|  |
| --- |
| UniversitÉ du quÉbec en outaouais |
| Mini-Projet : Agent intelligent dans un environnement à obstacle |
| INF1183 - Hiver 2014 – Intelligence artificielle |
| **Guillaume Plouffe**  **Jean-Philippe Gauthier**  **Julien Bassompierre** |
| **2/28/2014** |

Table des matières

[1 Introduction 1](#_Toc386093004)

[2 Description du PEAS 2](#_Toc386093005)

[2.1 Performances 2](#_Toc386093006)

[2.2 Environnement 2](#_Toc386093007)

[2.3 Actuateurs 3](#_Toc386093008)

[2.4 Senseurs 3](#_Toc386093009)

[3 Fonctionnement du programme 4](#_Toc386093010)

[4 Description du code 6](#_Toc386093011)

[4.1 Algorithme A\* 6](#_Toc386093012)

[4.2 Heuristique utilisées 8](#_Toc386093013)

[5 Analyse des performances 12](#_Toc386093014)

[5.1 Analyse de l’heuristique chemin de Manhattan 12](#_Toc386093015)

[6 Conclusion 18](#_Toc386093016)

[7 Références 20](#_Toc386093017)

Table des figures

[Figure 1 : Le terrain et l'agent 2](#_Toc386093018)

[Figure 2 : Agent utilisant une évaluation restreinte VS globale du terrain avant de réaliser une combinaison d'action 3](#_Toc386093019)

[Figure 3 : Message demandant le commencement de l'analyse de l'algorithme 4](#_Toc386093020)

[Figure 4 : Affichage du plus court chemin (points rouges) trouver par l'algorithme A\* et des cases qu'il a inutilement exploré (point rouge). Note : Le curseur n’est pas afficher sur l’image. 5](#_Toc386093021)

[Figure 5 : Confirmation du déplacement de l'agent sur une autre case 5](#_Toc386093022)

[Figure 6 : Recherche du plus court chemin avec l'algorithme de Disjstra 6](#_Toc386093023)

[Figure 7 : Recherche du plus court chemin avec Greedy Best first search 7](#_Toc386093024)

[Figure 8 : Recherche du plus court chemin avec l'algorithme A\* 7](#_Toc386093025)

[Figure 10 : Exemple de test utilisant l’heuristique de Manhattan avec un cout D de 1.2. 13](#_Toc386093026)

[Figure 11 : Graphe du nombre pas total des chemin trouvée en fonction du cout D de h(n) 14](#_Toc386093027)

[Figure 12 : Graphe du nombre de case essayer inutilement en fonction du D de h(n) 15](#_Toc386093028)

# Introduction

Pour ce mini-projet nous avons choisis l’énoncé 2 consistant à définir un agent intelligent qui se deplace dans une enivronnement pouvant avoir des obstacles. Le but (performance) de l’agent est de trouver le plus court chemin entre un point de depart et un point d’arrivé tout en contournant les obstacles.

Nous allons tout d’abord definir le PEAS de l’agent, donner une breve description sur le fonctionnement du programme et sur l’algorithme et les heuristiques utilisé. Ensuite, une analyse sera presenté en fonction de different heuristique, action possible, arrangement des obstacles et propriete de l’environnement.

# Description du PEAS

Dans cette section du document, nous allons décrire les différentes composantes du PEAS entourant l’agent : La performance, l’environnement, les actuateurs et les senseurs.

## Performances

L’objectif principale de l’agent est de trouvée le chemin le plus court entre un point départ et d’arrivée. Également, il doit pouvoir trouver ce chemin rapidement, et donc essayé un minimum de case de l’environnement dans sa recherche.

## Environnement

Il s’agit d’un terrain de tuile à deux dimension pouvant être considérer comme un graphe dont les cases sont les nœuds et dont les arêtes sont placé entre les tuiles adjacente. Les cases case peuvent contenir des obstacles ou non. Les obstacles sont représenter en noir avec un arbre vert au centre et les espace libre sont représenter par de l’herbe verte. L’agent est représenté par le tank en rouge. La grandeur du terrain est de 30 x 20 tuiles. La case départ est là où apparait l’agent, alors que la case d’arriver est la case au-dessus de laquelle se trouve notre curseur souris (non représenté). Dans notre environnement de test, les obstacles sont configurés afin de constituer un cul-de-sac. Ce dernier est donc un piège que notre algorithme tentera au meilleur de ses moyens d’éviter à moindre cout pour trouver la solution du plus court chemin.

