Algorithme A Star

Lecavelier Pierre - Laurence Adrien

18/01/2007

Table des matières

1 Algorithme A Star standard

1.1 Introduction

Le but de notre algorithme A* est de trouver le chemin le plus court d'une position de départ à une position d'arrivée dans un labyrinthe. Il prend donc en entrée :

- un labyrinthe
- un état initial
- un état final

Et il va appliquer à chaque état intermédiaire une évaluation heuristique tel que :

- f = g + h l'évaluation heuristique
- g l'évaluation du cout pour aller de l'état initial à l'état courant
- h l'évaluation du cout pour aller de l'état courant à l'état final selon un critère

Puis enfin, il va visiter les états par ordre de cette évaluation heuristique jusqu'à arriver à l'état final.

1.2 Opérations initiales

Pour implémenter l'algorithme A* standard en Haskell, il faut effectuer un certain nombre d'étapes déjà vues en cours que nous n'allons donc pas expliquer dans les détails.

Il faut d'abord définir les types et les structures de données utilisés :

- ADT file à priorité : Celle-ci se comporte comme une file classique à l'exception qu'elle trie les éléments suivant un opérateur passé en paramètre.
- Type Labyrinthe : On le représente par une largeur, une hauteur et une liste d'obstacles (positions).
- Type Etat : Il est composé d'une position et d'un labyrinthe.
- Type EtatHisto : Logiquement, il est composé d'un état auquel on ajoute une liste de positions représentant l'historique des positions déja parcourues.

On ajoute à cela un ensemble de fonctions pour les manipuler :

- Fonctions de la file : Toutes les fonctions qui permettent d'enfiler et de défiler un ou plusieurs éléments, de retourner la tête ou encore de tester si une file est vide.
- estObstacle : Teste si il y a un obstacle à la position indiquée.
- estDansListe : Teste si une position se trouve dans la liste passée en arguments.
- estDepart, estArrivee : Ces deux fonctions permettent d'indiquer si on a à faire à la position de départ ou d'arrivée.
- estValide : Teste si un état est valide (position dans le cadre et pas sur un obstacle).

Il faut pour finir écrire deux fonctions essentielles :

- La fonction successeurs qui permet de construire la liste des successeurs à un état, en tenant compte de ceux déja visités.
- La fonction principale profondeurA qui permet d'effectuer le parcours pour trouver le chemin le plus court d'un état initial à un état final.

1.3 Critères utilisés

Pour l'algorithme A* qu'il soit standard ou multicritères, il faut pouvoir calculer une valeur représentant le cout de l'état courant à l'état final selon un critère.

On peut définir ce cout comme étant la fonction h présentée en introduction.

Nous allons présenter dans cette partie les trois critères utilisés.

1.3.1 Distance Manhattan

La fonction suivante permet donc de calculer la distance Manhattan avec la formule mathématique :

$d = |x2 - x1| + |y2 - y1|$

```
distanceMan :: Position -> Position -> Double                                    1
distanceMan (x1, y1) (x2, y2) = fromIntegral (abs (x2 - x1) + abs (y2 - y1))     2
```

1.3.2 Distance Euclidienne

La fonction suivante permet de définir le cout selon la distance euclidienne avec le théorême de pythagore :

$hypotenuse^2 = cote1^2 + cote2^2$

$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

```
distanceEucli :: Position -> Position -> Double                                  1
distanceEucli (x1, y1) (x2, y2) = sqrt (fromIntegral (carre (x2 - x1) + carre (y2 - y1)))   2
        where                                                                    3
                carre x = x * x                                                  4
```

1.3.3 Proximité du chemin des obstacles

La partie importante du devoir repose sur ce critère.

