L'expression Intelligence Artificielle a été proposée en 1956 au cours d'un congrès. La question qui s'est posée concernait la définition d'une *machine intelligente*. Deux types de réponses ont été proposées :

1. Une machine intelligente reproduit le comportement d'un être humain.
2. Une machine intelligente modélise le fonctionnement d'un être humain.

L'ambition de fabriquer des machines intelligentes existait depuis l'antiquité ; *Hephaistos créa des femmes en or qui avaient des capacités de parler et de travailler*.

Jusqu'au milieu du XIXe siècle, la logique faisait partie de la philosophie. Ensuite la logique commence à migrer dans la discipline de la mathématique (Boole, 1850). La notion de systèmes formels utilisant un langage défini et des axiomes logiques a été ensuite élaborée. Ces systèmes formels seront ensuite utilisés dans la démonstration et la déduction automatique en intelligence artificielle.

Les deux notions de décidabilité et de la calculabilité ont été ensuite introduites en tant que critères dans les systèmes formels. Un système est décidable s'il existe une procedure permettant de décider en un temps fini si une formule est déduite des axiomes. D'un autre côté, une fonction est calculable s'il existe un algorithme qui permet de déterminer en un temps fini la valeur à partir des paramètres. La notion de calculabilité est parfaitement caractérisée par la notion de machine de Turing.

Aujourd'hui, les domaine de l'intelligence artificielle sont très diversifiés , nous citons : les systèmes experts, le calcul formel, la représentation des connaissances, la simulation du raisonnement humain, l'apprentissage et la résolution de problèmes. L'intelligence artificielle est aussi appliquée dans les domaines de reconnaissances de traitement de langage naturel. De point de vue sub-symbolic, nous citons les réseaux de neurones et les algorithmes génétiques.

L'objectif de ce poly est d'élaborer une introduction aux bases de resolution des problèmes et plus particulièrement les problèmes de planification et de satisfaction de contraintes ainsi qu'à la problématique de l'apprentissage automatique. Les algorithmes d'exploration dans un espace d'états sont détaillés et notamment les algorithmes d'exploration arborescente aveugle, heuristique et stratégique. Les éléments principaux de la théorie de jeux, tels que les noyaux de graphes et les algorithmes minmax et alpha-beta sont ensuite détaillés. Un ensemble de cours sur l'apprentissage symbolique est présenté afin d'illustrer à l'élève les techniques d'apprentissage de concepts, les arbres de décision, la résolution inverse, les règles d'association ainsi que l'inference grammaticale. Finalement, les algorithmes de résolution de contraintes utilisant le backtracking simple et amélioré ainsi que la notion de l'arc consistance sont présentés et validés sur des cas d'école.

Table des matières

Chapitre 1

Recherche dans un espace d'états

1.1 Introduction

L'objectif de ce chapitre est d'introduire les principes de la résolution d'un problème donné en utilisant un espace de recherche. Cet espace de recherche permet de recenser les états d'un système donné et de trouver parmi ces états une ou plusieurs solutions. Le passage d'un état à un autre se fait par l'application d'une action donnée. Nous verrons que le problème de recherche de l'état solution dans l'espace nous ramènera à développer *un arbre de recherche* et à définir une stratégie de recherche sur cet arbre. Le but de la recherche dans un tel arbre serait la diminution du temps de recherche en trouvant une stratégie qui converge rapidement vers la solution. Nous allons dans un premier temps présenter les algorithmes de recherche classique : en profondeur et en largeur qu'on appellera la recherche aveugle. Ce seront les algorithmes les plus coûteux évidemment en temps ou en espace car ils ont tendance à développer l'arbre *complet* pour aboutir à une solution où à l'ensemble de solutions. Ensuite, nous étudions l'algorithme A* qui intègre une heuristique courant le développement de l'arbre de recherche ; cette heuristique permettra d'élaguer un ensemble de branches de l'arbre et de converger rapidement vers une solution si cette solution existe. Dans la suite, nous définissons les notions utilisées à savoir : *les états et les actions* . Ensuite, nous montrons les techniques de résolution en effectuant une recherche dans un espace d'états. L'application la plus directe à ces méthodes sera les problèmes de planification pour lesquels on ne connaît pas d'algorithme général ; mais on sait que étant donnés, un état initial, un état final et un ensemble d'actions élémentaires valides nous permettant de passer d'un état à un autre, on doit trouver une séquence d'action à executer pour résoudre le problème.

