Structuri de Date și Algoritmi
Laboratorul 12: Grafuri orientate

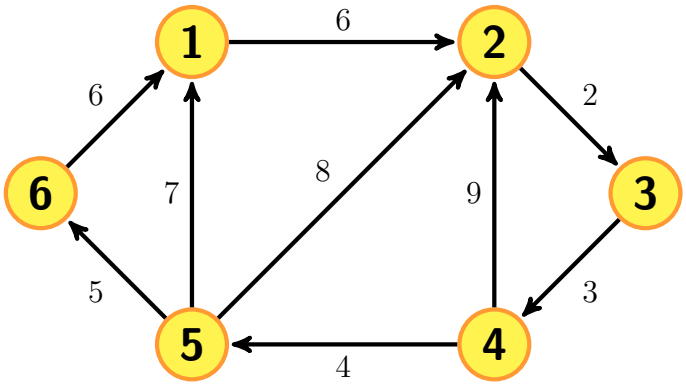
Mihai Nan

16 mai 2022

1. Introducere

Se numește *graf orientat* o pereche ordonată de mulțimi $\boldsymbol{G = (V, E)}$, unde $\boldsymbol{V}$ este o mulțime nevidă de elemente numite *vârfuri* sau *noduri*, iar $\boldsymbol{E}$ este o mulțime de perechi ordonate de elemente distincte din $\boldsymbol{V}$ numite *arce*.

Pentru un arc $\boldsymbol{e = (x, y) \in E}$, spunem că $\boldsymbol{x}$ și $\boldsymbol{y}$ sunt *adiacente*, $\boldsymbol{x}$ este *extremitatea inițială* a arcului $\boldsymbol{e}$, $\boldsymbol{y}$ este *extremitatea finală*, $\boldsymbol{x}$ și $\boldsymbol{y}$ sunt *incidente* cu $\boldsymbol{e}$, $\boldsymbol{x}$ este *predecesorul* lui $\boldsymbol{y}$, iar $\boldsymbol{y}$ este *succesorul* lui $\boldsymbol{x}$.



$$\boldsymbol{V = \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}}$$
$$\boldsymbol{E = \{(1,\ 2),\ (2,\ 3),\ (3,\ 4),\ (4,\ 2),\ (4,\ 5),\ (5,\ 1),\ (5,\ 2),\ (5,\ 6),\ (6,\ 1)\}}$$

Se numește *graf ponderat* o structură $\boldsymbol{(V,\ E,\ W)}$, unde $\boldsymbol{G = (V,\ E)}$ este graf și $\boldsymbol{W}$ este o funcție, numită *pondere*, care asociază fiecărei muchii a grafului un cost al parcurgerii ei.

Se numește *gradul interior* al unui vârf $\boldsymbol{x}$, notându-se cu $d_-(x)$, numărul arcelor de forma $\boldsymbol{(y,\ x)}$, $\boldsymbol{y \in V}$ și $\boldsymbol{(y,\ x) \in E}$.

Se numește *gradul exterior* al unui vârf $\boldsymbol{x}$, notându-se cu $d_+(x)$, numărul arcelor de forma $\boldsymbol{(x,\ y)}$, $\boldsymbol{y \in V}$ și $\boldsymbol{(x,\ y) \in E}$.

Definiție: Fie $\boldsymbol{G = (V,\ E)}$ un graf orientat. Se numește *lanț* o succesiune de arce $L = [e_1, e_2, ...e_k]$ cu proprietatea că oricare două arce consecutive e_i, e_{i+1} au o extremitate comună.

Observație: Într-un lanț, orientarea arcelor nu este importanta.

Definiție: Fie ***G = (V, E)*** un graf orientat. Se numește *drum* o succesiune de vârfuri $\mathbf{L} = [x_1, x_2, ...x_k]$ cu proprietatea (x_1, x_2), ... (x_{k-1}, x_k) sunt arce. Un drum în care toate vârfurile sunt distincte se numește *drum elementar*.

Definiție: Se numește *circuit* un drum în care primul vârf este identic cu ultimul, iar arcele nu se repetă. Un drum în care vârfurile nu se repetă, cu excepția primului și a ultimului se numește *circuit elementar*.