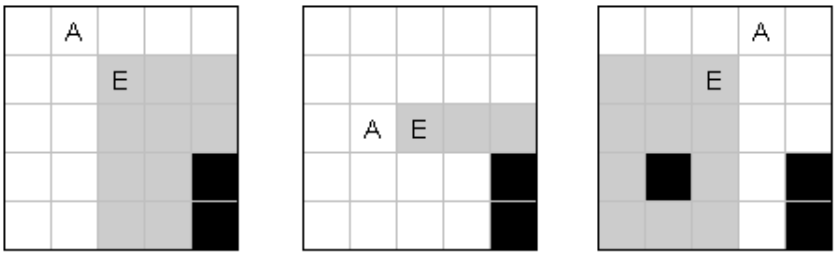
Selon l'énoncé, "plus on s'éloigne des obstacles, mieux le chemin est". Nous avons donc effectuer une première implémentation en prenant comme critère, la distance de l'état courant à l'obstacle le plus proche. Après une batterie de tests, nous avons déclaré cette méthode peu satisfaisante.

Nous avons ensuite constaté que cette première implémentation ne prenait en compte que l'état courant et le labyrinthe pour les obstacles. Ceci etait problèmatique étant donné que la fonction h doit calculé le cout de l'état courant par rapport à l'état final.

Mais comment prendre en compte cet état final ?

Après réflexion, nous avons décidé de prendre en compte que les obstacles "derrière" l'état courant. Ainsi la position d'arrivée est bien prise en compte.

Plus explicitement, on peut voir sur les schémas suivants les obstacles qui seront pris en compte, ce sont ceux se trouvant dans la partie grisée. Le A correspond à la position d'arrivée et le E à l'état courant.



Nous avons donc créé une fonction qui permet de découper la grille comme sur les schémas précédents, celle-ci retourne deux ensembles d'entiers représentant cette sous-grille.

```
decouper :: Position -> Position -> Labyrinthe -> ([Int], [Int])
decouper (x, y) (xf, yf) (l, h, obs)
        | x < xf && y < yf = ([0..x], [0..y])
        | x < xf && y > yf = ([0..x], [y..(h - 1)])
        | x > xf && y < yf = ([x..(l - 1)], [0..y])
        | x > xf && y > yf = ([x..(l - 1)], [y..(h - 1)])
        | x == xf && y < yf = ([x], [0..y])
        | x == xf && y > yf = ([x], [y..(h - 1)])
        | x < xf && y == yf = ([0..x], [y])
        | otherwise = ([x..(l - 1)], [y])
```

Puis la fonction suivante filtre les obstacles se trouvant dans cette sous-grille.

```
filtrer :: Position -> Position -> Labyrinthe -> [Position]
filtrer pos arrivee (l, h, obs) = filter (\(x, y) -> (elem x intervalleX) && (elem y intervalleY)) obs
        where
                intervalles = decouper pos arrivee (l, h, obs)
                intervalleX = fst intervalles
                intervalleY = snd intervalles
```

Nous avons ensuite décidé de considérer les parois du labyrinthe comme des obstacles. La fonction suivante calcule la paroi la plus proche "derrière" l'état courant.

```
distanceBordure :: Position -> Position -> Labyrinthe -> Int
distanceBordure (x, y) (xf, yf) (l, h, obs)
        | x < xf && y < yf = (min x y) + 1
        | x < xf && y > yf = min (x + 1) (h - y)
        | x > xf && y < yf = min (l - x) (y + 1)
        | x > xf && y > yf = min (l - x) (h - y)
        | x == xf && y < yf = y + 1
        | x == xf && y > yf = h - y
        | x < xf && y == yf = x + 1
        | otherwise = l - x
```

Finalement, voici la fonction qui calcule la distance à l'obstacle le plus proche. Elle prend comme distance initiale la paroi la plus proche. Puis elle parcourt les différents obstacles, si il y en a un qui est plus proche que la paroi on redéfinit la distance.

```
distanceObs :: Position -> Position -> Labyrinthe -> Double
distanceObs pos arrivee laby = distanceObs pos arrivee (filtrer pos arrivee laby)
                (fromIntegral (distanceBordure pos arrivee laby))
        where
                distanceObs _ _ [] tmp = tmp
                distanceObs pos arrivee (x:xs) tmp = distanceObs pos arrivee xs
                                (min tmp (distanceMan pos x))
```

1.4 Méthodes

Nous allons dans cette partie établir les méthodes unicritères de parcours utilisées. Chaque méthode correspond à un opérateur qui placera les éléments dans la file par ordre de priorité.

