TME 9 : Algorithme de routage A*

Objectif

Le but de ce TME est de programmer, en langage C, l'algorithme de routage A*. Cet algorithme recherche le chemin le plus court entre deux points (un point source et un point destination) dans un espace à deux dimensions contenant des obstacles qu'il faut contourner. L'espace est discrétisé afin de travailler sur des coordonnées entières. Un chemin est constitué de segments de droites. On contraint ces segments à être parallèles aux axes des abscisses et des ordonnées. On utilisera donc une distance de Manhattan.

A) Principe Général

L'espace est représenté par un tableau à deux dimensions DS[X,Y]. Ce tableau est indexé par deux index (X,Y) représentant les coordonnées, et chaque case correspond donc à une position dans le plan. Une case a quatre voisins (Nord, Sud, Est, Ouest), et deux positions i et j sont voisines si [X(i) = X(j) +/- 1 et Y(i) = Y(j)] ou si [Y(i) = Y(j) +/- 1 et X(i) = X(j)].

Certaines cases (X,Y) jouent un rôle particulier :

- position de la source S (Xs,Ys)
- position de la destination T (Xt,Yt)
- positions Z contenant un obstacle (il peut y en avoir plusieurs).

Le tableau DS[X,Y] est un tableau d?entiers. La valeur contenue dans une case (X,Y) de ce tableau contient en principe la distance DS(P) entre la source S et le point P de coordonnées (X,Y). La fonction principale de l?algorithme consiste à calculer ces distances DS(P). Au début de l?algorithme, ces distances sont donc inconnues, et toutes les cases du tableaux ne contenant pas d?obstacle sont initialisées à une valeur « infinie » (en pratique MAXINT). La case S est initialisée à la valeur 0. On représente le fait qu?une case Z contient un obstacle, en initialisant DS(Z) avec la valeur -1.

Le principe général de l?algorithme consiste à calculer progressivement les distances DS(P), en commençant par les points au voisinage de S, puis en élargissant peu à peu la zone autour de S dans laquelle les distances ont été calculées. Cette zone *connue* grossit peu à peu jusqu?à atteindre le point T. A un instant donné, il existe donc une zone connexe (autour du point S) regroupant les points pour lesquels la distance DS est connue et une autre zone (contenant le point T) regroupant les points pour lesquels la distance est DS est inconnue. Ces deux zones sont séparées par une *frontière*, qui contient les points dont la distance DS est connue, mais qui peuvent avoir des voisins dont la distance DS n?a pas encore été calculée. Cette frontière constitue le périmètre de la zone connue, et évolue au cours de l?algorithme. Pour représenter cette frontière, on utilise une structure de donnée FRONTIER, qui est une liste chaînée ordonnée et dynamique, contenant tous les points P(X,Y) appartenant à la frontière à un

instant donné.

B) Algorithme de LEE et algorithme A*

Il y a différentes façons de faire évoluer la frontière entre la zone connue et la zone inconnue, correspondant à différentes stratégies d?exploration de l?espace autour de S, et nous étudierons successivement deux algorithmes dans le cadre de ce TME.

B1) Algorithme de Lee

Publié en 1961 (C. Y. Lee, "An algorithm for path connection and its application", IRE Trans. Electronic Computer, EC-10, 1961) cet algorithme permet de trouver de façon certaine le chemin le plus court ente deux points, mais il est coûteux en temps de calcul. Il consiste à faire grossir la zone connue de façon isotrope, dans toutes les directions autour du point S. Plus précisément, cet algorithme garantit que tous les points situés à une distance D du point source S auront été calculés avant de commencer à calculer les points situés à une distance D+1.

Cet algorithme travaille en trois étapes phases : une phase d?initialisation, une phase d'expansion, consistant à faire grossir la zone connue en stockant les résultats dans les tableaux DS[x,y], et une phase de retour, où on calcule effectivement le chemin le plus court pour revenir de T à S, en utilisant le tableau DS[x,y].

Etape initialisation pour LEE :

On initialise toutes les cases du tableau DS[x,y] à la valeur MAXINT, sauf les cases contenant un obstacle, qui prennent la valeur -1, et la case S, qui prend la valeur 0. On initialise la liste ordonnée FRONTIER qui ne contient que le point source S.

