Proiectarea Algoritmilor 2011-2012

Laborator 6

Parcurgerea Grafurilor. Sortare Topologică

Cuprins

1 Obiective laborator

- Înțelegerea conceptului de graf și a modurilor de parcurgere aferente
- Utilitatea și aplicabilitatea sorțarii topologice

2 Importanţă – aplicaţii practice

Grafurile sunt utile pentru a modela diverse probleme și se regăsesc implementați în multiple aplicații practice:

- Rețele de calculatoare (ex: stabilirea unei topologii fără bucle)
- Pagini Web (ex: Google PageRank [1])
- Rețele sociale (ex: calcul centralitate [2])
- Hărți cu drumuri (ex: drum minim)
- Modelare grafica (ex: prefuse [3], graph-cut [4])

3 Descrierea problemei și a rezolvărilor

Graful poate fi modelat drept o pereche de mulțimi G = (V, E). Mulțimea V conține nodurile grafului (vertices), iar mulțimea E conține muchiile (edges), fiecare muchie stabilind o relație de vecinătate intre doua noduri. O mare varietate de probleme se modelează folosind grafuri, iar rezolvarea acestora presupune explorarea spațiului. O parcurgere își propune să ia în discuție fiecare nod al grafului, exact o singura data, pornind de la un nod ales, numit în continuare nod sursa.

Reprezentarea în memorie a grafurilor se face, de obicei, cu liste de adiacenta sau cu matrice de adiacenta. Se pot folosi însă și alte structuri de date, de exemplu un map de perechi <<sursa,destinatie>,cost> .

Pe parcursul rulării algoritmilor de parcurgere, un nod poate avea 3 culori:

- Alb = nedescoperit
- Gri = a fost descoperit și este în curs de procesare
- Negru = a fost procesat

Se poate face o analogie cu o pată neagră care se extinde pe un spațiu alb. Nodurile gri se afla pe frontiera petei negre.

Algoritmii de parcurgere pot fi caracterizați prin completitudine și optimalitate. Un algoritm de explorare complet va descoperi întotdeauna o soluție, dacă problema accepta soluție. Un algoritm de explorare optimal va descoperi soluția optima a problemei din perspectiva numărului de pași care trebuie efectuați.

3.1 Parcurgerea în lățime - BFS

Parcurgerea în lățime (Breadth-first Search - BFS) este un algoritm de căutare în graf, în care, atunci când se ajunge intr-un nod oarecare v, nevizitat, se vizitează toate nodurile nevizitate adiacente lui v, apoi toate vârfurile nevizitate adiacente vârfurilor adiacente lui v, etc.

Atenție! BFS depinde de nodul de start. Plecând dintr-un nod se va parcurge doar componenta conexa din care acesta face parte. Pentru grafuri cu mai multe componente conexe se vor obține mai mulți arbori de acoperire.

In urma aplicării algoritmului BFS asupra fiecărei componente conexe a grafului, se obține un arbore de acoperire (prin eliminarea muchiilor pe care nu le folosim la parcurgere). Pentru a putea reconstitui acest arbore, se păstrează pentru fiecare nod dat identitatea părintelui sau. In cazul în care nu exista o funcție de cost asociata muchiilor, BFS va determina și drumurile minime de la rădăcina la oricare nod.

Pentru implementarea BFS se folosește o coadă. In momentul adăugării în coada, un nod trebuie colorat gri (a fost descoperit și urmează să fie prelucrat).

Algoritmul de explorare BFS este complet și optimal.

Algoritm:

```
BFS(s, G) {
	foreach (u ∈ V) {
		p(u) = null; // initializari
		dist(s,u) = inf;
		c(u) = alb;
	}
	dist(s) = 0; // distanta pana la sursa este 0
	c(s) = gri; //incepem prelucrarea nodului, deci culoarea devine gri
	Q = (); //se foloseste o coada cu nodurile de prelucrat
	Q = Q + s; // adaugam sursa în coada
	while (!empty(Q)) { // cat timp mai am noduri de prelucrat
		u = top(Q); // se determina nodul din varful cozii
		foreach v ∈ succs(u) { // pentru toti vecinii
		if (c(v) = alb) {// nodul nu a fost gasit, nu e în coada
			// actualizam structura date
			dist(v) = dist(u) + 1;
			p(v) = u;
			c(v) = gri;
			Q = Q + v;
			} // close if
		} // close foreach
		c(u) = negru; //am terminat de prelucrat nodul curent
		Q = Q - u; //nodul este eliminat din coada
	} //close while
}
```

