Implémentation en Prolog d'un algorithme A* d'un algorithme Negamax

Julien Ferry, Paul Florence, Célia Prat

INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
TOULOUSE

2 Avril 2019

Partie 1: A*

Familiarisation avec le problème du Taquin

Pour représenter la situation finale du taquin on utilise le prédicat `final_state/1`. Cela correspond à la matrice :

```
[[a,b,c],
[h,vide,d],
[g,f,e]
```

La requête prolog : `?-initial_state(Ini), nth1(L,Ini,Ligne),nth1(C,Ligne,d).` permet de trouver les coordonnées `(L,C)` de l'élément `d` sur le taquin (dans l'état initial de ce dernier).

La requête prolog : `?-final_state(Fin), nth1(3,Fin,Ligne), nth1(2,Ligne,P).` permet de trouver l'élément de coordonnées (3,2) dans l'état final du taquin.

Enfin, pour vérifier qu'une pièce est bien placée dans l'état initial par rapport à l'état final, on peut écrire le code suivant :

```
pos_init(K, X, Y) :-
    initial_state(Ini),
    nth1(Y, Ini, Ligne),
    nth1(X, Ligne, K).

pos_finale(K, X, Y) :-
    final_state(Fin),
    nth1(y, Fin, Ligne),
    nth1(X, Ligne, K).

is_correct(P) :-
    pos_init(P, X, Y),
    pos_finale(P, X, Y).
```

Pour trouver une des situations suivant l'état initial du taquin, on peut utiliser le prédicat suivant :

```
follow_init(Follow) :-
    initial_state(Ini),
    rule(Op, 1, Ini, Follow).
```

On peut utiliser ce prédicat dans un `findall` pour regrouper les réponses dans une liste, comme suit :

```
findall(States, follow_init(States), List).
```

Ou ainsi :

```
initial_state(Ini),
findall(States, rule(Op, 1, Ini, States), Liste).
```

Pour finir, si l'on veut obtenir la liste de tous les couples `[A, S]` tels que `S` est la situation qui résulte de l'action `A` depuis l'état `Ini`, on peut utiliser la requête suivante :

```
initial_state(Ini),
findall([Op, State], rule(Op, 1, Ini, State), Liste).
```

Analyse expérimentale

L'implémentation de l'algorithme A* présentée ici utilise 3 AVL (arbres binaires de recherche). D'une part, Pu et Pf, qui contiennent les nœuds représentant la « frontière d'exploration » de l'algorithme (c'est-à-dire les états que l'on a atteints mais pas encore développés), et d'autre part Q, qui contient les états déjà explorés et développés. Pu permet d'accéder en temps logarithmique à l'état de la frontière le plus intéressant à développer (celui qui a la meilleure somme (heuristique + coût) associée). Pf permet de récupérer les informations sur un état de la frontière efficacement (car il permet de trouver un élément en temps logarithmique à partir de son nom). De même, Q permet de récupérer les informations sur un état déjà développé et exploré (car, de même, un nœud peut être trouvé dans l'AVL en temps logarithmique). L'utilisation de ces trois arbres permet donc de manipuler les états au cours de l'exploration en temps logarithmique pour chaque accès (O(log(n)).

Nous avons testé les deux heuristiques avec les cas de tests proposés sur un taquin 3x3. A noter que la taille du taquin n'influe que très peu sur la durée d'exécution de l'algorithme, puisque le nombre de coup possibles à explorer est toujours compris entre 2 et 4 peu importe la taille de la grille. Les cas de tests se différencient donc par le nombre de coups nécessaires à l'atteinte de la situation finale.

Tout d'abord, les deux heuristiques trouvent les mêmes résultats ce qui indique qu'elles n'influent pas sur la validité de l'algorithme. Cependant, l'heuristique 2 est clairement plus performante en temps que l'heuristique 1, en particulier quand le nombre de coups nécessaires est grand.

Nous avons effectué les tests de performances sur un processeur d'ordinateur portable I5 6300HQ @ 2.30 GHz. A noter également qu'avec la taille de stack de swipl par défaut, le programme subissait un Stack Overflow pour les cas 5 et 6 avec l'heuristique 1, et pour le cas 6 seulement avec l'heuristique 2. Nous avons donc augmenté la mémoire allouée à prolog en exécutant swipl ainsi :

```
swipl -G100g -T20g -L2g
```

Voici les résultats de notre analyse de performance des deux heuristiques.

Cas de test	Temps de l'heuristique 1 (en ms)	Temps de l'heuristique 2 (en ms)	Nombre de coups de la solution
initial_state2	<1	1	2
initial_state	2	3	5
initial_state3	17	5	10
initial_state4	628	20	20
initial_state5	54 689	677	30
initial_state6	78 541	90 232	Impossible

Partie 2 : Negamax & alpha-beta

Familiarisation avec le problème

La requête `?- situation_initiale(S), joueur_initial(J).` permet de définir la situation initiale dans `S` et le joueur initial dans `J`.

La requête `?- situation_initiale(S), nth1(3,S,Lig), nth1(2,Lig,o)` permet de lire le contenu de la case `[3,2]` dans l'état `S` (ici l'état initial).

On peut facilement accéder à la ligne d'une matrice en utilisant le prédicat `member/2`. Pour accéder à la colonne, c'est un peu plus compliqué , on utilise un prédicat d'aide `colonne_k/3` qui nous retourne la `k`_iéme colonne.

```
colonne(C,M) :-
    colonne_k(C,_,M).

colonne_k(C,K,[L|M]) :-
    colonne_k(Col,K,M),
    nth1(K,L,E),
    append([E],Col,C).

colonne_k([],_,[]).
```

Pour la diagonale, en s'inspirant de la première implémentation de la diagonale on peut implémenter la seconde diagonale comme suit :

```
seconde_diag(_,[],[]).
seconde_diag(K,[E|D],[Ligne|M]) :-
    nth1(K,Ligne,E),
    K1 is K - 1,
    seconde_diag(K1,D,M).
```

Il est alors possible de composer les deux implémentations ainsi :

```
diagonale(D, M) :-
    premiere_diag(1,D,M);
    length(M,K),
    seconde_diag(K,D,M).
```

On a aussi implémenté un mode de jeu interactif qui permet de jouer contre notre algorithme.

Jouer contre l'ordinateur

Nous avons implémenté des prédicats permettant de jouer contre notre algorithme.

Pour contourner l'absence d'input utilisateur en Prolog, il est nécessaire de construire la requête correspondant à la partie entière « coup après coup », en copiant-collant et en rajoutant le coup suivant.

Pour jouer contre l'ordinateur, il faut réaliser les opérations suivantes :

1. Initialiser le jeu en appelant `situation_initiale(Situation)`
2. pour que l'ordinateur joue un coup, appeler `play_ordi(Situation,Pmax)`
3. pour jouer un coup, appeler `play_joueur(Situation, [Ligne, Colonne])`

Quand quelqu'un joue, le programme l'indique dans la console (et affiche la grille résultante). Quand quelqu'un a gagné, le programme l'indique dans la console (et affiche la grille gagnante).

Voici un exemple de séquence menant à la victoire de l'ordinateur :

```
situation_initiale(S),  % On initialise le jeu
play_ordi(S,5), % On fait d'abord jouer l'ordinateur, avec une
    profondeur d'analyse de 5 (choix arbitraire)
play_joueur(S,[1,1]), % On joue en [1,1]
play_ordi(S,5),       % ..etc
play_joueur(S,[1,3]),
```

```
play_ordi(S,5). % La l'ordinateur a gagné...
```

Analyse expérimentale

Comportement de l'algorithme pour différentes profondeurs

Pour une profondeur allant de 1 à 7, l'algorithme marche toujours. Par exemple, dans le cas où notre situation correspond au premier coup du premier joueur, l'algorithme conseille de jouer au centre.

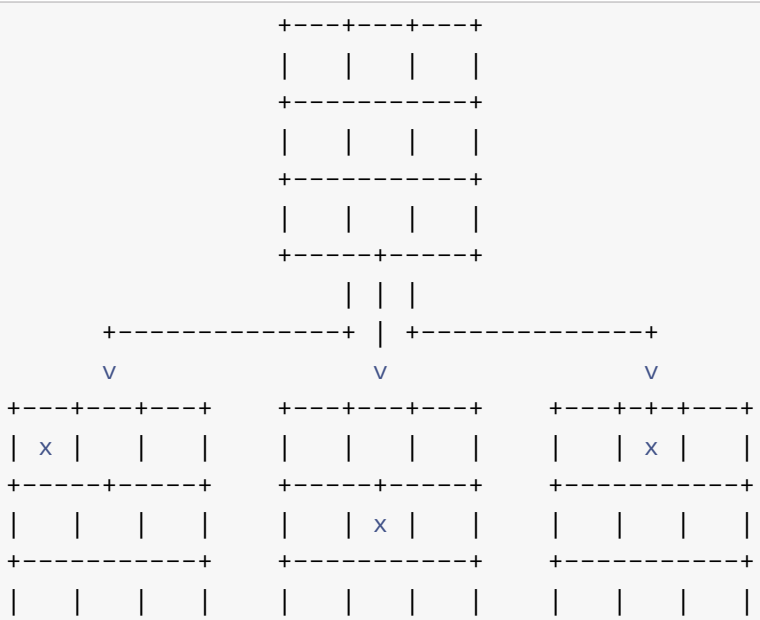
Pour une profondeur de 8, l'algorithme implémenté tel que présenté dans le sujet arrive dans une situation gagnante ce qui est anormal. L'ajout d'une clause `negamax/5` correspondant à la situation dans laquelle un joueur ne peut plus jouer car quelqu'un a déjà gagné a réglé ce problème et, comme dans le cas d'une profondeur de 9, l'algorithme nous prédit une égalité dans la moins pire des situations.

Pour une profondeur de 9, les feuilles correspondent à des situations dans lesquelles la grille est entièrement remplie et aucun coup n'est possible. L'algorithme nous renvoie alors 0 comme meilleure solution. En effet cela nous garantit que dans le pire des cas le résultat soit une égalité.

Optimisation possible

Afin d'éviter de développer inutilement des situations symétriques de situations déjà développées il faudrait jouer les coups de manière unique en tenant compte des symétries possibles (il y en a 4).

Ainsi, avec une profondeur de 1 l'arbre ressemblerait à cela :



```
17  +---+---+---+     +---+---+---+     +---+---+---+
```

Il faut cependant faire attention à ce que le coût du calcul des symétries ne soit pas supérieur au coût de calcul des noeuds.

Adaptation au puissance 4

Cet algorithme est adaptable au puissance 4. Un coup sera caractérisé uniquement par l'indice de la colonne dans laquelle on insère le jeton.

Cette adaptation nécessite la modification de quelques prédicats :

- `alignement_gagnant/2` : un alignement gagnant pour un joueur donné correspond à 4 jetons alignés peu importe où dans la grille, qui peut être plus grande que 4x4. Il faut donc réutiliser les prédicats `diagonale/2`, `colonne/2`, `ligne/2` mais en rajoutant un accumulateur pour « compter » jusqu'à 4.
- `possible/2` : même problématique que pour `alignement_gagnant/2`, il faut vérifier qu'il reste des emplacements libres pour faire un alignement de 4 jetons.
- `successeur/3` : trouver les coups jouables en suivant doit prendre un compte l'empilement vertical des jetons de puissance 4, colonne par colonne : un coup n'est jouable que si la colonne n'est pas déjà pleine, et le jeton sera ajouté au-dessus des jetons déjà présents.
- `heuristique/3` : la présence d'un alignement gagnant ou perdant correspond toujours à une heuristique de respectivement $+\infty$ et $-\infty$. Les autres états peuvent être estimés de la façon suivante : plus le joueur est proche d'un (ou plusieurs) alignement(s) gagnant(s) (c'est-à-dire plus la quantité de jetons présents dans un alignement possible est grande), plus l'heuristique augmente. À l'inverse, si c'est l'adversaire qui se rapproche d'un ou des alignements gagnants, elle diminue. Ainsi cette heuristique prend en compte le fait qu'un coup est plus avantageux si il nous rapproche de deux (voir trois) alignements différents en même temps (ex : il est placé au croisement d'un alignement colonne et d'un alignement diagonal).

Élagage des branches (type $\alpha\beta$)

On peut optimiser le temps de recherche en utilisant AlphaBeta pour ne pas explorer certaines branches. L'arbre de recherche comprend des nœuds Max (joueur) et des nœuds Min (adversaire). Les nœuds Max cherchent à prendre la valeur maximale trouvée parmi leurs nœuds Min fils. A l'inverse, les nœuds Min cherchent à prendre la valeur minimale renvoyée par leurs nœuds Max fils. Si l'on trouve, en explorant les fils d'un nœud Min, lui-même fils d'un nœud Max dont on sait déjà par exploration d'un autre fils que son heuristique sera supérieure à X (valeur de α), une valeur inférieure à X sur une branche fille, alors on sait que l'heuristique du nœud Max restera X car celle du nœud Min sera forcément inférieure

à X (on est dans le cas $\alpha > \beta$), on peut donc arrêter l'exploration des fils du nœud Min. Le raisonnement inverse vaut pour le choix de l'heuristique des nœuds Min ($\alpha < \beta$).