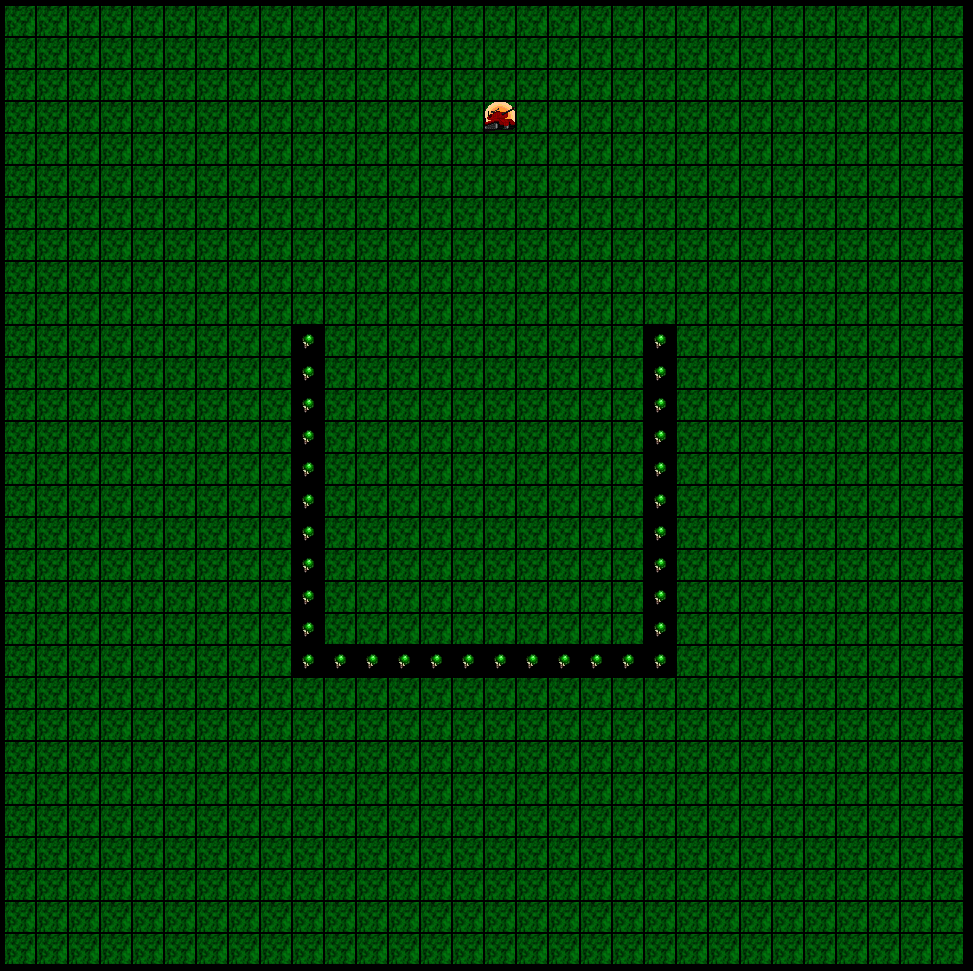


Figure : Le terrain et l'agent

## Actuateurs

L’agent (tank rouge) peut réaliser quatre actions de déplacement : Vers le nord, sud, est, ouest. Le cout de déplacement par case libre est 1 et il ne peut passer par une case obstacle ou sortir du terrain.

## Senseurs

Il s’agit de la camera de notre agent représenter par sa capacité à planifier d’avance son parcours et évaluer n’importe quel case par-rapport au point de départ et d’arrivé. Ce senseur est crucial dans le calcul de l’algorithme A\*. Dans la réalité, ce genre d’algorithme est plus lent comparativement à un algorithme qui réalise une action et qui effectue une évaluation plus restreinte des cases à proximité pour réaliser la suivante. Cependant, cette dernière ne pourra donner un parcours optimal en distance du fait de son manque de connaissance sur son environnement. En effet, il ne pourra éviter de souvent tomber et rester pris dans des pièges (cul-de-sac).

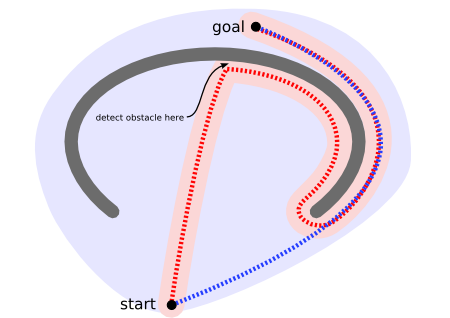


Figure : Agent utilisant une évaluation restreinte VS globale du terrain avant de réaliser une combinaison d'action

# Fonctionnement du programme

La méthode main du programme est localisée dans la classe PathTest. Lors de son exécution, le terrain est construit et affiché à l’écran. Un message apparait alors et demande à l’utilisateur de presser le bouton OK pour commencer les tests d’analyse de l’algorithme. Une description détaillée de ces tests est donnée plus dans la section 5 de ce document. Le résultat est sauvegarder automatiquement dans un fichier nommer *TestB.txt* .

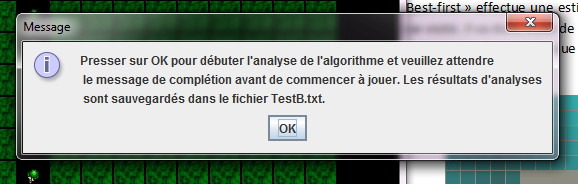
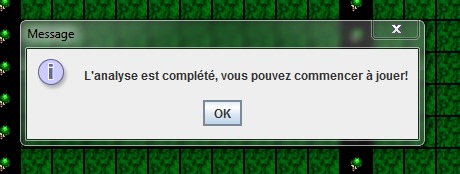


Figure : Message demandant le commencement de l'analyse de l'algorithme

Lorsque l’analyse est terminer, une fenêtre apparait alors et avertie l’utilisateur qu’il peut maintenant jouer avec le programme.



