Outline

Contents

1 Backtracking

Metoda Backtracking

Metoda backtracking este o paradigmă de dezvoltare a algoritmilor care este utilă în cazul problemelor care nu pot fi rezolvate (mai eficient) prin algoritmi ad-hoc sau de tip greedy, programare dinamică, etc.

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 (explicite) a spațiului de soluții.

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 asignare σ astfel încât f să fie adevărată în asignarea σ. De exemplu, formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ este satisfiabilă, deoarece este adevărată în asignarea $\sigma : \{x_1, x_2, x_3\} \to \{0, 1\}$, 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 asignări posibile, unde n este numărul de variabile propoziționale.

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

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:

```
dfs(i)
{
  if (i == n + 1) {
    if "f este adevarata in sigma"
```

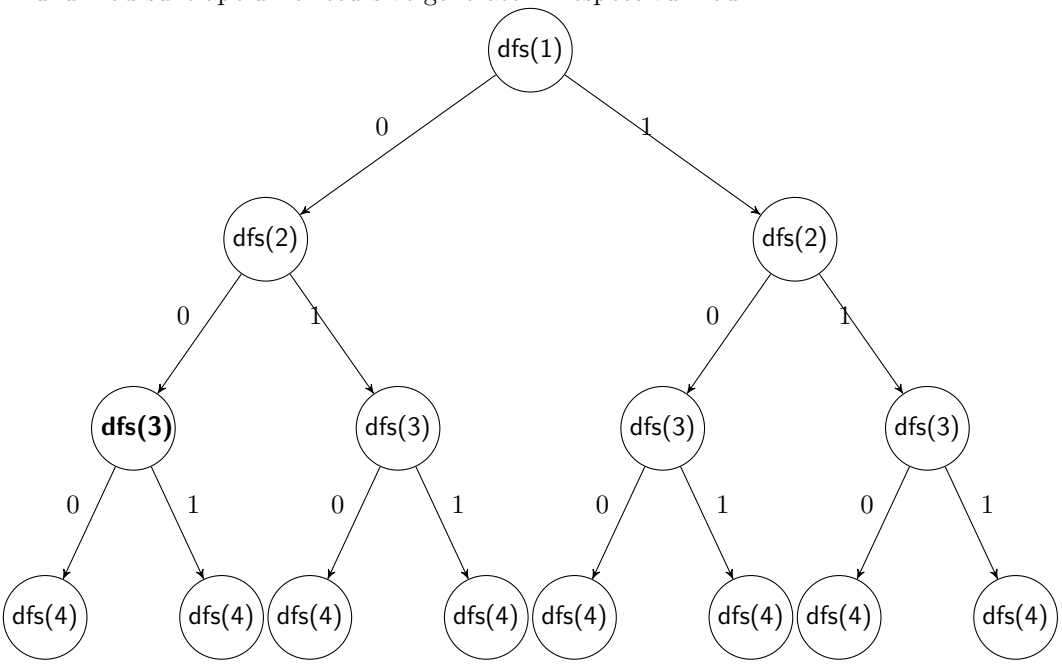
```
      return 1
  } else {
    for (v = 0; v <= 1; ++v) {
      sigma[i] = v;
      dfs(i + 1);
    }
  }
}

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. 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 fii unui nod sunt apelurile recursive generate în respectivul nod:



Motivul pentru care funcția de căutare este numită `dfs` (de la "depth-first search" 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.

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

Astfel, putem observa că pentru 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.

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

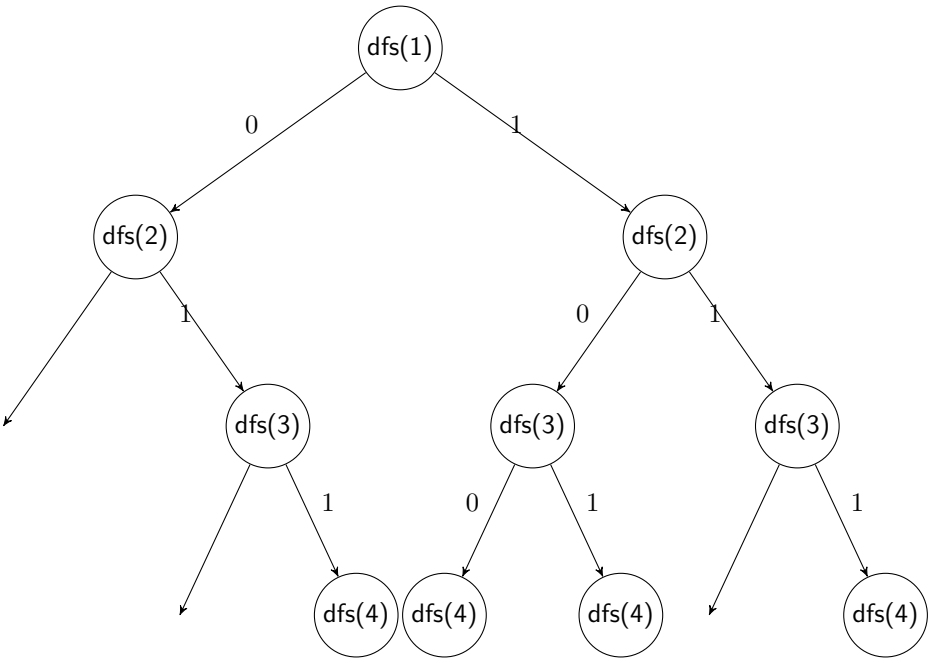
```
dfs(i)
{
  if (i == n + 1) {
    return 1
  } else {
    for (v = 0; v <= 1; ++v) {
      sigma[i] = v;
      if not "f este in mod necesar falsa in asignarea partiala sigma"
        dfs(i + 1);
    }
  }
}

