Retour-en-arrière et branch-and-bound

Pour résoudre un problème, on peut représenter notre recherche de solutions comme un graphe où chaque noeud contient une solution partielle de notre problème et chaque arête représente une façon d'étendre cette solution

On appelle graphe implicite un graphe pour lequel on a une description des sommets et des arêtes mais qui n'est pas gardé complètement en mémoire.

Seulement les parties intéressantes de ce graphe seront construites lors de la recherche d'une solution

Les méthodes de retour-en-arrière et de branch-and-bound utilisent cette idée de graphe implicite.

Par contre ici le graphe est toujours un arbre

La racine de cet arbre contient la solution partielle nulle.

Retour-en-arrière: Problème du sac à dos

Problème:

On dispose de n types d'objets de poids positifs $w_1, w_2, \ldots, w_n$ et de valeurs positives $v_1, v_2, \ldots, v_n$. Notre sac à doc à une capacité maximale en poids de W. Ici, on suppose qu'on peut choisir autant d'objets de chaque type qu'on veut.

But: Maximiser $\sum_{i=1}^{n} x_i v_i$ tel que $\sum_{i=1}^{n} x_i w_i \leq W$

avec $v_i, w_i > 0$ et $x_i \geq 0$

Retour-en-arrière: Problème du sac à dos

Représenter ce problème sous forme d'arbre:

- chaque noeud correspond à une solution partielle du problème avec poids; valeur

  poids = l'ensemble des poids des objets couramment dans la sac

  valeur = la valeur totale de ces objets

- chaque arête d'un sommet à son fils i, correspond à la décision d'inclure un objet de plus de type i dans le sac

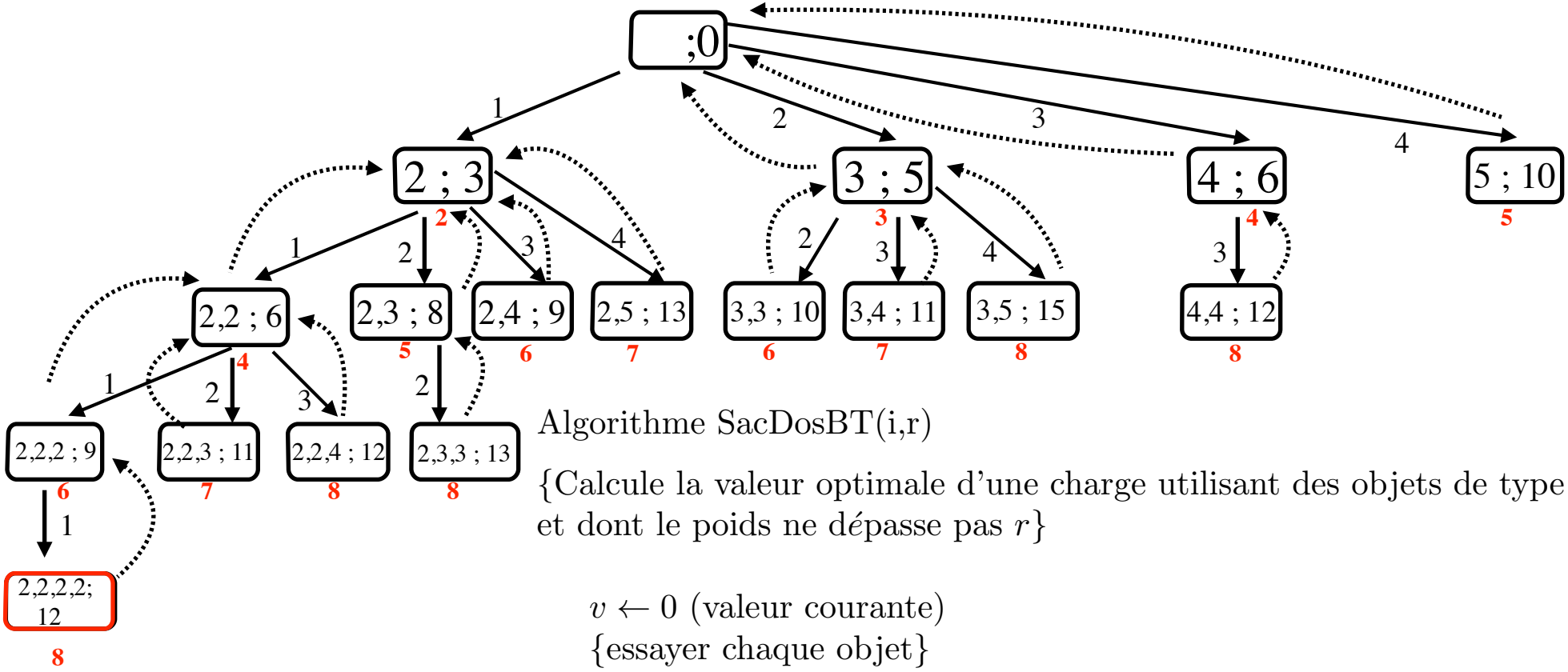
  Sans perte de généralité, on peut ordonner les types d'objets par ordre croissant de poids (solution type 1, 1, 2 = 1, 2, 1 = 2, 1, 1)