Etape expansion pour LEE :

```
POUR tous les points P appartenant à la frontière
        Retrait du point P de la tête de la liste FRONTIER
        POUR tous les voisins Q du point P
                SI DS(Q) == MAXINT
                        DS(Q) = DS(P) + 1
                        Ajout du point Q en queue de la liste FRONTIER
                FINSI
                SI Q==T ALORS FINPOUR
        FINPOUR
FINPOUR
SI DS(T) != MAXINT
        RETOURNE succes
ALORS
        RETOURNE echec
FINSI
```

Etape retour pour LEE :

On reconstruit le chemin, de la cible T vers la source S, en recherchant pour chaque point P situé à une distance D un voisin situé à une distance (D-1). On place à chaque position faisant parti du chemin, le code -2. Il peut y avoir plusieurs solutions. Toutes les solutions sont équivalentes pour ce qui concerne la distance. Le choix d'une solution particulière peut être guidée par des contraintes du genre « éviter les changements de directions ».

B2) Algorithme A*

L'algorithme A* (prononcer A-star) est une variante de l?algorithme de Lee. Il est beaucoup plus rapide, car on ne déplace plus la frontière de façon isotrope dans toutes les directions, mais, on essaie de calculer en priorité les points qui permettent de se rapprocher de la destination T (la frontière est « attirée » par la destination T). Il y a un prix à payer pour cette optimisation : la solution trouvée n?est pas toujours optimale. L?algorithme A* travaille également en trois phases. Les deux phases d?initialisation et de retour sont inchangées. Seule la phase d?expansion est légèrement modifiée.

Pour cela on définit, pour chaque point P de la frontière la quantité COUT(P), qui est une estimation optimiste de la distance entre S et T, si on passe par le point P :

```
COUT(P) = DS(P) + ESTIME(P,T)
```

Le premier terme est un calcul exact de la distance entre S et P, tenant compte des obstacles rencontrés. Le deuxième terme est une estimation (incertaine) de la distance entre P et T, qui est calculé comme la distance de Manhattan entre P et T, et ne prend pas en compte les obstacles qui n?ont pas encore été rencontrés.

La principale différence entre les deux algorithmes LEE et A* consiste à ne faire bouger la frontière que pour le point P dont le coût est minimal, et qui est donc en principe le plus proche de T. Pour cela la liste des points appartenant à la frontière est ordonnée par coût croissant:

Etape expansion pour A* :

```
POUR le point P de coût minimal appartenant à la frontière
        Retrait du point P de la liste FRONTIER
        POUR tous les voisins Q du point P
                SI DS(Q) == MAXINT
                        DS(Q) = DS(P) + 1
                        Ajout à sa place du point Q dans la liste FRONTIER
                FINSI
                SI Q==T ALORS FINPOUR
        FINPOUR
FINPOUR
SI DS(T) != MAXINT
        RETOURNE succes
ALORS
        RETOURNE echec
FINSI
```

C) Structures de données et fonctions d?accès

C1) Gestion du tableau

Le tableau est géré par une bibliothèque de fonctions graphiques (cf. xmaze.h), qui va se charger de l'initialisation et de l'affichage du tableau dans une fenêtre X11. Seules quelques fonctions sont externalisées pour pouvoir être utilisées par les fonctions de routage

```
#define MAZ_HAUT  40
#define MAZ_LARG  40
```

Ces directives définissent la taille du tableau.

```
enum {VIDE=MAXINT, OBSTACLE=-1, CHEMIN=-2};

typedef struct maze_t
```

```
{
    int * MAZE;     /* pointeur vers le tableau */
    int HAUT, LARG;     /* hauteur et largeur du tableau */
} maze_t;
static inline int get_maze(maze_t * ds, int x, int y)
```

Cette fonction rend le contenu la case DS[X,Y].

```
static inline set_maze(maze_t * ds, int x, int y, int val)
```

Cette fonction initialise la case DS[X,Y] avec la valeur VAL.

```
extern void display_maze (maze_t * ds)
```

Cette fonction affiche sur la fenêtre X11, l'état du tableau DS. Les cases VIDE (valeur=MAXINT) sont affichées en noir. Les cases OBSTACLE (valeur=-1) sont affichées en rouge. Les cases CHEMIN (valeur=-2) sont affichées en jaune. Les cases positives (mais non VIDE) sont affichées en bleu.

