CHAPITRE 13

Méthodes faibles

Jean-Marc Alliot

13.1 Introduction

Pour pouvoir résoudre un problème, quel qu'il soit, il faut avant tout le poser correctement, l'analyser, et définir les moyens pour le résoudre. Le but de ce chapitre est de développer les méthodes permettant de réaliser systématiquement ces trois opérations.

Il portera donc sur cinq points :
- la définition de l'espace d'états du problème ;
- la définition d'un système de production permettant de se déplacer à l'intérieur de l'espace d'états ;
- la définition des structures de représentation ;
- la définition de stratégies de contrôle efficaces ;
- l'analyse des caractéristiques du problème permettant de choisir une représentation et une stratégie de contrôle adaptées.

Les méthodes pour la résolution (et la représentation) que nous décrivons dans ce chapitre sont appelées *méthodes faibles*. Elles sont basées pour la plupart sur des techniques de parcours de graphe et peuvent être adaptées à de nombreux domaines. Leur efficacité est généralement mauvaise pour un problème particulier, car elles sont incapables de maîtriser l'explosion combinatoire très importante que ce type de représentation induit (cette analyse a conduit à la réalisation de *« systèmes experts »* dont nous aurons l'occasion de reparler). Comme a pu le montrer la théorie de la complexité (cf. chapitre 11), certains problèmes semblent combinatoires de façon inhérente, et ce n'est donc pas surprenant.

13.2 Espaces d'états

Nous posons dans ce paragraphe l'ensemble des postulats de base de la résolution de problèmes.

13.2.1 Principes généraux

Définition 13.1 – Variables d'état – *Tout problème est relatif à un certain ensemble d'objets que l'on peut décrire sous forme de* variables d'état *du problème.*

Définition 13.2 – État d'un problème – *Un état du problème est l'ensemble des valeurs que prennent ses variables d'état à un instant donné.*

Définition 13.3 – Espace d'états – *L'espace d'états d'un problème est l'ensemble des états possibles pour le problème considéré.*

On parle plus volontiers d'espace d'états que d'ensemble d'états. Nous verrons tout à l'heure que ce choix terminologique est justifié : *ensemble* évoque une collection d'objets sans rapport les uns avec les autres, alors qu'*espace* implique déjà des notions de relation entre les objets (déplacements dans l'espace). Notons tout de même une légère ambiguïté : qu'est-ce qu'un état *possible* ? En fait, nous verrons que les états possibles sont les états qui peuvent être générés à partir des états initiaux en suivant les règles de production du système. Mais même ainsi, il n'est pas toujours simple de savoir si un état donné est possible ou non (aux échecs cela s'appelle *réaliser une analyse rétrograde*) et cela peut même se révéler impossible[1]. D'autre part, il est des problèmes où il est nécessaire d'étendre cette notion d'états possibles pour rendre le problème plus facile à résoudre. Nous y reviendrons dans les exemples développés dans ce chapitre.

Les définitions que nous venons de présenter marquent bien les limites des mécanismes que nous allons décrire dans cette partie. Les problèmes que nous allons examiner doivent être parfaitement connus et déterminés, puisque nous devons pouvoir ramener leur description à un ensemble fini de variables d'état. D'autre part, il doit être possible de discrétiser le problème d'une façon ou d'une autre, afin d'avoir un espace d'états au plus dénombrable, pour qu'il soit descriptible par une machine de Turing.

13.2.2 Exemples

Nous allons présenter quelques exemples de problèmes classiques que l'on sait facilement décrire sous forme d'espace d'états :

Les missionnaires et les cannibales : Ce problème, très classique, est connu sous bien d'autres formes (problèmes des lions et des dompteurs, ou du lama, de la chèvre, du lion, et du berger). Il remonte au Moyen Âge. Voici un des énoncés :

> « Trois cannibales et trois missionnaires doivent traverser un fleuve en empruntant une barque qui ne peut contenir que deux personnes. Or, si sur une des deux rives, à un moment quelconque, on trouve plus (strictement) de cannibales que de missionnaires, les premiers nommés font chauffer la marmite. Les missionnaires doivent donc trouver une méthode pour qu'à aucun moment ils ne soient en infériorité numérique. »

Un état peut être représenté par un triplet (X, Y, P) avec $0 \leq X \leq 3, 0 \leq Y \leq 3$ et $P = D$ ou $P = G$. X représente le nombre de missionnaires sur la rive gauche, Y le nombre de cannibales sur cette même rive et P représente la position de la barque (D pour rive droite et G pour rive gauche). Il est clair que nous avons ainsi défini l'espace d'état de notre problème. L'état initial est $(3, 3, G)$ et on cherche à atteindre l'état final $(0, 0, D)$.

---

1 On peut utiliser le problème de Post pour le montrer.

La démonstration de théorèmes : Considérons le problème de construction d'un système réalisant la démonstration automatique de théorème en arithmétique formelle. L'espace d'états est évidemment l'ensemble des formules bien formées.

Le jeu d'échecs : Le jeu d'échecs peut aisément être ramené à l'état de chacune des 64 cases à un instant donné. Le nombre de variables d'états (64) est fini, les valeurs que peuvent prendre ces variables également : une case peut contenir un pion, une tour, un cavalier, un fou, une dame ou un roi de l'une ou l'autre couleur, ou peut être vide, soit 13 valeurs différentes possibles. D'autre part le problème est parfaitement discret : on passe d'un état à un autre par une modification discrète d'une ou plusieurs variables.

Quels sont les états possibles du système ? Nous pourrions prendre en première approximation l'ensemble des affectations possibles de chacune des 13 valeurs dans les 64 variables. Mais il est clair que tous ces états ne sont pas possibles. En particulier, on ne peut jamais trouver deux rois de la même couleur simultanément sur l'échiquier, ni de pion blanc sur la première rangée, ni de pion noir sur la huitième rangée. Nous pouvons donc exclure ces états de notre espace. Mais il est d'autres états qui sont impossibles, quoi que plus difficiles à déterminer. Faut-il en tenir compte ? Nous nous trouvons là face au dilemme que nous soulignions tout à l'heure : déterminer les états possibles d'un système de façon précise peut se révéler extrêmement difficile. On se contente souvent de considérer comme espace d'états un sur-ensemble de l'espace des états possibles, plus facile à définir.

Le problème du voyageur de commerce : Nous avons déjà évoqué ce problème dans le chapitre 11 (théorie de la complexité). Il s'agit de trouver le plus court circuit passant par n villes sans jamais passer deux fois dans la même ville. L'espace d'états du problème est alors un ensemble composé de tous les singletons, doublets, triplets, n-uplets ne contenant qu'une seule fois une ville. Ainsi, pour un problème à quatre villes, le doublet $(1, 3)$ signifierait que le voyageur a commencé son circuit par la ville 1, puis qu'il est allé dans la ville 3 et qu'il lui reste deux villes à parcourir. Seuls les quadruplets sont susceptibles d'être des solutions du problème. L'espace d'états que nous avons défini est plus vaste que l'espace des solutions possibles du problème.

Le jeu de poker sans écart : L'espace[2] d'états est constitué de l'ensemble des mains possibles pour le joueur représenté par le programme, augmenté de la séquence de toutes les enchères possibles. Remarquons que nous ne considérons pas l'ensemble des mains de l'adversaire : c'est une caractéristique des jeux à cartes cachées sur laquelle nous reviendrons.

Pour mieux comprendre l'intérêt d'espace d'états plus vaste que l'espace contenant les solutions possibles, il nous faut aborder la notion de *marche vers la solution* et de *système de production*.

---

2 Nous nous contenterons du jeu de Poker sans écart, qui est plus simple à présenter que le jeu de Poker traditionnel, mais l'ensemble des résultats que nous énoncerons peuvent aisément se généraliser.

