

Tehnici de programare

backtracking

Metoda **backtracking** (a revenirilor succesive) de rezolvare a problemelor este una dintre cele mai puternice și generale din informatică. Multe probleme complexe, care nu pot fi rezolvate prin alte metode, se pot totuși rezolva prin *backtracking*, uneori chiar într-un mod simplu.

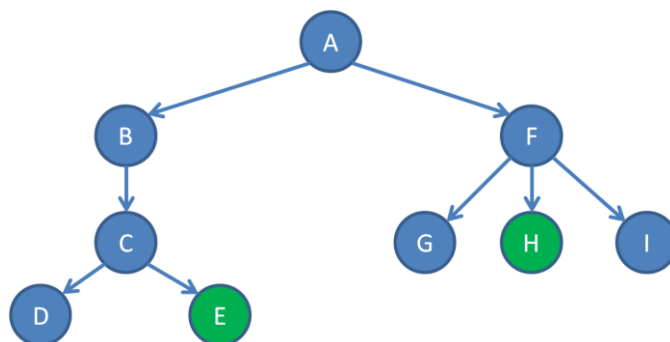
Algoritmii de tip backtracking funcționează în felul următor:

- soluția problemei se construiește succesiv, pas cu pas
- dacă la un pas există mai multe posibilități de continuare, se va încerca pe rând fiecare dintre ele
- dacă s-a ajuns într-un punct în care nu se mai poate continua, se revine la pasul anterior pentru a se încerca următoarea variantă posibilă
- după ce s-au epuizat toate posibilitățile de la pasul anterior, se revine cu încă un pas mai înainte și tot așa, în mod recursiv, până când au fost încercate toate posibilitățile din toți pașii

Prin această abordare se testează toate posibilitățile de a se ajunge la o soluție, cu alte cuvinte are loc o căutare completă, exhaustivă, în spațiul soluțiilor. Deoarece algoritmi backtracking testează toate variantele posibile, ei se pot modifica ușor pentru diverse cerințe:

- testarea dacă există o soluție
- găsirea unei soluții
- găsirea tuturor soluțiilor
- găsirea unei soluții optime

Pentru a ilustra modul de funcționare a algoritmilor backtracking, să considerăm că avem o problemă în care pașii posibili arată ca în figura următoare:



Se începe din A și se dorește să se ajungă în E sau H, care reprezintă soluții. De exemplu, A poate fi punctul de intrare într-un labirint, iar E și H ieșirile. Nodurile intermediare reprezintă intersecții în care se pot alege mai multe variante de drum. Pașii pe care îi parcurge un algoritm backtracking pentru rezolvarea acestei probleme sunt următorii:

- se începe din punctul inițial A și se testează pe rând fiecare variantă posibilă
- în A avem variantele {B,F}, deci le vom încerca pe rând pe fiecare:
 - în B avem doar varianta C:
 - în C avem variantele {D,E}:
 - în D nu mai avem nicio posibilitate de continuare, deci ne întoarcem un pas înapoi

- în E am ajuns la o soluție. Dacă ne interesează doar o soluție, atunci algoritmul se poate opri aici. Altfel, dacă dorim să aflăm toate soluțiile sau soluția optimă, continuăm testarea celorlalte posibilități, deci revenim din nou în C
 - nu mai avem alte variante pentru C, revenim în B
 - nu mai avem alte variante pentru B, revenim în A
- în A, testăm varianta F
 - în F avem variantele {G,H,I}
 - în G nu mai avem nicio posibilitate de continuare, deci ne întoarcem un pas înapoi
 - în H am găsit încă o soluție. Dacă ne interesează toate soluțiile, o vom utiliza direct, de exemplu o afișăm. Dacă ne interesează soluția optimă, o vom compara cu cea mai bună soluție deja găsită și o vom reține doar pe cea mai bună dintre ele. Revenim în F.
 - în I nu mai avem nicio posibilitate de continuare, deci ne întoarcem un pas înapoi
 - nu mai avem alte variante pentru F, revenim în A
- în A nu mai sunt alte variante, deci algoritmul se încheie

Se poate constata că, dacă este nevoie, se testează toate variantele posibile. De fiecare dată când nu se mai poate continua, se revine un pas înapoi și se încearcă variantele rămase. Pe urmă se revine încă un pas înapoi și se încearcă toate variantele rămase de acolo, etc. Deoarece este nevoie de aceste reveniri, algoritmul trebuie să țină minte pentru fiecare pas ce variante au fost deja testate, pentru ca atunci când se revine la acel pas, variantele deja testate să nu mai fie încercate si a doua oară.

