



## Chapitre 11

---

### Faire face aux limitations algorithmiques



## Aperçu du chapitre

---

- Que fait-on lorsque notre problème à résoudre est NP-difficile?
- Deux principales options s'offrent à nous pour les **problèmes d'optimisation**:
  - Tenter de trouver la solution optimale par une méthode d'exploration de type « **branch and bound** »
    - Le temps d'exécution sera nécessairement exponentiel (ou plus) en pire cas mais il peut être « raisonnable » pour plusieurs instances qui ne font pas parti des pires cas.
  - Tenter de trouver une bonne solution sans qu'elle soit optimale à l'aide d'un **algorithme d'approximation** dont le temps d'exécution est polynomial en pire cas.
- Nous examinerons ces deux approches dans ce chapitre.
- Il n'existe pas de solution approximative pour les **problèmes de décision**.
- Cependant, la méthode d'exploration du **retour arrière** (« **backtracking** ») permet parfois d'obtenir une solution en un temps « raisonnable » pour des instances « faciles » des problèmes NP-complets.
  - Examinons d'abord cette approche.



## Le retour arrière (« backtracking »)

---

- C'est une version « intelligente » d'une recherche exhaustive.
- L'idée principale est de construire des **solutions partielles** en ajoutant une **composante** à la fois de la manière suivante:
  - Si la prochaine composante de disponible peut-être ajoutée à notre solution partielle sans violer les contraintes du problème, nous l'ajoutons à notre solution partielle et continuons.
  - S'il n'existe pas une telle composante de disponible, alors aucune solution ne peut être obtenue à partir de cette solution partielle.
    - L'algorithme fait alors un **retour arrière** (« **backtrack** ») et remplace la dernière composante (de notre solution partielle) avec la prochaine composante légitime de disponible.
      - Légitime signifie ici que cette nouvelle solution partielle ne viole pas les contraintes du problème.



## Le retour arrière (suite)

---

- Nous avons alors un **arbre de solutions partielles** à explorer.
- La racine de cet arbre contient habituellement 0 composante.
- Les nœuds du premier niveau sont les choix possibles pour la première composante.
- Les nœuds du k-ième niveau sont les choix possibles pour la k-ième composante.
- Chaque nœud de l'arbre représente donc une solution partielle: celle obtenue en concaténant les composantes obtenues en parcourant l'arbre de la racine au nœud en question.
  - Un nœud est qualifié de **prometteur** si sa solution partielle peut supporter l'ajout d'une autre composante.
  - Autrement, le nœud est qualifié de **non prometteur**.
- Un nœud non prometteur est donc soit un '**cul-de-sac**' (« **dead end** ») ou une **solution complète** du problème.



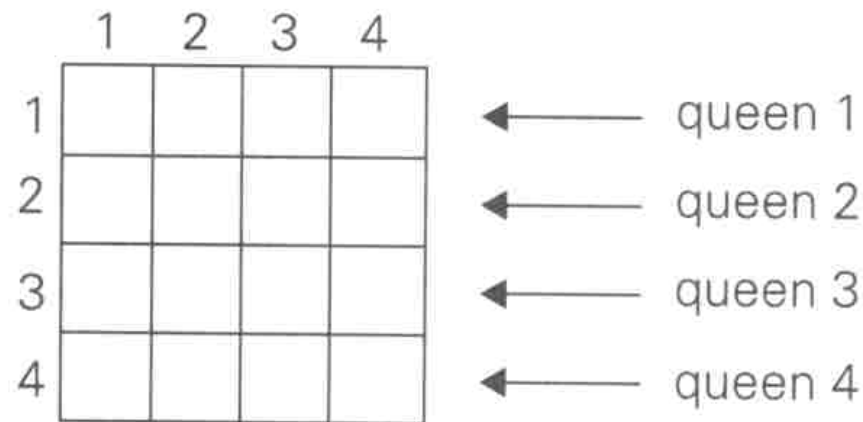
## Retour arrière et fouille en profondeur

---

- Dans la majorité des cas, l'arbre des solutions partielles est parcouru en profondeur (« depth-first »).
  - Si le nœud courant est prometteur, un enfant est généré en ajoutant la prochaine composante légitime et nous explorons ensuite cette nouvelle solution partielle (à une composante de plus).
  - Si le nœud courant est non prometteur, l'algorithme retourne en arrière au nœud parent et l'on remplace l'enfant par la prochaine composante légitime.
    - Si cette prochaine composante légitime n'existe pas, alors l'algorithme retourne en arrière un niveau de plus dans l'arbre...
- Lorsqu'une solution complète est trouvée, l'algorithme peut continuer (si on le désire) pour trouver d'autres solutions complètes.
  - Dans ce cas, l'algorithme termine uniquement lorsqu'il retourne jusqu'à la racine (et qu'il a 'épuisé' le niveau 1).

## Exemple: le problème des n reines

- Nous devons positionner  $n$  reines sur un jeu d'échecs  $n \times n$  de manière à ce qu'aucune reine en attaque une autre.
- Résolvons le cas  $n = 4$  par la technique du retour arrière.
- Puisque chaque reine doit être sur une rangée distincte, il ne reste qu'à assigner une colonne à chaque reine.