  Pour un fils i, enfants de i à n seulement

- un algorithme de retour-en-arrière pour ce problème commence à la racine (solution nulle) et explore l'arbre en profondeur (parcours préfixe) en construisant les noeuds et solutions partielles au fur et à mesure

Retour-en-arrière: Problème du sac à dos

Exemple: Supposons qu'on ait 4 types d'objets de poids respectifs 2, 3, 4 et 5 et de valeurs 3, 5, 6 et 10 et supposons que W=8.



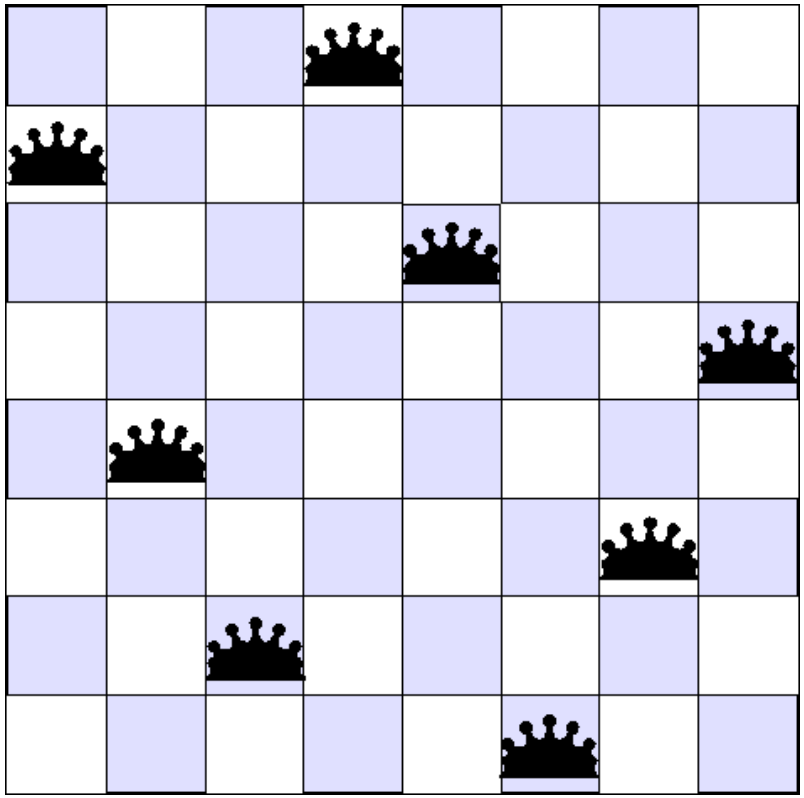
Algorithme SacDosBT(i,r)