Dezavantajul major al algoritmilor de tip backtracking este faptul că în general ei au o complexitate exponențială, $O(k^n)$, ceea ce îi face să necesite timpi foarte mari de execuție, chiar și pentru date de intrare de dimensiuni reduse. Pentru a explica acest comportament, să presupunem că la fiecare pas avem în medie $k=2$ variante posibile. În acest caz, la primul pas vom avea un total de 2 variante, la al doilea va fi un total de 4 variante (2^2), la al treilea totalul este de 8 variante (2^3), etc. În această situație, pentru date de intrare din 32 de componente, numărul situațiilor posibile depășește 4 miliarde.

Algoritmii backtracking se pot implementa atât în variantă recursivă cât și nerecursivă. În acest laborator vom discuta varianta de implementare recursivă, deoarece ea ne scutește de implementarea unor structuri de date în care să memorăm progresul din pașii intermediari. În varianta recursivă, algoritmul backtracking se implementează printr-o funcție, conform următorului pseudocod:

```
funcție caută(poziție_curentă)
    dacă poziție_curentă nu este validă sau face deja parte din calea curentă, revenire
    adaugă poziție_curentă la calea curentă
    dacă poziție_curentă determină o soluție
        atunci folosește soluția găsită
    altfel
        pentru fiecare poziție_următoare la care se poate ajunge din poziție_curentă
            caută(poziție_următoare)
        sterge poziție_curentă din calea curentă
```

O implementare recursivă a algoritmului backtracking în limbajul de programare C:

```
void back(int k)
{
    for(int i=1;i<=n;i++)
    {
        v[k]=i;
        if (valid(k))
            if(solutie(k))
                afisare();
            else
                back(k+1);
    }
}
```

```
}
```

Pentru exemplificare, vom considera problema ieșirii dintr-un labirint, pe care o vom rezolva pentru cerințe diferite: căutarea tuturor soluțiilor posibile, căutarea soluției optime și căutarea unei singure soluții.

