Projet de 3I008 (MPIL) - Partie 1
ORush

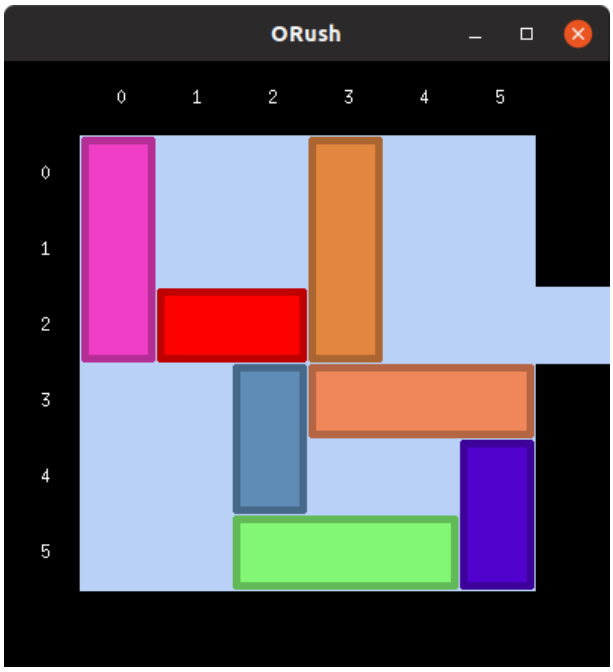
19 mars 2019

1 Présentation

Dans ce projet, nous allons réaliser un jeu de casse-tête de déplacement, version maritime du jeu *Rush Hour*, ou *Blocked In*, eux-mêmes dérivés des plus anciens jeux de *Taquin* ou *Âne rouge*.

Dans ce jeu, que nous nommerons ORush, plusieurs bateaux sont amarrés dans un port, dans des positions diverses et variées : certains bateaux étant orientés horizontalement, tandis que certains autres sont orientés verticalement. Le but du jeu est de permettre au bateau des gardes-côtes de sortir du port en déplaçant un-à-un un ensemble de bateaux afin de libérer le chemin vers la sortie (située à droite). Les bateaux, difficiles à manœuvrer dans un espace si restreint, ne pourront être déplacés que d'avant en arrière, et ne peuvent évidemment pas chevaucher d'autres bateaux.

En représentation simplifiée, et vue du dessus, voici un exemple de configuration initiale du port :



Les bateaux sont ici représentés par des rectangles colorés. La pièce rouge représente le bateau à libérer. La sortie correspond à case bleutée située le plus à droite. Pour que le bateau rouge puisse sortir, plusieurs autres bateaux devront alors être déplacés selon une certaine séquence de mouvements afin de libérer le chemin vers la sortie.

Instructions : Cette première partie du projet consiste à réaliser le moteur d'exécution et d'affichage textuel du jeu, ainsi qu'un *vérificateur* de solution pour le jeu, et un *solveur*. La partie 2, concernant l'interface graphique du projet, vous sera communiquée dans les semaines suivantes. Les différentes étapes de réalisation de cette première partie sont décrites dans la suite de ce document.

Rendu : Cette première partie du projet devra être envoyée par e-mail à l'équipe pédagogique comme indiqué sur la page du cours [1] sous la forme d'une archive `orush.tar.gz` contenant le code source de votre projet, ainsi qu'un `Makefile` permettant de le compiler (le programme devra être compilable avec la version OCaml de la PPTI, et ne pas dépendre de librairies externes non accessibles depuis les salles machines de l'université). L'archive devra également contenir un court rapport (au format PDF) décrivant vos principaux choix techniques (structures de données), une explication de votre algorithme de recherche de solution, ainsi que sa complexité.

2 Données du projet

2.1 Fichiers d'entrée

Un port est un espace quadrillé de dimension 6×6, qui peut contenir plusieurs bateaux. Nous vous fournissons plusieurs fichiers représentant à chaque fois l'état initial d'un jeu. Cet état correspond à la description ligne par ligne des différents bateaux positionnés dans le port.