Definiție: Se numește *graf parțial* pentru un graf orientat ***G = (V, E)*** un graf ***G' = (V, E')*** cu proprietatea că ***E'*** $\in$ ***E***.

Definiție: Se numește *subgraf* a lui ***G*** un graf ***G' = (V', E')***, unde ***V'*** $\in$ ***V*** și ***E'*** conține toate arcele din ***E*** care au ambele extremități în ***V'***.

Definiție: Un graf orientat se numește *complet* dacă oricare două vârfuri distincte sunt adiacente.

Observație: Spre deosebire de grafurile neorientate, unde pentru o mulțime de vârfuri există un singur graf complet, in cazul grafurilor orientate, pentru o mulțime de vârfuri date putem avea mai multe grafuri complete.

Două vârfuri ***x*** și ***y*** sunt adiacente într-un graf orientat în oricare din situațiile: există arcul ***(x, y)***, arcul ***(y, x)*** sau arcele ***(x, y)*** și ***(y, x)***.

Sunt ***n(n-1)/2*** posibilități de a alege două vârfuri distincte.

Pentru fiecare dintre acestea, există 3 situații. În concluzie, în total sunt $3^{n(n-1)/2}$ grafuri orientate complete cu ***n*** vârfuri.

Definiție: Se numește *graf turneu* un graf cu proprietatea că între oricare două vârfuri distincte există exact un arc.

Proprietate: În orice graf turneu, există un drum elementar care conține toate vârfurile grafului.

Definiție: Un graf orientat ***G = (V, E)*** se numește *conex* dacă între oricare două vârfuri distincte există cel puțin un lanț.

2. Metode de reprezentare

Se numește *matricea drumurilor* asociată unui graf ***G*** cu ***n*** vârfuri o matrice pătratică de dimensiune ***n***, în care elementele sunt ***0*** sau ***1***, cu următoarea semnificație:

$$M_{i,j} = \begin{cases} 1 & \text{dacă există drum de la i la j în } \mathbf{G} \\ 0 & \text{dacă nu există drum de la i la j în } \mathbf{G} \end{cases} \quad (1)$$

Matricea drumurilor poate fi determinată plecând de la matricea de adiacență folosind algoritmul lui ***Roy-Warshall***.

3. Sortarea topologică

Dându-se un graf orientat aciclic, *sortarea topologică* realizează o aranjare liniară a nodurilor în funcție de arcele dintre ele. Orientarea arcelor corespunde unei relații de ordine de la nodul sursă către cel destinație. Astfel, dacă ***(u,v)*** este una dintre arcele grafului, ***u*** trebuie să

apară înaintea lui v în înșiruire. Dacă graful ar fi *ciclic*, nu ar putea exista o astfel de înșiruire (nu se poate stabili o ordine între nodurile care alcătuiesc un ciclu).

