3I008 — Projet
ORush

Ariana Carnielli

Table des matières

1 Introduction

Ce projet s'intéresse à l'implémentation du jeu de casse-tête de déplacement *ORush*, une version du jeu *Rush Hour* avec des bateaux. Comme dans le jeu original, les pièces sont disposées sur une grille 6×6, orientées à l'horizontale ou à la verticale, et ne peuvent bouger que selon leur orientation. L'objectif du jeu est de ramener un bateau spécifique, nommé dans notre implémentation de `A`, à la sortie. Ce bateau est toujours de taille 2, orienté à l'horizontale et à la ligne $y = 2$, la sortie étant à l'extrémité droite de cette ligne. On repère un bateau par la coordonnée de son point le plus en haut à gauche.

Cette première partie du projet s'intéresse à représenter ce problème en OCaml en utilisant les principes de la programmation fonctionnelle, implémenter des fonctions permettant sa manipulation et, à la fin, implémenter un solveur permettant de résoudre une grille donnée. Le solveur est naïf et se base sur un parcours en largeur du graphe dont les sommets sont les états possibles du jeu, deux états étant reliés par une arête si et seulement si on peut passer d'un à l'autre par un mouvement élémentaire (un mouvement d'un seul bateau d'une seule case).

L'implémentation se décompose en trois parties. Dans un premier temps, on écrit, dans un fichier `port.ml`, la représentation d'un bateau et d'un état et des fonctions élémentaires permettant de le manipuler et l'afficher. On a ensuite écrit, dans le fichier `moves.ml`, la représentation des mouvements des bateaux et des fonctions permettant de les appliquer lorsque possible. Finalement, le fichier

`solver.ml` implémente le solveur. Un fichier complémentaire `orush.ml` donne un point d'entrée du programme, permettant de lire un fichier externe contenant un problème, le résoudre et afficher la grille aux états initial et final ainsi que la séquence de mouvements correspondant à la solution. Un fichier `Makefile` permet une compilation facile de l'ensemble du code.

2 Port

Le fichier `port.ml` contient les six fonctions et les deux types demandées pour que le module `Port` soit compatible avec la signature donnée dans l'énoncé. D'autres fonctions ont été codées afin de faciliter l'implémentation des fonctions de ce module et des suivants.

La représentation d'un bateau est faite à l'aide d'un enregistrement à cinq champs : deux champs de type caractère représentant le nom et l'orientation (`H` ou `V` pour horizontale ou verticale) du bateau et trois champs de type entier contenant la taille du bateau et les coordonnées de son point de référence. Cela permet facilement de convertir ce type en une chaine de caractères conforme à l'énoncé et vice-versa à l'aide des fonctions `boat_of_string` et `string_of_boat`.

La représentation d'un état du port est faite par un type `state` défini comme étant équivalent à `boat list`. Une implémentation alternative utilisant une autre représentation pour le `state` est décrite dans l'Appendice A.

Pour la fonction `add_boat`, la première étape implémentée à été de vérifier que le bateau `b` à rajouter se trouve bien dans la grille, ce qui est fait à l'aide d'une fonction auxiliaire nommée `boat_in_grid`. Ensuite, on parcourt la liste des bateaux déjà présents dans la grille et, pour chacun de ces bateaux, on vérifie s'il intersecte `b`. Pour cela, une fonction auxiliaire `list_case_of_boat` a été implémentée, permettant d'obtenir la liste des cases occupées par un bateau. Si l'ajout n'est pas possible, on lève une exception `Invalid_argument`.

La fonction `grid_of_state` commence en construisant une grille vide, représentée par des cases contenant le caractère ~. Ensuite, pour chaque bateau, on construit, à l'aide de la fonction `list_case_of_boat`, la liste des cases qu'il occupe, qui est ensuite parcourue en remplissant les cases correspondantes de la grille par la lettre représentant le bateau.