Dans ces fichiers, un bateau est décrit par 5 caractères `ILOXY` :

1. Son identifiant `I` (un caractère de 'A' à 'Z'). La pièce à libérer sera toujours représentée par l'identifiant 'A'.
2. Sa longueur `L` (en nombre de cases : 2 ou 3)
3. Son orientation `O` (V - vertical , ou H - horizontal)
4. Les coordonnées `X` et `Y` de la case la plus haute et la plus à gauche de la pièce. [2]

Par exemple, le fichier suivant :

```
A2H12
B3V00
C2V42
```

FIGURE 1 – Représentation d'un état initial

représente un port contenant trois bateaux : A,B, et C.

1. Le bateau '**A**' est de longueur **2** (il prend deux cases sur le plateau), est positionné **H**orizontalement, et sa case la plus à gauche est placée en position (**1**, **2**).
2. Le bateau '**B**' est de longueur **3**, est positionné **V**erticalement, et sa case la plus haute est placée en position (**0**, **0**).
3. Le bateau '**C**' est de longueur **2**, est positionné **V**erticalement, et sa case la plus à gauche est placée en position (**4**, **2**).

2.2 Mouvements et représentation d'une solution

Un bateau peut *avancer* ou *reculer*. Ces actions sont à intérpréter différemment si la pièce est posée horizontalement ou verticalement :

— Une pièce horizontale avance d'une case vers la droite et recule d'une case vers la gauche.
— Une pièce verticale avance d'une case vers le bas et recule d'une case vers le haut.

Le bateau à libérer atteint la *position gagnante* dès lors que ses coordonnées sont (4, 2).

La figure 2 représente une séquence de déplacement permettant de libérer la pièce A vers la sortie.

1. `http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I008-2019fev/`
2. L'origine (0, 0) est la case située dans le coin supérieur gauche de la grille