Il existe d'autre types d'application des méthodes de recherche dans un espace comme la démonstration de théorèmes, les jeux et la résolution de contraintes .

1.2 Notions utilisées

Nous modélisons dans cette section un problème général de planification. Un problème de planification est modélisé par le quadruplet (S, E_0, F, T) où :

S est l'ensemble de tous les états.

E_0 est l'état initial, $E_0 \in S$

F est l'ensemble des états finaux $F \subset S$

T est la fonction de transition qui associe à chaque état E_i dans S un ensemble de couples (A_{ij}, E_{ij}) tels que A_j soit une action élémentaire permettant de passer de l'état E_i à l'états E_{ij}.

1.2.1 Comment résoudre un problème de planification ?

Résoudre un problème de planification signifie trouver une séquence $E_0, E_1, .., E_j, ..E_n$ tel que $\exists A_1, ..A_j, ..A_n$ tels que $(A_j, E_j) \in T(E_{j-1})$ et $E_n \in F$. Pour effectuer une recherche, un arbre de recherche est construit ; la racine de l'arbre correspond à l'état initial, un état E_{ij} est fils d'un autre état E_i s'il existe une action qui permet d'obtenir E_{ij} à partir de E_i. Si une des feuilles de l'arbre correspond à un état final, la solution est trouvée par la stratégie de la recherche.

1.3 Recherche aveugle

La recherche aveugle de la solution peut s'effectuer en profondeur ou en largeur, le parcours en profondeur signifie le développement d'une branche entière avant de parcourir le reste de l'arbre en effectuant du "backtracking". Ce développement peut ramener à trois situations différentes :

- La solution est trouvée et dans ce cas le développement de la branche s'arrête avec une possibilité de "backtraking" si d'autres solutions sont encore sollicitées.
- La solution n'est pas trouvée et un état d'échec est détecté (c'est un état qui n'engendre pas d'autres états) et dans ce cas le "backtraking" est appliqué pour poursuivre la recherche.
- Une branche infinie est à explorer et dans ce cas, un test d'arrêt sur une profondeur maximale doit être appliqué.

Nous remarquons que les performances de la recherche en profondeur sont liées à l'ordre d'exploration des branches de l'arbre. Autrement dit, si la solution se trouve dans la première branche à explorer cette stratégie devient optimale.

Par contre la recherche en largeur signifie que les états doivent être visités en parcourant l'arbre niveau par niveau ; autrement dit, tous les successeurs d'un état donné sont visités l'un après l'autre avant le passage au niveau suivant. Pour effectuer le parcours en largeur, une file est utilisée. Le parcours s'arrête quand un état final est trouvé ou quand une profondeur maximale est atteinte. Ce parcours est très cher en temps et en espace mais il garantie de trouver la solution si elle existe ; tandis que le parcours en profondeur peut ne pas converger même si la solution existe à cause d'une branche infinie.

Nous présentons l'algorithme d'exploration en profondeur ***ExplorationProf*** $(E_i, DejaVu, N)$ ***: BOOLEEN*** qui prend en paramètre E_i : l'état actuel en cours de visite, $DejaVu$: la liste des états déjà visités, et N : la profondeur de l'état actuel. Remarquer bien que la première appelle de cette fonction sera :$ExplorationProf(E_0, [E_0], d)$ où d est la profondeur maximale tolérée.

1.3.1 Les différents problèmes de recherche d'une solution

Nous pouvons distinguer trois types de problèmes de recherche d'une solution :

Algorithm 1 Recherche en profondeur

FONCTION ExplorationProf ($E_i, DejaVu, N$) : BOOLEEN
VAR res : BOOLEEN
  SI $E_i \in F$ **ALORS**
    $res \leftarrow VRAI$
  SINON
    SI $N = 0$ **ALORS**
      $res \leftarrow FAUX$
    SINON
      POUR TOUT $(A_j, E_j) \in T(E_i)$ ET NON ($E_j \in DejaVu$) **FAIRE**
        SI $ExplorationProf(E_j, DejaVu \cup E_j, N-1) = VRAI$ **ALORS**
          $Afficher A_j, E_j$
          $res \leftarrow VRAI$
        FIN SI
      FIN POUR
    FIN SI
  FIN SI
RETOURNER res

Algorithm 2 Recherche en largeur

FONCTION ExplorationLarg(E_0) : LISTE
VAR F : FILE
  $F \leftarrow fileVide$
  $L \leftarrow listeVide$
  $Ajouter(F, E_0)$
  TANTQUE NON vide(F) **FAIRE**
    inserer(L,premier(F))
    SI NON $premier(F) \in F$ **ALORS**
      POUR TOUT $(A_j, E_j) \in T(premier(F))$ **FAIRE**
        $ajouter(F, E_j)$
      FIN POUR
      supprimer(F)
    SINON
      $F \leftarrow fileVide$
    FIN SI
  FIN TANTQUE
RETOURNER L

1. Le problème de la recherche d'une *solution quelconque*. On peut utiliser les algorithmes de recherche en profondeur d'abord pour effectuer ce type de recherche.
2. Le problème de la recherche de *toutes les solutions*. Nous pouvons dans ce cas là construire l'arbre en largeur d'abord en introduisant une stratégie qui n'explore pas les branches ne menant pas à une bonne solution (à condition de les détecter le plus rapidement possible).
3. Le problème de la recherche de la *meilleure solution* selon un critère donné. Souvent ce critère est formalisé par un coût qui sera associé à l'ensemble des *actions* formalisant le problème. Nous pouvons également explorer l'arbre en largeur avec une stratégie qui assure la non exploration de certaines branches.

1.4 Introduction d'un coût

On suppose qu'on associe à chaque action A_i un coût (réel positif). Le coût associé à une suite d'états est la somme des coûts associés aux actions exécutées.

Une solution est *optimale* s'il n'existe pas aucune solution de coût strictement inférieur. Une méthode est dite *admissible* si chaque fois qu'il existe une solution optimale, elle est trouvée en un temps fini. Dans la suite nous adoptons les notations suivantes :

$k(E_i, E_j)$ Le coût de l'action la moins chère pour aller de E_i à E_j si elle existe.

$k^*(E_i, E_j)$ Le coût de la séquence d'actions la moins chère pour aller de E_i à E_j.

$g^*(E_i)$ $g^*(E_i) = k^*(E_0, E_i)$

$h^*(E_i)$ $h^*(E_i) = \min k^*(E_i, E_j)$ avec $E_j \in F$, autrement dit $h^*(E_i)$ représente le coût minimal pour atteindre l'objectif si ce chemin existe.

$f^*(E_i)$ $f^*(E_i) = g^*(E_i) + h^*(E_i)$ Autrement dit $f^*(E_i)$ représente le coût minimal d'une solution passant par E_i si elle existe.

Nous utilisons dans la suite la notion de successeur d'états qui est définie par : $Succ(E_i) = \{E_j : (A_j, E_j) \in T(E_i)\}$

1.5 Recherche guidée

Nous allons montrer comment introduire une *heuristique* qui permet de guider le parcours en largeur en recherchant une solution dans l'arbre de recherche. Cette heuristique tente de diriger le parcours en coupant certaines branches et en allant en profondeur dans d'autres. Une heuristique est une mesure associée à un état donné qu'on notera $h(E_i)$.

1.5.1 Propriétés d'une heuristique

Une heuristique $h(E_i)$ est dite *coïncidente* si elle reconnaît les états de l'objectif : $\forall E_j \in F, h(E_j) = 0$.

Elle est *presque parfaite* si elle donne sans erreur la branche à explorer : $h(E_j) < h(E_i)$ où E_j est un état qui suit E_i.

Elle est dite *consistante* si pour toute paires d'états nous avons : $h(E_i) - h(E_j) \leq k^*(E_i, E_j)$.

Elle est *monotone* si $\forall(E_i, E_{ij}), (A_{ij}, E_{ij}) \in T(E_i), h(E_i) - h(E_{ij}) \leq k(E_i, E_{ij})$.

Une heuristique est dite minorante si elle sous-estime systématiquement le coût du chemin restant à parcourir : $h(E_i) \leq h^*(E_i)$

Théorème 1.5.1 *h est monotone ssi h est consistante*

Supposons que h est consistante , dons pour toute paire d'états nous avons $h(E_i) - h(E_j) \leq k^*(E_i, E_j)$ et plus particulièrement si $E_j \in Succ(E_i)$ alors $h(E_i) - h(E_j) \leq k^*(E_i, E_j)$ et par définition nous avons $k^*(E_i, E_j) \leq k(E_i, E_j)$.

Maintenant supposons que la stratégie est monotone Soient (E_0, E_n) une paire d'états pour laquelle il y a un chemin optimal $E_1, ..E_{n-1}, E_n$, nous avons $\forall i(h(E_{i-1}) - h(Ei) \leq k(E_{i-1}, Ei))$ En sommant nous obtenons : $h(E_0) - h(E_n) \leq \sum_{i=1}^{n} k(E_{i-1}, Ei)$ et donc : $h(E_0) - h(E_n) \leq k^*(E_0, E_n)$

Théorème 1.5.2 *Si h est monotone et coïncidente, alors h est minorante.*

Par hypothèse de monotonie et sur un chemin optimal, on obtient que $h(E_0) - h(E_n) \leq k^*(E_0, E_n)$ et puisque le chemin est optimal alors $h(E_0) - h(E_n) \leq h^*(E_0)$ et puisque l'heuristique est coïncidente alors $h(E_0) \leq h^*(E_0)$

1.5.2 Présentation de l'algorithme A*

L'algorithme A* effectue un parcours en largeur guidé par une heuristique définie préalablement. Deux files sont utilisées pour stocker les états visités et à visiter ; la file *Actif* contient les états à visiter tandis que la file *Inactif* contient les états déjà visités. Parmi tous les successeurs d'un état donné, sont ajoutés à la file *Actif*, les états non visités ou les états déjà visités mais dont le coût du passage actuel est moins élevé qu'avant. Une mesure $f(e) = g(e) + h(e)$ est associée à chaque état ; cette mesure combine l'heuristique $h(e)$ avec le coût actuel du passage par l'état $g(e)$. La file *Actif* est triée par ordre de f croissant. Autrement dit, l'état dont le coût estimé est le moins élevé sera sélectionné.

Théorème 1.5.3 *Si tout chemin de longueur infinie a un coût infini, si sur un chemin optimal, la valeur de l'heuristique est bornée et si chaque état a un nombre fini de successeurs alors l'algorithme A* termine.*

Théorème 1.5.4 *Si les conditions de terminaisons sont réalisées et si l'heuristique est minorante alors l'algorithme A* est admissible.*

Selon ce théorème, toute heuristique qui sous-estime la réalité est une heuristique qui permet de trouver le chemin optimal. Nous disons que h2 est *plus informée* que h1 si toutes les deux sont minorantes et si $h2(e) > h1(e), \forall e \in S$. En effet h2 permet de trouver le chemin optimal plus rapidement parcequ'elle est plus proche de $h^*(e)$.

Algorithm 3 l'algorithme A*

PROCEDURE A*()
VAR e,e' : ETAT,
Actif, Inactif : FILE
$Actif \leftarrow [e_0]$
$Inactif \leftarrow []$
$g(e_0) \leftarrow 0$
$E \leftarrow e_0$
TANTQUE non fileVide(Actif) ET non $e \in F$ **FAIRE**
supprimer(Actif)
inserer(Inactif, e)
POUR TOUT $e' \in Succ(e)$ **FAIRE**
SI non ($e' \in Actif$ ET $e' \in Inactif$) OU g(e') > g(e)+k(e,e') **ALORS**
$g(e') \leftarrow g(e) + k(e, e')$
$f(e') \leftarrow g(e') + h(e')$
$pere(e') \leftarrow e$
ajouterTrier(Actif,e')
FIN SI
FIN POUR
SI non(fileVide(Actif)) **ALORS**
$e \leftarrow premier(Actif)$
FIN SI
FIN TANTQUE

1.6 Implémentation en Prolog

1.6.1 Le parcours en Profondeur d'abord

Nous présentons un programme prolog qui effectue la recherche en profondeur d'abord. Ce programme est basé sur la récursivité. Le prédicat solve permet de retrouver l'état final à partir d'un état donné. Les états déjà visités sont stockés dans une liste. Le prédicat *move* associe des actions à des états de départs. Le prédicat *update* change l'état en appliquant une action. Finalement le prédicat *legal* teste si un état existe ou non. Le programme en prolog est donné par :

```
solve(State,History,[]) :-etat_final(State).
solve(State,History,[Move|Moves]) :-
move(State,Move), update(State,Move,State1), legal(State1), not member(State1,History),
solve(State1,[State1|History],Moves).
test(Moves) :-etat_initial(State), solve(State,[State],Moves).
```

1.6.2 Le parcours en Largeur d'abord

Ce programme est composé de trois prédicats :

solveB effectue l'exploration en largeur et s'arrête dès qu'il trouve une solution ;

update-set met à jour la liste des nœuds à explorer en remplaçant le premier élément par ses fils (ces fils sont insérés à la fin) ;

insertF insert un élément à la fin d'un liste donnée si jamais cet élément correspond à un nœud non visité auparavant.

```
solveB([state(State,Path)|Reste],_,Moves) :-
etat_final(State), reverse(Path,Moves).
solveB([state(State,Path)|Reste],History,Finalpath) :-
not(etat_final(State)), findall(M,move(State,M),Moves),
update-set(Moves,State,Path,History,Reste,Reste1), solveB(Reste1,[State|History],Finalpath).
update-set([M|Ms],State,Path,History,R,R1) :-
update(State,M,State1), insertF(state(State1,[M|Path]),History,R,R2), update-set(Ms,State,Path,History,R2,R1).
update-set([],S,P,H,R,R).
insertF(state(State,_),History,[],[]) :-
member(State,History), !. insertF(X,History,[],[X]).
insertF(X,History,[T|Reste],[T|Reste1]) :-
insertF(X,History,Reste,Reste1).
test(Moves) :-etat_initial(State), solveB([state(State,[])],[State],Moves).
```

1.6.3 La modélisation du problème du taquin

Nous modélisons le problème du taquin en utilisant quatre prédicats qui sont :

etat_initial décrit l'état initial du taquin en le lisant de haut en bas et de gauche à droite ;

etat_final décrit l'état final du taquin ;

move décrit un action possible appliquée à un état donné ;

update décrit l'état résultant de l'application d'une action à un état donné.

```
etat_initial([1,b,2,3]).
etat_final([2,1,3,b]).
move([b,X,Y,Z],droite) :-X≡b,Y≡b,Z≡b.
move([b,X,Y,Z],bas) :-X≡b,Y≡b,Z≡b.
move([X,b,Y,Z],gauche) :-X≡b,Y≡b,Z≡b.
move([X,b,Y,Z],bas) :-X≡b,Y≡b,Z≡b.
move([X,Y,b,Z],droite) :-X≡b,Y≡b,Z≡b.
move([X,Y,b,Z],haut) :-X≡b,Y≡b,Z≡b.
move([X,Y,Z,b],gauche) :-X≡b,Y≡b,Z≡b.
move([X,Y,Z,b],haut) :-X≡b,Y≡b,Z≡b.
update([b,X,Y,Z],bas,[Y,X,b,Z]).
update([b,X,Y,Z],droite,[X,b,Y,Z]).
update([X,b,Y,Z],bas,[X,Z,Y,b]).
update([X,b,Y,Z],gauche,[b,X,Y,Z]).
update([X,Y,b,Z],haut,[b,Y,X,Z]).
```

```
update([X,Y,b,Z],droite,[X,Y,Z,b]).
update([X,Y,Z,b],haut,[X,b,Z,Y]).
update([X,Y,Z,b],gauche,[X,Y,b,Z]).
```