I Résolution de problèmes par recherche dans un espace d'états

1 Introduction

Enoncé informel $\xrightarrow[\text{linéaire}]{\text{analyse}}$ représentation formelle du problème $\xrightarrow[\text{de résolution du problème}]{\text{algorithme}}$ solution

Exemples : le problème de l'échiquier écorné et des dominaux / le problème des 4 cavaliers

2 Représentation formelle du problème

- énoncé type : – soient un état initial, un (des) état(s) terminal(aux), un ensemble d'opérateurs de changement d'état
  – trouver une séquence d'opérateurs permettant de passer de l'état initial à un état terminal.
- représentation des états et des opérateurs (modélisation)
  - représentation des états = choix de structure Prolog pour représenter les différents états
  - spécification des opérateurs =
    + OP : < Précondition, Postcondition, coût >

condition d'appel de OP sur un état ↳ effets de l'appel de OP dans un prédicat

    + fonction successeur s appliquée à chaque état
      $s(e) = \{e_1, e_2, \dots, e_m\}$ ensemble des états que l'on peut atteindre à partir de e par application d'un opérateur (pas forcément le même) applicable à e.
      (Prolog = succ(S_1, S_2, C), S_2 vrai si il est le résultat de l'appel d'un opérateur de coût C à S_1).
  - exemples : + Jeu de Taquin (4 opérateurs Bas, Haut, Gauche, Droite)
    + Le problème des 8 reines (formulation à états complets et formulation incrémentale)

3 Résolution du problème

On explore le graphe d'états : recherche d'un chemin de l'état initial à un état terminal.

- graphe d'états : noeuds ⟷ états et arcs ⟷ opérateurs
- principe des algorithmes de recherche :
  - par expansions successives de l'espace de recherche
  - un pas d'expansion :
    + choisir l'extrémité n d'un chemin de l'espace de recherche courant
    + étendre ce chemin en prenant en compte tous les successeurs du noeud n (développement de n)
- choix du noeud à développer :
  - algorithme non informés : critères de choix non liés au problème spécifique considéré
  - algorithme informés (heuristiques) : critères de choix liés au problème spécifique considéré
- structures de données représentant l'espace de recherche : graphe/arbre de recherche.

4 Propriétés par rapport auxquelles les algorithmes sont évalués :

- complétude si il existe, au problème spécifié, une solution alors l'algorithme termine et renvoie une solution du problème.
- complexité de l'algorithme en temps/en mémoire directement liée à la taille de l'espace de recherche
- optimalité si il existe une solution au problème spécifié en entrée de l'algorithme, alors l'algorithme s'arrête et renvoie une solution de coût minimal (optimal)
  → A*

II Algorithmes non informés d'exploration de graphes d'états (recherche d'une solut

① Algorithmes de recherche en largeur d'abord

Principe : un nœud de profondeur p n'est développé que si tous les nœuds de profondeur $p-1$ ont été développés.

Notations : $p(n)$ profondeur de n ; e_0 état initial ; p longueur du chemin le plus court ; b facteur de branchement.

Complexité : (pire des cas) • temps : $\sum_{k=0}^{p} b^k = \frac{b^{p+1}-1}{b-1}$ $O(b^p)$

• espace : $\sum_{k=0}^{p+1} b^k$ $O(b^{p+1})$

② Algorithme de coût uniforme

Principe : adaptation de l'algorithme en largeur d'abord pour obtenir un algorithme optimal dans le cas général (les opérateurs de changement d'état ont un coût uniforme)

Notations : $k(n,s)$ coût de l'opérateur pour aller de n à s ; $g(n)$ coût du chemin de e_0 à n.

③ Algorithme en profondeur d'abord

Principe = privilégier le développement du nœud le plus profond dans l'arbre de recherche.

Remarque = risque de non terminaison (on peut retomber sur un état déjà vu → boucle / branche infinie)
⇒ on peut ajouter une profondeur maximum d'exploration d.