1.4.1 Distance Manhattan

Cette méthode préfére un état à un autre si celui-ci est plus proche de la position d'arrivée selon la distance Manhattan (ce critère correspond au premier élément du vecteur).

```
opDistanceMan :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opDistanceMan etatHisto1 etatHisto2 depart arrivee = f1Etat1 < f1Etat2
        where
                f1Etat1 = head (vecteurF etatHisto1 arrivee)
                f1Etat2 = head (vecteurF etatHisto2 arrivee)
```

1.4.2 Distance Euclidienne

Comme pour la méthode précédente, celle-ci préfére un état à un autre si celui-ci est plus proche de la position d'arrivée mais cette fois-ci en prennant comme critère la distance Euclidienne (qui correspond au deuxième élément du vecteur).

```
opDistanceEucli :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opDistanceEucli etatHisto1 etatHisto2 depart arrivee = f1Etat1 < f1Etat2
        where
                f1Etat1 = head (tail (vecteurF etatHisto1 arrivee))
                f1Etat2 = head (tail (vecteurF etatHisto2 arrivee))
```

1.4.3 Proximité du chemin des obstacles

Pour cette méthode, il faut chercher à s'éloigner le plus possible des obstacles. On utilise donc l'opérateur supérieur, car un état est préféré à un autre si il est plus loin des obstacles qui se trouvent "derrière" lui.

Il ne faut pas oublier d'utiliser le vecteur avec les historiques négatives, comme pour toutes les méthodes utilisant l'opérateur supérieur (conférer la partie suivante).

```
opDistanceObs :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opDistanceObs etatHisto1 etatHisto2 depart arrivee = f1Etat1 > f1Etat2
        where
                f1Etat1 = head (tail (tail (vecteurFNeg etatHisto1 arrivee)))
                f1Etat2 = head (tail (tail (vecteurFNeg etatHisto2 arrivee)))
```

2 Algorithme A Star multicritères

2.1 Introduction

Un algoritme A* multicritères fonctionne selon le même principe qu'un standard. Mais il fait une évaluation heuristique selon un ensemble de critères tels que :

- (f1, f2, f3) = (g1, g2, g3) + (h1, h2, h3) l'évaluation heuristique
- (g1, g2, g3) = (g, g, g) avec g l'évaluation du cout pour aller de l'état initial à l'état courant
- (h1, h2, h3) avec h1, h2, h3 les évaluations du cout pour aller de l'état courant à l'état final selon les trois critères

Il faut donc pouvoir calculer les différents vecteurs, ce que font les fonctions suivantes.

2.2 Calcul des vecteurs

2.2.1 Vecteur (h1, h2, h3)

Le vecteur (h1, h2, h3) d'un état est l'ensemble des couts de cet état à l'état final suivant les critères établis dans la partie précédente.

```
vecteurH :: EtatHisto -> Position -> [Double]
vecteurH etatHisto arrivee=[distanceMan pos arrivee,distanceEucli pos arrivee,distanceObs pos arrivee laby]
        where
                etat = lireEtat etatHisto
                pos = lireRobot etat
                laby = lireLabyrinthe etat
```

2.2.2 Vecteur (f1, f2, f3)

Le vecteur (f1, f2, f3) est égal à la somme du vecteur précédent et du vecteur composé des couts pour aller de l'état initial à l'état courant.

Mais comme ce cout est toujours le même (taille de l'historique), on fait simplement un map sur le vecteur (h1, h2, h3)

```
vecteurF :: EtatHisto -> Position -> [Double]
vecteurF etatHisto arrivee = map (+ histo) (vecteurH etatHisto arrivee)
        where
                histo = (fromIntegral (length (lireHisto etatHisto)))
```

Il faut aussi prévoir le cas ou les opérateurs préféreront un vecteur à un autre si le premier est supérieur au deuxième. On crée donc un vecteur (f1, f2, f3) qui soustrait les historiques.

```
vecteurFNeg :: EtatHisto -> Position -> [Double]
vecteurFNeg etatHisto arrivee = map (+ (negate histo)) (vecteurH etatHisto arrivee)
        where
                histo = (fromIntegral (length (lireHisto etatHisto)))
```

Enfin, il faut pouvoir créer un vecteur qui donne un poids à chaque critère, celui-ci sera utilisé pour la méthode de la moyenne pondérée.

```
vecteurFCoeff :: EtatHisto -> Position -> [Double]
vecteurFCoeff etatHisto arrivee = map (+ (negate histo)) (zipWith (*) coeff (vecteurH etatHisto arrivee))
        where
                histo = (fromIntegral (length (lireHisto etatHisto)))
                coeff = [0.25, 0.25, 0.5]
```

2.3 Méthodes multicritères

Nous allons présenter dans cette partie les méthodes utilisant l'ensemble des critères demandés, c'est à dire le vecteur (f1, f2, f3).

