Algorithmes avec *backtracking* et algorithmes *branch-and-bound*

Table des matières

1 Stratégies d'exploration des sommets d'un graphe

De façon générale, on s'intéresse ici aux problèmes dont la solution s'exprime comme une séquence de choix à effectuer et où il faut explorer *un espace de toutes les solutions*. Il existe différentes stratégies d'exploration *exhaustive* d'un tel espace. Les deux principales stratégies peuvent être illustrées à l'aide d'algorithmes d'exploration des sommets d'un graphe.

- Exploration *en profondeur* (*depth-first*) :

  Dans cette stratégie, on initie autant d'appels récursifs que possible avant de retourner à l'appelant. La récursion se termine lorsque l'exploration plus avant du graphe devient impossible (par exemple, il n'y a aucun sommet adjacent ou bien tous les sommets adjacents ont déjà été visités). Cette stratégie est présentée dans l'algorithme 1, où l'indicateur booléen `visite(v)` indique si un sommet `v` a été visité ou non.

```
PROCEDURE explorerEnProfondeur( G: Graphe )
PRECONDITION
  G = (V, E)
DEBUT
  POUR v IN V FAIRE
    visite(v) ← FAUX
  FIN
  POUR v IN V TEL QUE !visite(v) FAIRE
    enProfondeur( G, v )
  FIN
FIN

PROCEDURE enProfondeur( G: Graphe, v: Sommet )
PRECONDITION
  G = (V, E)
  !visite(v)
DEBUT
  visite(v) ← VRAI
  POUR w IN V TEL QUE (v, w) IN E & !visite(w) FAIRE
    enProfondeur( G, w )
  FIN
FIN
```

Algorithme 1: Algorithme récursif pour exploration *en profondeur* d'un graphe

Une façon équivalente, mais *non récursive*, d'effectuer un parcours en profondeur est présentée à l'algorithme 2.

- Exploration *en largeur* (*breadth-first*) :

  Dans cette stratégie, on tente de visiter tous les sommets immédiatement adjacents à un sommet (voisins immédiats) avant d'aller plus profondément dans le graphe, i.e., avant de visiter les voisins des voisins, etc. Contrairement à l'exploration en profondeur, ce n'est pas une stratégie naturellement récursive. On va donc utiliser un algorithme non récursif, lequel utilise alors une *file* (FIFO) plutôt qu'une pile (LIFO), tel qu'illustré dans l'algorithme 3.

2 Graphes implicites

De nombreux problèmes peuvent être vus comme des parcours dans un arbre ou un graphe représentant l'espace des solutions. Pour certains problèmes, toutefois, il n'est pas nécessaire,

```
PROCEDURE explorerEnProfondeur( G: Graphe )
PRECONDITION
  G = (V, E)
DEBUT
  POUR v IN V FAIRE
    visite(v) ← FAUX
  FIN
  POUR v IN V TEL QUE !visite(v) FAIRE
    enProfondeur( G, v )
  FIN
FIN

PROCEDURE enProfondeur( G: Graphe, v: Sommet )
PRECONDITION
  G = (V, E)
  !visite(v)
INVARIANT
  Tous les sommets contenus dans p sont déjà visités,
  mais leurs sommets adjacents n'ont pas nécessairement encore été explorés.
DEBUT
  p ← creer Pile()
  visite(v) ← VRAI
  p.empiler( v )
  TANTQUE !p.estVide() FAIRE
    TANTQUE il existe w IN V
                TEL QUE (p.sommet(), w) IN E
                      & !visite(w) FAIRE
      visite(w) ← VRAI
      p.empiler( w )
    FIN
    p.depiler()
  FIN
FIN
```

