Tema 2 – Analiza Algoritmilor

Pisica Alin-Georgian, 324CC

Problema 1

Definitie: Arborele de acoperire al grafului $G = (V, E)$ este un subgraf $G' = (V, E')$ care contine toate varfurile grafului G si o submultime minima de muchii $E' \subseteq E$ cu proprietatea ca uneste toate varfurile si nu contine cicluri.

Arborele minim de acoperire se determina folosind Algoritmul lui Kruskal sau Algoritmul Prim. Spre deosebire de algoritmul Kruskal, in algoritmul Prim nu se construiesc mai multi subarbori care se unesc si in final ajung sa formeze un arbore minim de acoperire, ci exista un arbore principal la care se adauga muchia cu cel mai mic cost care uneste un nod din arbore cu un nod din afara sa.

Ipoteza pentru rezolvarea problemei: arborele este conex. Daca exista un nod izolat crearea arborelui minim de acoperire nu este posibila.

In rezolvarea problemei K-Subtree vom folosi algoritmul Kruskal, obtinand arborele minim de acoperire al carui cost il vom compara cu K.

a) Algoritm determinist
Vom avea ca baza laboratorul 10 din cadrul materiei “Proiectarea Algoritmilor” (anul 2, semestrul 2) - https://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-10 . Vom nota MuchiiAMA muchiile arborelui minim de acoperire.

K-Subtree(G(V, E))

```
MuchiiAMA = ∅
Pentru v in V
        MakeTree(v)      // Fiecare nod reprezinta in mod initial un arbore diferit
SorteazaMuchiiCrescatorDupaCost(E)
Pentru (u, v) in E
        // Capetele muchiei trebuie sa faca parte din arbori disjuncti
        Daca (FindTree(u) != FindTree(v))
                MuchiiAMA = MuchiiAMA + {(u, v)}
                UnesteArborii(u, v) // conectam arborii corespunzatori nodurilor
Cost = 0

Pentru e in E // Parcurgem toate muchiile
        Cost = Cost + Cost(e) // Adaugam costul muchiei e
Daca Cost ≤ k
        Return 1
Return 0
```

b) Algoritm nedeterminist
Vom transforma algoritmul de la punctul a) intr-un algoritm nedeterminist

K-Subtree(G(V, E))

```
MuchiiAMA = ∅
Pentru v in V
        MakeTree(v)      // Fiecare nod reprezinta in mod initial un arbore diferit
SorteazaMuchiiCrescatorDupaCost(E)
(u, v) = Choice(E) // Folosim choice pentru a itera muchiile in paralel
Daca (FindTree(u) != FindTree(v))
        MuchiiAMA = MuchiiAMA + {(u, v)}
        UnesteArborii(u, v) // conectam arborii corespunzatori nodurilor
Cost = 0

Pentru e in E // Parcurgem toate muchiile
        Cost = Cost + Cost(e) // Adaugam costul muchiei e
Daca Cost ≤ k
        Return 1
Return 0
```

c) Complexitatea algoritmilor de mai sus *(Este necesara explicarea, si deducerea ei, deci nu demonstratia acesteia)*

Pentru **algoritmul determinist**, complexitatea este data atat de sortarea muchiilor in ordine crescatoare (care poate fi alterata in functie de metoda de sortare folosita) cat si de iterarea muchiilor (care necesita minim E parcurgeri).

Consideram V multimea nodurilor din graf si E multimea muchiilor din graf. E este cel mult V^2 iar $\log V^2 = 2 \log V \ \ este \ O(\log V)$. Fiecare nod izolat este o componenta separate in arborele minim de acoperire. Daca ignoram nodurile izolate obtinem ca $V \leq 2E$.

Initializarea se face intr-un timp O(V). Sortarea, intr-un mod optim, o vom reduce la o complexitate de O(E logE). In bucla principala se vor executa card E operatii, care presupun doua operatii de gasire a arborilor bazate pe nodul de verificare si, in maxim V cazuri si o reuniune a acestor arbori, ce se reduce la un timp de O(E logE).

Astfel, putem reduce complexitatea la O(E logV) sau, echivalent, O(E logE), pe baza explicatiilor din al doilea paragraf.

Pentru **algoritmul nedeterminist** complexitatea este redusa de iterarea muchiilor, ce se face simultan. Astfel este simulat un process de multi-threading in constructia arborelui minim de acoperire. Functia choice() va itera prin toate muchiile din multimea E, in paralel, arborele de acoperire construindu-se intr-un timp minim. Astfel, complexitatea se reduce la complexitatea initializarii – O(V), complexitatea sortarii – O(E log E) si numarul de muchii finale (care in cel mai rau caz va fi V – 1), pentru

fiecare muchie adaugata executandu-se reuniuni a 2 arbori. Astfel, complexitatea se va reduce la O(E log E).

