Acta et Commentationes, Sciences of Education, nr. 4(26), 2021 ISSN 1857-0623
p. 109-118 E-ISSN 2587-3636

CZU: 004.02 DOI: 10.36120/2587-3636.v26i4.109-118

METODOLOGIA UTILIZĂRII PARCURGERII ÎN ADÂNCIME (DFS) LA REZOLVAREA PROBLEMELOR INFORMATICE DE CONCURS

Sergiu CORLAT, Universitatea de Stat din Moldova
https://orcid.org/0000-0002-5471-2957
Roxana MEHRYAR-RAD, Liceul Teoretic Orizont
https://orcid.org/0000-0001-6205-7940

Rezumat. Parcurgerea în adâncime (Depth First Search, DFS) este o metodă de parcurgere a grafului bine cunoscută. În diverse situații, în particular în problemele de competiții, necesitatea aplicării eficiente a parcurgerii DFS este mascată de condițiile problemei, sau structurile de date utilizate. În aceste cazuri problema este fie identificarea necesității utilizării DFS în soluție, fie adaptarea DFS la condițiile particular ale problemei. Prezentul articol reprezintă un studiu al problemelor de concurs reprezentative, cu identificarea soluțiilor DFS, însoțit de soluțiile informatice. Rezultatele vor fi utile tuturor celor pasionați de informatica competitive, dar și celor care studiază algoritmica grafurilor.

Cuvinte cheie: parcurgere în adâncime, reprezentare a grafului, listă de adiacență, componente conexe, arbori.

METHODOLOGY OF USING DEPTH-FIRST SEARCH (DFS) FOR SOLVING PROGRAMMING COMPETITION PROBLEMS

Abstract. Depth First Search (DFS) is a well-known method for traversing a graph. In various situations, particularly in competition problems, the need for an effective application of DFS is masked by the conditions of the problem, or the data structures used. In these cases, the problem is either to identify the need to use DFS in the solution or to adapt the DFS to the particular conditions of the problem. This article is a study of representative competition problems, with the identification of DFS solutions, accompanied by IT solutions. The results will be useful to all those who are passionate about competitive programming, but also to those who study graph algorithms.

Keywords: depth-first search, graph representation, adjacency list, conex component, tree.

1. Introducere

Algoritmul parcurgerii DFS este suficient studiat, are o descriere compactă și o implementare elegantă, clară, de cele mai multe ori descrisă în formă recursivă. Înțelegerea și acceptarea operației de parcurgere de către studenți sau elevi nu prezintă o problemă, în special dacă analiza metodei este însoțită de descrierile pe pași, în baza unor exemple concrete. Pentru ilustrare în procesul de predare poate fi folosită orice implementare grafică DFS pe grafuri aleatorii, serii de imagini sau fișiere .gif animate pentru exemple definite apriori.

În tr-un graf $G(V, E)$ trecerea de la un vârf la altul este posibilă doar pe muchii, iar dacă graful este orientat – pe muchii în direcția permisă.

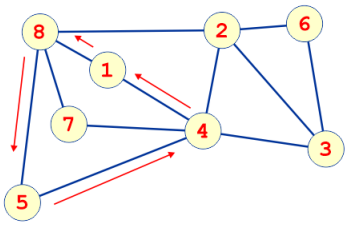

Figura 1. Mișcarea ciclică în timpul parcurgerii unui graf

Pentru a evita crearea ciclurilor în timpul parcurgerii grafului (Figura 1), se adăugă fiecărui vârf un indicator „de stare”. Se consideră că nodul se poate afla în două stări: „vizitat” - dacă a fost folosit în parcurgere, și „nevizitat” - în caz contrar. O asemenea parcurgere poate fi descrisă în mod recursiv:

Fie că suntem în vârful **v** „nevizitat”.

a. Se marchează v ca nod vizitat
b. Pentru toate vârfurile **u**, de la **1** la n, vecine pentru **v**: dacă **u** – nevizitat, se reia parcurgerea din vârful **u**.
c. Se afișează **v**. // sau este fixat într-un oricare alt mod faptul că nodul a fost parcurs

Acest mod de parcurgere a grafului se numește parcurgere în adâncime, deoarece tinde să ajungă la vârfurile cele mai îndepărtate de vârful din care începe și abia apoi să caută alte căi de parcurgere. Implementarea poate fi realizată într-o funcție recursivă, conform descrierii anterioare

