Chapitre 2

2. Le pathfinding

2.1. Introduction

Qu'est ce que le pathfinding ?

Pour déplacer un personnage d'un point à un autre point dans une scène (2D ou 3D), on peut faire appel à deux grandes méthodes: le personnage peut être déplacé avec les touches directionnelles du clavier, et dans le cas d'un jeu 2D, il avancera case par case ou pixel par pixel; autrement, il peut indiquer le point où il veut se rendre et le déplacement se fait alors automatiquement. Dans le second cas, le moteur devra calculer le chemin optimal entre le point de départ et le point d'arrivée. C'est le Pathfinding (ou recherche de chemin). Le Pathfinding est aussi fréquemment utilisé dans la prévision et la gestion des déplacements des ennemis [Amia]**.**

Nous nous sommes intéressés à établir une étude informatique large, des algorithmes de plus courts chemins, basée sur les algorithmes les plus récents. Nous nous proposons ainsi de proposer des algorithmes nouveaux motivés par les résultats empiriques. Ces algorithmes nous ont menés à des résultats théoriques intéressants confortés par les observations empiriques. Notre étude informatique est basée sur plusieurs classes de problèmes naturels qui identifient les forces et les faiblesses d'algorithmes divers.

Ces classes de problèmes et la mise en œuvre d'algorithmes palliatifs, forment un environnement pour évaluer l'exécution des algorithmes de plus courts chemins. L'interaction

entre l'évaluation expérimentale du comportement des algorithmes et l'analyse théorique de l'exécution de ces algorithmes joue un rôle important dans notre recherche.

Nous décrivons des algorithmes pour la détermination de plus courts chemins et des distances dans des graphes plats qui exploitent la topologie particulière de l'entrée graphique. Une particularité importante de nos algorithmes réside dans le fait qu'ils peuvent travailler dans un environnement dynamique, où le coût de n'importe quelle entité peut être modifié, l'entité pouvant, également, être supprimée. Pour un objet simple, le mouvement semble facile tout en étant novateur.

Aspect novateur

Considérons la situation suivante:

Le but de notre acteur est de parvenir au sommet. Ne détectant aucun obstacle dans le secteur, il parcourt alors son chemin vers la cible (en rose). Près du sommet, il détecte un obstacle et change la direction. Il trouve alors sa voie le long de l'obstacle de forme « U » jusqu'à aboutir à la cible. Au contraire, un acteur averti aurait considéré un plus grand secteur (en bleu clair), mais n'aurait trouvé aucun chemin plus court (bleu), car ne détectant la cible celle-ci étant dans le champ de l'obstacle concave formé (figure 2.1).

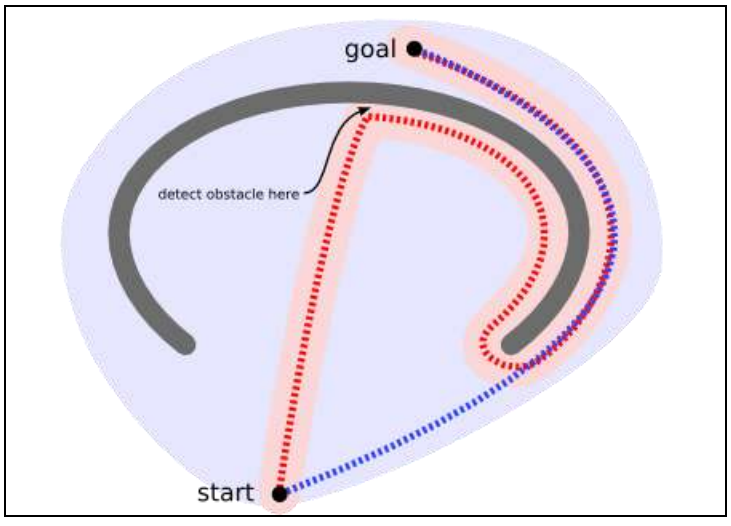

Figure 2.1 : Le mouvement pour un objet simple.

On peut cependant exploiter un algorithme de recherche pour travailler autour des pièges (figure 2.2). Le choix pourrait être l'évitement de création d'obstacles concaves, ou bien le marquage de la coque convexe comme étant dangereuse, l'acteur ne tentera une pénétration à l'intérieur de cette zone que s'il s'est avéré que le but est à l'intérieur.

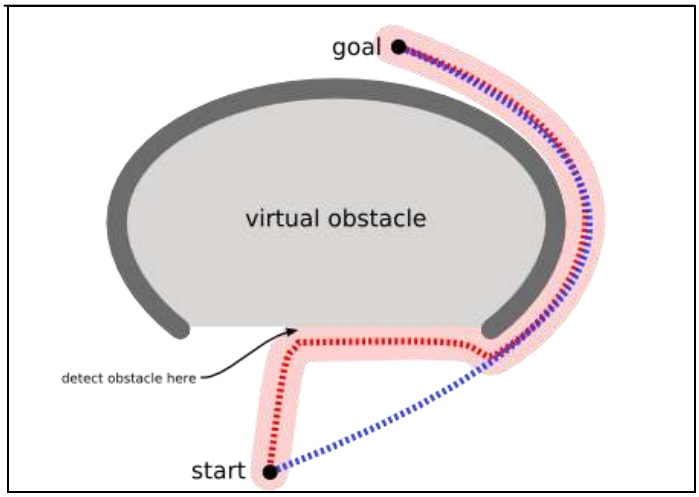

Figure 2.2 : Le mouvement pour l'évitement des obstacles

2.2. Les divers algorithmes de Pathfinding

Les algorithmes novateurs de recherche manuelle de chemins travaillent sur des graphiques, dans le sens mathématique. Un jeu de sommets avec bords: une carte de jeu carrelée peut être considérée comme un graphique dont chaque tuile représentant un sommet, les bords considérés comme des tuiles adjacents les uns des autres (figure 2.3).

Les algorithmes les plus novateurs de l'intelligence artificielle (algorithmes de recherche), sont conçus pour des graphiques arbitraires plutôt que des jeux basés sur réseau. La difficulté majeure réside dans le fait qu'il existe des aspects du comportement humain qui étaient naguère difficilement transposable pour une prise en charge informatique dans le but de l'animation d'acteurs virtuels. Nous nous intéressons dans ce qui suit à décrire un certain nombre de contributions et tentatives pour la prise en charge du problème de recherche de chemins.

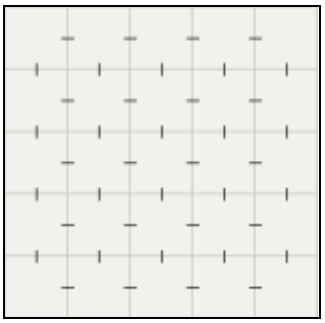

Figure 2.3 : Le jeu de vertices

2.2.1. La solution du fainéant

Une des premières solutions consiste à tracer une ligne droite entre les points de départ et d'arrivée et de demander à l'acteur ou l'agent de suivre cette ligne. S'il rencontre un obstacle, il le contourne par la droite ou par la gauche.

Ceci fonctionne plutôt bien pour des obstacles convexes mais présente beaucoup de faiblesses dans les cas des obstacles concaves. Il est impensable de l'utiliser tout seul dans tout jeu qui se respecte. Néanmoins, cet algorithme, couplé à un ou plusieurs autres, peut s'avérer très utile.

2.2.2. L'algorithme de Dijkstra

Les travaux de l'algorithme de ***Dijkstra*** consistent en un parcours des nœuds dans un graphe et ce en commençant avec le point de départ de l'objet. L'algorithme de ***Dijkstra*** calcule le chemin le plus court entre deux nœuds d'un réseau. L'idée de base est de maintenir pour chaque ensemble de nœuds P le chemin le plus court ayant été trouvé. Chaque nœud extérieur à P doit être atteint à partir d'un nœud déjà dans P. Il s'étend ainsi à l'extérieur du point de départ avant qu'il n'accède le but.

Le principe est relativement similaire:

- L'acteur part du point de départ puis calcule le poids de ses points adjacents;
- Il examine le point dont le poids est le plus faible;
- Il calcule, ensuite, le poids de ses points adjacents. Leurs poids sont éventuellement mis à jour (s'il est plus faible);
- Ensuite, il passe au point dont le poids est le plus faible parmi l'ensemble des points.
- Etc…….

L'implémentation d'un tel algorithme s'effectue en utilisant une structure de file. Quand un point est placé dans la file, il l'est en fonction de son poids calculé. Le nouveau point à examiner est alors celui dont le poids et le plus faible, qui se trouve en tête de la file. Cette fois-ci, le poids de la cellule est pris en compte, mais, comme l'algorithme précédent, les directions sont équivalentes.

Avantage

L'algorithme de Dijkstra garantit de trouver le plus court chemin du point de départ vers le point d'arrivée, tant qu'aucun des bords n'a un coût négatif. Dans le diagramme suivant, le carreau rose est le point de départ, le carreau bleu représente le but et l'exposition

de secteurs de sarcelle (d'élimination) les secteurs que l'algorithme de Dijkstra a parcouru. Les secteurs de sarcelle les plus légers sont les plus éloignés du point de départ et forment ainsi "la frontière" de l'exploration (Figure 2.4).

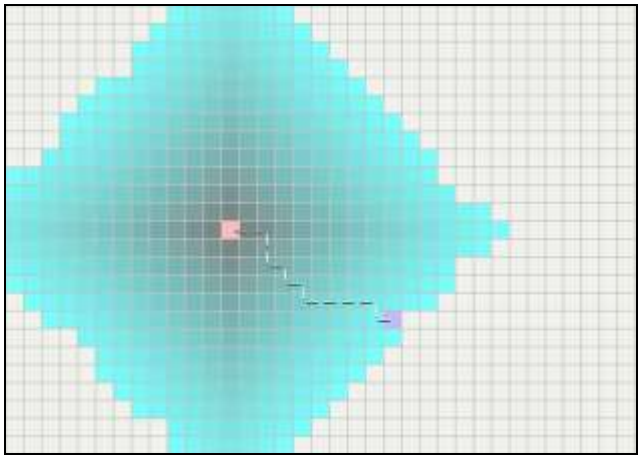

Figure 2.4 : L'algorithme de Dijkstra

2.2.3. La recherche du meilleur premier (BFS Breadth First Search)

L'algorithme travaille d'une façon semblable, sauf qu'il doit faire quelques évaluations (appelées heuristique). Au lieu de la sélection du sommet le plus proche au point de départ, il choisit le sommet le plus proche au but.

Inconvénient

BFS ne garantit pas de trouver le plus court chemin.

Avantage

Cet algorithme converge beaucoup plus rapidement que celui proposé par Dijkstra. Ceci est dû au fait qu'il emploie la fonction heuristique pour guider sa voie vers le but très rapidement.

Par exemple, si le but est au sud de la position de départ, l'algorithme BFS aura tendance à se concentrer sur des chemins orientés vers le sud. Dans le diagramme suivant, le jaune représente ces nœuds avec une haute valeur heuristique (coût élevé pour atteindre le but). Le noir représente, par contre, des nœuds avec une valeur heuristique basse (coût bas pour arriver au but). Il montre que l'algorithme BFS peut trouver des chemins très rapidement comparativement à l'algorithme de Dijkstra.

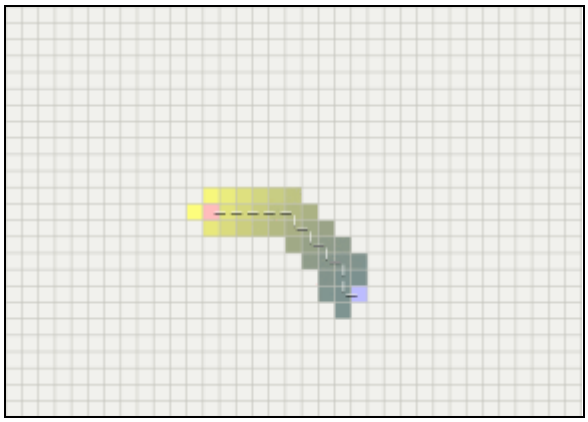

Figure 2.5 : La Recherche premier mieux (BFS)

Cependant, ces deux exemples illustrent le cas le plus simple. Quand la carte ne présente aucun obstacle, et le chemin le plus court est vraiment une ligne droite. Considérons l'obstacle concave comme décrit dans la section précédente. L'algorithme de Dijkstra travaille plus difficilement, mais garantit de trouver le plus court chemin (figure 2.6).

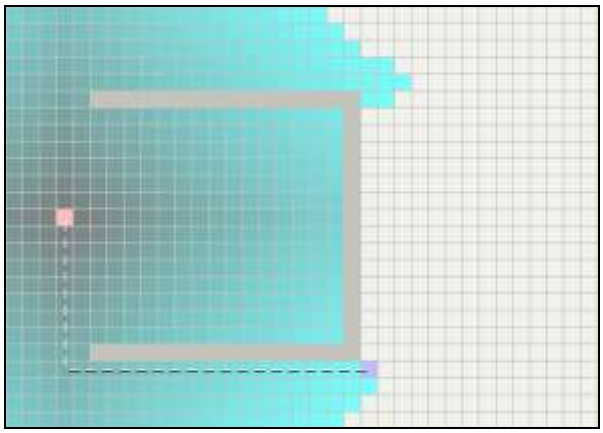

Figure 2.6 : Dijkstra avec des obstacles concaves

L'algorithme BFS nécessite, d'autre part, moins d'effort mais son chemin n'est pas visiblement le meilleur (figure 2.7).

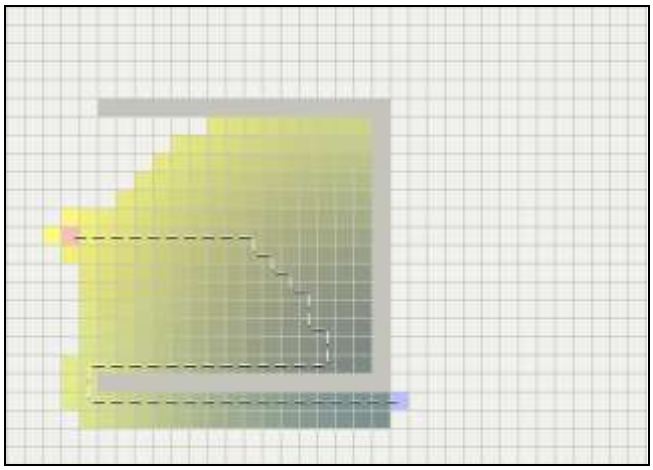

Figure 2.7 : BFS avec des obstacles concaves

Le trouble consiste en ce que BFS est passionné, et essaye de se déplacer vers le but même si ce n'est pas le bon chemin. Puisqu'il considère uniquement le coût lui permettant d'arriver au but, et ignore le coût du chemin parcouru jusqu'ici, il continue à avancer, même si le coût du chemin sur lequel il avance est devenu très élevé.

Il ne serait pas agréable de combiner le mieux d'entre toute les deux ?

L'algorithme A* a été développé en 1968 [Amia], pour combiner des approches heuristiques comme l'algorithme BFS, et des approches formelles comme l'algorithme de Dijsktra. C'est un peu commun dans des approches heuristiques, comme l'algorithme BFS donne d'habitude une façon approximative de résoudre un problème, sans garantir que l'obtention de la meilleure réponse. Cependant, l'algorithme A* est construit sur le sommet de l'heuristique, et bien que l'heuristique elle-même ne donne pas une garantie, l'algorithme A* peut garantir le chemin le plus court.

2.2.4. L'algorithme A*

Assez flexible et pouvant être employé dans un grand choix de contextes, l'algorithme A* a toujours été un choix des plus populaires [Amia].

De même que pour d'autres algorithmes de recherche graphique, l'algorithme A* permet d'effectuer des recherches virtuellement et ce dans un secteur énorme du domaine de recherche. Il est semblable à l'algorithme de Dijkstra, lequel pouvant être employé pour trouver le plus court chemin. Il est également proche de l'algorithme BFS, lequel peut employer une heuristique pour pouvoir se guider. Dans les cas les plus simples, il offre des vitesses de convergence proches de celles associées à l'algorithme BFS

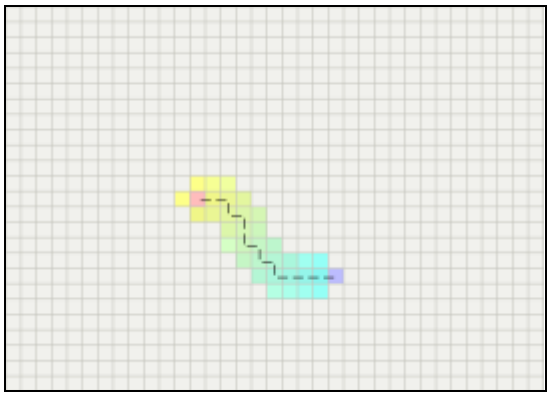

Figure 2.8 : La recherche graphique dans un secteur sans obstacles due plus cours chemin

Dans l'exemple avec un obstacle concave, A* trouve un chemin aussi bon que celui trouvé par l'algorithme de Dijkstra:

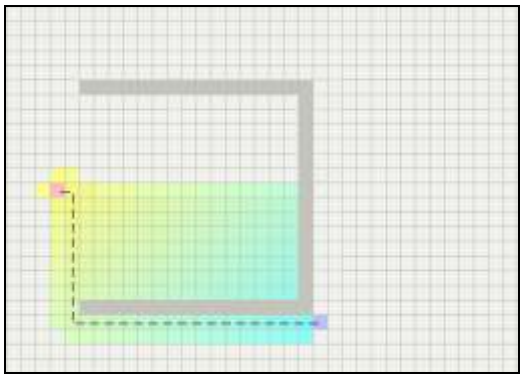

Figure 2.9 : l'algorithme A* avec un obstacle concave

Le secret lié au succès de l'algorithme A*, consiste dans le fait qu'il combine des morceaux d'information, les même que celles utilisées par l'algorithme de Dijkstra (favorisant le sommet qui est proche du point de départ), ainsi que l'information exploitée par l'algorithme BFS (favorisant le sommet qui est proche du but).

Dans la terminologie standard employée dans un algorithme A*:

g (n): représente le coût du chemin du point de départ à n'importe quel sommet n.

h (n): représente le coût heuristique estimé du sommet n au but.

Dans les diagrammes suscités, h (en jaune) représente les sommets qui sont loin du but, et g (bleu) représente les sommets loin du point de départ.

L'algorithme A* équilibre les deux tout en se déplaçant du point de départ au but. A chaque pas de la boucle principale, il examine le sommet n qui possède le coût le plus bas.

f (n) = g (n) + h (n).

2.2.4.1. Utilisation de la méthode heuristique *A**

L'heuristique peut être employée pour contrôler le comportement de l'algorithme A* [Amib].

- À une extrême, si (h(n) = 0), alors seul g(n) joue un rôle, l'algorithme A* conjugué à l'algorithme de Dijkstra, garantit l'existence du chemin le plus court.
- Si h(n) est toujours inférieur (ou égal) au coût de déplacement de n vers le but, on garantit donc que l'algorithme A* permet de trouver, lentement, le chemin le plus court.
- Si h(n) est exactement égal au coût de déplacement de n vers le but, alors l'algorithme A* suivra uniquement et très vite le meilleur chemin, et n'étendra jamais son domaine de recherche. Dans ce cas l'algorithme A* présente un comportement parfait.
- Si h(n) est parfois supérieur au coût de déplacement de n vers le but, dans ce cas on ne garantit pas que l'algorithme A* puisse trouver le plus court chemin, mais il peut converger plus rapidement.
- À l'autre extrême, si h(n) est très supérieur à g(n), dans ce cas seul h(n) joue un rôle dans la convergence de l'algorithme A* par rapport à l'algorithme BFS.

On a, donc, une situation intéressante dans ce qu'on peut décider, et dans la façon dont on peut exploiter l'algorithme A*. Dans ce cas, on peut prétendre, l'obtention des plus courts chemins et ce d'une manière rapide. Si on est trop bas, on continue donc à obtenir les chemins les plus courts, mais d'une manière moins rapide. Si on est trop haut, on abandonne donc la recherche des chemins les plus courts, mais l'algorithme A* convergera plus rapidement.

Dans un jeu, cette propriété de l'algorithme A* peut être très utile. Par exemple, il peut exister des situations où il n'importe pas de trouver le meilleur chemin et qu'un bon chemin soit suffisant. Pour changer l'équilibre entre g(n) et h(n), on peut procéder au changement de chacune des deux quantités.

2.2.4.2. Vitesse ou exactitude ?

La capacité de l'algorithme A* de varier son comportement est basée sur l'heuristique, et les fonctions de coût peuvent être très utiles dans la conception des jeux. La différence entre la vitesse et l'exactitude, peut être exploitée pour concevoir des jeux plus rapides. Pour la plupart des jeux, on n'a pas vraiment besoin du meilleur chemin entre deux points, on a juste besoin d'un chemin permettant de rallier les deux points en étant assez proche.

2.2.4.3. Échelle

L'algorithme A* calcule f(n) = g(n) + h(n). Pour ajouter les deux valeurs, ils doivent être à la même échelle.

2.2.4.4. Heuristique exacte

Si l'heuristique est exactement égal à la distance, le long du chemin optimal, on verra que l'algorithme A* étend très peu de nœuds, comme dans le diagramme montré dans la section suivante. Ce qui arrive à l'intérieur de l'algorithme A* est qu'il calcule f(n) = g(n) + h(n) au niveau de chaque nœud. Quand h(n) correspond exactement à g(n), la valeur de f(n) ne change pas tout au long du chemin. Tous les nœuds qui ne sont pas sur le juste chemin, auront une valeur f plus importante. [Amib]

2.2.4.5. Heuristique exacte pré calculée

Une façon de construire une heuristique exacte, est de pré-calculer la longueur du plus court chemin entre chaque paire de points. Ce n'est pas faisable pour la plupart des cartes de jeu, cependant, il y a des méthodes qui permettent de se rapprocher de cette heuristique:

Il y'a lieu d'ajouter alors, à l'heuristique h la fonction qui permet de calculer le coût pour aller de n'importe quel emplacement jusqu'à la voie voisine. L'heuristique finale devient:

H (n) = h ' (n, w1) + distance (w1, w2), h ' (w2, but).

On peut également opter pour une heuristique meilleure mais plus coûteuse, en évaluant toutes les paires w1, w2 qui proches du nœud et du but, respectivement.

Heuristique linéaire exacte.

Dans une circonstance spéciale, on peut opter pour une heuristique exacte sans pré-calcul. Si on a une situation sans obstacles ni de terrain lent, le plus court chemin du point de départ vers le but doit être en fait une ligne droite. En définitive, cette heuristique correspond à une heuristique exacte.

2.2.4.6. Heuristique pour des cartes de réseaux

Sur un réseau, il y a des fonctions heuristiques bien connues pouvant être employées. [Amib]

- **Distance de Manhattan**

La norme utilisée dans l'heuristique est la distance du Manhattan. Le coût pour le déplacement d'un espace vers un espace adjacent est D. Donc, l'heuristique dans notre jeu doit être:

$$h(n) = D * (abs (x_n - x_{but}) + abs (y_n - y_{but})).$$

L'échelle devant être utilisée est celle qui correspond à la fonction du coût.

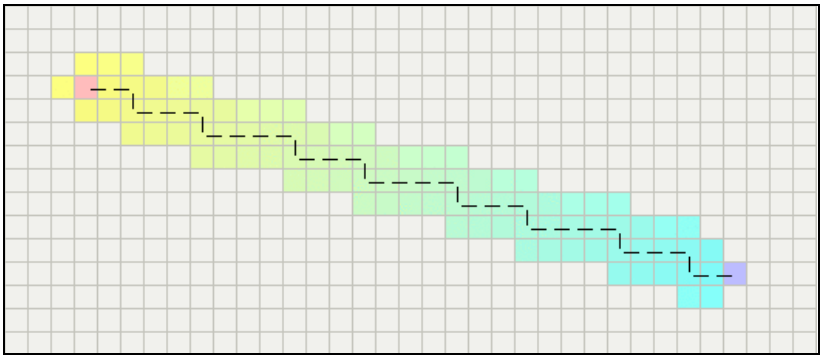

Figure 2.10 : La distance du Manhattan

- **Distance diagonale**

Si sur notre carte nous permettons le mouvement diagonal nous avons besoin d'un différent heuristique. La distance du Manhattan pour (4 est, 4 nord) sera 8*D. Cependant, nous pourrons simplement nous déplacer (4 nord-est) au lieu de cela, donc l'heuristique doit être 4*D. Cette fonction manipule des diagonales, assurant qu'et directement et le mouvement diagonal coûte D:

$$h(n) = D * max(abs (x_n - x_{but}) + abs (y_n - y_{but})).$$

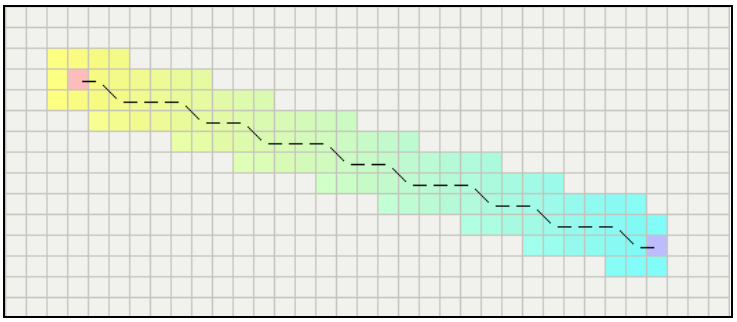

Figure 2.11 : Distance diagonale

Si le coût en diagonale du mouvement n'est pas D, mais une quantité semblable à $D_2 = \sqrt{2} * D$, ladite heuristique ne sera pas juste pour nous. Nous voudrions, au lieu de cela, quelque chose de plus sophistiqué:

h_diagonale (n) = minute (abs (x_n - x_{but}) + abs (y_n -y_{but}))

h_straight (n) = (abs (x_n - x_{but}) + abs (y_n -y_{but}))

h (n) = D_2 * h_diagonale (n) + D * (h_straight (n) – 2 * h_diagonale (n)))

Ici, nous calculons:

h_diagonale (n) = le numéro de pas que nous pouvons prendre le long d'une diagonale.

h_straight (n) = la distance du Manhattan et combiner ensuite deux en considérant tous les pas diagonaux pour coûter à D_2 et ensuite tout le maintien directement marche (notez que c'est le numéro de droit intervient la distance du Manhattan, moins deux directement pas pour chaque pas diagonal que nous avons pris au lieu de cela) coûtent D.

- **La distance Euclidienne**

Si les unités peuvent se déplacer suivant un angle (au lieu des directions du réseau), on donc probablement employer une distance de ligne droite: [Amic]

$h(n) = D * \sqrt{(x_n - x_{but})^2 + (y_n - y_{but})^2}$

Cependant, si c'est le cas, on peut avoir donc des difficultés pour l'utilisation directe de l'algorithme A* parce que la fonction du coût g ne correspondra plus à la fonction heuristique h.

De plus, pour cette méthode, la distance Euclidienne est plus courte que la distance de Manhattan ou la distance diagonale. On obtient toujours le chemin les plus courts, cependant l'algorithme A* prendra plus de temps à l'obtenir.

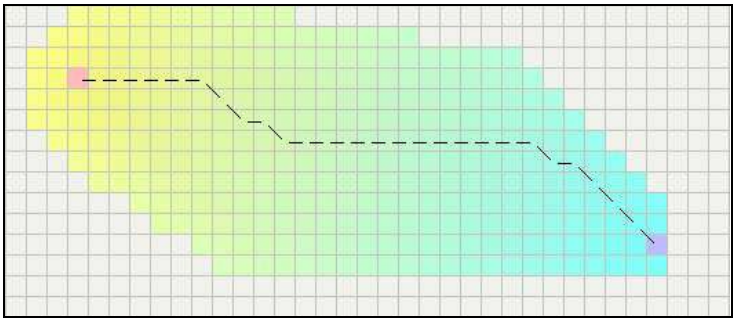

Figure 2.12 : Distance Euclidienne

- **Liens de casse**

Les liens dans l'heuristique peuvent parfois mener à une exécution inadéquate. Quand plusieurs chemins ont la même valeur de f, ils sont tous explorés bien que l'on a réellement besoin d'explorer l'un d'entre eux:

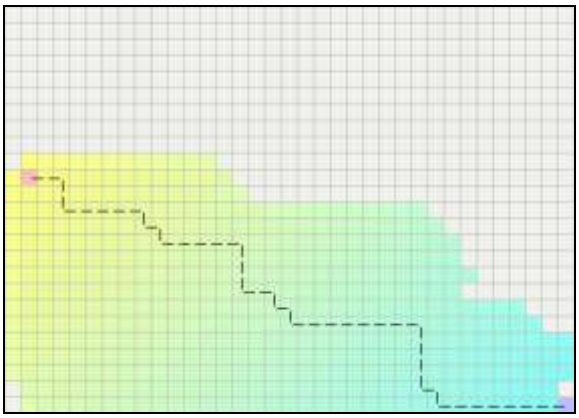

Figure 2.13 : Liens de casse dans les valeurs de f

Pour résoudre ce problème, nous pouvons ajouter un petit interrupteur de lien au niveau de l'heuristique. Cet interrupteur de lien doit être déterminé en rapport avec le sommet et doit faire de telle sorte que les valeurs de **f** soient différentes. C'est en fait une exploration des valeurs possibles de **f**.

Une façon de casser des liens est de pousser légèrement le coude l'échelle de la fonction h. Si l'échelle est peu élevée, donc f augmentera comme au fur et à mesure que l'on se déplace vers le but. Malheureusement, cela signifie que l'algorithme A* préférera étendre les sommets proches du point de départ au lieu d'étendre les sommets proches du but. Ainsi, au lieu d'augmenter l'échelle de la fonction h même légèrement (0.1 %, à titre d'exemple), l'algorithme A* préférera étendre les sommets près du but.

Heuristique * = (1.0 + p).

Le facteur p doit être choisi tel que: p < (coût minimum pour un pas) / (la longueur maximale de chemin attendue).

En garantissant que l'on ne s'attend pas à ce que les chemins soient d'une longueur supérieur à 1000 pas, on peut choisir p = 1/1000. Le résultat de ce coup de coude, cassant lien est une nouvelle variante de l'algorithme A* qui explore une partie moindre du champ de recherche par rapport aux variantes précédentes (Figure 4.14):

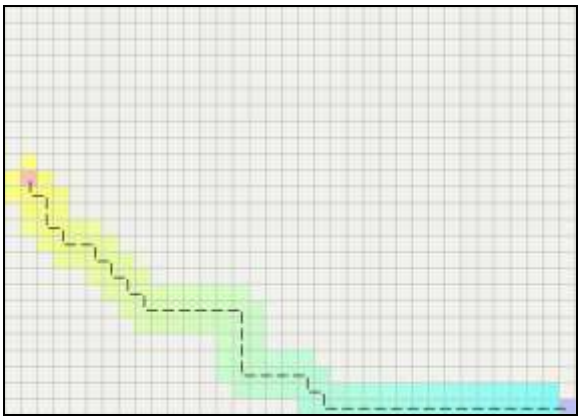

Figure 2.14 : Graduation cassant lien supplémentaire à heuristique

Lorsqu'il il y a des obstacles bien sûr, il doit toujours procéder à une exploration pour trouver une voie autour entre ces obstacles. Cependant, notons qu'une fois l'obstacle détourné, l'algorithme A* explore un champ de possibilités plus restreint (Figure 2.15).

Une façon différente de casser des liens est de préférer les chemins qui sont le long de la ligne droite du point de départ vers le but:

$Dx_1 = x_{current} - x_{goal}$
$Dy_1 = y_{current} - y_{goal}$
$Dx_2 = x_{start} - x_{goal}$
$Dy_2 = y_{start} - y_{goal}$
$Croix = abs\ (dx_1*dy_2 - dx_2*dy_1)$
$Heuristique\ + = cross*0.001$

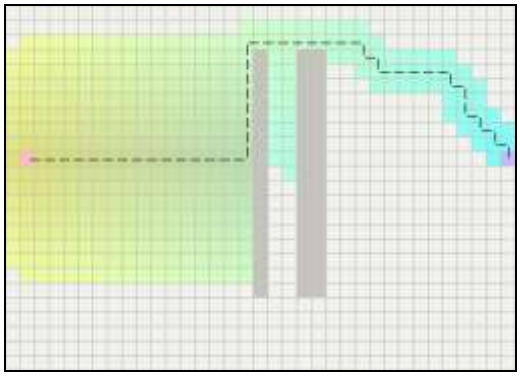

Figure 2.15 : La graduation cassant lien supplémentaire heuristique travaille gentiment avec des obstacles

Ce code calcule le produit vectoriel mutuel, entre le début au vecteur de but, et le point actuel au vecteur de but. Quand ces vecteurs ne s'alignent pas, le produit mutuel sera plus grand. Le résultat est que, ce code donnera une préférence légère à un chemin, qui se trouve le long de la ligne droite liant le point de départ et le but. Ainsi, lorsqu'il n'y a aucun obstacle,

l'algorithme A*, non seulement explore moins d'espace de recherche, mais génère des chemins plus réalistes (Figure 2.16).

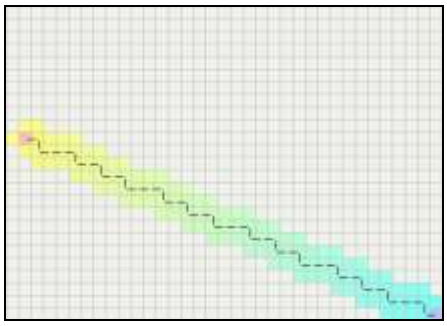

Figure 2.16 : Le produit mutuel cassant lien supplémentaire à .heuristique, produit de jolis chemins

Cependant, du fait que cet interrupteur de lien emprunte les chemins qui suivent la ligne droite reliant le point de départ au but, des artefacts peuvent survenir (discrètes) lorsque l'acteur rencontre des obstacles (Figure 2.17) (notez que le chemin est toujours optimal; cela semble juste étrange).

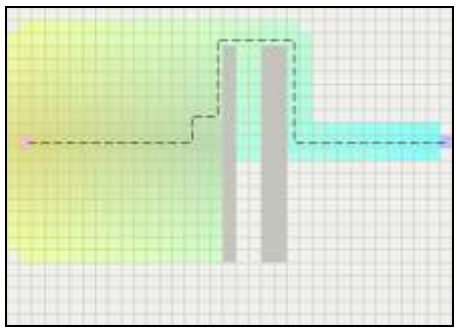

Figure 2.17 : Produit mutuel cassant un lien supplémentaire à heuristique avec obstacles (artefact)

Pour un mode interactif explorer l'amélioration de cet interrupteur de lien, (voir l'applette de James Macgill A*). L'utilisation "Claire" purifie la carte et choisit deux points à l'opposé des coins de la carte. Quand méthode classique de l'algorithme A* est utilisée, l'effet des liens est mis en évidence. D'un autre côté, lorsqu'on utilise la méthode "Fudge" (ou d'évitement), on constate l'effet du produit mutuel supplémentaire à l'heuristique.

Il existe également une autre façon de casser les liens, elle consiste à construire soigneusement la file d'attente des priorités de l'algorithme A* pour que de nouvelles insertions avec une valeur de f spécifique soient toujours classées comme étant les meilleures (inférieur) que les vieilles insertions avec la même valeur de f.

- **Recherche d'un secteur**

Si nous voulons chercher une tâche (un endroit) près d'un but quelconque, au lieu d'un espace particulier, nous pourrions construire h'(x) une heuristique qui est le minimum de h1 (x), h2 (x), h3 (x)... où h1, h2, h3 est l'heuristique à chacune des tâches (endroits) voisines.

Cependant, une voie plus rapide consiste à laisser l'algorithme A* chercher le centre du secteur de but. Une fois que l'on obtient un espace voisin du jeu OUVERT, nous pouvons arrêter et construire un chemin.

2.3. Recherche de chemin

Le déplacement d'humanoïdes à l'intérieur d'un environnement virtuel est, le plus souvent, lié à la volonté d'atteindre une position particulière. Il s'agit du problème traité par la recherche de chemins: comment trouver depuis une position donnée allant vers une autre position voulue, en évitant les obstacles statiques de l'environnement. Pour générer un chemin plausible, l'approche la plus adaptée est de rechercher un chemin minimisant certains critères comme la distance, le coût énergétique,… Même si cette recherche de chemin optimal n'est pas forcement corrélée avec la navigation humaine [WM03], elle permet de générer des chemins plausibles évitant des détours excessifs qui s'avèrent beaucoup moins réalistes. Deux grands types d'approches peuvent être distingués:

- Les approches à base de graphe;
- Et les approches à base de champs de potentiel.

2.3.1. Calcul sur les graphes

L'utilisation de l'algorithme des graphes pour le calcul d'un chemin nécessite l'utilisation d'une discrétisation de l'espace car un nœud du graphe est assimilé à un point de l'environnement. Les arcs représentent alors une notion d'accessibilité entre deux points et sont values par une estimation de l'effort à fournir pour relier ces points (il s'agit le plus souvent de la distance). La recherche d'un chemin va donc se résumer à la recherche d'une suite de nœuds, reliés par des arcs dont la somme des valeurs associées minimise le coût global du chemin.

2.3.1.1. Discrétisation de l'espace et construction du graphe

Avant de pouvoir appliquer les algorithmes de calcul sur les graphes, il faut disposer d'une représentation de l'environnement sous cette forme. Les cartes de cheminement [Lam03] fournissent directement cette information: chaque point clef de l'environnement devient un nœud alors que chaque arc représente la relation d'accessibilité entre ces nœuds. Dans le cas d'une discrétisation de l'espace sous la forme de cellules, les informations de cellules et de connexité sont utilisées pour générer une carte de cheminement. Traditionnellement, les points clefs sont choisis soit comme étant le centre de la cellule soit comme appartenant aux bords franchissables de cette même cellule. Les arcs du graphe sont alors construits en fonction des relations de connexité entre les cellules. La figure 2.18 montre les deux formes de graphes de cheminement qui peuvent être construites en utilisant comme base une triangulation de Delaunay contrainte; la figure de gauche utilise les centres de gravité des triangles comme points clefs; la figure de droite utilise le milieu des segments non contraints reliant deux triangles.

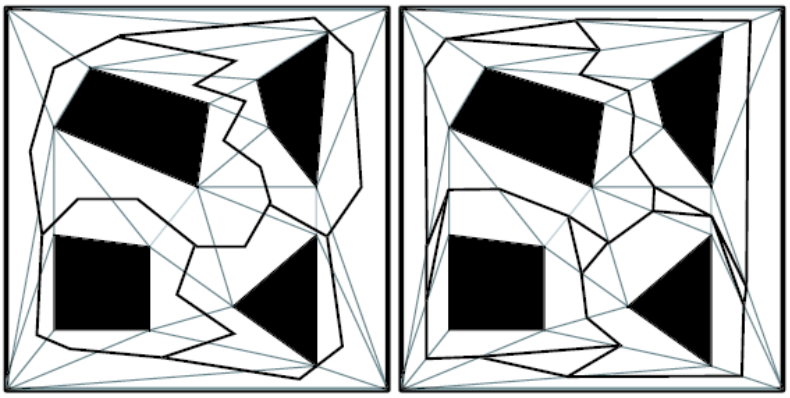

Figure 2.18 : Exemple de cartes de cheminement générées à partir d'une discrétisation par triangulation de Delaunay contrainte

Une autre approche, utilisée dans le cadre d'environnement intérieur, consiste à travailler sur les graphes d'adjacence des cellules (extraites de la subdivision spatiale) pour caractériser des espaces (pièces, couloirs, portes,…) [Lau89]. Chaque espace étant un agrégat de cellules, le graphe topologique des espaces est plus abstrait est donc moins complexe que le précédent. Cela permet de définir une stratégie de navigation hiérarchique s'appuyant sur le graphe topologique et raffinant les calculs, à posteriori, à l'intérieur de chaque espace.

2.3.1.2. Les algorithmes utilisés

Diverses formes d'algorithmes de calcul sur les graphes peuvent être utilisées pour effectuer le calcul du plus court chemin. Parmi celles-ci, l'algorithme de Dijkstra permet de trouver l'ensemble des meilleurs chemins issus d'un sommet et permettant d'atteindre les autres sommets du graphe. Schématiquement, cet algorithme stocke, pour chaque nœud exploré, sa distance par rapport à l'origine et son prédécesseur dans le chemin ayant permis l'obtention de cette distance. L'algorithme commence par le nœud d'origine en explorant successivement tous les successeurs tout en privilégiant ceux qui sont les plus
proches de l'origine. Cet algorithme travaille uniquement sur la structure du graphe sans utiliser d'informations supplémentaires. De ce fait, pour trouver le plus court chemin d'un point A à un point B, sachant que la longueur réelle du chemin de A à B est D, il va explorer tous les nœuds dont la distance est inférieure à D avant de pouvoir fournir un chemin.

Dans la mesure où les problèmes de planification de chemin, il est possible d'exploiter des propriétés sur l'environnement, l'algorithme A* lui est souvent préféré. La démarche est légèrement différente, chaque nœud exploré se voit attribuer une valeur correspondant à l'estimation du coût total du chemin passant par ce nœud. Cette valeur est calculée en fonction de la distance réelle par rapport à l'origine plus une estimation (fournie par une heuristique) de sa distance au but. La valeur $v(n_k)$, associée au nœud n_k, a est évaluée par l'expression suivante:

$v(n_k) = d(n_k,o) + h(n_k, b)$

où $d(n_k,o)$ représente la distance réelle calculée entre l'origine et le sommet n_k et $h(n_k, b)$ la distance estimée(heuristique) du but. L'algorithme s'initialise avec le nœud de départ et stocke dans une file triée de nœuds (suivant l'ordre décroissant de la distance estimée au but), que l'on peut qualifier d'ouverts car ils n'ont pas encore été explorés. L'algorithme explore ainsi systématiquement le nœud promettant d'obtenir le plus court chemin. La puissance de convergence de cet algorithme dépend de la qualité de l'heuristique h. Dans le cas de la recherche du plus court chemin dans un environnement, cette heuristique est le plus souvent une estimation de la distance. Un certain nombre de variantes existent par rapport à l'algorithme A*. Parmi celles-ci l'algorithme ABC (A* with Bounded Cost) est proposé par Logan[LA98]. Il s'agit d'une génération de l'algorithme A* permettant d'ajouter des contraintes molles à respecter (contraintes de limitation de temps et d'énergie par exemple) lors de la planification.

L'algorithme IDA* (Iterative-Deepening A*) utilise une approche différente en effectuant une recherche en profondeur d'abord, supervisée par une heuristique [Kor85]. Dans un premier temps, une borne de longueur de chemin B est initialisée avec la distance estimée au but. L'exploration commence par le nœud d'origine du chemin et explore successivement tous les successeurs, ainsi que les successeurs des successeurs…, de ce nœud. Cette recherche s'arrête dans deux cas: soit le chemin est trouvé, dans ce cas il s'agit du chemin optimal, soit un nœud est trouvé tel que sa distance réelle depuis l'origine sommée à cette distance estimée au but soit supérieur à B. Dans le cas où aucun chemin n'est trouvé durant une exploration, la borne B est remise à jour avec la plus petite estimation de la distance au but trouvée au cours des explorations. Certaines améliorations en termes de temps d'exécution peuvent être retenues en utilisant un cache de taille fixe des estimations déjà calculées pour les nœuds précédemment explorés [RM94]. Ces estimations permettent un étalage plus rapide de l'arbre de recherche.

En règle générale, l'algorithme de Dijkstra est peu utilisé dans le cadre de la recherche de chemin car il s'agit d'un domaine dans lequel il est souvent possible d'utiliser une heuristique qui réduit grandement le coût de la recherche. Le choix entre l'algorithme A* et l'algorithme IDA* est plus difficile. En général, l'algorithme IDA* est préféré à l'algorithme A* dans le cas de domaines de taille exponentielle car la taille de la liste triée conservée par l'algorithme A* devient elle-même exponentielle[RM94]. Cependant, dans le cadre de la planification de chemins, cette considération n'a pas vraiment de signification si on considère que l'on dispose déjà du graphe en mémoire. D'un point de vue théorique, ces deux algorithmes possèdent des complexités équivalentes mais l'algorithme IDA* nécessite moins de d'espace mémoire.

2.3.2. Méthodes à champs de potentiel

Contrairement aux approches utilisant les graphes, les méthodes à base de champs de potentiel ne travaillent pas forcément sur une représentation discrète de l'environnement. Les obstacles sont vus comme des émetteurs de forces répulsives, alors que le but est un attracteur [Lat91, Rei97]. Un potentiel est donc défini pour chaque point de l'environnement comme la somme des potentiels de répulsion liés aux obstacles avec le potentiel d'attraction lié au but. L'entité étant localisée dans l'environnent, la direction à prendre pour converger vers son but est alors opposée au gradient de potentiel défini en ce point.

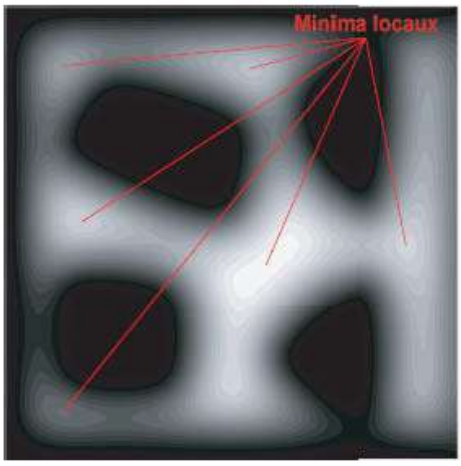

Figure 2.19 : Une carte de champ de potentiel, les parties noires représentent les obstacles, les parties plus claires les zones de navigation. Les dégradés de couleurs représentent pour leur part la valeur du potentiel associé au point de l'environnement. Cette image montre six minima locaux caractérisés par des couleurs plus claires.

Ces méthodes s'avèrent simples et efficaces mais posent le problème de chemin local (figure 2.19). La méthode de navigation associée pousse l'entité à se déplacer vers le minimum local le plus proche qui n'est pas forcément le minimum de la surface et en conséquence qui ne représente pas le but. Pour pallier ce type de problème, des méthodes à base de marche aléatoire sont utilisées. Par exemple, dans RPP (Random Path Planner) [BL91], lorsqu'un minimum local est atteint, un ensemble de configurations aléatoires sont tirées puis testées avec une phase de sortie du minimum et une phase de convergence vers le prochain minimum. Les informations sont alors stockées dans un graphe dont les nœuds sont les minima locaux et les arcs traduisent des chemins entre deux minima. D'autres méthodes, à base de tirage aléatoire de direction à suivre, existent [CRR01].

2.4. Conclusion

Dans ce chapitre, nous avons présenté le principe du pathfinding, et les divers algorithmes mis à notre disposition pour le gérer, pour la découverte des plus courts chemins et des distances dans des graphes plats qui exploitent la topologie particulière de l'entrée graphique. Nous avons vu aussi, les différentes étapes, puis les points forts et les points faibles de chacun des algorithmes. L'algorithme A* prend les avantages de l'algorithme BFS et de l'algorithme Dijkstra. Il est ainsi semblable à l'algorithme de Dijkstra dans lequel il peut être employé pour trouver le plus court chemin, et également semblable à l'algorithme BFS dans lequel il peut employer un heuristique pour se guider. Nous avons, par ailleurs, présenté l'heuristique liée à l'algorithme A*, et enfin les différentes variantes de l'algorithme A*.

Les méthodes de recherche de chemin présentées dans ce chapitre, ne sont pas très adaptées à la simulation comportementale. Les comportements induits par les tirages aléatoires, s'ils sont acceptables pour des robots, ils ne le sont pas pour des humanoïdes car ils sont en dehors de la logique de la navigation humaine