Complexitate:

- cu lista: O(|E|+|V|)
- cu matrice: O(|V|2)

3.2 Parcurgerea în adâncime – DFS

Parcurgerea în adâncime (Depth-First Search - DFS) pornește de la un nod dat (nod de start), care este marcat ca fiind în curs de procesare. Se alege primul vecin nevizitat al acestui nod, se marcheaza și acesta ca fiind în curs de procesare, apoi și pentru acest vecin se caută primul vecin nevizitat, și așa mai departe. In momentul în care nodul curent nu mai are vecini nevizitați, se marchează ca fiind deja procesat și se revine la nodul anterior. Pentru acest nod se caută primul vecin nevizitat. Algoritmul se repeta pana când toate nodurile grafului au fost procesate.

In urma aplicării algoritmului DFS asupra fiecărei componente conexe a grafului, se obține pentru fiecare dintre acestea cate un arbore de acoperire (prin eliminarea muchiilor pe care nu le folosim la parcurgere). Pentru a putea reconstitui acest arbore, păstram pentru fiecare nod dat identitatea părintelui sau.

Pentru fiecare nod se vor retine:

- timpul descoperirii
- timpul finalizării
- părintele
- culoarea

Algoritmul de explorare **DFS** nu este nici complet (în cazul unei căutări pe un subarbore infinit), nici optimal (nu găsește nodul cu adâncimea minima).

Spre deosebire de BFS, pentru implementarea DFS se folosește o stiva (abordare LIFO în loc de FIFO). Deși se poate face aceasta înlocuire în algoritmul de mai sus, de cele mai multe ori este mai intuitiva folosirea recusivității.

Algoritm:

```
DFS(G) {
	V = noduri(G)
	foreach (u ∈ V) {
		// initializare structura date
		c(u) = alb;
		p(u)=null;
	}
	timp = 0; // retine distanta de la radacina pana la nodul curent
	foreach (u ∈ V)
		if (c(u) = alb) explorare(u); // explorez nodul
}
explorare(u) {
	d(u) = timp++; // timpul de descoperire al nodului u
	c(u) = gri; // nod în curs de explorare
	foreach (v ∈ succes(u)) // incerc sa prelucrez vecinii
		if (c(v) = alb) { // dacă nu au fost prelucrati deja
			p(v) = u;
			explorare(v);
		}
	c(u) = negru; // am terminat de prelucrat nodul curent
	f(u) = timp++; // timpul de finalizare al nodului u
}
```

Complexitate:

- cu lista: O(|E|+|V|)
- cu matrice: O(|V|2)

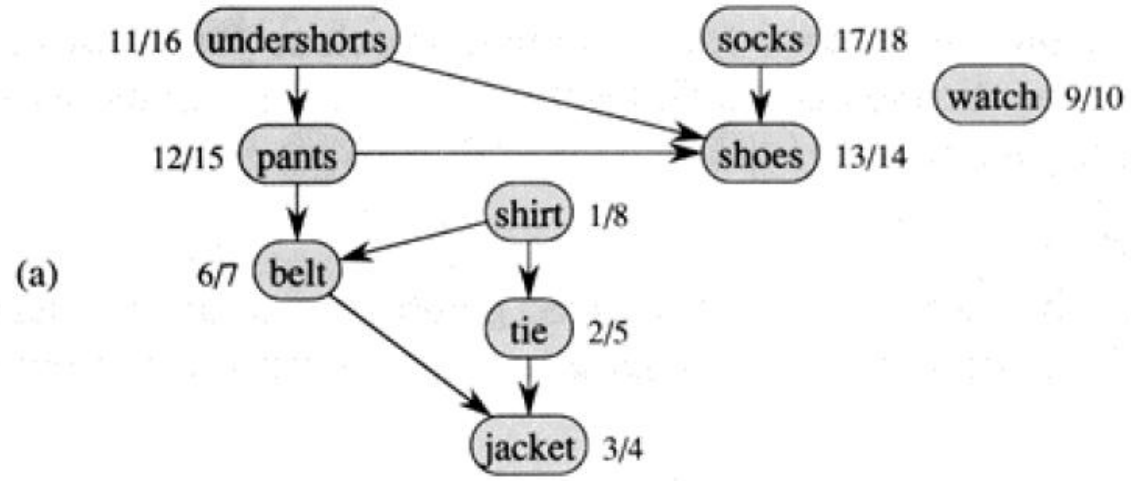
3.3 Sortarea Topologica

Dându-se un graf orientat aciclic, sortarea topologica realizează o aranjare liniara a nodurilor în funcție de muchiile dintre ele. Orientarea muchiilor corespunde unei relații de ordine de la nodul sursa către cel destinație. Astfel, dacă (u,v) este una dintre muchiile grafului, u trebuie sa apară înaintea lui v în înșiruire. Daca graful ar fi ciclic, nu ar putea exista o astfel de înșiruire (nu se poate stabili o ordine intre nodurile care alcătuiesc un ciclu).

