### **Tabelul 1** – Timpul (ms) de execuție pentru Algoritmul Prim și Kruskal pentru un graf dens

|  |  |  |
| --- | --- | --- |
| **Graf dens** | Prim | Kruskal |
| n = 10 | 0.39 | 0.7 |
| n = 50 | 8.79 | 10.8 |
| n = 100 | 22.8 | 63.4 |
| n = 500 | 263.8 | 15515.6 |
| n = 1000 | 1263 | - |

#### Din tabelul de mai sus observăm că pentru un graf dens algoritmul Kruskal este cu mult mai lent decât Prim indiferent de numărul vârfurilor, deoarece numărul muchiilor este mare, acesta fiind cazul defavorabil a algoritmului Prim.

#### Din grafic observăm că algoritmul Prim este cu mult mai rapid decât Kruskal. Mai mult decât atât, timpul de execuție nu este atât de tare influențat de n în algoritmul Prim în comparație cu Kruskal.

#### **Tabelul 2** – Timpul (ms) de execuție pentru Algoritmul Prim și Kruskal pentru un graf rar

|  |  |  |
| --- | --- | --- |
| **Graf rar** | Prim | Kruskal |
| n = 10 | 0.5 | 0.4 |
| n = 50 | 8.9 | 8.19 |
| n = 100 | 14.4 | 33.3 |
| n = 500 | 211.3 | 10722.5 |
| n = 1000 | 1727.6 | - |

#### Pentru un graf rar, Kruskal este mai eficient, decât algoritmul Prim atunci când numărul vârfurilor este mic, însă la n mare, Prim este mai eficient decât Krukal, chiar dacă dezavantajul algoritmului Prim este graful cu multe vârfuri.

#### În graficul de mai sus la fel observăm cât de rapid este algoritmul Prim. În ambele cazuri, atât pentru un graf dens, cât și pentru un graf rar, algoritmul Kruskal nu a suportat un graf de 1000 vârfuri.

#### Ca să analizăm timpul de execuție pentru fiecare algoritm în dependență de tipul grafului o să analizăm graficele de mai jos:

#### Observăm că algoritmul Prim este mai rapid pentru grafuri rare, atunci când numărul de vârfuri este mai mic, sau pentru grafuri dense când n-ul este mai mare. Algoritmul Kruskal însă este mai rapid pentru grafuri rare indiferent de numărul vârfurilor.

#### **Concluzie:**

#### În cadrul efectuării lucrării de laborator la tema “Algoritmii Greedy” am avut posibilitatea să studiez algoritmul Kruskal și Prim, avatnajele și dezavantajele acestora. Am elaborat un program care generează un graf rar, dens sau permite citirea unuia de la tastatură și determină arborelui cu cost minim prin algoritmul Prim sau Kruskal, astfel în urma executării programului pentru cazuri diferite am ajuns la concluzia că algoritmul Prim, chiar dacă este mai greu de implementat, are rezultate mai bune decât Kruskal în cele mai multe cazuri.

#### Algoritmul Kruskal întâmpină probleme atunci când numărul muchiilor este foarte mare, pe când algoritmul Prim atunci când numărul vărfurilor este mare, astfel dacă sunt grafuri rare cu multe vârfuri algoritmul Kruskal poate fi mai eficient, pe când algoritmul Prim este eficient pentru grafurile dense, care e punctul slab a algoritmului Kruskal.

#### **Bibliografia:**

#### Else APA16.3 - <https://else.fcim.utm.md/course/view.php?id=773>

#### Else APA16.1 - <https://else.fcim.utm.md/course/view.php?id=7>

#### Wikipedia - Algoritm greedy - <https://ro.wikipedia.org/wiki/Algoritm_greedy>

#### Algoritmi fundamentali -

#### <https://www.cwu.edu/faculty/sites/cts.cwu.edu.faculty/files/users/142/documents/cartea%20de%20algoritmi.pdf>

#### GitHub - <https://github.com/Nicoleta-Bors/APA.git>

#### Anexa 1 – Codul programului

let graf = [];

let n;

const unGraf = () => {

  n = 7;

  graf = [

    [0, Infinity, Infinity, Infinity, 12, 10, 7],

    [Infinity, 0, Infinity, Infinity, Infinity, Infinity, Infinity, 8],

    [Infinity, Infinity, 0, 5, Infinity, 15, Infinity],

    [Infinity, Infinity, 5, 0, 18, 20, Infinity],

    [12, Infinity, Infinity, 18, 0, Infinity, 5],

    [10, Infinity, 15, 20, Infinity, 0, Infinity],

    [7, 8, Infinity, Infinity, 5, Infinity, 0],

  ];

