INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport de projet

Titre principal

Rédigé par :

Lemmers Nathan - Jakubiak Adrien

- Version du 27 février 2023 -

Table des matières

Introduction

Le procédé de résolution de problèmes par intelligence artificielle est une méthode utilisée couramment aujourd'hui dans la résolution de problèmes NP (complets ou difficiles).

Tout au long des quatre séances de travaux pratiques qui nous ont été proposées, nous avons eu l'occasion de mettre en application nos connaissances de cours pour résoudre deux problèmes, modélisés sous forme de jeu : le taquin et le morpion, en utilisant la programmation logique et le langage Prolog. Au fur et à mesure de ce rendu, nous développerons les techniques utilisées pour résoudre chacun de ces problèmes, les tests unitaires que nous avons mis en place pour nous assurer du bon fonctionnement des algorithmes ainsi qu'une petite analyse sur la réutilisabilité du code.

Le code source de chaque programme est accessible sur github, sous ce lien :
`https://github.com/nathanlemmers/Projet-IA`
Toutes les images utilisées sont les images présentes dans les documents de TP fournis.

1 Jeu du taquin

1.1 Présentation du jeu

Le jeu du taquin est un jeu en forme de damier dont la taille peut varier. De manière générale, il est représenté par un damier de taille $N \times N$ où chaque case est numérotée de 1 à $N^2 - 1$ et où une case est laissée vide. Le but du jeu est d'ordonner les pièces d'une certaines façon en permutant la case vide avec une de ses cases adjacentes.

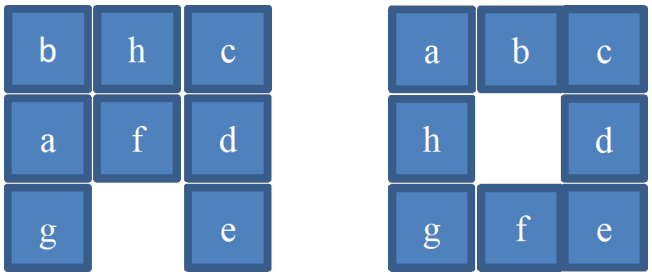

FIGURE 1 – Exemple d'état initial et d'état final pour un taquin en 3x3

1.2 Résolution du problème

Le problème est le suivant : comment, en partant de l'état initial, arriver à l'état final en minimisant le nombre de coups possible ? La résolution d'un taquin de taille $N \times N$ est un problème facile, mais trouver le plus petit ensemble de combinaisons pour résoudre le taquin est un problème NP-difficile. Il faut donc bien le modéliser pour trouver une solution en un temps acceptable et avec une quantité de mémoire demandée faible.

1.2.1 Modélisation du problème

Chaque état du jeu est représenté par une matrice de taille carrée, en spécifiant au départ un état initial et un état final. En réutilisant l'exemple donné en présentation, on va donc définir deux prédicats d'état initial et d'état final comme par exemple :

```
initial_state([ [b, h, c],
                [a, f, d],
                [g,vide,e] ]).

final_state([ [a, b, c],
              [h,vide, d],
              [g, f, e] ]).
```

Comme seules les pièces adjacentes au vide peuvent être déplacées, nous avons décidé de ne modéliser seulement le déplacement de la pièce vide. Nous modélisons ainsi un règle *rule* qui comprend le déplacement dans les quatres directions : haut, bas, gauche, droite, invoquant deux règles de permutation : horizontale et verticale.

```
% rule(+Rule_Name, ?Rule_Cost, +Current_State, ?Next_State)

rule(up,1,S1,S2) :-
   vertical_permutation(_X,vide,S1,S2).

rule(down,1,S1,S2) :-
   vertical_permutation(vide,_X,S1,S2).

rule(left,1,S1,S2) :-
   horizontal_permutation(_X,vide,S1,S2).

rule(right,1,S1,S2) :-
   horizontal_permutation(vide,_X,S1,S2).
```