13.3 Systèmes de production

13.3.1 Raisonnement en chaînage avant

Résoudre un problème par un raisonnement déductif, consiste à identifier l'*état initial*, l'*état final* et un mécanisme permettant de décrire les règles régissant le passage d'un état à un autre : le *système de production*. Les notions d'état initial et d'état final sont claires, en revanche il est bon de préciser celle de système de production.

Définition 13.4 – Système de production – *Un système de production est constitué d'un ensemble de règles, les règles de production, qui permettent à partir d'un état du problème donné, de générer l'ensemble des états que l'on peut légalement atteindre depuis cet état particulier.*

Définition 13.5 – Solution déductive d'un problème – *Obtenir une solution déductive d'un problème consiste à passer de l'état initial à un état final*[3] *en suivant une chaîne d'états intermédiaires, chaque état intermédiaire étant généré en accord avec le système de production.*

Ce type de système de production est dit à chaînage avant.

Cette notion de règle de production n'est pas si éloignée des systèmes de production tels que nous les avons définis en théorie des langages. Comme nous allons le voir, la définition des règles de production n'est pas toujours chose facile. On peut généralement définir plusieurs systèmes de production possibles, souvent très différents.

Nous avons d'autre part affirmé que les notions d'état initial et d'état final étaient claires. Il n'est cependant pas toujours évident de reconnaître un état comme étant final. Ainsi, aux échecs, vérifier que l'adversaire est mat n'est pas une chose triviale à formaliser.

13.3.2 Raisonnement en chaînage arrière

Ce type de raisonnement est aussi appelé raisonnement inductif. La seule différence entre raisonnement inductif et raisonnement déductif se trouve dans la définition de la solution :

Définition 13.6 – Solution inductive d'un problème – *Obtenir une solution inductive d'un problème consiste à passer de l'état final à l'état initial en suivant une chaîne d'états intermédiaires, chaque état intermédiaire étant généré en accord avec le système de production.*

Ce type de système de production est dit à chaînage arrière.

Bien entendu, un système de production adapté à un raisonnement inductif est généralement différent d'un système de production adapté à un raisonnement déductif.

Il est souvent possible de construire, pour un problème donné, un système de production déductif et un système de production inductif. Comment choisir ? Il faut considérer le facteur de branchement de chacun des systèmes. Ainsi, pour la démonstration de théorème, un système inductif a dans beaucoup de cas un facteur de branchement inférieur à un système déductif, dont la combinatoire est explosive.

On peut également, dans certains cas, mélanger les deux stratégies, et réaliser une partie de la recherche par chaînage avant et une autre partie par chaînage arrière. C'est le

3 Il existe souvent plusieurs états finaux.

cas en démonstration de théorème, où l'on peut par exemple commencer par « décomposer » en sous-buts le théorème à démontrer (chaînage arrière), puis tenter de démontrer chacun des sous-buts à partir des axiomes (chaînage avant). Ce type de mécanisme est appelé *chaînage mixte*.

13.3.3 Monotonie et commutativité

Les systèmes de production peuvent être classés en quatre catégories à l'aide des deux critères suivants :

Définition 13.7 – Système monotonique – *Un système de production est dit monotonique*[4] *si l'application d'une règle à l'instant t laisse toutes les autres règles qui étaient applicables à t applicables à* $t' > t$.

Un exemple de système monotonique est la démonstration de théorème en raisonnement déductif. En effet, l'application d'une règle se contente de rajouter une formule dans la base de théorèmes et laisse toutes les règles qui étaient applicables encore applicables.

Définition 13.8 – Système partiellement commutatif – *Un système de production est dit partiellement commutatif s'il vérifie la condition suivante : soit un état* X *quelconque et une séquence de règles de production* $(s_1, s_2, \ldots, s_n)$ *telle que*

$$X \overset{s_1}{\rightsquigarrow} X_1 \overset{s_2}{\rightsquigarrow} X_2 \ldots \overset{s_{n-1}}{\rightsquigarrow} X_{n-1} \overset{s_n}{\rightsquigarrow} Y$$

Alors pour toute permutation σ *la séquence* $(s_{\sigma(1)}, s_{\sigma(2)}, \ldots, s_{\sigma(n)})$ *doit vérifier :*

$$X \overset{s_{\sigma(1)}}{\rightsquigarrow} X'_1 \overset{s_{\sigma(2)}}{\rightsquigarrow} X'_2 \ldots \overset{s_{\sigma(n-1)}}{\rightsquigarrow} X'_{n-1} \overset{s_{\sigma(n)}}{\rightsquigarrow} Y$$

à condition que les conditions initiales d'application des règles $s_{\sigma(i)}$ *soient satisfaites à chaque étape.*

Il faut remarquer que l'on n'impose pas que pour toute permutation la séquence $(s_{\sigma(1)}, s_{\sigma(2)}, \ldots, s_{\sigma(n)})$ soit valide. Nous verrons cela sur le problème des missionnaires et des cannibales, qui est partiellement commutatif.

Définition 13.9 – Système commutatif – *Un système de production est dit commutatif s'il est monotonique et partiellement commutatif .*

Nous avons dit qu'il était possible de définir plusieurs systèmes de production pour un même système. Il est en fait possible de construire pour tout problème un système de production commutatif qui le résolve. Mais ce système peut être totalement inefficace en terme d'espace et de temps de calcul. Nous verrons dans un prochain chapitre comment choisir un système de production adapté, en fonction des caractéristiques du problème. Examinons auparavant quelques exemples.

---

4 « monotonique » est une traduction de l'anglais adoptée par nombre d'auteurs.

13.3.4 Exemples

Nous allons reprendre les exemples du paragraphe précédent et définir pour chacun d'entre eux un système de production adapté.

Les missionnaires et les cannibales : On peut ici décrire le système de production par les règles suivantes (rappelons que X représente le nombre de missionnaires sur la rive de gauche, Y le nombre de cannibales sur la même rive et P la position de la barque) :

$$
\begin{array}{llcll}
X \geq 2 & X-2 \geq Y & : & (X,Y,G) & \rightsquigarrow (X-2,Y,D) \\
 & & : & (2,Y,G) & \rightsquigarrow (0,Y,D) \\
Y \geq 2 & 3-X \geq (3-Y)+2 & : & (X,Y,G) & \rightsquigarrow (X,Y-2,D) \\
Y \geq 2 & & : & (3,Y,G) & \rightsquigarrow (3,Y-2,D) \\
X \geq 1 & X-1 \geq Y & : & (X,Y,G) & \rightsquigarrow (X-1,Y,D) \\
 & & : & (1,Y,G) & \rightsquigarrow (0,Y,D) \\
Y \geq 1 & 3-X \geq (3-Y)+1 & : & (X,Y,G) & \rightsquigarrow (X,Y-1,D) \\
Y \geq 1 & & : & (3,Y,G) & \rightsquigarrow (3,Y-1,D) \\
X \geq 1 & Y \geq 1 & : & (X,Y,G) & \rightsquigarrow (X-1,Y-1,D) \\
3-X \geq 2 & (3-X)-2 \geq 3-Y & : & (X,Y,D) & \rightsquigarrow (X+2,Y,G) \\
 & & : & (1,Y,D) & \rightsquigarrow (3,Y,G) \\
3-Y \geq 2 & X \geq Y+2 & : & (X,Y,D) & \rightsquigarrow (X,Y+2,G) \\
3-Y \geq 2 & & : & (0,Y,D) & \rightsquigarrow (0,Y+2,G) \\
3-X \geq 1 & (3-X)-1 \geq 3-Y & : & (X,Y,D) & \rightsquigarrow (X+1,Y,G) \\
 & & : & (2,Y,D) & \rightsquigarrow (3,Y,D) \\
3-Y \geq 1 & X \geq Y+1 & : & (X,Y,D) & \rightsquigarrow (X,Y+1,G) \\
3-Y \geq 1 & & : & (0,Y,D) & \rightsquigarrow (0,Y+1,G) \\
3-X > 1 & 3-Y > 1 & : & (X,Y,D) & \rightsquigarrow (X+1,Y+1,G)
\end{array}
$$

Muni de ces règles, il est clair que partant d'un état valide, je peux générer tous les états valides qui en découlent, et seulement les états valides. Ainsi partant de l'état initial, et appliquant à chaque étape l'ensemble des règles de production, je dois générer l'ensemble des états possibles et en particulier l'état final s'il appartient à l'ensemble des états possibles[5]. Le système de production que nous avons défini a la particularité de pouvoir être utilisé aussi bien pour un raisonnement déductif que pour un raisonnement inductif. On remarque également que ce système est non monotonique et partiellement commutatif.