Algorithme 2: Algorithme non récursif pour exploration *en profondeur* d'un graphe

```
PROCEDURE explorerEnLargeur( G: Graphe )
PRECONDITION
  G = (V, E)
DEBUT
  POUR v IN V FAIRE
    visite(v) ← FAUX
  FIN
  POUR v IN V TEL QUE !visite(v) FAIRE
    enLargeur( G, v )
  FIN
FIN

PROCEDURE enLargeur( G: Graphe, v: Sommet )
PRECONDITION
  G = (V, E)
  !visite(v)
INVARIANT
  Tous les sommets contenus dans f sont déjà visités,
  mais leurs sommets adjacents n'ont pas nécessairement encore été explorés.
DEBUT
  f ← creer File()
  visite(v) ← VRAI
  f.ajouterEnQueue( v )
  TANTQUE !f.estVide() FAIRE
    v ← f.tete()
    f.supprimerTete()
    POUR w IN V TEL QUE (v, w) IN E & !visite(w) FAIRE
      visite(w) ← VRAI
      f.ajouterEnQueue( w )
    FIN
  FIN
FIN
```

Algorithme 3: Algorithme non récursif pour exploration *en largeur* d'un graphe

ou même possible, de construire explicitement le graphe avant de l'explorer. Par exemple, si le graphe associé à l'espace des solutions contient un très grand nombre d'éléments (sommets ou arcs), ou même un nombre *infini* d'états, alors il n'est pas intéressant, ou possible, de construire explicitement le graphe dans son ensemble, et ce d'autant plus que seule une partie pourrait devoir être explorée. Par contre, dans plusieurs problèmes, il peut être possible d'explorer un *graphe implicite*. Dans de tels cas, le graphe n'existe pas *a priori*. Il est plutôt "construit", mais pas nécessairement de façon explicite (c'est-à-dire pas avec des noeuds et des arcs explicites), au fur et à mesure où il est exploré. Ceci évite tout d'abord de construire le graphe en entier, lorsque cela n'est pas nécessaire — en d'autres mots, l'espace des solutions est élaboré et développé *sur demande*. Ceci permet aussi de réduire de façon significative l'espace mémoire utilisé, puisque les parties déjà explorées et qui ne font pas partie de la solution partielle en cours d'exploration n'ont pas besoin d'être conservées.

2.1 Exemple : le problème des n reines

À compléter.

2.2 Exemple : l'affectation de n tâches à n agents

Le problème d'affectation de tâches à des agents est le suivant. On a n agents, qui peuvent effectuer des tâches diverses mais à des coûts qui varient selon l'agent et la tâche. On a n tâches qu'on désire faire effectuer par ces agents. Chaque agent peut faire n'importe quelle tâche, mais il ne peut en faire qu'une seule. Le coût pour faire effectuer la tâche j par l'agent i est de c_{ij}. On cherche à trouver une affectation des n tâches aux n agents qui minimisera le coût total d'exécution de ces n tâches.

Par exemple, supposons qu'on ait trois (3) agents a_1, a_2 et a_3 et trois (3) tâches t_1, t_2 et t_3 avec les coûts suivants pour la réalisation des tâches :

	t_1	t_2	t_3
a_1	5	8	4
a_2	3	7	2
a_3	4	9	5

Notons par $i \Rightarrow j$ l'affectation de la tâche t_j à l'agent a_i — c'est-à-dire que l'agent i effectue la tâche j. Voici quelques affectations possibles et leurs coûts associés :

- $1 \Rightarrow 1, 2 \Rightarrow 2, 3 \Rightarrow 3$: coût $= 5 + 7 + 5 = 17$.
- $1 \Rightarrow 2, 2 \Rightarrow 3, 3 \Rightarrow 1$: coût $= 8 + 2 + 4 = 14$.
- ...

L'espace de toutes les affectations de tâches, acceptables ou non, peut être représenté par un arbre des affectations (donc un graphe), tel qu'illustré à la Figure 1. Dans cette figure, chaque boîte représente une affectation possible de tâches à des agents, affectation partielle ou complète, correcte ou non. Un tel arbre peut être construit (en profondeur ou en largeur) en examinant tout d'abord les tâches pouvant être affectées à l'agent a_1, puis en faisant de même pour l'agent a_2, et finalement pour a_3.

