**MINISTERUL EDUCAŢIEI**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare Informatică și Microelectronică**

**Tehnologia Informației**

**Raport**

**Disciplina:**Analiza și proiectarea algoritmilor

**Lucrarea de laborator nr. 3**

**Tema:** *“*Algoritmi greedy.*”*

|  |  |  |
| --- | --- | --- |
| **Student:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Raevschi Grigore TI-231** |
| **Coordonator:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Asistent univ. Coșer Cătălin** |
|  |  |  |

Chișinău 2024

Cuprins

[**Scopul lucrării:** 3](#_Toc181828929)

[**Sarcina de bază:** 3](#_Toc181828930)

[**1** **Introducere:** 3](#_Toc181828931)

[**1.1** **Algoritm Greedy:** 4](#_Toc181828932)

[**1.2** **Principiul metodei Greedy:** 4](#_Toc181828933)

[**2** **Arbori parțiali de cost minim: Kruskal, Prim** 4](#_Toc181828934)

[**3** **Algoritmului Kruskal:** 5](#_Toc181828935)

[**3.1** **Complexitate** 5](#_Toc181828936)

[**3.2** **Avantaje și Dezavantaje** 5](#_Toc181828937)

[**4** **Algoritmul lui Prim** 6](#_Toc181828938)

[**4.1** **Complexitate** 6](#_Toc181828939)

[**4.2** **Avantaje și Dezavantaje** 6](#_Toc181828940)

[**5** **Analiza empirică** 7](#_Toc181828941)

[**6** **Concluzie** 8](#_Toc181828942)

[**7** **Bibliografie** 9](#_Toc181828943)

[**ANEXA A: Pseudocod pentru Algoritmul Kruskal** 10](#_Toc181828944)

[**ANEXA B: Pseudocod pentru Algoritmul lui Prim** 11](#_Toc181828945)

# **Scopul lucrării:**

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

# **Sarcina de bază:**

1. De studiat tehnica greedy de proiectare a algoritmilor.
2. De implementat într-un limbaj de programare algoritmii Kruskal, Prim.
3. De făcut analiza empirică a algoritmilor Kruskal şi Prim.
4. De alcătuit un raport.

# **Introducere:**

Metoda de programare Greedy se aplică problemelor de optimizare. Aceasta metoda constă în faptul că se construieşte soluția optimă pas cu pas, la fiecare pas fiind selectat în soluție elementul care pare „cel mai bun/cel mai optim” la momentul respectiv, în speranța că această alegere locală va conduce la optimul global.

Algoritmii Greedy sunt foarte eficienți, dar nu conduc în mod necesar la o soluție optimă. Şi nici nu este posibilă formularea unui criteriu general conform căruia să putem stabili exact dacă metoda Greedy rezolvă sau nu o anumită problemă de optimizare. Din acest motiv, orice algoritm Greedy trebuie însoțit de o demonstrație a corectitudinii sale . Demonstrația faptului că o anumită problemă are proprietatea alegerii Greedy se face de obicei prin inducție matematică.

Metoda Greedy se aplică problemelor pentru care se dă o mulţime A cu n elemente şi pentru care trebuie determinată o submulţime a sa, S cu m elemente, care îndeplinesc anumite condiţii, numite si condiții de optime.

Principiu (strategia) Greedy :

* adaugă succesiv la rezultat elementul care realizează optimul local
* decizie luată pe parcurs nu se mai modifică ulterior

## **Algoritm Greedy:**

GreedyAlgorithm(input):

sort(input) // Sortează elementele în funcție de o anumită metrică

solution = [] // Soluția finală

for each element in input:

if isFeasible(element, solution): // Verifică dacă elementul poate fi adăugat

solution.add(element) // Adaugă elementul în soluție

return solution

## **Principiul metodei Greedy:**

Algoritmii greedy (sau „algoritmi lacomi”) sunt o familie de algoritmi care fac alegeri optime locale în scopul de a găsi o soluție global optimă. Aceștia sunt utilizați în diverse probleme, cum ar fi găsirea arborelui de acoperire minim, problemele de rucsac, programarea intervalelor și multe altele.

