Capitolul 2
METODE DE CĂUTARE ŞI PROGRAMARE

2.1. Căutarea în lățime

În teoria grafurilor, **breadth-first search (BFS)** este un algoritm de căutare în grafuri, care începe cu vârful rădăcină şi explorează toate nodurile vecine. Apoi, pentru fiecare dintre aceste noduri se explorează nodurile vecine încă necercetate, ş.a.m.d.., până când scopul a fost atins.

BFS este o metodă de căutare, care ţinteşte extinderea şi examinarea tuturor nodurilor unui graf, cu scopul de a găsi soluția.

Din punct de vedere al algoritmului, toate nodurile „fii" obținute prin expansiunea unui nod sunt adăugate într-o „coadă" de tipul FIFO (First In First Out). În implementările tipice, nodurile care nu au fost încă examinate de către vecinii corespunzători sunt plasate într-un „recipient" (asemănător unei cozi sau unei liste de legătură), numit „deschis", iar odată examinați sunt plasați în „recipientul" „închis".

2.1.1. Algoritmul

1. Introducerea nodului rădăcină în coadă.
2. Extragerea unui nod din capătul listei şi examinarea acestuia.
   - Dacă elementul căutat se identifică cu acest nod, se renunță la căutare şi se returnează rezultatul.
   - Altfel, se plasează toți succesorii (nodurile „fii") (neexaminați încă) acestui nod la sfârşitul „cozii" (acesta în cazul în care există)
3. Dacă „coada" este goală, fiecare nod al grafului a fost examinat - se renunță la căutare şi se întoarce la „not found".
4. Repetă începând cu Pasul 2.

2.1.2. Implementarea C++

În continuare este implementarea algoritmului de mai sus, unde „neexaminații până în momentul de față" sunt gestionați de către tabloul părinte.

Fie structura *struct* şi structura de noduri

```
struct Vertex {
        ...
        std::vector<int> out;
        ...
};
```

```
std::vector<Vertex> graph(vertices);
bool BFS(const std::vector<Vertex>& graph, int start, int end) {
  std::queue<int> next;
  std::vector<int> parent(graph.size(), 0);
  parent[start] = -1;
  next.push(start);
  while (!next.empty()) {
    int u = next.front();
    next.pop();
    if (u == end) return true;
    for (std::vector<int>::const_iteratorj=graph[u].out.begin();
j != graph[u].out.end(); ++j)
      {
      // Look through neighbors.
      int v = *j;
      if (parent[v] == 0) {
        // If v is unvisited.
        parent[v] = u;
        next.push(v);
      }
    }
  }
  return false;
}
```

Sunt stocați părinții fiecărui nod, de unde se poate deduce drumul.

2.1.3. Complexitate şi optimalitate

Complexitate în spațiu. Având în vedere faptul că toate nodurile descoperite până la momentul de față trebuiesc salvate, complexitatea în spațiu a breadth-first search este $O(|V|+|E|)$, unde $|V|$ reprezintă numărul nodurilor, iar $|E|$ numărul muchiilor grafului. Altă modalitate de a consemna acest lucru: *complexitate* $O(B^M)$, unde B reprezintă cea mai lunga ramură, iar M lungimea maximă a drumului arborelui. Această cerere imensă de spațiu este motivul pentru care breadth-first search nu este practică în cazul problemelor mai ample.

Complexitatea în timp. Odată ce, în cel mai rău caz breadth-first search trebuie să ia în considerare toate drumurile către toate nodurile, complexitatea în timp a acestui tip de căutare este de $O(|V|+|E|)$. Cel mai bun caz în această căutare este conferit de complexitatea $O(1)$. Are loc atunci când nodul este găsit la prima parcurgere.

Completitudine. Metoda breadth-first search este completă. Această înseamnă că dacă există o soluție , metoda breadth-first search o va găsi, indiferent de tipul grafului. Cu toate acestea, dacă graful este infinit şi nu există nici o soluție, breadth-first search va "eşua".