Exemplul 1: Un labirint este reprezentat printr-o matrice pătratică cu elemente de tip *char*, care pentru pereți au valoarea '#' iar pentru străzi valoarea '.'. Labirintul este înconjurat de un zid în care poate exista una sau mai multe ieșiri. Se începe dintr-o poziție dată din interiorul labirintului și se cere să se ajungă la o ieșire, mergând doar pe orizontală sau pe verticală. Se citește dintr-un fișier dimensiunea labirintului, poziția inițială și apoi labirintul propriu-zis, dat sub forma unor linii de text, câte o linie pentru fiecare linie din matrice. Se cere să se afișeze toate soluțiile posibile de a ieși din labirint, atât sub forma unei succesiuni de coordonate care constituie acel traseu, cât și sub formă matriceală:

```
// cautarea tuturor solutiilor posibile
#include <stdio.h>
#include <stdlib.h>

// definirea labirintului
#define LMAX      100           // dimensiunea maxima pe o coordonata
// o celula are urmatoarele semnificatii:
// '.' - locatie libera
// '#' - zid
// alte caractere - locatii de pe traseu
char lab[LMAX][LMAX];          // labirintul propriu-zis; matrice patratica
int nLab;                      // latimea si lungimea labirintului
int startI,startJ;             // punctul de pornire

// citeste un labirint
void citire(const char *numeFisier)
{
    FILE *fis=fopen(numeFisier,"rt");
    if(!fis){
        printf("fisierul %s nu poate fi deschis\n",numeFisier);
        exit(EXIT_FAILURE);
    }
    fscanf(fis,"%d",&nLab);
    fscanf(fis,"%d %d",&startI,&startJ);
    int i,j;
    char linie[LMAX+1];
    fgets(linie,LMAX,fis);      // consuma \n de dupa startY
    for(i=0;i<nLab;i++){
        fgets(linie,LMAX,fis);
        for(j=0;j<nLab;j++)lab[i][j]=linie[j];
    }
    fclose(fis);
}

// afiseaza un labirint, inclusiv cu un posibil traseu
void afisare()
{
    int i,j;
    for(i=0;i<nLab;i++){
        for(j=0;j<nLab;j++){
            putchar(lab[i][j]);
        }
    }
}
```

```

    }
    putchar('\n');
}

#define TMAX 1000 // lungimea maxima a unui traseu

int traseul[TMAX],traseuJ[TMAX]; // coordonatele (i,j) ale punctelor de pe traseu
int nTraseu; // lungimea traseului
int nSolutii; // numarul de solutii
// delta i si j - cantitatea de adaugat la pozitia curenta (i,j) pentru a se obtine noua pozitie
int di[4]={1,-1,0,0};
int dj[4]={0,0,1,-1};

void solutie()
{
    int k;
    nSolutii++;
    printf("Solutia %d, lungime %d:",nSolutii,nTraseu);
    for(k=0;k<nTraseu;k++)printf(" (%d,%d)",traseul[k],traseuJ[k]);
    putchar('\n');
    afisare();
    putchar('\n');
}

// cauta toate solutiile
void cauta(int i,int j) // (i,j) - pozitia curenta
{
    // revenire daca pozitia curenta nu este valida:
    if(i<0||i>=nLab||j<0||j>=nLab)return; // daca este in afara labirintului
    if(lab[i][j]!='.')return; // sau daca nu este libera

    // adauga pozitia curenta la traseu
    traseul[nTraseu]=i;
    traseuJ[nTraseu]=j;
    lab[i][j]='O'+nTraseu%10; // setare pozitie ocupata cu sugerarea vizuala a traseului
    nTraseu++;

    if(i==0||i==nLab-1||j==0||j==nLab-1){ // testare daca este solutie
        solutie();
    }else{ // daca nu este solutie, incearca toate variantele posibile
        int k;
        for(k=0;k<4;k++){
            cauta(i+di[k],j+dj[k]);
        }
    }

    // sterge pozitia curenta din traseu
    nTraseu--;
    lab[i][j]='.';
}

int main()
{
    citire("lab.txt");

```

```

cauta(startI,startJ);
printf("numar solutii: %d\n",nSolutii);
return 0;
}

```

Pentru fișierul de intrare de mai jos vor fi afișate 52 de soluții, dintre care cele mai scurte au lungimea 14:

```

10
2 3
#####
#.#.#.#...#
#.....#.#
#.#.#.#.#.#
#...#.#.###
#.....#
#...####.#
###...#..#
#...#.....#
#.#.#####

```

Labirintul este conținut în matricea *lab* și are dimensiunea *nLab*. Punctul de start este dat de *(startI,startJ)*. După ce s-a citit de pe disc fișierul, se apelează funcția *cauta*, care implementează o căutare de tip backtracking. Funcția *cauta* primește ca parametri poziția curentă *(i,j)*, care este inițial *(startI,startJ)*. La începutul funcției se testează dacă poziția curentă este validă, în acest caz dacă este în interiorul labirintului. Ulterior se testează dacă poziția curentă este liberă, adică nu este un zid sau nu face parte din traseul de până la acest punct.

Testarea dacă poziția curentă nu a fost deja folosită este foarte importantă și ea trebuie făcută în toate situațiile în care există posibilitatea ca într-un pas viitor să se ajungă din nou la configurația dintr-un pas anterior. În acest caz, dacă s-a ajuns din nou într-o celulă care deja a fost parcursă, înseamnă că ne-am învârtit în cerc, astfel că vom interzice asemenea deplasări. La modul general, trebuie interzise deplasările care ne duc într-o configurație identică cu una anterioară.

