



David Anderson - Jérôme Flesch

TOURS DE HANOÏ HEURISTIQUE
IA41

Automne 2005

Table des matières

Introduction	3
1 Décomposition du problème	4
2 Représentation des données	4
3 Manipulation du plateau de jeu	5
4 Affichage	6
5 Génération d'une situation de jeu	6
6 Fonctions d'évaluation	7
6.1 Distance de Manhattan	7
6.2 Distance de Belfort	7
7 Algorithme A^*	8
8 Problèmes rencontrés	9
Conclusion	10

Introduction

L'objectif de ce projet est d'utiliser les connaissances acquises au cours de l'UV IA41 pour réaliser un projet en Lisp. Le projet consiste à réaliser une version amélioré du jeu des tours de Hanoï, que l'ordinateur résout lui-même.

La version originale du jeu des tours de Hanoï est un problème connu dans la littérature informatique, et l'algorithme donnant sa solution optimisée est bien connu.

Dans cette version améliorée, la situation est compliquée par l'ajout d'une deuxième tour de disques. Il faut alors aller vers une position solution en déplaçant simultanément les deux tours, ce qui rend impraticable l'algorithme général de résolution du cas simple.

De plus, nous nous sommes fixés comme but de permettre la résolutions de systèmes plus complexes encore, en laissant la liberté de résoudre des systèmes avec plus de deux tours, de construire des tours de plus de trois disques de haut, et en permettant un nombre arbitraire de tiges.

Dans ces versions, la position d'arrivée n'est pas définie globalement, mais est également fournie par l'utilisateur. Par exemple, voici l'une des situations de jeu que nous avons été amenés à résoudre au cours de notre projet :

Situation initiale

~~~~~

|           |   |   |           |
|-----------|---|---|-----------|
|           |   |   |           |
| a a       |   |   | b b       |
| aa aa     |   |   | bb bb     |
| aaa aaa   |   |   | bbb bbb   |
| aaaa aaaa |   |   | bbbb bbbb |
| =====     |   |   |           |
| A         | B | C | D         |

Situation finale

~~~~~

a a	aa aa	aaa aaa	aaaa aaaa
b b	bb bb	bbb bbb	bbbb bbbb
=====			
A	B	C	D

1 Décomposition du problème

Une fois le problème défini et les restrictions établies, nous avons décomposé le problème de l'écriture du programme en plusieurs sous-problèmes :

- Définir la représentation des données (plateau de jeu)
- Écrire un jeu de fonctions permettant de manipuler ce plateau de jeu.
- Écrire des fonctions permettant d'afficher ce plateau dans une forme agréable
- Écrire des fonctions permettant de générer aléatoirement un état de jeu
- Écrire les fonctions de résolution du jeu proprement dite

2 Représentation des données

La manière de représenter les données conditionne une grande partie de l'implémentation et des performances finales du jeu. Toutefois, une représentation qui semble efficace a été assez simple à trouver.

D'abord, nous avons défini chaque état de jeu comme étant une liste, contenant l'ensemble des tiges du jeu. Cette disposition rend facile l'accès à chaque tige.

Il nous a ensuite fallu définir la représentation des tiges. Pour cela, nous avons d'abord pensé les représenter elles aussi directement sous forme de listes simples. Cependant cette représentation s'est révélée problématique lorsque nous avons mis au point la fonction (`get-peg-n N S`), `S` étant l'état de jeu, et `N`, le numéro de la tige à extraire, en partant de 0. En effet, avec cette représentation, il est impossible de savoir, quand la fonction renvoie nil, si cette valeur signifie la tige n'existe pas ou si aucun disque ne s'y trouve.

Afin de remédier à ce problème, nous avons choisi de représenter chaque tige sous forme de liste associative : Chaque liste correspondant à une tige contient alors systématiquement deux éléments : Un atome donnant au nom de la tige, et une sous-liste listant les disques présents sur la tige. Comme en Lisp il est plus aisé de retirer la tête d'une liste plutôt que son élément de fin, et comme les règles du jeu de Hanoï ne permettent que de retirer le disque le plus haut sur une tige, les disques ont été mis du plus haut au plus bas dans la liste.

