1 Metoda Backtracking

Metoda backtracking este o paradigmă de proiectare a algoritmilor care este folosită în general pentru problemele care nu pot fi rezolvate mai eficient, prin algoritmi ad-hoc sau de tip greedy, programare dinamică, etc.

În particular, metoda backtracking este folosită pentru rezolvarea problemelor NP-complete, probleme pentru care nu se cunosc algoritmi polinomiali.

În cele mai multe cazuri, un algoritm backtracking conduce la un timp exponențial de execuție.

Pentru a înțelege metoda backtracking, vom începe cu *metoda enumerării exhaustive* a spațiului de soluții al problemei.

1.1 Metoda căutării exhaustive

Pentru a exemplifica metoda, vom folosi problema SAT (problema satisfiabilității):

`SAT`

`Input`: o formula f din logica propozițională

`Output`: *da*, dacă formula f este satisfiabilă; *nu*, altfel

O formulă este satisfiabilă dacă există o atribuire σ de la mulțimea variabilelor propoziționale la mulțimea valori de adevăr astfel încât f să fie adevărată în atribuirea σ.

De exemplu, formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ este satisfiabilă, deoarece este adevărată în atribuirea $\sigma : \{x_1, x_2, x_3\} \to \{0, 1\}$, definită prin: $\sigma(x_1) = 1$, $\sigma(x_2) = 0$ și $\sigma(x_3) = 0$.

Putem verifica satisfiabilitatea unei formule prin enumerarea exhaustivă a tuturor celor 2^n atribuiri posibile, unde n este numărul de variabile propoziționale. Vom presupune, fără să pierdem din generalitate, că variabilele propoziționale din formulă sunt $x_1, x_2, \ldots, x_n$.

De exemplu, pentru $n = 3$, este suficient să verificăm dacă formula este adevarată în următoarele 8 atribuiri:

1. $\sigma(x_1) = 0, \sigma(x_2) = 0, \sigma(x_3) = 0$: f este falsă;
2. $\sigma(x_1) = 0, \sigma(x_2) = 0, \sigma(x_3) = 1$: f este falsă;
3. $\sigma(x_1) = 0, \sigma(x_2) = 1, \sigma(x_3) = 0$: f este falsă;
4. $\sigma(x_1) = 0, \sigma(x_2) = 1, \sigma(x_3) = 1$: f este adevărată;
5. $\sigma(x_1) = 1, \sigma(x_2) = 0, \sigma(x_3) = 0$: f este adevărată;
6. $\sigma(x_1) = 1, \sigma(x_2) = 0, \sigma(x_3) = 1$: f este adevărată;
7. $\sigma(x_1) = 1, \sigma(x_2) = 1, \sigma(x_3) = 0$: f este falsă;
8. $\sigma(x_1) = 1, \sigma(x_2) = 1, \sigma(x_3) = 1$: f este adevărată.

Putem enumera cele 2^n asignări posibile folosind următorul algoritm de căutare:

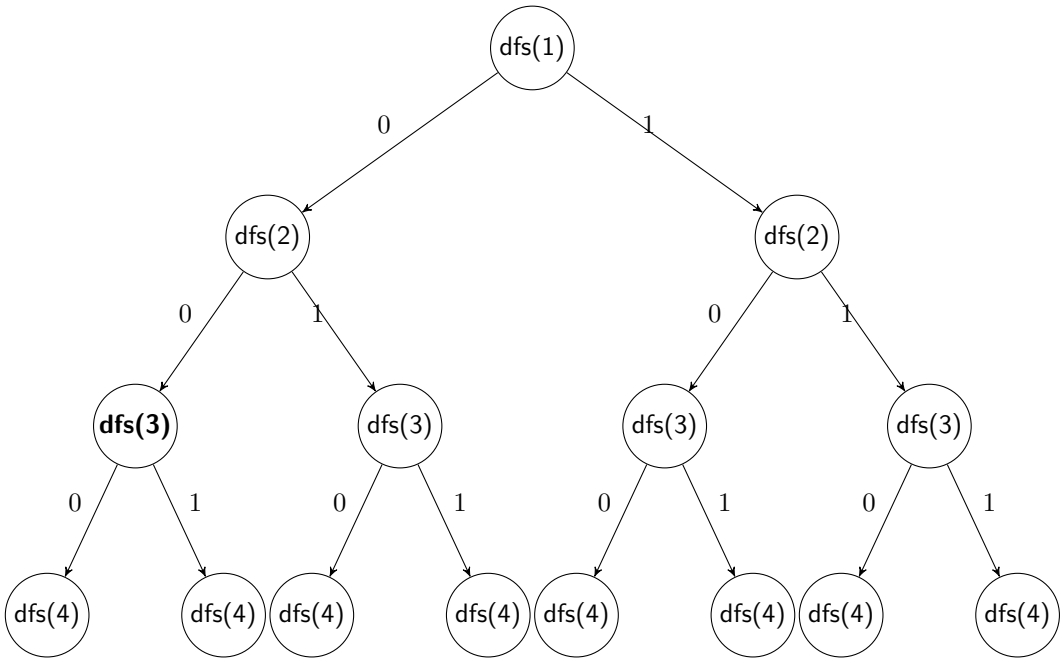
```
// sigma[1..n] este un tablou global reprezentând o atribuire.
// sigma[i] = 0, dacă x_i are valoarea fals
//          = 1, dacă x_i are valoarea adevarat
dfs(i)
{
  if (i == n + 1) {
    if (eval(f, sigma) == 1)
      return 1;
    else
      return 0;
  } else {
    for (v = 0; v <= 1; ++v) {
      sigma[i] = v;
      if (dfs(i + 1)) {
        return 1;
      }
    }
    return 0;
  }
}

return dfs(1);
```

Apelul `dfs(i)` setează

1. `sigma[i] = 0` și apelează `dfs(i + 1)` pentru completarea (recursivă a) `sigma[i + 1], sigma[i + 2], ...`
2. apoi `sigma[i] = 1` și apelează `dfs(i + 1)` pentru completarea (recursivă a) `sigma[i + 1], sigma[i + 2], ....`

În acest fel se ajunge la `i = n + 1` după ce toate valorile `sigma[1], ..., sigma[n]` au fost completate. Observăm că apelul `dfs(i)` returnează `1` ddacă se poate completa atribuirea parțială `sigma[1], ..., sigma[i - 1]` (cu anumite valori de adevăr pentru `sigma[i], ..., sigma[n]`) astfel încât formula să fie adevărată în atribuirea completă.

Apelurile recursive pot fi vizualizate mai ușor folosind următorul arbore binar, în care un nod reprezintă un apel recursiv al funcției `dfs`, pe muchii este trecut modul în care este schimbat vectorul `sigma`, iar fiii unui nod sunt apelurile recursive generate în respectivul nod:



Motivul pentru care funcția de căutare este numită `dfs` (de la "depth-first search", adică căutare în adâncime) este că arborele de mai sus este parcurs *depth-first*: întâi se parcurge complet prima ramură (se merge "în adâncime"), apoi se ia drumul spre următoarea frunză, ș.a.m.d.

În fiecare frunză, se verifică dacă formula dată la intrare este adevărată pentru asignarea `sigma` din nodul respectiv. Din păcate, deoarece există 2^n frunze, timpul de rulare este exponențial.

1.2 Backtracking ca optimizare a căutării exhaustive

Metoda backtracking îmbunătățește astfel de algoritmi de explorare exhaustivă prin *retezarea* unor ramuri ale arborelui (în engleză, procesul se numește *pruning*).

