Rapport projet Hashiwo kakero

Baudot Pierre – Nguyen Danh

Chargé de TP : Anthony Perez

I – Méthode de résolution

Pour résoudre un puzzle donné, nous avons décidé de d'abord convertir la variable de type puzzle vers le type solver_puzzle, qui n'est rien d'autre que le type solution a la différence près que pour les îles on stocke deux information au lieu d'une:

- la 1ère est l'importance initiale de l'île, c.a.d. l'importance que l'île a avant de commencer a résoudre le puzzle(c'est cette valeur qui sera conservée lors de la conversion vers le type solution).
- La 2ème est l'importance résiduelle, c.a.d. le nombre de ponts restants a construire a partir de cette île

Ainsi dans la matrice obtenue on va utiliser les fonctions get_neighbors (qui permet de récupérer la coordonnée et l'importance résiduelle des voisins d'une île encore non connecté par un pont avec cette île), build_bridge (permettant de construire un pont entre deux îles), is_connected (qui permet de savoir si le réseau de ponts est connexe) et is_solved (permettant de savoir si l'importance résiduelle de toutes les îles est a zéro) qui vont nous permettre d'appliquer plus facilement d'implémenter les fonctions pour appliquer les astuces 1 et 2 pour finir par la fonction solve.

Pour voir quelques cas de tests vous reporter aux fichier test.ml dans le dossier src et le fichier résultats_attendus_des_tests.txt et en exécutant le fichier nommé test qui apparaît après avoir compilé le programme via le fichier compile.sh (clean.sh permet de retirer les fichiers produits par la compilation).

II – Description plus détaillée de l'outil

Les fonctions implémentées par Danh sont surlignées en jaune, tandis que les fonctions implémentés par Pierre sont surlignées en vert.

module ListExtension

Fonctions additionnelles pour le module List

```
val isplit : int -> 'a list -> 'a list * 'a list
```

`isplit i l` divise l en deux parties et renvoie le résultat sous forme de couple dont le 1er élément est la sous liste de l composé de tout les éléments dont l'indice est inférieur a i exclu, le 2ème élément est le reste de la liste

```
val tail_fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Cette fonction a le même effet que fold_left mais est écrite en récursif terminal.

```
val index_of : ('a -> bool) -> 'a list -> int
```

index_of p l donne l'indice de la 1ère occurrence satisfaisant le prédicat p dans l.

```
val all_index_of : ('a -> bool) -> 'a list -> int list
```

all_index_of p l donne une liste contenant les indices de toutes le éléments de l satisfaisant le prédicat p.

```
val apply_to_nth : ('a -> 'a) -> int -> 'a list -> 'a list
```
apply_to_nth op n l remplace le n ième élément de l par le résultat de op appliqué a cet n ième élément.

module Coordinates

```
type coord = int * int
```

```
val coord_comp : coord -> coord -> int
```
Comparateur de coordonnées, coord_comp (x,y) (u,v) renvoie 1 si x > u ou si (x = u et y > v), 0 si (x,y) = (u,v), -1 sinon.
Exemples : (1,4) < (2,4), (4,2) < (4,3), (1,1) < (2,2), …

module Matrix

Fonctions permettant de faciliter les opérations sur les variables de type ‘a list list
Dans notre cas, les coordonnées sont agencés comme sur l’exemple suivant :

(1,1)	(2,1)	(3,1)
(2,1)	(2,2)	(3,2)
(3,1)	(3,2)	(3,3)

Pour les fonctions de ce module toutes les sous listes de la liste principale doivent obligatoirement avoir la même taille, des erreur pourraient être levés ou des résultats inattendus pourraient être produis sinon.

```
val get_elt : Coordinates.coord -> 'a list list -> 'a
```
get_elt (x,y) mat renvoie l’élément se trouvant aux coordonnées (x,y) dans mat.

```
val size : 'a list list -> int * int
```
size mat renvoie un couple (a,b) ou b est la longueur en abscisse de mat et b est la longueur en ordonnée de mat.

```
val for_all : ('a -> bool) -> 'a list list -> bool
```
for_all p mat renvoie true si le prédicat p renvoie true pour chaque élément de mat , false sinon.

```
val tail_fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list list -> 'a
```
tail_fold_left op neutral mat est équivalent List.fold_left op neutral (List.flatten mat) mais en plus efficace et en récursif terminal.

```
val coords_of : ('a -> bool) -> 'a list list -> Coordinates.coord option
```
coords_of p mat renvoie les coordonnées (x,y) du 1er élément de mat satisfaisant p.

```
val all_coords_of : ('a -> bool) -> 'a list list -> Coordinates.coord list
```
all_coords_of p mat renvoie la liste de toutes les coordonnées (x,y) des éléments de mat satisfaisant p.

```
val apply_to_coord :
      ('a -> 'a) -> Coordinates.coord -> 'a list list -> 'a list list
```
apply_to_coord op (x,y) mat remplace dans mat la valeur aux coordonnées (x,y) par la valeur du résultat de op appliqué a l’élément aux coordonnées (x,y) et retourne le résultat.