C2) Algorithme de Lee

Les fonctions correspondantes sont déclarées dans le fichier *maze.h*

```
typedef struct boundlee_t
{
        struct boundlee_t * NEXT;
        int X, Y;
} boundlee_t;
```

Cette structure représente un élément de la liste ordonnée des points appartenant à la frontière dans le cas de l?algorithme de Lee.

```
typedef struct frontierlee_t
{
        boundlee_t * FIRST;  /* premier élément de la liste ordonnée*/
        boundlee_t * LAST;   /* dernier élément de la liste ordonnée*/
} frontierlee_t;
```

Cette structure contient les deux pointeurs permettant d?accéder aux éléments de la liste ordonnée représentant la frontière dans le cas de Lee.

Les fonctions d?accès sont les suivantes :

```
extern frontierlee_t *cons_frontierlee();
```

Cette fonction construit une structure frontierlee_t.

```
extern void display_frontierlee(frontierlee_t *list);
```

Cette fonction de debug affiche à l?écran tous les éléments de la frontière. Vous utiliserez la mise au point des fonctions de gestion de cette liste.

```
extern void free_frontierlee(frontierlee_t *list);
```

Cette fonction permet de re-initialiser à « vide » la liste ordonnée représentant la frontière.

```
extern void add_boundlee(frontierlee_t *list, int x, int y);
```

Cette fonction ajoute un point en fin de liste dans la liste ordonnée représentant la frontière.

```
extern boundlee_t * get_boundlee(frontierlee_t *list);
```

Cette fonction renvoie un pointeur sur le point qui se trouve entête de la liste ordonnée représentant la frontière, et retire ce point de la liste.

C3) Algorithme A*

```
typedef struct boundastar_t
{
        struct boundastar_t     *NEXT;
        int                     X, Y;   /* coordonnées du point */
        int                     COUT ;  /* cout estimé          */
} boundastar_t;
```

Cette structure représente un élément de la liste ordonnée des points appartenant à la frontière dans le cas de l?algorithme A*.

```
typedef struct frontierastar_t
{
        boundastar_t    *FIRST;  /* premier élément de la liste ordonnée */
        int XT, YT;              /* Coordonnées de la cible */
} frontierastar_t;
```

Cette structure contient le pointeur permettant d?accéder aux éléments de la liste ordonnée représentant la frontière dans le cas de A* et les coordonnées de la cible, ce qui est nécessaire pour calculer le coût de la position.

Les fonctions d?accès sont les suivantes :

```
extern frontierastar_t *cons_frontierastar(int xt, int yt);
```

Cette fonction construit une structure frontierastar_t. Elle prend en paramètres les coordonnées de la cible.

```
extern void display_frontierastar(frontierastar_t *list);
```

Cette fonction de debug affiche à l?écran tous les éléments de la frontière.

```
extern void free_frontierastar(frontierastar_t *list);
```

Cette fonction permet de re-initialiser à « vide » la liste ordonnée représentant la frontière.

```
extern void add_boundastar(frontierastar_t *list, int x,int y,int cout);
```

Cette fonction ajoute un point dans la liste dans la liste ordonnée représentant la frontière, à la place qui lui convient, de façon à respecter le classement par coût croissant.

```
extern boundastar_t * get_boundastar (frontierlastar_t *list);
```

Cette fonction renvoie un pointeur sur le point qui se trouve entête de la liste ordonnée représentant la frontière, et retire ce point de la liste.

Les fonctions suivantes réalisent les phases d?expansion et de retour :

```
extern int expandlee(maze_t *ds,int xs,int ys,int xt,int yt);
extern int expandastar(maze_t *ds,int xs,int ys,int xt,int yt);
extern int findpath(maze_t *ds, int xt,int yt);
```

Le pointeur ds pointe sur le tableau des distances. (xs,ys) et (xt,yt) sont les coordonnées du point source et du point destination.

D) Travail à effectuer

Créez un répertoire de travail **tme9**, et copiez dans ce répertoire tous les fichiers et répertoires qui se trouvent dans

```
/users/enseig/encadr/cao/tme9
```

Le Makefile fourni permet de générer un exécutable nommé xlee fonctionnel. Avec cet exécutable, vous pouvez créer un labyrinthe, choisir un point source, un point cible, et demander le calcul du chemin en utilisant l'algorithme de LEE ou de Astar. Vous allez devoir remplacer progressivement les fichiers objets fournis par vos propres fichiers objets.

Q1) réécrire la fonction expandlee()

Q2) réécrire les fonctions add_boundlee()

Q3) réécrire la fonction findpath()

Q4) réécrire la fonction expandastar()

Q5) réécrire les foncions add_boundastar()

Compte-Rendu

Il ne vous est pas demandé de compte-rendu écrit pour ce TME, mais vous devrez faire une démonstration de votre code au début du prochain TME.