În următoarea implementare se va considera că descrierea grafului **G(V, E)** este dată de matricea de adiacență **a[][]**, în calitate de structură auxiliară este folosit tabloul de stări **b[]**, iar nodurile parcurse se afișează în output-ul standard în ordinea parcurgerii.

```
int DFS (int s)
{
   b[s] = 1;
   for(int i = 1; i <= n; i++)
      if(a[s][i] != 0 && b[i] == 0) DFS(i);
   printf("%d ", s);
   return 0;
}
```

Situația se schimbă în mod radical în cazul în care parcurgerea DFS apare în calitate de subproblemă într-o problemă de natură complexă, mascată de condiții, restricții sau structuri de date propuse spre utilizare. Identificarea în asemenea cazuri a soluțiilor, care au la baza lor parcurgerea DFS este o problemă cu soluții particulare pentru fiecare dintre clasele de probleme formulate.

2. Structuri de date eficiente

Structurile de date matriceale, utilizate tradițional pentru descrierea grafurilor devin puțin eficiente în cazul entităților de dimensiuni mari, unde numărul de vârfuri depășește

10^3- în acest caz doar o singură parcurgere a structurii presupune efectuarea a circa 10^6 operații, ceea ce se estimează la o secundă de lucru a sistemului de calcul, or, de cele mai multe ori, restricțiile impuse nu depășesc în total două secunde.

Soluția, care combină eficiența parcurgerilor (DFS, BFS) cu utilizarea rațională a memoriei, devine în acest caz lista de adiacență, cunoscută și sub numele de listă de vecini. În limbajul clasic al structurilor de date ea reprezintă un tablou unidimensional de pointeri către liste unidirecționale (stive), alocate dinamic [1, p. 18]. Implementarea directă în cod program a unei asemenea structuri necesită resurse importante de timp și efort: descrierea structurii (tipului) pentru implementarea listelor, definirea operațiilor de adăugare a elementelor în listă și de parcurgere a listei, testarea funcționării corecte a acestora [2, p. 59].

Implementarea eficientă poate fi însă realizată în cazul utilizării librăriilor limbajelor de programare, care conțin descrierile funcțiilor de implementare a operațiilor de adăugare sau parcurgere a elementelor unei liste alocate dinamic, în particular – a unei stive. De exemplu, pentru limbajul C++ pot fi utilizate librăriile vector, bits/stdc++ sau oricare altele care conțin descrierile operațiilor enumerate anterior [3, p. 133].

Declararea și inițializarea structurii de date pentru reprezentarea grafului poate fi realizată de exemplu, de secvența de directive și declarații:

```
#include <bits/stdc++.h>
    using namespace std;
vector<int>adj[200009];
```

3. DFS și determinarea componentelor de conexitate cu proprietăți specifice

Modelul standard de utilizare repetată a parcurgerii DFS până la parcurgerea tuturor vârfurilor grafului permite identificarea tuturor componentelor conexe ale acestuia, ceea ce face posibil, ulterior, determinarea componentelor cu cele mai diverse proprietăți: cel mai mare număr de vârfuri, cel mai mic număr de vârfuri, cel mai mare număr de muchii, etc.

În continuare va fi analizată o problemă aparent simplă, care presupune determinarea numărului de componente conexe ale unui graf neorientat, care au proprietatea de a fi cicluri. Problema prototip este Cyclic Components [4]. Problema are indicele de complexitate pe platforma de programare egal cu 1500, ceea ce corespunde nivelului mediu de complexitate pentru concursurile internaționale sau nivelului înalt de complexitate pentru concursurile naționale.

Enunț: fie dat un graf neorientat $G(V, E)$ cu $|V| = n$. Se cere să se determine câte componente conexe din G sunt componente ciclice.

În terminologia problemei se consideră că o secvență de muchii formează un ciclu simplu, izolat, dacă vârfurile, care determină secvența de muchii, pot fi renumerotate astfel,

încât sfârșitul muchiei curente din secvență coincide cu începutul muchiei următoare, pentru oricare pereche de muchii vecine. Suplimentar, fiecare vârf, care face parte din ciclu aparține în graf la exact două muchii.

În continuare, o componentă de conexitate sa consideră componentă ciclică, dacă ea este un ciclu simplu, izolat.

Restricții: $|V| = n \leq 2 \times 10^5$, $|E| = m \leq 2 \times 10^5$, timpul de execuție nu va depăși 2 secunde.

Analiza: se va face abstracție de dimensiunea problemei – modul de a reprezenta eficient graful a fost deja descris.

Observație: într-un ciclu simplu, respectiv într-o componentă ciclică – toate puterile vârfurilor sunt egale cu doi.