**FIGURE 11.1** Board for the four-queens problem

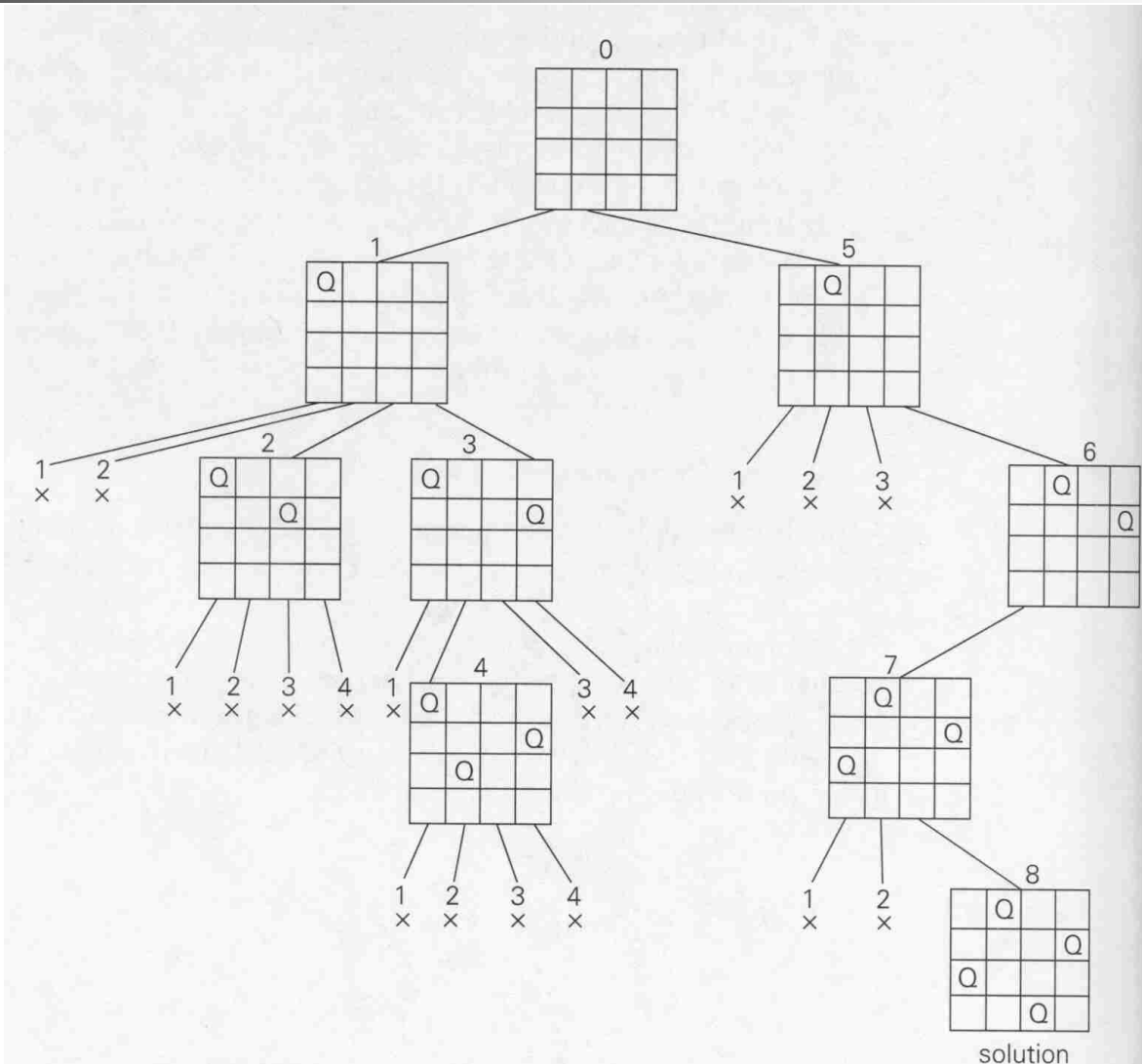


## Le problème des n reines (suite)

---

- La racine représente le jeu vide avec 0 reine de positionnée.
- Au niveau 1 on retrouve les colonnes possibles pour la reine 1
- Au niveau k on retrouve les colonnes possibles pour la reine k
- Après avoir positionné la reine 1 en colonne 1, on positionne la reine 2 en colonne 3 (après avoir refusé les colonnes 1 et 2)
  - Cette solution conduit à un cul-de-sac. **(voir figure page suivante)**
    - Alors on retourne en arrière d'un niveau et nous positionnons la reine 2 en colonne 4.
      - La reine 3 est ensuite positionnée en colonne 2 (c'est sa seule position légitime).
        - Cela conduit également à un cul-de-sac
  - Alors : retour arrière pour positionner la reine 1 en colonne 2
    - Ensuite la reine 2 en colonne 4
      - Puis la reine 3 en colonne 1
        - Finalement la reine 4 en colonne 3. **Solution!**

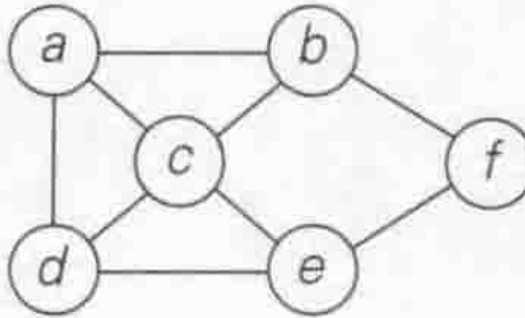
# Le problème des n reines (suite)



**FIGURE 11.2** State-space tree of solving the four-queens problem by backtracking. x denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

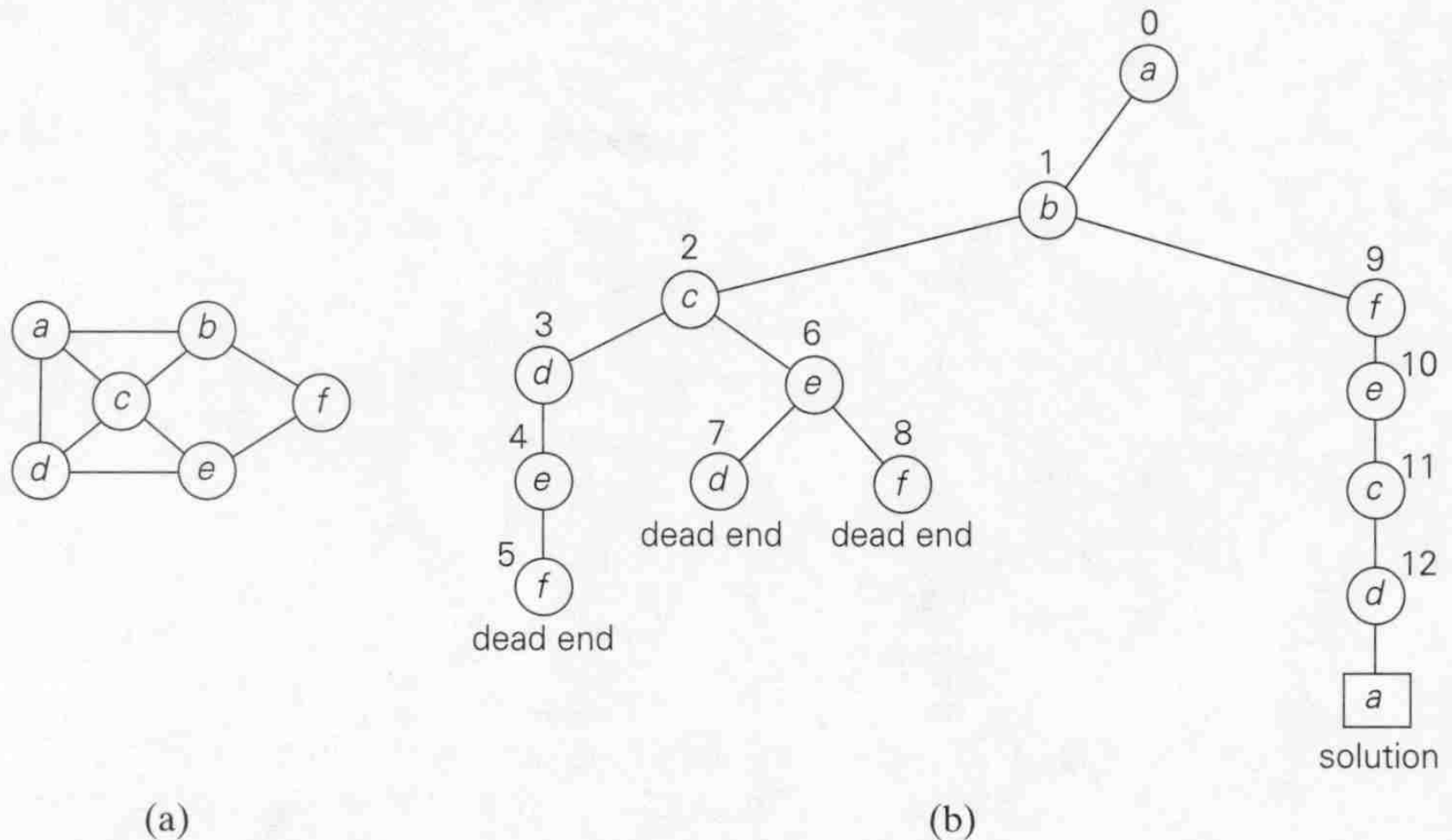


## Exemple: trouver un cycle hamiltonien



- Sans perte de généralité, nous pouvons supposer que si un cycle hamiltonien existe, alors il doit commencer au nœud A.
  - Le nœud A occupera donc la racine de l'arbre.
- La première composante, si elle existe, sera un nœud adjacent à A.
  - S'il y en a plusieurs, alors on utilise le premier selon l'ordre alphabétique. Nous choisirons alors B.
  - Ensuite nous irons à C, puis D, E et F. Cul-de sac!
  - Il faut alors retourner jusqu'à C.
- L'arbre d'exploration est illustré à la page suivante.