Cet arbre de toutes les affectations possibles est tel que certaines configurations *ne représentent pas des solutions acceptables*, dans la mesure où une même tâche peut être affectée à plusieurs agents différents. On verra ultérieurement comment éviter d'affecter une tâche déjà affectée, donc comment *élaguer* l'arbre des configurations.

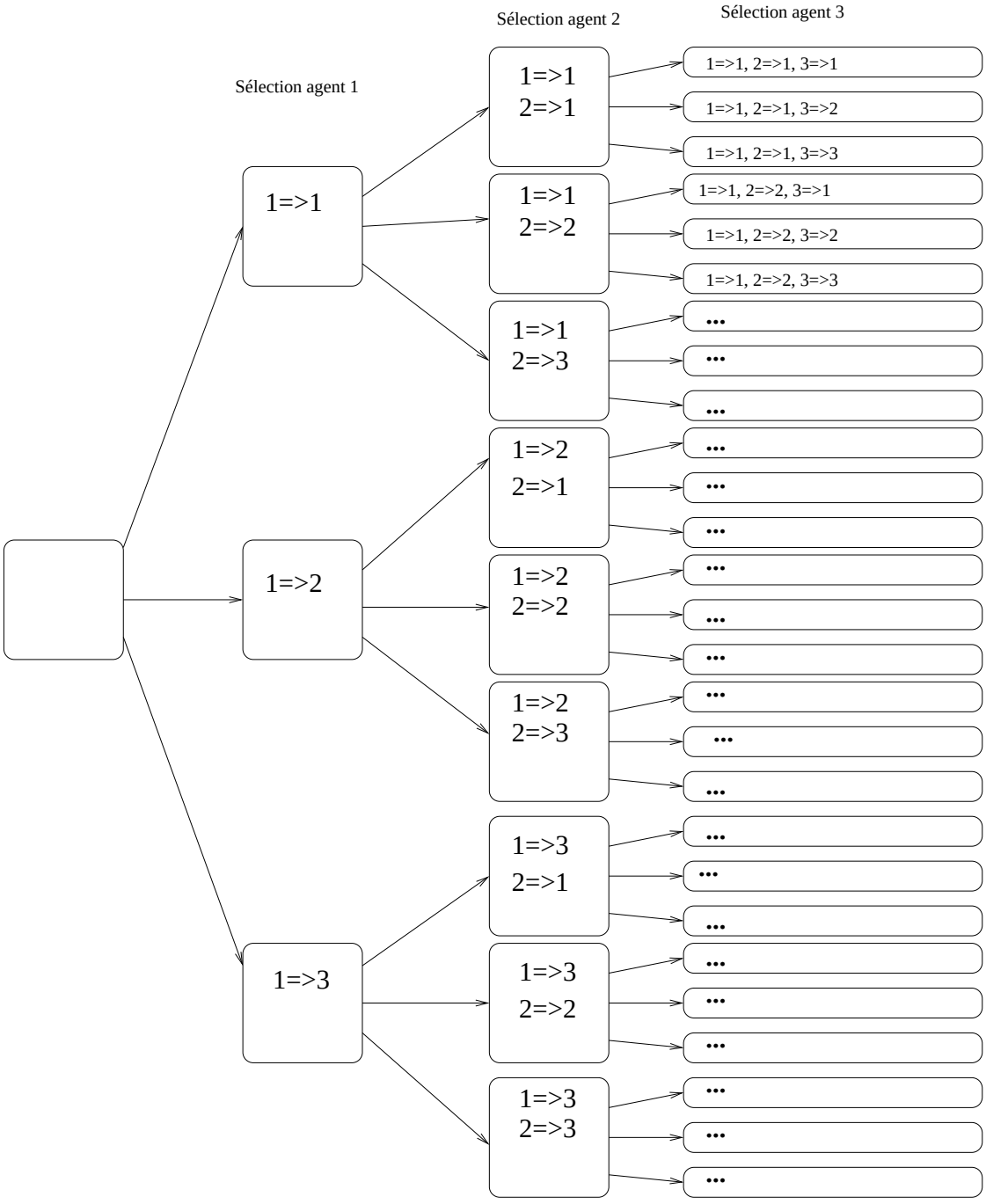

Figure 1: L'espace complet des affectations de tâches (correctes ou non) pour l'exemple avec trois agents et trois tâches

3 Algorithmes avec marche arrière (*backtracking*)

Un algorithme d'exploration avec marche arrière est d'abord et avant tout un algorithme d'exploration en profondeur. Un tel algorithme peut être utilisé soit pour déterminer si une solution existe — auquel cas on termine l'exécution dès qu'une solution est trouvée —, soit pour identifier toutes les solutions possibles.