Pour représenter chaque disque, nous avons choisi d'utiliser des listes de deux éléments : Le premier est nombre définissant le rayon du disque, et le deuxième est un atome indiquant à quel tour le disque appartient. Il peut en effet y avoir plusieurs tours ayant des disques de même rayon, alors que les disques ne sont pas interchangeables. L'identifiant de tour nous permet de les différencier dans ce cas là.

3 Manipulation du plateau de jeu

Une fois les structures de données mises en place, nous avons écrit un jeu de fonctions permettant de manier et valider ces états. Ces fonctions sont regroupées dans le fichier source `structure.el`.

Les deux fonctions principales de ce fichier sont les fonctions `move-disc` et `check-move`. La première déplace le disque en haut d'une pile donnée vers une autre pile, sans vérifier si le mouvement est valide ou non. La seconde reçoit en entrée un état de jeu et l'indice de la tige de destination du dernier mouvement, et contrôle si ce dernier mouvement était valide. S'il ne l'est pas, `nil` est renvoyé à la fonction appelante.

Le contrôle de validité s'effectue simplement en vérifiant que le disque en haut de la pile est soit le seul disque de la pile, soit que le disque directement en dessous a un rayon supérieur ou égal à celui du dessus. Ce contrôle possède l'inconvénient de ne pas contrôler l'ensemble des disques de l'état, mais présente l'avantage non-négligeable d'être rapide et tout aussi efficace, pour peu que tous les mouvements effectués soient faits à partir d'un état initial valide, et que chaque déplacement soit validé. La fonction `move-disc-checked` regroupe les fonctions de déplacement et de validation en un seul appel.

La fonction `get-discs-from-game` et ses sous-fonctions sont utilisées exclusivement par le module d'affichage d'état de jeu. En effet, pour fonctionner, celui-ci a besoin de lire les disques dans un ordre orthogonal à l'ordre dans lequel ils sont stockés. Cette fonction est donc peu efficace, car elle doit effectuer beaucoup d'opérations que la structure de jeu ne facilite pas. Ceci est néanmoins acceptable, car l'affichage n'est appelé que rarement, et que la disposition choisie permet d'optimiser d'autres fonctions, bien plus souvent employées.

get-disc Renvoie le disque situé en haut de la pile donnée.

remove-disc Retire totalement le disque en haut de la pile donnée de l'état de jeu.

add-disc Ajoute un disque donné en haut de la pile donnée.

move-disc Déplace le disque en haut de la pile de départ sur la pile d'arrivée.

check-move Vérifie que le haut de la tige donnée est conforme aux règles de Hanoï.

move-disc-checked Déplace un disque d'une tige à une autre et vérifie la validité du mouvement.

get-discs-from-game Renvoie la liste des disques présents dans l'état de jeu, avec leurs coordonnées associées.

4 Affichage

Le fichier `display.el` contient toutes les fonctions d’affichage.

La fonction la plus couramment utilisée pour l’affichage est (`display-game S`), `S` étant un état de jeu. Cette fonction fait appel à une série de sous-fonctions pour afficher graphiquement ligne par ligne, tige par tige, l’état du jeu.

Afin de permettre l’affichage ligne par ligne sans devoir systématiquement chercher la plus haute tour de l’état de jeu, une constante a été définie au début du fichier `display.el` indiquant l’hauteur de départ de l’affichage.

De même, pour éviter de devoir chercher les disques les plus larges de chaque tour, une autre constante définit l’écart, en nombre d’espaces, entre chaque tige.

Ces constantes suffisent à afficher des cas assez extrêmes, à savoir des tours faisant 9 disques de haut, avec des rayons de disque de 5. Étant donné les ressources nécessaires pour les calculs, nous n’avons jamais réussi à dépasser ces limitations dans des situations utiles à représenter.

display Affiche la valeur passée en argument.

endl Affiche le caractère fin de ligne, pour passer à la ligne suivante.

display-game Affiche l’état de jeu donné en argument.

display-games Affiche chaque état de jeu de la liste donnée en argument, puis affiche le nombre d’états que contient la liste.

