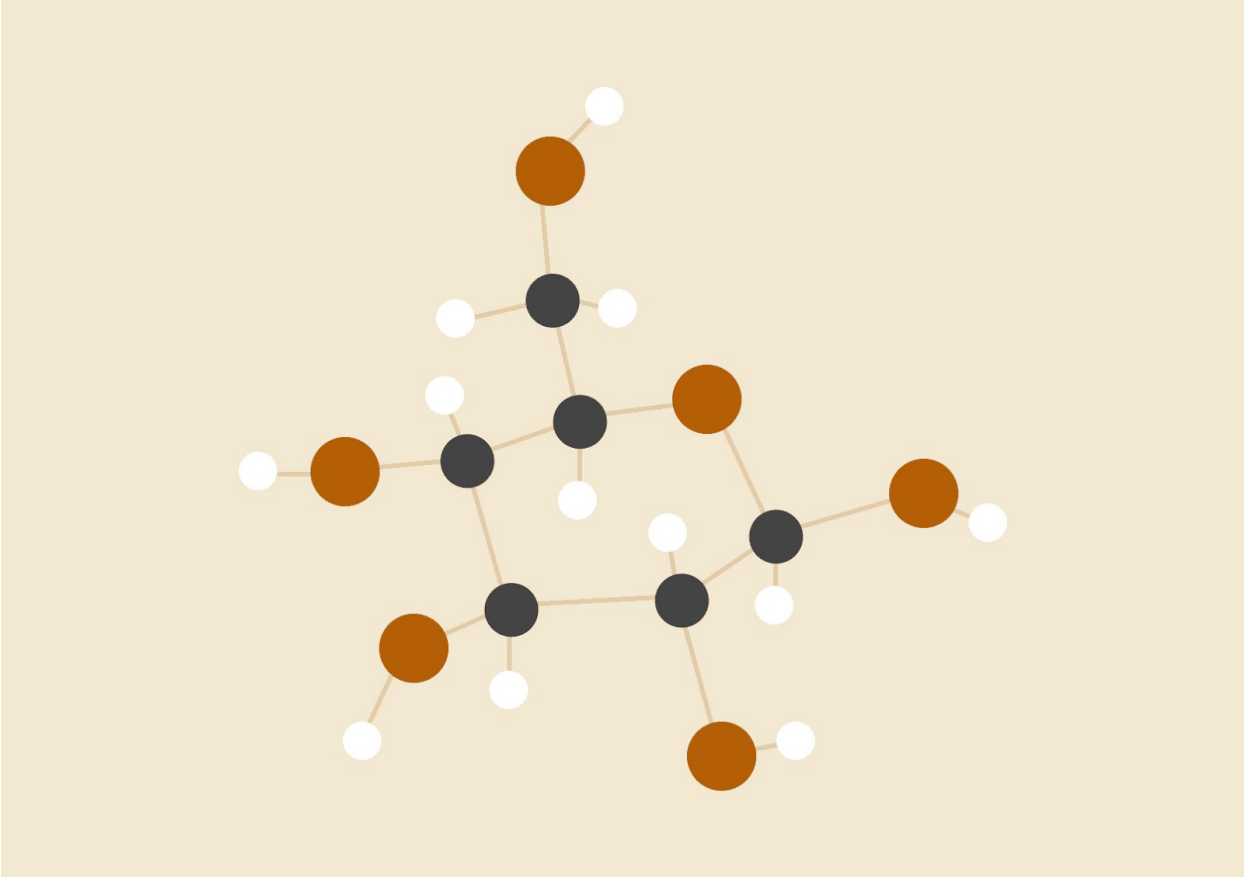
RAPPORT DE PROJET

*Implémentation de l'algorithm A**



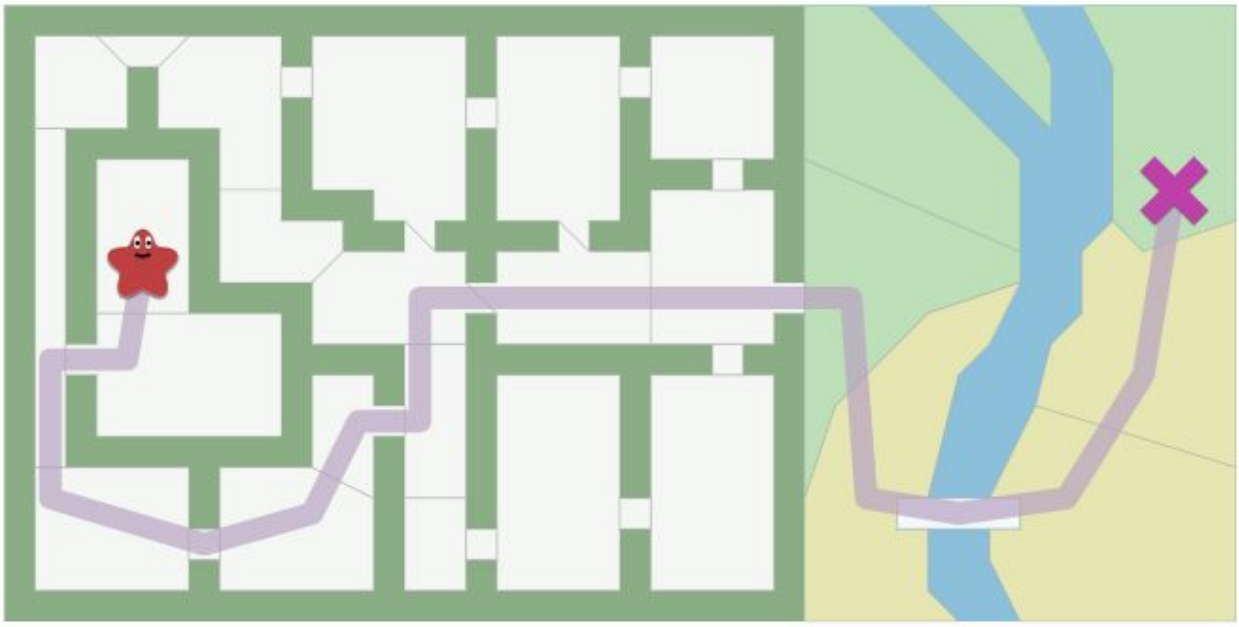
Amine HAMOU

10/09/2020

Structures de Données – GLSID 1

INTRODUCTION

Souvent dans les jeux vidéo, nous voulons trouver le chemin de point A vers le point B, nous ne voulons pas juste trouver la plus court chemin mais aussi tenir compte de temps nécessaire pour y traverser passer.



Pour trouver ce chemin nous pouvons utiliser des algorithmes de parcours des graphes lorsque l'espace étudié est représenté sous la forme d'un graphe. A* (A-étoile ou *A-star* en anglais) est un choix populaire parce que si on choisit une heuristique admissible *(une heuristique est dite admissible si elle ne surestime jamais la distance ou plus généralement le coût)* il est garanti de trouver un chemin optimal sans traiter deux fois le même nœud.

L'éclat de A* est qu'il utilise une évaluation heuristique (h) sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. à partir de la source, l'algorithme vise à trouver un chemin vers la destination tout en minimisant le coût (poids des arcs).

À chaque itération de sa boucle principale, A * doit déterminer lequel de ses chemins s'étendre. Il le fait sur la base du coût du chemin **g(n)** et d'une estimation du coût nécessaire pour étendre le chemin jusqu'à la destination **h(n)**. Pour ce faire, A * sélectionne le chemin optimal de manière à minimiser une fonction **f (n)** = **g (n)** + **h (n)** .