La première étape de la fonction `input_state` est de revenir au début du fichier lu. Ensuite, le fichier est parcouru ligne à ligne à l'aide d'une fonction récursive qui se termine lorsque l'exception `End_of_file` est levée. Pour chaque ligne, la fonction `boat_of_string` est utilisée pour convertir la ligne lue en `boat`.

Finalement, la fonction `output_state` imprime sur un canal de sortie une représentation de l'état en convertissant d'abord l'état en une grille de caractères à l'aide de la fonction `grid_of_state` avant de parcourir cette grille et l'afficher caractère par caractère.

3 Moves

Dans le fichier `moves.ml`, on a implémenté les cinq fonctions demandées dans l'énoncé pour correspondre à la signature donnée ainsi que le type pour représenter un mouvement et une exception pour signaler l'impossibilité d'un mouvement.

Le type `move` a été implémenté comme un enregistrement contenant la lettre représentant le bateau à bouger et un entier valant +1 ou -1 qui représente la direction du mouvement, vers l'avant ou l'arrière. Cela traduit fidèlement la représentation d'un mouvement par une chaine de deux caractères donnée dans l'énoncé et permet une implémentation immédiate des fonctions `string_of_move` et `move_of_string`.

Pour faciliter l'implémentation de la fonction `apply_move`, quelques fonctions ont été rajoutées à `port.ml`. Dans `apply_move`, on récupère d'abord le bateau correspondant au mouvement, ce qui se fait à l'aide de la fonction `get_boat` dans `Port`. Ensuite, on supprime ce bateau de l'état (à l'aide de la fonction `remove_boat`), on le bouge (en utilisant la fonction `move_boat`) et on le rajoute à nouveau. En cas d'une exception du type `Invalid_argument` lors de ce rajout, le mouvement tenté est impossible, et on lève ainsi une exception `Cannot_move`.

Pour tester la victoire, la fonction `win` cherche, à l'aide de `get_boat`, le bateau `A` et teste si sa position de référence est la case $(4, 2)$. Avec cela, on implémente la fonction `check_solution`. Elle commence d'abord par convertir la chaine de caractères passée en argument en une liste de mouvements à l'aide d'une fonction auxiliaire `list_move_of_string`. Elle parcourt ensuite cette liste en appliquant les mouvements un après l'autre dans l'ordre. Si elle rencontre une exception du type `Cannot_move` lors d'un déplacement, elle renvoie `false`. Sinon, à la fin de la liste de mouvements, on teste si l'état final est un état gagnant avec la fonction `win`.

4 Solver

Le module `Solver` implémente les quatre fonctions nécessaires pour satisfaire la signature donnée dans l'énonce. La fonction `all_possible_moves` utilise une fonction auxiliaire`all_moves` qui construit d'abord une liste de tous les mouvements, possibles ou impossibles, en prenant tous les bateaux et rajoutant à la liste les déplacements de +1 ou -1. La fonction `all_possible_moves` ne fait que filtrer cette liste en supprimant tous les mouvements pour lesquels une application de `apply_move` lève l'exception `Cannot_move`. Avec cette fonction, on implémente assez facilement `all_reachable_states` : il suffit d'utiliser un `List.map` pour appliquer à la liste de tous les mouvements possibles la fonction `apply_move`.

La partie principale du code, le solveur, est codé dans une fonction `solve_state` qui fait un parcours en largeur du graphe des états jusqu'à ce qu'un état gagnant soit trouvé (ou que tous les états aient été explorés sans trouver d'état gagnant). Pour l'implémentation classique du parcours en largeur, il faut garder en mémoire les états déjà visités pour éviter de repasser sur ces états et tomber sur une boucle infinie. Cela est fait à l'aide d'une table de hachage, uti-

lisant le module `Hashtbl`. Il se trouve que, en rajoutant dans cette table directement l'état (du type `state`), deux états identiques peuvent être considérés comme distincts à cause de la méthode de hachage utilisée internement par `Hashtbl`. En particulier, deux listes de bateaux contenant les mêmes bateaux dans un ordre différent représentent le même état, mais donnent des valeurs de hachage différents. Pour surmonter cette difficulté, on a transformé chaque état en une chaine de caractères de sorte que deux états sont égaux si et seulement si ces chaines sont égales. Cela est fait à l'aide de la fonction `string_of_state` implémentée dans `Port`, qui ne fait que transformer la représentation de l'état par une grille de caractères obtenue avec `grid_of_state` en une seule chaine.