Problema 2

Definitie Definim problema D-LongestRoad: Pornind de la o configuratie valida a asezarilor pe harta, verificati daca se poate construi un drum de lungime maxima D care sa treaca prin toate asezarile pe harta, fara a forma cicluri.

a) Algoritm nedeterminist care stabileste daca se poate genera o configuratie valida a asezarilor

O prima intentie de rezolvare a algoritmului ar fi de a genera toate aranjamentele posibile ale asezarilor si de a verifica daca ce am obtinut la momentul respective este corect. De la o prima impresie putem observa si concluziona ca algoritmul ar fi unul inefficient.

Conform hartii Catan oferite in cerinta, putem observa ca fiecare nod are cel mult 3 vecini (3 daca este in interiorul hartii, 2 daca este la marginea acesteia). Totodata, pentru ca o asezare sa fie la o distanta de cel putin 2 muchii se reduce la situarea acesteia in asa fel incat in nodul vecin sa nu fie o alta asezare. Astfel, concluzionam ca intre oricare 2 asezari trebuie sa existe un nod liber. Putem astfel reduce problema la o solutie relative simpla:

i) Alegem nodul de start
ii) Amplasam o asezare
iii) Excludem toti vecinii asezarii plasate
iv) Repetam de la pasul ii pana cand toate asezarile sunt amplasate sau pana cand ramanem fara noduri libere

```
ExistaConfiguratieValida( G(V, E), A ):
    AsezareCurenta = 1
    Cat_timp AsezareCurenta < A si V != ∅:
        NodCurent = choice(V)
        NodCurent.Value = AsezareCurenta
        V = V – { NodCurent }
        Pentru vecin in Vecinii(NodCurent):
            V = V – { vecin }
        AsezareCurenta += 1
    Daca AsezareCurenta > A
        Success
    Fail
```

Analiza Complexitatii

In analiza complexitatii algoritmului anterior, putem observa ca intreaga actiune ce determina complexitatea se executa in cadrul instructiunii Cat_timp. Aceasta este repetata de un numar maxim A ori. In cadrul acesteia, operatiile se pot reduce la un timp constant O(1) (chiar si for-ul, tinand cont ca

exista maxim 3 vecini pentru fiecare nod => Worst case ar fi O(3), care este de fapt O(1)). Astfel, complexitatea algoritmului se reduce in final la O(A).

b) Algoritm nedeterminist care rezolva problema D-LongestRoad

Problema D-LongestRoad se rezuma la a gasi cel mai scurt drum intre toate nodurile si a verifica costul drumul mai mic decat D. Tinand cont de harta jocului de Catan, vom considera ca pentru fiecare muchie costul echivalent este 1.

D-LongestRoad(G(V,E), D):

```
NodStart = V[0]
V.Cost = 0
Cost = 0
Cat_timp (NodStart != null):
        Cost = Cost + 1
        v = choice(Vecinii(NodStart))
        v.Cost = Cost
        NodStart = v
MaxCost = 0
For v in V:
        Daca v.Cost > MaxCost:
                MaxCost = v
Daca MaxCost < D
        Success
Fail
```

Explicarea algoritmului:

Vom alege nodul de start cu cost 0, si vom itera graf-ul. Multumita functiei choice, iterarea se va face in paralel pe nivel si secvential in adancime. Astfel, complexitatea structurii Cat_timp se va reduce la numarul de nivele ale grafului (In cazul jocului de Catan maxim 5, in cazul unui graf la mod general maxim V). In urma umplerii tuturor nodurilor cu costul aferent ajungerii la el, vom itera nodurile si vom pastra cel mai mare cost (este cel necesar ajungerii de la inceput la sfarsit). Daca este mai mic decat D, algoritmul a fost executat cu succes si a gasit o varianta corecta.

Complexitatea algoritmului

Complexitatea functiei Cat_timp va fi data de O(V) – cum am mentionat si anterior, worst case in cadrul jocului de Catan ar fi 5, iar pe caz general V (presupunand ca nodurile sunt legate sub forma unui singur sir – fiecare parinte are doar un copil). Iterarea prin lista de noduri va insemna tot o complexitate O(V), deoarece in cadrul ei se executa operatii in timp constant. In final complexitatea se reduce la executia a doua instructiuni timp constant de iteratie pe fiecare element intr-o complexitate finala de O(V).