Suport curs algoritmica grafurilor
III. Parcurgeri, drumuri de cost minim în grafuri orientate și ponderate

3 DFS

DFS(G)
  for fiecare vârf $u \in G.V$ **do**
    u.color = alb
    u.π = NIL
  time = 0
  for fiecare $u \in G.V$ **do**
    if u.color == alb **then**
      DFS_VISIT(G,u)

DFS_VISIT(G, u)
  $time = time + 1$
  $u.d = time$
  u.color = gri
  for fiecare $v \in G.Adj[u]$ **do**
    if v.color == alb **then**
      $v.\pi = u$
      DFS_VISIT(G,v)
  u.color = negru
  $time = time + 1$
  $u.f = time$

Teorema 3.1 (Teorema parantezelor)**.** *în orice căutare în adâncime a unui graf $G = (V, E)$ (orientat sau neorientat), pentru orice două vârfuri u şi v, exact una din următoarele trei afirmaţii este adevărată:*

- *intervalele $[u.d, u.f]$ şi $[v.d, v.f]$ sunt total disjuncte*
- *intervalul $[u.d, u.f]$ este conţinut în întregime în intervalul $[v.d, v.f]$ iar u este descendent al lui v în arborele de adâncime*
- *intervalul $[v.d, v.f]$ este conţinut în întregime în intervalul $[u.d, u.f]$ iar v este descendent al lui u în arborele de adâncime*

Teorema 3.2 (Teorema drumului alb)**.** *într-o pădure de adâncime a unui graf $G = (V, E)$ (orientat sau neorientat), vârful v este descendent al vârfului u dacă şi numai dacă la momentul u.d, când căutarea descoperă vârful u, vârful v este accesibil din u printr-un drum format în întregime din vârfuri albe.*

Clasificarea muchiilor

- pentru un graf $G = (V, E)$, fie $(u, v) \in E$, în funcție de timp tipul arcelor pentru DFS:
- $u.d$ marchează timpul când a fost descoperit vârful u
- $u.f$ marchează timpul când a fost explorat vârful u

tip arc	d	f
t (tree)	$u.d < v.d$	$u.f > v.f$
b (back)	$u.d > v.d$	$u.f < v.f$
f (forward)	$u.d < v.d$	$u.f > v.f$
c (cross)	$u.d > v.d$	$u.f > v.f$

3.1 Sortare topologică

- folosind algoritmul DFS se poate sorta topologic un graf orientat fără circuite (DAG - directed acyclic graph)
- realizează o aranjare liniară a vârfurilor unui graf în funcție de arcele grafului
- Sortare topologică fie un graf orientat aciclic $G = (V, E)$, sortarea topologică reprezintă ordonarea vârfurilor astfel încât dacă G conține arcul (u, v) atunci u apare înaintea lui v în înșiruire.
- multe aplicații folosesc grafuri orientate fără circuite pentru a indica precedența între evenimente
- un set de acțiuni ce trebuie îndeplinite într-o anumită ordine
- unele sarcini trebuie executate înaninte ca alte acțiuni să înceapă
- *în ce ordine trebuie executate sarcinile?*
- problema poate fi rezolvată reprezentând sarcinile ca vârfuri într-un graf
- un arc (u, v) indică precedență între activități, activitatea u înaintea activității v
- sortând topologic graful se arată ordinea efectuării acțiunilor

sortare_topologică(G)

1: apel DFS(G) pentru a determina timpii $v.f, v \in V$
2: sortare descrescătoare în funcție de timpul de finalizare (când fiecare vârf e terminat e inserat într-o listă înlănțuită)
3: **return** lista înlănțuită de vârfuri

- un graf se poate sorta topologic în timpul $\Theta(V + E)$
  - DFS durează $\Theta(V + E)$
  - pentru a insera un vârf $v \in V$ în listă e nevoie de $O(1)$ timp

Lema 3.3. *un graf orientat G este aciclic dacă şi numai dacă DFS aplicat pe el nu găseşte arce pentru care $u.d > v.d$ şi $u.f < v.f$.*

Teorema 3.4. *procedura sortare_topologică(G) produce o sortare topologică a unui graf orientat aciclic primit ca şi parametru.*

