Laura Nordmann – Thomas Petazzoni

Jeu à 1 joueur
Taquin amélioré

Projet IA41

Semestre d'automne 2002-2003

Université de Technologie
de Belfort Montbéliard

Table des matières

Introduction

L'objectif du projet est d'utiliser les connaissances acquises au cours de l'UV IA41 pour réaliser un projet en Lisp. Le projet consiste à réaliser un jeu à un joueur, que l'ordinateur résout lui même.

La toute première partie du projet consistait à choisir un jeu. Pour notre part, nous avons choisi de réaliser un taquin "amélioré". Dans le taquin "classique", chaque case est un numéro et il s'agit de remettre les numéros dans l'ordre de gauche à droite et de haut en bas.

Dans notre taquin amélioré, il y a 4 numéros par case, un en haut, un en bas, un à droite et un à gauche. Il s'agit de positionner les cases de manière à ce que les numéros entrent en correspondance. Le numéro de gauche d'une case doit entrer en correspondance avec le numéro de droite de la case immédiatement à gauche.

```
-----------------------------
|  64  |  42  |  34  |  99  |
|94  85|85  31|31  66|66  28|
|  18  |  87  |   1  |  63  |
-----------------------------
|  18  |  87  |   1  |  63  |
|78  62|62  82|82  85|85  28|
|  76  |  51  |  99  |  93  |
-----------------------------
|  76  |  51  |  99  |  93  |
|21  98|98  60|60  96|96  73|
|  14  |  70  |  89  |  55  |
-----------------------------
|  14  |  70  |  89  |      |
|46  81|81  86|86  46|   *  |
|  64  |  26  |   1  |      |
-----------------------------
```

Exemple de taquin amélioré résolu

La règle du jeu est donc très simple, même si la recherche de la solution est plus complexe que pour le taquin "classique".

1 Décomposition du problème

Le problème de l'écriture du programme a d'abord été découpé en sous-problèmes :

- Définir la représentation des données (le plateau de jeu).
- Écrire un jeu de fonctions de base permettant de manipuler ce plateau de jeu
- Écrire des fonctions permettant de réaliser l'affichage de ce plateau de jeu
- Écrire des fonctions permettant de générer aléatoirement un problème solutionnable
- Écrire les fonctions de résolution du jeu proprement dite

2 Représentation des données

La manière de représenter le plateau de jeu a été simple à déterminer. Nous avons tout d'abord décidé de représenter chaque case (contenant 4 nombres) par une liste. Par exemple

```
'(64 85 18 94)
```

Représente la première case en haut à gauche du plateau de jeu présenté en introduction. Les nombres sont écrits à partir du nombre du haut, puis en tournant dans le sens inverse des aiguilles d'une montre. La case vide, sera notée `'*`, tout simplement.

Chaque ligne est composée de quatres cases : on représentera donc chaque ligne à l'aide d'une liste, elle même composée de 4 sous-listes, ou 3 sous-listes et un élément `'*`. Par exemple, la première ligne du plateau d'exemple est représentée de la manière suivante :

```
'( (64 85 18 94) (42 31 87 85) (34 66 1 31) (99 28 63 66))
```

Tandis que la dernière ligne sera représentée de la manière suivante :

```
'( (14 81 64 46) (70 86 26 81) (89 46 1 86) *)
```

Enfin, le plateau de jeux est constitué de quatres lignes, on le représentera donc sous la forme d'une liste de quatres lignes :

```
'( ( (64 85 18 94) (42 31 87 85) (34 66 1 31)  (99 28 63 66) )
   ( (18 62 76 78) (87 82 51 62) (1 85 99 82)  (63 28 93 85) )
   ( (76 98 14 21) (51 60 70 98) (99 96 89 60) (93 73 55 96) )
   ( (14 81 64 46) (70 86 26 81) (89 46 1 86)  * ))
```

3 Manipulation du plateau de jeu

Nous avons tout d'abord commencé par écrire des fonctions permettant de manipuler le plateau de jeu.

extract-case Fonction permettant d'extraire une case donnée du plateau de jeu.

chnieme Fonction changeant le nieme terme d'une liste par un élément donné.

set-case Fonction changeant une case du plateau de jeu par une case donnée.

exchange-case Fonction échangeant deux cases du plateau de jeu, sans aucune vérification.

find-empty-case Fonction retournant la position de la case vide dans le plateau de jeu.

check-movements Fonction éliminant d'une liste de mouvements les mouvements impossibles, parce qu'ils débordent du plateau de jeu.

generate-possible-movements Fonction générant une liste des mouvements possibles, connaissant la possition de la case vide.

compare-case-y Fonction testant la correspondance en hauteur de deux cases.

compare-case-x Fonction testant la correspondance en largeur de deux cases.

Ces fonctions de base sont utilisées tout au long du programme.