În nodul `dfs(3)` din arborele precedent care este îngroșat, știm deja că `sigma[1] = 0, sigma[2] = 0`, dar încă nu am stabilit dacă `sigma[3] = 0` sau `sigma[3] = 1`. O astfel de asignare, în care au fost stabilite doar valorile anumitor variabile propoziționale, se numește *asignare parțială*. Pentru asignarea parțială `sigma[1] = 0, sigma[2] = 0` din nodul îngroșat, indiferent cum o completăm (i.e., dacă alegem `sigma[3] = 0` sau `sigma[3] = 1`), observăm că formula $f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ este falsă. Din moment ce f este falsă indiferent cum extindem asignarea parțială, înseamnă că nu este nevoie să continuăm să explorăm această ramură a arborelui (nu avem șanse să ajungem pe această ramură la o soluție).

Algoritmul de mai sus se poate modifica foarte ușor pentru a realiza această optimizare:

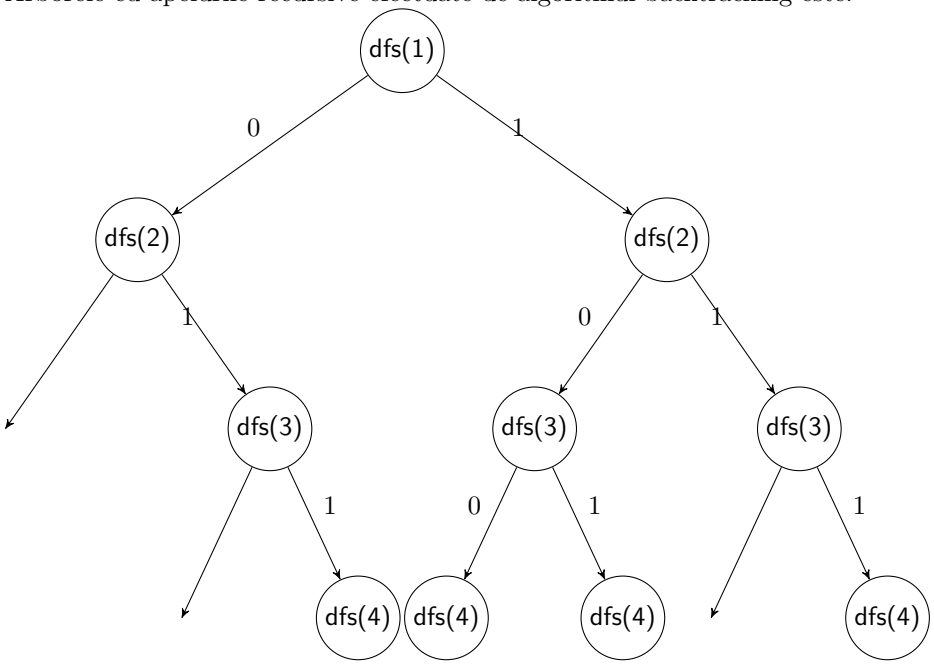
```
dfs(i)
{
  if (i == n + 1) {
```

```
    return 1;
  } else {
    for (v = 0; v <= 1; ++v) {
      sigma[i] = v;
      if (eval(f, sigma, i) != 0)
        if (dfs(i + 1) == 1)
          return 1;
    }
    return 0l
  }
}

return dfs(1);
```

Testul `eval(f, sigma, i) != 0` poate fi implement prin înlocuirea valorilor celor i variabile propoziționale deja fixate. De exemplu, dacă $sigma[0] = 0$ și $sigma[1] = 1$, atunci $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) = (0 \vee 0) \wedge (\neg 0 \vee x_3) = 0 \wedge ? = 0$ (deci f este in mod necesar falsă in acest caz).

În algoritmul de mai sus, apelul recursiv `dfs(i + 1)` este efectuat doar dacă există o șansă ca asignarea parțială `sigma` să conducă la un rezultat. Din acest motiv, verificarea că formula f este adevărată la intrarea în cazul `i == n + 1` este redundantă și am renunțat la ea.