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

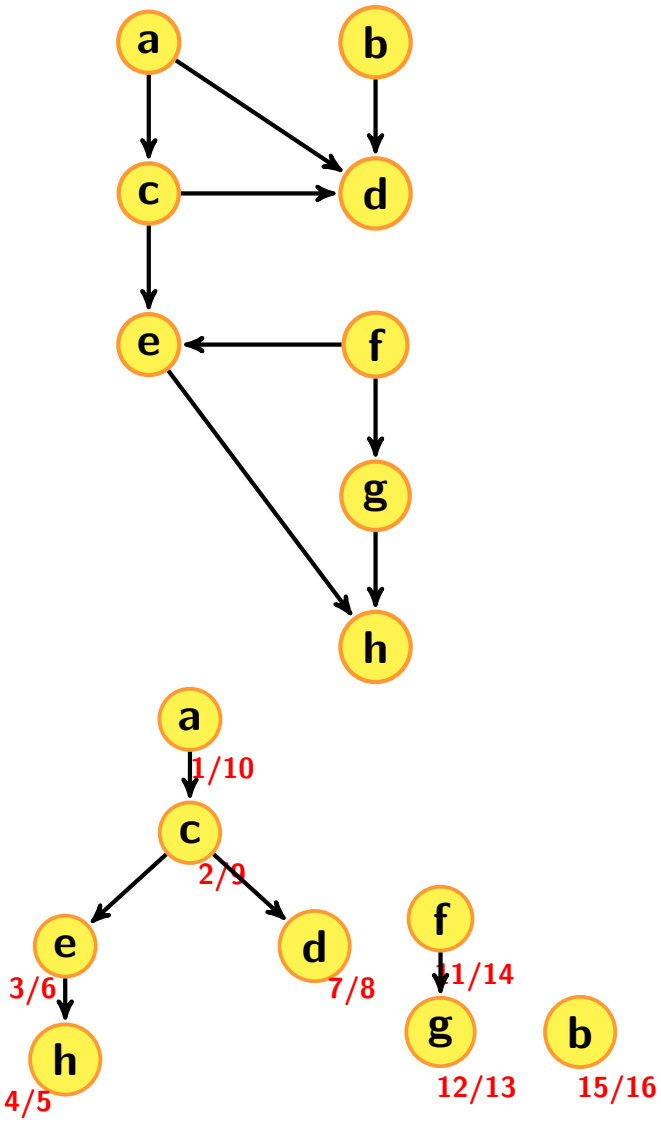
3.1 Algoritmul lui Kahn

Pseudocod

```
TopSort(G) {
    V = noduri(G)
    L = vida; // lista care va contine elementele sortate
    foreach (u ∈ V) {
        if (u nu are in-muchii)
          S = S + u; //noduri care nu au in-muchii
    }
    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 in-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
}
```

Observație

Putem implementa sortarea topologică utilizând o parcurgere DFS a grafului pentru determinarea timpilor de vizitare și finalizare. În acest caz, sortarea topologică a grafului se rezumă la sortarea descrescătoare a nodurilor în funcție de timpul de finalizare.



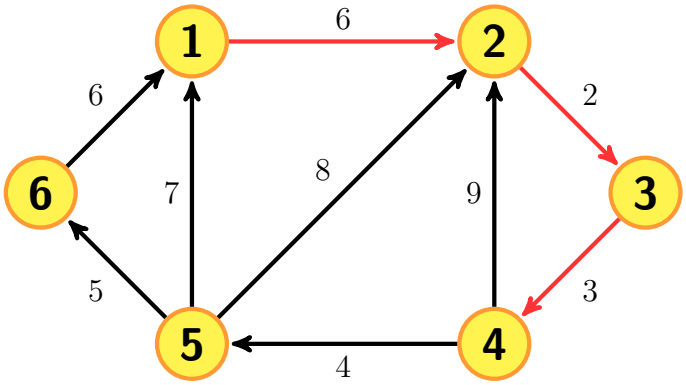
Rezultatul: b, f, g, a, c, d, e, h

4. Distanțe minime

Fiind dat un graf orientat ***G = (V, E)***, se consideră funcția w: E → W, numită funcție de cost, care asociază fiecărei muchii o valoare numerică. Domeniul funcției poate fi extins, pentru a include și perechile de noduri între care nu există muchie directă, caz în care valoarea este ∞.

Costul unui drum format din muchiile ***p12 p23 ... p(n-1)n***, având costurile $w_{12}, w_{23}, ..., w_{(n-1)n}$, este suma $w = w_{12} + w_{23} + ... + w_{(n-1)n}$.

Costul minim al drumului dintre două noduri este minimul dintre costurile drumurilor existente între cele două noduri.



w(1,4) = 6 + 2 + 3 = 11

Observație

Deși, în cele mai multe cazuri, costul este o funcție cu valori nenegative, exista situații în care un graf cu muchii de cost negativ are relevanță practică. O parte din algoritmi pot determina drumul corect de cost minim inclusiv pe astfel de grafuri.
Totuși, nu are sens căutarea drumului minim în cazurile în care graful conține cicluri de cost negativ – un drum minim ar avea lungimea infinită, întrucât costul său s-ar reduce la fiecare re-parcurgere a ciclului.

4.1 Algoritmul Floyd – Warshall

Algoritm prin care se calculează distanțele minime între oricare 2 noduri dintr-un graf (***drumuri optime multipunct–multipunct***).

Exemplu clasic de ***programare dinamica***.

Idee: la pasul ***k*** se calculează cel mai bun cost intre ***u*** și ***v***, folosind cel mai bun cost ***u..k*** și cel mai bun cost ***k..v*** calculat până în momentul respectiv.

Observație: Se aplică pentru grafuri ce nu conțin cicluri de cost negativ.

Pseudocod

```
FloydWarshall(G):
    n = |V|
    int d[n, n]
    foreach (i, j) in (1..n,1..n)
        d[i, j] = w[i,j] // costul muchiei, sau infinit
    for k = 1 to n
        foreach (i,j) in (1..n,1..n)
            d[i, j] = min(d[i, j], d[i, k] + d[k, j])
```

4.2 Algoritmul Bellman — Ford

Algoritmul Bellman Ford poate fi folosit și pentru grafuri ce conțin muchii de cost negativ, dar nu poate fi folosit pentru grafuri ce conțin cicluri de cost negativ (când căutarea unui drum minim nu are sens).

Cu ajutorul său putem afla dacă un graf conține cicluri de cost negativ. Algoritmul folosește același mecanism de relaxare ca Dijkstra, dar, spre deosebire de acesta, nu optimizează o soluție folosind un criteriu de optim local, ci parcurge fiecare muchie de un număr de ori egal cu numărul de noduri și încearcă să o relaxeze de fiecare dată, pentru a îmbunătăți distanța până la nodul destinație al muchiei curente.

Motivul pentru care se face acest lucru este că drumul minim dintre sursă și orice nod destinație poate să treacă prin maximum $|V|$ noduri (adică toate nodurile grafului), respectiv $|V|-1$ muchii; prin urmare, relaxarea tuturor muchiilor de $|V|-1$ ori este suficientă pentru a propaga până la toate nodurile informația despre distanța minimă de la sursă.

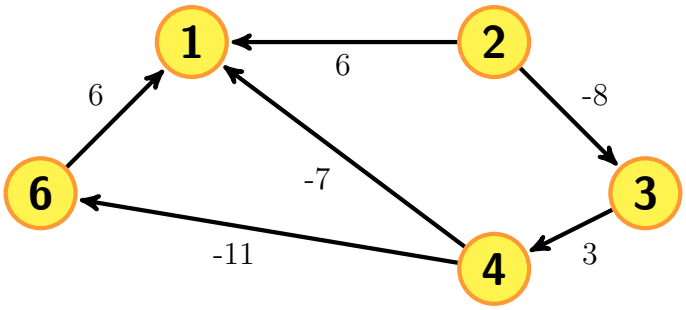
Dacă, la sfârșitul acestor $|E| * (|V|-1)$ relaxări, mai poate fi îmbunătățită o distanță, atunci graful are un ciclu de cost negativ și problema nu are soluție.

Pseudocod

```
BellmanFord(sursa):
    // initializari
    foreach nod in V // V = multimea nodurilor
        daca muchie[sursa, nod]
            d[nod] = w[sursa, nod]
            P[nod] = sursa
        altfel
            d[nod] = +∞
            P[nod] = null
    d[sursa] = 0
    p[sursa] = null

    // relaxari succesive
    // cum in initializare se face o relaxare (daca exista drum direct de
    la sursa la nod =>
    // d[nod] = w[sursa, nod]) mai sunt necesare |V-2| relaxari
    for i = 1 to |V|-2
        foreach (u, v) in E  // E = multimea muchiilor
            daca d[v] > d[u] + w(u,v)
                d[v] = d[u] + w(u,v)
                p[v] = u;

    // daca se mai pot relaxa muchii
    foreach (u, v) in E
        daca d[v] > d[u] + w(u,v)
            fail (''exista cicluri negativ'')
```



Observație

În cazul grafurilor ce au ponderi negative nu putem să transformăm graful în unul ce conține ponderi obținute prin adunarea în modul a ponderii minime pe care să aplicăm algoritmul lui Dijkstra, deoarece prin această transformare a costurilor se modifică drumurile minime (nu sunt conservate drumurile minime).