Une autre caractéristique importante de la technique d'exploration avec marche arrière est qu'une notion de noeud ou de choix *non-prometteur* est utilisée pour éviter de parcourir certains sous-arbres : un noeud est *non-prometteur* si on est certain qu'aucun de ses descendants ne pourra conduire à une solution acceptable (faisable), auquel cas le sous-arbre associé n'a donc pas besoin d'être parcouru. C'est donc cette notion d'élagage (de "non exploration") des sous-arbres associés aux noeuds prometteurs qui distingue principalement les algorithmes de *backtracking* des algorithmes d'exploration *exhaustive* de l'espace des solutions.

La forme générale d'un algorithme d'exploration avec marche arrière est présentée à l'algorithme 4. Soulignons que la façon exacte dont est représentée une solution (séquence, ensemble, etc.), la façon d'identifier les noeuds à explorer, ce qui caractérise un choix prometteur (i.e., qui n'est pas non-prometteur), etc., varie d'un problème à un autre.

```
PROCEDURE explorerAvecMarcheArriere()
DEBUT
  S ← ()                  # Solution initiale vide
  explorerMARec( S )      # Appel de la procedure recursive d'exploration
FIN

PROCEDURE explorerMARec( (v1, ..., vk): SolutionPartielle )
PRECONDITION
  (v1, ..., vk) est une solution partielle prometteuse.
DEBUT
  SI (v1, ..., vk) est une solution complète ALORS
    Traiter la solution (v1, ..., vk)
    TERMINER l'execution SI on doit trouver une unique solution
  SINON
    POUR tout v TEL QUE (v1, ..., vk, v) est prometteur FAIRE
      explorerMARec( (v1, ..., vk, v) )
    FIN
  FIN
FIN
```

Algorithme 4: Forme générale d'un algorithme d'exploration avec marche arrière

3.1 Exemple : l'affectation de n tâches à n agents (suite)

À la Figure 1, nous avons vu que l'espace de toutes les configurations pour le problème d'affectation de n tâches à n agents pouvait être représenté par un arbre. Or, nous avons aussi vu que cet arbre pouvait contenir des configurations qui ne représentent pas des solutions possibles ou acceptables, c'est-à-dire que cet arbre contenait des solutions où une même tâche était affectée à plus d'un agent.

Un algorithme avec marche arrière pour ce problème, plutôt que de simplement énumérer ou explorer toutes les combinaisons possibles d'affectation de tâches, n'examinera que celles conduisant à des configurations *prometteuses*, c'est-à-dire, pour lesquelles chaque tâche n'est associée qu'à un seul agent. La Figure 2 représente l'arbre des configurations pour l'exemple

vu précédemment, arbre où certaines branches sont élaguées (le lien n'a pas été indiqué) puisqu'elles correspondent à des solutions non prometteuses. Les feuilles, qui correspondent à des solutions complètes, ont été annotées avec le coût de la solution ainsi obtenue.

4 Algorithmes *branch-and-bound*

Dans les problèmes d'optimisation, on associe à chaque solution une certaine valeur (ou un certain coût) et on doit trouver la (ou les) solution(s) ayant la meilleure valeur (le meilleur coût). Pour de tels problèmes, la stratégie d'exploration avec marche arrière peut être améliorée en évitant d'explorer les solutions partielles pour lesquelles on est certain que la valeur (ou le coût) *ne sera pas optimale*. D'une certaine façon, il s'agit donc de raffiner la notion de "*solution prometteuse*" pour tenir compte de la valeur (ou du coût) de la solution, et non pas uniquement de sa *faisabilité*, comme c'est le cas dans un algorithme avec marche arrière typique. Évidemment, au fur et à mesure où on explore l'espace des solutions, on doit mettre à jour la valeur (ou le coût) de la meilleure solution trouvée jusqu'à présent.