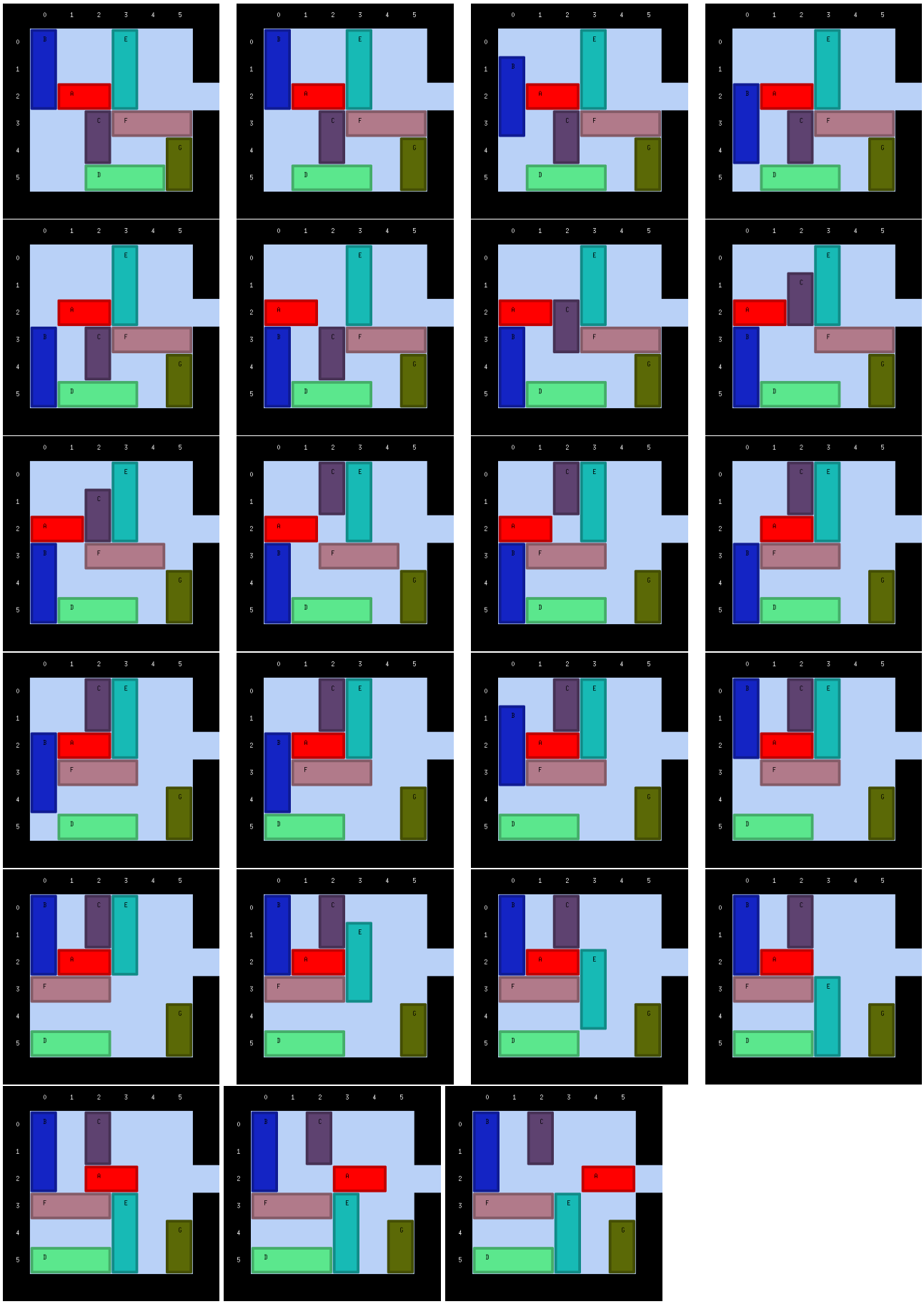


FIGURE 2 – Une solution

2.2.1 Représentation d'une séquence de mouvements

Un mouvement sera représenté par une chaîne de deux caractères, le premier indiquant l'identifiant du bateau à déplacer, et le second un caractère '>' pour indiquer qu'il avance d'une case ou '<' pour indiquer qu'il recule d'une case. Par exemple, "A>" signifie que le bateau A avance.

Une séquence de mouvements sera représentée sous la forme d'une chaîne issue d'une concaténation de mouvements. Par exemple, la solution représentée dans la figure 2 correspond à la chaîne de caractères suivante :

```
"D<B>B>A<C<C<F<C<F<A>B<D<B<B<F<E>E>E>A>A>A>"
```

3 Instructions et détails d'implémentation

Votre projet sera constitué de plusieurs modules, chacun représenté par un fichier du nom de ce module, qui correspondront à des aspects distincts du jeu (représentation du port, mouvements, solveur, ...). En raison de l'utilisation d'une méthode de correction semi-automatique des projets, votre implémentation devra respecter **scrupuleusement** la structure détaillée dans cette section (en particulier les signatures de chaque module devront être compatibles avec celles du sujet).

Attention : *Ces signatures détaillent simplement l'ensemble minimal des types et fonctions qui **doivent** être définis. Elles ne vous empêchent pas en revanche de réaliser, si besoin, d'autres définitions de types ou fonctions auxiliaires à l'intérieur de chaque module.*

3.1 Création et affichage d'un port

Dans un premier temps, vous devrez réaliser, à partir de la représentation décrite précédemment, la création et l'affichage (textuel) du port.

Pour ce faire, vous définirez un module `Port` (donc un fichier `port.ml`), qui représentera le plateau et les pièces du jeu (les bateaux). Un état du port sera représenté par un type `state`, et un bateau par un type `boat`, que vous définirez de la façon dont vous le désirez.

Ce module devra être compatible avec la signature suivante :

```
module type PORT = sig
  type boat
  type state

  (* transforme une chaine (ex : ''A2H45'') en une valeur de type boat *)
  val boat_of_string : string -> boat

  (* represente un bateau sous la forme definie dans le sujet (ex : ''
     A2H45'') *)
  val string_of_boat : boat -> string

  (* add_boat c s ajoute a l'etat s un nouveau bateau c. Leve
     Invalid_argument ''add_boat'' si le bateau ne peut etre mis a cette
     position *)
  val add_boat : boat -> state -> state

  (* retourne une representation matricielle d'un etat du port*)
  val grid_of_state : state -> char array array

  (* transforme une representation de l'etat initial du port depuis un
     canal d'entree vers une valeur de type state *)
```

```
  val input_state : in_channel -> state

  (* imprime une representation matricielle d'un etat du port sur un
     canal de sortie *)
  val output_state : state -> out_channel -> unit

end
```

En particulier, la fonction `grid_of_state` devra représenter le plateau de jeu sous une forme matricielle dans laquelle chaque case occupée par un bateau est représentée par le nom de ce dernier, et chaque case vide est représentée par une tilde ('~'). Par exemple, l'application de la fonction `grid_of_state` sur l'état du port issu du fichier de la figure 1 devra produire la matrice suivante :