Optimalitate. Pentru costul unitar pe muchii, bread-first search este o metodă optimă. În general, breadth-first search nu este o metodă optimă, şi aceasta deoarece returnează întotdeauna rezultatul cu cele mai puține muchii între nodul de start şi nodul vizat. Dacă graful este un graf ponderat, şi drept urmare are costuri asociate fiecărei etape, această problemă se rezolvă îmbunătățind metoda breadth-first search astfel încât să se uniformizeze costurile de căutare, identificate cu: costurile drumului. Totuşi, dacă graful nu este ponderat, şi prin urmare toate costurile etapelor sunt egale, breadth-first search va identifica cea mai apropiată şi optimă soluție.

2.1.4 Aplicații ale BFS

Breadth-first search poate fi folosită pentru rezolvarea unei game variate de probleme de teoria grafurilor, printre care:

- Găsirea tuturor componentelor conexe dintr-un graf.
- Identificarea tuturor nodurilor într-o componentă conexă.
- Găsirea celui mai scurt drum între nodurile *u* şi *v* (într-un graf neponderat).
- Testarea bipartiției unui graf.

Găsirea Componentelor Conexe

Mulțimea vârfurilor accesate prin metode BFS reprezintă cea mai mare componentă conexă care conține vârful de start.

Testarea bipartiției

BFS poate fi folosită pentru testarea bipartiției, începând căutarea cu orice vârf şi atribuind etichete alternative vârfurilor vizitate în timpul căutării. Astfel, se atribuie eticheta 0 vârfului de start, 1 tuturor vecinilor săi, 0 vecinilor acelor vecini, şi aşa mai departe. Dacă într-un anumit moment al procesului un vârf are vecini vizitați cu aceeaşi etichetă, atunci graful nu este bipartit. Dacă parcurgerea se sfârşeşte fără a se produce o astfel de situație, atunci graful este bipartit.

2.2. Căutarea în adâncime

Depth-first search (DFS) este un algoritm căutare a arborelui, structurii arborelui, sau a grafului. Formal, DFS reprezintă o căutare care evoluează prin expansiunea la primul vârf „fiu" a arborelui ce ia naştere pe măsură ce se coboară în adâncime, până în momentul în care vârful „țintă" este descoperit sau până când se întâlneşte un vârf care nu are „fii". La pasul următor, căutarea se reia (backtracking), revenind la nodul cel mai recent vizitat, însă pentru care explorarea nu este încheiată. Într-o implementare ne-recursivă, toate vârfurile recent vizitate sunt adăugate într-o stivă de tipul LIFO (Last In First Out), în scopul explorării acestora. Complexitatea în spațiu a DFS este cu mult mai mică decât cea a BFS (Breadth-First Search). De asemenea se pretează mult mai bine metodelor euristice de alegere a

ramurilor asemănătoare. Complexitatea în timp a ambilor algoritmi este proporţională cu numărul vârfurilor plus numărul muchiilor grafului corespunzător $\left(O\left(|V|+|E|\right)\right)$.

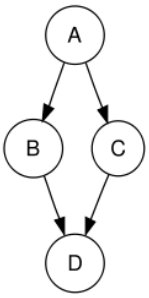
Căutarea în adâncime se poate folosi şi la ordonarea liniară a vârfurilor grafului (sau arborelui). Există trei astfel de posibilităţi:

■ O **preordine** reprezintă o listare a vârfurilor în ordinea în care au fost vizitaţi prin intermediul algoritmului căutării în adâncime. Aceasta este o modalitate naturală şi compactă de descriere a progresului căutării. O preordine a unei expresii arbore este ceea ce numim expresie în notaţia Poloneză.

■ O **postordine** reprezintă o listare în care cel din urmă vârf vizitat este primul element al listei. O postordine a unui expresii arbore este de fapt expresia în oglindă a expresiei în notaţie Poloneză.