## Trouver un cycle hamiltonien (suite)



**FIGURE 11.3** (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.



## Retour arrière : remarques générales

- De manière générale, la sortie d'un algorithme de retour arrière est un n-tuple  $(x_1, x_2, \dots, x_n)$  où chaque composante  $x_i$  appartient à un ensemble fini et ordonné  $S_i$ . **Indice = reine, la valeur = num de colonne**
  - Pour les n reines: chaque  $S_i = \{1, \dots, n\}$  = ensemble des numéros de colonne satisfaisant les contraintes définies par la position des reines précédentes  $x_1, \dots, x_{i-1}$ .
  - Pour cycle hamiltonien:  $S_i =$  ensemble des nœuds adjacents à  $x_{i-1}$ .
- Un algorithme de retour arrière génère explicitement ou implicitement un arbre de solutions partielles  $(x_1, x_2, \dots, x_i)$  selon le pseudo-code suivant :

**Algorithme** *Backtrack*( $X[1..i]$ ) //premier appel avec  $i=0$

//Entrée:  $X[1..i]$  = solution partielle constituée des  $i$  premières composantes.

// $X[1..0]$  est le n-tuple vide.

// *Backtrack*( $X[1..0]$ ) affiche toutes les solutions.

**if**  $X[1..i]$  est une solution **write**  $X[1..i]$  // ok si le n-tuple est solution

**else** // on suppose qu'une solution n'est jamais préfixe d'une autre solution.

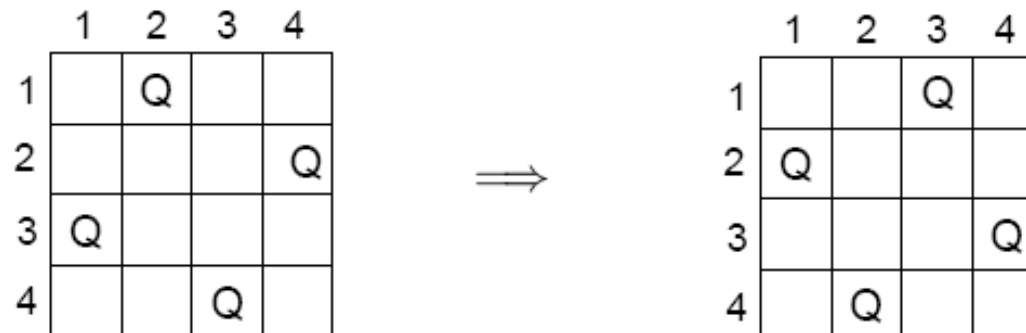
**for** each  $x \in S_{i+1}$  satisfaisant les contraintes définies par  $X[1..i]$  **do**

$X[i+1] \leftarrow x$

*Backtrack*( $X[1..i+1]$ )

## Retour arrière : remarques générales (suite)

- Cette méthode est typiquement utilisée pour résoudre des problèmes combinatoires difficiles.
  - Le temps d'exécution sera exponentiel en pire cas mais cette technique d'exploration est suffisamment intelligente pour espérer obtenir des temps d'exécution raisonnables sur plusieurs instances pas trop « difficiles ».
  - De plus, il est souvent possible d'exploiter une certaine symétrie pour diminuer la taille de l'arbre de recherche
    - Ex: pour les  $n$  reines, nous pouvons restreindre la position de la reine 1 aux  $\lfloor n/2 \rfloor$  premières colonnes car les solutions où la reine 1 occupe les autres colonnes sont obtenues par réflexion autour de la colonne (ou l'axe) centrale.





# Problèmes d'optimisation

---

- Un **problème d'optimisation** consiste à trouver un **objet (ou point)** qui **optimise** (minimise ou maximise) **une fonction** (souvent appelée **fonction objectif**)
  - L'objet (ou point) doit satisfaire certaines **contraintes** spécifiées par le problème
- Exemple: le commis voyageur. Ayant un graphe où chaque paire de nœuds est reliée par une arête possédant une distance, trouver le circuit de longueur minimale.
  - Chaque objet (ou point) est ici un cycle qui doit satisfaire la contrainte d'être un cycle hamiltonien : c.-à-d., passer par chaque nœud une seule fois.
- Un objet (ou point) satisfaisant les contraintes du problème est appelé une **solution réalisable** (« feasible solution »).
- L'objectif d'un problème d'optimisation est de trouver une **solution optimale**: une solution réalisable qui optimise la fonction objectif.

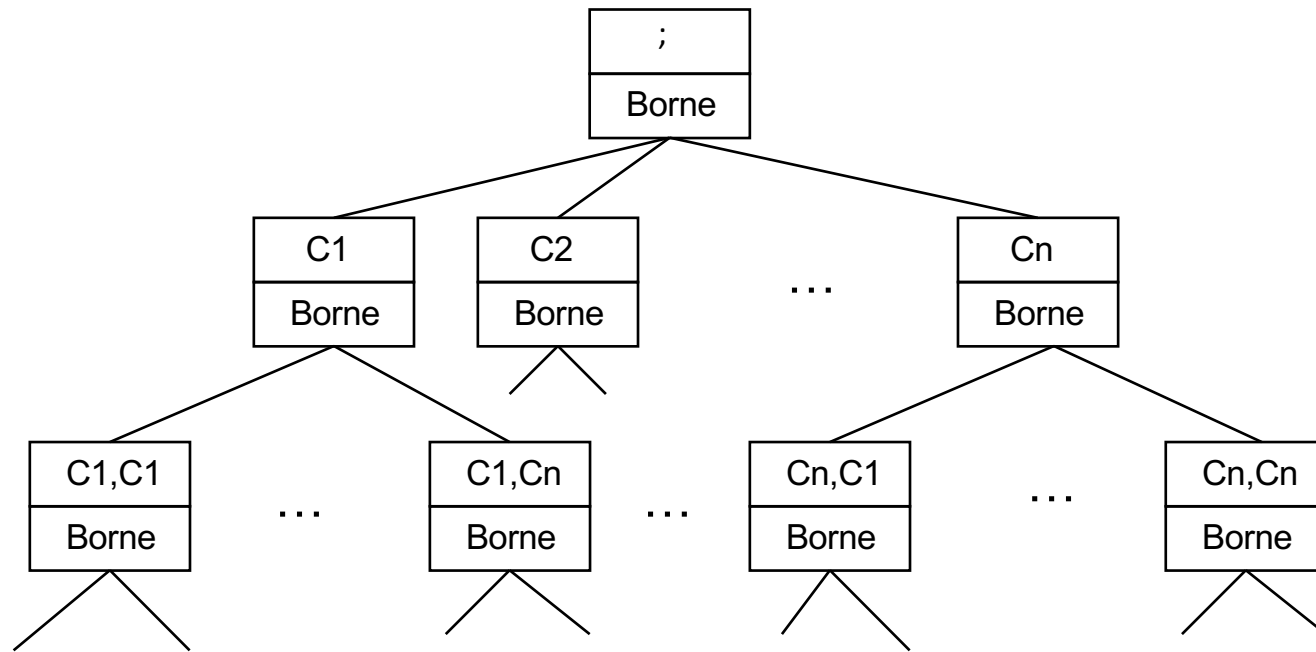


## La méthode « branch and bound »

---

- C'est une méthode d'exploration (de solutions) pour problèmes d'optimisation
- L'idée de base est de construire des solutions en ajoutant une **composante** à la fois et en évaluant chaque **solution partielle** ainsi obtenue.
  - Une solution partielle n'est habituellement pas une solution réalisable: elle ne satisfait pas nécessairement les contraintes
- Nous débutons avec une solution partielle contenant 0 composante
  - Ce sera la racine d'un **arbre de solutions partielles**
- Ensuite nous générons toutes les solutions partielles à une composante
  - C'est le niveau 1 de l'arbre des solutions partielles
- Pour chaque nœud (solution partielle) du niveau 1 nous pourrions générer des solutions partielles à deux composantes où la première composante est celle du nœud du niveau 1