La démonstration de théorèmes : Les règles d'inférence de l'arithmétique formelle constituent un système de production pour la démonstration de théorèmes. Si nous considérions un système de Post de l'arithmétique formelle, les règles de production (au sens de la théorie des langages) seraient exactement les règles de production de notre problème. Ce système de production est de type déductif, monotonique, et partiellement commutatif donc commutatif. En revanche, les mécanismes de résolution que nous avons introduits au chapitre 6 constituent un système de production inductif.

---

5 Nous ne considérons pas ici la technique utilisée pour générer ces états (c'est-à-dire pour se déplacer dans l'espace d'états), ni la méthode utilisée pour représenter le monde d'états ainsi généré. Nous aborderons plus loin ces techniques.

	Monotonique	Non-monotonique
Partiellement commutatif	Démonstration de théorème	Missionnaires et cannibales
Non partiellement commutatif	Voyageur de commerce	Poker

Table 13.1 – Caractéristiques de problèmes

Le jeu d'échecs : L'ensemble des règles de déplacement des pièces constitue un système de production du jeu d'échecs. Remarquons que ce n'est pas le seul. En fait, on peut appeler ce système de production, par analogie avec la théorie des ensembles, un système défini en compréhension. On pourrait aussi définir un système en extension, qui consisterait à stocker l'ensemble des états possibles et l'ensemble des liaisons possibles entre chaque état. Cette méthode est bien entendu prohibitive aux échecs, vu le nombre de configurations possibles, mais possible en théorie car le nombre d'états est fini. Un système de production inductif présenterait peu d'intérêt pour le jeu d'échecs, sauf dans un cas : si l'on souhaite rechercher s'il existe un moyen d'arriver à une position donnée (problème de rétro-analyse) ; ce type de problème n'a pas grand-chose à voir avec le jeu d'échecs à proprement parler.

Le problème du voyageur de commerce : Soit un état $(x_1, x_2, \ldots, x_k)$, on construit les états $(x_1, x_2, \ldots, x_k, x_{k+1})$ où x_{k+1} décrit l'ensemble des villes différentes de $x_1, x_2, \ldots, x_k$. On génère ainsi l'ensemble des états possibles, qui contient forcément la solution. Il est impossible de construire un mécanisme inductif puisqu'on ne connaît pas l'état final de façon exacte. Ce système de production est monotonique puisque le choix d'une ville a à un instant t n'empêche pas de choisir une ville b à t' si b était aussi disponible à t. En revanche, il n'est pas commutatif, car l'ordre du choix des villes influe sur la solution.

Le jeu de Poker : Le système de production est également simple : il s'agit des règles régissant les enchères et la valeur des mains. Ce système n'est ni partiellement commutatif, ni monotonique : toute enchère est irréversible et entraîne des modifications sur la suite du déroulement de la partie. Remarquons cependant que nous ne pouvons pas prévoir, à un instant donné, le chemin suivi vers la solution. Nous allons y revenir. La notion de système de production inductif est ici sans intérêt.

Dans le tableau 13.1, sont résumées les caractéristiques de nos problèmes relativement à la monotonie et à la commutativité.

Il nous reste maintenant à trouver une représentation formelle adaptée à nos espaces d'états et à dégager des méthodes de résolution aussi efficaces que possibles.

13.4 Représentation formelle

Nous allons nous intéresser maintenant à la représentation formelle de l'espace d'états construit par un système de production.

13.4.1 Arbres

La méthode de représentation par arbre consiste, à chaque fois que l'on génère un état, à le relier simplement à son prédécesseur sans faire aucun test d'occurrence : on ne vérifie pas si l'état a déjà été généré.

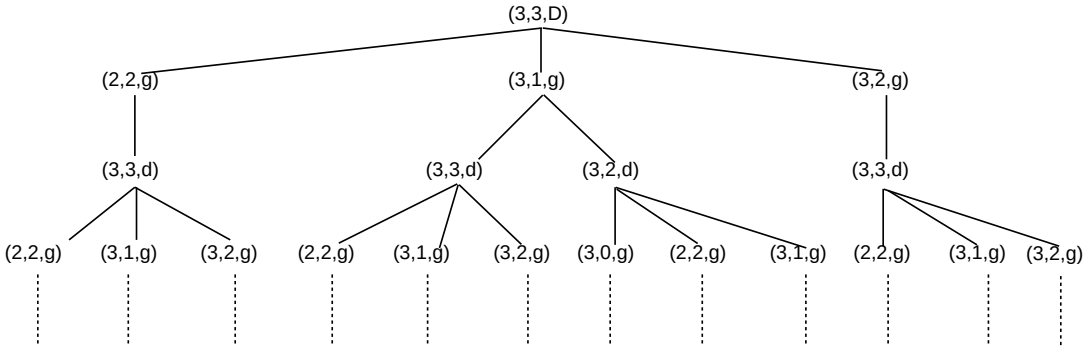

Figure 13.1 – Recherche en utilisant un arbre

L'avantage principal de la méthode est sa rapidité. Un test d'occurrence est toujours chose coûteuse en temps. La méthode de représentation par arbre est très utilisée en théorie des jeux, où le risque de voir une position se dupliquer est faible[6].

Ses inconvénients sont :
- la duplication d'état, qui peut entraîner un encombrement mémoire excessif ;
- le risque de ralentissement global dans la recherche de la solution : en ne s'apercevant pas que l'on génère plusieurs fois les mêmes états, on peut décrire plusieurs fois sans s'en apercevoir le même parcours. À l'extrême, on peut arriver à des cas de bouclages empêchant le système de parvenir à la solution.

Un exemple d'arbre appliqué au problème des missionnaires et des cannibales est représenté dans la figure 13.1.

13.4.2 Graphes

La méthode des graphes est similaire à celle des arbres. Chaque nouvel état généré est stocké, mais on effectue avant le stockage un test d'occurrence : si l'état a déjà été généré, on se contente de construire un arc indiquant le rebouclage.

Les avantages et les inconvénients de la méthode des graphes sont la duplication en creux des avantages et des inconvénients de la méthode des arbres. Un exemple de recherche en utilisant un graphe pour résoudre le problème des missionnaires et des cannibales est représenté figure 13.2.

13.4.3 Graphes ET-OU

Dans les graphes tels que nous les avons présentés tout nœud suivi d'un certain nombre d'arcs signifie que l'on peut passer de ce nœud à n'importe lequel des nœuds qui lui sont connectés[7]. On parle alors de graphe (ou d'arbre) *OU*.

---

6 Pas dans tous les cas cependant. Ainsi, les meilleurs programmes d'échecs traitent différemment les finales, voir chapitre 15.