■ O **postordine inversată (în oglindă)** este, de fapt, reversul postordinii, i.e. o listare a vârfurilor în ordinea inversă a celei mai recente vizite a vârfurilor in cauză. În căutarea unui arbore, postordinea inversată coincide cu preordinea, însă, în general, diferă atunci când se caută un graf.

Spre exemplu când se caută graful:



începând cu vârful A, preordinile posibile sunt A B D C, respectiv A C D B (în funcţie de alegerea algoritmului de a vizita mai întâi vârful B sau vârful C), în timp ce postordinile inversate (în oglindă) sunt: A B C D şi A C B D. Postordinea inversată produce o sortare topologică a oricărui graf orientat aciclic. Această ordonare este folositore şi în analiza fluxului de control, reprezentând adesea o liniarizare naturală a fluxului de control. Graful mai sus amintit poate reprezenta fluxul de control într-un fragment de cod ca cel de mai jos:

```
if (A) then {
       B
     } else {
       C
     }
     D
```

şi este natural să considerăm că acest cod urmează ordinea A B C D sau A C B D, însă nu este normal să urmeze ordinea A B D C sau A C D B.

```
PSEUDOCOD (recursiv)
dfs(v)
    process(v)
    mark v as visited
    for all vertices i adjacent to v not visited
        dfs(i)
```

O altă variantă

```
dfs(graph G)
{
  list L = empty
  tree T = empty
  choose a starting vertex x
  search(x)
  while(L is not empty)
  {
    remove edge (v, w) from beginning of L
    if w not yet visited
    {
      add (v, w) to T
      search(w)
    }
  }
}
search(vertex v)
{
  visit v
  for each edge (v, w)
    add edge (v, w) to the beginning of L
}
```

Aplicații

Iată câțiva algoritmi în care se foloseşte DFS:

- Găsirea componentelor conexe.
- Sortarea topologică.
- Găsirea componentelor tare conexe.

2.3. Metoda Greedy

Descrierea metodei Greedy

Metoda Greedy (*greedy = lacom*) este aplicabilă problemelor de optim.

Considerăm mulțimea finită

$$A = \{a_1, ..., a_n\}$$

şi o proprietate p definită pe mulțimea submulțimilor lui A:

$$p : P(A) \to \{0,1\} \text{ cu } \begin{cases} p(\emptyset) = 1 \\ p(X) = 1 \Rightarrow p(Y) = 1, \forall Y \subset X \end{cases}$$

O submulțime $S \subset A$ se numeşte *soluție* dacă p(S) = 1.

Dintre soluții va fi aleasă una care optimizează o funcție de cost

$$p : P(A) \rightarrow \mathbf{R}$$

dată.

Metoda urmăreşte evitarea căutării tuturor submulțimilor (ceea ce ar necesita un timp de calcul exponențial), mergându-se "direct" spre soluția optimă. Nu este însă garantată obținerea unei soluții optime; de aceea aplicarea metodei Greedy trebuie însoțită neapărat de o demonstrație.

Distingem doua variante generale de aplicare a metodei Greedy:

Prima variantă alege în mod repetat câte un element oarecare al mulțimii A şi îl adaugă soluției curente S numai dacă în acest mod se obține tot o soluție. În a doua variantă procedura `prel` realizează o permutare a elementelor lui A, după care elementele lui A sunt analizate în ordine şi adăugate soluției curente S numai dacă în acest mod se obține tot o soluție.

Exemplu. Se consideră mulțimea de valori reale $A = \{a_1, ..., a_n\}$. Se caută submulțimea a cărei sumă a elementelor este maximă.

Vom parcurge mulțimea şi vom selecta numai elementele pozitive, care vor fi plasate în vectorul soluție s.

```
k ← 0
for   i = 1,n
      if a_i > 0
      then k ← k + 1; s_k ← a_i
write(s)
```

2.4. Metoda backtracking

Un algoritm este considerat "acceptabil" numai dacă timpul său de executare este polinomial, adică de ordinul $O(n^k)$ pentru un anumit k; n reprezintă numărul datelor de intrare.