```
val matrix_of_coord_list : 'a -> (Coordinates.coord * 'b) list -> 'a list list
```
matrix_of_coord_list neutral clist convertis clist en matrice. cliste qui est une liste d'association ou les clés sont les coordonnées auxquels placer les valeurs associées dans la matrice résultante. Toutes les cases de la matrice ne correspondant pas a une valeur dans clist sont remplies avec la valeur neutral.

Exemple : matrix_of_coord_list 0 [((1,1),6); ((1,1),6);((1,1),6)] donne la matrice suivante :

6	0	0
0	6	0
0	0	6

```
val string_of_matrix : ('a -> string) -> 'a list list -> string
```
string_of_matrix op mat convertis mat en chaine de caractère, op est l'opérateur permettant de passer du type 'a au type string pour convertir chaque élément de la matrice.

module Hkakero

module comportant les outils nécessaires a la manipulation et la résolution d'un puzzle du jeu hashiwo hakero.

Types

```
type imp = int
type puzzle = (coord * imp) list
type bridge = {isVertical : bool; isDoubled : bool}
type cell = Nothing | Island of imp | Bridge of bridge
type solution = cell list list
```

```
type solver_cell = SNothing | SIsland of imp * imp | SBridge of bridge
```
Ce type a le même but que solution a la différence que pour les îles on stocke deux informations au lieu d'une : la 1ère est l'importance initiale de l'île, c.a.d. l'importance que l'île a avant de commencer a résoudre le puzzle(c'est cette valeur qui sera conservée lors de la conversion vers le type solution). La 2ème est l'importance résiduelle, c.a.d. le nombre de ponts restants a construire a partir de cette île.

```
type solver_puzzle = solver_cell list list
```
Matrice servant d'intermédiaire pour résoudre un puzzle.

Exceptions

```
exception Unsolvable_puzzle
```
Exception levée dans le cas ou le puzzle que l'on essaye de résoudre ne comporte pas de solution.

```
exception Insufficient_island_importance
```
Exception levée dans le cas ou l'on veut construire un pont et que l'une des îles cibles n'a pas une importance résiduelle suffisante.

Fonctions

```
val solver_puzzle_of_puzzle : puzzle -> solver_puzzle
```
solver_puzzle_of_puzzle puz convertis puz en matrice. Chaque case de cette matrice de coordonnées (x,y) comporte soit la valeur SIlsand (imp,imp) qui représente l'île d'importance imp ayant pour coordonnés (x,y) dans la liste puz, soit la valeur Snothing si il n'y a aucune île aux coordonnées (x,y).

```
val solution_of_solver_puzzle : solver_puzzle -> solution
```
solution_of_solver_puzzle sp convertis sp en solution, les cases SNothing deviennent Nothing, les cases SBridge b deviennent Bridge b et les cases SIsland (init_imp, imp) deviennent Island init_imp.

```
val string_of_cell : cell -> string
```
Conversion de type cell en string.

```
val string_of_solution : solution -> string
```
Conversion de solution en string destinée a l'affichage.

```
val get_neighbors :
      solver_puzzle -> Coordinates.coord -> (Coordinates.coord * imp) list
```
get_neighbors s_puz (x,y) donne une liste de couple, où chaque couple (c,imp) correspond aux coordonnées c et a l'importance résiduelle imp d'un voisin de l'île situé aux coordonnées (x,y) dans s_puz. (ne donne que les voisins avec lesquelles l'île aux coordonnées (x,y) n'est pas relié avec un pont)

```
val build_bridge :
 solver_puzzle ->
 Coordinates.coord -> Coordinates.coord -> bool -> solver_puzzle
```
build_bridge s_puz (x,y) (u,v) is_doubled construit dans s_puz un pont allant de l'île aux coordonnées (x,y) à l'île aux coordonnées (u,v), met a jour l'importance résiduelle des îles concernées et renvoie le solver_puzzle résultant. Construit un pont double si is_doubled est a true, un pont simple sinon.
Leve Failure si il n'y a pas d'île aux coordonnées (x,y) ou (u,v), si on essaye de construire un pont par dessus un autre pont ou une île ou `Insufficient_island_importance` si l'importance d'une des îles choisies n'est pas suffisante.

```
val nb_of_island : solver_puzzle -> int
```
Donne le nombre d'îles dans le solver_puzzle passé en paramètre.

```
val is_connected : solver_puzzle -> bool
```
Teste la connexité du réseau de pond du solver_puzzle passé en paramètre, c.a.d renvoie true si toutes les îles sont reliés entre elles, false sinon.

```
val is_solved : solver_puzzle -> bool
```
Renvoie true si l'importance résiduelle de toutes les îles dans le solver_puzzle en paramètre sont a zéro, false sinon.

```
val apply_tip_one : solver_puzzle -> solver_puzzle
```
Applique plusieurs fois l'astuce 1 au solver_puzzle passé en paramètre jusqu'à ce qu'elle ne permete plus d'avancer dans la résolution du puzzle.
Leve Unsolvable_puzzle si le puzzle n'a pas de solutions.

```
val apply_tip_two : solver_puzzle -> solver_puzzle
```
Applique l’astuce 2 au solver_puzzle en paramètre puis applique l’astuce 1 pour vérifier si le bon pont a été construit a partir de l’île choisie, si cela abouti a une exception Unsolvable_puzzle, on réessaye en construisant un autre pont vers un autre voisin a partir de l’île choisie.
Si l’application de l’astuce 1 mène a une situation de blocage on réapplique l’astuce 2 et ainsi de suite jusqu’à trouver la solution finale.
Leve Unsolvable_puzzle si le puzzle n’a pas de solutions.

```
val solve : puzzle -> solution
```
Résout le puzzle passé en paramètre.
Leve Unsolvable_puzzle si le puzzle n’a pas de solutions.