7 Dit autrement : on peut considérer qu'un nœud mène à une solution si le premier fils *ou* le deuxième fils *ou* ... le n-ième fils mène à la solution.

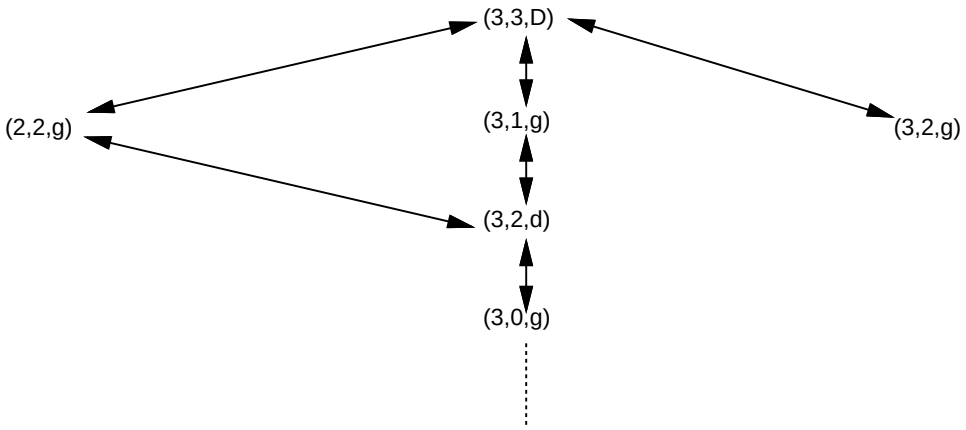

Figure 13.2 – Recherche en utilisant un graphe

Cependant, dans certaines catégories de problèmes, il est parfois utile d'utiliser une représentation plus complexe, appelée représentation *ET-OU*. Dans ce type de représentation, il part toujours plusieurs arcs de chaque nœud, mais certains arcs sont reliés entre eux. Cela signifie que pour résoudre le nœud courant, il faut résoudre tous les sous-nœuds auxquels il est raccordé par un de ces « paquets d'arcs »[8] ou hyper-arcs.

Voici un exemple de représentation d'arbre de résolution *ET-OU* en démonstration automatique de théorème appliquée à la formule $p \wedge ((q \vee (r \wedge s))$ en utilisant un système de production inductif (figure 13.3).

13.5 Stratégies de résolution

Il s'agit ici d'examiner les différentes stratégies permettant de parcourir un arbre de la façon la plus efficace possible.

13.5.1 Algorithme du British Museum

À chaque étape, on génère un seul nœud en choisissant au hasard une règle de production. À partir de ce nouveau nœud on itère le processus jusqu'à l'obtention d'une solution. Si, à un instant donné, on aboutit à une impasse (plus de règles de production sélectionnables), on recommence l'opération depuis la racine de l'arbre.

Cet algorithme[9] est extrêmement inefficace. Il ne garantit même pas qu'une solution sera trouvée.

---

8 Ou encore : un nœud ne mène à la solution que si l'ensemble de ses fils mène à une solution.

9 Le nom de l'algorithme vient du paradoxe suivant : si on laissait des singes devant des machines à écrire, ils réaliseraient une implantation de cet algorithme, et produiraient, si on leur laissait assez de temps, tous les livres de la bibliothèque du musée (preuve statistique).

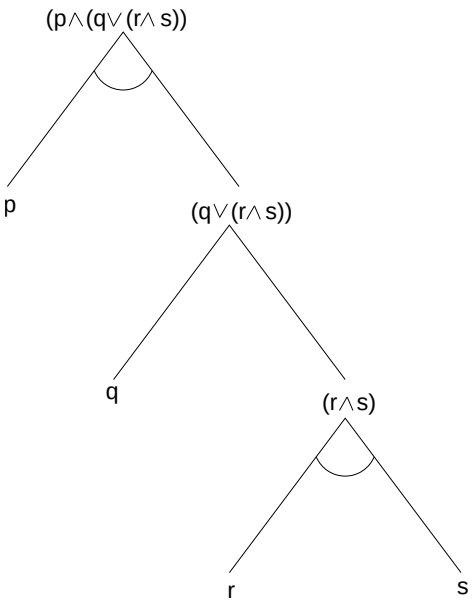

Figure 13.3 – $p \wedge ((q \vee (r \wedge s))$ représenté par un arbre *ET-OU*

13.5.2 Recherche en profondeur et retour-arrière

Il s'agit d'une implantation plus intelligente de l'algorithme précédent[10]. À chaque échec, on réalise un retour au nœud précédant le nœud où l'on a constaté l'impasse et on choisit une autre règle de production parmi celles qui n'ont pas été encore utilisées. Celle-ci générera ainsi un autre nœud. Nous voyons sur la figure 13.4 une recherche dans un arbre en utilisant cet algorithme. Le but de la recherche est de trouver une position dont la valuation est supérieure à 50. Les chiffres en italique représentent la valuation des nœuds et les chiffres gras le numéro de l'étape à laquelle l'algorithme génère le nœud.

Quelques remarques s'imposent. Tout d'abord, cet algorithme peut, dans un cas défavorable, nous obliger à parcourir l'intégralité de l'espace d'état avant de trouver la solution. Il est incapable de profiter de la structure du problème pour éviter certains chemins absurdes. Enfin, il ne garantit d'aboutir à la solution que si l'on effectue des tests d'occurrence pour éviter d'éventuels bouclages. Sinon, il peut rentrer dans un cycle et ne plus en sortir, ce qui entraînera le développement d'une branche infinie dans l'arbre de recherche.

On peut appliquer ce type de mécanisme sur des problèmes comprenant un espace d'états réduit, comme par exemple le problème des missionnaires et des cannibales.

Il peut aussi être appliqué pour prouver l'inconsistance d'une formule sous forme clausale de Horn en calcul propositionnel ou en logique du premier ordre. C'est d'ailleurs la méthode utilisée par la plupart des langages PROLOG[11].

