IA Symbolique – Travaux Pratiques

Algorithme IDA*

Application au jeu « La rumba des chiffres »

Introduction

Il s'agit pendant ces 4 séances de TP de 2h, qui devront être complétées par du travail personnel) d'implémenter progressivement un algorithme **IDA*** pour résoudre des problèmes du jeu **La rumba des chiffres**.

Dans un premier temps on implémentera l'algorithme **Profondeur d'abord** que l'on modifiera ensuite en **Profondeur d'abord bornée** par une valeur passée en paramètre.

Le travail se fera en binôme et sera évalué par un QCM individuel et une démonstration (en binôme) de votre travail devant l'enseignant. Nous vous donnons ci-après quelques indications ainsi que le pseudo-code des différents algorithmes (profondeur d'abord, profondeur d'abord bornée, IDA*) qui devra être adapté pour votre implémentation.

Le langage conseillé pour votre implémentation est **Python** mais vous pouvez, **sous réserve de l'acceptation de votre professeur de TP**, utiliser un autre langage de programmation.

Etape 1 : implémentation de l'algorithme « Profondeur d'abord »

Pseudo-code à adapter (c'est simplement un guide pour la compréhension)

```
Fonction ProfondeurDAbord {retourne un booléen vrai et un but  si on atteint un but, faux et l'état initial sinon}
        Arguments
                depart {état initial}
        Fonctions utilisées
                testEtatBut(etat) ; {prédicat caractérisant les buts}
                filsEtat(etat) ; {retourne la liste des fils d'un état}
                pop(liste) ; {enlève le premier élément de liste et le renvoie}
                ajouterTete(e, liste) ; {ajoute l'élément e en tête de liste}
        Variables locales
                enAttente ; {la liste des états non encore explorés}
                vus ; {la liste des états déjà explorés}
                trouve ; {booléen vrai si on atteint un but, faux sinon}
                prochain ; {le prochain état à explorer}
                e ; {état}
Début
        enAttente  <- {départ} ;
        vus <- ⊘ ;
        trouve <- faux ;
        Tant que enAttente  ≠ ⊘ et non trouve Faire
                prochain <- pop(enAttente ) ; {on récupère la tête de la liste d'attente, gestion FIFO}
                vus <- vus ∪ {prochain} ; {on rajoute prochain à la liste vus}
                Si testEtatBut(prochain) Alors
                        trouve <- vrai
                        retourner(vrai, prochain) ;
                Sinon
                        Pour Tout e ∈ filsEtat(prochain) Faire
                                Si e ∉ vus Alors enAttente  <- ajouterTete(e, enAttente ) ; FinSi ;
                                {pour couper les boucles, la liste enAttente contient seulement les e non déjà vus}
                                {attention à l'ordre des fils de prochain dans enAttente }
                                {pour un parcours en profondeur le premier fils doit être devant}
                        Fin Pour Tout ;
                FinSi ;
        Fin Tant Que ;
        Si non trouve Alors retourner(faux, depart) ; FinSi ;
Fin.
```

Etape 2 : implémentation de l'algorithme « Profondeur d'abord bornée »

Modifiez l'algorithme précédent pour limiter la profondeur de la recherche par un entier passé en paramètre. L'algorithme doit ainsi travailler en profondeur d'abord en ne descendant pas plus loin que la profondeur passée en paramètre. Par convention l'état initial est à profondeur 0.

Etape 3 : Implémentation de l'algorithme IDA* (Iterative Deepening A*)

Rappels sur l'algorithme IDA*

Les algorithmes de type A* et IDA* utilisent, pour estimer l'état courant e, une fonction heuristique (appelée ici fEtat) qui est de la forme f(e) = g(e) + h(e). Dans cette fonction, g(e) représente le coût du chemin déjà parcouru depuis l'état initial jusqu'à e et h(e) représente une estimation minorante du coût du chemin restant à parcourir depuis e jusqu'à un état-but. On calcule g(e) en ajoutant le coût du chemin déjà parcouru depuis l'état initial jusqu'au père de e, et le coût de la transition entre e et son père. La fonction h étant minorante, on démontre que l'algorithme est admissible, c'est-à-dire qu'il retourne la solution de coût optimal.

Description du fonctionnement de IDA*

L'algorithme IDA* [Korf85] est la combinaison de la recherche itérative en profondeur d'abord (DFID, Deep First Iterative Deepening, algorithme de recherche non informée) et de la recherche heuristique A*. L'idée est non pas de limiter la profondeur mais le coût de la recherche. IDA* attribue à chaque état e de l'arbre un coût calculé comme étant f(e) = g(e) + h(e), où g(e) est le coût du chemin parcouru de l'état initial à l'état e et h(e) est une estimation minorante du coût du chemin à parcourir de l'état e au but.