return dfs(1)
```

Testul `"f este in mod necesar falsa in asignarea partiala sigma"` poate fi implement prin înlocuirea valorilor variabilelor propoziționale deja fixate. De exemplu, dacă $sigma[0] = 0$ și $sigma[1] = 1$, $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) = (0 \vee 0) \wedge (\neg 0 \vee x_3) = 0 \wedge ? = 0$ (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 dată la intrare în cazul `i == n + 1` este reduntantă 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 (problema 7).

Backtracking este tot o formă de căutare exhaustivă, dar implicită, deoarece anumite ramuri ale arborelui sunt retezate; aceste ramuri nu sunt parcurse explicit (ci doar implicit, deoarece ne asigurăm ca nu există soluții în subarborele aferent) și astfel economisim timp față de căutarea exhaustivă.

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

Problema celor n regime

`Input`: $n \in \mathbb{N}$ `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.

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.

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 $c[1..n]$, completat doar parțial: $c[1..j]$ conține numere între 1 și n, reprezentând coloane, iar $c[j+1..n]$ conține doar 0,

reprezentând faptul că încă nu am ales o coloană pentru reginele $j+1, \ldots, n$.

Soluția goală este vectorul $c[1..n]$ care conține doar 0 (nu am ales coloana pentru nicio regină).

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

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

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

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

Schema generală pentru algoritmul backtracking este următorul:

```
dfs(s)
{
  // s este o solutie partiala
  if s este o solutie completa
    return s; // (am gasit o solutie)
  else
    for s' in succesori(s) do
      if viabil(s')
        dfs(s')
}