La forme générale d'un algorithme d'exploration *branch-and-bound* est présentée à l'algorithme 5. On suppose ici que la valeur d'une solution est un nombre entier non négatif et qu'on cherche à *maximiser* cette valeur. (Si l'on désirait plutôt minimiser le coût, on utiliserait alors la valeur `high(int)` comme valeur initiale pour `valOptimale` et on utiliserait "`<`" comme opérateur de comparaison). L'exploration se fait ici en profondeur — notons toutefois que ce n'est pas une caractéristique de tous les algorithmes *branch-and-bound*.

Comme pour l'algorithme générique d'exploration avec marche arrière, la façon exacte dont est représentée une solution (séquence, ensemble, etc.), la façon d'identifier les noeuds à explorer, ce qui caractérise un choix prometteur, etc., varie d'un problème à un autre. La façon de déterminer si une solution partielle, en plus d'être prometteuse (c'est-à-dire peut conduire à une solution acceptable), a aussi "une chance" d'avoir une valeur qui soit meilleure que la meilleure solution trouvée jusqu'à présent (fonction `valeurEstimee`) dépend évidemment du problème traité.

Une autre différence entre les algorithmes avec marche arrière et les algorithmes *branch-and-bound* est la suivante : alors que les algorithmes avec marche arrière utilisent toujours des fouilles *en profondeur*, les algorithmes *branch-and-bound* peuvent utiliser diverses stratégies de fouille, par exemple, en profondeur, en largeur, *best-first*, etc. — on verra des exemples dans le chapitre sur le voyageur de commerce.

4.1 Choix de la fonction borne

La borne associée à un noeud N représente *la valeur de la meilleure solution possible pouvant être obtenue à partir du noeud* N (i.e., en complétant la solution partielle représentée par le noeud N).

Pour qu'un algorithme *branch-and-bound* soit efficace, il est évidemment important que la borne associée à un noeud puisse être calculée en temps "raisonable" — préférablement en temps polynomial d'ordre inférieur.

Une façon souvent utilisée pour obtenir une borne consiste à relâcher certaines des contraintes du problème de façon à obtenir un problème simplifié pour lequel une solution optimale peut être obtenue plus rapidement. La technique utilisée pour résoudre le problème simplifié peut alors être utilisée pour calculer une borne au problème initial.

Par exemple (cf. exercice), si on tente de résoudre le problème du sac à dos 0–1 à l'aide d'un algorithme *branch-and-bound*, on pourrait utiliser comme estimation d'une solution partielle le fait de sélectionner comme prochain item celui ayant le meilleur rapport bénéfice par unité de poids.

Notons que pour que cette stratégie consistant à utiliser un problème simplifié pour produire une borne fonctionne correctement, il faut qu'une solution S au problème initial P soit