Le jeu est simple, pour sélectionner notre agent (tank rouge) il suffit de faire un clique gauche ou droite de la souris sur la case qu’il occupe et ensuite déplacer le curseur au-dessus d’une autre case pour que l’algorithme A\* effectue le calcul et affiche le trajet de plus court chemin trouvé et les cases qu’il a évalué et écarter de la solution (Figure 4). Pour confirmer son déplacement vers une autre case, on clique à nouveau un des boutons de la souris (Figure 5). On peut terminer le programme en tout temps en cliquant sur le X en haut à droite de la fenêtre principale.

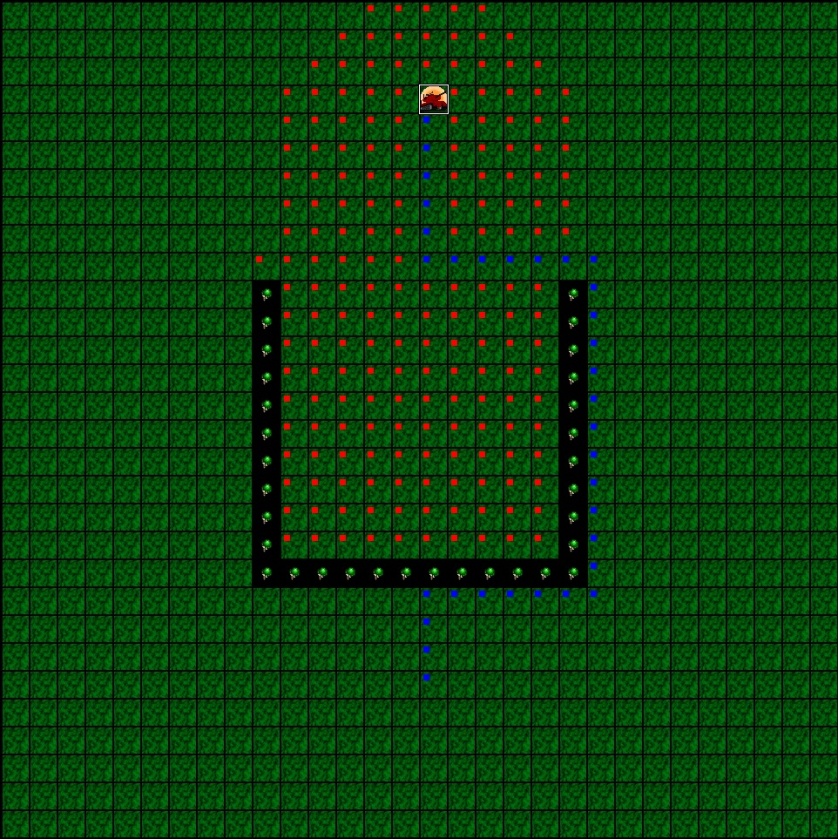


Figure : Affichage du plus court chemin (points rouges) trouver par l'algorithme A\* et des cases qu'il a inutilement exploré (point rouge). Note : Le curseur n’est pas afficher sur l’image.