După ce ne-am asigurat că poziția curentă este validă, o introducem în vectorii *traseul* și *trauseuJ*, care memorează succesiunea pașilor, iar apoi marcăm în matrice faptul că am parcurs deja acea celulă. Marcarea am făcut-o pentru fiecare pas cu un caracter din secvența '0' la '9', repetată în funcție de lungimea drumului, astfel încât să fie mai ușor de urmărit vizual deplasarea. Această marcă se poate face și simplu, folosind un caracter oarecare, gen '*'.

În continuare testăm dacă poziția curentă determină o soluție. În cazul nostru, o soluție este atunci când am ajuns la o deschizătură în zidul exterior. În acest caz, se apelează funcția *solutie*, care afișează traseul curent și totodată labirintul. În labirint deja este marcat drumul pe care l-am parcurs.

Dacă nu am ajuns la o soluție, căutăm în continuare. Pentru aceasta, încercăm toate variantele posibile în care ne putem deplasa din poziția curentă. Sunt 4 variante, deoarece ne putem deplasa pe orizontală sau pe verticală. Cu fiecare dintre noile poziții posibile, apelăm recursiv funcția *cauta*. De remarcat faptul că aceste noi poziții sunt posibile din punctul de vedere al regulilor de a ajunge la o poziție viitoare, dar ele nu sunt neapărat și valide. Validitatea lor va fi testată la începutul noului apel al funcției *cauta*, așa cum s-a discutat mai sus.

În final, după ce am tratat soluția sau am epuizat toate variantele de a merge mai departe, vom șterge atât din traseu cât și din matrice pasul curent, pentru a reveni la starea anterioară pasului curent. Pe această revenire la pasul anterior se bazează și numele algoritmului, backtracking.

Exemplul 2: Să se modifice exemplul 1, astfel încât să se afișeze soluția de lungime minimă. În continuare este redat doar ceea ce este modificat față de exemplul anterior:

```

// cautarea solutiei de lungime minima
#include <limits.h>

```

```

...
int traseulMin[TMAX],traseuJMin[TMAX];           // coordonatele (i,j) ale punctelor de pe traseul minim
int nTraseuMin=INT_MAX;                          // lungimea traseului minim

void solutie()
{
    int k;
    for(k=0;k<nTraseu;k++){
        traseulMin[k]=traseul[k];
        traseuJMin[k]=traseuJ[k];
    }
    nTraseuMin=nTraseu;
}

// cauta toate solutiile
void cauta(int i,int j)                          // (i,j) - pozitia curenta
{
    if(nTraseu==nTraseuMin)return;               // daca nu mai sunt sanse pentru o solutie mai buna, revenire
    ...
}

int main()
{
    citire("lab.txt");
    cauta(startI,startJ);
    if(nTraseuMin==INT_MAX){
        printf("nu exista solutie");
    }else{
        int k;
        printf("Solutia minima are lungimea %d si este compusa din:",nTraseuMin);
        for(k=0;k<nTraseuMin;k++){
            printf(" (%d,%d)",traseulMin[k],traseuJMin[k]);
            lab[traseulMin[k]][traseuJMin[k]]='0'+k%10;
        }
        putchar('\n');
        afisare();
        putchar('\n');
    }
    return 0;
}

```

Pentru a memora soluția de lungime minimă, am mai introdus un set de variabile: *traseulMin*, *traseuJMin*, *nTraseuMin*. Variabila *nTraseuMin* a fost inițializată cu *INT_MAX*, o constantă definită în antetul *<limits.h>*. *INT_MAX* definește cea mai mare valoare posibilă pentru tipul *int*. Astfel, orice primă soluție care va fi găsită va avea o lungime mai mică decât *nTraseuMin* și va fi memorată. Singurul rol al funcției *solutie* este să copieze soluția curentă în acest set de variabile.