# Arbre des solutions partielles



- Pour chaque solution partielle nous calculons une **borne** sur la meilleure valeur possible de la fonction objectif atteignable à partir de ce noeud.
  - C'est une borne inférieure pour un problème de minimisation
  - C'est une borne supérieure pour un problème de maximisation
- **La borne est une valeur de la fonction objectif qui est impossible d'améliorer avec les solutions construites à partir de cette solution partielle.** (la valeur de la borne n'est pas nécessairement atteignable)



## Terminaison d'une branche de l'arbre

- Cette méthode maintient, en tout temps, la valeur  $f_m$  (de la fonction objectif) de la meilleure solution obtenue jusqu'à maintenant.
- Une **branche est terminée** lorsque la valeur  $B$  de la borne d'une solution partielle n'est pas meilleure que  $f_m$ 
  - Car, dans ce cas, il est impossible d'obtenir une solution qui est meilleure que celle que nous avons déjà en générant des solutions à partir de cette solution partielle
    - Pour un problème de minimisation: la branche est terminée lorsque  $B \geq f_m$
    - Pour un problème de maximisation: la branche est terminée lorsque  $B \leq f_m$
- Une branche de l'arbre est également terminée lorsque la solution partielle ne peut plus générer d'autres solutions réalisables.
  - Cas 1: une solution réalisable est obtenue et nous ne pouvons plus en obtenir d'autres (à partir de ce nœud)
  - Cas 2: une des contraintes du problème est violée par cette solution partielle (et donc par toutes les autres issues de ce nœud)





## Exemple: le problème de l'assignation de tâches

- Il faut assigner  $n$  tâches à  $n$  personnes (1 personne par tâche)
- Chaque instance de ce problème est représentée par une matrice  $C$  de coûts.
- $C[p,t]$  indique le coût d'assigner la personne  $p$  à la tâche  $t$ .
- Chaque rangée de  $C$  représente une personne
- Chaque colonne de  $C$  représente une tâche
- Pour chaque rangée  $p$  il faut choisir une seule colonne  $t$  telle que la somme des coûts est minimal.
- Exemple:

	Job 1	Job 2	Job 3	Job 4	
$C =$	9	2	7	8	Person <i>a</i>
	6	4	3	7	Person <i>b</i>
	5	8	1	8	Person <i>c</i>
	7	6	9	4	Person <i>d</i>



## Assignation de tâches (suite)

---

- Chaque composante d'une solution partielle sera l'assignation d'une tâche à une personne. Exemple  $b \rightarrow 3$ .
- Chaque solution partielle sera constituée d'une séquence de composantes avec la contrainte que chaque personne et chaque tâche n'apparaisse qu'une seule fois dans une solution partielle.
- Cette contrainte est satisfaite en ayant un arbre de solutions partielles où chaque niveau assigne des tâches à une seule personne
  - Le niveau 1 pour la personne a, le niveau 2 pour la personne b ...
  - Pour une instance de n personnes (et n tâches), l'arbre des solutions partielles aura donc  $n + 1$  niveaux
    - Le niveau 0 contient uniquement la racine avec aucune assignation de tâches



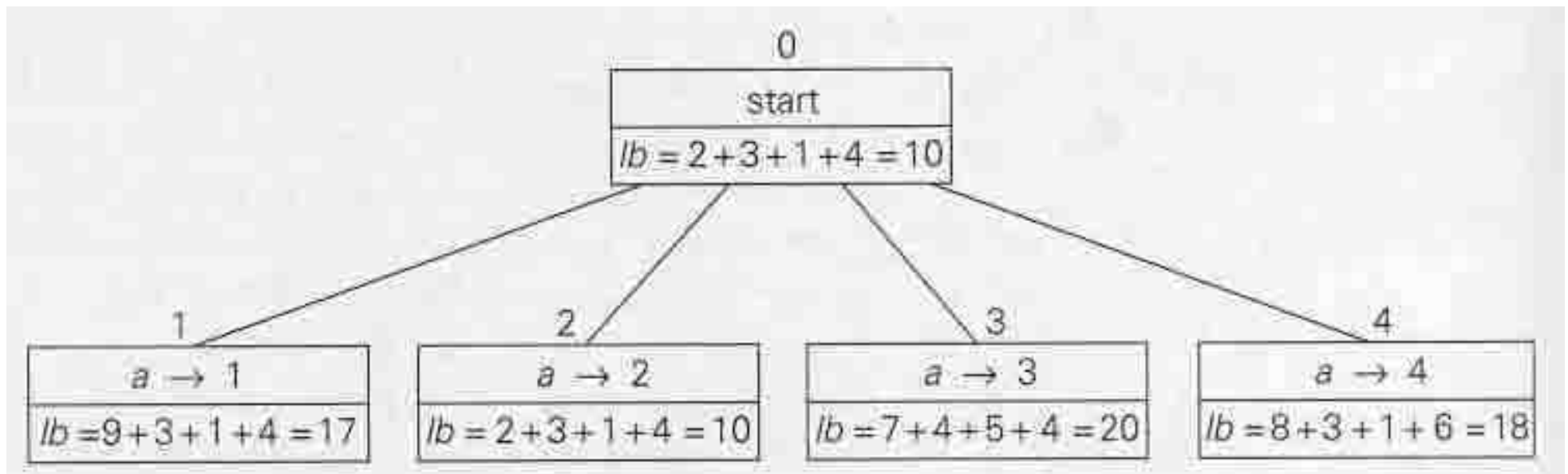
## Assignation de tâches (suite)

---

- C'est un problème de minimisation: la fonction objectif à minimiser est le coût d'une solution (assignation d'une tâche à chaque personne)
  - la valeur de la borne pour chaque solution partielle sera donc une borne inférieure au coût des solutions que nous pouvons obtenir à partir de cette solution partielle
    - En d'autres mots: il est impossible d'obtenir, à partir de ce point, une solution dont le coût est inférieur à la borne
- Comment choisir une borne pour une solution partielle?
- Le coût de chaque solution réalisable doit être supérieur ou égal à la somme des plus petits éléments de chaque rangée
  - Pour notre exemple, cette valeur est  $= 2 + 3 + 1 + 4 = 10$
- Ce sera notre borne initiale: celle de la racine de l'arbre (qui contient aucune assignation de tâches)

## Assignment de tâches (suite)

- Lorsque notre solution partielle assigne 1 tâche à une personne (ex:  $a \rightarrow 3$ ) la borne sera égale au coût de cette assignation + somme des plus petits éléments de chaque rangée excluant la rangée et la colonne correspondant à cette assignation.
  - Pour  $a \rightarrow 3$ , cela donne  $7 + 4 + 5 + 4 = 20$
- Les 2 premiers niveaux de l'arbre sont donc comme suit:



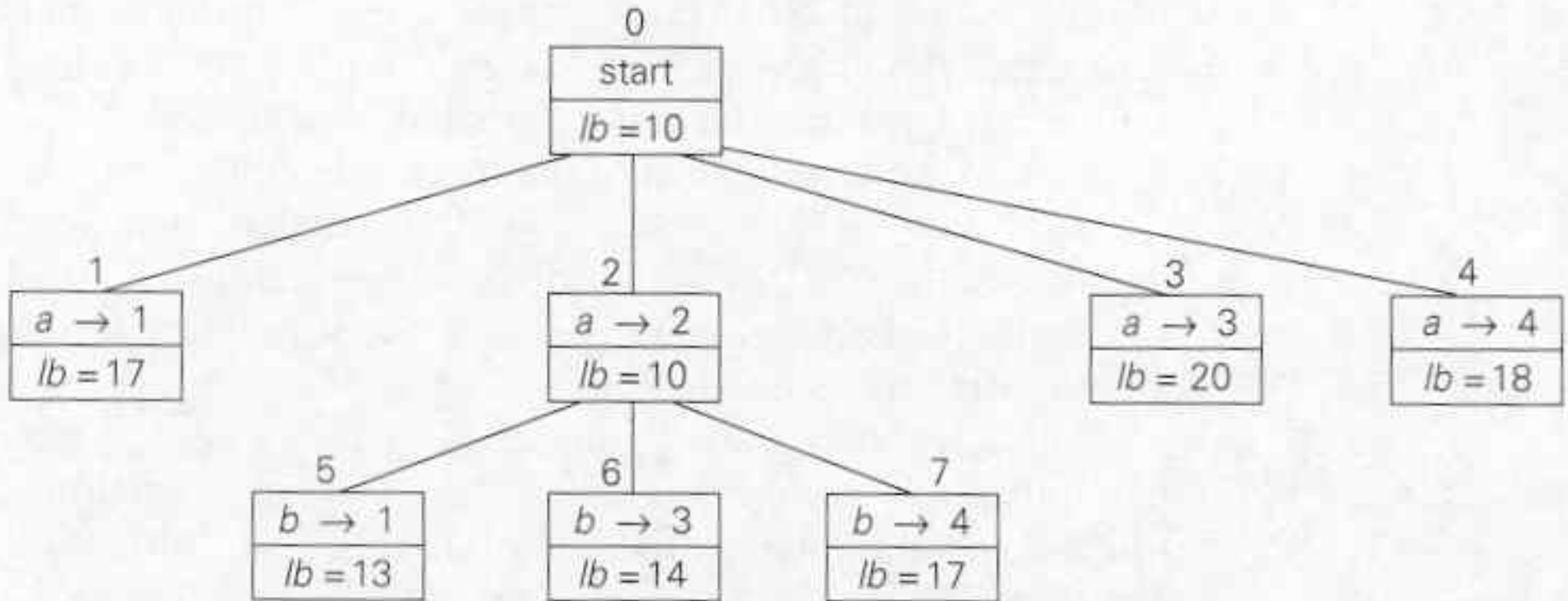


## Assignation de tâches (suite)

---

- Pour chaque nœud (solution partielle) du niveau 1, il faudrait, en principe construire une solution partielle avec une assignation de tâche à la personne b
- Au lieu de faire cela systématiquement pour tous les nœuds du niveau 1, nous considérons d'abord **la solution partielle la plus prometteuse** :
  - Pour un problème de minimisation: c'est la solution partielle qui possède la borne la plus petite
  - Pour un problème de maximisation: c'est la solution partielle qui possède la borne la plus grande
- Et nous explorons les autres solutions partielles générées à partir de ce nœud.
- La solution partielle la plus prometteuse de la figure précédente se trouve au nœud 2, car c'est le nœud actif ayant la plus petite borne.
- En générant les solutions partielles à partir de ce nœud, nous obtenons alors l'arbre de solutions partielles illustré à la page suivante.

## Assignation de tâches (suite)



**FIGURE 11.6** Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

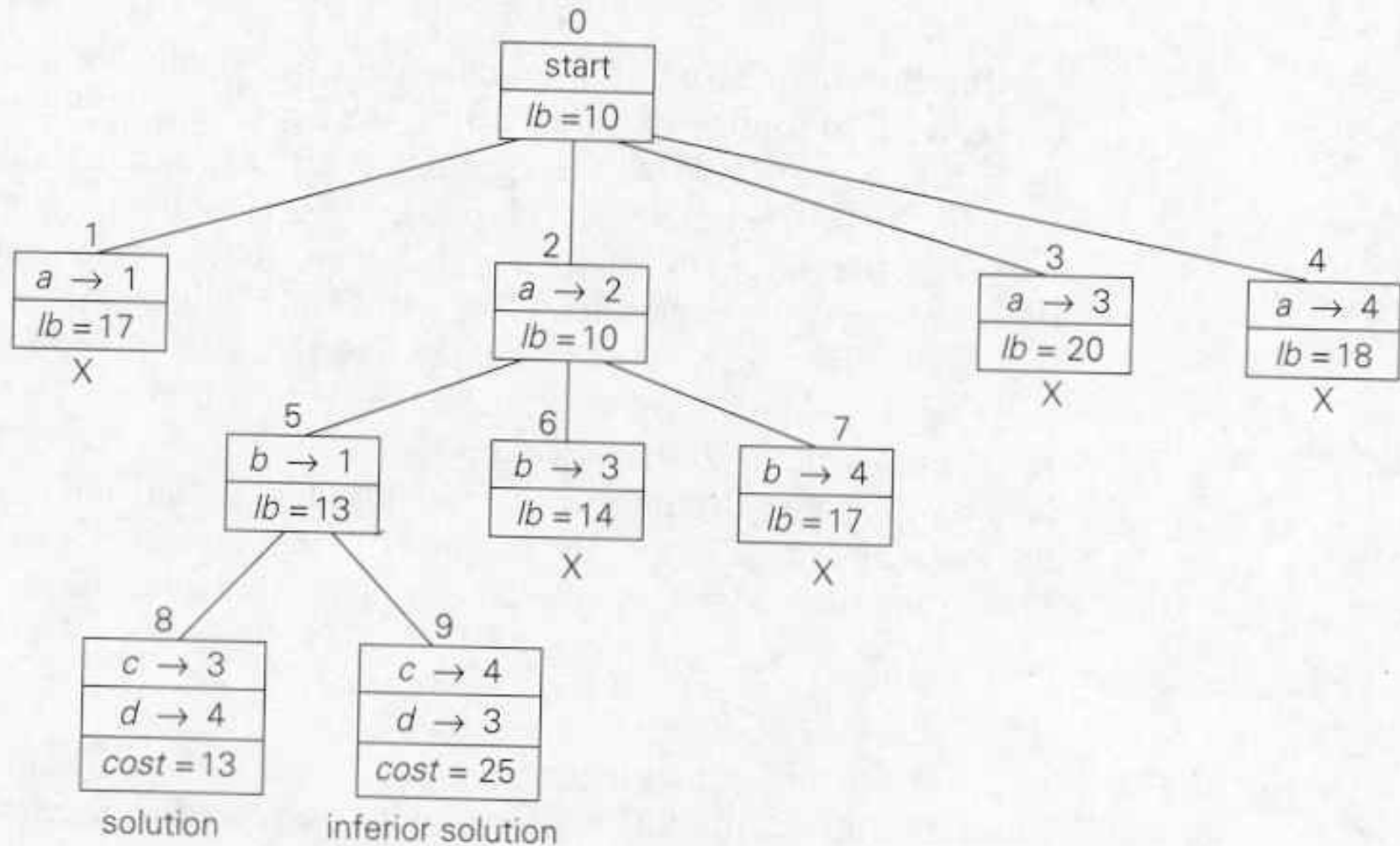


## Assignation de tâches (suite)

---

- À tout moment nous avons un certain nombre de **solutions partielles actives** (c.-à-d., non-terminées)
- La stratégie d'exploration normalement utilisée est celle qui consiste à générer d'autres nœuds à partir de la solution partielle la plus prometteuse (« best-first branch-and-bound »)
  - Il n'est pas assuré que ce soit la meilleure stratégie, car il est possible que la solution optimale soit obtenue à partir d'une solution partielle ayant une moins bonne borne.
- La solution partielle la plus prometteuse de la figure précédente se trouve au nœud 5 (car c'est le nœud actif ayant la plus petite borne)
- Les solutions partielles générées à partir de ce nœud seront des solutions réalisables, car les 4 personnes seront affectées à des tâches. Nous obtenons alors l'arbre de la figure suivante.