3.2 Componente tare conexe

componente_tare_conexe(G)

1: apel DFS(G) pentru a determina timpii $v.f, v \in V$
2: determină G^T
3: apel DFS(G^T) dar în bucla principala a DFS nodurile sunt sortate descrescător după $v.f$
4: fiecare arbore din pădurea găsită de DFS în pasul 3 este o componentă tare conexă

- pentru $G = (V, E)$, $G^T = (V, E^T)$ unde $E^T = \{(u, v) : (v, u) \in E\}$
- pentru reprezentarea sub formă de listă de adiacență, pentru a determină G^T e nevoie de $O(V + E)$ timp
- fie G reprezentat ca listă de adiacență, pentru procedura componente_tare_conexe(G) complexitatea în timp este
  - $\Theta(V + E)$

Lema 3.5. *fie C şi C' două componente tare conexe din graful $G = (V, E)$. $u, v \in C$, $u', v' \in C'$ şi G conţine un drum $u \rightsquigarrow u'$. Atunci G nu poate avea un drum $v' \rightsquigarrow v$.*

- graful format din componente tare conexe este un graf orientat aciclic
- fie $U \subseteq V$, putem defini $d(U) = min_{u \in U}\{u.d\}$ și $f(U) = max_{u \in U}\{u.f\}$
  - $d(U)$ reprezintă timpul pentru primul vârf descoperit de DFS din U
  - $f(U)$ reprezintă timpul pentru ultimul vârf prelucrat de DFS din U

Lema 3.6. *fie C şi C' două componente tare conexe distincte din graful orientat $G = (V, E)$. Dacă există un arc $(u, v) \in E$, unde $u \in C$ şi $v \in C'$ atunci $f(C) > f(C')$.*

Corolar 3.6.1. *fie C şi C' două componente tare conexe distincte din graful orientat $G = (V, E)$. Dacă există un arc $(u, v) \in E^T$, unde $u \in C$ şi $v \in C'$ atunci $f(C) < f(C')$.*

Teorema 3.7. *procedura **componente_tare_conexe(G)** găseşte corect componentele tare conexe din graful orientat G.*

3.3 Drum de lungime minimă - sursă unică

Problema drumului de cost minim într-un graf pentru un singur vârf sursa se poate defini în felul următor: fie un graf $G = (V, E)$, se vrea să se găsească un drum de la un vârf **sursă** $s \in V$ la fiecare vârf din graf $v \in V$. Algoritmii care rezolvă problema drumului de cost minim de la un vârf sursă pot fi folosiți pentru a rezolva și alte tipuri de probleme, printre care și variante ale problemei drumului de cost minim:

- **Drum de cost minim pentru o singură destinație**: se caută drumul de cost minim de la toate vârfurile din graf $v \in v$ către un singur vârf **destinație** t.
- **Drum minim între o pereche de vârfuri**: se caută drumul de cost minim între vârfurile u și v (pentru u și v date). Dacă se rezolvă problema drumului de cost minim de la un vârf sursă către toate vârfurile din graf se rezolvă si această variantă a problemei.
- **Drum de cost minim între toate perechile de vârfuri**: se caută drumul de cost minim între toate perechile de vârfuri u și v din graf.

Structura optimă a unui drum de cost minim

În general algoritmii care rezolvă problema drumului de cost minim se bazează pe proprietatea drumului minim: un drum de cost minim între două vârfuri conține drumuri de cost minim.

Lema 3.8 ((sub)drumuri de cost minim într-un drum de cost minim)**.**
Pentru graf orientat şi ponderat $G = (V, E)$ cu funcţia de pondere $w : E \to \mathbb{R}$, fie $p = \langle v_0, v_1, ..., v_k \rangle$ un drum de cost minim de la vârful v_0 la vârful v_k şi pentru oricare i şi j, $0 \leq i \leq j \leq k$, fie $p_{ij} = \langle v_i, v_{i+1}, ..., v_j \rangle$ un drum din p (subdrum) între v_i şi v_j. Atunci p_{ij} este un drum de cost minim între v_i şi v_j

Demonstraţie.
Drumul p poate fi descompus în $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, în acest caz costul drumului este $w(p) = w(p_{0i}) + w(p_{ij}) + w(pkj)$ (suma ponderilor arcelor ce formează drumul).