La începutul funcției *cauta* s-a introdus o condiție de revenire în caz că nu mai este posibilă o soluție mai bună decât cea găsită deja. În cazul nostru, dacă traseul curent are deja lungimea traseului minim, evident că el nu va mai putea constitui o soluție mai bună decât cea găsită până atunci. Această condiție introduce totodată o optimizare foarte importantă: pe orice traseu ne-am afla, în momentul în care el nu mai poate constitui prefixul unei soluții valide (mai bună decât minimul deja găsit), se renunță la a se mai testa pașii următori de pe acel traseu. În cazul problemelor complexe, cu milioane, miliarde sau chiar mai multe posibilități, această optimizare poate deveni fundamentală. Din cauza acestei condiții, în funcția *solutie* nu mai este necesar să testăm dacă soluția curentă este mai bună decât cea minimă, fiindcă dacă nu ar fi fost, nu s-ar fi ajuns la apelul lui *solutie*.

În funcția *main*, după ce s-a apelat *cauta*, se testează prima oară dacă s-a găsit o soluție. Dacă s-a găsit, atunci se afișează lungimea, traseul și matricea corespunzătoare ei. Înainte de a se putea afișa matricea, a trebuit să se refacă în ea traseul, deoarece la revenirea din apelul *cauta(startI, startJ)* în matrice nu a mai rămas marcat niciun traseu.

Exemplul 3: Să se modifice exemplul 1, astfel încât să se găsească doar o soluție. În continuare este redat doar ceea ce este modificat față de exemplul 1:

```
// cautarea primei solutii

void solutie()                // nu mai afiseaza numarul solutiei, fiindca acum va fi gasita doar prima solutie
{
    int k;
    printf("Lungime %d:", nTraseu);
    for(k=0; k<nTraseu; k++) printf(" (%d,%d)", traseul[k], traseuJ[k]);
    putchar('\n');
    afisare();
    putchar('\n');
}

// daca s-a gasit o solutie returneaza 1, altfel 0
int cauta(int i, int j)
{
    // in caz ca nu se trece de testele de validitate, inseamna ca inca nu s-a gasit o solutie => return 0
    if(i<0 || i>=nLab || j<0 || j>=nLab) return 0;
    if(lab[i][j]!='.') return 0;

    traseul[nTraseu]=i;
    traseuJ[nTraseu]=j;
    lab[i][j]='0'+nTraseu%10;
    nTraseu++;

    if(i==0 || i==nLab-1 || j==0 || j==nLab-1){                // testare daca este solutie
        solutie();
        return 1;                // s-a gasit o solutie => return 1
    }else{
        int k;
        for(k=0; k<4; k++){
            // daca pentru oricare dintre variante s-a gasit o solutie => return 1
            if(cauta(i+di[k], j+dj[k])) return 1;
        }
    }

    nTraseu--;
    lab[i][j]='.';
    // inca nu s-a gasit nicio solutie, fiindca altfel s-ar fi executat return 1
    // ori pe ramura if-ului de testare solutie, ori pe ramura else
    return 0;
}

int main()
{
    citire("lab.txt");
    if(!cauta(startI, startJ)){
        printf("nu s-a gasit nicio solutie\n");
    }
}
```

```
return 0;  
}
```

În această versiune, funcția *cauta* returnează 1 în caz că s-a găsit o soluție și 0 în caz contrar. Din momentul în care s-a găsit prima soluție, deci se returnează 1, se poate constata că acest 1 este imediat propagat mai departe, până când se iese din toate apelurile recursive ale funcției *cauta*, fără a se mai face revenirea la pașii anteriori. Astfel, atât traseul memorat cât și matricea rămân în starea în care erau când s-a descoperit prima soluție.

Optimizări ale algoritmilor de tip backtracking

Din cauză că algoritmi de tip backtracking au o complexitate exponențială $O(k^n)$, timpul computațional necesar poate depăși destul de repede posibilitățile existente. De exemplu, după unele estimări, numărul de mutări posibile într-un joc de șah este mai mare decât numărul de electroni din univers. Astfel, un algoritm de backtracking neoptimizat, care să joace șah, nu va putea testa decât foarte puține mutări în avans. Din acest motiv, sunt foarte importante modalitățile prin care putem reduce numărul de cazuri luat în considerare. Câteva optimizări sunt mai importante:

- limitarea la prima soluție găsită, chiar dacă ea nu este optimă
- renunțarea la a căuta mai departe pe o anumită cale, din momentul în care ea duce la un rezultat mai slab decât cel mai bun rezultat găsit anterior (a se vedea exemplul 2)
- selectarea variantelor posibile la fiecare pas într-o manieră *Greedy*, astfel încât primele variante încercate să fie cele care ne apropie cât mai mult de destinație
- dacă există mai multe variante posibile, testarea a doar câtorva dintre ele, cele mai promițătoare
- implementarea unui contor de timp sau de număr de pași, iar atunci când contorul depășește o limită maximă, returnarea celui mai bun rezultat găsit până la acel moment
- identificarea unor *puncte nodale* prin care trebuie să treacă toate soluțiile și apoi împărțirea căutării în două: prima căutare înainte de punctul nodal și a doua după punctul nodal. De exemplu, dacă între două orașe există un singur drum și vrem să ajungem dintr-un oraș în altul, se poate face o căutare de la adresa de pornire până la drumul de legătură și o altă căutare de la drum până la adresa de destinație. Astfel, complexitatea algoritmului va fi $O(k^m) + O(k^n)$, în loc de să fie $O(k^{m+n})$

Aplicații propuse

Aplicația 12.1: Se citesc două numere naturale n și k . Generați și afișați toate numerele naturale formate din n cifre care conțin exact k cifre de 1.

Aplicația 12.2: Se citește un număr natural n . Afișați permutările multimii $1, 2, 3, \dots, n$ în care până la jumătate elementele sunt în ordine descrescătoare, iar de la jumătate până la final în ordine crescătoare.

Exemplu:

$n=5$

permutările care respectă condiția sunt:

3 2 1 4 5

4 2 1 3 5

4 3 1 2 5

5 2 1 3 4

5 3 1 2 4

5 4 1 2 3

Aplicația 12.3: Se citesc două numere naturale n și k , k fiind mai mic decât numărul de cifre ale numărului n . Afișați cel mai mare număr care se poate forma eliminând k cifre din numărul n .

Exemplu:

$n=3452234$

$k=4$

numărul căutat este 534

Aplicația 12.4: Se dau următoarele 6 culori: alb, galben, roșu, verde, albastru și negru.

Costruiți toate steagurile formate din 3 culori care îndeplinesc următoarele condiții:

- orice steag trebuie să conțină culoarea verde sau culoarea galben la mijloc.
- culorile din fiecare steag trebuie să fie distincte.

Aplicația 12.5: Se cere un $0 < n \leq 100$ și apoi n valori reale, fiecare reprezentând volumul unui obiect. În final se cere v , volumul unei cutii. Se cere să se umple cutia cu unele dintre obiecte date, astfel încât volumul ei să fie utilizat în mod optim.

Aplicația 12.6: De-a lungul unei șosele trebuie amplasate una lângă alta următoarele entități: *case*, *blocuri*, *grădini*, *ateliere*. O entitate se poate învecina doar cu una de același tip (ex: *casă* cu *casă*) sau conform următoarelor reguli (orice relație o implică și pe cea inversă): un *bloc* poate avea ca vecini *case*; o *casă* sau un *atelier* poate avea ca vecini *grădini*. Se cer de la tastatură numerele c, b, g, a care reprezintă respectiv numărul de *case*, *blocuri*, *grădini* și *ateliere*. Să se determine toate variantele în care acestea pot fi aranjate.

Aplicația 12.7: Se cere $0 < n \leq 100$ și apoi n valori pozitive întregi, reprezentând laturile unor pătrate. În final se cere $0 < p \leq 100$, valoare întreagă, reprezentând latura unui pătrat. Se cere să se determine dacă există un aranjament al celor n pătrate în interiorul pătratului de latură p , astfel încât toate pătratele să fie conținute în acesta și să nu existe suprapuneri între pătrate.

Aplicația 12.8: Se cere d de tip întreg reprezentând lungimea unei drepte și $0 < n \leq 100$ un număr de segmente. În câte feluri se poate împărți dreapta dată în n segmente consecutive, având fiecare lungimi întregi pozitive, astfel încât fiecare segment să fie strict mai mare decât cel de dinaintea sa?