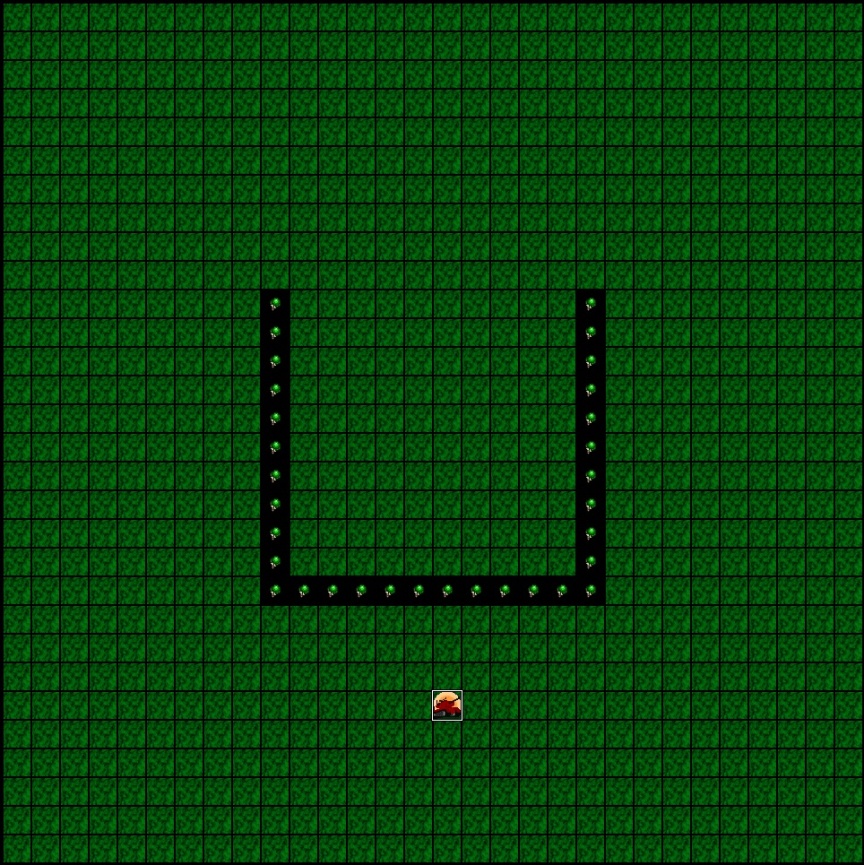


Figure : Confirmation du déplacement de l'agent sur une autre case

# Description du code

## Algorithme A\*

L’algorithme A\*est la méthode la plus populaire dans la recherche du plus court chemin. Il combine les avantages de l’algorithme de Disjstra et de la recherche « best-first » (un algorithme de type glouton). L’algorithme de Disjstra consiste a visité de manière incrémental le nœud le plus prés encore inexploré. Il s’agit d’une procédure longue, mais qui garantie d’obtenir le chemin optimal. De son coté, l’algorithme « Best-first » effectue une estimation (heuristique) sur le chemin qu’il reste à parcourir à chacune des case visité. Il va donc visité de manière incrémental les nœuds les plus proches du but. Cette algorithme est beaucoup plus rapide que Disjstra, mais la solution obtenue n’est pas garantie d’être la distance la plus courte.

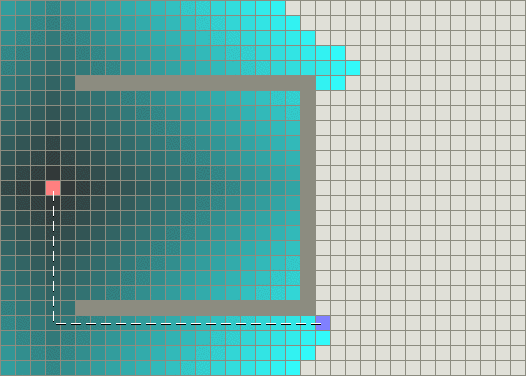


Figure : Recherche du plus court chemin avec l'algorithme de Disjstra

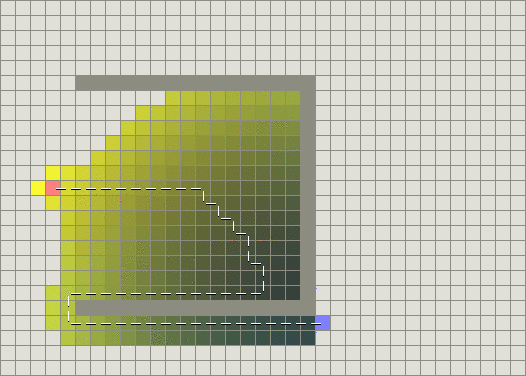


Figure : Recherche du plus court chemin avec Greedy Best first search

L’algorithme A\* est communément représenter par F(n) = g(n)+h(n). La partie Disjstra est représentée par g(n). Elle calcule le cout exact du chemin parcouru depuis la case départ jusqu'à la case n. La partie heuristique (best first search), est représentée par h(n). Elle calcul le cout estimé entre la case n et celle d’arrivée. L’algorithme A\* va réaliser une boucle afin de vérifier a chaque itération la case ayant la plus faible valeur de f(n) = g(n)+h(n).

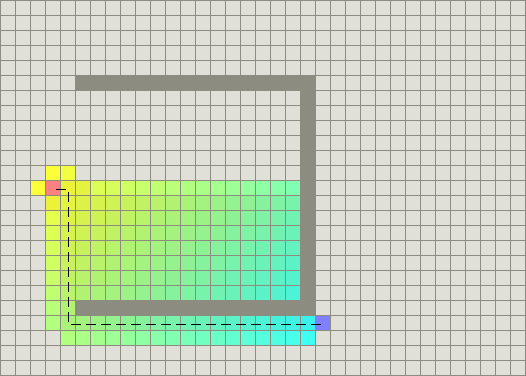


Figure : Recherche du plus court chemin avec l'algorithme A\*

## Heuristique utilisées

Puisque nos actions de déplacement sont restreintes à quatre directions, il est conseillé d’utiliser l’heuristique de la distance de Manhattan. Voici sa fonction implanté dans la classe *ClosestHeuristic*:

/\*\*

\* Heuristique de A\* : Le chemin de Manhattan

\*

\* **@param** map Le terrain de tuile sur lequel le chemin est recherché

\* **@param** mover L'agent se déplacant sur le terrain, utile pour récupérer comportement special

\* **@param** D Le poid de l'heuristique

\* **@param** x Coordonnée en x de la case à évaluer

\* **@param** y Coordonnée en y de la case à évaluer

\* **@param** tx Coordonnée en x de la case à destination

\* **@param** ty Coordonnée en y de la case à destination

\*/

**public** **float** getCostManhattan(TileBasedMap map, Mover mover,**float** D, **int** x, **int** y, **int** tx, **int** ty) {