---

10 En anglais : *depth-first*.

11 Malheureusement, cette technique rend la résolution PROLOG incomplète (voir chapitre 18).

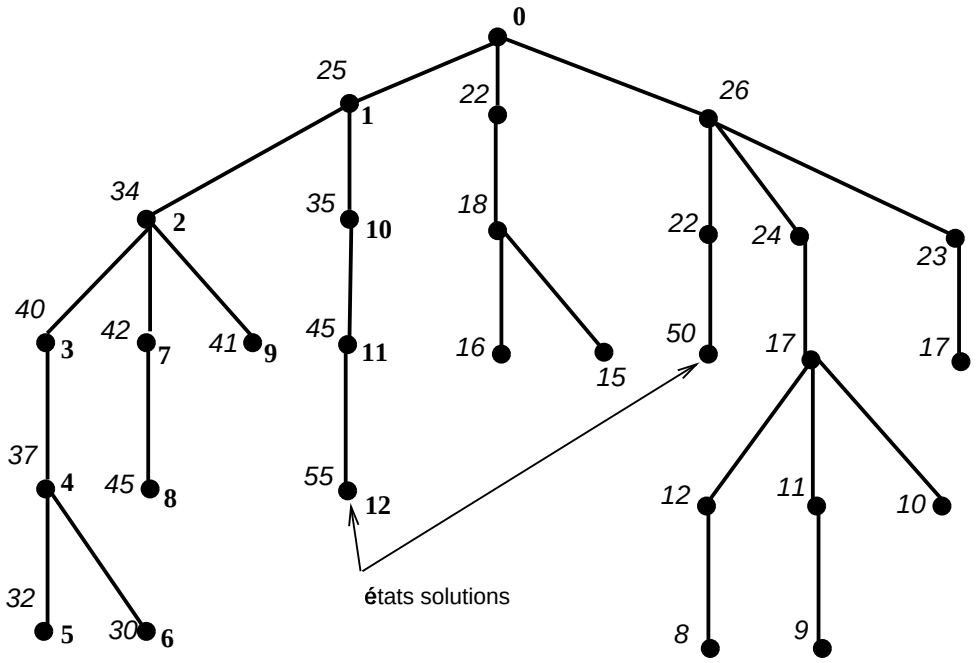

Figure 13.4 – Recherche en profondeur et retour-arrière

13.5.3 Recherche en largeur

La recherche en largeur consiste à générer l'ensemble des nœuds niveau par niveau. L'avantage de la recherche en largeur est qu'elle garantit de trouver une solution, même s'il existe un cycle. D'autre part, cette solution sera également la moins profonde. Son inconvénient majeur est sa grande consommation de mémoire[12]. Comme la recherche en profondeur, la recherche en largeur peut, dans des cas défavorables, être amenée à considérer l'intégralité de l'arbre et ne sait pas tenir compte de la structure du problème pour améliorer la recherche (figure 13.5).

13.5.4 La notion d'heuristique

Une heuristique[13] a pour but de diriger la recherche dans l'arbre, de façon à réduire le temps de résolution du problème ; elle introduit des mécanismes qui dérivent de connaissances spécifiques au problème.

Il est commun de combiner l'utilisation d'une heuristique avec des méthodes de recherche ne garantissant plus la complétude de la résolution, afin d'améliorer encore l'efficacité. Ce type de technique est justifiable, car, bien souvent, on ne cherche pas la meilleure solution à un problème mais simplement une solution satisfaisante : une personne cherchant à établir un emploi du temps ne cherche pas le meilleur emploi du

12 Un moyen terme intéressant est la recherche par « approfondissement itératif » : on effectue une recherche en profondeur d'abord limitée à la profondeur 1, puis 2… jusqu'à résolution. Le coût mémoire est fortement réduit alors que le temps machine est asymptotiquement du même ordre (Cf. (Korf 1985)).

13 *heuristique* : emprunté au grec *euristikê, (tekhnê)*, [art] de découvrir. *Dictionnaire étymologique Larousse.*

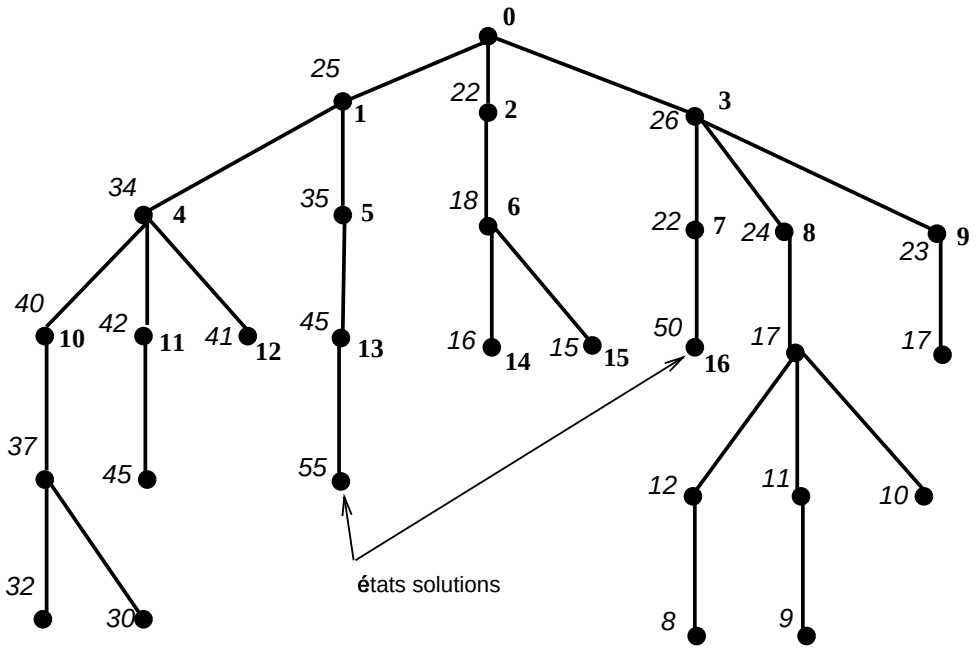

Figure 13.5 – Recherche en largeur

temps possible , mais un bon compromis entre qualité et temps de construction. Ce type d'exemple est très général : il va de même lorsqu'on cherche une démonstration de théorème, un circuit passant par plusieurs villes ou une place de parking.

Malheureusement, introduire une heuristique ne suffit pas toujours à résoudre un problème, même de façon à peu près satisfaisante. Tout dépend de la complexité du problème et de la qualité de l'heuristique. Nous examinerons de façon plus formelle différents types d'heuristiques en présentant les algorithmes de type A.

La notion de *recherche heuristique* est une des notions de base de l'intelligence artificielle.

13.5.5 L'escalade

L'escalade[14] est une méthode de recherche en profondeur dans laquelle on introduit une fonction heuristique pour choisir à chaque étape le nœud à générer, au lieu de générer un nœud en prenant une règle de production au hasard.

Ainsi, considérons le problème du voyageur de commerce. Si nous générons les nœuds en utilisant l'algorithme de recherche en profondeur avec retour-arrière, il se peut que nous générions $n!$ nœuds.

Si nous choisissons comme heuristique de générer à chaque fois le nœud qui ajoute à la liste des villes déjà visitées la ville la plus proche de la dernière traversée, notre algorithme a une complexité en : $n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$ donc une complexité

14 *hill-climbing* en anglais.

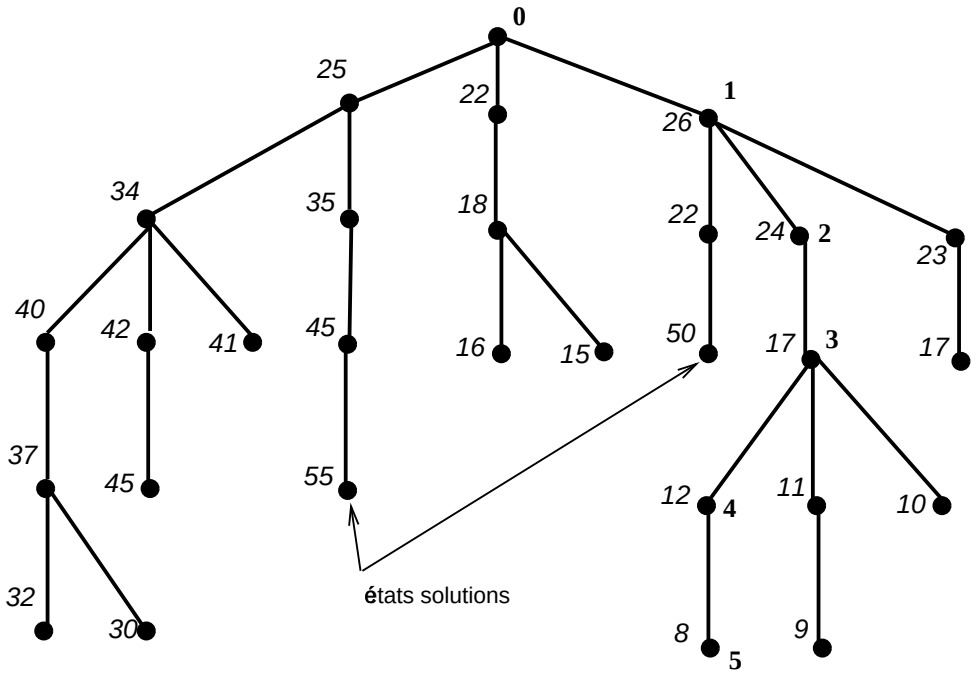

Figure 13.6 – Escalade et recherche en profondeur

polynomiale. Certes, la solution trouvée n'est pas forcément la meilleure[15], et il nous faudrait poursuivre la recherche et générer autant de nœuds que précédemment pour trouver la meilleure solution. Mais la solution approchée apparaît comme admissible à la plupart des gens et peut donc, bien souvent, être retenue.