On a aussi besoin, pour l'implémentation du solveur, d'une file contenant les prochains états à être examinés. Cette file a été implémentée à l'aide du module `Queue`. On y rajoute à chaque étape un couple contenant l'état en soi et la chaine de caractères représentant la suite des mouvements permettant d'aller de l'état initial à cet état, ce qui permet de retrouver assez facilement à la fin la chaine représentant la solution. Le parcours en soi est fait dans une sous-fonction récursive qui boucle tant que la file n'est pas vide, levant une exception `Not_found` si la file est vide, ce qui indique que tous les états ont été explorés sans trouver une solution.

Il s'agit d'un algorithme naïf qui ne fait pas d'heuristique cherchant à diminuer le nombre d'états à explorer. Sa complexité est en $O(n+m)$, où n est le nombre d'états possibles et m est le nombre d'arcs du graphe représentant les mouvements élémentaires entre deux états. On remarque qu'on a l'estimée grossière $n = O(5^b)$, où b est la quantité de bateaux : chaque bateau peut être à au maximum 5 positions différentes (avec moins de positions si sa taille est plus grande que 2). La quantité 5^b sur-estime le nombre d'états en ne prenant pas en compte le nombre de chevauchements. Comme dans tout graphe simple non-orienté, on a $m = O(n^2)$.

Une possibilité pour améliorer le temps d'exécution serait d'utiliser un autre parcours, comme celui de l'algorithme $A*$. Cet algorithme permet de donner une priorité aux états à explorer à partir du nombre de mouvements depuis l'état initial et une heuristique donnant une borne par le bas du nombre de mouvements nécessaires pour arriver à la solution. La difficulté est de trouver une bonne heuristique adaptée à ce problème. Une heuristique possible est de considérer le nombre de cases séparant le bateau `A` de sa position-cible $(4,2)$ en rajoutant une estimée du nombre de mouvements nécessaires pour dégager les bateaux entre `A` et sa position-cible (une estimée triviale étant d'au moins 1 mouvement par bateau).

Finalement, la fonction `solve_input` lit une grille à partir d'un fichier d'entrée à l'aide de `input_state` et calcule sa solution par `solve_state`.

A Implémentation alternative

Les fichiers `port_grille.ml`, `moves_grille.ml` et `solver_grille.ml` donnent une implémentation alternative utilisée dans un premier moment pour ce projet qui fait un choix d'implémentation différente pour `state`. En effet, cer-

taines fonctions sont plus simples à implémenter si le `state` est représenté par une grille de bateaux, ce qui est le cas évident de `grid_of_state`, `output_state`, `boat_in_grid`, ainsi que pour tester si un état correspond à une victoire ou pour tester la présence d'un bateau dans une case donnée. Cependant, cette représentation, bien que assez naturelle, rend aussi d'autres problèmes plus compliqués, comme celui de bouger un bateau. Ainsi, le choix fait dans cette implémentation alternative a été de stocker à la fois cette grille et la liste des bateaux. Si l'implémentation de la plupart des fonctions est alors plus directe, pouvant utiliser la représentation plus adaptée, cela a l'inconvénient de contenir de l'information dupliquée et d'occuper une place plus importante en mémoire. De plus, les fonctions modifiant un état doivent s'occuper de modifier les deux représentations.

Malgré ces inconvénients, l'implémentation donnée dans ces fichiers suit aussi les consignes de l'énoncé et permet de résoudre le jeu *ORush*. Il est à noter que la compilation de ces fichiers n'est pas faite par le fichier `Makefile`.