# **Arbori parțiali de cost minim: Kruskal, Prim**

Atât în algoritmul lui Prim cât și în algoritmul lui Kruskal putem identifica elementele generale

specifice metodei Greedy astfel:

* există o mulțime inițială 𝐴𝐴 din care se aleg elementele care vor compune soluția (𝐴𝐴 este
* mulțimea muchiilor grafului);
* fiecărui element al mulțimii 𝐴𝐴 îi este asociată o valoare numerică (ponderea muchiei);
* mulțimea inițială se sortează crescător, în ordinea ponderilor asociate muchiilor;
* din mulțimea 𝐴𝐴 se aleg, în ordine, primele 𝑚𝑚 − 1 elemente care nu formează cicluri
* (criteriul de alegere).

# **Algoritmului Kruskal:**

În acest raport, vom explora principiile de bază ale algoritmului Kruskal, pașii săi esențiali și aplicabilitatea sa în diverse domenii, cum ar fi rețelele de calculatoare, planificarea infrastructurii și optimizarea resurselor. De asemenea, vom analiza complexitatea temporală a algoritmului și vom compara eficiența acestuia cu alte metode de calculare a arborelui de acoperire minim, precum algoritmul Prim. Această evaluare va evidenția relevanța algoritmului Kruskal în rezolvarea problemelor practice și teoretice din domeniul informaticii și al ingineriei.

Algoritmul Kruskal este utilizat pentru a determina arborele de acoperire minim (MST - Minimum Spanning Tree) al unui graf neorientat. Un arbore de acoperire minim este un subgraf care conține toate vârfurile originale și are suma minimă a greutăților muchiilor. Algoritmul funcționează pe baza principiului de selecție a celor mai ușoare muchii, asigurându-se că nu se formează cicluri.

## **Complexitate**

* **Complexitatea de timp**:
  + Sortarea muchiilor are o complexitate de unde EE este numărul de muchii. Operațiile de unire și căutare folosind Union-Find au o complexitate aproape constantă, , cu αα fiind funcția inversă a lui Ackermann.
  + Prin urmare, complexitatea totală a algoritmului Kruskal este
* **Complexitatea de spațiu**:
  + Algoritmul necesită spațiu pentru stocarea muchiilor și a structurii de date Union-Find, deci complexitatea de spațiu este

## **Avantaje și Dezavantaje**

**Avantaje:**

* **Simplicity**: Algoritmul este intuitiv și ușor de implementat.
* **Eficiență**: Este adesea mai eficient decât alte metode pentru grafuri sparse.

**Dezavantaje:**

* **Sortarea**: Procesul de sortare a muchiilor poate fi costisitor pentru grafuri cu un număr mare de muchii.
* **Grafuri Dense**: În cazul grafurilor dense, alte algoritmi, cum ar fi Prim, pot avea performanțe mai bune.

# **Algoritmul lui Prim**

Algoritmul lui Prim este un alt algoritm fundamental folosit pentru a găsi arborele de acoperire minim (MST - Minimum Spanning Tree) al unui graf neorientat. Spre deosebire de algoritmul Kruskal, care selectează muchii în funcție de greutate, Prim construiește arborele de acoperire minim începând cu un nod și adăugând treptat cele mai ușoare muchii care extind arborele existent.

## **Complexitate**

* **Complexitatea de timp**:
  + Algoritmul poate fi implementat eficient utilizând o structură de date precum un **min-heap** (tas minim). În acest caz, complexitatea de timp devine unde EE este numărul de muchii și VV este numărul de vârfuri.
  + O altă implementare, utilizând matricea de adiacență, are o complexitate de .
* **Complexitatea de spațiu**:
  + Algoritmul necesită spațiu pentru a stoca graful și pentru structura de date utilizată, deci complexitatea de spațiu este .

## **Avantaje și Dezavantaje**

**Avantaje:**

* **Eficiență**: Algoritmul lui Prim este adesea mai eficient pentru grafuri dense, unde numărul de muchii este aproape de maximul posibil.
* **Ușurința de Implementare**: Algoritmul este intuitiv și relativ simplu de implementat, în special cu ajutorul structurilor de date moderne.