Concluzie: la parcurgerea componentei conexe curente se verifică prezența vârfurilor cu putere diferită de 2.

Pentru a ajunge la un model eficient de verificare a puterilor vârfurilor, acestea vor fi calculate nemijlocit în procesul de citire a datelor inițiale. Pentru aceasta va fi utilizat un tablou unidimensional puteri, în care elementul puteri[i] va indica numărul de vecini ai vârfului i.

Baza soluției problemei o vaconstitui o funcție DFS similară, care, pe lângă parcurgerea componentei conexe, va verifica prezența vârfurilor cu putere diferită de 2.

Pe final, după parcurgerile repetate atât timp cât în graful $G(V, E)$ mai există vârfuri necercetate, vor fi numărate toate parcurgerile, care corespund cerințelor problemei.

Implementare:

```
#include <bits/stdc++.h>
    using namespace std;
int n, m;
vector<int>adj[200009]; // graful
int b[200009], puteri[200009]; //tablourile de stări și puteri
int flag, u, v;

void dfs(int u)
{
    b[u] = 1;
    if (puteri[u]!= 2) flag = 0;
    for (int i = 0; i < adj[u].size(); i++)
                                //pentru fiecare vecin a lui u
if (!b[adj[u][i]]) dfs(adj[u][i]);
                               //dacă nu e vizitat - start DFS
}

int main()
```

```
  {
      cin >> n >> m;     // citire numar varfuri si muchii
      while (m--)
      {
          cin >> u >> v;      // citire muchie curenta
          puteri[u]++; puteri[v]++;     // actualizare puteri
          adj[u].push_back(v),adj[v].push_back(u);
                                        //si lista vecini
      }
      int ans = 0;
      for (int i = 1; i <= n; i++)
          if (!b[i])     // daca nodul curent nu este vizitat
              {
                  flag = 1; dfs(i);
                  if (flag) ans++;
              }
      cout << ans;
      return 0;
}
```

4. DFS în probleme de numărare

În calitate de problemă prototip a fost analizată problema Edgy Trees [5]. Este o problemă cu nivel de complexitate care variază de la „înalt” pentru concursuri de nivel național până la „mediu” pentru concursurile internaționale de programare. Indicele intern de complexitate pe platformă este 1500[1].

Enunț: este dat un arbore cu n vârfuri. fiecare din cele $n-1$ muchii ale arborelui este vopsită în roșu sau negru. Suplimentar, se dă un număr întreg k. Se cercetează o consecutivitate de k vârfuri. Se spune că secvența de vârfuri $[a_1, a_2, \dots, a_k]$ este bună, dacă satisface condițiile:

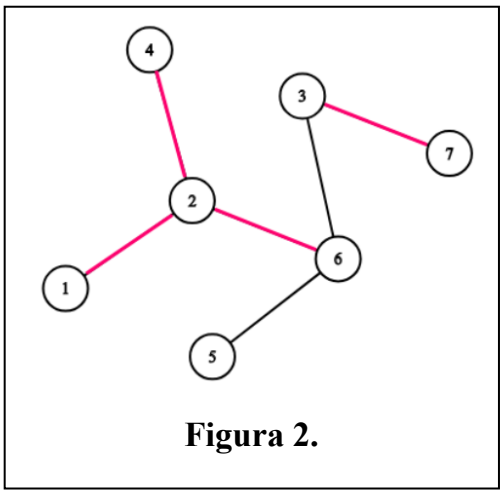

Figura 2.

- Se parcurge un drum oarecare în arbore cu Începutul în a_1 și sfârșitul în a_k, posibil revenind în unele vârfuri sau parcurgând repetat unele muchii
- Drumul începe în a_1 apoi se trece pe cel mai scurt drum de la a_1 la a_2, apoi în mod similar se trece de la a_2 la a_3 și tot așa, până nu se parcurge cel mai scurt drum de la a_{k-1} la a_k.

[1] La momentul analizei problemei, 12.2021

- Dacă în procesul parcurgerii acestui drum se trece pe cel puțin o muchie neagră, secvența se consideră bună.

La o analiză a arborelui din Figura 2 se observă că pentru $k = 3$ se pot obține secvențele bune [1, 4, 7], [5, 5, 3] și [2, 3, 7]. În același exemplu secvențele [1, 4, 6], [5, 5, 5], [3, 7, 3] nu sunt bune.