## Assigination de tâches (suite)



**FIGURE 11.7** Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm





## Assignation de tâches (suite)

---

- Après avoir obtenu une solution réalisable (ici, c'est en fait deux solutions), nous terminons toutes les solutions partielles dont la borne excède le coût de notre meilleure solution (obtenue jusqu'ici).
  - Ceci est indiqué par un « X » à la figure précédente.
- Nous terminons l'algorithme lorsqu'il ne reste aucune solution partielle active.
  - La meilleure solution réalisable obtenue est alors la solution optimale à notre problème.
- Ce problème d'assignation de tâche illustre bien la méthode du « branch-and-bound », mais ce problème est, en fait, résoluble en temps polynomial ...
- Cela n'est pas le cas pour le problème suivant: le sac à dos. Celui-là est vraiment NP-difficile.



## Le problème du sac à dos

---

- Il s'agit de trouver le sous ensemble de  $n$  objets de valeur maximale dont le poids total n'excède pas la capacité  $W$  du sac à dos.
- Chaque objet  $i$  possède un poids  $w_i$  et une valeur  $v_i$ .
- Il est naturel d'ordonner les objets par ordre décroissant de leur valeur par unité de poids. Nous avons alors:

$$v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$$

- Après ce ré-ordonnement, l'objet 1 est celui nous donnant le plus de valeur par unité de poids et l'objet  $n$  est celui nous donnant le moins de valeur par unité de poids.
- L'arbre des solutions partielles sera l'arbre binaire suivant:
  - Le niveau 0, constituée de la racine, est une solution où aucun objet est choisi (l'ensemble vide)
  - Le niveau 1 possède 2 nœuds: le nœud gauche identifie une solution partielle qui inclue l'objet 1 et le nœud droit identifie une solution partielle qui n'inclue pas l'objet 1



## Le problème du sac à dos (suite)

---

- Chaque nœud gauche du niveau  $i$  indique que l'objet  $i$  est inclut dans la solution partielle et chaque nœud droit indique que l'objet  $i$  est exclut de la solution partielle.
- Ainsi, chaque chemin allant de la racine à un nœud du niveau  $i$  représente un sous-ensemble des  $i$  premiers objets
- Chaque nœud représente alors un chemin et donc un sous ensemble d'objets
- Pour chaque nœud nous maintiendrons:
  - Le poids total du sous ensemble (représenté par ce nœud)
  - La valeur total du sous ensemble (représenté par ce nœud)
  - La borne sur la valeur totale qu'il est possible d'avoir à partir de ce sous-ensemble des  $i$  premiers objets et, possiblement, en incluant d'autres objets parmi  $\{i+1, \dots n\}$ 
    - Puisque c'est un problème de maximisation, chaque borne sera une borne supérieure sur la valeur qu'il est possible d'obtenir à partir de ce nœud
    - Il sera impossible, à partir de ce nœud, d'obtenir une valeur supérieure à celle indiquée par la borne.



## Le problème du sac à dos (suite)

- Pour le calcul des bornes, nous procédons comme suit:
  - Soit  $w$  le poids total des objets présentement sélectionnés (i.e appartenant à l'ensemble des objets représenté par le nœud)
  - Soit  $v$  la valeur totale des objets présentement sélectionnés
  - Soit un nœud situé au niveau  $i$ .
  - Puisque les objets sont énumérés par ordre décroissant de leur valeur par unité de poids,  $v_{i+1}/w_{i+1}$  est alors la valeur maximale par unité de poids que nous pouvons ajouter à la solution représentée par un nœud au niveau  $i$ .
  - Puisque la capacité résiduelle du sac à dos est  $= W - w$ , la valeur maximale que nous pouvons ajouter à la solution représentée par un nœud au niveau  $i$  ne peut pas dépasser  $(W - w) v_{i+1}/w_{i+1}$
  - Ainsi, la valeur d'une solution émergeant d'un nœud au niveau  $i$  ne peut pas excéder la **borne**  $= v + (W - w) v_{i+1}/w_{i+1}$ .
  - C'est ce que nous choisissons pour chaque borne.



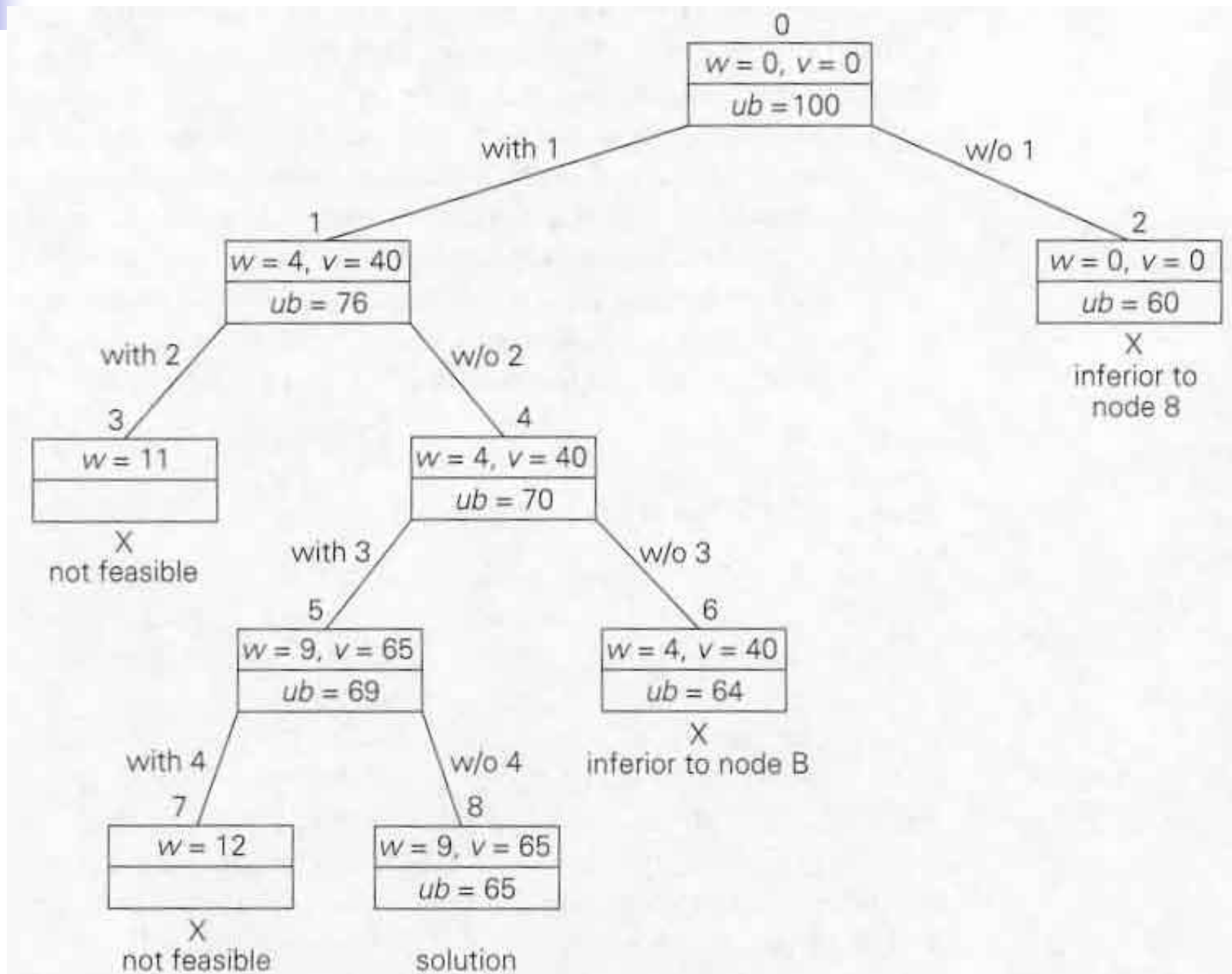
## Exemple

- Dans l'exemple suivant nous avons ordonné les objets par ordre décroissant de valeur par unité de poids.
- La borne de la racine est donnée par  $W v_1/w_1 = 100$ .
- L'arbre des solutions partielles générées par la méthode « branch-and-bound » est illustré à la figure de la page suivante.

item	weight	value	<u>value</u> <u>weight</u>
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity  $W$  is 10.