Afin de parcourir tous les états intéressants du jeu, nous avons décidé (plus modestement, il nous a été imposé) de stocker tous les états sous la forme d'un arbre de recherche équilibré (avl) et de le parcourir à l'aide de l'algorithme A*. Cet algorithme est un algorithme de recherche de type "Best-first search" qui va associer à chaque état exploré une valeur, déterminée par une heuristique, et qui pour chaque itération va continuer d'explorer l'état ayant la plus faible valeur d'heuristique. Dans le jeu du taquin, on peut définir la fonction f telle que pour tout état x, $f(x) = h(x) + g(x)$, h étant la fonction heuristique donnée et g le nombre de combinaisons déjà réalisées. On demande donc à A* de choisir la valeur de f la plus petite dans les états déjà développés afin de continuer l'exploration et développer de nouveaux états.

L'exploration de l'arbre engendre parfois la création de nouveaux noeuds, états non encore explorés. Ces nouveaux états sont l'ensemble des permutations possibles et valides par rapport à l'état précédent. Par exemple en partant de l'état initial de la figure, il n'est pas possible de faire un déplacement vers le bas. Les déplacements possibles seraient haut, gauche et droite. La règle pour A* est découpée en 3 cas : quand il n'y a pas de solution, quand une solution a été trouvée et quand il reste encore à chercher.

```
aetoile(Pf, _, _) :-     % cas 1
    empty(Pf), % La pile est vide donc pas de solutions
    !,
    write("PAS DE SOLUTION : L'ETAT FINAL N'EST PAS ATTEIGNABLE !").
```

```
aetoile(Pf,Pu,Q) :-      % cas 2
    suppress_min([[F,_,_], U], Pf, _), % On recupere l'etat U avec la plus faible valeur de
        F
    final_state(U),                    % Et U est un etat final
    write("Solution trouvee : "),
    writeln(F) ,
    !,
    affiche_solution(Pu, Q).           % On affiche la solution


aetoile(Pf, Pu, Q) :-    % cas 3
    suppress_min([[F,H,G], U], Pf, Pf1),       % On recupere l'etat U avec la plus faible
        valeur de F
    suppress([U, [F,H,G], Pere, A], Pu, Pu1),  % On le supprime dans la deuxieme pile
    expand(U, G ,Lu),                          % On genere les nouveaux de l'arbre
    loop_successor(Lu, Pf1, Pu1, Q, Pf2, Pu2), % On les ajoute dans les piles Pf et Pu
    insert([U, [F,H,G], Pere, A], Q, Q1),      % On marque le noeud visite
    aetoile(Pf2, Pu2, Q1).                     % On reboucle
```

Si jamais un noeud est déjà présent dans les piles Pf et Pu, alors on ne garde que la plus petite de f. Si un noeud était déjà présent dans Q, alors il n'était pas ajouté dans les autres piles car il a déjà été marqué.

Comme dit précédemment, A* prend une fonction heuristique qui retourne une valeur en fonction de l'état considéré. Nous avons développé deux heuristiques différentes pour constater de l'importance du choix de l'heuristique dans un problème NP-difficile : le nombre de pièces, hors vide, mal placées (notée h_1) et la distance de *Manhattan* de chaque pièce (notée h_2). En prenant l'exemple de l'état initial S_0 de la figure proposée, $h_1(S_0) = 4$ et $h_2(S_0) = 5$.

```
% heuristique(+U, ?H)

heuristique1(U, H) :-              % Pieces mal placees
    final_state(F),
    findall(P, malplace(P,U,F), L), % On fait la liste des pieces mal placees
    length(L, H).                  % On retourne la longueur de la liste

distance(P,U, F, R) :-
    coordonnees([L,C], U, P),
    coordonnees([L1,C1], F, P),
    P\=vide,
    L2 is abs(L-L1),
    C2 is abs(C-C1),
    R is (L2+C2).

heuristique2(U, H) :-                  % Distance de Manhattan
    final_state(F),
    findall(R, distance(_, U, F, R), L), % On calcule la distance de chaque piece par
        rapport a l'arrivee et on le met dans une liste
    sumlist(L,H).                      % On retourne la somme de la liste
```

1.3 Analyse des résultats

Nous avons pu analyser le comportement de la résolution pour les deux heuristiques et réfléchi sur la réutilisabilité de notre production.