A* se termine lorsque le chemin qu'il choisit d'étendre est un chemin de la source à la

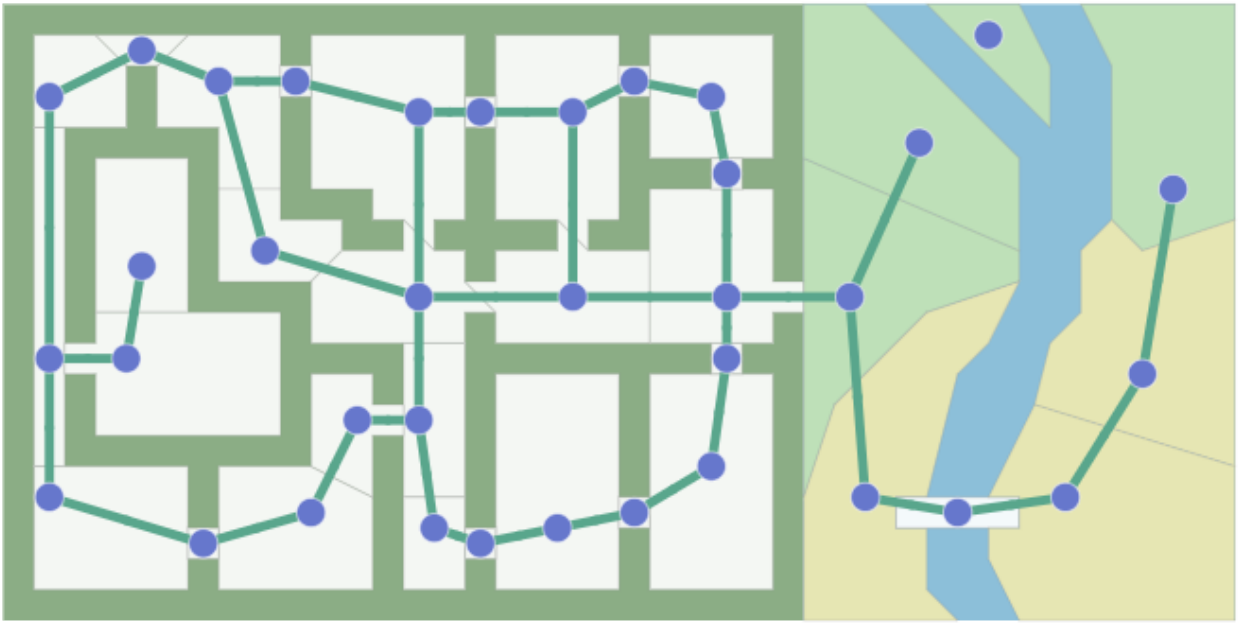
destination ou s'il n'y a pas de chemin pouvant être étendu.

