LUCRAREA DE LABORATOR NR.4

Tema: Algoritmi greedy

Scopul lucrării:

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

Note de curs:

1. Tehnica greedy

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;
- 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.

2. Arbori de acoperire cu 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 w (sau o *lungime* nenegativă). Problema este să găsim o submulţime $A \subseteq M$, 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 de acoperire minim* al grafului G (*minimal spanning tree* (MST)). Un graf poate avea mai mulţi arbori de acoperire de cost minim. Vom prezenta doi algoritmi greedy care determină arborele de acoperire minim al unui graf. În terminologia metodei greedy, vom spune că o mulţime de muchii este o *soluţie*, dacă constituie un arbore de acoperire 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.

2.1. 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 două vârfuri *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:

MST - Kruskal(G, w)
1: A$\leftarrow\varnothing$
2: **for** fiecare vârf $v \in V[G]$ **do**
3: ***Make-Set*** (v)
4: sortează muchiile din M crescător în funcţie de cost
5: **for** fiecare muchie (u, v) $\in M$ **do**
6: **if** ***Find-Set*** (u) $\neq$ ***Find-Set*** (v)
7: **then** $A \leftarrow A \cup \{\{u, v\}\}$
8: ***Union*** (u, v)
9: **return** A

Modul de lucru al algoritmului Kruskal:

Liniile 1-3 iniţializează mulţimea A cu mulţimea vidă şi creează $|V|$ arbori, unul pentru fiecare vârf. Muchiile din M sunt ordonate crescător după cost, în linia 4. Bucla **for** din liniile 5-8 verifică, pentru fiecare muchie (u, v), dacă punctele terminale u şi v aparţin aceluiaşi arbore. Dacă fac parte din acelaşi arbore, atunci muchia (u, v) nu poate fi adăugată la pădure fără a se forma un ciclu şi ea este respinsă. Altfel, cele două vârfuri aparţin unor arbori diferiţi, şi muchia (u, v) este adăugată la A în linia 7, vârfurile din cei doi arbori fiind reunite în linia 8.

2.2. 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 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 $U = V$.

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, $key[v] = \infty$ dacă nu există o astfel de muchie. Câmpul $\pi[v]$ reţine „părintele" lui v din arbore. $Adj[u]$ este lista de adiacenţă cu vârful u.

MST - Prim(G, w, r)
1: $Q \leftarrow V[G]$
2: **for** fiecare vârf $u \in Q$
3: **do** $key[u] \leftarrow \infty$
4: $key[r] \leftarrow 0$
5: $\pi[r] \leftarrow$ NIL
6: **while** $Q \neq \varnothing$
7: **do** $u \leftarrow$ ***Extract-Min***(Q)
8: **for** fiecare vârf $v \in Adj[u]$
9: **do if** $v \in Q$ **and** $w(u, v) < key[v]$
10: **then** $\pi[v] \leftarrow u$
11: $key[v] \leftarrow w(u, v)$

Modul de lucru al algoritmului Prim:

În liniile 1-4 se iniţializează coada de prioritate Q, astfel încât aceasta să conţină toate vârfurile şi se iniţializează câmpul *key* al fiecărui vârf cu ∞, excepţie fîcând rădăcina r, al cărei câmp *key* este iniţializat cu 0. În linia 5 se iniţializează $\pi[r]$ cu NIL, deoarece rădăcina r nu are nici un părinte. Pe parcursul execuţiei algoritmului, mulţimea $V - Q$ conţine vârfurile arborelui curent. În

linia 7 este identificat un vârf $u \in Q$ incident unei muchii uşoare care traversează tăietura ($V - Q$, Q) (cu excepţia primei iteraţii, în care $u = v$ datorită liniei 4). Eliminarea lui u din mulţimea Q îl adaugă pe acesta mulţimii $V - Q$ a vârfurilor din arbore. În liniile 8-11 se actualizează câmpurile *key* şi π ale fiecărui vârf v adiacent lui u, dar care nu se află în arbore. Actualizarea respectă condiţiile $key[v] = w(v, \pi[v])$, şi $(v, \pi[v])$ să fie o muchie uşoară care îl uneşte pe v cu vârf din arbore.

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.

Întrebări de control:

1. Descrieţi metoda greedy.
2. Când se aplică algoritmul Kruskal şi când algoritmul Prim ?
3. Cum pot fi îmbunătăţite performanţele algoritmului Dijkstra?
4. Ce tip de date este comod de folosit la elaborarea programului al unui algoritm de tip greedy?
5. Care sunt avantajele şi dezavantajele algoritmilor Prim si Kruskal?