B	~	~	~	~	~
B	~	~	~	~	~
B	A	A	~	C	~
~	~	~	~	C	~
~	~	~	~	~	~
~	~	~	~	~	~

La fonction `output_state` suivra cette même représentation pour produire la sortie suivante (dans le canal adéquat) :

```
B~~~~~
B~~~~~
BAA~C~
~~~~C~
~~~~~~
~~~~~~
```

3.2 Mouvements et vérification d'une solution

Vous devez ensuite réaliser les fonctions nécessaires aux mouvements des bateaux sur une grille, ainsi qu'à la vérification de la correction d'une solution donnée (sous la forme décrite préalablement).

Vous réaliserez alors un module `Moves` (donc un fichier `moves.ml`) implantant ces fonctionnalités. Vous définirez en particulier un type `move` ainsi qu'une fonction permettant de traduire un `move` vers une chaîne de caractère dans la forme décrite dans la section précédente (par exemple `"A>"` ou `"B<"`) .

Ce module devra être compatible avec la signature suivante :

```
module type MOVES = sig

  type move

  exception Cannot_move

  (* convertit un move en chaine de caracteres *)
  val string_of_move : move -> string

  (* convertit une chaine de caracteres en move *)
  val move_of_string : string -> move
```

```
  (* apply_move m s applique le mouvement m depuis l'etat s et retourne
     le nouvel etat. Leve l'exception Cannot_move si le mouvement est
     impossible *)
  val apply_move : move -> Port.state -> Port.state

  (* indique si un etat est une position gagnante *)
  val win : Port.state -> bool

  (* verifie si une suite de mouvements depuis l'etat en parametre est
     une solution *)
  val check_solution : Port.state -> string -> bool

end
```

3.3 Calcul d'une solution

Enfin, nous vous demandons de réaliser un *solveur* capable de trouver une séquence de mouvements permettant d'atteindre une solution.

Pour ce faire, votre programme devra être capable de calculer tous les mouvements possibles à partir d'un état du jeu.

Par exemple, depuis l'état suivant :

```
B~FFF~
B~~G~K
AA~GIK
CCC~IK
~~E~JJ
DDEHH~
```

Il y a 9 mouvements possibles : A peut avancer ; C peut avancer ; F peut avancer ou reculer ; G peut avancer ; H peut avancer ; I peut reculer ; J peut reculer et K peut reculer. Avec cette position initiale, votre programme doit alors être capable de calculer l'ensemble de mouvements suivant :

```
{ A> ; C> ; F> ; F< ; G> ; H> ; I< ; J< ; K< }
```

Une fois ces mouvements connus, on peut les appliquer afin de calculer l'ensemble des états résultant de ces mouvements. Il suffit ensuite d'itérer ce calcul sur tous les états résultants de tous les mouvements (i.e. re-calculer à chaque fois l'ensemble des états atteignables à partir de ces états) pour finir à une position gagnante.

Cette méthode est une méthode de force brute : elle explore l'ensemble des possibilités. Il y a au moins deux manières d'appliquer cette méthode : en profondeur ou en largeur. On préfèrera ici la manière en largeur : elle donnera en premier une solution parmi les plus courtes. Naturellement, il sera inutile de parcourir les chemins issus de positions déjà explorées.

N.B : *Vous êtes libre d'implanter un autre algorithme pour trouver la solution. Dans ce cas, vous le comparerez dans votre rapport avec celui proposé dans ce sujet.*

La solution calculée devra être représentée par une chaîne de caractères représentant une séquence de mouvements, comme indiqué précédemment.

Vous définirez donc un module `Solver` permettant de trouver une solution.

Ce module devra être compatible avec la signature suivante :

```
module type SOLVER = sig

 (* calcul de tous les mouvements possibles depuis un etat *)
 all_possible_moves : Port.state -> Moves.move list

 (* calcul de tous les etats atteignables depuis un etat et une liste de
    move *)
 all_reachable_states : Port.state -> Moves.move list -> Port.state list

 (* calcul d'une solution a partir d'un etat *)
 solve_state : Port.state -> string

 (* calcul d'une solution a partir d'un canal d'entree qui contient une
    representation de l'etat initial *)
 solve_input : in_channel -> string

end
```