{Calcule la valeur optimale d'une charge utilisant des objets de type i *à* n et dont le poids ne dépasse pas r}

$v \leftarrow 0$ (valeur courante)
{essayer chaque objet}
Pour k de i *à* n faire
  Si $w[k] \leq r$ alors
    $v \leftarrow \max(v, v[k] + \text{SacDosBT}(k, r - w[k])$
  Fin Si
Fin Pour
Retouner v

Retour-en-arrière: Problème des 8 reines

Problème: Placer 8 reines sur un échiquier de sorte qu'aucune reine n'en menace une autre. (Une reine menace toutes les pièces positionnées dans la même rangée, la même colonne ou la même diagonale).



Parallel Programming in C for the Transputer
© D. Thiébaut, 1995

Retour-en-arrière: Problème des 8 reines

Idée: Essayer toutes les façons de placer les 8 reines sur l'échiquier, en regardant, à chaque fois, si on a une solution.

$$\binom{64}{8} = 4\ 426\ 165\ 368 \text{ façons}$$

Amélioration 1: Ne placer qu'une reine par ligne

- on peut représenter les positions des reines par un vecteur de taille 8
- l'entrée à la position i du vecteur représente la colonne dans laquelle la reine de la ligne i doit être positionnée
- $8^8 = 16\ 777\ 216$ façons
- si on implante cette idée sous forme de 8 boucles imbriquées, une première solution est trouvée après avoir considéré 1 299 852 positionnements.

Retour-en-arrière: Problème des 8 reines

Algorithme Reine1

Pour i_1 de 1 $\grave{a}$ 8 faire
  Pour i_2 de 1 $\grave{a}$ 8 faire
    $\vdots$
    Pour i_8 de 1 $\grave{a}$ 8 faire
      sol $\leftarrow [i_1, i_2, \ldots, i_8]$
      Si **sol** est une solution alors
        retourner sol et stop
      Fin Si
    Fin Pour
    $\vdots$
  Fin Pour
Fin Pour

Retour-en-arrière: Problème des 8 reines

Amélioration 2: Ne placer qu'une reine par ligne ET par colonne

- on peut représenter les positions des reines par une permutation de $\{1, 2, 3, 4, 5, 6, 7, 8\}$

- 8! = 40 320 façons de positionner les 8 reines

- Dépendant de la façon de générer les permutations, aussi peu que 2830 positionnements de reines peuvent être considérés avant de trouver une solution.

