

Aplicații ale parcurgerii grafurilor orientate

Victor Manz

Colegiul Național de Informatică „Tudor Vianu”, București

Rezumat

Acest curs își propune să prezinte câteva aplicații interesante ale parcurgerii grafurilor orientate: sortarea topologică, determinarea componentelor tare conexe (folosind algoritmul Kosaraju – Sharir) și rezolvarea problemei 2-SAT.

Cuvinte cheie: Sortarea topologică, Componente tare conexe, 2-SAT.

Aplicații ale parcurgerii grafurilor orientate

1. Recapitulare: parcurgerea grafurilor orientate

Parcurgerea grafurilor (orientate sau neorientate) se referă la vizitarea tuturor vârfurilor acestora într-o anumită ordine, însoțită (eventual) de prelucrarea informațiilor asociate acestora, astfel încât fiecare vârf să fie luat în considerare o singură dată.

Parcurgerea în lățime (Breadth First Search - BFS) vizitează toate vârfurile accesibile din nodul inițial x_0 în ordinea distanțelor față de x_0 . (prin distanța de la x la y înțelegem lungimea unui drum minim cu extremitatea inițială x și cea finală y).

```

BFS(G, x0)
1:   pentru fiecare vârf x al lui G
2:   {
3:       d[x] = -1
4:       pred[x] = -1
5:   }
6:   d[x0] = 0
7:   pred[x0] = 0
8:   ADAUGĂ_ÎN_COADĂ(Q, x0)
9:   cât timp Q NU este vidă
10:  {
11:      x = PRIMUL_ELEMENT(Q)
12:      ELIMINĂ_DIN_COADĂ(Q)
13:      pentru fiecare vecin (succesor) y al lui x
14:      {
15:          dacă d[y] == -1
16:          {
17:              d[y] = 1 + d[x]
18:              pred[y] = x
19:              ADAUGĂ_ÎN_COADĂ(Q, y)
20:          }
21:      }
22:  }

```

Algoritmul *BFS* folosește o coadă Q în care este adăugat la început vârfurile inițial x_0 (linia 8), iar apoi, cât timp aceasta NU e vidă, este scos din Q vârfurile accesibile x (liniile 11 și 12) și sunt adăugați toți vecinii (succesorii în cazul orientat) y ai lui x , care nu au fost încă vizitați (linia 19). Pentru fiecare vârf y adăugat în Q se poate verifica relativ ușor (folosind

inducția matematică) faptul că distanța de la x_0 la y , notată $d[y]$ este cu 1 mai mare decât $d[x]$.

Cu alte cuvinte, vârfurile sunt vizitate în ordinea distanțelor lor față de x_0 .

Algoritmul poate fi adaptat astfel încât, pe lângă distanțele de la x_0 , reținute în vectorul d , să construiască un al doilea vector, $pred$ (liniile 4, 7 și 18), cu semnificația: „ $pred[x]$ = penultimul vârf al unui drum de lungime minimă de la x_0 la x ”. Pe baza lui $pred$ vom putea apoi obține un drum minim de la x_0 la fiecare vârf accesibil din el.

Complexitatea timp a procedurii BFS este $O(m)$, unde prin m am notat numărul muchiilor (arcelor) grafului G . Numărul de iterații ale buclei *pentru* de la linia 13 coincide cu gradul lui x . Prin urmare, numărul total al iterațiilor este mai mic sau egal cu suma gradelor (externe în cazul orientat), adică $2m$ pentru cazul neorientat, respectiv m pentru cazul orientat.

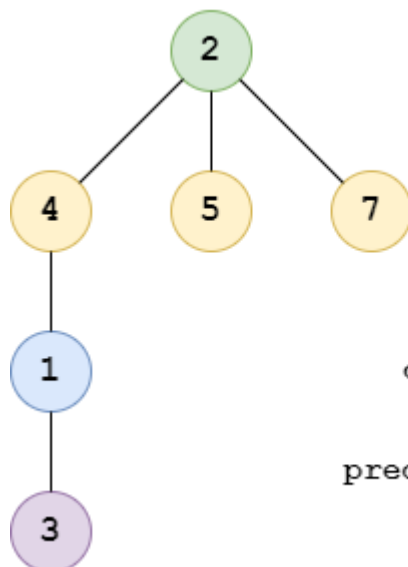
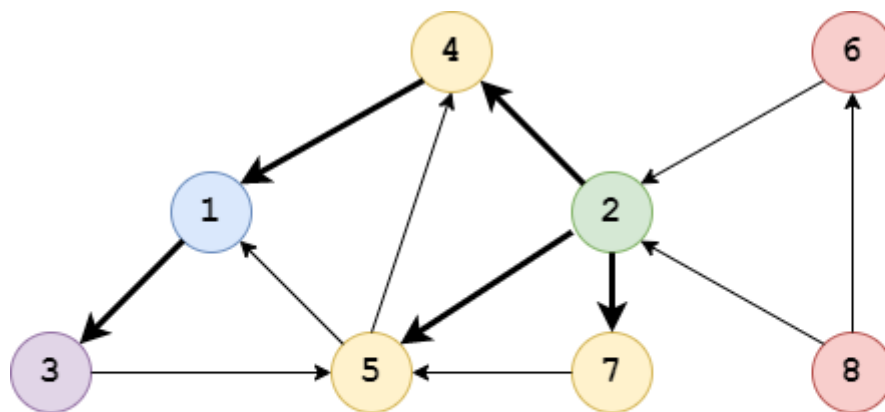
Observația 1.1 Apelarea procedurii BFS are ca efect obținerea valorilor $d[x]$ și $pred[x]$ pentru fiecare **vârf accesibil** din x_0 . Vârfurile x pentru care **NU există drum (lanț)** de la x_0 la x vor putea fi de asemenea ușor identificate, fiind acelea pentru care $d[x] = -1$ (și $pred[x] = -1$).

Observația 1.2 Procedura BFS poate fi modificată astfel încât să aibă ca efect obținerea valorilor $d[x]$ și $pred[x]$ pentru fiecare **vârf accesibil** din fiecare vârf x_0 aparținând unei mulțimi de vârfuri V_0 fără a crește complexitatea timp a algoritmului. Singura modificare necesară este includerea instrucțiunilor de la liniile 6, 7 și 8 într-o buclă care să parcurgă toate vârfurile $x_0 \in V_0$. O astfel de modificare ar face ca $d[x]$ să reprezinte lungimea celui mai scurt drum care poate lega un vârf din mulțimea V_0 de x ; $pred[x]$ va reprezenta predecesorul lui x pe un astfel de drum.

Observația 1.3 În cazul în care se dorește vizitarea **tuturor** vârfurilor grafului (în alt scop decât cel al determinării unor distanțe) este necesară apelarea procedurii BFS pentru fiecare vârf x_0 care nu a fost încă prelucrat. Într-o astfel de situație secțiunea de inițializare (liniile 1 - 5) trebuie mutată în afara procedurii, condiția pentru apelarea $BFS(G, x_0)$ fiind ca

$d[x_0]$ să fi rămas la valoarea inițială, -1. Complexitatea timp a acestei versiuni a algoritmului este $O(m+n)$, unde am notat cu m numărul muchiilor (arcelor) grafului și cu n numărul vârfurilor acestuia.

Observația 1.4 Fiecare apel al procedurii *BFS* induce un arbore cu rădăcină, cu vectorul de tați *pred*. În cazul vizitării tuturor vârfurilor grafului, *pred* va descrie o *pădure* (reuniune de arbori cu rădăcină disjuncți).



$Q = (2, 4, 5, 7, 1, 3)$

$d = (2, 0, 3, 1, 1, -1, 1, -1)$

$pred = (4, 0, 1, 2, 2, -1, 2, -1)$

Parcurgerea în adâncime (Depth First Search - DFS) vizitează de asemenea toate vârfurile accesibile din nodul inițial x_0 , dar într-o ordine diferită de cea a distanțelor față de x_0 . La fiecare pas, după vizitarea unui vârf x se încearcă trecerea la **un vecin (succesor)** oarecare y al acestuia. În cazul găsirii unui y se continuă cu acesta și se reține x ca fiind $pred[y]$; în caz contrar se revine la $pred[x]$. Dacă x este vârful inițial, algoritmul se încheie.

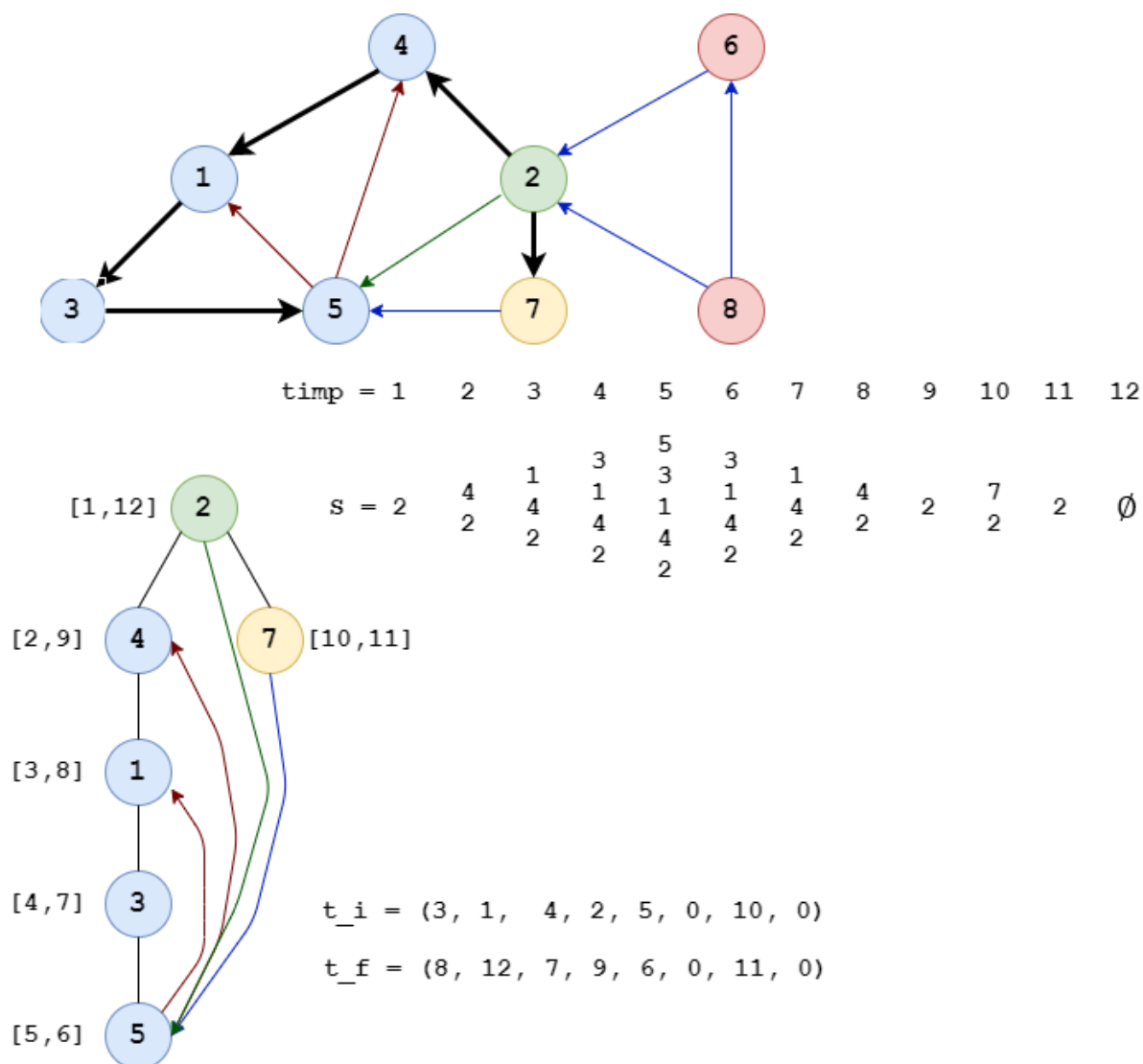
Algoritmul poate fi implementat iterativ, folosind o stivă S în care este la început adăugat vârful inițial x_0 . Apoi, cât timp S **nu** este vidă, se reține în x elementul din vârful stivei și se caută un vârf y accesibil direct din x care să nu fi fost încă vizitat (liniile 13 – 21). În cazul în care nu este găsit un astfel de vârf, x este eliminat din stivă; altfel y este adăugat.

```

DFS (G, x0)
1:   pentru fiecare vârf x al lui G
2:   {
3:       t_i[x] = t_f[x] = 0
4:       pred[y] = -1
5:       ultimul_s[x] = primul succesor din lista lui x
6:   }
7:   timp_c = 0
8:   pred[x0] = 0
9:   ADAUGĂ_ÎN_STIVĂ(S, x0)
10:  t_i[x0] = ++timp_c
11:  cât timp S NU este vidă
12:  {
13:      x = VÂRFUL_STIVEI(S)
14:      y = 0
15:      pentru y_c succesor al lui x începând cu ultimul_s[x]
16:      {
17:          dacă t_i[y_c] == 0
18:          {
19:              y = y_c
20:              ultimul_s[x] = următorul succesor
21:                          al lui x după y_c
22:              break
23:          }
24:      }
25:      dacă y = 0
26:      {
27:          t_f[x] = ++timp_c
28:          ELIMINĂ_DIN_STIVĂ(S)
29:      }
30:      altfel
31:      {
32:          t_i[y] = ++timp_c
33:          ADAUGĂ_ÎN_STIVĂ(S, y)
34:      }
35:  }

```

Observația 1.5 Ca și în cazul parcurgerii în lățime, procedura *DFS* induce un arbore cu rădăcină, cu vectorul de tați *pred*. Dacă sunt vizitate toate nodurile, se obține o pădure. În acest caz, complexitatea timp a algoritmului este de asemenea $O(m+n)$, fiecare vârf fiind vizitat exact o dată și la fiecare prelucrare fiind parcursă lista vecinilor (succesorilor) săi. Trebuie subliniat rolul vectorului *ultimul_s*, cu $ultimul_s[x] =$ următorul succesor al lui x (primul care nu a fost prelucrat). În lipsa acestuia parcurgerea listei de succesorii ai lui x s-ar relua de la început de fiecare dată când x ar ajunge în vârful stivei S .



Observația 1.6 Parcurgerea în adâncime NU vizitează nodurile în ordinea distanțelor față de vârful inițial. Prin urmare, nu mai are sens să construim vectorul d . În schimb, vom

reține pentru fiecare vârf accesibil x valoarea $t_i[x]$ = momentul de timp la care este adăugat x în stiva S și $t_f[x]$ = momentul de timp la care este eliminat din stivă vârful x .

Observația 1.7 Vectorii t_i și t_f (care ar reține momentele de intrare și respectiv ieșire din coada Q) ar putea fi construiți și în cazul parcurgerii în lățime, inserând în procedura BFS instrucțiunea $t_i[x] = ++timp_c$ după linia 8, instrucțiunea $t_f[x] = ++timp_c$ după linia 12 și instrucțiunea $t_i[y] = ++timp_c$ după linia 19, dar semnificația lor nu ar fi la fel de importantă ca în cazul parcurgerii în adâncime.

Observația 1.8 În cazul parcurgerii în adâncime, pentru oricare două vârfuri x_1 și x_2 intervalele $[t_i[x_1], t_f[x_1]]$ și $[t_i[x_2], t_f[x_2]]$ verifică exact una dintre următoarele 3 condiții:

- $[t_i[x_1], t_f[x_1]] \cap [t_i[x_2], t_f[x_2]] = \emptyset$ ceea ce corespunde situației în care niciunul dintre x_1 și x_2 nu este strămoș al celuilalt în arborele (pădurea) DFS;
- $[t_i[x_2], t_f[x_2]] \subseteq [t_i[x_1], t_f[x_1]]$, caz în care x_1 este strămoș al lui x_2 în arborele (pădurea) DFS;
- $[t_i[x_1], t_f[x_1]] \subseteq [t_i[x_2], t_f[x_2]]$, pentru cazul în care x_2 este strămoș al lui x_1 în arborele (pădurea) DFS.

Observația 1.9 Aplicarea algoritmului de parcurgere în adâncime împarte arcele grafului în 4 categorii:

- arce ale arborelui DFS** (colorate cu **negru** în desenul de mai sus);
- arce înapoi**, care unesc un vârf cu un strămoș (ascendent) al său în arborele DFS (colorate cu **roșu** în desenul de mai sus);
- arce înainte**, care unesc un vârf cu un descendent al său în arborele DFS (colorate cu **verde** în desenul de mai sus);
- arce transversale**, care unesc un vârf x cu un altul y , iar x nu este nici ascendent și nici descendent al lui y în arborele DFS (colorate cu **albastru** în desenul de mai sus).

Observația 1.10 Deși varianta iterativă de mai sus are beneficii didactice și are avantajul de a nu folosi neapărat zona de memorie stivă, de cele mai multe ori vom prefera implementarea recursivă a procedurii *DFS*, în care am presupus că atât variabila *timp*, cât și vectorii *pred*, *t_i* și *t_f* au fost declarați global și inițializați corespunzător:

```
DFS(G, x)
1:   t_i[x] = ++timp_c
2:   pentru fiecare vecin (succesor) y al lui x
3:   {
4:       dacă t_i[y] == 0
5:       {
6:           pred[y] = x
7:           DFS(G, y)
8:       }
9:   }
10:  t_f[x] = ++timp_c
```

2. Sortarea topologică

Fie $G = (V, E)$ un graf orientat. Numim *sortare topologică* a lui G un șir $sort_top = (x_1, x_2, \dots, x_n)$ cu proprietățile:

- a) $x_i \neq x_j \forall i, j, 1 \leq i < j \leq n$ (elementele șirului sunt distincte);
- b) $\{x_1, x_2, \dots, x_n\} = V$ (toate vârfurile aparțin șirului, și acesta nu conține decât vârfuri ale grafului);
- c) Dacă $(x_i, x_j) \in E$, atunci $i < j$ (pentru orice arc al grafului G extremitatea inițială a sa se află înaintea celei finale în șirul $sort_top$).

Observația 2.1 Pentru un graf orientat $G = (V, E)$ o sortare topologică există dacă și numai dacă acesta este **aciclic** (nu are circuite). Un astfel de graf poartă în limba engleză numele „Directed Acyclic Graph” (prescurtat DAG).

Observația 2.2 În cazul în care există, sortarea topologică **nu** este unică.

O sortare topologică poate fi obținută atât prin adaptarea algoritmului de parcurgere în lățime, cât și a celui de parcurgere în adâncime.

În cazul *BFS*, vom folosi un vector npr , cu $npr[i] =$ numărul de predecesori ai celui de-al i -lea vârf al grafului care nu au fost încă prelucrați. Inițial, pentru fiecare arc (x_i, x_j) vom incrementa $npr[j]$. Vom adăuga apoi în coada Q toate vârfurile x_i cu proprietatea că $npr[i]=0$.

Pentru fiecare vârf x scos din coada Q vom parcurge succesorii y ai săi și vom decrementa $npr[y]$; vom adăuga y în Q doar atunci când $npr[y]$ devine 0. Ordinea în care vârfurile sunt adăugate sau eliminate din Q (ordinea în care sunt adăugate vârfurile coincide cu cea în care sunt eliminate fiind vorba despre o coadă) corespunde unei sortări topologice a grafului întrucât fiecare vârf este prelucrat (adăugat în vectorul $sort_top$) doar în momentul în care nu mai există predecesori ai săi neprelucrați.

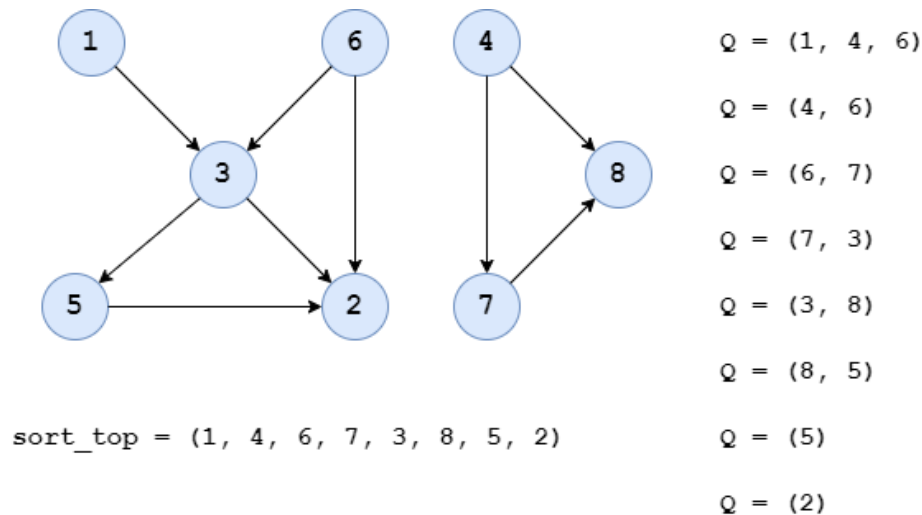
```

BFS_SORT_TOP(G)
1:   pentru fiecare vârf x al lui G
2:   {
3:       nrp[x] = 0
4:   }
5:   pentru fiecare arc (x,y) al lui G
6:   {
7:       nrp[y]++
8:   }
9:   pentru fiecare vârf x al lui G
10:  {
11:      dacă nrp[x] == 0
12:      {
13:          ADAUGĂ_ÎN_COADĂ(Q, x)
14:      }
15:  }
16:  nr_s_t = 0
17:  cât timp Q NU este vidă
18:  {
19:      x = PRIMUL_ELEMENT(Q)
20:      ELIMINĂ_DIN_COADĂ(Q)
21:      sort_top[++nr_s_t] = x
22:      pentru fiecare y succesori al lui x
23:      {
24:          nrp[y]--
25:          dacă nrp[y] == 0
26:          {
27:              ADAUGĂ_ÎN_COADĂ(Q, y)
28:          }
29:      }
30:  }

```

Complexitatea timp a algoritmului este aceeași cu cea a *BFS*, adică $O(m+n)$. Fiecare vârf intră și iese din coadă o dată (liniile 13, 27 și respectiv 20), iar fiecare listă de succesori a fiecărui vârf x este parcursă de asemenea o singură dată (liniile 22 – 29), după eliminarea lui x din coada Q .

Observația 2.3 În cazul în care se încearcă apelarea procedurii *BFS_SORT_TOP* pentru un graf orientat G care NU este DAG, nu toate vârfurile vor fi adăugate în vectorul *sort_top* (acele vârfuri care aparțin unui circuit nu vor fi adăugate în coadă).



Dacă în cazul algoritmului *BFS_SORT_TOP* sortarea topologică corespunde chiar ordinii în care vârfurile sunt prelucrate (adăugate/eliminate în/din coadă), în varianta derivată din parcurgerea în adâncime, vectorului obținut va trebui să i se inverseze ordinea elementelor (primul va trebui interschimbat cu ultimul, al doilea cu penultimul etc.).

Mai exact, un vârf x va fi adăugat în vectorul *sort_top* la momentul $t_f[x]$, adică doar atunci când nu mai există vârfuri accesibile din el neprelucrate. În acest fel avem certitudinea că toate vârfurile y accesibile din x au fost deja adăugate în vector (conform observației 1.8 $t_f[y] < t_f[x]$), aflându-se inițial înaintea lui x în vectorul *sort_top* și după x în urma inversării ordinii acestuia.

```

DFS_SORT_TOP (G, x)
1:   t_i[x] = ++timp_c
2:   pentru fiecare succesori y al lui x
3:   {
4:       dacă t_i[y] == 0
5:       {
6:           DFS_SORT_TOP(G, y)
7:       }
8:   }
9:   t_f[x] = ++timp_c
10:  sort_top[++nr_s_t] = x

```

```

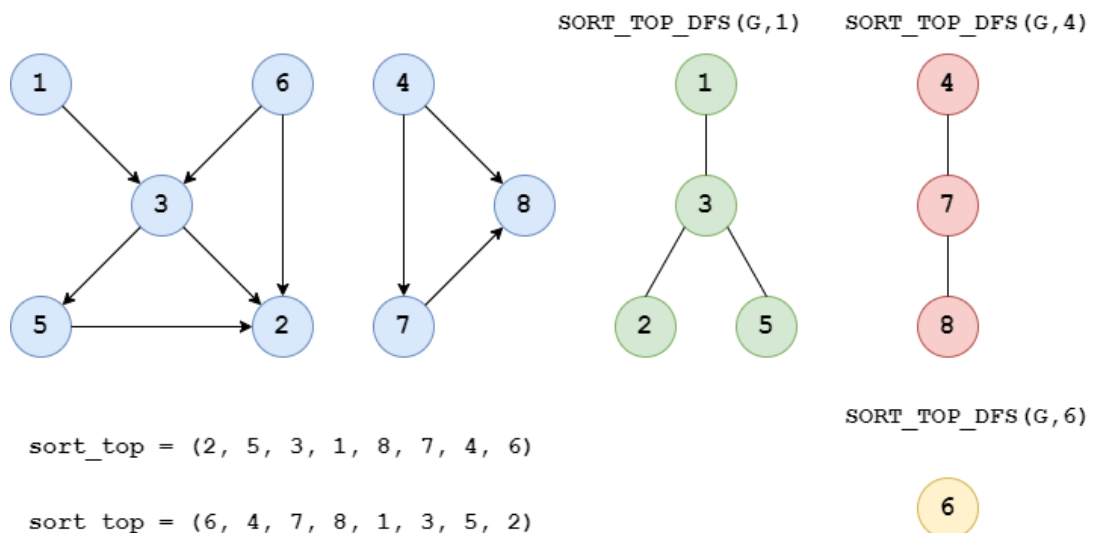
SORTAREA_TOPOLOGICĂ(G)
1:   pentru fiecare vârf  $x$  al lui  $G$ 
2:   {
3:        $t_i[x] = t_f[x] = 0$ 
4:   }
5:    $nr\_s\_t = 0$ 
6:   timp = 0
7:   pentru fiecare vârf  $x$  al lui  $G$ 
8:   {
9:       dacă  $t_i[x] == 0$ 
10:      {
11:          DFS_SORT_TOP( $G, x$ )
12:      }
13:  }
14:  INVERSARE_ORDINE(sort_top, nr_s_t)

```

Observația 2.4 Am presupus că vectorii $t_i, t_f, sort_top$ și variabila nr_s_t (care reține numărul de vârfuri ce au fost adăugate în vectorul $sort_top$) au fost declarate global.

Observația 2.5 Complexitatea timp a algoritmului de mai sus este aceeași cu cea a parcurgerii în adâncime, adică $O(m+n)$, întrucât procedura *INVERSARE_ORDINE*, apelată la linia 14 are complexitate liniară în n .

Observația 2.6 Spre deosebire de versiunea derivată din parcurgerea în lățime, în cazul sortării topologice bazate pe algoritmul DFS, vectorul $sort_top$ poate fi obținut chiar dacă graful G nu este DAG. Într-o astfel de situație însă, vectorul **nu** va corespunde unei sortări topologice a vârfurilor.



3. Determinarea componentelor tare conexe

Definiția 3.1 Un graf orientat $G = (V, E)$ se numește **tare conex** dacă între oricare două vârfuri ale sale există (cel puțin) un drum.

Definiția 3.2 Fie un graf orientat $G = (V, E)$. Un **subgraf** al său este un graf orientat $G' = (V', E')$ cu proprietățile:

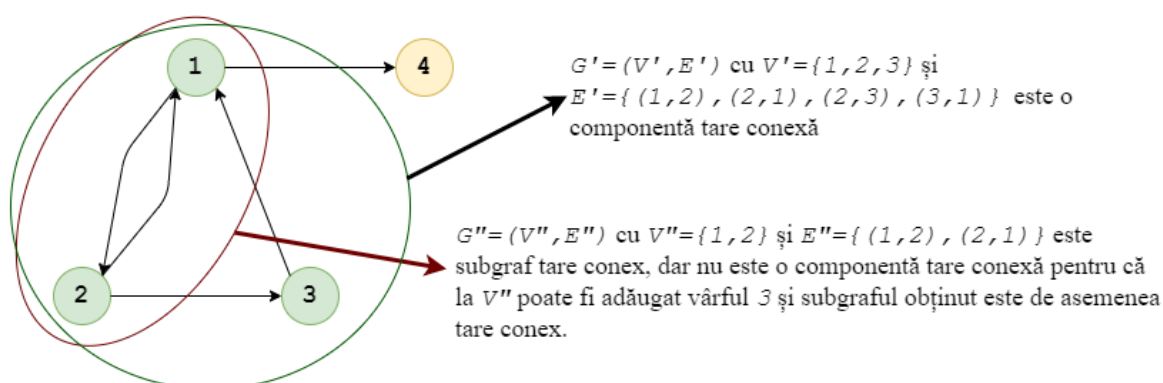
- $V' \subseteq V$
- $E' = \{(x, y) \mid x \in V', y \in V', (x, y) \in E\}$

Definiția 3.3 Fie A și B două mulțimi, $A \subseteq B$. Spunem că A este o submulțime **maximală** cu proprietatea P dacă:

- A are proprietatea P ;
- A' **nu** are proprietatea $P \forall A', A \subseteq A' \subseteq B$.

Cu alte cuvinte, A are proprietatea și **nu** poate fi extinsă astfel încât să își păstreze proprietatea P .

Definiția 3.4 O **componentă tare conexă** a unui graf orientat $G = (V, E)$ este un subgraf $G' = (V', E')$ tare conex maximal al lui G . (Cu alte cuvinte, G' este tare conex și oricare ar fi $G'' = (V'', E'')$ un alt subgraf al lui G cu proprietatea că $V' \subseteq V''$, G'' **nu** este tare conex.)



Observația 3.5 Componentele tare conexe sunt unic determinate.

Observația 3.6 Componentele tare conexe determină o partiție a vârfurilor grafului, dar nu și o partiție a arcelor, întrucât pot exista arce care nu aparțin niciunei componente (arcul $(1,4)$ în exemplul de mai sus).

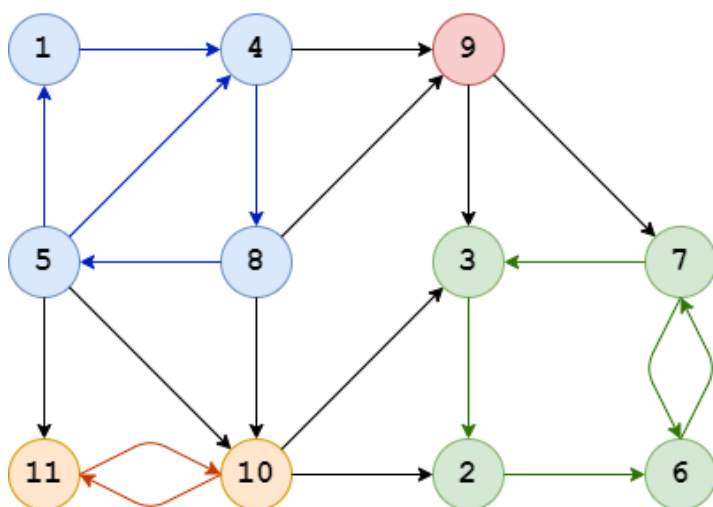
Definiția 3.7 Dacă $G = (V, E)$ este un graf orientat, atunci **graful transpus** asociat lui G , notat G^T se obține prin inversarea sensului de parcurgere al tuturor arcelor lui G . Cu alte cuvinte G^T are aceeași mulțime a vârfurilor ca și G , iar mulțimea arcelor sale este definită de relația: $E^T = \{(x, y) | (y, x) \in E\}$.

Observația 3.8 Componentele tare conexe ale grafului transpus determină aceeași partiție a mulțimii vârfurilor grafului ca și componentele tare conexe ale grafului inițial.

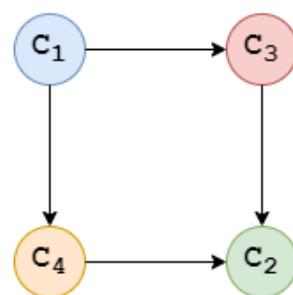
Definiția 3.9 Pornind de la un graf orientat $G = (V, E)$ putem construi un metagraf orientat numit **graful de condensare al componentelor tare conexe (CTC)**, definit ca $G^{CTC} = (V^{CTC}, E^{CTC})$ cu:

- V^{CTC} = mulțimea componentelor tare conexe ale lui G
- $(C_1, C_2) \in E^{CTC} \Leftrightarrow (\exists x_1 \in V(C_1), \exists x_2 \in V(C_2) \text{ cu proprietatea } (x_1, x_2) \in E)$

Graful orientat inițial



Metagraful CTC-urilor



Observația 3.10 Graful de condensare $G^{CTC} = (V^{CTC}, E^{CTC})$ asociat unui graf orientat $G = (V, E)$ este DAG. Într-adevăr existența unui circuit în G^{CTC} ar contrazice modul de construire a componentelor conexe (Dacă C_1 și C_2 ar aparține unui circuit din G^{CTC} , ar rezulta că de fapt C_1 și C_2 formează o singură componentă). Rezultă imediat putem construi o sortare topologică pentru G^{CTC} .

Algoritmul Kosaraju – Sharir folosește într-un mod ingenios observațiile 2.6, 3.8 și 3.10 pentru a determina componentele tare conexe. Mai exact, într-o primă fază este aplicat pur și simplu algoritmul de sortare topologică derivat din DFS pentru graful inițial. Vectorul *sort_top* rezultat nu este o sortare topologică a vârfurilor, dar are o proprietate interesantă.

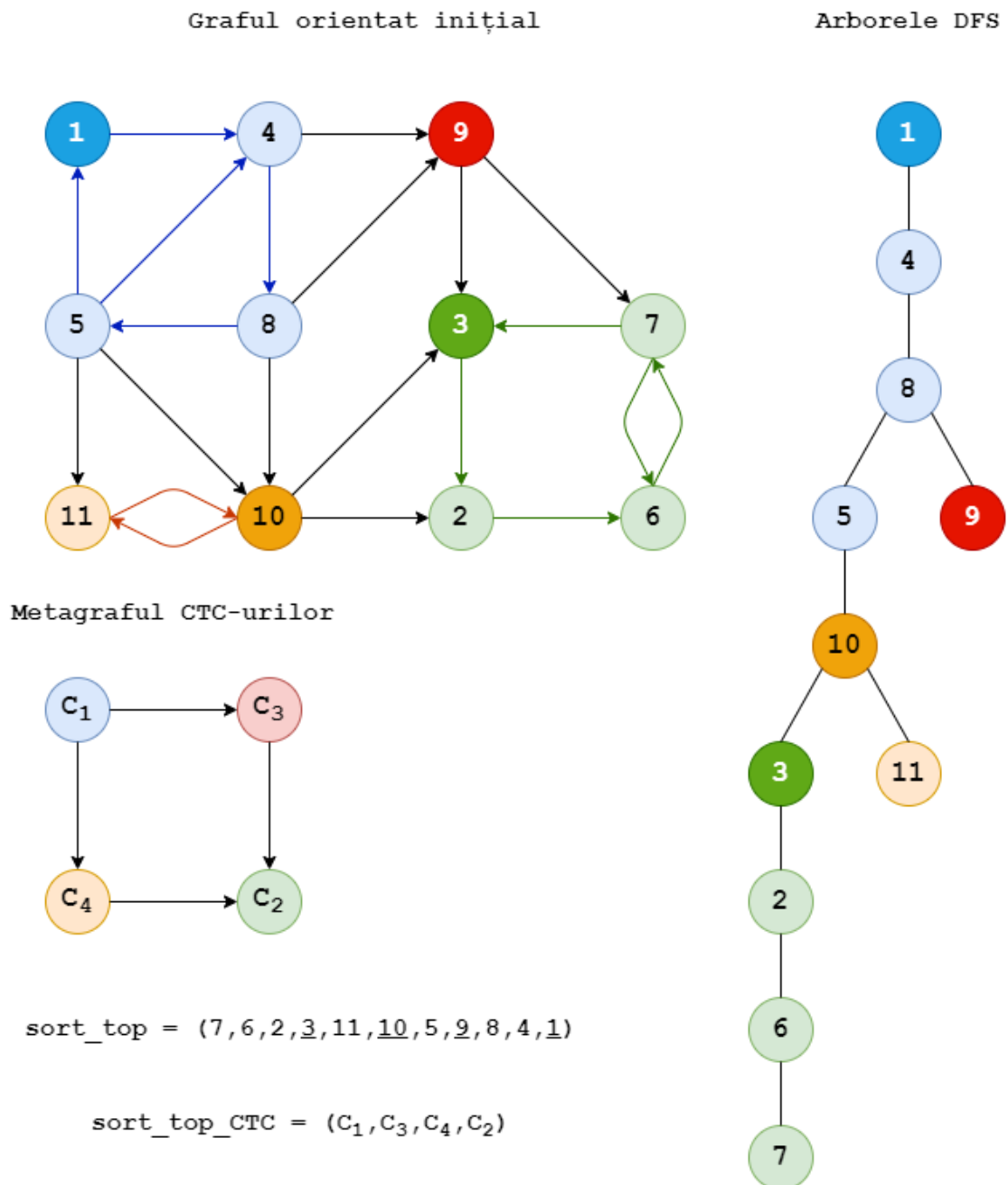
```

DFS_SORT_TOP(G, x)
1:   t_i[x] = ++timp_c
2:   pentru fiecare succesor y al lui x
3:   {
4:       dacă t_i[y] == 0
5:       {
6:           DFS_SORT_TOP(G, y)
7:       }
8:   }
9:   t_f[x] = ++timp_c
10:  sort_top[++nr_s_t] = x

```

Observația 3.11 Inițializând elementele vectorilor t_i și t_f cu valoarea 0 și apelând apoi procedura $DFS_SORT_TOP(G, x)$ pentru fiecare x cu $t_i[x] = 0$, obținem vectorul *sort_top* care are proprietatea că subșirul format din acei reprezentanți x ai fiecărei componente tare conexe cu $t_f[x]$ maxim constituie (după inversarea ordinii) o sortare topologică a grafului de condensare. Cu alte cuvinte, dacă x_1 este reprezentantul componentei tare conexe C_1 ($\forall x'_1, x'_1 \in V(C_1)$ avem: $t_f[x'_1] < t_f[x_1]$) și x_2 este reprezentantul componentei C_2 și $t_f[x_1] < t_f[x_2]$, atunci **nu există** drum de la x_1 la x_2 .

Observația 3.12 Rezultă imediat faptul că nu există drum de la **niciun vârf** aparținând C_1 la niciun vârf al lui C_2 . În fine, această ultimă afirmație este echivalentă cu inexistența drumurilor de la vârfurile lui C_2 la cele ale lui C_1 în graful transpus G^T .



Pornind de la observațiile 3.8, 3.11 și 3.12 deducem că, parcurgând graful transpus (procedura *DFS_CTC* de mai jos) în ordinea dată de sortarea topologică a grafului de condensare, vom obține componentele tare conexe ale grafului inițial. Într-adevăr, în exemplul de mai sus, parcurgând arcele lui G^T și pornind dintr-un vârf al lui C_1 vom vizita doar vârfuri din C_1 , apoi pornind dintr-un vârf al lui C_3 vom putea vizita vârfurile lui C_3 și pe

cele din C_1 , dar cum acestea din urmă au fost deja vizitate, le vom descoperi doar pe cele din C_3 etc.

```

DFS_CTC(G, x, nr_ctc)
1:   ctc[x] = nr_ctc
2:   pentru fiecare predecesor y al lui x
3:   {
4:       dacă ctc[y] == 0
5:       {
6:           DFS_CTC(G, y, nr_ctc)
7:       }
8:   }

```

Pentru a construi vectorul *ctc*, care în final va avea semnificația: $ctc[x]$ = numărul de ordine al componentei tare conexe din care face parte x , vor fi necesare inițializări pentru vectorii utilizați și apelarea celor două proceduri: *DFS_SORT_TOP* pentru fiecare vârf cu valoarea din t_i nemodificată și apoi *DFS_CTC* pentru fiecare vârf cu valoarea din *ctc* nemodificată, în ordinea sortării topologice a grafului de condensare (ordinea dată de vectorul *sort_top* după apelurile *DFS_SORT_TOP* și inversarea ordinii). Variabila *nr_ctc* reprezintă numărul de componente tare conexe găsite până la momentul curent.

Complexitatea **algoritmului Kosaraju – Sharir**, prezentat mai jos este $O(m+n)$ ca și în cazul sortării topologice), deoarece parcurgerea în adâncime a grafului transpus G^T are evident aceeași complexitate.

```
KOSARAJU_SHARIR(G)
1:   pentru fiecare vârf x al lui G
2:   {
3:       t_i[x] = t_f[x] = ctc[x] = 0
4:   }
5:   nr_s_t = 0
6:   timp = 0
7:   pentru fiecare vârf x al lui G
8:   {
9:       dacă t_i[x] == 0
10:      {
11:          DFS_SORT_TOP(G, x)
12:      }
13:  }
14:  INVERSARE_ORDINE(sort_top, nr_s_t)
15:  nr_ctc = 0
16:  pentru i = 1, nr_s_t
17:  {
18:      dacă ctc[sort_top[i]] == 0
19:      {
20:          nr_ctc++
21:          DFS_CTC(G, sort_top[i], nr_ctc)
22:      }
23:  }
```

4. Rezolvarea problemei 2-SAT.

Problema satisfacerii („**Boolean Satisfiability Problem**”) prescurtat: SAT constă în găsirea unui șir de valori logice (adevărat / fals) care pot fi atribuite variabilelor dintr-o expresie booleană astfel încât aceasta să aibă valoarea adevărat.

Folosind diverse formule, orice astfel de expresie poate fi redusă la **forma normal conjunctivă**, care constă dintr-o conjuncție de expresii (clauze) $E_1 \wedge E_2 \wedge \dots$, cu fiecare E_i de forma $x_{i1} \vee x_{i2} \vee \dots$, x_{i1}, x_{i2}, \dots fiind literali, adică variabile sau negații ale unor variabile.

Pentru problema generală nu se cunosc algoritmi polinomiali de rezolvare, în schimb cazul particular în care fiecare E_i are exact 2 literali, numit **2-SAT**, admite o soluție de complexitate $O(m+n)$, unde m reprezintă numărul clauzelor, iar n numărul variabilelor.

Observația 4.1 Dacă definim operația logică de implicație „ \Rightarrow ” conform tabelului de mai jos

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

observăm că $x \vee y \equiv (\neg x \Rightarrow y) \wedge (\neg y \Rightarrow x)$

x	y	$\neg x \Rightarrow y$	$\neg y \Rightarrow x$	$(\neg x \Rightarrow y) \wedge (\neg y \Rightarrow x)$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	1	1

Astfel, vom putea rescrie o expresie logică dată în forma normal conjunctivă sub forma unei conjuncții de implicații.

De exemplu:

$$\begin{aligned}
 & (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \equiv \\
 & (\neg x_1 \Rightarrow \neg x_2) \wedge (x_2 \Rightarrow x_1) \wedge (x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow \neg x_1) \wedge (x_1 \Rightarrow \neg x_2) \wedge (x_2 \Rightarrow \neg x_1) \\
 & \quad \wedge (\neg x_1 \Rightarrow x_3) \wedge (x_3 \Rightarrow x_1)
 \end{aligned}$$

În general o expresie logică dată sub forma normal conjunctivă corespunzătoare 2-SAT (conjuncție de disjuncții de câte 2 variabile) se va transforma astfel într-o conjuncție de implicații. Dacă expresia inițială este conjuncția a m clauze, atunci cea rezultată va fi conjuncția a $2m$ clauze.

Observația 4.2 Unei conjuncții de implicații cu n variabile: x_1, x_2, \dots, x_n și m implicații a câte două variabile sau negații ale acestora îi putem asocia un graf orientat $G = (V, E)$ în felul următor:

- fiecărei variabile $x_i, 1 \leq i \leq n$ îi vom asocia 2 vârfuri, etichetate cu $2i$ și respectiv $2i-1$;
- fiecărei implicații de forma $x_i \Rightarrow x_j$ îi vom asocia arcul $(2i, 2j)$, fiecărei implicații de forma $x_i \Rightarrow \neg x_j$ îi vom asocia arcul $(2i, 2j-1)$, fiecărei implicații de forma $\neg x_i \Rightarrow x_j$ îi vom asocia arcul $(2i-1, 2j)$, fiecărei implicații de forma $\neg x_i \Rightarrow \neg x_j$ îi vom asocia arcul $(2i-1, 2j-1)$.

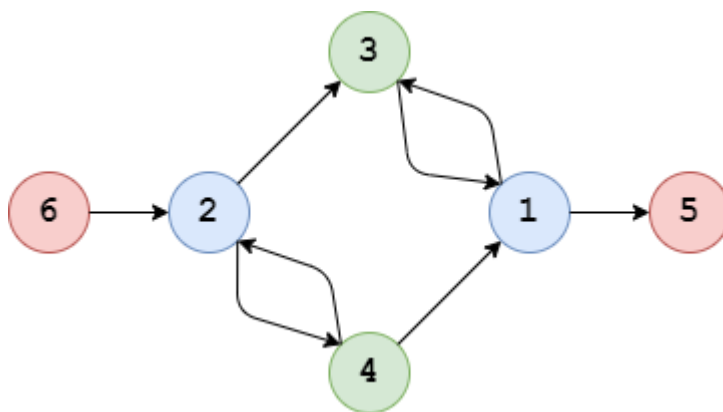
Graful $G = (V, E)$ astfel definit va avea $2n$ vârfuri și m arce.

Observația 4.3 Ținând cont de observațiile anterioare deducem că oricărei expresii logice aflate în forma normal conjunctivă corespunzătoare 2-SAT îi putem asocia un graf orientat parcurgând cei doi pași descriși mai sus (transformarea fiecărei disjuncții în câte două implicații și construirea grafului). Dacă expresia inițială are n variabile și este conjuncția a m clauze, atunci graful obținut va avea $2n$ vârfuri și $2m$ arce.

Exemplul 4.4 Pornind de la expresia $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3)$, aflată în forma normal conjunctivă corespunzătoare 2-SAT, obținem într-o primă etapă expresia logică echivalentă:

$$(\neg x_1 \Rightarrow \neg x_2) \wedge (x_2 \Rightarrow x_1) \wedge (x_1 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow \neg x_1) \wedge (x_1 \Rightarrow \neg x_2) \wedge (x_2 \Rightarrow \neg x_1) \\ \wedge (\neg x_1 \Rightarrow x_3) \wedge (x_3 \Rightarrow x_1)$$

și în final graful orientat corespunzător:



Lema 4.5 Dacă o expresie în forma normal conjunctivă corespunzătoare 2-SAT are soluție, atunci în graful orientat asociat nu există drumuri cu extremitatea inițială având asociată valoarea 1 (conform soluției problemei) și extremitatea finală având valoarea 0.

Demonstrație Putem folosi inducția matematică. Pentru drumurile de lungime 1, afirmația este evidentă: prezența unui arc de forma $1 \Rightarrow 0$ ar însemna că valoarea uneia dintre clauzele din expresia obținută prin trecerea la implicații este 0 și deci valoarea întregii expresii este 0.

Presupunem că afirmația este adevărată pentru orice drum de lungime strict mai mică decât k și demonstrăm rezultatul pentru drumurile de lungime k . Presupunem prin absurd că ar exista un drum de lungime k cu valoarea asociată primului vârf 1 și cea asociată ultimului vârf 0. Conform ipotezei de inducție valoarea asociată penultimului vârf este 1 (drumul până la penultimul vârf are lungimea $k - 1$), deci ultimul arc ar trebui să fie de forma $1 \Rightarrow 0$, ceea ce ar face ca valoarea expresiei să fie 0.

Teorema 4.6 O expresie în forma normal conjunctivă corespunzătoare 2-SAT cu variabilele x_1, x_2, \dots, x_n are soluție dacă și numai dacă în graful orientat asociat vârfurile $2i$ și $2i-1$ aparțin unor componente tare conexe diferite pentru $\forall i, 1 \leq i \leq n$.

Demonstrație „ \Rightarrow ” Presupunem prin absurd că două vârfuri $2i$ și $2i-1$ aparțin aceleiași componente tare conexe și că expresia are soluție. Cum $2i$ și $2i-1$ aparțin aceleiași componente tare conexe, rezultă că avem atât drum de la $2i$ la $2i-1$, cât și drum de la $2i-1$ la

2i. Dar cele două vârfuri au asociate valori diferite, prin urmare unul dintre drumuri contrazice lema 4.5.

„ \Leftarrow ” Pentru a demonstra implicația inversă vom descrie algoritmul prin care asociem valori pentru variabilele date. Vom determina componentele tare conexe ale grafului asociat expresiei aplicând algoritmul Kosaraju – Sharir și vom obține $ctc[i]$ = numărul de ordine al componentei tare conexe căreia îi aparține i pentru $\forall i, 1 \leq i \leq n$. Cum componentele sunt găsite conform algoritmului în ordinea dată de sortarea topologică a CTC, rezultă că dacă $ctc[i] < ctc[j]$, atunci în graf **nu** există un drum de la j la i .

Pentru $\forall i, 1 \leq i \leq n$, dacă $ctc[2i] < ctc[2i-1]$ (ceea ce înseamnă că putem avea sau nu drum de la $2i$ la $2i-1$, dar cu siguranță **nu avem** drum de la $2i-1$ la $2i$), vom asocia lui x_i valoarea 0. În caz contrar (dacă $ctc[2i] > ctc[2i-1]$), lui x_i îi vom asocia valoarea 1. O astfel de alegere ne garantează faptul că nu vom avea drumuri de forma „ $1 \Rightarrow 0$ ” cu extremitățile în vârfuri asociate aceleiași variabile. Pentru a demonstra că alegerea conform criteriului de mai sus ne garantează că soluția astfel construită satisface expresia dată, vom arăta că toate implicațiile sunt adevărate.

Presupunem prin absurd că există o implicație $x \Rightarrow y$ cu valoarea 1 asociată lui x și valoarea 0 asociată lui y . Modul de trecere de la disjuncții la implicații ne garantează că în graf vom avea atât arcul (x, y) , cât și arcul $(\neg y, \neg x)$, de unde rezultă că $ctc[x] \leq ctc[y]$ și $ctc[\neg y] \leq ctc[\neg x]$. Pe de altă parte, conform strategiei noastre de atribuire a valorilor, cum valoarea asociată lui y este 0, rezultă că $ctc[y] < ctc[\neg y]$. Punând cap la cap aceste trei inegalități obținem $ctc[x] \leq ctc[y] < ctc[\neg y] \leq ctc[\neg x]$, deci $ctc[x] < ctc[\neg x]$, ceea ce nu corespunde strategiei noastre de atribuire a valorilor (pornind de la această inegalitate, valoarea asociată lui x ar fi trebuit să fie 0). Am obținut o contradicție, de unde rezultă că toate implicațiile sunt adevărate, deci soluția construită conform strategiei de mai sus satisface expresia logică dată.

Pornind de la demonstrația de mai sus putem prezenta algoritmul de rezolvare a problemei 2-SAT. Plecând de la expresia dată, cu n variabile și m disjuncții, algoritmul construiește graful asociat G și, pe baza acestuia, găsește o soluție în cazul în care expresia are soluții, sau afișează un mesaj corespunzător în caz contrar:

```

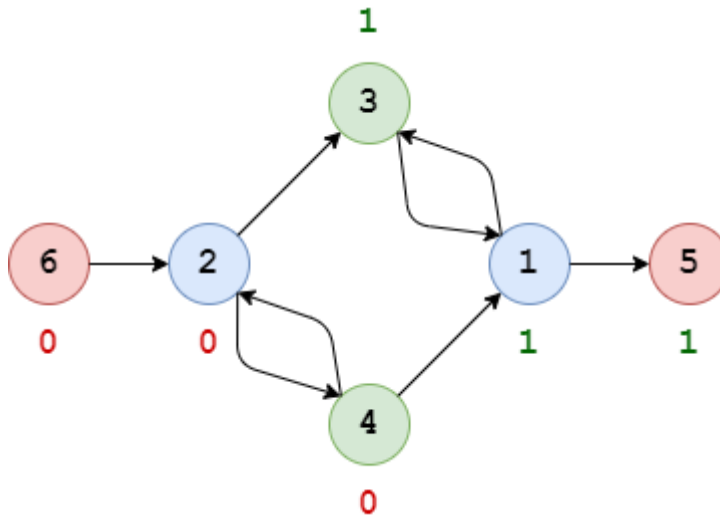
REZOLVARE_2-SAT( $G$ )
1:    $V(G) = \{1, 2, \dots, 2n\}$ 
2:   pentru fiecare disjuncție dată
3:   {
4:       adaugă arcele corespunzătoare
5:   }
6:   KOSARAJU_SHARIR( $G$ )
7:   există_sol = 1
8:   pentru  $i = 1, n$ 
9:   {
10:      dacă  $ctc[2i] == ctc[2i-1]$ 
11:      {
12:          există_sol = 0
13:      }
14:  }
15:  dacă există_sol
16:  {
17:      pentru  $i = 1, n$ 
18:      {
19:          dacă  $ctc[2i] < ctc[2i-1]$ 
20:          {
21:              sol[i] = 0
22:          }
23:          altfel
24:          {
25:              sol[i] = 1
26:          }
27:      }
28:  }
29:  altfel
30:  {
31:      scrie „nu există soluții”
32:  }

```

Algoritmul are complexitatea timp $O(m+n)$, reducându-se la aplicarea algoritmului

Kosaraju – Sharir pentru un graf orientat cu $2n$ vârfuri și $2m$ arce.

Exemplul 4.7 Prezintă mai jos un exemplu de aplicare a algoritmului pentru expresia $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3)$ care admite soluția $(x_1 = 0, x_2 = 0, x_3 = 0)$.



Componentele tare conexă ale grafului sunt C_1 , cu $V(C_1) = \{1,3\}$, C_2 , cu $V(C_2) = \{2,4\}$, C_3 , cu $V(C_3) = \{5\}$ și C_4 , cu $V(C_4) = \{6\}$. Sortarea topologică a grafului de condensare este (C_4, C_2, C_1, C_3) .

5. Probleme propuse:**5.1. Parcurgerea grafurilor orientate**

- [Petrica](#)
- [Multiplu](#)

5.2. Sortarea topologică

- [Sortare topologică \(Arhiva educațională Infoarena\)](#)
- [Project management](#)

5.3. Determinarea componentelor tare conexe

- [Componente tare conexe \(Arhiva educațională Infoarena\)](#)
- [Matroid](#)
- [Plan](#)

5.4. Rezolvarea problemei 2-SAT

- [2-SAT \(Arhiva educațională Infoarena\)](#)
- [Party](#)
- [The Door Problem](#)

Referințe

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – *Introduction to*

Algorithms (3rd edition)

Algorithms for Competitive Programming (<https://cp-algorithms.com/>)

Tim Roughgarden – *Stanford Algorithms*

(<https://www.youtube.com/@stanfordalgorithms2264>)