La recherche en escalade peut aussi être appliquée classiquement (figure 13.6). Dans le cas de notre problème, c'est la valuation d'un nœud qui sert de fonction heuristique. Comme le montre la figure, le résultat donné par « escalade » dans ce cas n'est même pas une solution convenable. L'escalade reste une technique intéressante dans le cas de systèmes commutatifs.

13.5.6 Recherche *meilleur en premier* : graphe OU

La recherche *meilleur en premier*[16] est un compromis entre une recherche en largeur et une recherche en profondeur, en utilisant une fonction heuristique pour guider la génération des nœuds.

Le mécanisme est le suivant : on développe à chaque étape le meilleur nœud encore non développé. On inclut les nœuds ainsi générés dans l'ensemble des nœuds connus et on recommence le processus (figure 13.7). Nous allons présenter un raffinement de cette méthode de façon plus formelle dans le cadre des graphes OU et des problèmes de

15 On a même pu voir, au chapitre 11, que pour ce problème, il est impossible de borner la perte de qualité qui résulte de l'emploi d'un algorithme polynomial, *qu'il soit d'origine IA ou non*.

16 *best-first* en anglais.

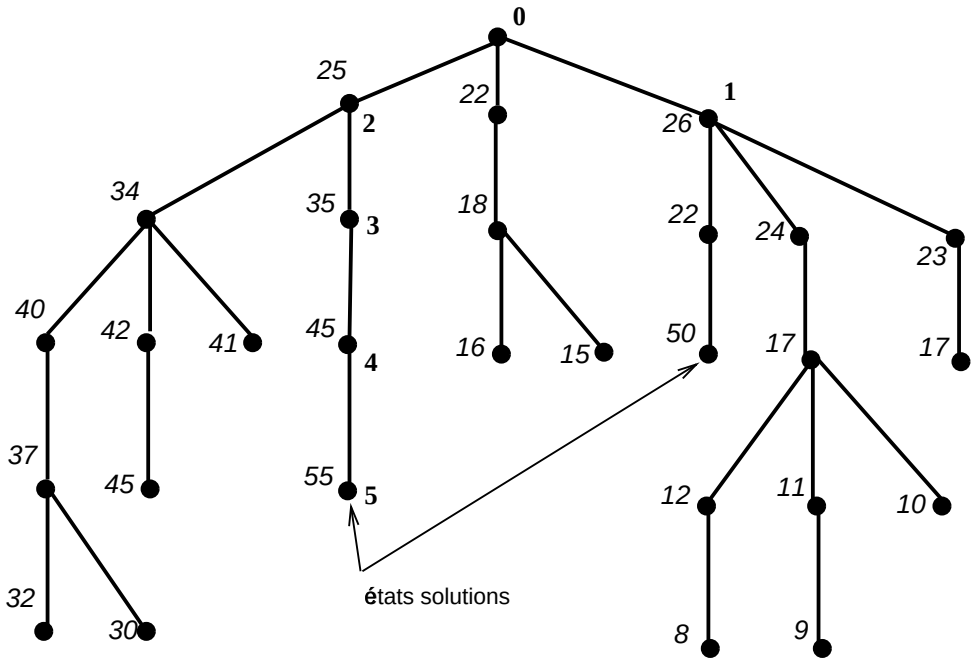

Figure 13.7 – Recherche meilleur en premier

minimisation de coût additif : l'algorithme[17] A^*, chargé de calculer le plus court chemin menant d'un état initial à un état final.

Les données du problème seront :

u_0 : l'état initial.

T : l'ensemble des états terminaux.

$\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$: l'ensemble des règles de production.

$h(u)$: u est un état, h est une fonction heuristique qui estime le coût de passage de u à l'état final.

$k(u, v)$: u et v sont des états, k est une fonction qui donne le coût du passage par l'arc (u, v).

Nous utiliserons comme variables annexes :

D : une liste contenant l'ensemble des états déjà développés (un état est développé si on a déjà généré l'ensemble de ses successeurs).

G : une liste contenant l'ensemble des états générés non encore développés.

$g(v)$: un tableau indexé par les états qui contient le coût du trajet pour atteindre l'état v.

$f(v)$: un tableau indexé par les états qui contiendra pour chaque état le coût total estimé.

père(v) : un tableau indexé par les états qui donne, pour chaque état, l'état qui l'a généré.

17 Cet algorithme a été présenté pour la première fois dans (Hart *et al.* 1968).

Nous utiliserons également deux fonctions annexes, *premier*(G) qui renverra le premier élément de G, et *insérer-selon-*$f(G, v)$ qui insérera v dans G dans l'ordre f croissant puis g décroissant.

Il ne nous reste plus qu'à présenter l'algorithme :

1. $G \leftarrow (u_0)$; $D \leftarrow \varnothing$; $g(u_0) \leftarrow 0$; $f(u_0) \leftarrow 0$
2. **Tant que** $G \neq \varnothing$ **faire**
   (a) $u \leftarrow$ *premier*(G) ; $G \leftarrow G \setminus (u)$; $D \leftarrow D \cup (u)$
   (b) **Si** $u \in T$ **alors** imprimer_solution ; **fin_programme** ; **fin_si**
   (c) **Pour** i **allant de** 1 **jusqu'à** n **faire**
      i. $v \overset{p_i}{\leftarrow} u$
      ii. **Si** $[v \notin D \cup G]$ **ou** $[g(v) > g(u) + k(u, v)]$ **alors**
      iii. $g(v) \leftarrow g(u) + k(u, v)$
      iv. $f(v) \leftarrow g(v) + h(v)$
      v. *père*$(v) \leftarrow u$
      vi. *insérer-selon-*$f(G, v)$[18]
      vii. **fin_si**
   (d) **fin_faire**
3. **fin_faire**

Quelques remarques s'imposent.

– la fonction $k(u, v)$ peut être prise identiquement nulle si on ne se préoccupe pas de la longueur du chemin menant à la solution. Si on cherche le chemin comprenant le moins d'arcs possibles, on prendra $k(u, v) = 1$ pour tout arc (u, v).
– la fonction $h(u)$ est une heuristique qui estime la distance entre l'état courant et l'état final. De façon formelle, $h(u)$ tente d'estimer le minimum sur tous les chemins $(u, v_1, \ldots, v_n, u_t)$, où u_t est un état terminal, de la somme :

$$k(u, u_1) + k(u_1, u_2) + \cdots + k(u_n, u_t)$$

La valeur réelle de ce minimum est notée $h^*(u)$. On dégage plusieurs catégories de fonction heuristique h.

Définition 13.10 – Heuristique parfaite – *Une heuristique h est dite parfaite si et seulement si :*

$$\forall u, v, h(u) = h(v) \Leftrightarrow h^*(u) = h^*(v)$$

Définition 13.11 – Heuristique presque parfaite – *Une heuristique h est dite presque parfaite si et seulement si :*

$$\forall u, v, h(u) < h(v) \Rightarrow h^*(u) < h^*(v)$$

18 Nous rappelons au lecteur que la fonction *insérer-selon-*f utilise la fonction (ou tableau) f pour insérer v à la bonne position dans la liste G.