Dacă se presupune ca există drumul p'_{ij} de la v_i la v_j cu costul $w(p'_{ij}) < w(p_{ij})$, atunci $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ este un drum de la V_0 la v_k de cost $w(p_{0i}) + w(p'_{ij}) + w(pkj) < w(p)$ ceea ce contrazice presupunerea că p este un drum minim de la v_0 la v_k. □

Arce cu cost negativ

Unele instanțe ale problemei de cost minim permit ca ponderea unui arc să poată lua valori negative. Dacă un graf $G = (V, E)$ nu conține circuite de pondere negativă accesibile din vârful sursă s, pentru toate $v \in V$, ponderea drumului de cost minim $\delta(s, v)$ are o valoare finită (chiar dacă este negativă). În schimb dacă $G = (V, E)$ conține un circuit de pondere negativă accesibil din vârful s ponderea drumului de cost minim nu este bine definită. Nu există un drum de la s către un vârf din $G = (V, E)$ prin circuit care să fie drum minim (se poate găsi un drum mai bun dacă se urmărește drumul și se traversează circuitul negativ). Dacă există un circuit negativ pe un drum de la s la v definim $\delta(s, v) = -\infty$.

Circuite

Un drum de cost minim nu poate conține un circuit.

Dacă $G = (V, E)$ conține pe drumul de la s la v un circuit de pondere negativă tot timpul s-ar găsi un drum mai bun.

În cazul unui circuit de pondere pozitivă dacă se elimină circuitul din drum se obține un drum între vârfurile s și v cu un cost mai mic. Altfel, fie $p = \langle v_0, v_1, ..., v_k \rangle$ un drum și $c = \langle v_i, v_{i+1}, ..., v_j \rangle$ un circuit de pondere pozitivă pe drumul p ($v_i = v_j$ și $w(c) > 0$), atunci drumul $p' = \langle v_0, v_1, ..., v_i, v_{j+1}, v_{j+2}, ..., v_k \rangle$ are costul $w(p') = w(p) - w(c) < w(p)$ și p nu poate fi drumul minim de la v_0 la v_k.

Mai rămân circuite de pondere 0. Se poate elimina un circuit de pondere 0 dintr-un drum oarecare pentru a obține un drum cu același cost. Se poate elimina orice circuit de pondere 0 de pe un drum de la vârful sursă s la vârful destinație v. Dacă există un drum minim de la s la v care conține un circuit de pondere 0, există un drum de cost minim de la s la v fără acest circuit.

Un drum de cost minim cu conține circuite. Orice drum aciclic din graful $G = (V, E)$ conține cel mult $|V|$ vârfuri distincte și cel mult $|V| - 1$ arce.

Reprezentarea unui drum de cost minim

Pentru un graf dat $G = (V, E)$, pentru fiecare vârf $v \in V$ se menține un **predecesor** $v.\pi$ care este un vârf sau *NIL*. Algoritmii de drum minim vor modifica atributul π astfel încât lanțul predecesorilor de la vârful v merge înapoi pe drumul minim de la s la v. Astfel, fie un vârf v pentru care

$v.\pi \neq NIL$, procedura AFISARE_DRUM(G, s, v) va afișa drumul de cost minim de la s la v.

```
AFISARE_DRUM(G, s, v)
1: if v == s then
2:     print s
3: else if v.π == NIL then
4:     print "nu este drum de la" s " la " v
5: else
6:     AFISARE_DRUM(G, s, v.π)
7:     print v
```

Se poate defini graful predecesor $G_\pi = (V_\pi, E_\pi)$ indus de valorile atributului π. Setul V_π este setul vârfurilor lui G pentru care atributul π este diferit de *NIL* plus vârful sursă s:

$$V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\}.$$

Setul E_π este setul arcelor induse de valorile lui π pentru vârfurile din V_π:

$$E_\pi = \{(v.\pi, v) \in E \mid v \in V_\pi - \{s\}\}.$$

Pentru valorile lui π găsite de algoritmi, la terminare, G_π este un arbore minim (în sensul sumei ponderii arcelor ce compun drumul de la s la v, unde vârful s este rădăcina arborelui). Fie $G = (V, E)$ un graf ponderat și orientat cu funcția de pondere $w : E \to \mathbb{R}$ și G nu conține circuite de pondere negativă accesibile din vârful sursă $s \in V$, astfel încât drumurile de cost minim sunt bine definite. Un arbore minim de drumuri cu rădăcina în s este un graf orientat $G' = (V', E')$ unde $V' \subseteq V$ și $E' \subseteq E$ astfel încât:

1. V' este setul vârfurilor accesibile din s;
2. G' este un arbore cu rădăcina în s;
3. pentru oricare vârf $v \in V'$ drumul de la s la v în G' este un drum de cost minim de la s la v în G.

Procedura de relaxare

Algoritmii de drum minim folosesc procedura de *relaxare*. Pentru fiecare vârf $v \in V$ se menține un atribut $v.d$ care reprezintă limita superioară a ponderii (costului) drumului de la s la v. $v.d$ estimează costul minim al drumului. Folosind o procedură în $\Theta(V)$ se inițializează estimările drumului de cost minim.

```
INITIALIZARE_S(G, s)
1: for v ∈ G.V do
2:     v.d = inf
3:     v.π = NIL
4: s.d = 0
```

Procesul de *relaxare* a arcului (u, v) presupune testarea dacă drumul de cost minim până la v găsit până în acest moment se poate îmbunătăți dacă se trece prin u, dacă da se actualizează $v.d$ și $v.\pi$. Codul de mai jos face acest pas.

```
RELAXARE(u, v, w)
1: if v.d > u.d + w(u, v) then
```

2: $v.d = u.d + w(u, v)$
3: $v.\pi = u.\pi$

Procedura de relaxare modifică estimatorii drumului de cost minim ($v.d$) și predecesorii vârfurilor.

Diferența între algoritmii de drum de cost minim este numărul de apeluri ale procedurii de relaxare, algoritmul lui *Bellman-Ford* apelează procedura de relaxare de $|V| - 1$ ori pentru fiecare arc iar algoritmul lui *Dijkstra* și algoritmul de drum de cost minim pentru grafuri aciclice apelează procedura de relaxare o singură dată pentru fiecare arc.

3.4 Algoritmul Bellman-Ford

Algoritmul Bellman-Ford rezolvă problema drumului minim de la un nod sursă s pentru cazul general când avem și ponderi negative.

Bellman_Ford(G, w, s))

1: INITIALIZARE_S(G,s)
2: **for** $i = 1$ **la** $|V| - 1$ **do**
3: **for** fiecare arc $\{u, v\} \in E$ **do**
4: RELAX(u, v, w)
5: **for** fiecare arc $\{u, v\} \in E$ **do**
6: **if** $v.d > u.d + w(u, v)$ **then**
7: **return** FALSE
8: **return** TRUE

- algoritmul rulează în $O(VE)$ timp
- pasul de inițializare (linia 1) durează $\Theta(V)$
- parcurgerea din liniile 2-4 durează $O(VE)$
- bucla for din liniile 5-7 durează $O(E)$

Lema 3.9. *fie $G = (V, E)$ un graf ponderat orientat cu nodul sursă s şi funcţia de pondere $w : E \to \mathbb{R}$, presupunem că G nu conţine circuite de pondere negativă accesibile din vârful s. După $|V| - 1$ iteraţii ale buclei for din liniile 2-4 a procedurii Bellman_Ford(G) avem $v.d = \delta(s, v)$ pentru toate vârfurile v accesibile din s.*

Corolar 3.9.1. *fie $G = (V, E)$ un graf ponderat orientat cu nodul sursă s şi funcţia de pondere $w : E \to \mathbb{R}$. Pentru fiecare vârf $v \in V$ există un drum de la s la v dacă şi numai dacă procedura Bellman_Ford(G) se termină cu $v.d < \infty$.*

Teorema 3.10. *Corectitudine Bellman-Ford. Fie procedura Bellman_Ford(G) care este rulată pe un graf orientat şi ponderat $G = (V, E)$ din nodul sursă s şi funcţia de pondere $w : E \to \mathbb{R}$. Dacă G nu conţine circuite de pondere negativă accesibile din s algoritmul va întoarce TRUE, $v.d = \delta(s, v) \forall v \in V$ iar graful predecesorilor G_π este un arbore minim cu rădăcina în s. Dacă G conţine un circuit de pondere negativă accesibil din s, algoritmul întoarce FALSE.*