## Exemple : arbre des solutions



- Notez que pour le problème du sac à dos, chaque nœud (solution partielle) représente une solution réalisable (admissible)
- Pour chaque nœud, nous pouvons alors mettre à jour notre meilleure solution obtenue jusqu'à maintenant et, ainsi, terminer les nœuds actifs ayant une borne  $\leq$  à la valeur de la meilleure solution obtenue jusqu'à maintenant.
- L'efficacité de la méthode « branch-and-bound » dépend grandement des bornes que nous utilisons.
  - Plus les bornes sont serrées, plus grand sera le nombre de branches que nous pourrons terminer et plus l'espace de recherche sera diminué.
  - Cependant, cela ne sera pas rentable si le calcul des bornes prends un temps prohibitif.
- En pratique, il faut choisir le bon compromis entre la qualité des bornes et le temps requis pour les obtenir.



# Algorithmes d'approximations

---

- Un problème d'optimisation consiste à trouver la solution  $s^*$  qui optimise une fonction objectif  $f(s)$ 
  - (Pour chaque solution réalisable  $s$ , nous avons une valeur  $f(s)$  de la fonction objectif)
- Lorsque le problème d'optimisation est NP-difficile, il n'existe pas d'algorithme qui puisse trouver, en pire cas, la solution optimale  $s^*$  en un temps polynomial (si  $P \neq NP$ ).
- Tentons alors de trouver une solution approximative  $s_a$  en temps polynomial (en pire cas) à l'aide d'un algorithme d'approximation.
- De plus, nous désirons obtenir une garantie de la qualité de la solution approximative  $s_a$ .
  - Plus précisément, nous désirons que  $f(s_a)$  ne soit pas trop différent de  $f(s^*)$ .





## Le ratio d'approximation d'un algorithme

- La solution approximative  $s_a$  est obtenue en exécutant un algorithme d'approximation  $A$  sur une instance  $x$  de taille  $|x|$ . Alors  $s_a = A(x)$ .
- La solution approximative est donc fonction de  $x$ . Alors  $s_a = s_a(x)$ .
- La solution optimale  $s^*$  dépend également de  $x$ . Alors  $s^* = s^*(x)$ .
- Le **ratio d'approximation  $R(x)$  de l'algorithme  $A$  sur l'instance  $x$**  est défini par:

$$R(x) \stackrel{\text{def}}{=} \begin{cases} \frac{f(s^*(x))}{f(s_a(x))} & \text{pour problèmes de maximisation} \\ \frac{f(s_a(x))}{f(s^*(x))} & \text{pour problèmes de minimisation} \end{cases}$$

- Ainsi, avec cette définition, nous avons toujours que  $R(x) \geq 1$ .
- Nous avons également que  $R(x) = 1$  ssi  $f(s_a(x)) = f(s^*(x))$ .



## Ratio d'approximation en pire cas

---

- Définissons alors le **ratio d'approximation en pire cas**  $R_w(n)$  de l'**algorithme A** par:

$$R_w(n) \stackrel{\text{def}}{=} \max_{x: |x|=n} R(x)$$

- Un algorithme **A** possède un **ratio d'approximation en pire cas** borné par une **constance c** si et seulement si  $R_w(n) \leq c \ \forall n$  (où **c** est une **constante indépendante de n**).
- Examinons maintenant un algorithme d'approximation pour un problème NP-difficile très connu: le commis voyageur.
  - Rappel: soit un graphe complètement connecté de **n** nœuds, trouvez le plus court cycle hamiltonien.



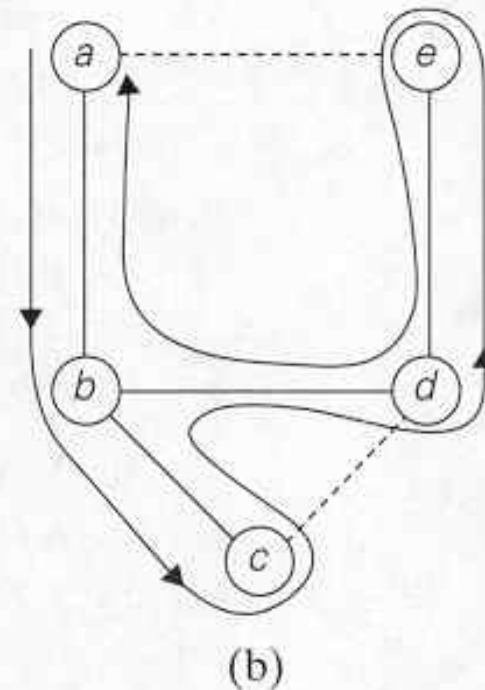
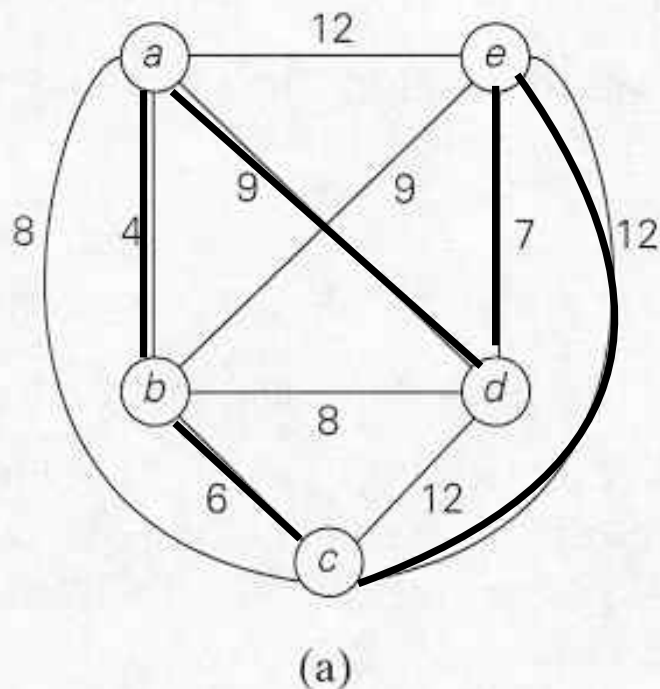
## L'algorithme « twice-around-the-tree »

---

- L'algorithme « twice-around-the-tree » exploite la relation qu'il existe entre un cycle hamiltonien et un arbre de recouvrement minimal.
- Voici cet algorithme:
  - Étape 1: construire l'arbre de recouvrement minimal (à l'aide de l'algorithme de Prim ou celui de Kruskal).
  - Étape 2: choisir un nœud de départ (peu importe lequel) et parcourir le pourtour de l'arbre en mémorisant la séquence de nœuds visités durant ce trajet.
  - Étape 3: parcourir la séquence de nœuds obtenu à l'étape 2 et éliminez, de cette séquence, chaque répétition de nœud (à l'exception du dernier nœud de la séquence).
    - Cette étape produira forcément un cycle hamiltonien
    - Cette étape produit (possiblement) des raccourcis dans la séquence comme le montre l'exemple de la page suivante
- Cet algorithme s'exécute en un temps polynomial.