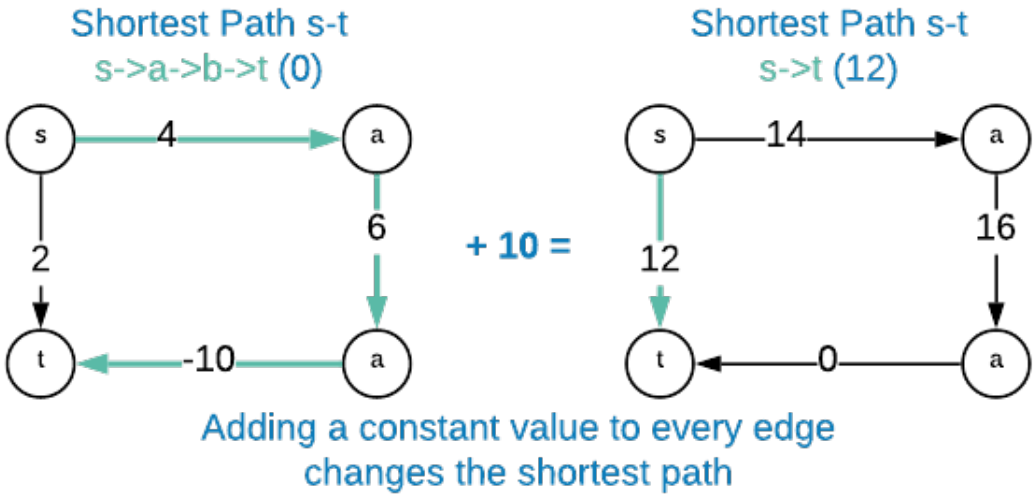


Figura 1: Exemplu de situație în care drumurile minime se modifică după transformarea grafului

Probleme propuse

Reprezentarea pe care o folosim pentru graful orientat în cadrul acestui laborator este următoarea:

```
typedef struct pair {
   int v, cost;
} Pair;

typedef Pair V;
```

```c
typedef struct list {
   V data;
   struct list *prev, *next;
}*List;

typedef struct graph {
   int V; // nr de noduri din graf
   int type; // 0 - neorientat ; 1 - orientat
   List *adjLists; // vectorul cu listele de adiacență
   int *visited; // vector pentru marcarea nodurilor vizitate
   int *start, *end;
}*Graph;

// Structură ce poate fi folosită dacă vreți să extrageți muchiile
typedef struct edge {
   int u, v, cost;
} Edge;
```

1. Implementați o funcție ce realizează sortarea topologică a vârfurilor unui graf orientat aciclic, reprezentat folosind liste de adiacență. Sortarea topologică este o operație de ordonare liniară a vârfurilor, astfel încât, dacă există un arc ***(i, j)***, atunci ***i*** apare înaintea lui ***j*** în această ordonare.

   ```c
   /*
   * Funcție care determină sortarea topologică pentru un graf orientat aciclic
   * Obs: Rezultatul este reținut în vectorul result care a fost deja alocat!
   */
   void topologicalSort(Graph graph, int *result);
   ```

2. Implementați o funcție care determină drumurile de cost minim de la x la toate celelalte noduri accesibile pentru un graf orientat ce poate conține ponderi negative, dar care nu conține un ciclu de cost negativ.

   ```c
   /*
   * Funcție care determină costurile drumurilor minime care pornesc din start
   * Obs: Rezultatul este reținut în vectorul distances care a fost deja alocată!
   */
   void BellmanFord(Graph graph, int start, int *distances);
   ```

3. Implementați o funcție care primește ca parametru un graf orientat și determină pentru orice pereche de noduri x și y lungimea minimă a drumului de la nodul x la nodul y. Prin lungimea unui drum înțelegem suma costurilor arcelor care-l alcătuiesc.

   ```c
   /*
   * Funcție care determină costurile drumurilor minime de la orice nod
   ↪  la orice nod
   ```

```
* Obs: Rezultatul este reținut în matricea distances care a fost deja
↪  alocată!
*/
void FloydWarshall(Graph graph, int **distances);
```