

LICENCE INFORMATIQUE

RAPPORT DU PROJET MPIL

O'Rush Hour

3i008, Licence d'Informatique L3

REALISER PAR:
DJEDDAL Hanane.

2018/2019

INTRODUCTION

Rush Hour est un casse-tête dont le but est de faire sortir d'un terrain de jeu rectangulaire une voiture, alors que le chemin vers la sortie est encombré par d'autres véhicules.

Dans le cadre de ce projet, on considère une version maritime du jeu : on manipule des bateaux dans un port et on cherche à faire sortir le bateau des gardes-côtes.

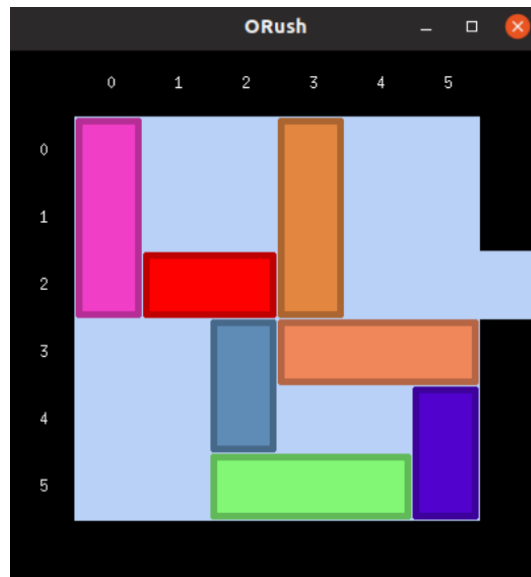
L'objectif est donc de trouver une modélisation du problème qui, étant donnée une configuration initiale, permet de retrouver une liste minimale de mouvements permettant de résoudre le problème, tout en exploitant les aspects fonctionnels et impératifs du Ocaml.

1. Présentation du problème:

Le jeu se déroule sur un plateau de 6 lignes et 6 colonnes (le port) avec la sortie sur la troisième ligne à droite.

Un bateau a un identifiant, une longueur de 2 ou bien 3, est soit orienté horizontalement soit verticalement (mais pas les deux) et peut être déplacé que d'avant ou en arrière.

Une configuration initiale définit les bateaux présents sur le plateau.



L'entrée est un fichier contenant une configuration initiale où chaque ligne est une représentation d'un bateau sous une chaîne de caractères de la forme:

IdentifiantLongueurOrientationLigneColonne

La sortie est une chaîne de caractères représentant la liste des mouvements à faire pour résoudre le problème, sous la forme:

BateauDéplacementBateauDéplacementBateauDéplacement... Avec Déplacement soit «<» pour reculer soit «>» pour avancer.

2. Modélisation du problème:

Pour résoudre le problème, on commence, d'abord, par représenter les notions statiques et dynamiques du jeu.

Le port, une grille de 6x6, passe d'un état à un autre selon la configuration des bateaux. Donc on définit:

- Une orientation: H (Horizontal) ou V (Vertical)
- Un bateau: un enregistrement contenant l'ensemble des informations concernant un bateau: son id, sa longueur, son orientation (H ou V) et ses coordonnées (x,y)
- Un Etat: une matrice de bateaux, où les valeurs des cases occupées par un bateau vaut ce bateau. Par exemple si A2H00 est une représentation du bateau 'b' alors, Etat[0,0]=b et Etat[0,1]=b

Ensuite, il faut modéliser l'aspect dynamique: les actions. Pour ça on définit un type move. Un bateau peut soit avancer soit reculer, et selon l'orientation du bateau on effectue ce déplacement (horizontalement ou verticalement) donc le déplacement dépend du bateau. Ainsi, move peut être A (avancer) ou bien R (reculer) en prenant le bateau concerné en paramètre.

3. Résolution du problème :

Pour la résolution, on utilise un parcours en largeur pour trouver la configuration gagnante.

