Recherche de chemin par l'algorithme A*

par khayyam90 (Mon site)

Date de publication : 23/08/2006

Dernière mise à jour : 26 février 2008

Ce tutoriel explique le fonctionnement de l'algorithme A* pour rechercher un chemin dans un graphe. Il sera illustré par un exemple en C++.

I - Introduction

Vous souhitez écrire un programme qui puisse se débrouiller tout seul pour amener un objet d'un point à un autre, le plus rapidement possible et en évitant les obstacles éventuels, l'algorithme A* (prononcer A star) est fait pour vous.

C'est un **algorithme** de recherche de chemin dans un **graphe**. C'est l'un des plus efficaces en la matière. Il ne donne pas toujours la solution optimale mais il donne très rapidement une bonne solution.

II - Principe de l'algorithme

Au premier abord, on pourrait se dire que pour trouver un chemin d'un point à un autre il faut commencer par se diriger vers la destination. Et bien ... c'est justement cette idée qu'utilise l'algorithme A*. L'idée est très simple : à chaque itération (oui, c'est un algorithme itératif), on va tenter de se rapprocher de la destination, on va donc privilégier les possibilités directement plus proches de la destination, en mettant de côté toutes les autres.

Toutes les possibilités ne permettant pas de se rapprocher de la destination sont mises de côté, mais pas supprimées. Elles sont simplement mises dans une liste de possibilités à explorer si jamais la solution explorée actuellement s'avère mauvaise. En effet, on ne peut pas savoir à l'avance si un chemin va aboutir ou sera le plus court. Il suffit que ce chemin amène à une impasse pour que cette solution devienne inexploitable.

L'algorithme va donc d'abord se diriger vers les chemins les plus directs. Et si ces chemins n'aboutissent pas ou bien s'avèrent mauvais par la suite, il examinera les solutions mises de côté. C'est ce retour en arrière pour examiner les solutions mises de côté qui nous garantit que l'algorithme nous trouvera toujours une solution (si tenté qu'elle existe, bien sûr).

On peut donc lui donner un terrain avec autant d'obstacles qu'on veut, aussi tordus soient-ils, s'il y a une solution, A* la trouvera.

II-1 - Les listes A*

A* utilise deux listes, ces listes contiennent des points. Pour être plus général, on peut même dire que ces listes contiennent des noeuds d'un graphe, lequel graphe représentant notre terrain.

La première liste, appelée *liste ouverte*, va contenir tous les noeuds étudiés. Dès que l'algorithme va se pencher sur un noeud du graphe, il passera dans la liste ouverte (sauf s'il y est déjà).

La seconde liste, appelée *liste fermée*, contiendra tous les noeuds qui, à un moment où à un autre, ont été considérés comme faisant partie du chemin solution. Avant de passer dans la liste fermée, un noeud doit d'abord passer dans la liste ouverte, en effet, il doit d'abord être étudié avant d'être jugé comme bon.

II-2 - Déroulement de l'algorithme

Pour déterminer si un noeud est suceptible de faire partie du chemin solution, il faut pouvoir quantifier sa qualité. Vous êtes entièrement libre de ce côté là, vous pouvez mesurer la pertinence d'un noeud de la manière qui vous plait. Néanmoins, une méthode souvent utilisée et qui donne de bons résultats est de mesurer l'écartement entre ce noeud et le chemin à vol d'oiseau. On calcule donc la distance entre le point étudié et le dernier point qu'on a jugé comme bon. Et on calcule aussi la distance entre le point étudié et le point de destination. La somme de ces deux distances nous donne la qualité du noeud étudié. Plus un noeud à une qualité faible, meilleur il est.

Vous pouvez calculer ces distances de la manière que vous voulez, distance euclidienne, distance de Manhattan ou autre, elles peuvent convenir.

Attention cependant avec la distance euclidienne : elle fait intervenir une racine carrée et donc des nombres flottants, beaucoup plus lents à manipuler que des nombres entiers. Vous pouvez donc aussi manipuler la carré de la distance euclidienne. Les résultats peuvent être légèrement différents, mais c'est beaucoup plus rapide.

La caractère minorant ou non de l'heuristique utilisée pour calculer la distance va également jouer sur la convergence optimale ou non de l'algorithme : l'utilisation d'une heuristique minorante fournira le résultat optimal.

On ne sait pas grand chose du chemin solution si ce n'est que le point qui pourra nous rapprocher de la solution est un point voisin du point que nous étudions. On va donc étudier chacun des noeuds voisins du noeud courant pour déterminer celui qui a le plus de chances de faire partie du chemin solution.

La recherche du chemin commence par le premier point, en étudiant tous ses voisins, en calculant leur qualité, et en choisissant le meilleur pour continuer. Chaque point étudié est mis dans la liste ouverte et le meilleur de cette liste passe dans la liste fermée, il va servir de base pour la recherche suivante.

Pour déterminer le meilleur point pour continuer le chemin, il ne faut pas uniquement chercher celui qui a la meilleure qualité dans ses voisins, mais dans toute la liste ouverte. C'est comme ça qu'on pourra abandonner un chemin qui avait l'air bon au début et qui ne l'était pas.

Ainsi, à chaque itération on va regarder parmi tous les noeuds qui ont été étudiés (et qui n'ont pas encore été choisis) celui qui a la meilleure qualité. Et il est tout à fait possible que le meilleur ne soit pas un voisin direct du point courant. Cela signifira que le point courant nous éloigne de la solution et qu'il faut corriger le tir.

L'algorithme s'arrête quand la destination a été atteinte ou bien lorsque toutes les solutions mises de côté ont été étudiées et qu'aucune ne s'est révélée bonne, c'est le cas où il n'y a pas de solution.

II-3 - Détermination des noeuds voisins

Si on considère notre domaine de recherche comme un graphe, tous les noeuds voisins d'un noeud courant sont les noeuds adjacents. Mais si on considère notre domaine de recherche comme une carte (ou une image), ce sont tous les points adjacents (en haut, en bas, à gauche, à droite et les points en diagonale) sauf si ceux-ci contiennent des obstacles infranchissables (mur, montagne, rivière ...).

Donc, avant de se lancer dans l'étude de la qualité de chacun des noeuds adjacents, il ne faut prendre que ceux qui sont vraiment utilisables. Il faut aussi mettre de côté tous les noeuds déjà présents dans la liste ouverte ou dans la liste fermée. Et pour être plus précis, je dirais qu'il ne faut pas prendre un point s'il est déjà dans la liste ouverte, à moins qu'il ne soit meilleur, auquel cas on va mettre à jour la liste ouverte.

Ce qu'il faut regarder sur chacun des noeuds voisins potentiels :

- Est-ce un obstacle ? si oui, on oublie ce noeud.
- Est-il dans la liste fermée ? si oui, ce noeud a déjà été étudié ou bien est en cours d'étude, on ne fait rien.
- Est-il dans la liste ouverte ? si oui, on calcule la qualité de ce noeud, et si elle est meilleure que celle de son homologue dans la liste ouverte, on modifie le noeud présent dans la liste ouverte.
- Sinon, on l'ajoute dans la liste ouverte et on calcule sa qualité.

II-4 - Notion de parent d'un noeud

Chaque noeud a un parent, c'est par lui qu'on arrive là où on est. Le parent représente le meilleur chemin entre deux noeuds. Le parent est très important à la fin de l'algorithme, pour retrouver ton chemin.

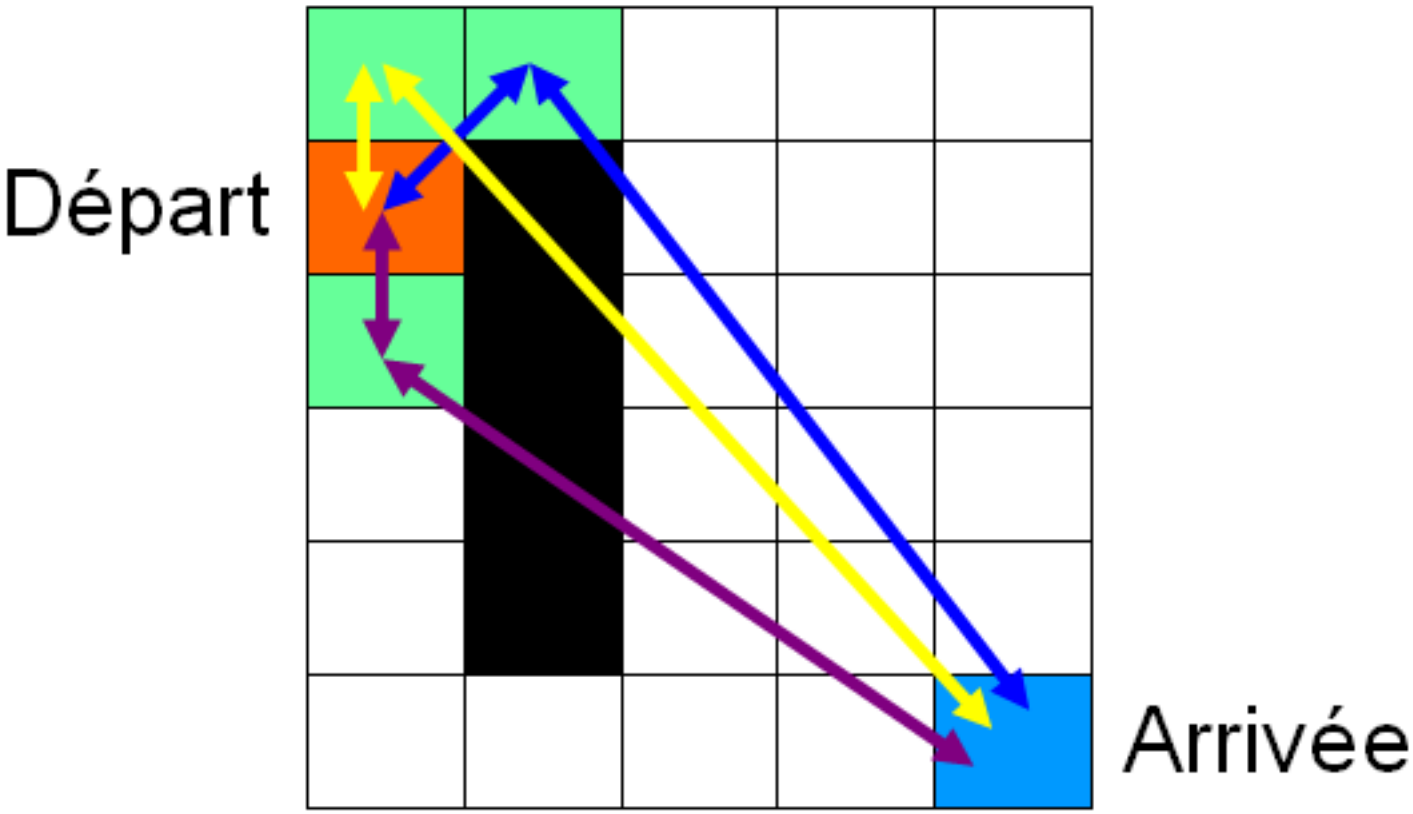
Il joue aussi un rôle important lors de la mise à jour d'un noeud dans la liste ouverte. Nous avons vu qu'il fallait mettre à jour la liste ouverte dans le cas où un noeud avait une meilleure qualité que ce même noeud dans la liste ouverte. Nous avons mis à jour la qualité du noeud dans la liste ouverte, mais il faut aussi mettre à jour son parent, pour bien sigifier que cette qualité est possible par tel parent précis.

Une fois que la destination a été atteinte, il faut retrouver le chemin en suivant à chaque fois les parents des noeuds présents dans la liste fermée. On remonte le fil jusqu'à arriver au point de départ.

II-5 - Résumé des étapes

- On commence par le noeud de départ, c'est le noeud courant
- On regarde tous ses noeuds voisins
- si un noeud voisin est un obstacle, on l'oublie
- si un noeud voisin est déjà dans la liste fermée, on l'oublie
- si un noeud voisin est déjà dans la liste ouverte, on met à jour la liste ouverte si le noeud dans la liste ouverte a une moins bonne qualité (et on n'oublie pas de mettre à jour son parent)
- sinon, on ajoute le noeud voisin dans la liste ouverte avec comme parent le noeud courant
- On cherche le meilleur noeud de toute la liste ouverte. Si la liste ouverte est vide, il n'y a pas de solution, fin de l'algorithme
- On le met dans la liste fermée et on le retire de la liste ouverte
- On réitère avec ce noeud comme noeud courant jusqu'à ce que le noeud courant soit le noeud de destination.

Voici une illustration d'une itération de l'algorithme. On souhaite aller du point orange au point bleu. Les noeuds voisins de la case orange sont les cases marquées en vert, ils passent en liste ouverte. Et de chacune d'elles on calcule les couts G et H pour aller à la case orange et pour aller à la destination. J'ai choisi la distance à vol d'oiseau. Et la case qui aura le cout le plus faible sera la case verte du dessous, elle va donc passer en liste fermée. L'algorithme va réitérer à partir de cette case.



III - Implémentation

Tout ceci est très théorique, je vais expliquer mon implémentation en C++, elle pourra éclaircir mes propos. Je choisi de travailler sur une image, donc sur un espace discret. Les noeuds voisins d'un pixel sont donc les pixels directement voisins.

Boost.graph implémente l'algorithme A pour les parcours de graphe, ça peut donc être une alternative très intéressante si vous developpez en C++.*

III-1 - Choix des structures de données

Il nous faut de quoi représenter la liste ouverte et la liste fermée. Un noeud ne peut apparaître qu'une fois dans chaque liste. S'il y a des doublons, c'est une erreur dans l'implémentation. J'ai choisi de représenter mes listes par des std::map. Chaque noeud ayant pour clé une position (x,y) représentée par un std::pair<int,int>. Un noeud n'étant rien de plus que le regroupement des informations suivantes :

- le coutG (le cout pour aller du point de départ au noeud considéré)
- le coutH (le cout pour aller du noeud considéré au point de destination)
- le coutF (somme des précédents mais mémorisé pour ne pas le recalculer à chaque fois)
- le parent, représenté par ses coordonnées (suffisantes pour retrouver ensuite l'élément dans les listes car les coordonnées sont les clés dans les std::maps)

Le parent peut être considéré comme un pointeur vers le noeud parent. Mais il ne faut en aucun cas le représenter avec un pointeur mémoire. En effet, rien ne nous dit que la STL ne va pas réarranger l'espace mémoire des maps lors des ajouts/suppression successifs, modifiant les adresses mémoires de nos objets (et par la même, déréférençant nos pointeurs). C'est la raison pour laquelle j'ai choisi un système d'adressage indépendant des adresses mémoires : les pair<int,int>

Ce qui nous donne en C++ :

```
struct noeud{
    float cout_g, cout_h, cout_f;
    std::pair<int,int> parent;    // 'adresse' du parent (qui sera toujours dans la map fermée)
};
```

Il faudra aussi une structure représentant un point (x,y), bien qu'on aurait pu prendre un simple std::pair<int,int> :

```
struct point{
    int x,y;
};
```

Le type représentant nos listes se résume donc à un simple

```
typedef map< pair<int,int>, noeud> l_noeud;
```

III-2 - Outils nécessaires

Je vais avoir besoin de plusieurs traitements annexes : les fonctions d'acquisition des données, d'enregistrement, et la calcul des distances. Pour des raisons de simplicité, j'ai choisi d'utiliser de représenter mes données entrantes

par une image (le noir représentant les obstacles) et de présenter les résultats également sous forme d'une image (l'image de départ, avec le chemin solution dessiné en bleu).

Chargement des données
```
SDL_Surface *s = SDL_LoadBMP("carte.bmp");
```

Enregistrement des résultats
```
SDL_SaveBMP(s, "resultat.bmp");
```

Calcul des distances
```
/* calcule la distance entre les points (x1,y1) et (x2,y2) */
float distance(int x1, int y1, int x2, int y2){
    /* distance euclidienne */
 return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

 /* carré de la distance euclidienne */
 /* return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2); */
}
```

Et une petite fonction pour tester l'existence d'un élément dans une liste donnée :

```
bool deja_present_dans_liste(pair<int,int> n, l_noeud& l){
    l_noeud::iterator i = l.find(n);
    if (i==l.end())
        return false;
    else
        return true;
}
```

III-3 - L'algorithme proprement dit

Une grosse partie de l'algorithme consiste à repérer les noeuds adjacents et à les ajouter ou non à la liste ouverte.

```
void ajouter_cases_adjacentes(pair <int,int>& n){
    noeud tmp;
```

On recherche les noeuds voisins du noeud dont les coordonnées sont passées en paramètre. tmp sera un noeud temporaire.

```
    /* on met tous les noeud adjacents dans la liste ouverte (+vérif) */
    for (int i=n.first-1; i<=n.first+1; i++){
        if ((i<0) || (i>=s->w))  /* en dehors de l'image, on oublie */
            continue;
        for (int j=n.second-1; j<=n.second+1; j++){
            if ((j<0) || (j>=s->h))   /* en dehors de l'image, on oublie */
                continue;
            if ((i==n.first) && (j==n.second))  /* case actuelle n, on oublie */
                continue;

            if (*((Uint8 *)s->pixels + j * s->pitch + i * s->format->BytesPerPixel) == NOIR)
                /* obstace, terrain non franchissable, on oublie */
                continue;

            pair<int,int> it(i,j);
```

Le code ci-dessus détermine si un pixel voisin peut faire partie de la solution, il élimine tous les cas ne correspondant pas : les pixels en dehors de l'image, ceux infranchissables et celui occupé par le pixel courant.

```
            if (!deja_present_dans_liste(it, liste_fermee)){
                /* le noeud n'est pas déjà présent dans la liste fermée */

                /* calcul du cout G du noeud en cours d'étude : cout du parent + distance jusqu'au
 parent */
                tmp.cout_g = liste_fermee[n].cout_g + distance(i,j,n.first,n.second);

                /* calcul du cout H du noeud à la destination */
                tmp.cout_h = distance(i,j,arrivee.x,arrivee.y);
                tmp.cout_f = tmp.cout_g + tmp.cout_h;
                tmp.parent = n;

                if (deja_present_dans_liste(it, liste_ouverte)){
                    /* le noeud est déjà présent dans la liste ouverte, il faut comparer les couts
 */
                    if (tmp.cout_f < liste_ouverte[it].cout_f){
                        /* si le nouveau chemin est meilleur, on met à jour */
                        liste_ouverte[it]=tmp;
                    }

                    /* le noeud courant a un moins bon chemin, on ne change rien */


                }else{
                    /* le noeud n'est pas présent dans la liste ouverte, on l'y ajoute */
                    liste_ouverte[pair<int,int>(i,j)]=tmp;
                }
            }
        }
    }
}
```

Ce code s'occupe de l'insertion d'un noeud dans la liste ouverte si besoin est. Il vérifié si un noeud est présent ou non dans la liste fermée et dans la liste ouverte, et il met à jour la liste ouverte si le nouveau noeud s'avère meilleur que celui déjà présent.

Il nous faut aussi une fonction permettant d'obtenir le meilleur noeud de la liste ouverte :

```
pair<int,int> meilleur_noeud(l_noeud& l){
    float m_coutf = l.begin()->second.cout_f;
    pair<int,int> m_noeud = l.begin()->first;

    for (l_noeud::iterator i = l.begin(); i!=l.end(); i++)
        if (i->second.cout_f< m_coutf){
            m_coutf = i->second.cout_f;
            m_noeud = i->first;
        }

    return m_noeud;
}
```

Ce code se contente de parcourir la liste pour repérer le noeud qui a le cout F le plus faible. Il retourne les coordonnées (x,y) de ce noeud.

Il peut être très intéressant d'utiliser une structure triée. La recherche de maximum s'en trouve donc considérablement accélérée.

Le jeu entre la liste ouverte et la liste fermée est fait par la fonction d'ajout en liste fermée :

```
void ajouter_liste_fermee(pair<int,int>& p){
    noeud& n = liste_ouverte[p];
```

```
        liste_fermee[p]=n;

        /* il faut le supprimer de la liste ouverte, ce n'est plus une solution explorable */
        if (liste_ouverte.erase(p)==0)
            cerr << "Erreur, le noeud n'apparait pas dans la liste ouverte, impossible à supprimer" <<
    endl;
        return;
    }
```

Pas grand chose à dire sur cette petite fonction. Elle passe un noeud de la liste ouverte vers la liste fermée.

Le dernier traitement dont nous avons besoin est celui qui consiste à retrouver le chemin une fois que la destination a été atteinte. On remonte les noeuds de parent en parent. Chaque noeud est ajouté en tête d'une liste, de manière à reconstituer le chemin à partir de la fin

```
    void retrouver_chemin(){
        /* l'arrivée est le dernier élément de la liste fermée */
        noeud& tmp = liste_fermee[std::pair<int, int>(arrivee.x,arrivee.y)];

        struct point n;
        pair<int,int> prec;
        n.x = arrivee.x;
        n.y = arrivee.y;
        prec.first  = tmp.parent.first;
        prec.second = tmp.parent.second;
        chemin.push_front(n);

        while (prec != pair<int,int>(depart.parent.first,depart.parent.first)){
            n.x = prec.first;
            n.y = prec.second;
            chemin.push_front(n);

            tmp = liste_fermee[tmp.parent];
            prec.first  = tmp.parent.first;
            prec.second = tmp.parent.second;
        }
    }
```

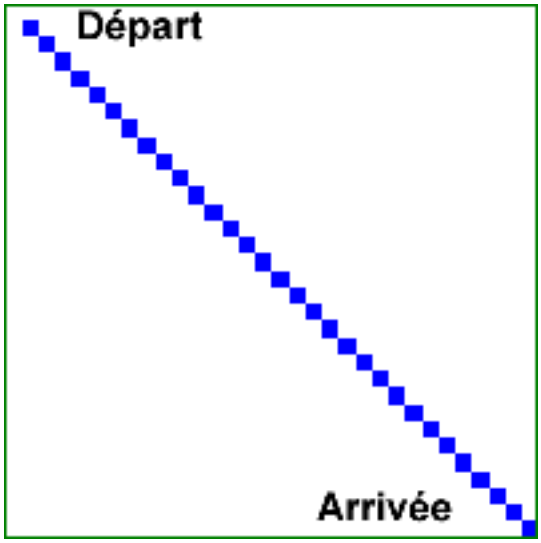
Bien, passons maintenant à l'ordonancement de ces traitements : la boucle principale de recherche

```
        arrivee.x = s->w-1;
        arrivee.y = s->h-1;

        depart.parent.first  = 0;
        depart.parent.second = 0;

        pair <int,int> courant;

        /* déroulement de l'algo A* */

        /* initialisation du noeud courant */
        courant.first  = 0;
        courant.second = 0;

        /* ajout de courant dans la liste ouverte */
        liste_ouverte[courant]=depart;
        ajouter_liste_fermee(courant);
        ajouter_cases_adjacentes(courant);

        /* tant que la destination n'a pas été atteinte et qu'il reste des noeuds à explorer dans la
     liste ouverte */
        while( !((courant.first == arrivee.x) && (courant.second == arrivee.y))
                &&
               (!liste_ouverte.empty())
```

```
            ){

        /* on cherche le meilleur noeud de la liste ouverte, on sait qu'elle n'est pas vide donc il
existe */
        courant = meilleur_noeud(liste_ouverte);

        /* on le passe dans la liste fermee, il ne peut pas déjà y être */
        ajouter_liste_fermee(courant);

        /* on recommence la recherche des noeuds adjacents */
        ajouter_cases_adjacentes(courant);
    }

    /* si la destination est atteinte, on remonte le chemin */
    if ((courant.first == arrivee.x) && (courant.second == arrivee.y)){
        retrouver_chemin();

        ecrire_bmp();
    }else{
        /* pas de solution */
    }
```

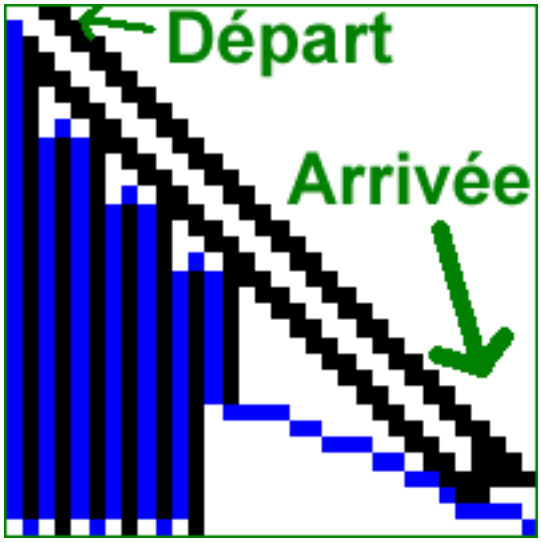
IV - Exemple de recherches de chemin

Voici quelques exemples plus ou moins tarabiscotés de recherche de chemin :



Aucun obstacle, le chemin est direct

Un obstacle, la solution le contourne



La solution n'est pas le chemin direct

V - Améliorations possibles

Essayez de lancer cet algorithme sur une très grande carte et vous verrez que ça commence à prendre du temps. Et si vous avez besoin d'un réponse rapide de votre programme vous ne pourrez pas vous permettre de prendre plusieurs secondes pour calculer le chemin. Une solution est donc de découper le chemin entre plusieurs points connus.

- Utilisez une structure de liste triée pour repérer rapidement le meilleur noeud
- Faites des calculs en valeurs entières

V-1 - Applications dans les jeux

- Quand vous demandez à un bonhomme de se rendre à l'autre bout de la carte, il n'est pas nécessaire de calculer dès le début tout le chemin. Définissez des points clés et faites déjà la recherche de chemin vers le point clé qui correspond à sa position et à sa destination. Vous aurez ensuite le temps de préparer la suite du chemin.
- Le jeu connais la carte, vous pouvez donc précalculer des parcours type ou des morceaux de parcours entre des points clés
- Si vous calculez tout le chemin dès le début, votre bonhomme évitera dès le début toutes les impasses et tous les culs de sac qu'il aurait pu rencontrer. Pas très réaliste. Découpez la recherche en fonction de l'avancement du personnage.
- Si le chemin proposé par l'algorithme passe dans une zone dangereuse, il n'est pas très réaliste d'y aller. Prévoyez un indicateur de danger selon l'endroit et tenez-en compte dans la recherche de chemin en alourdissant le cout pour se rendre à un noeud dangereux. Il cherchera d'abord les chemins les moins dangereux.

Toute mon implémentation a été faite à partir d'une image. Mais dans le cas où vous devez manipuler une grille de jeu ou toute autre structure différente, il y a des modifications à apporter au code. Bien évidemment l'algorithme reste le même.

Les modifications auront lieu dans la recherche des noeuds voisins. Sur une image, les noeuds voisins sont les pixels voisins, mais sur une grille de jeu différente, les noeuds voisins peuvent être représentés différemment. C'est pourquoi il sera peut-être nécessaire de réaliser une conversion entre votre grille de jeu et une structure plus facilement exploitable par A*. Mais ceci n'est qu'une question d'implémentation et ne dépend pas directement de l'algorithme.

VI - Conclusion

Voilà, vous savez comment fonctionne l'algorithme A*. Il ne donne pas toujours la meilleure solution mais il en donne une bonne. On pourrait comparer ses performances avec celles de l'algorithme de Dijkstra. Dijkstra donne la meilleure solution, mais A* est plus rapide.

Et comme dans beaucoup d'algorithmes, le programmeur a une grande liberté dans son implémentation : vous pouvez modifier les méthodes de calcul de distances, pondérer certains noeuds, en privilégier d'autres ...

VII - Téléchargement et liens

Téléchargez l'archive contenant le projet Code::Blocks et les sources

Téléchargez l'archive contenant le projet Code::Blocks, les sources, un makefile unix/linux et l'exécutable windows pour la version graphique, réalisée par fearyourself (avec SDL et SDL_gfx)

Pour plus d'informations sur les parcours de graphe et la recherche de chemin :

- **Introduction aux graphes**
- Livre : **Algorithmes de graphes**, pour voir d'autres méthodes de parcours de graphes
- Livre : **Ai Game Programming Wisdom**, pour voir des domaines d'applications et une approche axée autour des jeux vidéos
- **Intérêts et problématiques liées au pathfinding**
- **Réflexions sur A* et quelques approfondissements**

Je remercie les équipes des rubriques 2D/3D/Jeux et Algo pour leurs remarques et relectures.