5 Génération d’une situation de jeu

Afin de pouvoir mettre à l’épreuve notre système, nous avons écrit une fonction permettant de générer une position aléatoire qui soit conforme aux règles du jeu. Le code correspondant à cette génération se trouve dans le fichier `posgen.el`. Il est possible de générer des états de jeu comportant un nombre arbitraire de tiges, de tours et disques.

Pour générer une position aléatoire qui soit valide, nous sommes tout simplement partis d’un état de jeu vide. À partir de là, nous ajoutons récursivement tous les disques sur des tiges aléatoires, en commençant par les disques de plus grand rayon. Cela nous assure d’obtenir un état de jeu qui soit valide, puisque les plus petits disques sont posés en dernier.

Afin de pouvoir étalonner les différents algorithmes testés par la suite sur des cas identiques, nous avons enregistré un nombre de situations possibles dans le `game.el`.

generate-position Génère un état de jeu aléatoire avec les tours, nombres de disques et de tiges donnés.

drop-discs Pose aléatoirement tous les disques d’un rayon donnée de toutes les tours dans l’état de jeu.

6 Fonctions d'évaluation

La fonction d'évaluation heuristique est la fonction qui est au coeur du programme. C'est pourquoi nous avons passé un temps considérable à rechercher un indicateur qui reflète fidèlement l'avancement de la résolution.

6.1 Distance de Manhattan

La première idée qui nous soit venue a été de comparer la position actuelle de chaque disque avec sa position dans l'état final. Nous avons donc réfléchi à un algorithme. Après l'avoir trouvé, nous avons fait des recherches et trouvé que nous avons redécouvert la *distance de Manhattan*, ou *Distance L_1* .

Ce calcul correspond, pour chaque disque, à l'opération suivante, si on considère que (x_1, y_1) est la position actuelle du disque et (x_f, y_f) est sa position finale à atteindre :

$$D_m = |x_1 - x_f| + |y_1 - y_f|$$

L'heuristique additionne ensuite toutes ces distances afin d'obtenir un nombre représentatif de l'écart entre un état de jeu donné et l'état final voulu.

Après avoir mis cette heuristique à l'épreuve, nous avons trouvé qu'elle apportait un gain sur l'algorithme de Dijkstra qui allait de 10% à 30% environ (gain plus élevé sur les situations complexes).

6.2 Distance de Belfort

Après une réflexion plus approfondie, nous avons constaté que la distance de Manhattan ne rendait vraiment pas efficacement compte de l'approchement de la solution, car elle ne tient pas compte du nombre de disques supplémentaires qui se trouvent au dessus du disque considéré, dans ses deux positions. Ces disques induiront des déplacements supplémentaires, dont l'heuristique devrait tenir compte. Nous avons donc développé un algorithme modifié, qui complète la distance de Manhattan en multipliant celle-ci par la somme du nombre de disques situés au dessus des deux positions étudiées.

Avec cette heuristique, certains résultats sont trouvés moins efficacement qu'avec la distance de Manhattan. Mais pour d'autres, et en particulier pour les plus complexes (tours de 4 disques), les améliorations sont frappantes, avec une optimisation sur l'algorithme de Dijkstra pouvant aller jusqu'à 90% (soit 11 fois plus rapide), et une amélioration par rapport à la distance de Manhattan de 70% (3.5 fois plus rapide). Nous avons donc décidé de garder cette heuristique pour notre programme, qui nous semblait plus optimisée pour les cas qui le nécessitent.

heuristic Calcule l'estimation de distance entre l'état donné et un but donné.

compare-positions Renvoie l'estimation de distance entre deux états (actuel et cible) d'un disque donné.

