Tare-conexitate

Capitolul 4

- **Considerații teoretice**
- **Un algoritm ineficient**
- **Algoritmul plus-minus**
- **Algoritmul optim**
- **Rezumat**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

În cadrul acestui capitol vom introduce noțiunea de tare-conexitate şi de componentă tare-conexă. De asemenea, vom prezenta trei algoritmi care pot fi utilizați pentru a verifica tare-conexitatea unui graf şi pentru a determina componentele sale tare-conexe.

4.1. Considerații teoretice

Această secțiune este dedicată prezentării noțiunilor elementare legate de noțiunea de tare-conexitate. Vom prezenta noțiunile de graf tare-conex şi componentă tare-conexă şi vom descrie diferențele dintre noțiunea de conexitate şi cea de tare-conexitate. În final vom introduce noțiunea de graf al componentelor tare-conexe.

4.1.1. Grafuri orientate conexe

Cunoaştem deja faptul că un graf neorientat este conex dacă şi numai dacă există cel puțin un lanț între oricare două vârfuri ale sale. Noțiunea de conexitate poate fi extinsă (oarecum artificial) şi pentru grafurile orientate.

Practic, un graf orientat este conex dacă graful neorientat obținut prin eliminarea sensurilor arcelor (transformarea arcelor în muchii) este conex.

Chiar dacă această noțiune poate fi utilizată în unele cazuri (doar din considerente teoretice) ea este inutilă în practică. Nu are nici un rost să utilizăm proprietatea de conexitate a unui graf orientat deoarece, practic, ea nu se referă la graful propriu-zis, ci la un graf neorientat construit artificial.

4.1.2. Noțiunea de tare-conexitate

Totuşi, uneori am avea nevoie de "adevărata" conexitate a grafurilor orientate. Datorită faptului că noțiunea de ***conexitate*** a grafurilor orientate este deja "ocupată" de pro-

prietatea descrisă în subsecțiunea precedentă, un graf în care vor exista drumuri între oricare două vârfuri se va numi ***tare-conex***.

Aşadar, noțiunea de ***tare-conexitate*** a unui graf orientat se referă la existența a cel puțin un drum între oricare două dintre vârfurile sale.

În figura 4.1 este prezentat un graf tare-conex. Pentru a demonstra faptul că acest graf este tare-conex, vom prezenta care sunt drumurile între perechile de vârfuri:

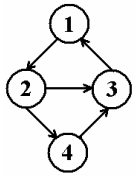


Figura 4.1: Un graf tare-conex

- 1 – 2
- 1 – 2 – 3
- 1 – 2 – 4
- 2 – 3 – 1
- 2 – 3
- 2 – 4
- 3 – 1
- 3 – 1 – 2
- 3 – 1 – 2 – 4
- 4 – 3 – 1
- 4 – 3 – 1 – 2
- 4 – 3

Graful din figura 4.2 nu este tare-conex (deşi este conex), deoarece există perechi de noduri între care nu există drumuri (de exemplu, nu există nici un drum de la nodul 4 la nodul 1; de fapt nu există nici un drum care să pornească de la nodul 4, deoarece acesta are gradul exterior 0).

Se poate spune că un graf orientat tare-conex este întotdeauna şi conex, dar afirmația reciprocă nu este întotdeauna adevărată. Aşadar, există grafuri orientate conexe care nu sunt tare-conexe (un exemplu în acest sens este graful din figura 4.2). Cu alte cuvinte, noțiunea de tare-conexitate este mai restrictivă decât cea de conexitate.

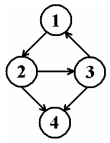


Figura 4.2: Un graf care nu este tare-conex

4.1.3. Componente tare-conexe

În cazul grafurilor neorientate există noțiunea de componentă conexă care este definită ca fiind un subgraf conex al unui graf. Avem componente conexe şi pentru grafurile orientate dar, din nou, această noțiune este practic inutilă.

Totuşi, în cazul în care un graf orientat nu este tare-conex, el are, cu siguranță, ***componente tare-conexe*** (chiar dacă sunt formate dintr-un singur vârf). Acestea sunt definite ca fiind *subgrafuri tare-conexe ale unui graf orientat*.

O componentă tare-conexă va fi ***maximală*** dacă şi numai dacă nu există nici o altă componentă tare-conexă care să o includă integral (cu alte cuvinte, nu face parte dintr-o componentă tare-conexă mai "mare").

De exemplu, graful din figura 4.2 conține două componente tare-conexe şi anume cea formată din nodurile 1, 2 şi 3 şi cea formată doar din nodul 4.

4.1.4. Graful componentelor tare-conexe

Se observă că pentru o componentă conexă maximală toate arcele adiacente nodurilor ei şi care nu fac parte din ea fie pleacă (toate!) din nodurile ei, fie ajung (toate!) la nodurile ei.

Dacă ar exista atât arce care *pleacă* de la nodurile componentei, cât şi arce care *ajung* la nodurile componentei, atunci componenta nu ar mai fi maximală.

Vom prezenta în continuare noţiunea de ***graf al componentelor tare-conexe***. Nodurile acestui graf reprezintă componentele tare-conexe maximale ale grafului "original". Un arc va pleca de la un nod corespunzător unei componente tare-conexe la un nod corespunzător unei alte componente tare-conexe, dacă în graful original există cel puţin un arc care pleacă de la un nod care face parte din prima componentă tare-conexă considerată, şi ajunge la un nod care face parte din cea de-a doua.

Să considerăm graful din figura 4.3. Acesta conţine patru componente tare-conexe pe care le vom identifica prin T_1, T_2, T_3, respectiv T_4. Componenta tare-conexă T_1 este formată din nodurile 1, 2 şi 3, T_2 este formată din nodurile 4, 6, 7 şi 9, T_3 este formată din nodurile 5 şi 8, iar T_4 este formată doar din nodul 10.