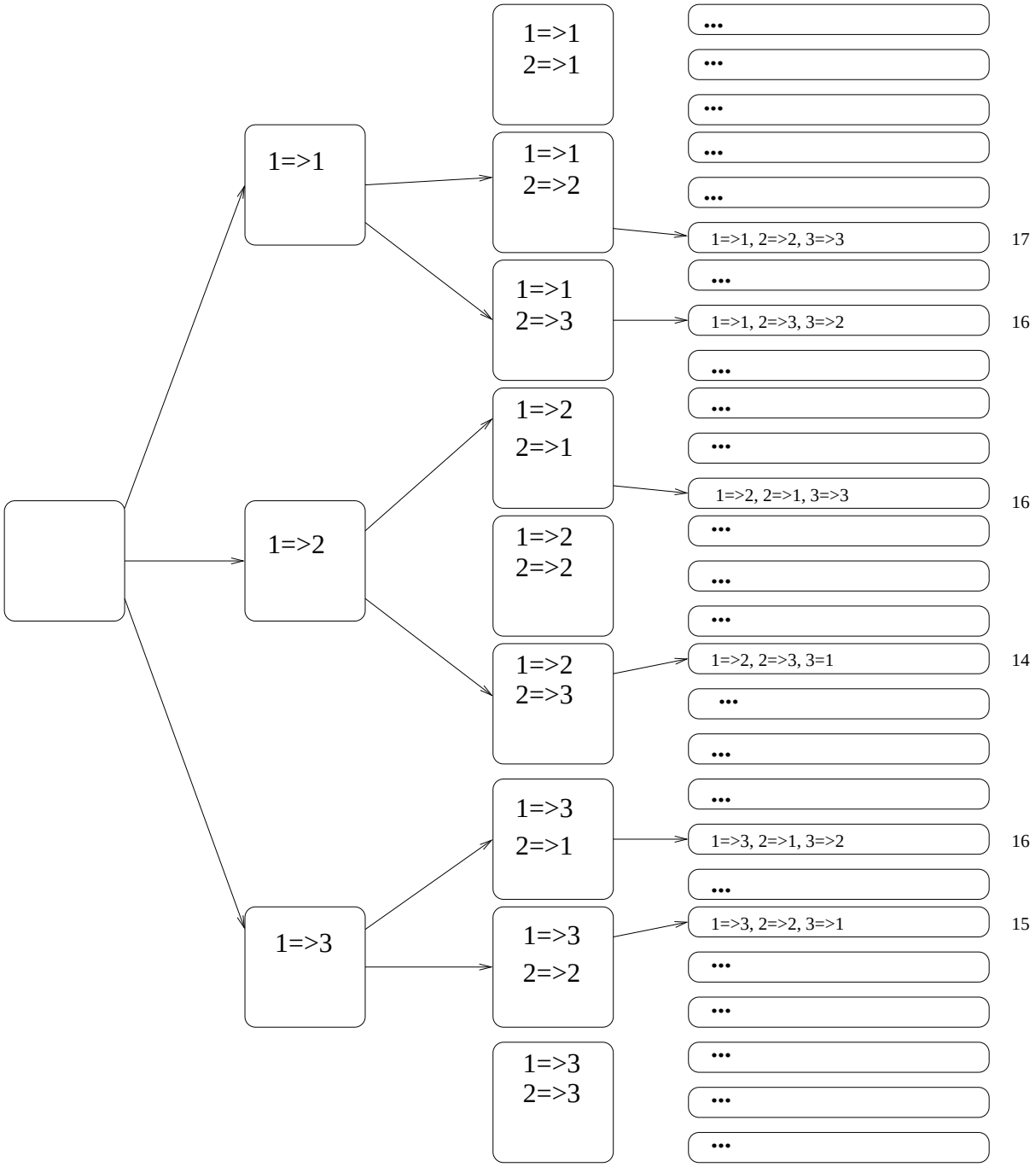

Figure 2: L'espace des affectations de tâches acceptables pour l'exemple avec trois agents et trois tâches

```
VAR
  SOptimale: Solution
  valOptimale: int

PROCEDURE explorerBranchAndBound()
POSTCONDITION
  SOptimale contient la solution optimale.
  valOptimale contient la valeur associée à cette solution.
DEBUT
  valOptimale ← 0
  S ← ()                  # Solution initiale vide
  explorerBABRec( S )     # Appel de la procedure recursive d'exploration
FIN

PROCEDURE explorerBABRec( (v1, ..., vk): SolutionPartielle )
PRECONDITION
  (v1, ..., vk) est une solution partielle prometteuse.
DEBUT
  SI (v1, ..., vk) est une solution complète ALORS
    val ← valeur( (v1, ..., vk) )
    SI val > valOptimale ALORS
      SOptimale ← (v1, ..., vk)
      valOptimale ← val
    FIN
  SINON
    POUR tout v TEL QUE (v1, ..., vk, v) est prometteur
                      & valeurEstimee( (v1, ..., vk, v) ) > valOptimale FAIRE
      explorerBABRec( (v1, ..., vk, v) )
    FIN
  FIN
FIN
```

Algorithme 5: Forme générale d'un algorithme *branch-and-bound* (ici, avec fouille en profondeur)

aussi une solution au problème simplifié P' et que la solution S ne soit pas meilleure que S', la solution optimale de P'.