2.3.1 Somme

On fait simplement la somme de chaque vecteur et on donne la priorité à l'état ayant la valeur la plus grande.

```
opSomme :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opSomme etatHisto1 etatHisto2 depart arrivee =
                (sum (vecteurFNeg etatHisto1 arrivee)) > (sum (vecteurFNeg etatHisto2 arrivee))
```

2.3.2 Moyenne pondérée

On fait de même que pour la méthode précédente à l'exception qu'on utilise les vecteurs munis de coefficients.

```
opMoyenne :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opMoyenne etatHisto1 etatHisto2 depart arrivee =
                (sum (vecteurFCoeff etatHisto1 arrivee)) > (sum (vecteurFCoeff etatHisto2 arrivee))
```

2.3.3 MinMax

On prend le maximum de chaque vecteur et on donne la priorité à l'état qui a la valeur la plus petite.

```
opMinMax :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opMinMax etatHisto1 etatHisto2 depart arrivee =
                (maximum (vecteurF etatHisto1 arrivee)) < (maximum (vecteurF etatHisto2 arrivee))
```

2.3.4 Lexicographique

Pour cette méthode, il faut tester les maximums un à un jusqu'a qu'ils soient différents, dans ce cas on prend le plus petit.

```
opLexico :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opLexico etatHisto1 etatHisto2 depart arrivee =
        opLexico (reverse (sort (vecteurF etatHisto1 arrivee))) (reverse (sort (vecteurF etatHisto2 arrivee)))
        where
                opLexico [] [] = False
                opLexico (x:xs) (y:ys)
                        | x < y = True
                        | otherwise = opLexico xs ys
```

2.3.5 MinEcart

Pour chaque vecteur, on fait la différence du minimum et maximum. On donne la priorité à l'état ayant la valeur minimale.

```
opMinEcart :: EtatHisto -> EtatHisto -> Position -> Position -> Bool
opMinEcart etatHisto1 etatHisto2 depart arrivee =
                ((minimum fEtat1) - (maximum fEtat2)) < ((minimum fEtat2) - (maximum fEtat2))
        where
                fEtat1 = vecteurF etatHisto1 arrivee
                fEtat2 = vecteurF etatHisto2 arrivee
```

2.3.6 Somme des bornes

Comme pour la méthode précédente, on prend le minimum et le maximum de chaque état, mais cette fois-ci on les additionne pour donner une priorité à l'état ayant la valeur la plus grande.

```
opSommeBorne :: EtatHisto -> EtatHisto -> Position -> Position -> Bool                                          1
opSommeBorne etatHisto1 etatHisto2 depart arrivee =                                                             2
                ((minimum fEtat1) + (maximum fEtat2)) > ((minimum fEtat2) + (maximum fEtat2))                   3
        where                                                                                                   4
                fEtat1 = vecteurFNeg etatHisto1 arrivee                                                         5
                fEtat2 = vecteurFNeg etatHisto2 arrivee                                                         6
```

3 Tests de l'algorithme

Pour simplifier l'utilisation de notre algorithme A*, nous avons créé une application exécutable et utilisable par un utilisateur lambda.

Nous n'allons pas expliquer son codage, qui est lourd et assez basique. On peut quand même préciser qu'elle repose sur un système de boucle qui permet d'utiliser l'application en continu.

La manuel de cette application est fourni en annexe. C'est avec celle-ci que nous avons effectué les tests suivants.

3.1 Avec un unique chemin

Dans cette partie, nous avons testé plusieurs grilles de complexités variées. Il n'existe dans chacune d'elle qu'un unique chemin le plus court pour arriver à l'état final. Après avoir testé chacune des méthodes (standards et multicritères), voici les résultats que nous avons obtenu.