Pentru a ne convinge de acest lucru, vom considera un calculator capabil să efectueze un milion de operații pe secundă. în tabelul următor apar timpii necesari pentru a efectua n^3, 2^n şi 3^n operații, pentru diferite valori mici ale lui n:

	n = 20	n = 40	n = 60
n^3	-	-	0,2 sec
2^n	1 sec	12,7 zile	366 secole
3^n	58 min	3855 secole	10^{13} secole

Tabelul de mai sus arată că algoritmii exponențiali nu sunt acceptabili.

Descrierea metodei Backtracking

Fie produsul cartezian $X = X_1 \times ... \times X_n$. Căutam $x \in X$ cu $\varphi(x) = 1$, unde $\varphi : X \to \{0,1\}$ este o proprietate definită pe X.

Din cele de mai sus rezultă că generarea tuturor elementelor produsului cartezian X nu este acceptabilă.

Metoda backtracking încearcă micşorarea timpului de calcul. X este numit spaţiul soluţiilor posibile, iar φ sintetizează condiţiile interne.

Vectorul X este construit progresiv, începând cu prima componentă. Nu se trece la atribuirea unei valori lui x, decât dacă am stabilit valori pentru $x_1,...,x_{k-1}$ şi $\varphi_{k-1}(x_1,...,x_{k-1}) = 1$. Funcţiile $\varphi_k : X_1 \times ... \times X_n \to \{0,1\}$ se numesc condiţii de continuare şi sunt de obicei restricţiile lui φ la primele k variabile. Condiţiile de continuare sunt strict necesare, ideal fiind să fie şi suficiente.

Distingem următoarele cazuri posibile la alegerea lui x_k:

1) "*Atribuie şi avansează*": mai sunt valori neanalizate din X_k şi valoarea x_k aleasă satisface φ_k=> se măreşte k.
2) "*Încercare eşuată*": mai sunt valori neconsumate din X_k şi valoarea x_k aleasă dintre acestea nu satisface φ_k=> se va relua, încercându-se alegerea unei noi valori pentru x_k.
3) "*Revenire*": nu mai există valori neconsumate din X_k (X_k epuizată) $\Rightarrow$ întreaga X_k devine disponibilă şi k <- k-1.
4) "*Revenire după determinarea unei soluţii*": este reţinută soluţia.

Reţinerea unei soluţii constă în apelarea unei proceduri `retsol` care prelucrează soluţia şi fie opreşte procesul (dacă se doreşte o singură soluţie), fie prevede $k \leftarrow$ k-1 (dacă dorim să determinăm toate soluţiile).

Notăm prin $C_k \subset X_k$ mulţimea valorilor consumate din X_k. Algoritmul este următorul:

```
Ci<- 0, ∀ i;
k<-1;
while k > 0
       if k = n+1
              then retsol (x); k<- k-1;
else if Ck ⊂ Xk
       then alege v ∈ Xk \ Ck ;   Ck ← Ck ∪ {v};
              if φk(x1,...,xk-1, v) = 1
              then xk ← v ; k<- k+1;
              else
       else Ck<- ∅ ; k<- k-1;
```

Pentru cazul particular $X_1 = ... = X_n = \{1,...,s\}$, algoritmul se simplifică

```
k<-1; x_i<-0, ∀i = 1, …, n;
while k > 0
       if k = n+1
              then retsol (x); k<-k-1;
              else if x_k < s
                     then x_k ← x_k + 1;
                            if φ_k(x_1,...,x_k) = 1
                            then k<- k+1;
                            else
              else x_k<-0; k<-k-1;
```

2.5. Metoda divide et impera

Metoda Divide et Impera ("desparte şi stăpâneşte") consta în împărțirea repetată a unei probleme de dimensiuni mari în mai multe subprobleme de acelaşi tip, urmată de rezolvarea acestora şi combinarea rezultatelor obținute pentru a determina rezultatul corespunzător problemei inițiale. Pentru fiecare subproblemă procedăm în acelaşi mod, cu excepția cazului în care dimensiunea ei este suficient de mică pentru a fi rezolvată direct. Este evident caracterul recursiv al acestei metode.