Définition 13.12 – Heuristique monotone – *Une heuristique h est dite monotone (ou consistante) si et seulement si :*

$$\forall u, \forall v \textit{ descendant de } u, h(u) - h(v) \leq k(u, v)$$

Définition 13.13 – Heuristique minorante – *Une heuristique h est dite minorante (ou admissible) si et seulement si :*

$$\forall u, h(u) \leq h^*(u)$$

- si l'heuristique h est parfaite, alors l'algorithme convergera immédiatement vers le but ;
- si h est une heuristique minorante alors l'algorithme A^* trouvera toujours le chemin optimal ;
- une heuristique monotone est non seulement admissible, mais garantit, dans le cadre de l'algorithme A^*, que pour tout nœud u développé ($u \in D$), le chemin calculé est optimal ($g(u)$ est effectivement égal au coût du plus court chemin menant à u).

On s'est également intéressé à la complexité de A^*.

- dans le pire des cas, la complexité est en 2^N, où N est le nombre de nœuds du graphe, puisque chaque état u sera développé autant de fois qu'il y a de chemins de u_0 à u ;
- si h est une heuristique minorante, on peut, en modifiant légèrement l'algorithme, avoir une complexité en N^2 ;
- Si h est monotone, la complexité est linéaire.

Cependant, il faut remarquer que même une complexité linéaire sur le nombre d'états est clairement inutilisable dans la plupart des cas pratiques. Ainsi, le problème du voyageur de commerce a un nombre d'états en $N = n!$ où n est le nombre de villes.

On peut s'intéresser à un autre type de complexité, celle du nombre de nœuds que l'on considérera en fonction du nombre d'arcs entre l'état initial et un état final, que nous noterons M. Quelques résultats connus sont :

- si h est presque parfaite, alors la complexité est linéaire en M ;
- si h est monotone, en considérant un arbre dont le facteur de branchement est K, la complexité est exponentielle en K^M ;
- Pour une heuristique minorante, toujours pour un arbre K-aire, on peut déterminer la complexité de l'algorithme en fonction de l'écart entre h et h^* :

Erreur	Complexité
$(1-r)h^* \leq h$	$K^{\alpha M}$
$h^* - \sqrt{h^*} \leq h$	$\sqrt{M}K^{\sqrt{M}}$
$h^* - \log h^* \leq h$	$M^{\log K}$
$h^* - r \leq h$	MK^r

Enfin, notons qu'un certain nombre de résultats à caractère général montre que si on ne dispose pas d'une « très bonne » heuristique (cas le plus fréquent), l'algorithme A^* développera en moyenne un nombre exponentiel d'états en fonction de la longueur optimum

du chemin. Pour un énoncé plus détaillé de toutes ces propriétés on peut se reporter à (Farreny and Ghallab 1987) ou à (Pearl 1990).

L'algorithme A^* est extrêmement populaire malgré les résultats que nous venons d'énoncer. En effet, il est général, simple à implanter et amplement suffisant pour nombre de problèmes de taille réduite. D'autre part, on sait qu'il est « optimal », c'est-à-dire qu'il fait « au mieux » avec les informations dont il dispose. Pour résoudre les problèmes de consommation mémoire que l'A^* fait apparaître avec facilité, des algorithmes de recherche en approfondissement itératif (*iterative deepening*, assez semblables à ceux construits au-dessus de l'algorithme α-β) ont été développés (Korf 1985; Sarkar *et al.* 1991).

Enfin, signalons le développement d'algorithme à seuil, ou ε-admissible. Ces algorithmes ne cherchent pas forcément la meilleure solution, mais une solution dont l'écart relatif par rapport à l'optimum est inférieur à ε. Il existe ainsi un algorithme A^*_ε qui applique ce principe à l'algorithme A^* (cf. (Farreny and Ghallab 1987; Pearl 1990)).

13.5.7 Recherche *meilleur en premier* : graphe ET-OU

L'algorithme A^*, que nous venons de présenter, ne peut s'appliquer dans des graphes ET-OU. En effet, nous ne devons plus considérer des chemins de moindre coût, mais des groupes de chemins, puisque pour résoudre un nœud, il faut résoudre plusieurs sous-nœuds.

Une extension de l'algorithme A^* a été définie dans ce but. Il s'agit de l'algorithme AO^*[19]que nous ne présenterons pas (voir (Farreny and Ghallab 1987) ou (Pearl 1990)).

13.6 Analyse d'un problème

Nous allons tenter dans ce paragraphe de dégager les éléments majeurs qui caractérisent un problème[20]. L'analyse des caractéristiques d'un problème permet de choisir de façon adéquate le système de production ainsi que le type de représentation formelle et la stratégie de résolution.

13.6.1 Calculabilité et complexité du problème

Nous avons consacré un chapitre entier à la complexité et à la calculabilité, nous supposons acquises les principales notions.

Un problème non-calculable ne peut pas être résolu *dans le cas général*, que ce soit par des méthodes heuristiques ou par d'autres méthodes. Un problème polynomial peut

---

19 Cet algorithme a été présenté pour la première fois dans (Martelli and Montanari 1978).

20 Nous reprenons pour partie la classification de (Rich 1987), simplement nous ne nous intéressons pas aux systèmes utilisant des bases de connaissances importantes ou contradictoires, car nous avons estimé que ce type de problème relevait plus des systèmes experts que des techniques de résolution que nous présentons dans ce chapitre. De même, nous ne nous sommes guère intéressés aux problèmes d'interaction avec l'opérateur que nous développerons plus en détail lorsque nous parlerons des systèmes experts.

(presque) toujours être résolu par des stratégies de parcours classiques (largeur ou profondeur). Pour un problème NP-complet, il est bon d'examiner avec attention dans quel cadre nous souhaitons utiliser notre programme. De façon générale, les méthodes heuristiques peuvent apporter un gain important en terme de temps de résolution (par rapport à la force brute non guidée), mais seul le renoncement à la résolution d'un problème NP-complet (recherche d'une solution seulement « proche » de l'optimum, satisfiabilité partielle) peut permettre d'aboutir à une complexité raisonnable dans le cas général et sur des tailles importantes (cf. chapitre 11).

Examinons chacun de nos cinq problèmes :

Les missionnaires et les cannibales : Le nombre d'états de ce problème est faible. Il est possible de le résoudre par une stratégie de recherche en largeur dans un graphe.

La démonstration de théorème : En stratégie déductive, des méthodes heuristiques doivent être employées pour tenter de limiter l'explosion combinatoire. En stratégie inductive sur des clauses de Horn, il est possible d'utiliser une stratégie de recherche en profondeur.

Le jeu d'échecs : Il est évident que la complexité du problème (cf. chapitre 11) est telle que des méthodes particulières devront être utilisées.

Le voyageur de commerce Le problème est connu pour être NP-complet et n'avoir aucune ε-approximation. Il est donc évident que pour un nombre important de villes une méthode approchée devra être employée.

Le jeu de poker : Comme pour le jeu d'échecs, des stratégies adaptées doivent être utilisées en raison de l'importante complexité du problème.

13.6.2 Problèmes décomposables

Un problème est décomposable si on peut le réduire en un ensemble de problèmes plus petits que l'on essaiera alors de résoudre *séparément*. Pouvoir décomposer un problème est un grand avantage, car cela réduit considérablement la complexité de la résolution.

Nous allons examiner chacun de nos cinq problèmes sous cet aspect :

Les missionnaires et les cannibales : Le problème n'est clairement pas décomposable. Résoudre le problème avec deux missionnaires ou deux cannibales au lieu de trois ne permet pas de résoudre le problème avec trois individus.

