Devoir programmation fonctionnelle

Julien Van Den Bossche/Benoît Moulin

PARTIE 1 : Recherche heuristique

1. Concept du robot

Data Etat = (Case Integer Integer)
Un état représente une case de la grille. Un état aura deux coordonnées qui définissent sa position sur la grille.

etatInitial = (Case 1 1)
Ici notre état initial est la case en haut, à gauche sur la grille de jeu. On peut choisir un autre état initial si on le souhaite.

etatFinal= (Case n n) où n est la largeur et hauteur de la grille (grille carré dans notre cas). L'état final ici est donc la case en bas, à droite. On peut choisir une autre case si on le souhaite.

obstacles = [Case 2 2, Case 3 3, Case 4 2, Case 3 5]
obstacles est une liste d'états, chaque état étant un obstacle.

estTerminal :: Etat -> Bool
On compare l'état pris par la fonction avec l'état terminal. Si ce sont les mêmes on renvoi un booléen true sinon false.

estObstacle :: Etat -> Bool
On regarde si l'état en question appartient à la liste des obstacles (avec la fonction *elem*)

2. La fonction successeurs

Cette fonction doit prendre un état et doit renvoyer ses successeurs possibles. On doit donc regarder si les états à droite, à gauche, en haut et en bas de l'état courant sont des états possibles pour un éventuel déplacement. On doit donc regarder si l'état sur lequel on souhaite se déplacer est un obstacle ou bien s'il appartient encore à la grille.
Pour les obstacles on utilise la fonction estObstacle. Pour la validité de la case on regarde si ses indices sont supérieurs à 1 et inférieur à la taille de la grille.

3. L'algorithme A*

Pour les heuristiques h1 et h2 on a appliqué les définitions de la distance de Manathan et la distance euclidienne.

```
h1 :: Etat -> Integer
h1 (Case x y) = abs(x-n)+abs(y-n)
h2 :: Etat -> Integer
h2 (Case x y) = floor(sqrt(fromInteger (sommeCarre)))
        where sommeCarre = ((x-n)*(x-n)) + ((y-n)*(y-n))
```

Pour définir l'algorithme A* on utilise un file à priorité. Les éléments qui sont rangés dans la file par heuristique croissante. La tête de la file contiendra l'état qui à la plus petite heuristique. On utilise les heuristiques *h1* et *h2*.

La file appelée Queue est une liste d'éléments de n'importe quels types.
type Queue a =[a]

emptyQueue permet de créer une file vide.
emptyQueue = []

enQueue a q permet d'insérer un élément a dans une queue: l'élément en tête de la queue est celui pour lequel la valeur de l'heuristique est la plus petite.
Si la file est vide on insère l'état dans la file.
Si le nouvel élément à une heuristique plus petite que la tête de la file on l'insère en tête de la file sinon on cherche à l'insérer après la tête de la file.

isEmptyQueue :: Queue a -> Bool, teste si une file est vide

front :: Queue a -> a, permet de renvoyer le premier élément de la file.

deQueue :: Queue a -> Queue a, renvoie la file sans son élément de tête.

search :: Etat -> [Etat] permet de renvoyer la liste des états qui permettent d'aller d'un état à l'état terminal.
On ne doit pas retourner dans un état déjà visité. Si l'état est un obstacle on ne doit pas y aller.
-Si la tête de la file est un obstacle ou est déjà visité alors on insère cet élément dans la liste des états déjà visités puis on enlève cet élément de la file et on recommence avec le nouvel état de tête de file.
-Sinon on ajoute l'état de tête de file à la liste des états qui formeront le meilleur chemin et on l'enlève de la file puis on insère les successeurs de cet état dans la file par ordre d'heuristique croissante. On ajoute cet état dans la liste des états visités. On repart donc avec une file contenant en tête le meilleur état successeur de l'état que l'on vient de visiter.
On s'arrête quand la tête de la file est l'état terminal.
On obtiendra donc le meilleur chemin pour aller de notre état à l'état final selon notre heuristique.

4. L'heuristique h3

h3 :: Etat -> Integer
Cette heuristique doit tenir compte des obstacles. Pour ce faire on regarde si l'état est un obstacle, si c'est le cas on renvoie un nombre très grand (ici on peut prendre n*n, n étant la taille de la grille). Si ce n'est pas un obstacle on renvoie la distance de Manathan.
De ce fait, dans l'algorithme A* les obstacles seront mis en fin de file et seront donc évités.