În total există n^k secvențe distincte de lungime k. Trebuie să se calculeze câte dintre ele sunt bune (în contextul definiției de mai sus). Deoarece rezultatul poate fi foarte mare, se va afișa modulo $10^9 + 7$.

Analiza problemei

Restricțiile impuse sunt: $2 \leq n \leq 10^5, 2 \leq k \leq 100$, timpul de execuție nu va depăși 2 secunde.

Se poate observa că, deoarece graful reprezintă un arbore, drumurile între oricare pereche de vârfuri sunt unice. Apoi, se observă, că deoarece numărul total de vârfuri în arbore este cunoscut, se poate calcula numărul de secvențe bune ca diferența dintre numărul total de secvențe posibile (n^k) și numărul de secvențe „rele".

Acest din urmă număr poate fi calculat, dacă se observă, că la excluderea muchiilor negre din arborele inițial, acesta se descompune într-un set de componente de conexitate, fiecare formată numai din muchii roșii, astfel secvențele de vârfuri conectate sunt rele! Mai mult, pentru a vizita un nod din altă componentă de conexitate neapărat se va folosi o muchie neagră – acestea sunt punțile de legătură între componentele colorate în roșu.

Deci, dacă este analizată o componentă conexă roșie R_i, numărul secvențelor rele în această componentă va fi $|R_i|^k$.

Prin urmare, numărul secvențelor bune va fi $n^k - \sum_{i=1}^{s} |R_i|^k$, s fiind numărul de componente conexe obținute după eliminarea muchiilor negre.

Pentru prima etapă a rezolvării – determinarea secvențelor conexe după eliminarea muchiilor negre – se va folosi procedura DFS. Tot cu ajutorul ei se va determina și numărul de vârfuri în fiecare componentă de conexitate.

O ultimă problemă de rezolvat este structura de date pentru stocarea grafului inițial. Date fiind restricțiile pentru numărul de vârfuri $n \leq 10^5$, utilizarea structurilor clasice tip matrice de adiacență, matrice de incidență va impune crearea unor structuri cu până la 10^{10} elemente, ceea ce presupune un timp de prelucrare care va depăși mult restricțiile de timp ale problemei. În consecință va fi folosit modelul compact de reprezentare a grafului, descris anterior.

Implementare:

```
#include <bits/stdc++.h>
    using namespace std;
#define M 1000000007
int b[100005];
```

```
vector<int>gr[100005]; // lista de vecini
int cnt, u, v, negru, k, n;
long long bad = 0, all = 1, ans;
void dfs(int x) // DFS
{
    cnt++;
    b[x] = 1;
    for(int i = 0; i < gr[x].size(); i++)
            if(!b[gr[x][i]]) dfs(gr[x][i]);
}

int main()
{
    cin >> n >> k;
    for(int i = 1; i < n; i++)
    {
        cin >> u >> v >> negru;
        if(!negru)                //se adauga muchia in graf
            gr[u].push_back(v), gr[v].push_back(u);
    }
    for(int i = 1; i <= n; i++)
        if(!b[i])
        {
            cnt = 0; ans = 1;
            dfs(i); //cu DFS calculam puterea componentei conexe
            for(int j = 0; j < k; j++) ans = (ans * cnt) % M;
            bad = (bad + ans) % M;
// secvetse rele in componenta conexa
        }
       for(int i = 0; i < k; i++) all = (all * n) % M;
//calcul n la puterea k
   cout << (all - bad + M) % M;
return 0;
}
```

5. DFS în probleme de optimizare

În calitate de problemă prototip apare problema Mouse Hunt [6]. La fel ca problema precedentă, se plasează în categoria probleme pentru concursuri internaționale, cu un indice de complexitate pe platformă – 1700.

Enunț (adaptat):

Într-un hotel sunt n camere și un șoarece. Proprietarii au decis să instaleze capcane în unele numere, pentru a-l prinde. Instalarea capcanei în numărul cu indicele i costă c_i unități monetare. Numerele de hotel sunt numerotate cu indici de la 1 la n.

Șoarecele aleargă permanent. Dacă în secunda t șoarecele se afla în numărul i , atunci în secunda $t + 1$ va trece direct în numărul cu indicele a_i (dacă $i = a_i$ atunci șoarecele va rămâne în același număr). Alergarea începe în secunda 0. Dacă șoarecele se află într-un număr cu capcană, el nimerește în mod sigur în aceasta. Totul ar simplu, dacă s-ar cunoaște din care număr începe alergarea șoarecele, dar el se poate afla în secunda 0 în orice număr al hotelului - de la 1 la n.