Dans tous les labyrinthes présentés, il existe un autre chemin possible pour arriver à l'état final, mais on constate que l'algorithme a fait le bon choix avec chacune des méthodes, on peut donc conclure qu'il fonctionne normalement.

3.2 Avec plusieurs chemins

Dans cette partie, il existe plusieurs chemins le plus court pour arriver à l'état final, nous allons donc essayer d'expliquer le comportement des différentes méthodes.

3.2.1 Manhattan

On voit dans le chemin suivant, que le robot longe le mur du haut puis celui de droite, ce qui est tout à fait cohérent avec la distance Manhattan.

A chaque état, le robot a le choix entre l'état du bas et l'état de droite qui sont à égale distance de l'état final. Il choisit donc le premier rencontré dans la fonction successeurs qui est l'état de droite. Et à chaque déplacement il continue selon le même principe jusqu'à rencontrer la paroi de droite, à ce moment il se dirige vers la position d'arrivée.

3.2.2 Euclidienne

Pour cette méthode, le robot utilise comme critère la distance Euclidienne, il va donc suivre la ligne la plus directe pour aller de l'état initial à l'état final. Dans notre cas il suit logiquement la diagonale du carré.

3.2.3 Proximité du chemin des obstacles

On voit sur le labyrinthe suivant, que le robot cherche bien à s'éloigner le plus possible des obstacles, car il suit le chemin le plus centré entre la paroi du haut et les obstacles.

3.2.4 Méthodes multicritères

Il est très difficile d'expliquer le comportement des méthodes multicritères étant donné qu'elles prennent en compte plusieurs critères simultanèment. Nous allons quand même faire un certain nombre de remarques.

1. Pour la **somme**, il serait logique de penser qu'elle ne fonctionne pas puisque plus on s'approche de la cible, plus la distance manhattan et la distance euclidienne sont réduites. Pourtant elle trouve bien le chemin le plus court, pourquoi ?

   Prenons le cas ou le robot s'éloigne de la cible, à ce moment les distances manhattan et euclidienne augmentent, mais au contraire on se rapproche forcément d'un obstacle puisque on ne considère que ceux qui sont "derrière" le robot. Cela suffit-il à compenser la somme des deux distances précédentes ?

   Normalement non (sauf si on se trouve près de la cible), c'est à ce moment qu'apparait l'intérêt de soustraire les historiques dans les méthodes utilisant l'opérateur supérieur.

2. Pour la **moyenne pondérée**, tout dépend des poids attribués à chaque critère. Par exemple si on attribue 1 à l'un des critères, la méthode se comportera comme une méthode standard, si on attribue 1/3 à chaque critère, elle se comportera comme la somme.

3. Pour le **MinMax** et la méthode **lexicographique**, à chaque état le robot se comportera comme si il avait à faire à une méthode standard. Mais le critère utilisé pourra changer en cours de chemin.

4. La méthode **MinEcart** est un cas très difficile à étudier.

5. La méthode **somme des bornes** fonctionne selon le même principe que la méthode somme à l'exception qu'à chaque état, elle choisit deux critères (le minimum et le maximum).

4 Conclusion

Pour conclure, la mise en place d'un algorithme A* est assez simple, surtout avec un langage fonctionnel comme Haskell.

Pour atteindre notre objectif, il a fallu bien décomposer le problème en respectant l'énoncé, notamment en utilisant les vecteurs (f1, f2, f3) = (g1, g2, g3) + (h1, h2, h3).

Grace à cette décomposition, on se retrouve face à des formules mathématiques facilement vérifiables.

Quand au but à atteindre, qui est de trouver le chemin le plus court, avec chaque méthode utilisée pour un labyrinthe quelconque. Nous pensons l'avoir atteint, mais même après avoir vérifié les méthodes sur un nombre conséquent de parcours, nous ne sommes jamais à l'abri d'un défaut ou d'un bug.

Le mieux aurait été d'effectuer un test par oracle. Par exemple prendre un algorithme A* standard admis et vérifié, puis d'effectuer des parcours aléatoires, et enfin de vérifier la correspondance des résultats avec les méthodes multicritères.