IDA* effectue une série d'itérations jusqu'à trouver une solution ou avoir développé tout l'espace de recherche sans succès. A chaque itération il mène une recherche en profondeur d'abord à partir de l'état initial jusqu'à ce que le coût de l'état feuille dépasse un seuil fixé au début de l'itération. Au départ (à la première itération) ce seuil correspond au coût de l'état initial. En cas d'échec de l'itération courante, il est augmenté et remplacé par le minimum de tous les coûts ayant dépassé le seuil à l'itération précédente. IDA* est optimal sur la classe des algorithmes de recherche meilleur-d'abord avec heuristiques minorantes monotones.

IDA* utilise une fonction d'évaluation qui a les caractéristiques exigées par A*. Il ne suit pas le principe « développer d'abord un état de plus petite évaluation » mais « développer partiellement l'état le plus profond d'abord, pourvu que son évaluation ne dépasse pas le seuil courant S ».

Initialement, le seuil S vaut f(<état_initial>) = g(<état_initial>) + h(<état_initial>) soit h(<état_initial> puisque g(<état_initial> = 0. IDA* produit un premier fils f1 de l'état initial, puis un premier fils de f1 et ainsi de suite tant que l'évaluation de l'état le plus profond est inférieure ou égale au seuil courant S. Si cette évaluation dépasse le seuil S, IDA* coupe la recherche de cette branche, remonte au père et produit un nouveau fils f2 s'il n'y en a plus, il remonte au grand-père, etc. S'il ne reste plus de choix, IDA* lance une nouvelle exploration en profondeur, depuis l'état initial, mais avec comme nouveau seuil S le minimum des évaluations ayant dépassé le seuil précédent. IDA* s'arrête avec succés lorsque l'état à développer est un état-but ou sur un échec s'il a exploré tout l'espace d'états.

En résumé :

1. À chaque itération, on réalise une exploration en profondeur d'abord.
2. Chaque état e est évalué par une fonction de la forme f (e)= g(e)+h(e) comme dans A*.
3. Dans cette exploration, tous les états dont l'évaluation dépasse un certain seuil sont éliminés de l'exploration.
4. Ce seuil est initialement (à la première itération) égal à l'évaluation de l'état initial.
5. A chaque nouvelle itération, le seuil est mis à jour avec le minimum des évaluations qui dépassaient le seuil dans l'itération précédente.

[Korf85] R.E. Korf Depth-First Iterative-Deepening : An optimal Admissible Tree Search. Artificial Intelligence, 27, pp. 97-109, 1985.

Pseudo-code à adapter (c'est simplement un guide pour la compréhension)

```
Fonction IDA* {retourne un état-solution ou échec}
        Arguments
                départ ; {l'état initial}
                testEtatBut ; {prédicat caractérisant les buts}
                filsEtat ; {retourne la liste des fils d'un état}
                fEtat ; {estime la promesse d'un état, i.e. l'intérêt d'exploiter l'état, voir le rappel ci-après}
        Variables locales à la fonction IDA*
                seuil ; {valeur à ne pas dépasser dans une itération}
                terminé ; {booléen}
                solution ; {état}
Début
        solution <- nul ;
        seuil <- fEtat(départ) ;
        Répéter terminé <- ProfondeurBornée(seuil) ; Jusqu'à terminé ; Fin Répéter ;
        Si solution ≠ nul Alors retourner(solution) ; Sinon retourner(échec) ; FinSi ;
Fin


Fonction ProfondeurBornée {retourne un booléen et modifie les variables globales seuil et solution}
        Arguments
                s {valeur de seuil à ne pas dépasser pendant la recherche en profondeur}
        Fonctions et variables globales utilisées
                testEtatBut ; {prédicat caractérisant les buts}
                filsEtat ; {retourne la liste des fils d'un état}
                fEtat ; {estime la promesse d'un état, i.e. l'intérêt d'exploiter l'état, voir le rappel ci-après}
                seuil ; {valeur à ne pas dépasser pendant la recherche}
                solution ; {état}
        Variables locales
                nSeuil ; {nouveau seuil pour la prochaine itération}
                vus ; {la liste des états déjà explorés avec leur meilleure valeur}
                enAttente ; {la liste des états non encore explorés avec leur valeur}
                prochain ; {le prochain état à explorer}
                e ; {état}
Début
        nSeuil <- +∞ ;
        vus <- ∅ ;
        enAttente <- {départ} ;
        Tant que enAttente ≠ ∅ Faire
                prochain <- pop(enAttente ) ;
                vus <- {prochain} ∪ vus ;
                Si testEtatBut(prochain) Alors
                        solution <- prochain ;
                        retourner(vrai) ;
                Sinon
                        Pour Tout e ∈ filsEtat(prochain) Faire
                                Si (fEtat(e) ≤ s) et (e ∉ vus) Alors enAttente <- ajouterTête(e, enAttente ) ;
                                {attention à l'ordre des fils de prochain dans enAttente pour un parcours en profondeur}
                                Sinon nSeuil <- min(nSeuil, fEtat(e)) ; FinSi ;
                        Fin Pour Tout ;
                FinSi ;
        Fin Tant Que ;
        Si nSeuil = +∞ Alors
                retourner(vrai) ;
        Sinon
                {on met à jour le seuil avant de relancer une itération avec ce nouveau seuil dans la fonction IDA*}
                seuil <- nSeuil ;
                retourner(faux) ;
        FinSi ;
Fin.
```