4.2 Exemple : l'affectation de n tâches à n agents (suite)

Une approche *branch-and-bound* peut être utilisée pour le problème de l'affectation de n tâches à n agents. Pour ce faire, il suffit d'identifier une *approximation* du coût minimum de l'affectation de tâches pouvant être obtenue à partir d'une configuration donnée (une affectation partielle). Si cette approximation semble "promettre" un minimum meilleur que le minimum courant, alors la branche associée de l'arbre doit être explorée ; dans le cas contraire, par contre, la branche peut simplement être ignorée (élaguée).

Pour notre problème, une approximation du minimum possible peut être obtenue en prenant, parmi les affectations non encore effectuées, la somme des coûts minimums possibles. Plus précisément, supposons que des affectations ont été effectuées pour les k premiers agents $a_1, \ldots, a_k$ ($k < n$), affectations telles que l'agent i effectue la tâche j_i. Soit alors J l'ensemble des tâches déjà affectées ($J = \{j_1, j_2, \ldots, j_k\}$). Pour un agent a_i pour lequel l'affectation de tâche n'a pas encore été effectuée ($i = k+1, \ldots, n$), le coût minimum possible sera simplement le minimum suivant, coût pouvant être obtenu en examinant les éléments de la i ligne de la matrice des coûts :

$$CoutMin_i^J = \min_{j \notin J} c_{ij}$$

Une borne pour le coût minimum total pouvant être obtenu à partir d'une telle affectation partielle est alors la suivante :

$$borne = \sum_{i=1}^{k} c_{ij_i} + \sum_{i=k+1}^{n} CoutMin_i^J$$

Pour démarrer l'exploration des solutions, il faut aussi spécifier une borne initiale. Come il s'agit ici d'un problème de minimisation, la borne initiale utilisée sera $+\infty$.

La Figure 3 illustre alors l'arbre, élagué, obtenu en utilisant une telle approximation sur notre exemple avec trois tâches et trois agents. La valeur dans un cercle, près d'un noeud, indique la borne globale courante (`+INF` dénote $+\infty$).

Notons que la borne utilisée peut être interpréteé comme la valeur associée à une solution qui permet d'affecter une même tâche (parmi celles non encore sélectionnées) à plusieurs agents : pour chaque agent non encore traité, on prend simplement le minimum parmi les diverses tâches non sélectionnées, comme si une même tâche pouvait être affectée à plus d'un agent.

5 Conclusion : *backtracking* vs. *branch-and-bound*

En conclusion, on peut dire qu'un algorithme avec marche arrière peut être vu comme un cas spécial d'algorithme *branch-and-bound*. Plus précisément, si on a un algorithme d'exploration avec marche arrière qui examine toutes les solutions à un problème mais n'en conserve qu'une seule (la première trouvée), alors cet algorithme aurait un comportement essentiellement équivalent à celui d'un algorithme *branch-and-bound* où l'exploration se ferait en profondeur et où la valeur de chacune des solutions serait la même pour n'importe quelle solution — c'est alors la première solution trouvée qui serait prise en note et retournée comme résultat.

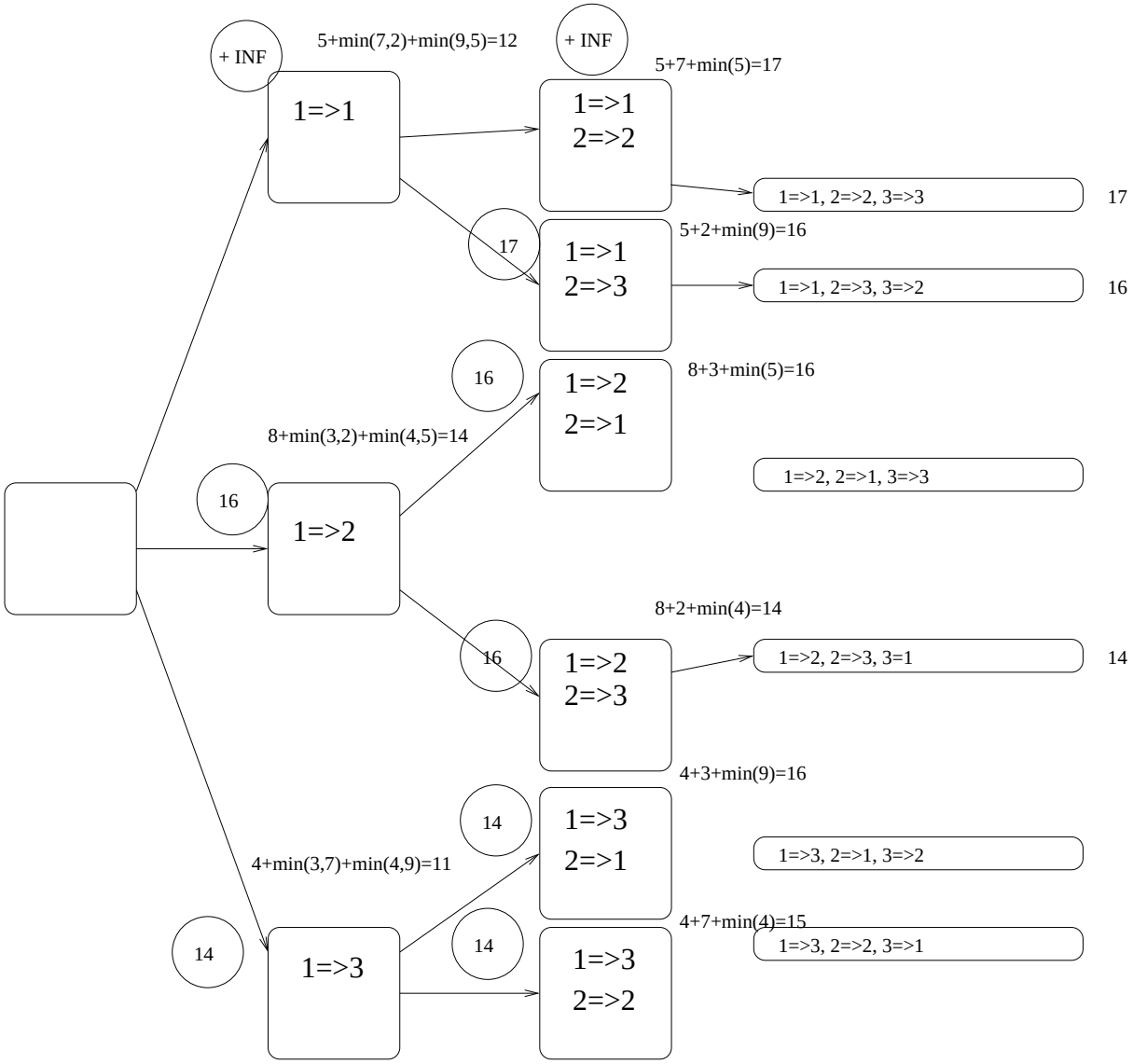

Figure 3: L'espace des affectations de tâches acceptables et élagués par approximation *branch-and-bound* pour l'exemple avec trois agents et trois tâches

Références

[BB96] G. Brassard and P. Bratley. *Fundamentals of Algorithmics.* Prentice-Hall, 1996. [QA76.6B73].

[NN04] R. Neapolitan and K. Naimipour. *Foundations of Algorithms Using C++ Pseudocode (Third edition).* Jones and Bartlett Publishers, 2004.

Cette partie des notes de cours est basée principalement sur les références suivantes :

- [NN04, Chapitres 5 et 6]
- [BB96, Chapitre 9]
- Transparents produits par Cédric Chauve, à l'automne 2004 et à l'hiver 2005, dans le cadre du cours INF7440.