5. Gestion de plusieurs états finaux

Si l'on choisit d'aller à un état final précis parmi plusieurs états finaux, les fonctions qui sont ci-dessus restent valables, il faut juste prendre en paramètre le nouvel état final.

```
search2 :: Etat -> Etat -> [Etat]
search2 e fin = bfs (enQueue e emptyQueue)[]
        where
        bfs q visit
                |isEmptyQueue q = []
                |elem (front q) visit || estObstacle (front q) = bfs (deQueue q) visit
                |fin == (front q) = [front q]
                |otherwise = (front q) : (bfs (foldr enQueue (deQueue q) (successeurs (front
q))) ((front q) : visit))
```

Si l'on doit parcourir tous les états finaux alors on définit une fonction search3 qui prend une liste d'états finaux et qui appelle la fonction search2. On parcourt la liste des états finaux et on applique search2 pour chaque état final. Quand la liste des états finaux à parcourir est vide on a le meilleur chemin pour parcourir tous les états. Quand on change d'état final on vide la liste des états déjà visités.

```
search3 :: Etat -> [Etat] -> [Etat]
search (Case x y) []  listeEtat =  listeEtat
search (Case x y) (z:zs) listeEtat =  search3 z zs ((search2 (Case x y) z) :listeEtat)
```

Problèmes rencontrés

Avec l'heuristique h2 on s'aperçoit, lorsque l'on part de l'état initial, que le robot va dans l'état (Case 3 1) puis dans l'état (Case 3 2). A ce moment là il entouré d'obstacles sauf en l'état (Case 3 1). Cet état étant déjà visité le robot passe à l'état suivant dans la file à priorité. L'état suivant dans la file à priorité à ce moment est l'état (Case 4 1) qui est un successeur de l'état (Case 3 1) et qui n'a pas encore était visité. Le robot passe donc dans l'état (Case 4 1). On pourrait croire quand on regarde le résultat que le robot regarde ses successeurs aussi en diagonal mais ce n'est pas le cas.
Pour que ce problème ne se pose pas il faudrait enlevé l'état (Case 3 1) des états visités puis mettre l'état (Case 3 2) dan les états visités. Pour généraliser on pourrait appliquer ce principe dès que l'état successeur est entouré de trois obstacles.

PARTIE 2 : génération de plan

1. Formalisation

Pour Etat, *etatInitial, etatFinal, estTerminal, estObstacle* on reprend ce que l'on a défini dans la partie 1.

Les actions correspondent au déplacement d'un état vers un autre avec une direction donnée (G pour aller à gauche, D pour aller à droite, H pour aller en haut et B pour aller en bas).
data Action = G | D | H | B

La fonction transition permet de passer d'un état à un autre par une action.

Si la transition amène à un état obstacle ou hors de la grille on renvoie le même état sinon on renvoie le nouvel l'état.
transition :: Etat -> Action -> Etat

deja_Visite :: Etat -> [Etat] -> Bool
Permet de savoir si un état a déjà été visité. On regarde si l'état qui nous intéresse appartient à la liste des états déjà visités.

2. Génération de plan

Dans cette question on doit renvoyer une liste d'action qui mène d'un état initial à un état final sans se préoccuper de savoir si le chemin emprunté est le moins coûteux.
Nous avons défini une fonction *estPossible :: Etat -> Action -> Bool* qui permet de savoir si une action est possible (si le déplacement peut se faire).
On regarde tout d'abord si on peut aller à droite, puis en bas, puis en haut, puis à gauche et on regarde si l'état destination n'a pas déjà été visité.
Dès qu'un de ces tests est vrai on se déplace dans l'état par l'action retenue et on insère l'action dans la liste des actions à retenir. On place l'état dans la liste des états visités. On recommence le test avec le nouvel état.

3. Génération de plan optimisé

On a vu que dans *genPlan* on choisissait la première action possible sans se soucier de savoir si elle était la meilleure. Ici on va regarder pour chaque transition possible l'heuristique de l'état destination de chaque transition puis on retiendra l'état destination qui à la plus petite heuristique.
Pour ce faire on définit la liste des actions qui seront parcourues :
listeAction :: [Action]
listeAction = [G,D,B,H]

On définit une fonction *casePossible* qui renvoie les états possible après avoir appliqué toutes les actions possibles. Par exemple pour l'état *(Case 1 1)* les états possibles après avoir appliqué les transitions sont *(Case 1 2) et (Case 2 1)*.

On définit une fonction *chercheAction* qui renvoie l'action qui après avoir été appliqué à un état initial donne un état avec qui a pour valeur heuristique valeurHeuristique. heuri est la fonction heuristique que l'on choisit.
chercheAction :: Etat -> (Etat -> Integer) -> Integer -> Action
chercheAction (Case x y) heuri valeurHeuristique

On définit la fonction *heuriMin* qui permet de renvoyer la plus petite valeur heuristique des états successeurs de l'état donné après avoir appliqué toutes les actions possibles.
heuriMin :: Etat -> (Etat -> Integer) -> Integer

La fonction *meilleur* utilise donc toutes ces fonctions et choisit pour chaque état la meilleure action menant à un état successeur ayant la valeur heuristique la plus faible.

Problèmes rencontrés

Lorsque l'on part de l'état initial avec la fonction *meilleur* et l'heuristique *h2* on tombe dans l'état (Case 3 2). Comme en (Case 3 3) on a un obstacle, on est obligé de remonter en (Case 3 1). Une fois dans cet état on repart dans l'état (Case 3 2) car cet état a la meilleure valeur heuristique pour aller à l'état final et on ne s'occupe pas des états déjà visités.
Le programme ne termine pas.

Pour résoudre le problème on devrait ajouter une liste des états visités et procéder comme pour le problème posé dans la partie 1.