Se cere să se determine care este investiția minimă în instalarea capcanelor, astfel încât să se garanteze capturarea șoarecelui, indiferent de locul de start al alergării lui.

Restricții: $1 \leq n \leq 2 \times 10^5, \quad 1 \leq c_i \leq 10^4, \quad 1 \leq a_i \leq n$ Restricții timp de execuție – 2 secunde

Analiza problemei

Observația 1: structura de date în care se desfășoară acțiunea reprezintă un graf orientat.

Observația 2: din fiecare nod al grafului pornește un singur arc, dar pot exista mai multe arce, care se sfârșesc în același nod.

Observația 3: pornind dintr-un nod oarecare, șoarecele, la un moment dat, va intra într-o buclă direcționată, în care își va continua la nesfârșit mișcarea.

Concluzia 1: graful va conține una sau mai multe bucle direcționate care nu se intersectează între ele[2].

Concluzia 2: amplasarea capcanelor pe lanțuri care „intră" în bucle nu este rațională, este suficientă amplasarea unei singure capcane în fiecare buclă, alegând pentru aceasta nodul, în care costul amplasării este minimal.

Problema care rămâne de rezolvat este – cum să fie identificate buclele în graful orientat? – și s rezolvă prin aplicarea repetată a operației DFS pornind de la noduri necercetate încă. Fiecare parcurgere nouă fie va degenera prin „intrare" într-o buclă deja identificată, fie va identifica o nouă buclă. Operația se va repeta până la epuizarea nodurilor necercetate în vectorul de stare.

La fel ca în studiile precedente, structura de date optimală în restricțiile impuse rămâne a fi lista de vecini, realizată printr-un vector.

Implementare:

```
#include<bits/stdc++.h>
    using namespace std;
int n, nod, cost, ans, a[200010], c[200010];
vector<int> gr[200010];
```

[2] Afirmația poate fi demonstrată, de exemplu, prin reducere la absurd

```
bool bucla = false, b[200010];
void dfs(int x)
    {
    b[x] = true;
    for(int i = 0; i < gr[x].size(); i++)
    {
        if(b[gr[x][i]]) // bucla!
        {
            nod = gr[x][i]; cost = c[nod];
            bucla = true;
            break;
        }
        dfs(gr[x][i]);
    }
    if(bucla) cost = min (cost, c[x]);

    if(x == nod)  //iesire din bucla
    {
        ans += cost;
        bucla = false;
    }
    return;
}
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> c[i];
    for(int i = 1; i <= n; i++)
{ cin >> a[i]; gr[i].push_back(a[i]);}
    for(int i = 1; i <= n; i++)
if(!b[i]) dfs(i);
    cout << ans;
return 0;
}
```

6. Concluzii

Parcurgerea DFS este un instrument eficient de cercetare a structurilor neliniare de date tip graf. Eficiența ei este determinată nu doar de algoritmul de parcurgere, dar și de structurile de date primare, utilizate. Utilizarea combinată a tablourilor de pointeri la stive distincte după numărul de vârfuri în graf permite evitarea verificărilor suplimentare sau analiza tuturor vârfurilor.

Adăugarea la funcția de parcurgere DFS a instrucțiunilor de verificare a unor condiții suplimentare nu modifică complexitatea generală a algoritmului, dar permite obținerea unor rezultate suplimentare, cum ar fi puterea componentei conexe, este sau nu aceasta un ciclu și multe altele.

Studierea modelelor de utilizare DFS urmează a începe de la situațiile simple, în care DFS nu este mascat de condiții sau enunțuri specifice, cu trecere treptată către probleme de complexitate medie și înaltă. Pe parcurs urmează să fie create și dezvoltate abilitățile de stocare eficientă a datelor, realizare a operațiilor specifice grafurilor.

Setul de probleme analizat prezintă o selecție minimală de probleme, suficiente pentru studierea aplicațiilor DFS în rezolvarea problemelor informatice.

Bibliografie

1. GIBBONS, A. *Algorithmic Graph Theory*. NY: Cambridge University Press, 1999.
2. АХО, А.; ХОПКРОФТ, Д.; УЛЬМАН, Д. *Построение и анализ вычислительных алгоритмов*. Москва: Мир, 1979.
3. CERCHEZ, E.; ȘERBAN, M. *Programarea în limbajul C/C++ pentru liceu*. Vol. IV, Iași: Polirom, 2013.
4. https://codeforces.com/problemset/problem/977/E
5. https://codeforces.com/problemset/problem/1139/C
6. https://codeforces.com/problemset/problem/1027/D