## Exemple

- Pour le graphe suivant, l'étape 2 donne la séquence a,b,c,b,d,e,d,b,a.
- L'étape 3 nous donne le cycle hamiltonien: a,b,c,d,e,a.
  - Remarque: la séquence a,b,c,d,e est obtenue par un parcours en pré-ordre de l'arbre de recouvrement minimal.



**FIGURE 11.11** Illustration of the twice-around-tree algorithm. (a) Graph. (b) Walk around the minimum spanning tree with the shortcuts.



## Analyse de la performance de cet algorithme

- La longueur du trajet obtenu à l'étape 2 est égale à deux fois la longueur de l'arbre de recouvrement minimal.
  - (La longueur d'un arbre est la somme des distances de ses arêtes)
- Soit  $T$  un arbre de recouvrement minimal et  $f(T)$  la longueur d'un arbre de recouvrement minimal
- La longueur du trajet à l'étape 2 est égale  $2f(T)$ .
- Considérons la solution optimale  $s^*$  de ce problème .i.e., le cycle hamiltonien de longueur minimale =  $f(s^*)$
- Puisque  $s^*$  est un cycle hamiltonien, nous obtiendrons un arbre  $T'$  de recouvrement si nous enlevons une arête de  $s^*$ .
  - La longueur  $f(T')$  doit être supérieure ou égale à celle d'un arbre de recouvrement minimale. Alors  $f(s^*) \geq f(T') \geq f(T)$ .
- La longueur du trajet à l'étape 2 est alors  $\leq 2 f(s^*)$ .
- La longueur du trajet à l'étape 3 est  $\leq$  à celui de l'étape 2 ssi les nœuds que l'on enlève produisent des raccourcis.



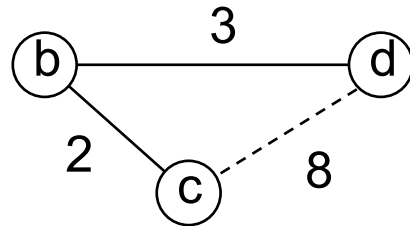
## Analyse de la performance de cet algorithme (suite)

- L'enlèvement d'un nœud dans une séquence produit un raccourci lorsque les distances des arêtes du graphe  $G$  satisfont l'inégalité du triangle:

$$d(i,j) \leq d(i,k) + d(k,j) \quad \forall i,j,k \in G$$

- Un graphe  $G$  complètement connecté est dit **Euclidien** lorsque toutes les distances des arêtes satisfont l'inégalité du triangle.
- Ainsi, pour toute instance  $x$  qui est un graphe Euclidien, la longueur  $f(s_a(x))$  du cycle hamiltonien  $s_a(x)$  produit par l'algorithme « twice-around-the-tree » satisfait  **$f(s_a(x)) \leq 2 f(s^*(x))$** .
- Alors  $R(x) = f(s_a(x))/f(s^*(x)) \leq 2 \quad \forall x$ . **Alors  $R_w(n) \leq 2$ .**
- **Alors, l'algorithme « twice-around-the-tree » possède un ratio d'approximation en pire cas borné par 2 pour les graphes Euclidiens.**

## Analyse de la performance de cet algorithme (suite)



$$8 \geq 2 + 3$$

- Cependant, si les distances ne satisfont pas à l'inégalité du triangle il est possible que l'enlèvement d'un nœud du parcours de l'étape 2 ne produise pas un raccourci mais, au contraire, rallonge le parcours!
- Dans ce cas, nous ne pouvons pas garantir que la longueur du cycle hamiltonien  $f(s_a(x))$ , obtenue à l'étape 3, sera inférieur ou égal à  $2f(T)$ .
- Pour les graphes non Euclidien, nous ne pouvons donc pas conclure que  $f(s_a(x)) \leq 2 f(s^*(x))$ . En fait, il est impossible de borner la longueur du parcours obtenu à l'étape 3 pour les graphes non Euclidiens.
- **Pour les graphes non Euclidien, l'algorithme « twice-around-the-tree » ne possède pas un ratio d'approximation en pire cas qui soit borné par une constante.**



# Non existence d'un algorithme d'approximation

---

- En fait, pour les graphes en général (possiblement non Euclidiens), nous avons le théorème suivant.
- **Théorème:** si  $P \neq NP$ , alors il n'existe pas d'algorithme, à temps polynomial, avec un ratio d'approximation en pire cas borné par une constante pour le problème du commis voyageur.
- **Preuve (par contradiction):**
  - Supposons qu'il existe un algorithme  $A$ , à temps polynomial, tel que  $f(s_a(x)) \leq c f(s^*(x)) \forall x$ . Nous allons démontrer que  $A$  pourrait être utilisé pour résoudre le problème NP-complet du cycle hamiltonien en temps polynomial.
  - Soit  $G$  une instance du problème cycle hamiltonien.
  - Transformons  $G$  en un graphe complet  $G'$ , instance du problème commis voyageur, de la manière suivante:
    - Assignons une distance 1 à chaque arête de  $G$ .
    - Ajoutons une arête, de distance  $cn + 1$ , à chaque paire de nœuds non connectés dans  $G$ . ( $n$  = nombre de nœuds).





## Non existence d'un algorithme d'approximation (suite)

---

- ... preuve (suite) ...
  - Si  $G$  est un graphe hamiltonien, alors  $G'$  possède un cycle hamiltonien de longueur  $n$  et, dans ce cas, l'algorithme  $A$  trouvera un circuit de longueur  $\leq cn$
  - Si  $G$  n'est pas un graphe hamiltonien, le plus petit circuit de  $G'$  aura une longueur  $\geq cn + 1 > cn$ . Car ce circuit doit contenir au moins une arête de longueur  $cn + 1$ .
    - Dans ce cas, l'algorithme  $A$  trouvera forcément un circuit de longueur  $\geq cn + 1$ .
  - Donc  $G$  est un graphe hamiltonien ssi  $A$  trouve un circuit dans  $G'$  de longueur  $\leq cn$ .
  - L'algorithme  $A$  nous informe donc, en temps polynomial, si oui ou non  $G$  possède un cycle hamiltonien.
  - Ceci contredit notre hypothèse de départ que  $P \neq NP$ . **CQFD.**



## Conclusion

---

- Il arrive souvent qu'un problème d'optimisation NP-difficile est tel qu'il n'existe pas (sous l'hypothèse  $P \neq NP$ ) d'algorithme à temps polynomial qui possède un ratio d'approximation borné par une constante comme c'est le cas pour commis voyageur.
- Nous pouvons, par contre, souvent rendre le problème un peu moins général pour qu'il puisse exister un algorithme à temps polynomial qui possède un ratio d'approximation borné par une constante.
  - C'est ce qui s'est passé en imposant au graphe d'être Euclidien
- Cette restriction peut nous satisfaire en pratique.



## Conclusion (suite)

---

- Si nous ne sommes pas satisfait de la borne sur l'approximation, nous pouvons utiliser la valeur  $f(s_a)$  de la solution approximative  $s_a = A(x)$  comme entrée à un algorithme « branch-and-bound » pour tenter de trouver la solution optimale  $s^*$ .
  - En effet, si  $f(s_a)$  est près de  $f(s^*)$ , beaucoup de solutions partielles seront terminées (celles dont la borne implique que nous ne pouvons pas faire mieux que  $f(s_a)$  à partir de ce noeud).
- Une telle combinaison d'utilisation d'un algorithme d'approximation suivi d'un algorithme « branch-and-bound » est une stratégie qui, en pratique, semble justifiée lorsque nous désirons trouver une solution optimale.