dfs(empty)
```

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

Branch-and-bound

Tehnica branch-and-bound poate fi privită ca o extensie a metodei backtracking pentru probleme de optimizare.

Vom ilustra tehnica branch-and-bound pe problema discretă a rucsacului:

`Input`: $n, g[1..n], c[1..n], G$ `Output`: câștigul maxim care poate fi obținut în condițiile problemei (exercițiu: completați specificația output-ului)

Problema discretă a rucsacului poate fi abordată și prin metoda programării dinamice (obținând un algoritm pseudopolinomial), dar vom exemplifica cu ajutorul ei metoda branch-and-bound.

Pentru orice algoritmul branch-and-bound, este nevoie de o metodă prin care poate fi aproximat (extrem de eficient/rapid, chiar cu riscul unei aproximări proaste) câștigul/costul unei soluții optime. Dacă problema este de maximizare (cum este cazul problemei discrete a rucsacului – unde dorim să maximizăm câștigul), aproximarea trebuie să fie o supraaproximare pentru ca algoritmul branch-and-bound să fie corect.

Vom nota cu h o funcție care calculează eficient o supraaproximare $h(n, g[1..n], c[1..n], G)$ a câștigului optim pentru instanța $n, g[1..n], c[1..n], G$ dată ca parametru. În continuare, vom discuta o posibilitate simplă de a defini h. Spre sfârșitul cursului, vom prezenta o metodă mai elaborată, dar care va conduce la un algoritm branch-and-bound mai bun.

Cea mai simplă supraaproximare h a câștigului optim poate fi obținută în felul următor: $h(n, g[1..n], c[1..n], G) = c[1] + \ldots + c[n]$ (nu se poate obține un câștig mai mare decât suma câștigurilor tuturor obiectelor disponibile).

Ca și în cazul backtracking, algoritmii branch-and-bound definesc noțiunea de soluție parțială și de succesori imediați. Pentru problema rucsacului, o soluție parțială este un vector $s[1..j]$ (pentr un anumit $1 \leq j \leq n$) în care $s[i] = 0$ dacă am hotărât să nu folosim obiectul i și $s[i] = 1$ dacă am hotărât să folosim obiectul i ($1 \leq i \leq j$). Pentru obiectele $j + 1, \ldots, n$ nu am făcut încă o alegere.

Câștigul și respectiv greutatea unei soluții parțiale sunt date de suma câștigurilor și respectiv suma greutăților obiectelor alese. Dacă greutatea obiectelor alese într-o soluție parțială depășește greutatea rucsacului, considerăm câștigul ca fiind $-\infty$.

Soluția vidă este vectorul $s[1..0]$ (cu 0 elemente).

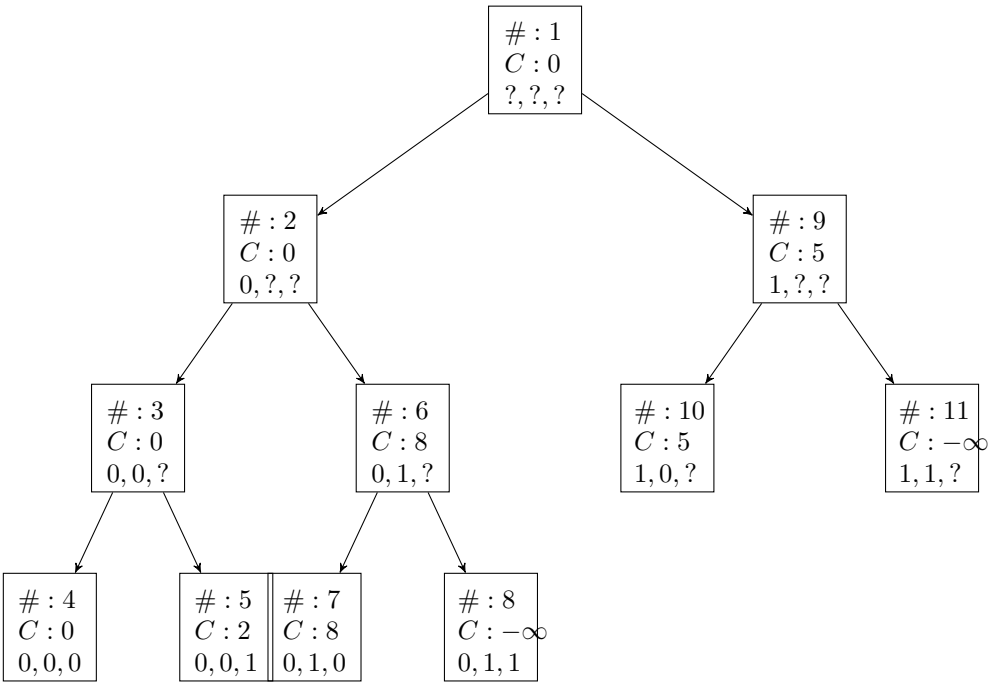
Algoritmul branch-and-bound începe, ca și în cazul algoritmilor de tip backtracking, cu soluția vidă. Spre deosebire de backtracking, pe parcursul explorării nodurilor, se menține o valoare B reprezentând câștigul maxim obținut până în momentul curent. Inițial, $B = -\infty$ (orice soluție are câștig mai bun decât B). Pe măsură ce sunt explorate nodurile (un nod reprezentând o soluție parțială), este actualizată valorea B. Optimizarea crucială în cazul algoritmilor branch-and-bound este: *dacă soluția parțială curentă nu are nicio șansă să conducă la o soluție completă de câștig mai mare decât B, nu se mai explorează subarborele curent.*

Cum se poate decide dacă soluția parțială curentă $s[1..j]$, care are un câștig C, nu poate duce la un câștig mai mare decât B? Putem calcula, folosind funcția h, o supra-aproximare a câștigului adus de obiectele $j+1, \ldots, n$ rămase disponibile: $X = h(n - j, g[j + 1..n], c[j + 1..n], G - g[1] - g[2] - \ldots - g[j])$. Soluția parțială nu poate duce la un câștig mai mare decât $C + X$. Astfel, dacă $B >= C + X$, ramura curentă poate fi uitată fără a exista riscul de a pierde vreo soluție optimă.

Ilustrăm algoritmul pe următorul exemplu, cu $n = 3$ obiecte și $G = 5$ (greutate maximă a rucsacului):

i	1	2	3
$g[i]$	3	4	2
$c[i]$	5	8	2

Arborele de mai jos suprinde execuția algoritmului pentru exemplul de mai sus. Fiecare nod din arbore conține câmpul # (numărul de ordine al nodului), câmpul C (câștigul generat de soluția parțială) și un șir de $0, 1, ?$ reprezentând soluția parțială.



Observați că la nodurile 10 și 11 are loc o "retezare" deoarece aceste soluții parțiale nu mai pot fi extinse astfel incât să conducă la o soluție de câștig mai mare decât valoarea lui B:

1. în momentul în care algoritmul ajunge la nodul 10, valoarea lui B este 8 (cel mai bun câștig observat în nodurile 1..9). În nodul 10, configurația parțială este $1, 0, ?$, adică am hotărât să alegem primul obiect, să nu îl alegem pe al doilea și încă nu am decis nimic pentru al treilea obiect. Câștigul parțial este 5 – valoarea primului obiect. Funcția h ne indică că, având la dispoziție doar obiectul 3, câștigul nu poate depăși 2. Astfel câștigul maxim realizat pe această ramură nu poate $5 + 2 \leq 8$; astfel nu mai are rost să explorăm ramura în continuare deoarece nu poate conduce la o soluție mai bună.

2. în momentul în care ajungem la nodul 11, valoarea lui B este 8. În nodul 11, valoarea parțială este $-\infty$ deoarece s-a depășit greutatea maximă admisă. Funcția h ne indică că nu putem câștiga mai mult de 2 folosind obiectele rămase. Deoarece $-\infty + 2 \leq 8$, această ramură nu poate duce la o soluție mai bună decât cea existentă și deci o putem ignora.

În general, schema algoritmului branch-and-bound este următoarea:

```
dfs(s) // s e o solutie partiala
{
  if (castig(s) + h(s) <= B) {
    // ignor aceasta ramura, deoarece nu poate
    // duce la o solutie mai buna decat cea existenta
  } else {
    // actualizez B - castigul adus de cea mai buna
```

```
    // solutie intalnita pana acum
    if (castig(s) < B) {
      B = castig(s);
    }

    for "toti successorii t ai lui s"
      dfs(t);
  }
}

B = -infinit;
dfs(empty); // incep cu solutia vida
return B;
```

Există și alte variante de branch-and-bound, în care în loc de dfs se folosește căutarea în lățime (breadth-first search) sau căutarea best-first search, dar ideea principală este să ignorăm ramurile care nu au șanse să conducă la o soluție mai bună decât soluția actuală.

Alegerea funcției h determină performanța algoritmului: cu cât funcția h este mai exactă, în sensul în care oferă un răspuns cât mai aproape de soluția optimă, cu atât mai multe ramuri ale arborelui de căutare sunt *retezate* și deci cu atât mai eficientă este căutarea.

Pentru problema rucsacului, o variantă mai bună pentru funcția h este dată de următoarea observație:

- Fie $n, c[1..n], g[1..n], G$ o instanță a problemei discrete a rucsacului și fie X soluția optimă pentru această instanță.

  Fie Y soluția optimă a problemei *continue* a rucsacului pentru aceeași instanță $n, c[1..n], g[1..n], G$ (orice instanță a problemei discrete a rucsacului este o instanță a problemei continue a rucsacului).

  Atunci $X \leq Y$ (exercițiu: demonstrați această inegalitate).

Putem alege deci ca supra-aproximare a soluției optime funcția h care calculează (folosind algoritmul greedy) soluția optimă pentru problema continuă a rucsacului.

Exercițiu: construiți arborele de căutare pentru algoritmul branch-and-bound pentru problema discretă a rucsacului folosind funcția h de mai sus.

Exercițiu: scrieți un algoritm de tip branch-and-bound care rezolvă problema 15-puzzle (n^2-1-puzzle în general). Atenție! Pentru problemele de minim (cum este 15-puzzle, unde se cere numărul minim de mutări, funcția h trebuie să fie o *subaproximare* a rezultatului optim).