En commençant par l'état initial, on génère tous les états possibles (les fils) en effectuant un seul déplacement à partir de cet état. Si parmi ces états, il y a une configuration gagnante, le jeu se termine et on retourne la chaîne des déplacements qu'on a construit au fur et à mesure. Sinon, on enfile ces états, et on recommence avec un nouvel état.

Cet algorithme, en explorant tous les états de niveau n (donc n déplacements) avant de passer au niveau $n+1$, permet de trouver la solution avec le minimum des déplacements possibles.

4. Architecture de l'application :

L'application, un mix du fonctionnel, impératif et modulaire, est composée de 3 modules qui manipulent des types communs (boat, state et move):

Port:

Permet de récupérer les données du jeu à partir d'un fichier, les transformer en types manipulables par l'application (de chaîne de caractères à un bateau ou state) et vice versa: imprimer les states et bateaux après les avoir convertis en chaîne de caractères

Moves:



Permet de générer des moves à partir d'une chaîne de caractères, tester si c'est possible de les appliquer (ne sort pas de la grille, ne chevauche pas avec d'autres bateaux et n'occupe pas des cases déjà occupées) puis les appliquer en générant des nouveaux états.

Solver:

C'est le module permettant de résoudre le jeu. Ici, on implémente en premier une structure de file qui va être utilisée dans le parcours en largeur.

- Les éléments de la file sont des couple (état, chaîne de caractères) où la chaîne de caractères représente la liste de mouvements résultant en cet état à partir de l'état initial.
- La fonction enfiler permet de rajouter un élément à la fin de la file s'il n'y est pas déjà.
- Pour tester l'existence d'un élément dans la file, on utilise une liste 'marked' qui permet de stocker les états déjà rencontrés sous forme d'une chaîne de caractères, puis on teste avec la fonction isMarked.

On utilise aussi une fonction all_possibles_moves, qui parcourt un State, teste chaque case et retourne la liste des déplacements possibles. La fonction all_reachable_states, utilise



une telle liste pour générer la liste des états possibles: chaque état correspond à l'application d'un déplacement de la liste à l'état passé en paramètre.

Finalement, à l'aide de ces fonctions, on implémente `solve_state` qui, étant donné un état initial, calcule les mouvements possibles, génère les états fils puis parcourt ces derniers pour les testers et les enfile. On réitère jusqu'à ce qu'on trouve une solution (exception `Fin`) ou bien la file est vide (pas de solution, exception `FileVide`).

5. Complexité de la solution:

L'utilisation du parcours en largeur entraîne que la complexité en temps vaut, au pire cas, $O(n)$ avec n le nombre d'états possibles. En effet, si la solution nécessite k déplacements et que k est le plus grand nombre de déplacements possible, alors l'algorithme visite $\sum_{i=1..k} n_i$ avec n_i le nombre des états générés à chaque niveau.

Le parcours en profondeur donne la même complexité, mais il ne garantit pas que la solution retournée est la meilleure en nombre de mouvements.

Quant à la complexité spatiale, avec l'implémentation adoptée, vaut aussi $O(n)$ au pire cas. Plus précisément $n \cdot (6 \cdot 6 \cdot b + m)$ avec b la taille d'un bateau et m la taille d'une chaîne de caractères. En effet, l'algorithme utilise une structure de file qui stocke les états visités avec une chaîne de caractères. Cette complexité est acceptable car la grille utilisée est d'une petite taille.

Il existe d'autre algorithme qui offre une meilleure complexité comme: A^* , Dijkstra...

CONCLUSION

En se basant sur le noyau fonctionnel d'Ocaml et un minimum de l'impératif, on a développé une application capable de résoudre la version simple du jeu Rush Hour dans un temps acceptable. Ça nous pousse à réfléchir à d'autres solutions qui, face à des variations du jeu plus difficiles, peuvent donner la même performance. Par exemple, en permettant à un bateau de déplacer sur la diagonale ou bien en utilisant plusieurs formes de bateaux.