Schema generală

Descriem schema generală pentru cazul în care aplicăm metoda pentru o prelucrare oarecare asupra elementelor unui vector. Funcția `DivImp`, care întoarce rezultatul prelucrării asupra unei subsecvențe $a_p,...,a_u$, va fi apelată prin `DivImp (1,n)`.

```
function DivImp(p,u)
       if u-p < ε
       then r <- Prel (p, u)
       else   m <- Interm (p,u);
              r1 <- DivImp (p,m);
              r2 <- DivImp (m+1,u);
              r <- Combin (r1,r2)
       return r
end;
```

unde:

- funcția `Interm` întoarce un indice în intervalul p..u; de obicei $m = \left\lfloor \frac{p+u}{2} \right\rfloor$;
- funcția `Prel` întoarce rezultatul subsecvenței p .. u, dacă aceasta este suficient de mică;
- funcția `Combin` întoarce rezultatul asamblării rezultatelor parțiale r1 şi r2.

2.6. Metoda Branch and Bound

Prezentare generală

Metoda *Branch and Bound* se aplică problemelor care pot fi reprezentate pe un arbore: se începe prin a lua una dintre mai multe decizii posibile, după care suntem puşi în situaţia de a alege din nou dintre mai multe decizii; vom alege una dintre ele etc. Vârfurile arborelui corespund stărilor posibile în dezvoltarea soluţiei.

Deosebim două tipuri de probleme:

1) Se caută un anumit vârf, numit *vârf rezultat,* care nu are descendenţi.
2) Există mai multe vârfuri finale, care reprezintă soluţii posibile, dintre care căutăm de exemplu pe cel care *minimizează* o anumită funcţie.

Deşi metoda este aplicabilă pe arbori. Există multe deosebiri, dintre care menţionăm:

- ordinea de parcurgere a arborelui;
- modul în care sunt eliminaţi subarborii care nu pot conduce la o soluţie;
- faptul ca arborele poate fi infinit (prin natura sa sau prin faptul că mai multe vârfuri pot corespunde la o aceeaşi stare).

În general arborele de stări este construit dinamic.

Este folosită o listă L de *vârfuri active*, adică de stări care sunt susceptibile de a fi dezvoltate pentru a ajunge la soluţie / soluţii. Iniţial, lista L conţine rădăcina arborelui, care este *vârful curent.* La fiecare pas, din *L* alegem un vârf (care nu este neapărat un fiu al vârfului curent!), care devine noul vârf curent.

Când un vârf activ devine vârf curent, sunt generaţi toţi fiii săi, care devin vârfuri active (sunt incluşi în L). Apoi din nou este selectat un vârf curent.

Legat de modul prin care alegem un vârf activ drept vârf curent, deci implicit legat de modul de parcurgere a arborelui, facem următoarele remarci:

- căutarea în adâncime nu este adecvată, deoarece pe de o parte arborele poate fi infinit, iar pe de altă parte soluţia căutată poate fi de exemplu un fiu al rădăcinii diferit de primul fiu şi căutarea în adâncime ar fi ineficientă: se parcurg inutil stări, în loc de a avansa direct spre soluţie;
- căutarea pe lăţime conduce totdeauna la soluţie (dacă aceasta există), dar poate fi ineficientă dacă vârfurile au mulţi fii.

Metoda Branch and Bound încearcă un "compromis" între cele două căutări menţionate mai sus, ataşând vârfurilor active cate un *cost* pozitiv, ce intenţionează sa fie o măsură a gradului de "apropiere" a vârfului de o soluţie. Alegerea acestui cost este decisivă pentru a obţine un timp de executare cât mai bun şi depinde de problema concretă, dar şi de abilitatea

programatorului.

Costul unui vârf va fi totdeauna mai mic decât cel al descendenţilor (fiilor) săi.

De fiecare dată drept vârf curent este ales cel de cost minim (cel considerat ca fiind cel mai "aproape" de soluţie).

Din analiza teoretică a problemei deducem o valoare lim care este o aproximaţie prin adaos a minimului căutat: atunci când costul unui vârf depaseste lim, vârful curent este ignorat: nu este luat în considerare şi deci este eliminat întregul subarbore pentru care este rădăcină. Dacă nu cunoaştem o astfel de valoare lim, o iniţializăm cu $+\infty$.

Se poate defini o *funcţie de cost ideală,* pentru care c(x) este dat de:

- nivelul pe care se află vârful x dacă x este vârf rezultat;
- $+\infty$ dacă x este vârf final, diferit de vârf rezultat;
- min {c(y) | y fiu al lui x} dacă x nu este vârf final.

Această funcţie este ideală din două puncte de vedere:

- nu poate fi calculată dacă arborele este infinit; în plus, chiar dacă arborele este finit, el trebuie parcurs în întregime, ceea ce este exact ce dorim să evităm;
- dacă totuşi am cunoaşte această funcţie, soluţia poate fi determinată imediat: plecăm din rădăcină şi coborâm mereu spre un vârf cu acelaşi cost, până ajungem în vârful rezultat.

Neputând lucra cu funcţia ideală de mai sus, vom alege o aproximaţie d a lui c, care trebuie să satisfacă condiţiile:

1) în continuare, dacă y este fiu al lui x avem d(x) < d(y);
2) d(x) să poată fi calculată doar pe baza informaţilor din drumul de la rădăcină la x;
3) este indicat ca $d \leq c$ pentru a ne asigura că dacă d(x) > lim, atunci şi c(x) > lim, deci x nu va mai fi dezvoltat.

O primă modalitate de a asigura compromisul între căutările în adâncime şi pe lăţime este de a alege funcţia d astfel încât, pentru o valoare naturală k, să fie îndeplinită condiţia: pentru orice vârf x situat pe un nivel nx şi orice vârf situat pe un nivel ny $\geq$ nx + k, să avem d(x) > d(y), indiferent dacă y este sau nu descendent al lui x.

Condiţia de mai sus spune că niciodată nu poate deveni activ un vârf aflat pe un nivel ny $\geq$ nx + k dacă în L apare un vârf situat pe nivelul nx, adică nu putem merge "prea mult" în adâncime. Dacă aceasta condiţie este îndeplinită, este valabilă următoarea propoziţie:

Propoziţie.

În ipoteza că este îndeplinită condiţia de mai sus şi dacă există soluţie, ea va fi atinsă într-un timp finit, chiar dacă arborele este infinit.

Algoritmul Branch & Bound pentru probleme de optim

Să presupunem că dorim să determinăm vârful final de cost minim şi drumul de la rădăcină la el. Fie `lim` aproximarea prin adaos considerată mai sus.

Algoritmul este următorul (`rad` este rădăcina arborelui, iar i_{final} este vârful rezultat):

```
i<-rad; L<={i}; min<-lim;
calculăm d(rad); tata(i)<-0
while L≠ Ø
       i <= L {este scos vârful i cu d(i) minim din min-ansamblul L}
       for toţi j fii ai lui i
              calculăm d(j); calcule locale asupra lui j; tata(j)<-i
              if j este vârf final
              then if d(j)< min
                     then min<-d(j); i_final<-j
                         elimină din L vârfurile k cu d(k)≥min (*)
              else if d(j)<min
                     then j => L
if min =lim then write {'Nu există soluţie')
           else writeln (min); i<-i_final
              while i≠ 0
                     write{i); i<-tata{i)
```

La (*) am ţinut cont de faptul că dacă j este descendent al lui i, atunci $d(i) < d(j)$.