3.5 Drum de cost minim în grafuri orientate aciclice

drum_minim_dag(G)

1: sortate_topologica(G)
2: INITIALIZARE_S(G,s)
3: **for** fiecare vârf v sortat topologic **do**
4: **for** $v \in G.Adj[u]$ **do**

5: RELAX(u,v,w)

- timp de rulare
  - sortate topologică: $\Theta(V + E)$
  - INITIALIZARE_S: $\Theta(V)$
  - bucla for (liniile 4-5) relaxează fiecae arc o singură dată
  - timpul total de rulare $\Theta(V + E)$

Teorema 3.11. *Corectitudine drum_minim_dag(G). Dacă un graf orientat, ponderat şi aciclic $G = (V, E)$ are ca şi sursă vârful s, la terminarea procedurii drum_minim_dag(G) $v.d = \delta(s, v) \forall v \in V$ iar graful predecesorilor G_π este un arbore minim.*

3.6 Algoritmul lui Dijkstra

- algoritmul lui Dijkstra rezolvă problema drumului minim pentru un graf orientat ponderat $G = (V, E)$ în care $w(u, v) \geq 0, \{u, v\} \in E$
- algoritmul menține un set S de vârfuri pentru care drumul minim de la sursa s a fost determinat
- în implementarea prezentata se folosește o coadă cu priorități pentru vârfuri, cheia fiind $v.d$

Teorema 3.12. *Corectitudine Dijkstra. Fie procedura Dijkstra_queue(G) rulată pe un graf orientat, ponderat $G = (V, E)$ ce nu conţine ponderi negative şi s vârful sursă. La terminare $u.d = \delta(s, u) \forall u \in V$.*

Cât de rapid este algoritmul Dijkstra_queue?

- menține o coadă cu priorități Q prin apelul operațiilor: INSERT (implicit în linia 3), EXTRACT_MIN (linia 5) și DECREASE_KEY (implicit în RELAX, linia 8)
- algoritmul apelează INSERT și EXTRACT_MIN pentru fiecare vârf
- fiecare vârf este adăugat în setul S o singură data, fiecare arc din $Adj[u]$ este examinat o singură dată pe liniile 7-8
- numărul total de arce din lista de adiacență este $|E|$, bucla for iterează de $|E|$ ori (liniile 7-8), algoritmul apelează DECREASE_KEY de cel mult $|E|$ ori
- timpul total de rulare depinde de implementarea cozii cu priorități
- dacă vârfurile sunt numerotate de la 1 la $|V|$
  - $v.d$ e stocat pe poziția v
  - fiecare operație INSERT și DECREASE_KEY necesită $O(1)$ timp iar operația EXTRACT_MIN necesită $O(V)$ timp
  - pentru un timp total $O(V^2 + E) = O(V^2)$
- dacă graful este suficient de rar, coada se poate implementa ca și *binary min-heap*
  - fiecare operație EXTRACT_MIN durează $O(\lg V)$, există $|V|$ astfel de operații
  - timpul necesar construirii *binary min-heap* este $O(V)$
  - fiecare operație DECREASE_KEY necesită $O(\lg V)$ timp, există $|E|$ astfel de operații

  - timpul total de rulare este $O((V+E)\lg V)$, dacă toate vârfurile sunt accesibile din sursă timpul este $O(E \lg V)$

- se poate obține un timp de rulare $O(V \lg V + E)$ dacă coada cu priorități este implementată cu un *heap Fibonacci*

3.7 Referințe

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press.

2. Geir Agnarsson and Raymond Greenlaw. 2006. Graph Theory: Modeling, Applications, and Algorithms. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

3. Mark Newman. 2010. Networks: An Introduction. Oxford University Press, Inc., New York, NY, USA.

4. Cristian A. Giumale. 2004. Introducere în analiza algoritmilor, teorie și aplicație. Polirom.

5. cursuri Teoria grafurilor: Zoltán Kása, Mircea Marin, Toadere Teodor.