**float** dx = Math.*abs*(tx - x);

**float** dy = Math.*abs*(ty - y);

**return** (**float**) ((D)\*(dx +dy));

}

Le paramètre D représente le cout de déplacement minimal entre deux cases adjacentes (cout réel). Pour un terrain sans obstacle, s’approcher du but d’une case revient à augmenter g(n) de D et diminuer h(n) de D. Puisque D est le même pour g(n) et h(n), nous avons donc un heuristique dite « admissible », c'est-à-dire que h(n) ne surestime jamais le cout réel pour atteindre la destination (l’objectif). De cette manière, il est garantie que A\* va trouver le chemin le plus court. Le problème de cette approche réside dans l’effort investi pour trouver ce chemin optimal. Peut-être serait-il mieux de regarder du coté des heuristiques non-admissible afin de trouver un algorithme rapide qui possède de bonne chance d’obtenir le chemin le plus court. Dans la prochaine section nous allons tenter de trouver une telle heuristique.

Voici l’implémentation du code permettant de trouver le plus court chemin entre la case de départ et d’arriver en utilisant l’algorithme A\*. Note : Le cout retourné, lors du calcul de g(n), est toujours 1 dans notre cas.

/\*\*

\* Trouve un chemin entre le noeud de depart(sx,sy) et de destination(tx,ty)

\* tout en évitant les zones interdites et en minimisant le cout de deplacement

\* f(n)

\* **@param** mover Il s'agit de notre agent se deplacant, ce parametre contient les informations

\* le concernant, ie les actuateurs et senseurs.

\* **@param** sx Coordonnée x du noeud de depart

\* **@param** sy Coordonnée y du noeud de depart

\* **@param** tx Coordonnée x du noeud d'arrivee

\* **@param** ty Coordonnée y du noeud d'arrivee

\* **@return** Le chemin trouvee entre le noeud de depart et d'arrivee, sinon null si pas trouvee

\*/

**public** Path findPath(Mover mover, **int** sx, **int** sy, **int** tx, **int** ty) {

// Si la case destination est un obstacle, alors annulle la recherche.

**if** (map.blocked(mover, tx, ty)) {

**return** **null**;

}

//Initialisation de l'etat de l'agent, la liste Closed (Liste de noeud //essayer et écarter de la solution) est vide,

//la liste Open (Liste de noeud a evaluer pouvant potentiellement faire //partie de la solution) ne contient que le noeud de depart avec

//un cout de 0 (on est deja là) et une profondeur atteinte de 0 (pas //fait de pas encore).

nodes[sx][sy].setCost(0);

nodes[sx][sy].setDepth(0);

closed.clear();

open.clear();

open.add(nodes[sx][sy]);

nodes[tx][ty].setParent(**null**);

// Tant que nous avons pas atteint la destination et depassé un nombre //max de pas, faire:

**int** maxDepth = 0;

**while** ((maxDepth < maxSearchDistance) && (open.size() != 0)) {

//Prendre le premier noeud de la liste. Il s'agit du prochain //noeud à être explorer (ou reevaluer).

Node current = getFirstInOpen();

//Si le noeud à evaluer est le noeud destination alors arreter la //recherche.

**if** (current == nodes[tx][ty]) {

**break**;

}

// On enleve le noeud de la liste Open et on la met dans la liste Closed

removeFromOpen(current);

addToClosed(current);

//Rechercher chaque noeud voisin du noeud en evaluation et //calculer leur cout

//si elle devienne le prochain noeud sur lequel se deplace l'agent

**for** (**int** x=-1;x<2;x++) {

**for** (**int** y=-1;y<2;y++) {

// Ce noeud est celui actuellement sous evaluation, //il n'est donc pas un voisin

**if** ((x == 0) && (y == 0)) {

**continue**;

}

//Si on interdit les deplacement en diagonale alors

//les voisins en coin sont ignorer

**if** (!allowDiagMovement) {

**if** ((x != 0) && (y != 0)) {

**continue**;

}

}

//Localison le noeud voisin sur le terrain

**int** xp = x + current.getX();

**int** yp = y + current.getY();

//Si le noeud voisin est une case de deplacement //potentiel valide,

//ie si c'est une case sans obstacle, a l'interieur //du terrain et n'est pas le noeud de depart.

**if** (isValidLocation(mover,sx,sy,xp,yp)) {

//Le cout pour atteindre ce noeud voisin est //le cout du noeud en evaluation plus le cout //du mouvement.

//Ce calcul n'inclue que la partie Disjstra (g(n)) de A\*

**float** nextStepCost = current.getCost() + getMovementCost(mover, current.getX(), current.getY(), xp, yp);

Node neighbour = nodes[xp][yp];

map.pathFinderVisited(xp, yp);

//Si le nouveau cout (g(n)) calculer pour le //noeud voisin est moindre que

//son ancien cout calculer, alors s'assurer //que le noeud n'est pas

//dans la liste Closed. Nous avons trouver un meilleur chemin pour nous

//rendre a ce noeud. Il faut donc reevaluer //son f(n):

//Si le noeud faisait deja partie de la liste //Open, nous allons le retirer

//pour le temps de la reevaluation.