4 Affichage

show-line-one-number Fonction affichant une ligne n'ayant qu'un seul chiffre (le haut et le bas de chaque case).

show-line-two-numbers Fonction affichant une ligne ayant deux chiffres (le centre de chaque case).

show-line Fonction affichant une ligne entière.

show-table Fonction affichant le plateau de jeu.

5 Génération aléatoire d'un plateau de jeu solutionnable

Nous souhaitons générer un plateau de jeu qui soit un problème ayant une solution. Pour celà, nous générons tout d'abord un plateau de jeu résolu :

nous le construisons de gauche à droite et de haut en bas, en respectant les règles de correspondance de notre taquin, puis nous mélangeons les cases de ce jeu.

La variable globale `*numberlist*` est une liste de nombres utilisée pour assurer l'unicité des nombres au sein d'un plateau de jeu.

get-left-number Fonction retournant le numéro en correspondance de la case de gauche.

get-up-number Fonction retournant le numéro en correspondance de la case du dessus.

generate-number Fonction générant un nombre entre 0 et 100 qui ne soit pas encore utilisé dans le plateau de jeu.

generate-element Fonction générant une case complète, de manière à ce qu'elle entre en correspondance avec la case du haut et la case de gauche.

generate-line Fonction générant une ligne complète.

generate-col Fonction générant les quatre lignes.

randomize-table Fonction mélangeant un plateau de jeu.

get-empty-table Fonction retournant un plateau de jeu vide.

generate-table Fonction générant un plateau de jeu complet, après mélange aléatoire.

6 Fonctions d'évaluations

6.1 Première fonction d'évaluation

L'idée de notre première fonction d'évaluation était de compter le nombre de chiffres en correspondance dans le plateau de jeu. Ce nombre augmentant au fur et a mesure que l'on se rapproche de la solution, notre fonction d'évaluation devient $22 - nb\_chiffres\_en\_correspondance$.

Cette fonction d'évaluation est répartie dans les fonctions Lisp suivantes :

evaluate-x Fonction calculant le nombre de chiffres en correspondance sur une ligne donnée.

evaluate-y Fonction calculant le nombre de chiffres en correspondance sur une colonne donnée.

evaluate-all-x Fonction calculant le nombre de chiffres en correspondance sur l'ensemble des lignes du plateau.

evaluate-all-y Fonction calculant le nombre de chiffres en correspondance sur l'ensemble des colonnes du plateau.

evaluate1 Fonction d'évaluation.

6.2 Deuxième fonction d'évaluation

Les raisons nous ayant poussé à réaliser une deuxième fonction d'évaluation sont exposées dans la section *Problèmes rencontrés*.

Cette fonction d'évaluation essaie de calculer la distance entre un chiffre et son correspondant dans le plateau de jeu. Pour chaque chiffre du plateau, on cherche son correspondant, puis on calcule la distance entre les deux. Toutes les distances ainsi sommees forment la valuation d'un état du jeu, valant 0 si l'état est solution du problème.

find-number Fonction cherchant dans le plateau de jeu un nombre donné ne se trouvant pas à une position donnée. Cette fonction retourne la position du nombre trouvé s'il existe, `nil` sinon.

evaluate-number-distance Fonction évaluant la distance entre un chiffre et son correspondant.

evaluate2 Fonction d'évaluation sommmant toutes les distances calculées.

7 Algorithme A*

L'algorithme A* fonctionne autour de deux ensembles *OUVERT* et *FERME*. Ces deux ensembles contiennent des éléments de la forme :

(état_courant, état_parent, valuation, profondeur)

parent Fonction extrayant le membre "parent" d'un élément d'*OUVERT* ou *FERME*.

valuation Fonction extrayant le membre "valuation" d'un élément d'*OUVERT* ou *FERME*.

profondeur Fonction extrayant le membre "profondeur" d'un élément d'*OUVERT* ou *FERME*.

is-solution Fonction déterminant si l'état donné est solution.

set-contains Fonction vérifiant d'un élément appartient à un ensemble. Si oui, l'élément complet est retourné.

add-to-ouvert-aux Fonction auxiliaire permettant d'insérer un élément au bon endroit dans *OUVERT*.

add-to-ouvert Fonction permettant d'insérer un élément au bon endroit dans *OUVERT*, par ordre croissant de valuation, puis par ordre croissant de profondeur.

show-path Fonction remontant de parent en parent afin d'afficher le chemin menant du problème de départ à la solution.

algo-a Fonction principale de l'algorithme A*. Des explications complémentaires sur le fonctionnement de l'algorithme sont données plus loin.

play Lancer le jeu !

Nous nous sommes basés sur le pseudo code suivant pour implémenter notre algorithme A* :

Listing 1 – Algorithme A*

```
1. OUVERT = { (état de départ) }
     avec parent(état de départ) = VIDE
          valuation(état de départ) = evaluer(état de départ)
          profondeur(état de départ) = 0
   FERME = VIDE

2. Si OUVERT est vide, sortir de l'algorithme (échec)

3. Retirer de OUVERT l'état ayant la somme valuation + profondeur la
   plus petite.
     L'appeler N
     Si N est un état d'arrivée, sortir (succés)
     Ajouter N à FERME

4. Pour tous les successeurs M de N, faire
     Si etat(M) n'est ni dans OUVERT ni dans FERME alors
        L'ajouter à ouvert
        Calculer sa profondeur
        Calculer sa valuation
     Fsi

     Si etat(M) appartient à OUVERT alors
        Si la profondeur de M dans OUVERT est supérieur à la
           profondeur de N + 1 alors
           Positionner la profondeur de M dans OUVERT à la
              profondeur de N + 1
```

```
            Rediriger le parent de M dans OUVERT vers N
        Fsi
    Fsi

    Si etat(M) appartient à FERME alors
        Si la profondeur de M dans FERME est supérieur à la
            profondeur de N + 1 alors
            Positionner la profondeur de M dans FERME à la profondeur
                de N + 1
            Rediriger le parent de M dans FERME vers N
        Fsi
    Fsi

5. Aller a 2
```

8 Problèmes rencontrés

Les règles de notre jeu sont relativement simples, toutefois, par rapport à un taquin classique de 4 cases par 4 cases, la complexité est nettement accrue. En fait, il est difficile de réaliser une fonction d'évaluation efficace.

Dans le taquin classique à 4 cases par 4, il est possible de savoir indépendamment pour chaque case si elle se trouve à la bonne place ou non. Dans notre problème cela n'est pas possible, car une case est à la bonne place si et seulement si elle est entourée des cases qui lui correspondent. La fonction d'évaluation est donc plus complexe à écrire, et ne reflète pas forcément l'avancement dans la résolution du problème.

Nous nous sommes aperçus que notre première fonction d'évaluation ne parvenait pas à estimer correctement la valuation d'un état du jeu : la moindre modification du plateau de jeu entrainait une enorme modification de la valuation, et lorsqu'une case se rapprochait d'une case correspondante (sans entrer directement en correspondance), cela n'influait pas sur la valuation. L'algorithme A* etait donc très peu guidé par cette fonction d'évaluation.

C'est la raison pour laquelle nous avons décidé d'écrire une deuxième fonction d'évaluation, tenant compte de la distance par rapport au nombre correspondant. Malheureusement, cette fonction d'évaluation s'est révélée pire que la précedente : elle engendrait des bouclages dans l'algorithme A*. En effet, lorsque la case vide se déplacait vers la droite (puisque c'était l'état le mieux noté), on recalculait les successeurs de l'état, et il se trouve que fréquemment le déplacement de la case vide vers la gauche était le meilleur

choix. La boucle est bouclée, le programme ne s'arrête plus.

Finalement, nous nous sommes rendus compte que le problème venait d'ailleurs. En effet, notre génération aléatoire d'un plateau de jeu à partir d'un jeu solutionnable ne fonctionnait pas correctement car elle interchangeait des cases sans se préoccuper de savoir si les mouvements étaient possibles. Ainsi, nous obtenions un plateau de jeu, parfois possible, parfois impossible à résoudre. Nous avons alors modifié notre fonction pour qu'elle génère des mouvements aléatoirement à partir des mouvements possibles. De plus, lors du lancement du jeu, l'utilisateur peut choisir le nombre de mouvements générés à partir du plateau de jeu solutionnable.

Par ailleurs, l'algorithme A* n'a pas été très simple à comprendre : les pseudos codes et implémentations que nous avons trouvés n'étaient pas toujours très clairs ni complets. Il a donc fallu étudier plusieurs codes avant de comprendre le fonctionnement précis de l'algorithme et pouvoir l'implémenter correctement.

Une fois que tout fonctionnait enfin, nous avons testé les deux fonctions d'évaluation, et la seconde fonction nous est parue comme étant la plus efficace, récompensant ainsi nos efforts à chercher une solution à un problème insoluble.

Conclusion

Ce projet nous a permis de nous familiariser largement avec le langage Lisp : nous avons pu perfectionner nos connaissances en récursivité et fonctions d'application, et découvrir quelques structures de contrôle itératives. D'autre part, nous avons appris à structurer un programme Lisp, langage bien différent des langages impératifs connus.

Le début du projet fut relativement rapide à réaliser (représentation du jeu, mouvements possibles, fonction d'évaluation, affichage), mais la fin du projet fut nettement plus complexe. D'une part parce que l'algorithme A* n'est pas simple à comprendre, et d'autre part parce que notre générateur aléatoire de taquins impossibles nous a bloqué un long moment (nous ne savions pas d'où venait le problème, nous étions persuadés que tous les taquins étaient solutionnables). Finalement, après moults rebondissements, l'affaire fut conclue.