1.3.1 Différences dans les heuristiques

Nous avons testé les résolutions avec nos deux heuristiques et nous avons conclu certaines choses sur l'importance de ces dernières.

Sur les résolutions demandant jusque 10 coups, les deux heuristiques étaient plus ou moins équivalentes, même si h_2 paraissait sur la fin plus efficace.

Sur les résolutions plus complexes, une nette différence est apparue. Alors que jusqu'à présent le temps d'attente pour obtenir une solution était acceptable, l'utilisation de h_1 s'est démontrée beaucoup plus lente et plus demandant en mémoire. Pour 30 coups, la résolution était impossible, déclenchant une *stack exception* après un certain temps de recherche. h_2 mettait aussi du temps avant de fournir un résultat mais est bien plus rapide que h_1 et donnait une solution même pour les 30 coups.

Nous avons déjà été avertis sur l'importance du choix de l'heuristique et nous en avons eu la preuve dans le développement de cette résolution. L'heuristique h_2 était plus adaptée que h_1 pour la résolution de notre problème. Nous pensons que cela est dû à la quantité d'information transportée par la valeur de l'heuristique. Alors que h_1 ne donne que le nombre de pièces mal placées, h_2 indique le nombre de déplacements minimum encore nécessaires pour arriver à la solution, ce qui donne une meilleure description de l'état à A* dans son choix d'exploration.

1.3.2 Réutilisabilité du code

Ce code est généralisable à toute taille de taquin si tant est que la taille de la mémoire et le temps à disposition soit suffisant. Par ailleurs, il est possible d'utiliser ce code pour un autre problème comme le taquin où les bords ne sont pas définis, comme dans un tore. En effet, le jeu définissant sa propre heuristique, l'algorithme A* sera en mesure de générer les noeuds et de choisir les meilleures permutations.

L'algorithme A* n'est pas le seul algorithme qui sert à la résolution de problèmes. A partir de l'exemple du morpion, nous allons vous montrer une seconde technique pour un jeu avec plusieurs joueurs.

2 Jeu du morpion

2.1 Présentation du jeu

Le jeu du morpion est un jeu tour par tour opposant deux joueurs où pour chacun est assigné un symbole au début de la partie : croix ('x') ou rond ('o'). Le jeu est joué sur une grille carrée 3x3, initialement vide. A tour de rôle, le joueur pose sur une case libre son symbole dans l'espoir de compléter une ligne, une colonne ou une diagonale de son symbole.

Ce jeu est reconnu pour être équilibré, c'est-à-dire que si les deux joueurs jouent parfaitement au jeu, chaque partie devrait finir en match nul.

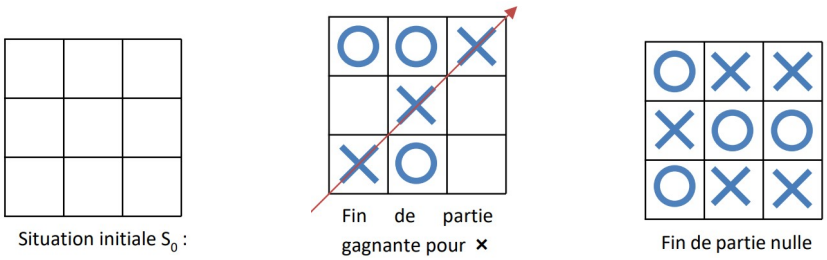

FIGURE 2 – Exemple d'état initial, gagnante et nulle pour un morpion 3x3

2.2 Représentation du problème

La première chose à faire a été de représenter le jeu comme une matrice 3x3 avec des variables non-unifiées à l'intérieur. Ainsi, l'algorithme pourra déterminer de lui même les meilleures possibilités, en vérifiant si certaines cases sont toujours unifiables ou fixées. Les premières parties ont donc été de savoir retrouver les différents alignements. Nous avons pu ensuite déterminer si une victoire était possible avec un alignement donné pour un joueur J, ainsi que déterminer si une situation S lui était gagnante ou perdante.