**if** (nextStepCost < neighbour.getCost()) {

**if** (inOpenList(neighbour)) {

removeFromOpen(neighbour);

}

**if** (inClosedList(neighbour)) {

removeFromClosed(neighbour);

}

}

//Si le noeud ne fait pas partie de la liste //de noeud candidate a evaluer(Open) ou de la //liste des noeud rejeter (Closed)

//alors calculer et assigner son nouveau cout //g(n) et h(n) et l'ajouter à la liste Open.

**if** (!inOpenList(neighbour) && !(inClosedList(neighbour))) {

neighbour.setCost(nextStepCost);

neighbour.setHeuristic(getHeuristicCost(mover, xp, yp, tx, ty, sx, sy));

maxDepth = Math.*max*(maxDepth, neighbour.setParent(current));

addToOpen(neighbour);

}

}

}

}

}

//Nous n'avons pas atteint la destination dans notre recherche. Retourne //null.

**if** (nodes[tx][ty].getParent() == **null**) {

**return** **null**;

}

//Nous avons atteint la destination, retourner le chemin trouver.

//Utiliser les references au parent pour faire le chemin du retour et //reconstituer l'ensemble des

//noeuds empruntés

Path path = **new** Path();

Node target = nodes[tx][ty];

**while** (target != nodes[sx][sy]) {

path.prependStep(target.getX(), target.getY());

target = target.getParent();

}

path.prependStep(sx,sy);

//Give the length of the path

System.*out*.println("Nombre de pas du chemin : "+ path.getLength());

System.*out*.println("Nombre case essayé sans succes : "+ **this**.getClosedList().size());

// thats it, we have our path

**return** path;

}

# Analyse des performances

Tout en gardant le cout réel (D=1) du coté de g(n), nous allons réaliser, dans cette section, une étude sur la valeur du cout donner à h(n) et déterminer celle qui nous permet d’optimiser nos chance d’obtenir le plus court chemin tout en étant rapide.

Pour chacun des essaie décrit ci-dessus nous allons utiliser un terrain contenant un mur d’obstacle de forme concave (piège en cul-de-sac) et faire varier la valeur de D (de h(n)), la position de départ et de destination. La fonction heuristique h(n) reste celle du chemin de Manhattan décrit dans la section précédente.

## Analyse de l’heuristique chemin de Manhattan

Pour cette analyse nous avons effectué 450 tests dans un environnement statique avec des obstacles. Chaque test utilise une combinaison différente de position de départ et d’arrivée. L’ensemble des positions de départ utilisé sont contenue dans la boite blanche supérieure (ligne 3, colonne 1 à 15) et ceux d’arrivée dans la boite blanche inferieur de la Figure 9 (ligne 18, colonne 1 à 30).

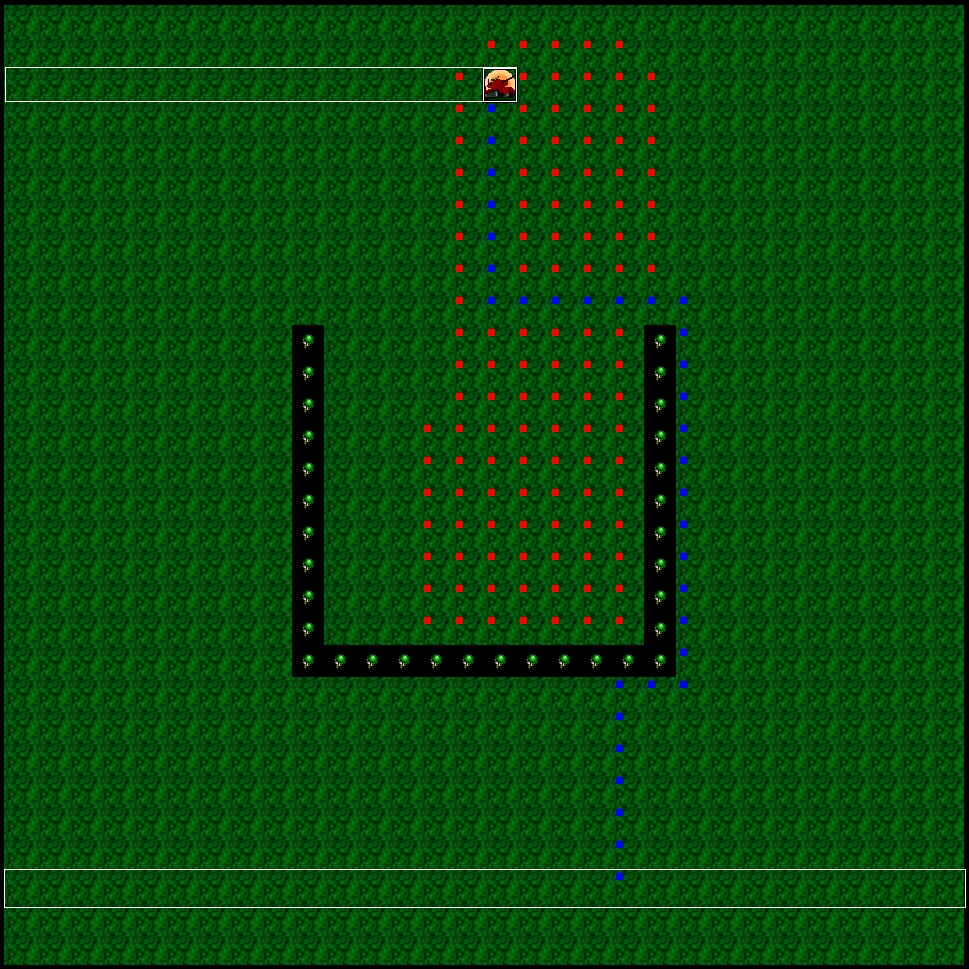


Figure  : Exemple de test utilisant l’heuristique de Manhattan avec un cout D de 1.2.