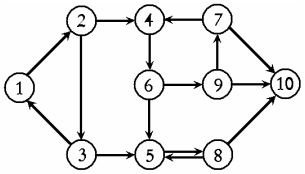


Figura 4.3: Un graf orientat

Aşadar, graful componentelor tare-conexe va conţine patru noduri (T_1, T_2, T_3 şi T_4). Datorită faptului că în graful considerat există un arc de la nodul 2 la nodul 4 (acestea fac parte din componentele T_1, respectiv T_2), vom avea un arc de la nodul T_1 la nodul T_2. Existenţa arcului de la nodul 3 la nodul 5 în graful original implică existenţa unui arc de la nodul T_1 la nodul T_3 în graful componentelor conexe. Există şi un arc de la nodul 6 la nodul 5, deci în graful componentelor tare-conexe vom avea un arc de la nodul T_2 la nodul T_3. Avem arce atât de la nodul 7 la nodul 10, cât şi de la nodul 9 la nodul 10 (ambele pleacă de la noduri ale componentei T_2 şi intră în singurul nod al componentei T_4); ca urmare vom avea un arc de la nodul T_2 la nodul T_4. Ultimul arc al grafului componentelor tare-conexe este de la nodul T_3 la nodul T_4 şi existenţa acestuia este cauzată de arcul de la nodul 8 la nodul 10 din graful original.

În concluzie, graful componentelor conexe va avea patru noduri şi cinci arce. El este prezentat în figura 4.4.

Este evident faptul că acest graf nu va fi niciodată ciclic deoarece componentele tare-conexe care s-ar afla pe un astfel de ciclu ar forma o componentă-conexă "mai mare", deci nu ar mai fi maximale.

În cazul în care graful original este conex, atunci şi cel al componentelor tare-conexe va fi conex, iar dacă graful original nu este conex, nici cel al componentelor nu va fi.

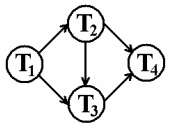


Figura 4.4: Graful componentelor tare-conexe corespunzător grafului din figura 4.3

4.2. Un algoritm simplu

Vom începe prezentarea modalităţilor de verificare a tare-conexităţii şi determinare a componentelor tare-conexe cu un algoritm simplu, dar neeficient. Acesta reprezintă, de fapt, o versiune modificată a algoritmului de determinare a drumurilor minime între toate perechile de vârfuri.

4.2.1. Preliminarii

Cel mai simplu algoritm cu ajutorul căruia se pot determina componentele tare-conexe ale unui graf se bazează pe definiția acestora. Practic, se construieşte o matrice D a drumurilor (un element D_{ij} al matricei va avea valoarea 1 în cazul în care există un drum de la nodul i la nodul j; se consideră că întotdeauna există un drum de la un nod la el însuşi, motiv pentru care toate elementele de pe diagonala principală a acestei matrice au valoarea 1) şi se verifică dacă aceasta conține elemente cu valoarea 0. În cazul în care există astfel de elemente, graful nu este tare-conex.

Folosind această matrice a drumurilor, putem determina şi componentele tare-conexe. Nodul i va face parte din aceeaşi componentă conexă ca şi toate numerele k pentru care valoarea D_{ik} este 1.

4.2.2. Verificarea tare-conexității

Aşa cum am afirmat anterior, pentru a verifica tare-conexitatea unui graf putem folosi o variantă a algoritmului *Floyd-Warshall*. Singura modificare care apare este înlocuirea condiției prin care se verifică dacă s-a găsit un drum mai scurt cu o expresie care exprimă faptul că dacă există un drum de la un nod i la un nod k şi există un drum de la nodul k la un nod j, atunci, cu siguranță, va exista un drum de la nodul i la nodul j.

Vom porni din nou de la matricea de adiacență A. Pentru început vom atribui elementelor de pe diagonala principală valoarea 1 pentru a arăta că există întotdeauna un drum de la un nod la el însuşi. După executarea algoritmului, matricea A va deveni matricea drumurilor. Algoritmul este următorul:

```
Subalgoritm VerificaTareConexitate(n,a,tareconex):
                                          { a – matricea de adiacență }
  d ← a                                       { matricea drumurilor }
  pentru i ← 1, n execută
    dii ← 1                          { elementele de pe diagonala principală }
  pentru k ← 1, n execută
    pentru i ← 1, n execută
      pentru j ← 1, n execută
        dacă dij = 0 atunci
          dij ← dik · dkj
        sfârşit dacă
      sfârşit pentru
    sfârşit pentru
  sfârşit pentru
  tareconex ← adevărat
  pentru i ← 1, n execută
    pentru j ← 1, n execută
      dacă dij = 0 atunci
```

```
        tareconex ← fals
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
  returnează tareconex
sfârşit subalgoritm
```

Expresia $d_{ij} \leftarrow d_{ik} \cdot d_{kj}$ va avea valoarea 1 dacă şi numai dacă atât d_{ik}, cât şi d_{kj}, au valoarea 1. Aşadar, această expresie poate fi folosită pentru a arăta faptul că dacă există un drum de la un nod *i* la un nod *k* şi există un drum de la nodul *k* la un nod *j*, atunci va exista şi un drum de la nodul *i* la nodul *j*.

4.2.3. Determinarea componentelor tare-conexe

După construirea matricei drumurilor putem identifica foarte simplu componentele tare-conexe. Vom porni cu primul nod şi vom stabili că toate nodurile *j*, pentru care valorile elementelor $D_{1,j}$ şi $D_{j,1}$ sunt ambele 1, fac parte din aceeaşi componentă tare-conexă. Vom marca toate aceste noduri pentru a arăta faptul că a fost identificată deja componenta tare-conexă din care fac parte.

În continuare, la fiecare pas, vom alege un nod *i* care nu face parte din nici o componentă conexă şi vom stabili că toate nodurile *j*, pentru care valorile elementelor D_{ij} şi D_{ji} sunt ambele 1, fac parte din aceeaşi componentă tare-conexă. Vom marca şi aceste noduri pentru a arăta faptul că a fost identificată deja componenta tare-conexă din care fac parte.

Procedeul va continua până în momentul în care nu mai este identificat nici un nod care nu face parte dintr-o componentă tare-conexă.

Datorită faptului că elementele de forma D_{ii} au valoarea 1, la fiecare pas va fi marcat cel puţin un nod (în cel mai defavorabil caz se va identifica o componentă tare-conexă formată dintr-un singur nod). Ca urmare, vom parcurge nodurile în ordine crescătoare şi vom verifica, pentru fiecare, dacă a fost inclus într-o componentă conexă la un pas anterior. Evident, pentru primul nod vom fi întotdeauna în situaţia în care el nu a fost inclus.

Vom păstra un tablou de valori booleene *c* care va arăta dacă pentru un nod a fost identificată componenta din care face parte. Iniţial toate valorile vectorului vor fi *fals*, iar după terminarea execuţiei algoritmului toate valorile sale vor fi *adevărat*.

Prezentăm în continuare versiunea în pseudocod a algoritmului descris:

```
Algoritm DeterminaComponenteTareConexe(n,d):
                                          { d – matricea drumurilor }
  nr ← 0
  pentru i ← 1, n execută:
    c_i ← fals
```

```
  pentru i ← 1, n execută:
    dacă nu ci atunci
      nr ← nr + 1
      pentru j ← 1, n execută:
        dacă dij = 1 şi dji = 1 atunci
          scrie j, "face parte din componenta", nr
          cj ← adevărat
        sfârşit dacă
      sfârşit pentru
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

În cadrul algoritmului anterior am numerotat componentele tare-conexe în ordinea în care au fost identificate şi pentru fiecare am afişat componenta din care face parte.

Este evident faptul că, în cazul în care graful este tare-conex, toate nodurile vor fi marcate încă de la primul pas; va fi identificată o singură componentă tare-conexă (întregul graf) şi nu se va mai executa nici un alt pas.

4.2.4. Analiza complexității

În continuare vom analiza ordinul de complexitate al algoritmului de verificare a tare-conexității unui graf orientat, precum şi cel al algoritmului de determinare a componentelor tare-conexe.

În primul rând, algoritmul de determinare a matricei drumurilor este o versiune fără modificări esențiale a algoritmului *Floyd-Warshall*; ordinul de complexitate al acestui algoritm este $O(n^3)$.

Aceasta este, de fapt, cea mai costisitoare operație efectuată, celelalte realizându-se în timp liniar sau pătratic:

- Inițializarea matricei drumurilor cu valorile matricei de adiacență implică n^2 atribuiri, deci ordinul de complexitate al operației este $O(n^2)$.
- Atribuirea valorii 1 pentru elementele de pe diagonala principală a matricei drumurilor implică o simplă traversare a acesteia, deci ordinul de complexitate al operației este $O(n)$.
- Verificarea existenței unui element cu valoarea 0 în matricea drumurilor implică parcurgerea tuturor elementelor acesteia, deci ordinul de complexitate al operației este $O(n^2)$.
- Determinarea unei componente tare-conexe implică parcurgerea unei linii (şi a unei coloane) a matricei drumurilor, deci ordinul de complexitate al operației este $O(n)$.

- În total vor fi determinate cel mult n componente tare-conexe (în cazul cel mai defavorabil avem n componente tare-conexe formate fiecare dintr-un singur nod), deci ordinul de complexitate al operației de determinare a acestora este $O(n^2)$.

În concluzie, dacă folosim acest algoritm ordinul de complexitate al operațiilor de verificare a tare-conexității şi identificare a componentelor tare-conexe este $O(n^3)$.

4.3. Algoritmul plus-minus

Deşi este foarte uşor de implementat, algoritmul prezentat anterior nu oferă performanțe satisfăcătoare. În cadrul acestei secțiuni vom prezenta un algoritm bazat pe traversarea grafurilor orientate a cărui ordin de complexitate este $O(n^2)$. Aşadar, algoritmul va fi cu un ordin de mărime mai performant.

Vom începe cu prezentarea noțiunii de graf *transpus*, datorită faptului că acest algoritm necesită construirea unui astfel de graf. În continuare vom descrie algoritmul şi apoi îi vom analiza complexitatea.

4.3.1. Graful transpus

O caracteristică fundamentală a grafurilor orientate este aceea că fiecărui arc îi este asociată o direcție (un arc pleacă de la un nod şi ajunge la altul). ***Graful transpus*** este obținut prin simpla modificare a direcțiilor tuturor arcelor.

Aşadar, pentru fiecare arc (i, j) al grafului original vom introduce în graful transpus un arc (j, i). În figura 4.5 este prezentat graful transpus corespunzător grafului din figura 4.3.

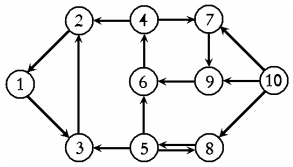


Figura 4.5: Graful transpus corespunzător grafului din figura 4.3

De obicei, pentru a nota graful transpus al unui graf G se utilizează notația G^T. Acesta va avea, întotdeauna, acelaşi număr de noduri şi acelaşi număr de arce cu graful G.

În cazul în care este cunoscută o reprezentare a unui graf, determinarea reprezentării grafului transpus nu ridică dificultăți deosebite.

4.3.2. Prezentarea algoritmului

Vom prezenta direct algoritmul utilizat pentru determinarea componentelor tare-conexe. Acesta poate fi modificat foarte uşor pentru a realiza verificarea tare-conexității. Practic, în momentul în care identificăm o a doua componentă tare-conexă, ştim cu siguranță că graful nu este tare-conex.

Vom folosi o abordare similară celei utilizate în cazul determinării componentelor tare-conexe folosind matricea drumurilor. În momentul în care determinăm o componentă tare-conexă vom marca toate nodurile care o constituie.

Aşadar, la fiecare pas vom porni cu un nod care nu a fost marcat la nici unul dintre paşii anteriori (inițial, vom alege nodul 1). Vom realiza o parcurgere în adâncime (se

poate utiliza şi parcurgerea în lățime) începând din nodul ales şi vom marca toate nodurile parcurse cu `'+'`. Practic, aceste noduri sunt cele la care se poate ajunge pornind din nodul ales.

În continuare, pornind de la acelaşi nod, vom realiza o nouă parcurgere dar, de această dată, vom utiliza graful transpus. Toate nodurile parcurse vor fi marcate cu `'-'`. Datorită faptului că în acest graf direcțiile arcelor sunt inversate, aceste noduri reprezintă toate nodurile de la care se poate ajunge la nodul ales (în graful original, nu în cel transpus).

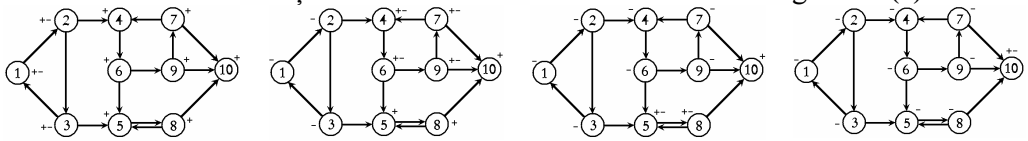
Componenta tare-conexă este formată din toate nodurile care au fost marcate atât cu `'+'`, cât şi cu `'-'`. Din toate aceste noduri vom putea ajunge la nodul ales şi apoi, de la acesta vom putea ajunge la oricare dintre aceste noduri, deci există cel puțin un drum de la oricare astfel de nod la oricare alt astfel de nod.

Pornind de la nodul 1, pentru graful din figura 4.3 vor fi marcate cu `'+'` toate nodurile. Totuşi, în graful transpus (figura 4.5), pornind de la acelaşi nod vom marca cu `'-'` doar nodurile 1, 2 şi 3. Aşadar, prima componentă tare-conexă va fi formată din aceste noduri. Acest pas este ilustrat în figura 4.6(a).

Primul nod nemarcat este 4; pornind de la acest nod vom marca cu `'+'` nodurile 4, 5, 6, 7, 8, 9 şi 10. Folosind graful transpus, nodurile marcate cu `'-'` vor fi 1, 2, 3, 4, 6, 7 şi 9. Ca urmare, cea de-a doua componentă tare-conexă este formată din nodurile 4, 6, 7 şi 9 (cele marcate cu ambele semne). Acest pas este ilustrat în figura 4.6(b).

Următorul nod ales este 5; nodurile marcate cu `'+'` sunt 5, 8 şi 10, iar cele marcate cu `'-'` sunt 1, 2, 3, 4, 5, 6, 7, 8 şi 9. A treia componentă conexă va fi formată din nodurile 5 şi 8, iar pasul este ilustrat în figura 4.6(c).

A rămas doar nodul 10. De la acesta nu pleacă nici un arc în graful original, deci doar el va fi marcat cu `'+'`. Deşi vor fi marcate cu `'-'` toate nodurile, ultima componentă tare-conexă va conține doar acest nod. Pasul este ilustrat în figura 4.6(d).



(a) **(b)** **(c)** **(d)**

Figura 4.6: Paşii algoritmului plus-minus

Vom prezenta în cele ce urmează versiunea în pseudocod a algoritmului descris:

```
Subalgoritm DF_PlusMinus(k,G,marcaj):
                                          { k – nodul curent }
                                          { G – graful }
                                   { marcaj – marcajul curent }
                                    { adevărat reprezintă '+' }
                                          { fals reprezintă '-' }
```

```
  dacă marcaj atunci
    plus_k ← adevărat
  altfel
    minus_k ← adevărat
  sfârşit dacă
  pentru toți vecinii i ai lui k execută:
    dacă nu vizitat_i atunci
      vizitat_i ← adevărat
      DF_PlusMinus(i,G,marcaj)
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm

Algoritm PlusMinus(G,n):
                                                    { G – graful }
                                                { n – numărul nodurilor }
  nr ← 0
  pentru i ← 1, n execută:
    v_i ← fals
  sfârşit pentru
  pentru i ← 1, n execută:
    dacă nu v_i atunci
      nr ← nr + 1
      pentru j ← 1, n execută:
        vizitat_j ← fals
        plus_j ← fals
      sfârşit pentru
      DF_PlusMinus(i,G,adevărat)
      pentru j ← 1, n execută:
        vizitat_j ← fals
        minus_j ← fals
      sfârşit pentru
      DF_PlusMinus(i,G^T,fals)
      pentru j ← 1, n execută:
        dacă plus_j şi minus_j atunci
          scrie j, "face parte din componenta", nr
          v_j ← adevărat
        sfârşit dacă
      sfârşit pentru
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

Modul prin care sunt identificați vecinii nodului curent depinde de reprezentarea aleasă pentru graf. Dacă avem la dispoziție o matrice de adiacență, vom parcurge linia corespunzătoare, dacă avem la dispoziție doar lista muchiilor, va trebui să o parcurgem în întregime, ceea ce este ineficient, iar dacă avem liste de vecini sau o listă a succesorilor, este foarte simplu să parcurgem vecinii unui element.

4.3.3. Analiza complexității

Practic, la fiecare pas există posibilitatea să parcurgem întregul graf; aşadar, operațiile efectuate în timpul unui pas au ordinul de complexitate $O(m)$, unde m este numărul arcelor din graf. Datorită faptului că putem avea până la n componente tare-conexe, ar putea părea că ordinul de complexitate al algoritmului prezentat este $O(m \cdot n)$.

Totuşi, această limită nu poate fi atinsă; se poate demonstra matematic că, de fapt, ordinul de complexitate este $O(n^2)$. Demonstrația necesită cunoştințe avansate de matematică, motiv pentru care nu o vom reda aici.

Datorită faptului că, în marea majoritate a cazurilor, numărul componentelor tare-conexe este relativ mic, se vor executa relativ puțini paşi. Ca urmare, pentru cazul mediu, ordinul de complexitate este $O(m + n)$.

4.4. Algoritmul optim

În cadrul acestei secțiuni vom prezenta algoritmul optim de determinare a componentelor tare-conexe ale unui graf. Acesta se bazează pe algoritmul plus-minus, dar foloseşte o caracteristică suplimentară, şi anume timpul final al unei parcurgeri *DF*.

Pentru început vom prezenta noțiunea de timp final, apoi vom prezenta algoritmul şi, în încheiere, îi vom analiza complexitatea.

4.4.1. Timpii finali ai parcurgerii DF

Timpul final corespunzător unui nod în graf este dat de momentul la care s-a terminat prelucrarea nodului respectiv în timpul parcurgerii. Aşadar, primul nod (cel de la care începe parcurgerea) va avea timpul final n; primul său succesor va avea timpul final $n - 1$; în cazul în care acest succesor are, la rândul său, succesori nevizitați încă (diferiți de 1), primul său succesor va avea timpul final $n - 2$; în caz contrar, timpul final al celui de-al doilea succesor al nodului de la care începe parcurgerea va avea timpul final $n - 2$.

Timpii finali corespunzători nodurilor sunt determinați foarte simplu dacă păstrăm o variabilă a cărei valoare este incrementată în momentul în care este vizitat un nou nod.

Vă prezentăm în continuare versiunea în pseudocod a unui algoritm care realizează o parcurgere în adâncime cu memorarea timpilor finali.

```
Subalgoritm DF_Final(k,G,timp):
                                          { k – nodul curent G – graful }
                                  { timp – variabilă transmisă prin referință }
  pentru toți vecinii i ai lui k execută:
    dacă nu vizitat_i atunci
      vizitat_i ← adevărat
      DF_Final(i,G)
      timp ← timp + 1
    sfârşit dacă
  sfârşit pentru
  final_i ← timp
sfârşit subalgoritm
```

După executarea acestui algoritm, vectorul `final` va conține timpii finali corespunzători nodurilor. Acesta reprezintă o permutare a mulțimii $\{1, 2, \ldots, n\}$. În continuare vom avea nevoie de ordonarea nodurilor în funcție de timpii finali (în ordine descrescătoare). Datorită particularității acestui vector, această ordonare poate fi realizată în timp liniar. Timpul final al unui nod va indica poziția acestuia în vectorul `ordine` (acesta va conține ordinea nodurilor). Dacă timpul final va fi k, atunci poziția în vectorul ordine va fi $n - k + 1$.

Algoritmul de determinare a vectorului `ordine` este următorul:

```
Subalgoritm Stabilire_Ordine(final,G,timp):
                                        { final – vectorul timpilor finali }
                                                  { G – graful }
                                  { timp – variabilă transmisă prin referință }
  pentru i ← 1, n execută:
    dacă nu vizitat_i atunci
      k ← final_i
      ordine_{n-k+1} ← i
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Să considerăm, din nou, graful din figura 4.3. În cazul în care succesorii unui nod vor fi luați în considerare în ordinea crescătoare a numerelor care îi identifică în graf, atunci nodurile vor fi parcurse în ordinea: 1, 2, 3, 5, 8, 10, 4, 6, 9, 7.

Timpii finali corespunzători celor zece noduri sunt prezentați în tabelul 4.1.

Nod	*Timp final*	*Nod*	*Timp final*
1	10	6	3
2	9	7	1
3	8	8	6
4	4	9	2
5	7	10	5

Tabelul 4.1: Timpii finali corespunzători nodurilor grafului din figura 4.3

4.4.2. Determinarea optimă a componentelor tare-conexe

După determinarea timpilor finali, vom efectua parcurgeri în adâncime pe graful transpus. Vom începe cu vârful care are cel mai mare timp final şi, după determinarea componentei tare-conexe din care face parte acesta, vom lua în considerare nodul cu cel mai mare timp final pentru care nu a fost determinată componenta tare-conexă din care face parte. Procedeul va continua până la determinarea tuturor componentelor.

Diferența esențială față de algoritmul plus-minus este faptul că parcurgerile pe graful transpus se realizează în ordinea inversă a timpilor finali, motiv pentru care orice nod va fi parcurs înaintea succesorilor săi.

Pentru graful din figura 4.3, vom începe cu nodul `1`. Prin parcurgerea în adâncime pe graful transpus vom vizita nodurile 1, 2 şi 3 care formează prima componentă tare-conexă.

În continuare alegem nodul 5 care este nodul cu cel mai mare timp final dintre cele care nu fac parte din prima componentă conexă. În urma parcurgerii vom vizita nodurile 5 şi 8, care formează cea de-a doua componentă tare-conexă.

Următorul nod ales va fi 10, care formează singur o componentă tare-conexă deoarece nu există nici un arc care pleacă de la acest nod, deci nu putem vizita nici un alt nod pornind de la el.

În final, vom alege nodul 4 şi vom vizita nodurile 4, 6, 7 şi 9, care formează ultima componentă tare-conexă.

În acest moment nu mai există noduri pentru care nu a fost determinată componenta conexă din care fac parte, motiv pentru care algoritmul se încheie.

Vom prezenta acum versiunea în pseudocod a algoritmului optim de determinare a componentelor tare-conexe (subalgoritmii `DF_Final` şi `Stabilire_ordine` au fost prezentați anterior, deci nu vor apărea în continuare):

```
Subalgoritm DF(k,G,nr):
                                                  { k – nodul curent }
                                                  { G – graful }
                                          { nr – numărul componentei tare-conexe }
  pentru toți vecinii i ai lui k execută:
    dacă nu vizitat_i atunci
      vizitat_i ← adevărat
      scrie j, "face parte din componenta", nr
      DF(i,G)
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

```
Algoritm Tare_Conexe_Optim(G,n):
                                                    { G – graful }
                                                { n – numărul nodurilor }
  nr ← 0
  timp ← 0
  pentru i ← 1, n execută:
    vizitat_i ← fals
  sfârşit pentru
  pentru i ← 1, n execută:
    dacă nu vizitat_i atunci
      DF_Final(i,G,timp)
    sfârşit dacă
  sfârşit pentru
  pentru i ← 1, n execută:
    vizitat_i ← fals
  sfârşit pentru
  pentru i ← 1, n execută:
    k ← ordine_i
    dacă nu vizitat_k atunci
      DF(k,G^T)
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

Se observă că, atât în timpul primei parcurgeri (pe graful original), cât şi în timpul celei de-a doua (pe graful transpus), fiecare nod este vizitat o singură dată, ceea ce creşte cu mult performanţele algoritmului.

4.4.3. Analiza complexităţii

Datorită faptului că realizăm doar două parcurgeri ale unor grafuri (cel original şi cel transpus) care au n noduri şi m arce, ordinul de complexitate al algoritmului este $O(m + n)$.

Celelalte operaţii efectuate nu afectează acest ordin de complexitate, deoarece se efectuează mai rapid (liniar). Astfel, operaţiile de determinare a timpilor finali şi ordinii în care sunt luate în considerare nodurile la a doua parcurgere, precum şi cele folosite pentru iniţializarea vectorului care indică dacă un nod a fost sau nu vizitat au, toate, ordinul de complexitate $O(n)$.

În concluzie, algoritmul optim de determinare a componentelor tare-conexe ale unui graf are ordinul de complexitate $O(m + n)$, unde n este numărul nodurilor, iar m este numărul arcelor.

4.5. Rezumat

În cadrul acestui capitol am prezentat noțiunile de tare-conexitate a unui graf orientat şi componentă tare-conexă a unui graf. De asemenea, am descris trei algoritmi care pot fi utilizați pentru verificarea tare-conexității sau pentru determinarea componentelor tare-conexe.

Primul dintre algoritmi, simplu dar ineficient, se bazează pe algoritmul *Floyd-Warshall*. Cel de-al doilea este cunoscut sub numele de algoritmul *plus-minus* şi oferă performanțe mult mai bune. În final, am prezentat algoritmul optim de determinare a componentelor tare-conexe.

Pentru a prezenta ultimii doi algoritmi am introdus noțiunea de *graf transpus* al unui graf. De asemenea, pentru a prezenta algoritmul optim am introdus noțiunea de *timp final* al unui nod în cadrul unei parcurgeri *DF*.

Am efectuat o analiză a complexităților tuturor celor trei algoritmi. Din aceste analize rezultă clar că cel de-al treilea algoritm oferă cele mai bune performanțe.

4.6. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor care implică deținerea unor cunoştințe referitoare la tare-conexitatea grafurilor orientate vă sugerăm să încercați să implementați algoritmi pentru:

1. crearea matricei de adiacență a grafului transpus pe baza matricei de adiacență a grafului original;
2. crearea listei de arce a grafului transpus pe baza listei de arce a grafului original;
3. crearea listelor de vecini ale grafului transpus pe baza listelor de vecini ale grafului original;
4. crearea şirului succesorilor pentru graful transpus pe baza şirului succesorilor pentru graful original;
5. determinarea timpilor finali folosind matricea de adiacență pentru reprezentarea grafului;
6. determinarea timpilor finali folosind listele de succesori pentru reprezentarea grafului;
7. determinarea componentelor tare-conexe folosind fiecare dintre cei trei algoritmi descrişi, precum şi diferite metode pentru reprezentarea grafurilor.

4.7. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoştințele suplimentare necesare sunt minime.

4.7.1. Străzi

Descrierea problemei
Într-un oraş există *N* pieţe, identificate prin numere cuprinse între 1 şi *N*. Între pieţe există un număr total de *M* străzi, fiecare stradă legând două pieţe. Datorită creşterii traficului, consiliul oraşului a decis că pentru fiecare stradă din oraş se va stabili un sens de circulaţie. Aşadar, fiecare stradă va deveni cu sens unic de circulaţie. Primarul trebuie să verifice dacă stabilirea sensurilor este corectă. Se ştie că, înaintea stabilirii sensului de circulaţie, din fiecare piaţă se putea ajunge în oricare altă piaţă folosind străzile existente. Stabilirea sensurilor este corectă dacă, după precizarea sensurilor de circulaţie, se poate ajunge din fiecare piaţă în oricare alta. În cazul în care stabilirea sensurilor nu este corectă, primarul va trebui să precizeze două pieţe *x* şi *y* cu proprietatea că nu se poate ajunge în piaţa *x* pornind din piaţa *y* sau nu se poate ajunge în piaţa *y* pornind din piaţa *x*.

Date de intrare
Prima linie a fişierului de intrare **`STRAZI.IN`** conţine numărul *N* al pieţelor reţelei şi numărul *M* al străzilor. Aceste numere vor fi separate printr-un spaţiu. Fiecare dintre următoarele *M* linii va conţine câte două numere întregi *x* şi *y* cu semnificaţia: există o stradă care uneşte direct pieţele *x* şi *y* şi sensul de circulaţie a fost stabilit de la piaţa *x* spre piaţa *y*.

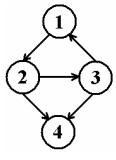
Date de ieşire
În cazul în care stabilirea sensurilor este corectă, fişierul de ieşire **`STRAZI.OUT`** va conţine o singură linie pe care se va afla mesajul `DA`. În caz contrar, prima linie a fişierului de ieşire va conţine mesajul `NU`, iar cea de-a doua linie va conţine două numere întregi *x* şi *y*, separate printr-un spaţiu, cu proprietatea că nu se poate ajunge în piaţa *x* pornind din piaţa *y* sau nu se poate ajunge în piaţa *y* pornind din piaţa *x*.

Restricţii şi precizări
- $1 \le N \le 500$;
- $1 \le M \le 5000$;
- există cel mult o stradă între oricare două pieţe.

Exemplu

`STRAZI.IN`	**`STRAZI.OUT`**

`4 5`
`1 2`
`2 3`
`2 4`
`3 1`
`3 4`
	`NU`
	`1 4`



Timp de execuţie: 1 secundă/test

4.7.2. Antene

Descrierea problemei
O companie are la dispoziție N antene pentru comunicații, identificate prin numere cuprinse între 1 şi N. Din nefericire, nu este posibilă comunicația între oricare două antene. Mai mult, dacă un mesaj poate fi transmis de la o antenă x la o antenă y, nu este sigur că un mesaj poate fie transmis de la antena y la antena x.

Compania doreşte să identifice grupurile de antene în cadrul cărora este posibilă comunicația directă sau indirectă între oricare două antene care fac parte din acelaşi grup. Numărul total al grupurilor trebuie să fie minim.

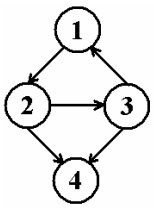
Date de intrare
Prima linie a fişierului de intrare **ANTENE.IN** conține numărul N al antenelor şi numărul M al legăturilor care pot fi stabilite între antene. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele M linii va conține câte două numere întregi x şi y cu semnificația: există posibilitatea de a transmite un mesaj de la antena x la antena y.

Date de ieşire
Fişierul de ieşire **ANTENE.OUT** va conține un număr de linii egal cu numărul grupurilor. Pe fiecare dintre aceste linii se vor afla numerele de ordine ale antenelor care fac parte dintr-un grup. Aceste numere vor fi separate prin spații.

Restricții şi precizări
- $1 \le N \le 500$;
- $1 \le M \le 5000$;
- există posibilitatea ca un grup să fie format dintr-o singură antenă;
- elementele unui grup pot fi scrise în orice ordine;
- grupurile pot fi scrise în orice ordine;
- o antenă poate face parte dintr-un singur grup.

Exemplu

ANTENE.IN	ANTENE.OUT
4 5 1 2 2 3 2 4 3 1 3 4	1 2 3 4



Timp de execuție: 1 secundă/test

4.7.3. Zvonuri

Descrierea problemei
Într-o şcoală se află N elevi, identificaţi prin numere cuprinse între 1 şi N. Fiecare elev are mai mulţi prieteni apropiaţi cărora le comunică orice zvon imediat după ce îl află. Directorul doreşte să cunoască numărul minim al grupurilor de elevi care pot fi formate astfel încât un zvon transmis oricărui elev dintr-un anumit grup să ajungă la toţi elevii din grupul respectiv. Dacă un elev x comunică imediat un zvon unui elev y, atunci nu este obligatoriu ca elevul y să-i comunice imediat un zvon elevului x.

Date de intrare
Prima linie a fişierului de intrare **ZVONURI.IN** conţine numărul N al elevilor. Fiecare dintre următoarele linii va conţine datele referitoare la un elev. Primul număr de pe o astfel de linie reprezintă numărul p al prietenilor apropiaţi, iar următoarele p numere reprezintă numerele de ordine ale prietenilor apropiaţi. Numerele de pe o linie sunt separate printr-un spaţiu; prima dintre aceste linii corespunde elevului identificat prin numărul `1`, cea de-a doua linie corespunde elevului identificat prin numărul `2` etc.

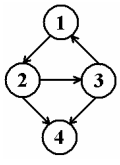
Date de ieşire
Fişierul de ieşire **ZVONURI.OUT** va conţine o singură linie pe care se va afla un singur număr care reprezintă numărul minim al grupurilor care pot fi formate respectându-se condiţia precizată.

Restricţii şi precizări
- $1 \le N \le 500$;
- suma numerelor prietenilor apropiaţi este mai mică decât 5000;
- pot exista elevi care nu au nici un prieten apropiat;
- există posibilitatea ca un grup să fie format dintr-un singur elev;
- un elev poate face parte dintr-un singur grup.

Exemplu

ZVONURI.IN
```
4
1 2
2 3 4
2 1 4
0
```

ZVONURI.OUT
```
2
```



Timp de execuţie: 1 secundă/test

4.8. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare şi va fi analizată complexitatea algoritmului prezentat.

4.8.1. Străzi

Putem privi rețeaua de străzi din oraşe ca fiind un graf orientat în care nodurile reprezintă piețele, iar arcele reprezintă străzile care le unesc. Sensurile arcelor sunt date de sensurile de circulație stabilite de consiliu.

În aceste condiții, problema se reduce la verificarea tare-conexității unui graf orientat. Deoarece nu este necesară determinarea componentelor tare-conexe este suficient să verificăm dacă toate nodurile se află în aceeaşi componentă tare-conexă cu primul nod.

Datorită numărului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaşte poziția la care încep succesorii săi în această listă. Evident, poziția la care se termină succesorii unui nod este dată de poziția la care încep succesorii următorului nod (cel care are numărul de ordine mai mare cu 1).

Pentru aceasta vom realiza o parcurgere în adâncime (depth first - DF) pornind de la primul nod şi vom verifica dacă au fost parcurse toate nodurile. Dacă am găsit un nod care nu este parcurs, înseamnă că nu se poate ajunge din prima piață în piața corespunzătoare nodului respectiv, deci am găsit două piețe care nu satisfac condiția impusă.

În continuare, vom verifica dacă din oricare piață se poate ajunge în piața corespunzătoare primului nod. Pentru aceasta vom determina graful transpus şi vom efectua o nouă parcurgere în adâncime, pornind tot din nodul corespunzător primei piețe. De această dată, dacă am găsit un nod care nu a fost parcurs, vom şti că din piața corespunzătoare nodului respectiv nu se poate ajunge în prima piață deci, şi în acest caz, am găsit două piețe care nu satisfac condiția impusă.

Dacă în urma celor două parcurgeri nu am identificat nici o pereche de piețe care nu satisface condiția, atunci graful este tare-conex şi planul consiliului este corect.

Analiza complexității

Citirea datelor de intrare implică citirea celor M muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este **O(M)**. În paralel cu citirea se determină numărul succesorilor fiecărui nod, ordinul de complexitate al acestei operații fiind tot **O(M)**.

Pentru a determina pozițiile de început ale succesorilor nodurilor, este necesară o parcurgere a vectorului care conține numerele succesorilor, ordinul de complexitate al acestei operații fiind **O(N)**.

Pentru a determina şirul succesorilor este necesară o nouă parcurgere a muchiilor; ca urmare, ordinul de complexitate al acestei operații este **O(M)**.

Operația de parcurgere în adâncime a unui graf are ordinul de complexitate **O(M)** deoarece este luat în considerare fiecare succesor.

Pentru a verifica dacă au fost parcurse toate nodurile va trebui să le parcurgem, deci această operație are ordinul de complexitate **O(N)**.

Determinarea grafului transpus este echivalentă (ca timp de execuție) cu determinarea grafului inițial; vom avea nevoie de un timp de ordinul **O(M)** pentru a determina numărul succesorilor nodurilor, de un timp de ordinul **O(N)** pentru determinarea pozițiilor de început şi de un timp de ordinul **O(M)** pentru determinarea propriu-zisă a listei succesorilor.

În continuare se realizează o nouă parcurgere în adâncime al cărei ordin de complexitate este tot **O(M)**, urmată de o nouă verificare care se realizează într-un timp de ordinul **O(N)**.

Scrierea datelor în fişierul de ieşire implică scrierea unui şir şi, eventual, a două numere, aşadar ordinul de complexitate al acestei operații este **O(1)**.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(M) + O(M) + O(N) + O(M) + O(M) + O(N) + O(M) + O(N) + O(M) + O(M) + O(1) = \mathbf{O(M + N)}$.

4.8.2. Antene

Putem privi antenele ca fiind nodurile unui graf orientat în care există o muchie de la un nod la altul dacă poate fi transmis un semnal de la antena corespunzătoare primului nod la antena corespunzătoare celui de-al doilea nod.

În aceste condiții problema se reduce la determinarea componentelor tare-conexe ale unui graf orientat.

Datorită numărului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaşte poziția la care încep succesorii săi în această listă. Evident, poziția la care se termină succesorii unui nod este dată de poziția la care încep succesorii următorului nod (cel care are numărul de ordine mai mare cu 1).

Folosind această reprezentare a grafului vom putea aplica fără dificultăți deosebite oricare dintre algoritmii rapizi de determinare a componentelor tare-conexe. Evident, datorită numărului relativ mare de noduri, algoritmul simplu (cel care are ordinul de complexitate $O(N^3)$) nu va putea fi folosit, în schimb algoritmul pătratic va furniza soluții în timpul pus la dispoziție.

Analiza complexității

Citirea datelor de intrare implică citirea celor M muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este **O(M)**. În paralel cu citirea se determină numărul succesorilor fiecărui nod, ordinul de complexitate al acestei operații fiind tot **O(M)**.

Pentru a determina pozițiile de început ale succesorilor nodurilor, este necesară o parcurgere a vectorului care conține numerele succesorilor, ordinul de complexitate al acestei operații fiind **O(N)**.

Pentru a determina şirul succesorilor este necesară o nouă parcurgere a muchiilor; ca urmare, ordinul de complexitate al acestei operaţii este **O(M)**.

În continuare vom presupune că am folosit algoritmul eficient de determinare a componentelor tare-conexe. Aşadar acestea sunt găsite într-un timp de ordinul **O(M + N)**.

Componentele tare-conexe sunt scrise în fişierul de ieşire pe măsura determinării lor. Datorită faptului că vor fi scrise exact *N* numere, ordinul de complexitate al operaţiei de scriere a datelor de ieşire este **O(N)**.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(M) + O(M) + O(N) + O(M) + O(M + N) + O(N) = \mathbf{O(M + N)}$.

4.8.3. Zvonuri

Putem privi elevii ca fiind nodurile unui graf orientat în care există o muchie de la un nod la altul, dacă un zvon este comunicat de către elevul corespunzător primului nod la nodul corespunzător celui de-al doilea nod.

În aceste condiţii problema se reduce la determinarea numărului componentelor tare-conexe ale unui graf orientat.

Datorită numărului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaşte poziţia la care încep succesorii săi în această listă. Evident, poziţia la care se termină succesorii unui nod este dată de poziţia la care încep succesorii următorului nod (cel care are numărul de ordine mai mare cu 1).

Folosind această reprezentare a grafului, vom putea aplica fără dificultăţi deosebite oricare dintre algoritmii rapizi de determinare a componentelor tare-conexe. Pe parcursul determinării acestor componente le vom număra. Evident, datorită numărului relativ mare de noduri, algoritmul simplu (cel care are ordinul de complexitate $O(N^3)$) nu va putea fi folosit. Totuşi, algoritmul plus-minus, care rulează în timp pătratic poate fi folosit.

Analiza complexităţii

Citirea datelor de intrare implică citirea succesorilor nodurilor grafului (care sunt folosiţi pentru crearea listei celor *M* arce), aşadar ordinul de complexitate al acestui algoritm este **O(M)**.

Pentru a determina poziţiile de început ale succesorilor nodurilor, este necesară o parcurgere a vectorului care conţine numerele succesorilor, ordinul de complexitate al acestei operaţii fiind **O(N)**.

Pentru a determina şirul succesorilor este necesară o nouă parcurgere a muchiilor; ca urmare, ordinul de complexitate al acestei operaţii este **O(M)**.

În continuare vom presupune că am folosit algoritmul eficient de determinare a componentelor tare-conexe. Aşadar acestea sunt găsite într-un timp de ordinul **O(M + N)**.

Datele de ieşire constau într-un singur număr, deci operaţia de scriere a rezultatului în fişierul de ieşire are ordinul de complexitate **O(1)**.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(M) + O(N) + O(M) + O(M + N) + O(1) = \mathbf{O(M + N)}$.