Voici quelques évaluations utiles que nous avons utilisées à partir des prédicats développés dans notre projet :

```
?- situation_initiale(S), joueur_initial(J).
?- situation_initiale(S), nth1(3, S, Ligne), nth1(2, Ligne, o).
```

La première évaluation permet de récupérer la situation initiale et de lier le joueur qui devra commencer à 'x'. La deuxième évaluation est un exemple de coup qui permet de jouer un 'o' a la troisième ligne de la deuxième colonne.

2.3 L'heuristique et sa profondeur

L'heuristique h d'un coup a été définie par la soustraction du nombre de situations gagnantes par le nombre de situations perdantes du joueur considéré. Par exemple, sur un jeu vide, il y aura 8 possibilités encore ouvertes de victoires pour le joueur et son adversaire, ce qui donnera 0. Pour un placement au centre pour le joueur 'x', $h(S) = 4$ vu que le joueur 'x' aura 8 possibilités de victoire alors que le joueur 'o' n'en aura plus que 4. Il devient alors possible de déterminer le meilleur coup à jouer en fonction du meilleur coup de l'adversaire, et donc prévoir le meilleur scénario pour le nombre de coups d'avance choisi (ce qu'on appellera la profondeur). Les situations gagnantes et perdantes sont respectivement représentées par les valeurs $+\infty$ et $-\infty$. Comme il est impossible de les représenter numériquement, nous avons choisi de les remplacer par une valeur arbitrairement grande, respectivement 10000 et -10000, afin de mettre un terme à la recherche.
Ainsi donc, nous définissons 3 cas du prédicat heuristique : la situation perdante, la situation gagnante et la situation jouable :

```
% heuristique(+J, +Situation, ?H)

heuristique(J,Situation,H) :-  % cas 1
    H = -10000,          % grand nombre approximant -infini
    alignement(Alig,Situation),
    alignement_perdant(Alig,J), !.
```

```
heuristique(J,Situation,H) :-  % cas 2
    H = 10000,        % grand nombre approximant +infini
    alignement(Alig,Situation),
    alignement_gagnant(Alig,J), !.

heuristique(J,Situation,H) :-   % cas 3
    findall(1, (alignement(Ali, Situation),possible(Ali,J)), Poss),
    length(Poss, K),
    adversaire(J, A),
    findall(1, (alignement(Ali, Situation),possible(Ali,A)), Poss2),
    length(Poss2, K2),
    H is (K-K2).
```

Le prédicat de test unitaire pour la fonction heuristique est "test_heuristique" et est utilisé pour vérifier que les résultats retournés sont corrects :

```
test_heuristique :-
    heuristique(x,[[o,o],[o,o]], -10000),     % Situation perdante
    heuristique(x,[[x,x],[x,x]], 10000),      % Situation gagnante
    heuristique(x,[[x,o],[o,o]], 0),          % 2 possibilites pour chaque joueur
    heuristique(x,[[x,_,_],[_,o,x],[_,x,o]],1), % 2 possibilites 'x' et 1 possibilite 'o'
    heuristique(x,[[x,_,_],[_,_,o],[_,x,_]],3). % 5 possibilites 'x' et 2 possibilites 'o'
```

2.4 Développement de l'algorithme, test et conclusion

Une fois le jeu modélisé, il reste encore à implémenter un algorithme de recherche dans un arbre pour trouver les solutions. Nous avons choisi l'algorithme Negamax, très utilisé dans les jeux à deux joueurs à somme nulle[1]. C'est une variante de l'algorithme minimax où $max(x, o) = -max(-x, -o)$, avec x la valeur de l'heuristique du joueur 'x' et o la valeur de l'heuristique du joueur 'o'.