Etape 4 : Application au jeu « La rumba des chiffres »

Dans ce jeu, le joueur dispose d'un support avec quatre tiges où sont enfilés 9 cubes qui se distinguent par leur couleur (jaune, bleu ou rouge) et un chiffre (1, 2 ou 3). Le joueur doit reconstituer une nouvelle configuration en déplaçant ses cubes un à un d'une tige à l'autre, sans en empiler plus de 3 sur une même tige. Seuls les cubes qui sont au sommet peuvent être déplacés.

Exemple : situation initiale

1J	1B	1R	
2J	2B	2R	
3J	3B	3R	
tige 1	tige 2	tige 3	tige 4

But :

1R	2R	1J	
2J	1B	2B	
3J	3B	3R	
tige 1	tige 2	tige 3	tige 4

Pour résoudre le problème ci-dessus on pourra effectuer les déplacements suivants : (3, 4) prendre un cube de la tige 3 pour l'amener vers la tige 4, ce qui conduira à la situation décrite par la figure ci-après à gauche :

1J	1B		
2J	2B	2R	
3J	3B	3R	1R
tige 1	tige 2	tige 3	tige 4

1J			2R
2J		2B	1B
3J	3B	3R	1R
tige 1	tige 2	tige 3	tige 4

puis les déplacements (2, 4) ; (3, 4) ; (2, 3) amèneront à la situation décrite par la figure ci-dessus à droite puis (4, 3) ; (4, 2) ; (3, 2) ; (1, 3) ; (4, 1) permettront d'arriver au but.

Travail à réaliser

Vous devez écrire un programme pour résoudre des problèmes avec 4 tiges de capacité 3 dans lequel tous les déplacements ont un coût uniforme de 1 mais qui pourront par la suite avoir un coût non uniforme. **Il devra renvoyer la séquence des actions à réaliser pour résoudre le problème posé (plan-solution) ou un plan vide en cas d'échec.** Pour simplifier, **au lieu de couleurs et de nombres de 1 à 3, vous considérerez simplement des cubes numérotés de 1 à 9**. Ce programme utilisera la fonction **rechercheIdaEtoile** qui implémentera l'algorithme IDA* que vous avez étudié durant les cours/TD et dont le pseudo-code vous a été donné ci-avant pour vous guider dans votre réflexion. Voici quelques suggestions qui pourront vous être utiles (mais vous êtes entièrement libres de vos choix pour l'implémentation).

La fonction **rechercheIdaEtoile** devra renvoyer le plan-solution (la séquence de déplacements) et son coût optimal (ou une séquence vide en cas d'échec). Votre implémentation devra aussi afficher, en fin de recherche, le nombre d'itérations de IDA* et, pour chaque itération, le nombre de nœuds créés et développés. Pour implémenter la fonction **rechercheIdaEtoile**, voici quelques fonctions que vous pourriez écrire (encore une fois, vous êtes cependant entièrement libres de vos choix pour l'implémentation) :

Réfléchissez à la représentation interne que vous allez choisir pour représenter un état de votre jeu. Vous êtes entièrement libres de vos choix mais ils devront être justifiés. On pourra par exemple utiliser un tuple, un entier, un tableau, une matrice, une chaîne de caractères… les possibilités sont nombreuses et il existe des fonctions d'accès spécifiques pour les listes, les vecteurs, les chaînes de caractères, etc. Pensez qu'il pourrait aussi être utile de stocker le nombre d'éléments présents sur chaque pique afin d'éviter d'avoir à le recalculer à chaque tentative de déplacement.

