Proiectarea algoritmilor: *Backtracking* și *Branch-and-Bound*

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2013/2014

1. Introducere
2. Bactracking
   - Studii de caz
3. Branch-and-Bound
   - Studii de caz

Plan

Cum rezolvăm problemele $\mathcal{NP}$-complete?

Problemele $\mathcal{NP}$-complete sunt întâlnite frecvent în orice domeniu.

Pentru o problemă $\mathcal{NP}$-completă, știm că nu avem șanse să găsim algoritmi care să rezolve eficient (în timp polinomial (?)) toate instanțele.

Există două abordări majore:

1. Dacă interesează găsirea unei soluții exacte (e.g., SAT, problemele de decizie în general, dar pot fi și probleme de optim), atunci trebuie găsit un algoritm "exponențial eficient".

2. Dacă este acceptată și o soluție aproximativă (e.g., probleme de optim), atunci se recurge la algoritmi de aproximare sau euristici.

În acest curs discută două paradigme pentru cazul 1: bactracking și branch-and-bound.

Partea comună a modelului matematic

Fie P o problemă $\mathcal{NP}$-completă, x_0 o instanță a problemei P.

Un algoritm nedeterminist care rezolvă P are două etape: 1) ghicește o structură de date, și 2) verifică dacă structura ghicită este soluție.

Ambele paradigme caută "sistematic și inteligent" în spațiul structurilor de unde este "ghicită" soluția de algoritmul nedeterminist.

Configurație = o pereche (x, y), unde x este partea de instanță (subinstanța) de rezolvat iar y este mulțimea alegerilor ce trebuie făcute pentru a ajunge de la instanța inițială x_0 la subinstanța x.

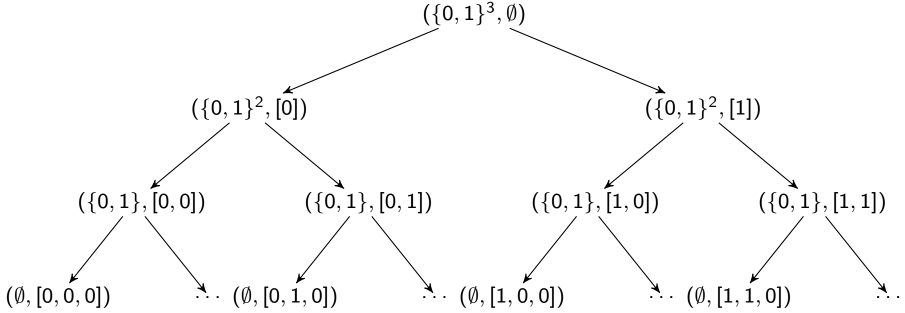
Configurațiile sunt organizate într-o structură arborescentă. Un nod (o configurație) este neviabil dacă subarborele cu rădăcina în acel nod nu include configurații ce descriu soluții.

"Explorarea inteligentă" a arborelui configurațiilor constă în găsirea configurațiilor neviabile pentru a nu mai parcurge subarborii acestora. Fără această parte "inteligentă" algoritmul devine unul "brute force", care explorează tot arborele (= spațiul soluțiilor fezabile)

Cele două paradigme diferă prin clasele de probleme pe care le rezolvă și modul de explorare a arborelui.

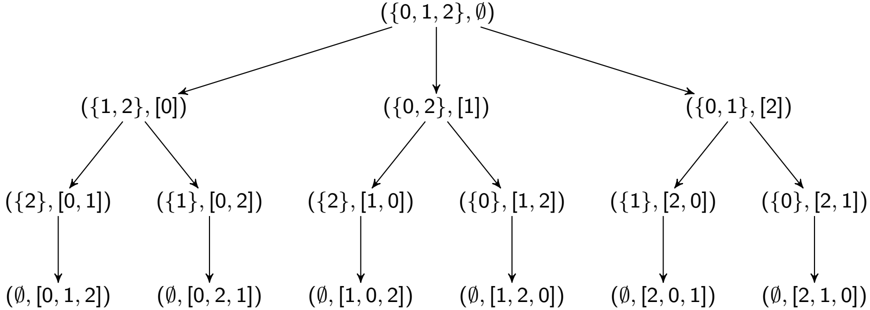
Exemplu de arbore de explorare: produs cartezian

Mulțimea $\{0, \ldots, m-1\}^n$ poate fi reprezentată printr-un arbore cu n nivele în care fiecare vârf intern are exact m succesori iar vârfurile de pe frontieră corespund elementelor mulțimii.



Exemplu de arbore de explorare: permutări

Spațiul soluțiilor este reprezentat ca un arbore în care orice drum de la rădăcină la frontieră descrie o permutare.



Plan

Prezentare intuitvă

- parcurge arborele configurațiilor pentru a găsi soluții sau noduri neviabile
- subarborii nodurilor neviabile nu mai sunt cercetați
- dacă toate configurațiile succesoare cele curente sunt neviabile, întrerupe căutările pentru această configurație și alege altă configurație nexplorată ("bactracks", de unde vine și numele metodei)
- configurația de start este $(x_0, \emptyset)$

Algoritmul generic

```
@input: o instanţă x0 a unei probleme NP-complete de decizie
@output: o soluţie x sau ”no solution” dacă nu există niciuna
bactrack(x0) {
  C = (x0, ∅);
  while (C != ∅) {
    selectează cea mai ”promiţătoare” configuraţie (x, y) ∈ C;
    expandează (x, y) făcând un număr mic de alegeri;
    fie (x1, y1), . . . , (xk, yk) noile alegeri;
    foreach (xi, yi) ales {
      verifică dacă (xi, yi) este consistentă;
      dacă verificarea întoarce ”solution”
        return soluţia derivată din (xi, yi);
      else dacă verificarea întoarce ”neviabilă”
        elimină (xi, yi); // backtrack
      else C = C ∪ {(xi, yi)};
  }
  return "no solution";
}
```

Instanțierea algoritmului generic

trebuie definit:

- exact ce e o configurație (x, y);
- ce înseamnă "cea mai promițătoare configurație";
- cum se expandează o configurație (x, y);
- cum se realizează un test simplu de consistență;
    - ce înseamnă configurație neviabilă;
    - când o configurație definește o soluție;

Submulțime de sumă dată

Se consideră o mulțime A cu n elemente, fiecare element $a \in A$ având o dimensiune $s(a) \in \mathbb{Z}_+$ și un număr întreg pozitiv M. Problema constă în a decide dacă există $A' \subseteq A$ cu proprietatea $\sum_{a \in A'} s(a) = M$.

Submulțime de sumă dată: backtracking

- configurație: (X, Y), $X =$ mulțimea obiectelor de unde se alege, $Y =$ mulțimea obiectelor alese
  inițial: $X = A$, $Y = \emptyset$
- cea mai promițătoare configurație: se poate lua următorul obiect a
- expandarea: configurațiile expandate sunt $(X \setminus \{a\}, Y \cup \{a\})$ (obiectul a este ales) și $(X \setminus \{a\}, Y)$ (obiectul a NU este ales);
- testul simplu de consistență:
  dacă $\sum\limits_{a \in X} s(a) \leq M$, s-a găsit soluție
  (X, Y) este viabilă dacă:
  - Suma parțială dată de prima parte (adică de candidații aleși) să nu depășească M:
  $$\sum_{a \in Y} s(a) \leq M \tag{1}$$
  - Ceea ce rămâne să fie suficient pentru a forma suma M:
  $$\sum_{a \in Y} s(a) + \sum_{b \in X} s(b) \geq M \tag{2}$$

SAT (CNF)

SAT

Fie $X = \{x_0, \ldots, x_{n-1}\}$ o mulțime de variabile. Un *literal* este o variabilă x sau negația sa $\overline{x}$. O atribuire este o funcție $\alpha : X \to \{0, 1\}$. Atribuirea α se extinde la literale astfel:

$$\alpha(u) = \begin{cases} \alpha(x) & \text{dacă } u = x \in X, \\ \neg\alpha(x) & \text{dacă } u = \overline{x}, x \in X. \end{cases}$$

O *clauză* este o mulțime finită c de literale. Clauza c este *satisfăcută* de atribuirea α dacă $\alpha(u) = 1$ pentru cel puțin un $u \in c$. Problema satisfiabilității constă în a determina dacă, pentru o secvență de clauze $C = (c_0, \ldots, c_{q-1})$, există o atribuire α care satisface orice clauză care apare în C.

SAT: bactracking naiv

- configurație: (F, α), F mulțime de clauze, α atribuire parțială
  inițial: $F = C, \alpha = \emptyset$
- cea mai promițătoare configurație: clauza de lungime cea mai mică (cu cei mai puțini literali), deoarece sunt șanse de a obține mai repede răspunsul "nesatisfiabil" (de ce?)
- expandarea: se alege un literal din clauza cea mai scurtă selectată și se generează configurațiile pentru $x_i = 0$ și respectiv $x_i = 1$, unde x_i este variabila din literalul ales; pentru fiecare alegere, F se simplifică prin înlocuirea literalilor ℓ ce includ x_i cu $\alpha(\ell)$, eliminarea lui 0 din clauze, eliminarea clauzelor care includ 1;
- testul simplu de consistență:
  dacă $F = \emptyset$, atunci F este satisfiabilă ("solution")
  configurație neviabilă: F include clauza vidă

SAT: algoritmul Davis-Putnam (DP, 1960)

regula de rezoluție:

$$\frac{(x, \ell_1, \ldots, \ell_m), (\overline{x}, \ell'_1, \ldots, \ell'_n)}{(\ell_1, \ldots, \ell_m, \ell'_1, \ldots, \ell'_n)}$$

- configurație: (F, Y), F mulțime de clauze, Y mulțimea de variabile alese
- cea mai promițătoare configurație: se alege o variabilă x care apare în F;
- expandarea: se aplică regula de rezoluție și se elimină cele acre includ pe x;
- testul simplu de consistență:
  dacă $F = \emptyset$, atunci F este nesatisfiabilă
  dacă Y include toate variabilele, atunci F este satisfiabilă (din Y se poate extrage un model (cum?))

SAT: algoritmul Davis-Putnam-Logemann-Loveland (DPLL, 1962)

Rafinează algoritmii naiv și DP.
literal pur: negația sa nu apare în F
clauza unitate - clauză cu un singur literal

- configurație: (F, α), unde α este reprezentat ca o mulțime de literali
- cea mai promițătoare configurație și expandarea:
  dacă F include o clauză unitate, atunci alege o astfel de clauză ℓ și expandează (alege) $(F|_\ell, \alpha \cup \{\ell\})$
  altfel dacă F include un literal pur ℓ, atunci alege ℓ și expandează (alege) $(F|_\ell, \alpha \cup \{\ell\})$
  altfel, alege un literal ℓ dintr-o clauză de lungime minimă și expandează (alege) $(F|_\ell, \alpha \cup \{\ell\})$ și $(F|_\ell, \alpha \cup \{\bar{\ell}\})$

unde $F|_\ell$ este F simplificat prin eliminarea valorilor 0 din clauze si clauzelor 1 obtinute după înlocuirea varibilei din ℓ cu valoarea corespunzatoare ($x \mapsto 1$ daca $\ell = x$ și $x \mapsto 0$ daca $\ell = \bar{x}$).

Plan

Intuitiv

Modelul este asemănător celui de la *backtracking*, cu următoarele diferențe:

- se aplică problemelor de optim; fie $f(x)$ funcția de optimizat
  pentru a simplifica prezentarea, presupunem $f(x)$ funcție de profit (deci probleme de maxim);
- o configurație este neviabilă dacă nu satisface constrângerile problemei;
- dacă algoritmul backtracking se opește la prima soluție găsită, *branch-and-Bound* continuă până o găsește pe cea mai bună;
- se consideră în plus o funcție $gb(x, y)$ care întoarce o margine superioară pentru orice soluție derivată din configurația (x, y);
- o configurație viabilă (x, y) este expandată numai dacă $gb(x, y)$ este mai mare decât profitul celei mai bune soluții b calculată până atunci;

Algoritmul generic

```
@input: o instanţă x0 unei probleme de optimizare NP-complete
@output: o soluţia optimă x sau ”no solution” dacă nu există niciuna
branchAndBound(x0) {
  C = (x0, ∅);  b = (−∞, ∅)
  while (C != ∅) {
    selectează cea mai ”promiţătoare” configuraţie (x, y) ∈ C;
    expandează (x, y) producând (x1, y1), . . . , (xk, yk);
    foreach (xi, yi) nou {
      verifică dacă (xi, yi) este consistentă;
      dacă verificarea întoarce ”solution”
        if (profitul p al lui (xi, yi) e mai bun decât b)
          b = (p, (xi, yi));
        else elimină (xi, yi);
      if (verificarea întoarce ”neviabilă”)
        elimină (xi, yi); // backtrack
      else
        if (gb(xi, yi) este mai mare decât profitul lui b)
          C = C ∪ {(xi, yi)}; // (xi, yi) porneşte o căutare promiţătoare
        else elimină (xi, yi); // ”bound prune”
  }
  return b;
}
```

RUCSAC 0/1

Reamintire:
Se consideră n obiecte $1, \ldots, n$ de dimensiuni (greutăți) $w_1, \ldots, w_n \in \mathbb{Z}_+$, respectiv, și un rucsac de capacitate $M \in \mathbb{Z}_+$. Un obiect i sau este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus de loc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvență $x_1, \ldots, x_n$ cu $x_i \in \{0, 1\}$ și $\sum_i x_i \cdot w_i \leq M$. Introducerea obiectului i în rucsac aduce profitul $p_i \in \mathbb{Z}$ iar profitul total este $\sum_{i=1}^{n} x_i p_i$. Problema constă în a determina o alegere $(x_1, \ldots, x_n)$ care să aducă un profit maxim.

RUCSAC 0/1

Formulare ca problemă de programare matematică:

- funcția obiectiv:

$$\max \sum_{i=1}^{n} x_i \cdot p_i$$

- restricții:

$$x_i \in \{0,1\}, 1 \le i \le n$$
$$\sum_{i=1}^{n} x_i \cdot w_i \le M$$

RUCSAC 0/1: algoritm branch-and-bound

- configurație: $(A, (X, B))$, A mulțimea de obiecte din care se alege, X capacitatea de umplut, B mulțime obiectelor alese
  inițial $A = \{1, \ldots, n\}$, $B = \emptyset$, $X = M$;

- cea mai promițătoare configurație și expandarea: obiectele se aleg în ordinea descrescătoare a profiturilor pe unitatea de greutate;
  dacă obiectele sunt sortate după acest criteriu, atunci $B \subseteq \{1, \ldots, k\}$ și $A = \{k + 1, \ldots, n\}$;
  dacă se alege obiectul i, noua configurație va fi $(A \setminus \{i\}, (X - w_i, B \cup \{i\}))$; dacă obiectul i nu este ales, noua configurație va fi $(A \setminus \{i\}, (X, B))$;

- testul simplu de consistență:
  configurație neviabilă: $\sum_{i \in B} x_i \cdot w_i > M$

RUCSAC 0/1: algoritm branch-and-bound

- marginea superioară *gb*: această margine poate fi obținută dacă relaxăm problema în modul următor:
  - funcția obiectiv:
$$\max \sum_{i \in A} x_i \cdot p_i$$
  - restricții:
$$0 \le x_i \le 1,\ \ i \in A$$
$$\textstyle\sum_{i \in A} x_i \cdot w_i \le X$$

Valorile pentru x_i cu $i \in B$ sunt deja atribuite. Se rezolvă problema relaxată utilizând algoritmul greedy, obținându-se o margine superioară *gb* (profitul maxim care se poate realiza cu $(x_i, \mid i \in B)$ deja atribuite). De fapt *gb* este $P_B + \sum\limits_{i \in B} x_i \cdot p_i$, unde P_B este profitul optim întors de lgoritmul greedy.