Complexité :

pire des cas : • temps : $\sum_{k=0}^{d} b^k$ $O(b^d)$ — meilleur des cas : • temps : p $O(p)$

• espace : d (ou $b \times d$) $O(d)$ — • espace : p (ou $b \times p$) $O(p)$

④ Algorithme de recherche à profondeur incrémentale (Iterative Deepening)

Principe : combine profondeur d'abord et largeur d'abord
on applique profondeur d'abord pour d variant de 0 à m (profondeur maximal)

Remarque : beaucoup de redondance d'une étape à l'autre mais ce qui coûte le plus cher, c'est le développement du dernier niveau d (b^d plus cher que $1+b+b^2+\dots+b^{d-1}$)
avantage de largeur d'abord → optimal
avantage de profondeur d'abord → complexité (espace et temps)

Complexité :

• temps : $\sum_{i=0}^{p} (p-i+1) b^i$ $O(b^p)$

• espace : $O(p)$

Largeur d'abord
(Coût uniforme)

```
Ouvert <- {e0}
Ferme <- vide
Succes <- faux
p(e0) <- 0 (g(e0) <- 0)
Tant que Ouvert <> vide et Succes = faux
Soit n le nœud de Ouvert tq p(n) (g(n)) minimal
Si terminal(n) alors   Succes = vrai
                       Return construit_chemin(n)
sinon     Supprimer n de Ouvert
          Ajouter n à Ferme
          Pour chaque successeurs s de n :
             Ajouter un nœud ns à Ouvert
             pere(s) <- n
             p(ns) <- p(n) + 1 (g(ns) <-g(n) + k(n,s))
Si Ouvert = vide alors ECHEC
```

Profondeur d'abord

```
Fonction RechercheB(n,d)
1.Si terminal(n) alors return [ ]
(2.Si impasse(n) alors return ECHEC)
3.Si d = 0 alors ECHEC
4.Soit Lsucc = succ(n)
5.Si Lsucc = [ ] alors return ECHEC
6.  s <- premier(Lsucc)
    Lsucc <- reste(Lsucc)
7.Chemin <- RechercheB(s,d - 1)
8.Si Chemin = ECHEC alors aller en 5
9.Recherche <- [n | Chemin]
```

Profondeur incrémentale

```
Fonction Iterative_Deepening(n,m)
Pour d allant de 0 à m
   Chemin <- RechercheB(n,d)
   Si Chemin <> ECHEC alors return Chemin
Return ECHEC
```

Algorithme informés d'exploration d'espace d'État (heuristiques)

introduction

Principe : ils utilisent une fonction d'évaluation pour guider le choix du nœud à développer.

Fonction d'évaluation : fonction numérique s'appliquant aux nœuds de l'espace de recherche, qui traduit sous forme numérique des critères de nature heuristique, spécifiques du problème à résoudre. ($f(n)$ estimation du caractère prometteur du nœud n)

Exemple : le problème des 6 flèches.

② Fonction d'évaluation pour un même problème

fonction d'évaluation de type "gradient".

$f(n) = h(n) + g(n)$ $\quad$ $f(n)$ = estimation du coût d'un chemin, de e_0 à un état terminal, passant par n.

$g(n)$ = coût du chemin de e_0 à n.

$h(n)$ = estimation du coût du chemin qu'il reste à parcourir.

③ Algorithme du meilleur d'abord (avec graphe de recherche)

entrées : • état initial, état(s) terminal(aux), succ.

• fonction d'évaluation.

④ L'algorithme A*

Algorithme du meilleur d'abord avec $f = g + h$, heuristique h minorante de type gradient.

Notations : $\forall n,\ f^*(n) = g^*(n) + h^*(n)$

$f^*(n)$ = coût du meilleur chemin, de e_0 à un état terminal, passant par n.

$g^*(n)$ = coût du meilleur chemin de e_0 à n.

$h^*(n)$ = coût du meilleur chemin de n à un état terminal.

Definition : h minorante $\Leftrightarrow \forall n,\ h(n) \leq h^*(n)$

Théorème 1 : si une solution existe, alors A* termine et le nœud terminal responsable de l'arrêt d'A* est l'extrémité d'un chemin solution de coût minimal.

Lemme du théorème 1 : à toute étape de A*, il existe dans ouvert un nœud n' appartenant à un chemin optimal de e_0 à un état terminal tel que $f(n') \leq f^*(e_0)$.

Définition : h monotone $\Leftrightarrow \forall n\ \forall s = succ(n)\quad h(n) \leq h(s) + k(n,s)$.

Théorème 2 : Si h monotone, $g(n) = g^*(n)$.

⑤ Preuve d'admissibilité d'A* (par l'absurde)

Hypothèse : • le chemin découvert au moment de l'arrêt d'A* n'est pas un chemin solution optimal de e_0 à un état terminal.

• Soit t, le nœud terminal responsable de l'arrêt

• $h(t) = h^*(t) = 0$

hypothèse $\rightarrow g(t) > g^*(t)$ $\quad \Big| \Rightarrow f(t) > f^*(t) = f^*(e_0)$

• à l'étape juste avant la terminaison, il existe n' nœud de Ouvert tel que $f(n') \leq f^*(e_0) < f(t)$

$\Rightarrow$ contradiction avec le choix de t.

⑥ Analyse d'A*

- A* admissible (trouve toujours la meilleure solution)
- A* optimalement efficace parmi les algorithmes admissible.
- complexité d'A* = nombre de nœud développés croît de façon exponentielle par rapport à la longueur du chemin solution.

⑦ Amélioration d'A* (IMA*, SMA*)

SMA* : Simplified Memory-Bounded A*

Principe : on oublie les noeuds les moins prometteurs et on garde la valeur de la meilleure estimation du noeud associé à la racine de chaque sous-arbre effacé.

SMA* est complet et admissible (trouve une solution et c'est la meilleure, si assez de mémoire)

IV Jeux avec adversaires : modélisation et algorithmes

① Modélisation par arbre de jeu

Modélisation du problème :
- configuration du jeu initiale
- coups légaux
- situation d'arrêt (gagné, perdu, égalité)

Modélisation de la résolution du problème :
- J1 commence et J2 ensuite
- choix de AMI / ENNEMI
- disymétrie entre les noeuds du graphe : AMI (noeud OU)
  ENNEMI (noeud ET)

② Recherche d'une stratégie gagnante

Definition : séquence de coups joués par AMI qui le mène à une situation gagnante, indépendemment des coups joués par ENNEMI

Formalisation : un arbre de jeu est gagnant pour AMI :
- si c'est une feuille gagnante
- si c'est un noeud OU, si au moins un de ses sous-arbre est gagnant
- sinon si c'est un noeud ET, si tous ses sous-arbres sont gagnants.

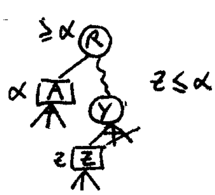
③ Recherche "heuristique" du meilleur coup à jouer (MinMax)

Principe : combine heuristique (estime le caractère prometteur pour AMI des situations de jeu) et anticipation (developpement de l'arbre de jeu jusqu'à une certaine profondeur) pour choisir le meilleur coup à jouer pour AMI.
La fonction $f(e_i)$ est d'autant plus grande que e_i est favorable pour AMI.

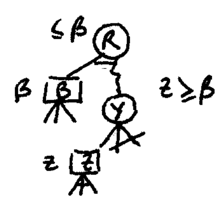
④ L'Algorithme Alpha-Beta

Principe : même données et résultats que MinMax mais ne nécessite pas le développement de tout l'arbre.

Coupe α



Coupe β



○ noeud en cours d'évaluation.
□ noeud dont l'évaluation est terminée.

Coût du MinMax : $O(b^p)$

coût du $\alpha\beta$: cas le pire $O(b^p)$ cas le meilleur $O(b^{p/2})$ cas moyen $O(b^{3p/4})$

```
Ouvert <- {e0}
Ferme <- vide
Succes <- faux
Tant que Ouvert <> vide et Succes = faux
Soit n le nœud de Ouvert tq  f(n) minimal
Si terminal(n) alors    Succes = vrai
                        Return construit_chemin(n)
sinon    Supprimer n de Ouvert
         Ajouter n à Ferme
         Pour chaque successeurs s de n :
           Si not(appartient(s,Ouvert U Ferme))
           alors ajouter s à Ouvert
                 pere(s) <- n
                 g(s) <- g(n) + k(n,s)
           sinon mise à jour de Ouvert selon f
```

MinMax

```
MaxMin(N Nœud OU) :
   Si feuille(N) alors result <- f(N)
   sinon soit N1,…,Nk les succ de N
         result <- max{MinMax(N1),…,MinMax(Nk)}

MinMax(N Nœud ET) :
   Si feuille(N) alors result <- f(N)
   sinon soit N1,…,Nk les succ de N
         result <- min{MaxMin(N1),…,MaxMin(Nk)}
```

Alpha-Beta

```
MaxMin(N,A,B)
   Si feuille(N) alors return h(n)
   sinon pour chaque succ s de N
         A <- max{A,MinMax(S,A,B)}
         Si A >= B alors return B
   return A

MinMax(N,A,B)
   Si feuille(N) alors return h(n)
   sinon pour chaque succ s de N
         B <- min{B,MaxMin(S,A,B)}
         Si A >= B alors return A
   return B
```