7 Algorithme A^*

L'algorithme A^* implémenté suit la procédure détaillée en cours. Tout d'abord, tous les états possibles sont engendrés à partir d'un état de jeu donné. Ils sont ensuite filtrés pour éliminer les mouvements invalides, et les états restants sont soumis à la fonction heuristique. Ces structures évaluées sont ensuite insérées de façon triée (meilleure estimation devant) dans la liste des ouverts.

Afin de garder la trace des états parents dans la liste (et donc former un arbre de recherche A^*), nous avons écrit la fonction `make-new-id` (située dans `astar.el`), qui renvoie un identifiant unique à chaque appel en se réécrivant pour incrémenter son compteur interne. Le compteur est réinitialisé à chaque recherche par `reset-id`.

Nous avons au départ conçu et implémenté l'algorithme A^* par récursion, comme il nous l'a été enseigné. Mais en raison de gros problèmes avec les limitations d'emacs (voir section suivante), nous avons décidé de faire une entorse à cette règle, et d'implémenter la boucle principale de l'algorithme de manière itérative. Implémenté de cette façon, le programme peut trouver des solutions pour des configurations plus complexes, car la limite de récursion est atteinte bien plus lentement.

Nous avons recherché une pondération optimale entre l'estimation heuristique et le coût effectif. Les tentatives que nous avons fait semblent indiquer que, dans la plupart des cas, le fait de favoriser légèrement le coût par rapport à l'heuristique diminue légèrement le temps de recherche, mais pas d'une marge vraiment significative. Nous avons finalement adopté une pondération (3, 2), où le coût est multiplié par 3 et l'heuristique par 2 pour former la valeur finale de l'intérêt du noeud.

find-solution Boucle principale de l'algorithme, qui recherche le chemin entre les deux positions données.

consider-move Etudie le déplacement donné et le classe ou non dans la liste des ouverts selon le cas.

astar-vars Fonction utilisée comme conteneur, stockant les listes des ouverts et des fermés pour l'algorithme.

reset-astar-vars Redéfinit la fonction `astar-vars` à son état initial.

8 Problèmes rencontrés

Au cours de notre développement, nous avons rencontré un certains nombres de problèmes.

Le premier de ces problèmes est liés aux limitations par défaut de emacs. En effet, deux variables définissent respectivement la profondeur maximum de récursivité et le nombre maximum de variables définies localement. Or leurs valeurs par défaut, respectivement 300 et 600, se sont rapidement révélées trop faibles pour permettre à notre programme de finir ses recherches de solutions.

Cependant, avec quelques optimisations (dont l'implémentation itérative décrite précédemment), nous avons réussi à approcher cette limite : Pour un jeu de tours de Hanoï classique à 3 tiges, avec uniquement des disques sur la première tige et qui doivent tous être déplacés sur la troisième tige, ces limites peuvent être respectivement descendu à 425 et 750.

Un autre problème que nous avons eût concerne la recherche de solution sur des jeu de quatre tiges : En effet, dans certains cas, emacs, après un temps de recherche assez long (plusieurs dizaines de minutes), voit son fonctionnement interrompu suite à une erreur de segmentation. Nous n'avons pas pu identifier la cause de ce problème, mais nous soupçonnons qu'il est lié à la grande quantité de mémoire devant être allouée par emacs pour maintenir la liste des ouverts et des fermés dans la recherche du chemin.

Conclusion

Au cours de la réalisation de ce projet, nous avons eût l'occasion de mettre en pratique ce que nous avons pu apprendre en cours, tel que la programmation récursive en Lisp, l'implémentation de l'algorithme A^* , etc. Ce projet nous a notamment permis de voir l'intérêt de cet algorithme, ainsi que ses limites, dans l'implémentation qui est faite dans Emacs.

Il a aussi été intéressant pour nous de voir les mises en relations possibles avec d'autres UV. Par exemple, en LO43 (Programmation orienté objet), nous avons du implémenter un système de recherche de chemins dans un réseau de transport. Les enseignements, ainsi que les recherches que nous avons déjà faites à ce sujet nous ont permis d'implémenter rapidement et efficacement une recherche A^* efficace.