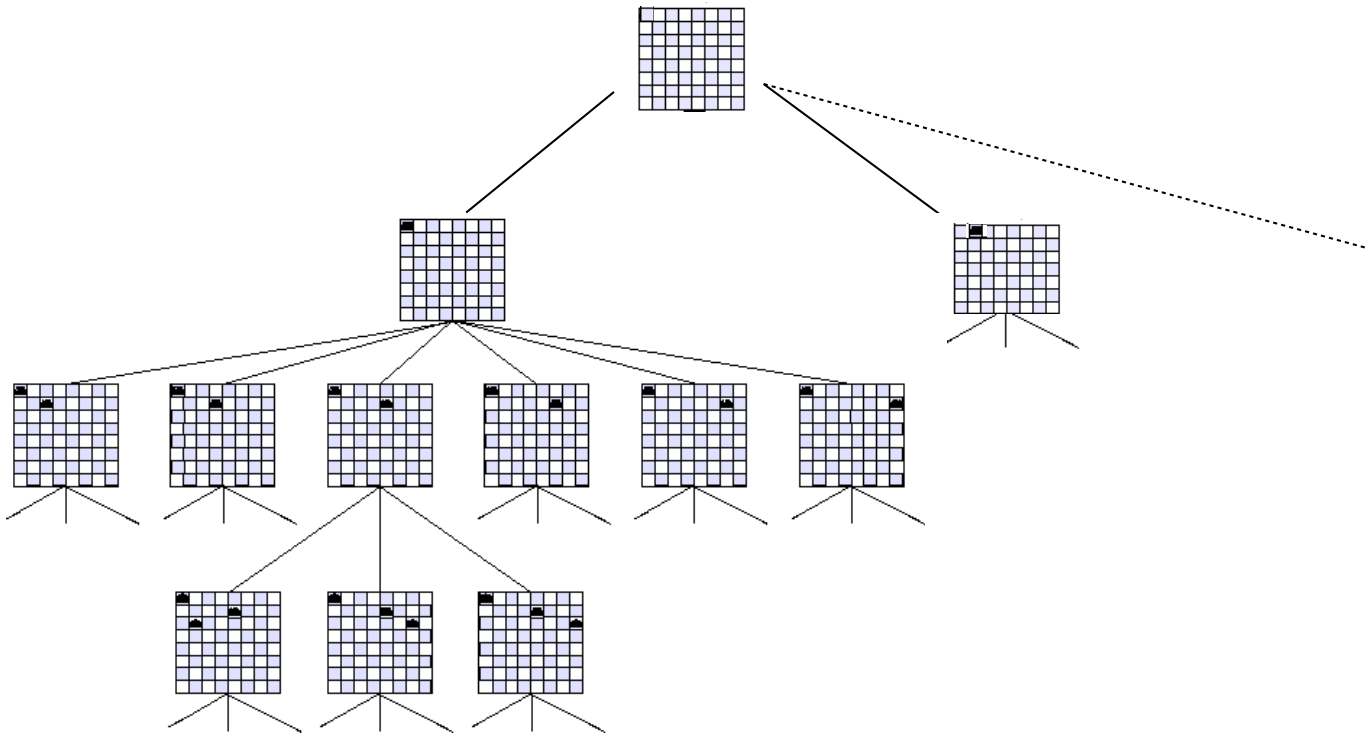
Algorithme Reine2

sol ← permutation_initiale
Tant que sol ≠ permutation_final et que sol n'est pas une solution **faire**
    sol ← prochaine_permutation
    Si sol est une solution **alors**
        Retourner sol et stop
    Fin si
Fin Tant que

Retour-en-arrière: Problème des 8 reines

Problème: Toutes les solutions considérées jusqu'à présent attendent d'avoir positionné les 8 reines avant de regarder si le positionnement est une solution ou non

Solution: Définir le problème comme un problème de retour-en-arrière



Parallel Programming in C for the Transputer
© D. Thiébaut, 1995

Branch-and-bound

L'idée ici est aussi d'explorer un graphe implicite (souvent un arbre) pour trouver la solution optimale à un problème d'optimisation.

À chaque noeud on calcule une borne pour les valeurs des solutions découlant de ce noeud.

Si cette borne démontre que les solutions découlant de ce noeud seront nécessairement non optimales, alors on n'explore pas cette partie du graphe.

Branch-and-bound - Problème d'assignation

On a n agents et n tâches. Chaque agent doit exécuter une et une seule tâche et le coût lié au fait que l'agent i exécute la tâche j est c_{ij}.

Étant donné la matrice des coûts, on veut assigner à chaque agent une tâche de sorte de minimiser le coût total.

$n!$ possibilités $\longrightarrow$ on utilise branch-and-bound

Branch-and-bound - Problème d'assignation

Exemple: Disons qu'on a 4 tâches (1, 2, 3, 4) et 4 agents (a,b,c,d)

	1	2	3	4
a	**11**	**12**	18	40
b	14	**15**	**13**	**22**
c	11	17	**19**	23
d	17	14	20	**28**