Pour visualiser les états, vous pouvez écrire une fonction d'affichage **afficherEtat**. Elle pourrait par exemple afficher un état donné e ainsi :

```
    4 7
2   5 8
3 1 6 9
--------
```

Vous pouvez écrire la fonction **trouverDestinations** telle que **trouverDestinations (e,pi)** renvoie la liste des piques destinataires possibles depuis la tige **pi** (pique initiale) de l'état **e**. S'il n'y en a pas, la fonction renvoie la liste vide. Par exemple, **trouverDestinations** appliquée à **e** (affichage précédent) depuis la tige 3 donnera la liste (1, 2) puisque seules ces deux piques ne sont pas complètes.

Vous pouvez écrire la fonction **deplacer** telle que **deplacer(e,p1,p2)** renvoie l'état obtenu en déplaçant, depuis l'état **e** le premier élément de la tige **p1** vers le sommet de la tige **p2**. Vous n'avez pas besoin de gérer spécifiquement les déplacements impossibles puisque **deplacer** ne sera utilisée que pour générer les déplacements possibles renvoyés par la fonction **trouverDestinations**.

Vous pouvez écrire la fonction **estBut**, telle que **estBut(e)** renvoie vrai si un état **e** donné est un but (c'est-à-dire si la configuration des tiges est la configuration recherchée). Cette fonction peut être définie à partir d'une fonction **egal** telle que **egal(e,but)** renvoie vrai si un état **e** donné est un but, ce qui permettra de changer de but facilement.

Vous pouvez écrire la fonction **opPoss**, telle que **opPos(e)** retourne la liste des triplets **(operation,etatFils,int)** : liste des opérations possibles à partir d'un état, associées aux états d'arrivée et aux coûts de ces opérations (dans un premier temps, on donnera un cout de 1 à chaque opération).

Vous pouvez écrire une fonction **heuristique** ayant deux composantes : **g** et **h**. La fonction **h** sera telle que **h(e)** donne le coût minimal estimé pour aller d'un état **e** donné à un état-but. Pour définir une première version exploitable de **h**, vous pourrez définir la fonction **nombreMalMis** telle que **nombreMalMis(e,but)** renvoie le nombre de cubes qui sont mal placés dans **e** par rapport à **but**.

Heuristiques

Comme pour le taquin, la première heuristique utilisée pour guider la recherche (nombre de cubes mal placés) donne une indication du nombre de déplacements minimum à effectuer pour passer de l'état courant au but mais cette estimation n'est pas très précise et guide mal la recherche. Vous pouvez l'affiner en remarquant que plus les cubes mal placés sont bas dans chaque tige et plus il y aura de déplacements. Une heuristique simple à programmer est donc celle qui calcule la profondeur du cube le plus mal placé sur chaque tige et en fait la somme sur toutes les tiges, elle donne de bons résultats surtout en pondérant fortement les différences de profondeur.

Plus finement, on peut essayer de prendre en compte le nombre de cubes sur le cube mal placé et le nombre de cubes à dégager pour libérer la place qu'il doit occuper. La garantie que l'on a de trouver une solution optimale ne fonctionne que si l'heuristique h est minorante mais vous pouvez cependant essayer des heuristiques non minorantes en sachant que la solution obtenue ne sera pas nécessairement la meilleure solution.

Test de votre implémentation

Coût uniforme : les déplacements ont tous un coût de 1. Testez votre fonction et les différentes heuristiques employées sur les problèmes suivants. Pour chacun d'entre eux, notez le chemin-solution optimal et son coût, le nombre d'itérations, et pour chacune de ces itérations, le nombre de nœuds créés et le nombre de nœuds développés.

situation initiale 1

		1B	1R
2J		2B	2R
3J	1J	3B	3R

but1

1J		1B	1R
2J		2B	2R
3J		3B	3R

but2

1J	3R	1B	
2J	2R	2B	
3J	1R	3B	

situation initiale 2

1J	1B	1R	
2J	2B	2R	
3J	3B	3R	

but3

1R	2R	1J	
2J	1B	2B	
3J	3B	3R	

but4

2J	2B	2R	
1J	1B	1R	
3J	3B	3R	

but5

2R		2B	
2J	1B	1R	
3J	3B	3R	1J

but6

1J	2J	3J	
1R	2R	3R	
1B	2B	3B	

Coût non uniforme : on peut aussi considérer que les déplacements ont des coûts qui dépendent des cubes à déplacer. Ainsi, chaque cube pourrait correspondre à un objet plus ou moins difficile à déplacer (objet plus ou moins lourd nécessitant plus ou moins d'énergie pour son déplacement par exemple). Comment définir de nouvelles heuristiques qui prennent en compte ces coûts afin d'obtenir un chemin-solution de coût minimal ? Testez votre algorithme et vos différentes heuristiques sur les problèmes précédents pondérés de façon quelconque.