La démonstration de théorèmes : C'est un excellent exemple de problèmes décomposables. Supposons en effet que nous ayons à démontrer $p \wedge (q \vee (r \wedge s))$. Nous pouvons réaliser une première décomposition en deux sous-problèmes : la démonstration de p et de $q \vee (r \wedge s)$. Mais nous pouvons alors développer la seconde expression en $(q \vee r) \wedge (q \vee s)$. Ce sous-problème se décompose à son tour en deux autres sous-problèmes, $q \vee r$ et $q \vee s$. De façon générale tout $\wedge$ permet de décomposer un problème en sous-problèmes.

Le jeu d'échecs : Le jeu d'échecs n'est pas décomposable. Cependant, nous verrons que certaines techniques de résolution réalisent des décompositions d'un problème général en sous-problèmes (composition de plans stratégiques de résolution). Cet exemple montre que la barrière n'est pas toujours aussi nette qu'on peut le penser.

Le problème du voyageur de commerce : Ce problème n'est pas décomposable. Le résoudre pour quatre villes données ne permet pas de calculer plus simplement la solution si l'on rajoute une cinquième ville.

Le jeu de Poker : Clairement non décomposable.

Si un problème est décomposable, on aura souvent intérêt à choisir une stratégie de résolution par chaînage arrière, ou du moins à faire intervenir une telle stratégie. D'autre part une technique de représentation par arbres ou graphes ET-OU est particulièrement adaptée.

13.6.3 Problèmes prévisibles

Un problème est dit *prévisible* (ou encore « à information totale ») si l'ensemble des données relatives à la résolution est parfaitement connu. Si un problème n'est pas prévisible, les stratégies de résolution doivent inclure des méthodes probabilistes de raisonnement. Examinons nos cinq problèmes :

Les missionnaires et les cannibales : prévisible.

La démonstration de théorème : prévisible.

Le jeu d'échecs : prévisible.

Le voyageur de commerce : prévisible.

Le jeu de poker : non prévisible. En effet, on ne peut connaître, à un instant donné, le jeu de son (ou ses) adversaire(s). Ainsi, nous ne pouvons construire de façon certaine une solution. Tout au plus pouvons-nous prévoir avec une certaine probabilité quel type de séquence il nous faut suivre. De plus, il nous faudra à chaque enchère nouvelle, recalculer les probabilités et reconsidérer le plan à exécuter.

Les problèmes non prévisibles sont des problèmes extrêmement délicats à résoudre. Il ne faut jamais espérer trouver des solutions certaines et toujours prévoir des stratégies de contrôle évolutives. Les techniques de parcours de graphes telles que nous les avons présentées sont souvent inutilisables.

13.6.4 Problèmes ignorables ou récupérables

On dit qu'un problème est *ignorable* si l'application d'une règle du système de production produisant un état inutile n'influencera en rien la suite du problème. Le problème est dit *totalement récupérable* si la génération d'un état inutile pourra être annulée par la suite par une autre règle du système de production. Le système est *partiellement récupérable* si nous devons implanter dans la structure de contrôle une technique consistant à revenir avant le point incorrect. Dans tout autre cas, le problème est dit irrécupérable. Remarquons que le problème de la récupération se pose surtout pour les stratégies de contrôle en profondeur, jamais pour les stratégies en largeur.

Reprenons une fois de plus nos cinq problèmes :

Les missionnaires et les cannibales : Le problème n'est pas ignorable mais il est totalement récupérable. En effet, si j'applique une règle i déplaçant des missionnaires ou des cannibales, je peux appliquer la règle j inverse de la règle i (c'est-à-dire celle qui replace le problème dans sa forme initiale).

La démonstration formelle : Il s'agit d'un exemple de problème ignorable. En effet, si je génère, lors de ma démonstration, un résultat inutile, cela n'a strictement aucune importance.

Le jeu d'échecs : Si le but de notre programme est de trouver un état final à partir d'un état initial (problème de mat), le système est non-ignorable et partiellement récupérable. En effet, il nous faut implanter dans la structure de contrôle la notion de retour en arrière pour pouvoir éliminer l'étape incorrecte dans notre résolution.
Si notre programme d'échecs doit, non plus résoudre des mats, mais bien jouer aux échecs, alors, pour des raisons de complexité, il est clair que le but n'est plus de calculer un état final, mais bien de jouer à chaque étape un coup aussi bon que possible. Le problème est que tout coup réellement joué ne peut plus être annulé : le problème est irrécupérable.

Le voyageur de commerce : Le problème du voyageur de commerce est un problème non-ignorable et récupérable.

Le jeu de poker : C'est un exemple parfait de système irrécupérable. Toute enchère effectuée ne peut être reprise.

Nous avons vu que la notion de problème récupérable est souvent liée à la technique de résolution et à la stratégie de contrôle. De façon générale, des problèmes que nous n'espérons pas résoudre en une étape, et qui se dérouleront avec une interaction avec le milieu extérieur (échecs, poker), sont toujours irrécupérables.

Les problèmes ignorables sont toujours des problèmes que l'on résoudra par des systèmes de production commutatifs.

13.6.5 Optimisation ou satisfiabilité

Comme le fait remarquer (Simon and Kadane 1975), il faut distinguer soigneusement les problèmes d'optimisation (trouver la meilleure solution), des problèmes de satisfiabilité (trouver une solution qui satisfait à).

Ainsi dans les cinq cas que nous étudions :

Les missionnaires et les cannibales : Problème de satisfiabilité, nous nous contentons évidemment d'une solution (on préférait probablement la plus courte, mais rien ne l'indique dans l'énoncé du problème).

La démonstration formelle : Problème de satisfiabilité. Nous nous préoccupons assez peu de trouver la « meilleure » démonstration, d'autant qu'il est assez délicat de définir ce qu'est une « bonne » démonstration[21].

Le jeu d'échecs : S'il s'agit de résoudre des problèmes de mat, on peut considérer qu'il s'agit d'un problème de satisfiabilité, bien que dans certains cas, le nombre de coups pour mater soit minimisé. S'il s'agit au contraire de jouer aux échecs, le problème est un problème d'optimisation (trouver le meilleur coup possible à un instant donné)[22].

---

21 On pourrait rechercher une démonstration comprenant un nombre minimal d'inférences, mais d'autres critères entrent en jeu aux yeux du mathématicien.

22 Les deux problèmes sont tellement différents que les ordinateurs d'échecs disposent de techniques différentes pour les résoudre. S'il était envisageable de le résoudre, on pourrait, dès le départ, se poser le problème en terme de satisfiabilité : « trouver un coup gagnant ». Ce problème étant trop difficile, on se limite à chercher le coup qui semble le meilleur.

Le voyageur de commerce : Il s'agit d'un problème d'optimisation, mais on ne connaît aucune méthode efficace pour le résoudre. On le transforme donc en un problème de « semi-satisfiabilité » en relâchant les contraintes sur l'optimisation.

Le jeu de poker : Il s'agit bien entendu d'un problème d'optimisation.

13.7 Conclusion

Nous n'avons fait que rapidement présenter les principales stratégies d'exploration. Ces techniques, très générales, ont eu des emplois très variés en intelligence artificielle (preuve de théorèmes, satisfaction de formules, de contraintes, systèmes experts) et en recherche opérationnelle (*branch and bound*...).

De nombreuses sophistications, aux implications importantes, ont été apportées depuis quelques années aux algorithmes de base présentés ici. Le lecteur intéressé pourra se reporter à (Pearl 1990; Farreny and Ghallab 1987) et aux nombreuses publications sur le sujet (Korf 1985; Dechter and Pearl 1988b; Kanal and V.Kumar 1988; Korf 1988; Korf 1990; Sarkar *et al.* 1991; Korf 1987).