Pour chacun de ces tests, nous avons incrémenté de 0.2 le cout de notre heuristique par rapport à g(n). Nous avons additionné ensemble tous les pas de chaque chemin optimal trouvé et de chaque case essayé inutilement par l’algorithme. Le résultat est présenté dans Tableau 1.

Tableau Etude de la precision et de la rapidité de l’heuristique de Manhantan en fonction du poid

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Essaie | Poid | TotalTest | totalCheminNonTrouver | totalPath (pas) | totalCaseEssayerEchec |
| 0 | 0 | 450 | 30 | 11380 | 212641 |
| 1 | 0.2 | 450 | 30 | 11380 | 185868 |
| 2 | 0.4 | 450 | 30 | 11380 | 155865 |
| 3 | 0.6 | 450 | 30 | 11380 | 129794 |
| 4 | 0.8 | 450 | 30 | 11380 | 103915 |
| 5 | 1 | 450 | 30 | 11380 | 84886 |
| 6 | 1.2 | 450 | 30 | 11380 | 39657 |
| 7 | 1.4 | 450 | 30 | 11694 | 39522 |
| 8 | 1.6 | 450 | 30 | 12000 | 37664 |
| 9 | 1.8 | 450 | 30 | 12102 | 35522 |
| 10 | 2 | 450 | 30 | 12192 | 34923 |
| 11 | 2.2 | 450 | 30 | 12292 | 35391 |
| 12 | 2.4 | 450 | 30 | 12356 | 35433 |
| 13 | 2.6 | 450 | 30 | 12410 | 36167 |
| 14 | 2.8 | 450 | 30 | 12430 | 36947 |
| 15 | 3.000001 | 450 | 30 | 12456 | 37926 |
| 16 | 3.200001 | 450 | 30 | 12546 | 37888 |
| 17 | 3.400001 | 450 | 30 | 12588 | 38463 |
| 18 | 3.600001 | 450 | 30 | 12670 | 38797 |
| 19 | 3.800001 | 450 | 30 | 12756 | 39137 |
| 20 | 4.000001 | 450 | 30 | 12786 | 39504 |
| 21 | 4.2 | 450 | 30 | 12890 | 39549 |
| 22 | 4.4 | 450 | 30 | 12948 | 39889 |
| 23 | 4.6 | 450 | 30 | 13000 | 40049 |
| 24 | 4.8 | 450 | 30 | 13054 | 40090 |
| 25 | 5 | 450 | 30 | 13082 | 40261 |

Figure : Graphe du nombre pas total des chemin trouvée en fonction du cout D de h(n)

Figure : Graphe du nombre de case essayer inutilement en fonction du D de h(n)

Les essaies utilisant une valeur sous-estimant la valeur réel du cout de l’heuristique donne plus de poids a la fonction de type Disjtra (g(n) dans le calcul de A\*. Ainsi, tel qu’attendu, pour ces essaie on obtient toujours les plus courts chemins avec un total consistant de 11380 pas. Cependant, on obtient le problème de lenteur du calcul de DIsjstra avec un nombre de cases essayé inutilement variant de 75179 à 214106. On constate que la valeur de *totalCaseEssayerEchec* diminue grandement avec l’augmentation graduelle du cout D de h(n) jusqu'à la D=1.2. Pour des valeurs de D dépassant 1.2, *totalCaseEssayerEchec* demeure pratiquement stable. Cependant, c’est le nombre de pas total des chemins trouvée qui commence à augmenter grandement. Il s’agit d’un indice que les chances d’obtenir un le plus court chemin diminue.

Comme on peut le constater, pour cette environnement l’heuristique est optimale en précision (chance d’obtenir le plus court chemin) et rapidité avec un cout D égale à environ 1.2. C'est-à-dire f(n) = 1\*g(n) + 1.2 \* h(n). Il est ainsi préférable dans ce type d’environnement de légèrement surestimer le cout donner par l’heuristique.

Le code réalisant cette analyse et placé dans la méthode *main* de la classe *PathTest* est présenté ci-dessous.

**public** **static** **void** main(String[] argv) {

String nomFichier = "TestB.txt";

File file = **new** File(nomFichier);

String content = **new** String();

Path path;

**int** n=0;

PathTest test = **new** PathTest();

//Calcul et enregistre le contenu des resultats d'analyse pour chaque //test dans le fichier

content = "Essaie; Poid; TotalTest; totalCheminNonTrouver; totalPath (pas);totalCaseEssayerEchec;\n";

test.finder.setHeuristicType(0);

**int** totalTest=0;

**int** totalPath=0;

**int** totalCaseEssayerEchec = 0;

**int** totalCheminNonTrouver =0;

test.finder.setHeuristicWeight((**float**)(0));

//Variation du poid (cout) de l'heuristique

JOptionPane.*showMessageDialog*(**null**, "Presser sur OK pour commencer l'analyse de l'algorithme et veuillez attendre le message de confirmation avant de commencer à jouer. Les resultat d'analyse sont sauvegarder dans le fichier "+nomFichier+".");

**for** (**float** z=0;z<5;z=(**float**)(z+0.2)){

totalTest=0;

totalPath=0;

totalCaseEssayerEchec = 0;

totalCheminNonTrouver = 0;

test.finder.setHeuristicWeight(z);

//Variation des positions de depart et de destination

**for** (**int** i = 0; i<15;i++){

**for** (**int** j = 0; j<30;j++){

path = test.finder.findPath(**new** UnitMover(test.map.getUnit(15, 3)),i, 2, j, 18);

totalTest++;

**if**(path==**null**)

totalCheminNonTrouver++;

**else**{

totalPath=totalPath+path.getLength();

totalCaseEssayerEchec=totalCaseEssayerEchec+test.finder.getClosedList().size()+test.finder.getOpenListSize()-path.getLength();

}

}

}

content += n +";";

content += z +";";

content += totalTest+";";

content += totalCheminNonTrouver+ ";";

content += totalPath+ ";";

content += totalCaseEssayerEchec+ ";\n";

n++;

}

*writeToFile*(content,file );

test.finder.setHeuristicWeight((**float**)1.2);

JOptionPane.*showMessageDialog*(**null**,"L'analyse est complété, vous pouvez commencer a jouer!");

test.setInTest(**false**);

}

# Conclusion

En résumé, nous avons choisis la méthode de recherche avec l’algorithme A\* pour obtenir le plus court chemin dans un environnement statique à obstacle. Nous avons établis un cout de 1 par case de déplacement. A partir de l’équation simple de l’algorithme A\*, f(n)=D\*g(n) + D\*h(n), où D représente le cout minimal de déplacement (soit D= 1 dans notre cas), nous avons effectué une analyse de l’heuristique afin d’améliorer nos performance dans un environnement contenant une série d’obstacle formant une ligne de forme concave.

Nous avons choisis un ensemble stratégique de 450 combinaisons de point de départ et de destination différente pour réaliser chacun de nos tests dans lesquels nous avons fait varier le cout D de l’heuristique h(n). Nous avons obtenue un résultat optimale en chance d’obtenir le plus court chemin et en rapidité d’exécution avec la formule f(n)= 1\*g(n)+1.2\*h(n).

Nous aurions pu améliorer la précision de la formule en reprenant l’analyse de D entre 1.0 et 1.4 tout en diminuant son incrémentation. Également, nous aurions pu réaliser un test sur l’ensemble de toutes les combinaisons possibles de point de départ et destination du terrain, mais le temps d’analyse aurait été significativement plus long.

Il existe une variante intéressante de l’algorithme A\*: le Dynamic weigthing. Cette algorithme permet de mettre l’accent sur l’exploration en début de recherche et de terminé en donnant plus d’importance à l’atteinte rapide de la destination. La fonction est décrite par f(p)=g(p)+w(p)\*h(p) où w(p)>1 est associé à l’heuristique. Nous avons implémenté cet algorithme dans notre code, mais nous n’avons pas eu le temps d’en faire une analyse. En voici le code localisé dans la classe *ClosestHeuristic* :

/\*\*

\* Heuristique de la variante de l'algorithme A\* : Le Dynamic weigthing

\*

\* **@param** map Le terrain de tuile sur lequel le chemin est recherché

\* **@param** mover L'agent se déplacant sur le terrain, utile pour récupérer \*comportement special

\* **@param** weight Le poid de l'heuristique

\* **@param** x Coordonnée en x de la case à évaluer

\* **@param** y Coordonnée en y de la case à évaluer

\* **@param** tx Coordonnée en x de la case à destination

\* **@param** ty Coordonnée en y de la case à destination

\* **@param** sx Coordonnée en x de la case à départ

\* **@param** sy Coordonnée en y de la case à départ

\*/

**public** **float** getCostWithDynamicWeighting(TileBasedMap map, Mover mover,**float** weight, **int** x, **int** y, **int** tx, **int** ty, **int** sx, **int** sy) {

**float** dx = Math.*abs*(tx - x);

**float** dy = Math.*abs*(ty - y);

**float** px = Math.*abs*(tx - sx);

**float** py = Math.*abs*(ty - sy);

**float** weightX=1;

**float** weightY=1;

**if** (px>0)

weightX = dx/px;

**if** (py>0)

weightY = dy/py;

**return** weightX\*dx + weightY\*dy;

}

# Références

1. Site web : *https://github.com/samwdon/AntColonyOptimization*, accédé le 28 février 2014.
2. Site web : *http://www.epi.asso.fr/revue/articles/a0309b.htm*, accédé le 28 février 2014.
3. Site web : http://realtimecollisiondetection.net/blog/?p=56