- On trouve une borne supérieure au problème en calculant sa valeur sur une solution aléatoire:

  a:1, b:2, c:3, d:4 est une solution

  coût: 11+15+19+28 = 73

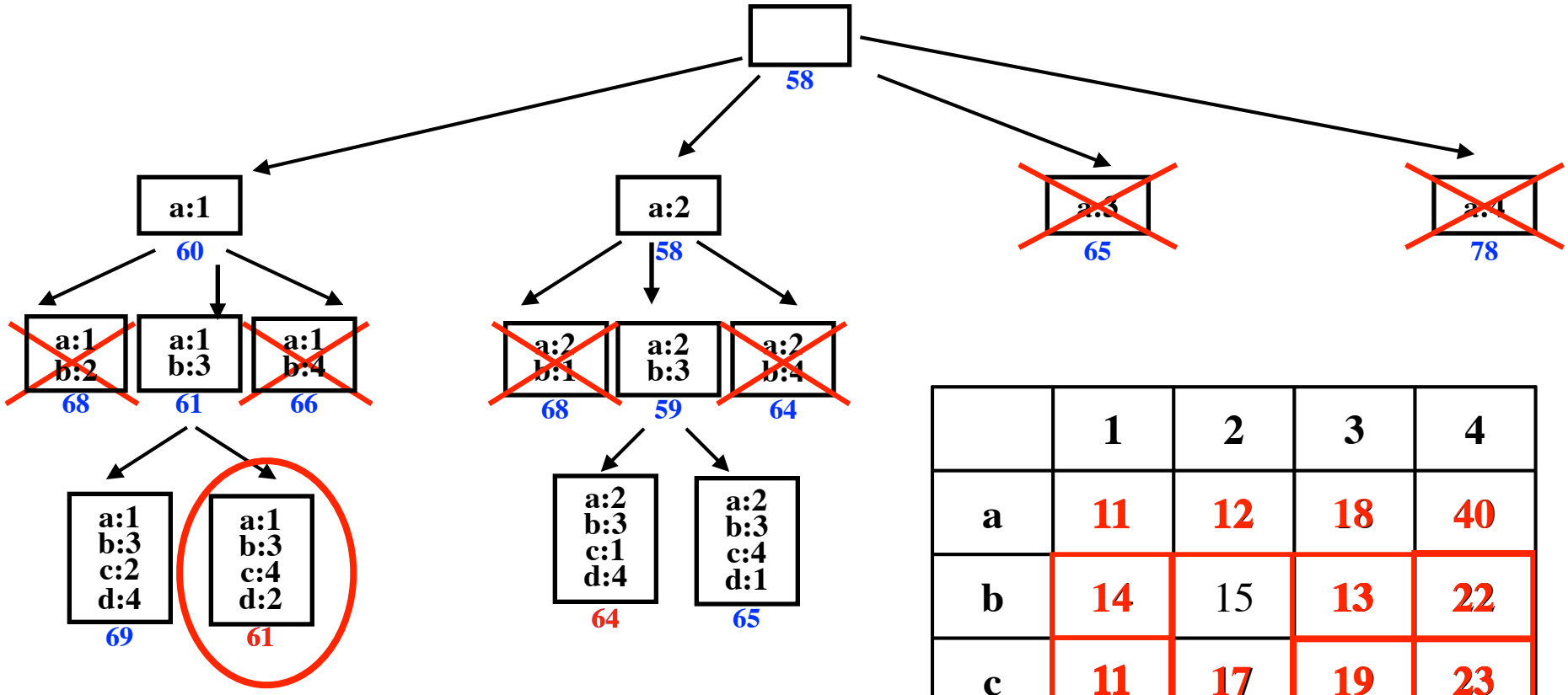
- Une borne inférieure peut être calculée en prenant la plus petite valeur de chaque colonne i.e. le coût minimum pour chaque tâche (attention, ce n'est pas nécessaire une solution)

  coût: 11+12+13+22 = 58

Branch-and-bound - Problème d'assignation

On va explorer l'arbre implicite des solutions partielles et on va calculer, pour chaque noeud, une borne inférieure pour les solutions découlant de ce noeud. Si cette borne est supérieure à la valeur courante de notre solution alors, on ne continue pas l'exploration à partir de ce noeud

valeur courante de notre solution: 74 64



	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28