# Outline

# Contents

# 1 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 rezolvată și prin metoda programării dinamice (obținând un algoritm pseudopolinomial), dar vom exemplifica cu ajutorul ei metoda branch-and-bound.

Pentru orice algoritm de tip 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).

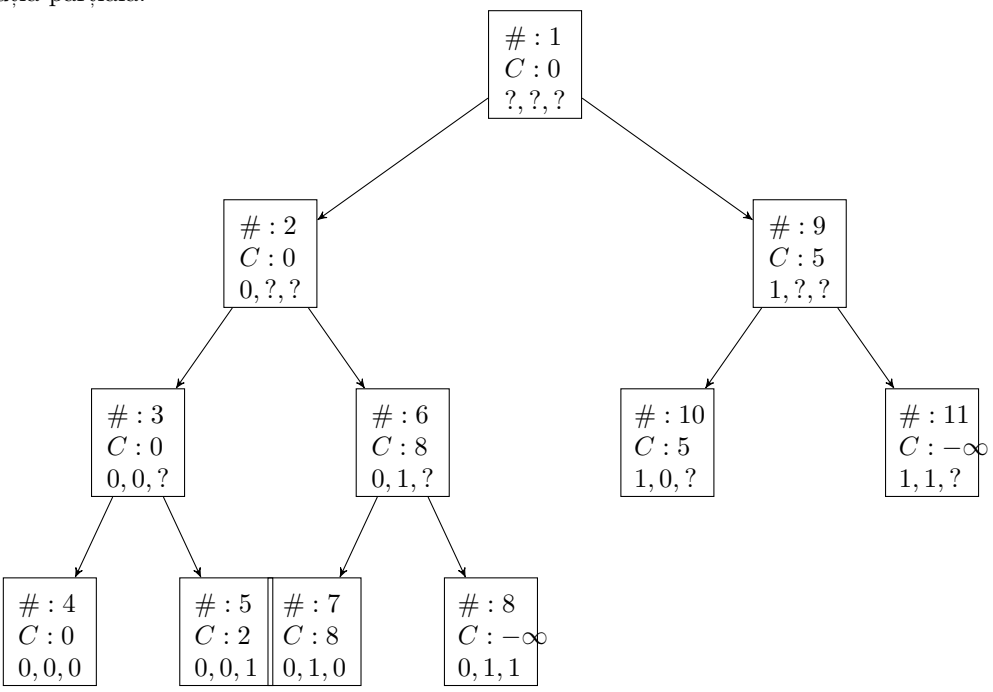
Algoritmul branch-and-bound începe, ca și în cazul algoritmilor de tip backtracking, cu soluția vidă. Spre deosebire de backtracking, pe parcursul explorării nodurilor, se menține o valoare $B$ reprezentând câștigul maxim obținut până în momentul curent. Inițial, $B = -\infty$ (orice soluție are câștig 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 complet(s) {
      if (castig(s) > B) {
        B = castig(s);
      }
    } else {
      for s' in succ(s)
       dfs(s');
    }
  }
}

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. Ideea principală rămâne: 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 (pentru obiectele ramase).

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).

# 2 Exercitii pentru seminar

1. Implementați în Alk algoritmul branch-and-bound pentru problema discretă a rucsacului.

2. Proiectați un algoritm de tip branch-and-bound pentru problema 15-puzzle (în general, $(n^2-1)$-puzzle). Specificați ce se înțelege prin soluție parțială, cum se calculează costul unei soluții parțiale, cum se poate mărgini inferior costul extensiei unei soluții parțiale și care sunt succesorii direcți ai unei soluții parțiale.

3. Proiectați un algoritm de tip branch-and-bound pentru problema Traveling Salesperson.

4. Proiectați un algoritm de tip branch-and-bound pentru problema Maximum Independent Set.