Il est composé d'un prédicat découpé en 3 cas différents : quand il n'y a plus de coups à jouer et qu'on a atteint la profondeur maximale, quand il n'y a plus de coups à jouer mais que la situation est terminale et quand il y a encore au moins un coup à jouer :

```
% negamax(+J, +Etat, +P, +Pmax, [?Coup, ?Val]

negamax(J, Etat, P, P, [rien, Val]) :-     % cas 1
   heuristique(J, Etat, Val),  % On retourne la valeur de l'heuristique
   !.

negamax(J, Etat, _, _, [rien, Val]) :-     % cas 2
   situation_terminale(J, Etat), % On verifie l'etat terminal
   heuristique(J, Etat, Val),  % On retourne la valeur de l'heuristique
   !.

negamax(J, Etat, P, Pmax, [Coup, Val]) :-  % cas 3
   successeurs(J, Etat, ListeSucc),             % On genere la liste des successeurs
```

1. Types de jeux où l'avantage de l'un est exactement le désavantage de l'autre. La somme de tous les avantages et désavantages des deux joueurs, est égale à 0

```
    loop_negamax(J, P, Pmax, ListeSucc, ListeValeur), % On relance Negamax sur les
        successeurs du point de vue de l'adversaire
    meilleur(ListeValeur,[Coup, V1]),            % On choisit le meilleur coup parmi tous
    Val is -V1.                                  % On retourne l'oppose de la meilleure
        valeur
```

Comme la valeur de l'heuristique pour l'adversaire est l'opposé de notre valeur d'heuristique, le meilleur coup va être le coup avec la plus faible valeur d'heuristique. C'est pour cela qu'on retourne l'opposé de la meilleure valeur.

Nous avons testé notre algorithme avec différentes profondeurs, et voici ce qui nous a été retourné :

— Pmax = 1, Coup = [2, 2], Val = 4
— Pmax = 2, Coup = [2, 2], Val = 1
— Pmax = 3, Coup = [2, 2], Val = 3
— Pmax = 4, Coup = [2, 2], Val = 1
— Pmax = 5, Coup = [2, 2], Val = 3
— Pmax = 6, Coup = [2, 2], Val = 1
— Pmax = 7, Coup = [2, 2], Val = 2
— Pmax = 8, Coup = [1, 1], Val = 0
— Pmax = 9, Coup = [1, 1], Val = 0

On constate qu'au début, la valeur d'heuristique oscille : elle va dépendre du dernier coup qu'il aura prédit. Pour les profondeurs impaires, le joueur joue en dernier et se voit vainqueur. Pour les profondeurs paires, l'adversaire joue en dernier et le joueur se voit perdant.

On remarque qu'à partir d'une profondeur de 8, la valeur d'heuristique est de 0. L'algorithme comprend qu'il ne peut en réalité pas gagner si l'adversaire joue bien. De plus, il ne joue plus en [2, 2] mais en [1, 1]. Il va choisir le premier coup avec une valeur d'heuristique de 0 qu'il a généré, et il se trouve que dans ce cas là c'est la première case [2].

Pour tester le bon fonctionnement de notre algorithme, nous avons développé un prédicat de test unitaire. Il vérifie les différentes réponses de l'algorithme dans des cas intéressants :

```
test_unitaire:-
    joueur_initial(J),
    negamax(J, [[x,o,_],
                [x,o,_],
                [_,_,_]],0, 8, [[3,1],10000]),

    negamax(J, [[_,o,_],
                [x,o,_],
                [_,_,_]],0, 2, [[3,2],_]),

    negamax(J, [[x,_,o],
                [_,x,o],
                [_,_,_]],0, 8, [[3,3],10000]),

    negamax(J, [[_,o,_],
                [x,o,_],
                [_,_,_]],0, 8, [_,-10000]).
```

2. Notons d'ailleurs que si le premier coup n'est pas dans un coin ou au centre, il est forcément perdant !

Notre algorithme Negamax est donc opérationnel, sur des jeux de morpion de taille $N \times N$. Il serait également possible de jouer à d'autres jeux, comme le puissance 4, puisqu'ils ne dépendent que des valeurs d'heuristiques et des situations suivantes générées par le jeu.

Cependant, il est à noter que l'algorithme peut encore être amélioré, en pouvant repérant les symétries et élaguant ainsi les possibilités redondantes.

Conclusion

Ces deux différents projets nous ont permis de mieux comprendre les différents concepts liés à l'intelligence artificielle, ainsi qu'une utilisation concrète des différents algorithmes que nous avions précédemment découvert. Il était intéressant de faire le lien entre les graphes et les profondeurs dans les algorithmes d'IA, tout en approfondissant nos connaissances sur le langage Prolog.

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE