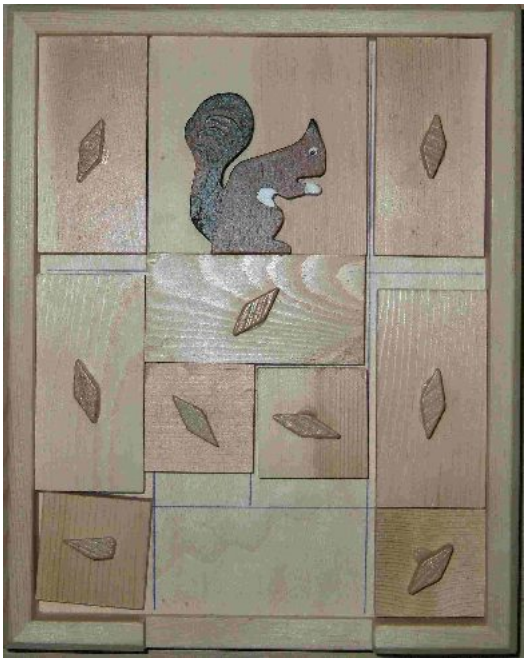
Projet de simulation : “L’écureuil dans sa cage”

22 janvier 2008

1 Introduction

C’est un jeu en bois : des pièces peuvent se déplacer dans un cadre. Parmi elles, un ecureuil. Il y a une sortie en bas du cadre. On doit déplacer les pièces (par translations), pour faire sortir l’écureuil de sa cage. Voici une photographie des pièces au départ du jeu :



Le but du projet est dans une première étape de faire un programme qui représente le jeu, et permette de jouer (de déplacer les pièces). Dans une deuxième étape, que l’ordinateur trouve lui même la solution. L’idée pour cette modélisation est que l’ordinateur doit créer un graphe : les sommets sont les configurations, et les arêtes sont les déplacements possibles entre deux configurations. A l’aide du programme, on pourra trouver la solution optimale (en 116 mouvements), mais aussi ce programme permettra ensuite de créer d’autres jeux similaires.

2 Première étape : modélisation du jeu

2.1 Conventions

Type : Afin de modéliser le jeu sur ordinateur, on va convenir d'un certain codage. Il y a 4 type de pièce. Voici le **type** correspondant :

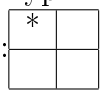
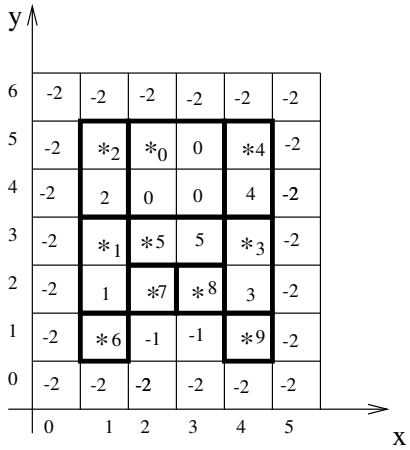
type 1 :  type 2 : type 3 : type 4 :

Tableau : On appelera un **tableau,** la répartition des pièces dans le système de coordonnées (x, y). Par exemple pour la configuration de départ, voici le tableau correspondant :



Dans la case x, y, le tableau contient une numéro de **pièce**, entre 0 et 9. Il contient (-1) si la case est vide. Il contient (-2) si la case est interdite (pour marquer les bords). Par convention, la **position** (x, y) d'une pièce est la position de la marque $*$ (coin supérieur gauche).

La liste d'une configuration : La **liste** d'une configuration est la liste des pièces avec leur type et leur position (x, y). Par exemple pour la configuration de départ, la liste est :

pièce	type	x	y
0	1	2	5
1	2	1	3
2	2	1	5
3	2	4	3
4	2	4	5
5	3	2	3
6	4	1	1
7	4	2	2
8	4	3	2
9	4	4	1

Comme la première colonne est inutile, on mémorise ce tableau par la liste des nombres obtenus par $(type, x, y)$. Soit dans cet exemple, la liste est :

$125, 213, 215, 243, 245, 323, 411, 422, 432, 441.$

Ordre de la liste : L'échange de deux pièce de même type ne change pas la configuration. Donc, pour que la façon d'écrire cette liste soit unique, on convient que la suite des nombres de la liste soit dans l'ordre croissant. C'est le cas dans l'exemple ci-dessus : $125 < 213 < 215 \ldots$.

Code d'une configuration : Pour coder de façon compacte une configuration, on conviendra de mettre bout à bout tous les nombres de cette liste. Le résultat appelé **code,** est dans l'exemple ci-dessus (configuration de départ) :

$code = 125213215243245323411422432441$

Ainsi si deux configurations ont le même code, elle sont identiques.

<u>Question :</u> quel est le code de la configuration finale ?

2.2 Consignes de programation

Ecrire les fonctions suivantes :

1. Fonction **code_to_liste() :**
   <u>en entrée :</u> le code d'une configuration.
   <u>en sortie :</u> la liste de cette configuration.
2. Fonction **liste_to_code() :**
   <u>en entrée :</u> la liste d'une configuration.
   <u>en sortie :</u> le code de cette configuration.
3. Fonction **Ordonne_liste() :**
   <u>en entrée :</u> la liste d'une configuration.
   <u>en sortie :</u> la même liste ordonnée dans l'ordre croissant.
4. Fonction **liste_to_tableau() :**
   <u>en entrée :</u> la liste d'une configuration.
   <u>en sortie :</u> le tableau $T[x, y]$ correspondant.
5. Fonction **Affiche_tableau()**
   <u>en entrée :</u> un tableau
   <u>en sortie :</u> affichage simple du tableau à l'écran, avec le numéro des pièces. (cette fonction sert à mettre au point le programme).
   (A ce moment on peut déjà tester les fonctions ci-dessus).
6. Fonction **Dessin_tableau()**
   <u>en entrée :</u> un tableau
   <u>en sortie :</u> dessin du tableau à l'écran.
7. Fonction **Test_bouge_pièce()**

en entrée : **code :** code d'une configation
pièce : le numéro d'une pièce que l'on souhaite bouger.
dir : la direction du déplacement souhaité (par exemple, 'E' : est, 'N' : nord, 'O' : ouest, 'S' : sud)

en sortie : accord ou non, si on peut bouger la pièce. Et si il y a accord, on renvoit le code (ordonnné) de la nouvelle configuration obtenue.
Dans l'exemple ci-dessus, d'après le tableau, si pièce=6, et dir='E', alors on peut bouger la pièce. Par contre, si pièce=8, et dir='O', on ne peut pas.

Voici l'algorithme détaillé de cette fonction :

(a) On crée la liste, et le tableau T de la configuration donnée.

(b) A partir de 'dir', on crée les variables dx, dy du déplacement souhaité. Par exemple, si dir="E', alors $dx = 1$, $dy = 0$.

(c) On initialise une variable : accord=1.

(d) On parcourt toutes les cases (x, y) du tableau. Si $T(x, y) = piece$, on regarde si $T(x + dx, y + dy)$ contient ni *piece* ni -1. Dans ce cas, on pose accord=0.

(e) Au final, si accord=0, on refuse.

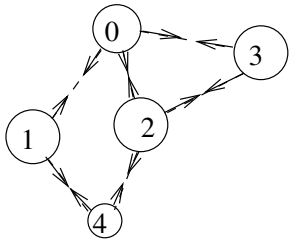
(f) Si accord=1, on crée la nouvelle liste, on changeant x, y de la pièce (c'est la ligne correspondante) par $(x + dx, y + dy)$. On ordonne cette liste, et renvoit le code correspondant.

8. **Programme principal :** Utilisant les fonctions précédantes, écrire un programme qui permettre de jouer grâce à l'ordinateur : le programme part de la configuration initiale, l'affiche, demande à l'utilisateur quelle pièce il veut bouger et dans quelle direction, et bouge si c'est possible, etc..

3 Deuxième étape : recherche de la solution

Pour cela on crée un graphe (dans la mémoire de l'ordinateur), dont les sommets sont les configurations, et les arêtes rejoignent deux configurations que l'on obtient par un déplacement élémentaire d'une pièce. L'ordinateur fera une exploration exhaustive de ce graphe.

Ce graphe est mémorisé sous la forme d'une liste et d'un dictionnaire. Par exemple pour le graphe :



La liste serait la suivante :

numéro	Code Config	
0	C_0	
1	C_1	
2	C_2	
3	C_3	
4	C_4	

et voici le dictionnaire :

	Code (clef)	liens
	C_0	$1, 2, 3$
	C_1	$0, 4$
	C_2	$0, 3, 4$
	C_3	$0, 2$
	C_4	$1, 2$

Les liens sont les numéros des configurations voisines.
Au début de la recherche, on part de la configuration de départ.

1. Faire une fonction **Cherche_config_voisines()**
   entrée : un code de configuration.
   sortie : liste des codes des configurations voisines, obtenues par mouvement élémentaire d'une pièce.
   Algorithme :
   (a) à partir du code de la configuration, on construit le tableau, et on trouve les deux cases vides (avec le numéro -1).
   (b) Pour chacune de ces cases vides, on parcourt les 4 directions (E,N,O,S) , et on considère la pièce voisine au trou dans cette direction. Avec la fonction **Test_bouge_pièce()**, on déduit si la pièce peut bouger vers le trou, (i.e. dans la direction inverse), et si oui, la nouvelle configuration.
2. Faire une fonction **Cree_graphe()**
   (a) Au départ, la liste et le dictionnaire sont vides. On met la configuration de départ C_0 dans la liste (numéro 0). Il y a un curseur qui pointe sur le numéro $n = 0$ (curseur=0).
   (b) On appelle la fonction **Cherche_config_voisines()** pour la configuration C au niveau du curseur. Cela donne les configurations voisines. On rajoute au dictionnaire les "liens", c'est à dire les numéros de ces configurations voisines. Et si elles ne sont pas déjà dans la liste, on rajoute ces configurations à la fin de la liste.
   (c) Test si la configuration C est la configuration finale recherchée (qui a un code précis). Si oui, on pose solution=curseur, pour la mémoriser.
   (d) On avance dans la liste : curseur=curseur+1, et on recommence à l'étape (b), jusqu'à ce que le curseur atteigne la fin de la liste.
3. Faire une fonction **Cherche_chemin()**
   entrée : n_1 =0, le numéro de la config de départ
   n_2 =solution, le numéro de la config finale.
   sortie : liste des numéros des configurations qui joignent n_1 à n_2 par le chemin le plus

court.
Algorithme :
Partie 1 : (où on assigne à chaque sommet, sa distance au sommet de départ). Ce sera une liste $dist\,[n]$, où n est le numéro du sommet. On aura aussi une liste des sommets qui sont à la distance d, notée `Liste_n[d]`. Dans l'exemple ci-dessus, `dist[]=[0,1,1,1,2]` et `Liste_n[]=[[0],[1,2,3],[4]]`.

(a) On crée le tableau $dist\,[n]$, remplit de -1, qui signifie que les distances ne sont pas encore calculées. On pose $d = 0$, et $dist\,[n_1] = d$, et Liste_n[d]=[n1].

(b) Grâce à la liste Liste_n, on parcourt tous les sommets n qui sont à la distance d.

(c) Pour chacun des ces sommets, on regarde ses voisins n'. Si ils ont $dist\,[n'] = -1$, on leur attribue $dist\,[n'] = d + 1$, et on les rajoute à la liste Liste_n[d+1].

(d) On passe à $d = d + 1$, et on recommence en (b), jusqu'à la fin.

–

–

Partie 2 :

(e) On part du sommet final $n = n_2$. On part d'une liste vide appelée **chemin.**

(f) On rajoute n dans la liste **chemin**. Ce sommet n est à une distance $d = dist\,[n_2]$.

(g) Grâce à la liste Liste_n, on cherche un sommet n' voisin de n qui est à la distance $d - 1$. On pose $n = n'$ et $d = d - 1$. On recommence en (f) jusqu'à atteindre $d = 0$.

A la fin on inverse la liste chemin.