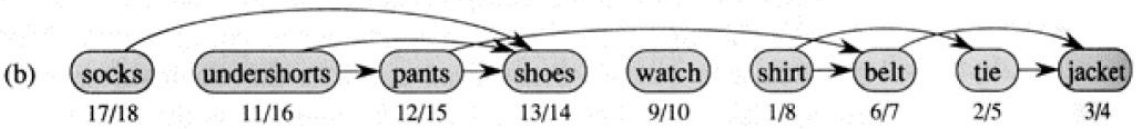
Sortarea topologica poate fi văzută și ca plasarea nodurilor de-a lungul unei linii orizontale astfel încât toate muchiile sa fie direcționate de la stânga la dreapta.

Exemplu:

Trudy (robotul nostru simpatic) își sortează topologic hainele înainte de a se îmbrăca.



(a) Fiecare muche (u,v) înseamnă ca obiectul de îmbrăcăminte u trebuie îmbrăcat înaintea obiectului de îmbrăcăminte v. Timpii de descoperire d(u) și de finalizare f(u) obținuți în urma parcurgerii DFS sunt notați lângă noduri.



(b) Același graf, sortat topologic. Nodurile lui sunt aranjate de la stânga la dreapta în ordinea descrescătoare a f(u). Observați ca toate muchiile sunt orientate de la stânga la dreapta. Acum Trudy se poate îmbrăca liniștit.

Algoritm:

Sunt doi algoritmi cunoscuți pentru sortarea topologică.

Algoritmul bazat pe DFS:

- parcurgere DFS pentru determinarea timpilor
- sortare descrescătoare în funcție de timpul de finalizare

Un alt algoritm este cel descris de Kahn:

```
TopSort(G) {
      V = noduri(G)
      L = vida;// lista care va contine elementele sortate
      // initializare S cu nodurile care nu au în-muchii
      foreach (u ∈ V)
            if (u nu are în-muchii)
                  S = S + u;
      }
      while (!empty(S)) { // cat timp mai am noduri de prelucrat
            u = random(S); // se scoate un nod din multimea S
            L = L + u;  // adaug U la lista finala
            foreach v ∈ succs(u) { // pentru toti vecinii
                  sterge u-v; // sterge muchia u-v
                  if (v nu are în-muchii )
                        S = S + v;  // adauga v la multimea S
            } // close foreach
      } //close while
```

```
      if ( G are muchii )
            print(eroare);  // graf ciclic
      else
            print(L); // ordinea topologica
}
```

Complexitate optima: O(|E|+|V|)

4 Concluzii și observatii

Grafurile sunt foarte importante pentru reprezentarea și rezolvarea unei multitudini de probleme.

Cele mai uzuale moduri de reprezentare a unui graf sunt:

- liste de adiacență
- matrice de adiacență

Cele doua moduri uzuale de parcurgere neinformata a unui graf sunt:

- BFS – parcurgere în lățime
- DFS – parcurgere în adâncime

Sortarea topologică este o modalitate de aranjare a nodurilor în funcție de muchiile dintre ele. In funcție de nodul de start al DFS, se pot obține sortări diferite, păstrând însă proprietățile generale ale sortării topologice.

5 Referințe

[1] http://en.wikipedia.org/wiki/PageRank

[2] http://en.wikipedia.org/wiki/Social_network#Social_network_analysis

[3] http://prefuse.org/

[4] http://classes.engr.oregonstate.edu/eecs/spring2008/cs419/Lectures/jun_graphcut.pdf

[5] http://en.wikipedia.org/wiki/Breadth-first_search

[6] http://en.wikipedia.org/wiki/Depth-first_search

[7] http://en.wikipedia.org/wiki/Topological_sorting

[8] Introducere în Algoritmi, T. Cormen s.a., pag 403-419

[9] http://ww3.algorithmdesign.net/handouts/DFS.pdf

[10] http://ww3.algorithmdesign.net/handouts/BFS.pdf