Arborele cu apelurile recursive efectuate de algoritmul backtracking este:



Ramurile care indică spre un nod lipsă reprezintă ramurile *retezate/tunse* (engl. *pruned*), care în mod necesar conduc la o asignare parțială care nu poate fi extinsă astfel încât formula să fie adevărată.

Denumirea de backtracking provine din faptul că algoritmul încearcă întâi să aleagă `sigma[i] = 0`. Dacă alegerea nu conduce la o soluție, atunci algoritmul *revine cu un pas înapoi* (engl. *backtracks*) și încearcă apoi `sigma[i] = 1`.

Exercițiu: scrieți în Alk un algoritm backtracking pentru problema 3-CNF-SAT, așa cum a fost formalizată în Seminarul $4-5$ (problema 7).

Backtracking-ul este o formă de căutare exhaustivă, dar implicită, deoarece anumite ramuri ale arborelui sunt retezate; aceste ramuri nu sunt parcurse explicit și astfel economisim timp față de căutarea exhaustivă.

Exercițiu: câte noduri apar în arborele "pruned" pentru formula $f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_4$?

1.3 Studiu de caz - problema celor n regime

În continuare, vom prezenta prin intermediul unui studiu de caz ingredientele principale ale metodei backtracking.

`Input`: $n \in \mathbb{N}$ (reprezentând dimensiunea tablei)

`Output`:

- (x_i, y_i) $(1 \leq i \leq n)$ cu $x_i, y_i \in \{1, \ldots, n\}$ cu proprietatea că dacă așezăm n regine la coordonatele (x_i, y_i) $(1 \leq i \leq n)$ pe o tablă de șah, acestea nu se atacă;
- -1, dacă nu există o astfel de așezare.

Exercițiu: formalizați specificația output-ului ca o formula de ordinul I.

Pentru a aborda problema celor n regine, trebuie să ne definim spațiul soluțiilor. Modul în care alegem spațiul soluțiilor va dicta atât ușurința implementării, cât și gradul de eficiență a acesteia.

O posibilitate ar fi să reprezentăm o soluție ca o matrice de 0/1, fiecare poziție din matrice având valoarea 1 ddacă o regină se găsește la poziția respectivă. Deși este o abordare posibilă, această reprezentare nu conduce la cea mai elegantă implementare de tip backtracking.

Putem observa ușor că, într-o soluție, două regine nu se pot afla niciodată pe aceeași linie (deoarece s-ar ataca). Așadar, fiecare regină are propria linie. Presupunem fără a pierde din generalitate că regina i este poziționată pe linia i. Astfel, putem reprezenta o soluție la problema celor n regine este sub forma unui vector c cu proprietatea că $c[i]$ este coloana pe care așezăm regina i $(1 \leq i \leq n)$.

O soluție parțială este un vector $c[1..k]$ $(0 \leq k \leq n)$ conține numere între 1 și n, reprezentând coloanele pe care sunt poziționate primele k regine, iar valorile $c[k+1..n]$ nu sunt încă completate, reprezentând faptul că încă nu am ales o coloană pentru reginele $k+1, \ldots, n$.

Soluția parțială inițială este vectorul $c[1..0]$ (cu 0 elemente) (nu am ales coloana pentru nicio regină).

Dacă avem o soluție parțială $c[1], c[2], \ldots, c[k]$ (completată până la regina k inclusiv), atunci soluțiile parțiale care sunt succesori imediați sunt:

1. $c[1], c[2], \ldots, c[k], 1$ (pun regina $k+1$ pe coloana 1)
2. $c[1], c[2], \ldots, c[k], 2$ (pun regina $k+1$ pe coloana 2)
3. ...
4. $c[1], c[2], \ldots, c[k], n$ (pun regina $k+1$ pe coloana n)

O soluție parțială $c[1], \ldots, c[k]$ este viabilă dacă reginele $1, \ldots, k$ nu se atacă între ele două câte două. Dacă o soluție parțială nu este viabilă (reginele se atacă între ele), atunci aceasta nu se poate extinde astfel încât să obținem o soluție a problemei.

