

TME 9 : Algorithme de routage A*

1. A) Objectif
 1. A1) Représentation de l'espace
 2. A2) Principe général de l'algorithme
2. B) Algorithme LEE Maze
3. C) Algorithme A*
4. D) Structures de données et fonctions d'accès
 1. D1) Gestion du tableau (cf. xmaze.h)
 2. D2) Algorithme de Lee (cf. maze.h)
 3. D3) Algorithme A*
5. E) Travail à effectuer
6. E1) réécrire la fonction expandlee() =
 1. E2) réécrire les fonctions add_boundlee()
 2. E3) réécrire la fonction findpath()
 3. E4) réécrire la fonction expandastar()
 4. E5) réécrire les fonctions add_boundastar()

A) Objectif

On cherche 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. En conséquence pour mesurer la distance, on utilisera donc une distance de Manhattan.

A1) Représentation de l'espace

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 $[X(i) = X(j) + 1 \text{ et } Y(i) = Y(j)]$ ou si $[Y(i) = Y(j) + 1 \text{ et } X(i) = X(j)]$.

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

- position de la source $S (X_s, Y_s)$
- position de la destination $T (X_t, Y_t)$
- 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 $P(X,Y)$ de ce tableau contient en principe la distance $DS(P)$ entre la source S et le point P , et 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 tableau ne contenant pas d'obstacle sont initialisées à une valeur « infinie » (en pratique $VIDE=MAXINT$). 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 $OBSTACLE=-1$.

A2) Principe général de l'algorithme

La méthode générale 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 est 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 les points P(X,Y) appartenant à la frontière à un instant donné.

Il y a différentes façons de faire évoluer cette frontière, correspondant à différentes stratégies d'exploration de l'espace autour de S, et nous étudierons deux algorithmes dans le cadre de ce TME.

B) Algorithme LEE Maze

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 entre 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].

Étape 0 (initialisation) :

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

Étape 1 (expansion) :

```

TANTQUE DS(T) == MAXINT
    POUR tous les points P appartenant à la frontière
        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
        FINPOUR
    Retrait du point P de la tête de la liste FRONTIER
FINPOUR
FINTANTQUE

```

Étape 2 (retour)

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 CHEMIN=-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 ».

C) 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 : La conséquence de cette optimisation est que 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 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.

Étape 1 (expansion pour A*) :

```
TANTQUE DS(T) == MAXINT
    POUR le point P appartenant à la frontière de coût minimal
        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
        FINPOUR
        Retrait du point P de la tête de la liste FRONTIER
    FINPOUR
FINTANTQUE
```

D) Structures de données et fonctions d'accès

D1) Gestion du tableau (cf. xmaze.h)

Le tableau est géré par une bibliothèque de fonctions graphiques, qui va se charger de l'initialisation (étape0) 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 définitions 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.

D2) Algorithme de Lee (cf. 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 boundlee_t * 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 en tête de la liste ordonnée représentant la frontière, et retire ce point de la liste.

D3) 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 xs, int ys);
```

Cette fonction construit une structure frontierastar_t.

```
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 boundastar_t * 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 (frontierastar_t *list);
```

Cette fonction renvoie un pointeur sur le point qui se trouve en tête 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.

E) Travail à effectuer

Le code source et les fichiers objets de ce TME sont dans le répertoire `cao/tme9`. Vous y trouverez un Makefile permettant 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 les fichiers objets fournis par vos propres fichiers objets.

E1) réécrire la fonction `expandlee()` =

E2) réécrire les fonctions `add_boundlee()`

E3) réécrire la fonction `findpath()`

E4) réécrire la fonction `expandastar()`

E5) réécrire les fonctions `add_boundastar()`