};

const initiereMatrice = (n, complete) => {

  for (let i = 0; i < n; i++) {

    graf[i] = [];

    for (let j = 0; j < n; j++) {

      graf[i][j] = complete;

    }

  }

};

const citireGraf = () => {

  n = parseInt(prompt('Introduceti numarul de varfuri al grafului: '));

  initiereMatrice(n, -1);

  for (let i = 0; i < n; i++) {

    for (let j = 0; j < n; j++) {

      if (i === j) {

        graf[i][j] = 0;

      } else {

        if (graf[i][j] === -1) {

          graf[i][j] = graf[j][i] = parseInt(

            prompt(`Introduceti ponderea muchiei [${i}] - [${j}]:`),

          );

        }

      }

    }

  }

};

const grafRar = () => {

  let random1, random2;

  initiereMatrice(n, Infinity);

  for (let i = 0; i < n - 1; i++) {

    graf[i][i + 1] = graf[i + 1][i] = Math.floor(Math.random() \* n) + 1;

    random1 = Math.floor(Math.random() \* (n - 1));

    random2 = Math.floor(Math.random() \* (n - 1));

    if (random1 !== random2) {

      graf[random1][random2] = graf[random2][random1] = Math.floor(Math.random() \* n) + 1;

    }

  }

};

const grafDens = () => {

*// n = 10;*

  initiereMatrice(n, Infinity);

  for (let i = 0; i < n; i++) {

    for (let j = 0; j < n; j++) {

      if (i === j) {

        graf[i][j] = 0;

      } else {

        if (graf[i][j] === Infinity) {

          graf[i][j] = graf[j][i] = Math.floor(Math.random() \* n) + 1;

        }

      }

    }

  }

};

*// ---------------------------------------------------------------- Kruskal*

const find = (i, parent) => {

  while (parent[i] != i) i = parent[i];

  return i;

};

const union = (i, j, parent) => {

  let a = find(i, parent);

  let b = find(j, parent);

  parent[a] = b;

};

const kruskal = (graf) => {

  console.log(`%cAlgoritmul lui Kruskal`, 'color: #E05297');

  let parent = Array(n).fill(0);

  let mincost = 0;

  let nrMuchii = 0;

  for (let i = 0; i < n; i++) parent[i] = i;

  while (nrMuchii < n - 1) {

    let min = Infinity,

      a = -1,

      b = -1;

    for (let i = 0; i < n; i++) {

      for (let j = 0; j < n; j++) {

        if (find(i, parent) != find(j, parent) && graf[i][j] < min) {

          min = graf[i][j];

          a = i;

          b = j;

        }

      }

    }

    union(a, b, parent);

    console.log(`Muchia ${nrMuchii} : (${a},${b}) cost: ${min}`);

    mincost += min;

    nrMuchii++;

  }

  console.log(`Minimum cost = ${mincost}`);

};

*// ---------------------------------------------------------------- Prim*

const isValidEdge = (i, j, varfuriParcurse) => {

  if (i == j) return false;

  if (varfuriParcurse[i] == false && varfuriParcurse[j] == false) return false;

  else if (varfuriParcurse[i] == true && varfuriParcurse[j] == true) return false;

  return true;

};

const prim = (graf) => {

  console.log(`%cAlgoritmul lui Prim`, 'color: #E05297');

  let varfuriParcurse = Array(n).fill(false);

  varfuriParcurse[0] = true;

  let nrMuchii = 0,

    mincost = 0;

  while (nrMuchii < n - 1) {

    let min = Infinity,

      a = -1,

      b = -1;

    for (let i = 0; i < n; i++) {

      for (let j = 0; j < n; j++) {

        if (graf[i][j] < min) {

          if (isValidEdge(i, j, varfuriParcurse)) {

            min = graf[i][j];

            a = i;

            b = j;

          }

        }

      }

    }

    if (a != -1 && b != -1) {

      console.log(`Muchia ${nrMuchii} : ( ${a}, ${b} )  cost: ${min} `);

      mincost += min;

      varfuriParcurse[b] = varfuriParcurse[a] = true;

      nrMuchii++;

    }

  }

  console.log('Minimum cost = ' + mincost);

};

*// ----------------------------------------------------------------*

*// unGraf();*

*// citireGraf();*

n = 10;

*// grafDens();*

grafRar();

*// console.log(graf);*

const startTime = window.performance.now();

kruskal(graf);

*// prim(graf);*

const endTime = window.performance.now();

console.log(

  `%cTimpul de rulare: ${endTime - startTime} ms = ${(endTime - startTime) / 1000} s`,

  'color: #E05297');