O soluție parțială $c[1], \ldots, c[k]$ este completă dacă $k = n$ (adică dacă am stabilit deja pozițiile tuturor reginelor).

Schema generală pentru algoritmul backtracking este următorul:

```
dfs(s)
{
  // s este o soluție parțială
  // dfs(s) întoarce DA ddacă soluția parțială poate fi extinsă până
  //        la o soluție completă
  if s este completă {
    return 1;
  } else {
    for s' in succesori(s) do {
      if viabila(s') {
        if dfs(s') {
          return 1;
        }
      }
    }
    return 0;
  }
}

dfs(empty)
```

Exercițiu: implementați în Alk un algoritm backtracking pentru problema celor n regine.

1.4 Concluzie

Metoda backtracking este folosită în general pentru probleme pentru care nu există algoritmi mai eficienți (e.g., folosind greedy, programare dinamică, sau alte tehnici).

De exemplu, dacă știm despre o problemă că este NP-completă (cum este problema SAT), atunci nu are sens să pierdem mult timp căutând un algoritm polinomial, deoarece este puțin probabil să existe un asemenea algoritm. Pentru astfel de probleme, un algoritm de tip backtracking poate fi mai eficient în practică, măcar pentru anumite instanțe, decât o căutare exhaustivă. În general, algoritmii bazați pe metoda backtracking rămân exponențiali în cazul cel mai nefavorabil.

Metoda backtracking se folosește în general pentru probleme de decizie. Dar poate fi ușor extinsă pentru a număra soluțiile (se înlocuiește `return 1` cu incrementarea unei variabile globale reprezentând numărul de soluții, inițializată cu 0, iar "early-exit"-ul din bucla `for` dispare). De asemenea, această adaptare permite și rezolvarea problemelor de optim, prin enumerarea tuturor soluțiilor posibile și păstrarea soluției de cost minim/câștig maxim. Totuși, vom vedea

în cursul următor că pentru probleme de optimizare este preferabilă metoda branch-and-bound, care este o generalizare a tehnicii backtracking pentru probleme de optim.

Remember. Pentru a rezolva o problemă computațională prin backtracking, este necesar să definim următoarele *ingrediente*:

1. o reprezentare a soluțiilor posibile (de obicei, un vector sau o matrice cu anumite elemente);
2. noțiunea de soluție parțială (de obicei un prefix al unei soluții);
3. noțiunea de soluție parțială viabilă (care, măcar aparent, are șanse să fie completată);
4. succesorii unei soluții parțiale (dacă soluția parțială nu are succesori, trebuie să fie o soluție completă);
5. soluția parțială inițială (de obicei, vectorul vid).

Având aceste ingrediente bine definite, este foarte simplu să scriem codul aferent algoritmului.

2 Exerciții pentru seminar

1. Câte noduri apar în arborele "pruned" pentru formula $f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_4$?
2. Scrieți în Alk un algoritm backtracking pentru problema 3-CNF-SAT, așa cum a fost formalizată în Seminarul $4 - 5$ (problema 7).
3. Implementați în Alk un algoritm backtracking pentru problema celor n regine.
4. Proiectați un algoritm de tip backtracking pentru problema "submulțime de sumă dată". Prin proiectarea se înțelege inclusiv specificarea următoarelor informații:
   (a) cum este reprezentată o soluție;
   (b) care sunt soluțiile parțiale;
   (c) care sunt succesorii direcți ai unei soluții parțiale;
   (d) când o soluție parțială este viabilă.
5. Proiectați un algoritm de tip backtracking pentru problema Sudoku.
6. Proiectați un algoritm de tip backtracking pentru problema Nonogram.
7. Proiectați un algoritm de tip backtracking pentru problema circuitului hamiltonian.