**Dezavantaje:**

* **Dependent de Graf**: Performanța sa poate scădea în cazul grafurilor sparse, unde Kruskal ar putea fi mai eficient.
* **Necesită Structuri de Date Avansate**: Utilizarea eficientă a min-heap-ului sau a altor structuri de date poate complica implementarea.

# **Analiza empirică**

## **Cazul favorabil**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algoritmul** | **Nr.de virfuri** | **Iteratii** | **Timpul** |
| Kruskal | 1000 | 3124 | 1,678856 |
| Kruskal | 100 | 186 | 0,007530 |
| Kruskal | 10 | 11 | 0,000663 |
| Prim | 1000 | 8717 | 0,488688 |
| Prim | 100 | 1000 | 0,013207 |
| Prim | 10 | 45 | 0,000592 |

**Concluzie**

# **Bibliografie**

1. „Algoritmi pentru determinare a arborelui parţial de cost minim.”

<https://www.mateinfo.net/muscalua/parcarbori.pdf>

1. „Arbori parțiali de cost minim: Kruskal, Prim.”

<https://www.studocu.com/ro/document/academia-de-studii-economice-din-bucuresti/algoritmi-si-tehnici-de-programare/s13-arbori-partiali-de-cost-minim-kruskal-si-prim/26218183>

1. „Curs 13 - Tehnici de programare.”

<https://www.cs.ubbcluj.ro/~istvanc/fp/curs/Curs13%20-%20Greedy,%20Programare%20dinamica.pdf>

# **ANEXA A: Pseudocod pentru Algoritmul Kruskal**

Kruskal(G):

// G este graful, reprezentat printr-o listă de muchii

MST = setul de muchii ale arborelui de acoperire minim

edges = sort(G.edges, key=weight) // Sortează muchiile în funcție de greutate

parent = [] // Array pentru părinți

rank = [] // Array pentru ranguri

for each vertex v in G:

parent[v] = v // Fiecare nod este inițial un set separat

rank[v] = 0 // Rang inițial este 0

for each edge (u, v) in edges:

if Find(parent, u) != Find(parent, v): // Verifică dacă u și v sunt în același set

MST.addEdge(u, v) // Adaugă muchia în MST

Union(parent, rank, u, v) // Unifică seturile

return MST

Find(parent, v):

if parent[v] != v:

parent[v] = Find(parent, parent[v]) // Căutare cu compresie

return parent[v]

Union(parent, rank, u, v):

rootU = Find(parent, u)

rootV = Find(parent, v)

if rootU != rootV:

if rank[rootU] > rank[rootV]:

parent[rootV] = rootU

else if rank[rootU] < rank[rootV]:

parent[rootU] = rootV

else:

parent[rootV] = rootU

rank[rootU] += 1 // Crește rangul

# **ANEXA B: Pseudocod pentru Algoritmul lui Prim**

Prim(G, start):

// G este graful, reprezentat printr-o matrice de adiacență

// start este nodul de început

V = numărul de vârfuri în G

MST = setul de muchii ale arborelui de acoperire minim

cost = 0

// Inițializare

for each vertex v in G:

key[v] = ∞ // Costul minim pentru a ajunge la v

parent[v] = NULL // Părintele lui v în MST

key[start] = 0 // Costul pentru nodul de start este 0

minHeap = MinHeap() // Heap minim pentru a selecta nodul cu costul minim

minHeap.insert(start)

while not minHeap.isEmpty():

u = minHeap.extractMin() // Extrage nodul cu costul minim

// Adaugă muchia (parent[u], u) în MST

if parent[u] is not NULL:

MST.addEdge(parent[u], u)

cost += weight(parent[u], u) // Adaugă greutatea muchiei la cost total

// Actualizează costurile pentru nodurile adiacente

for each vertex v adjacent to u:

if v is not in MST and weight(u, v) < key[v]:

key[v] = weight(u, v) // Actualizează costul

parent[v] = u // Actualizează părintele

minHeap.decreaseKey(v, key[v]) // Actualizează heap-ul

return MST, cost