REPRÉSENTATION DE L'ESPACE

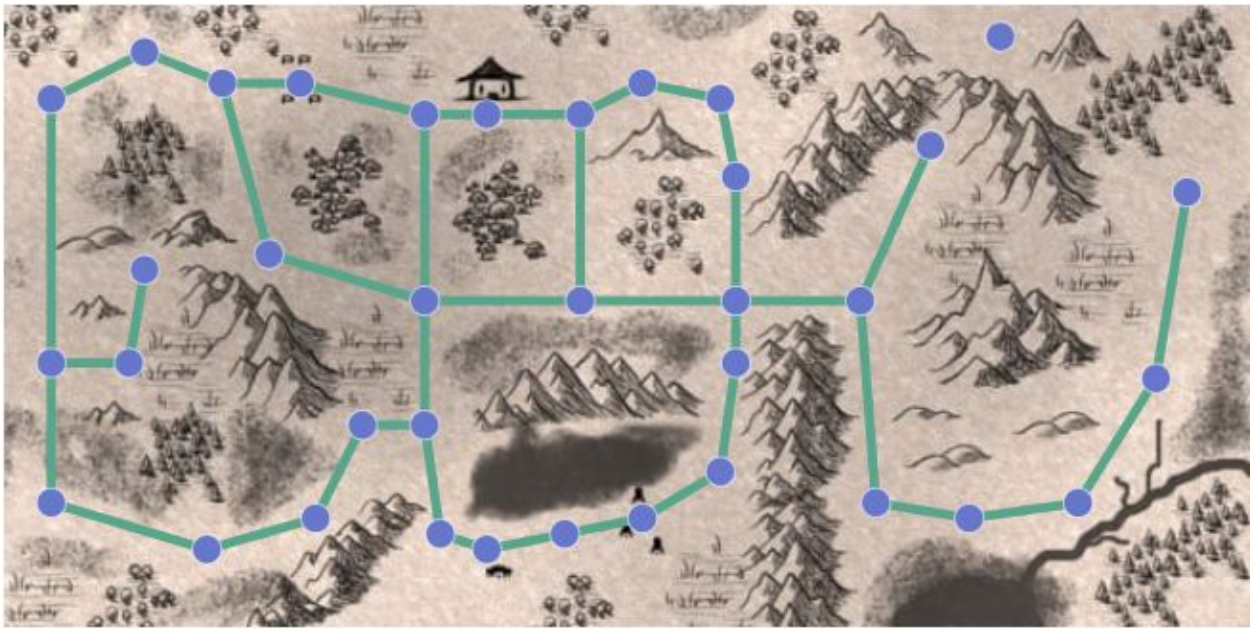
La première chose à faire quand on étudie un algorithme est de comprendre le *INPUT* et le *OUTPUT*.

1. ***INPUT :***

   La majorité des algorithmes de parcours des graphes accepte un graphe comme INPUT. Un graphe est un couple d'ensembles (X, C) où X est un ensemble d'éléments appelés sommets et C un ensemble d'éléments appelés arêtes, ces arêtes lient les sommets deux à deux.



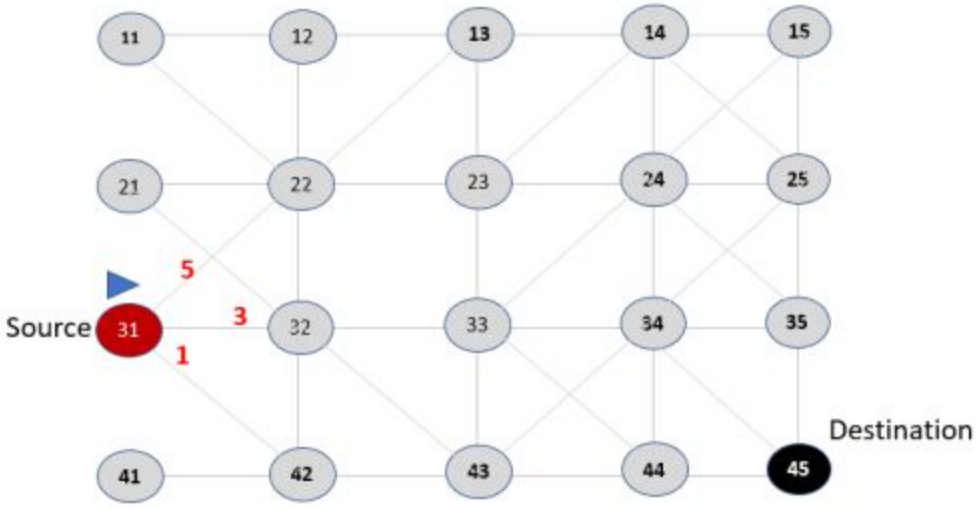
   A* ne voit que le graphe et rien d'autre, pour A* il n'y a aucune différence entre le graphe précédent et celui ci-dessous.



Pour des meilleurs précisions on choisit un graphe qui ressemble à une matrice comme ça on peut travailler avec les notions de coordonnées dans nos calculs.

Le graphe se compose des sommets accompagnés des arrêts sortants de chaque sommet.

Un sommet est défini par ses coordonnées x et y. par exemple la source est définie par ses cordonnées (x=3,y=1) et la destination est défini par ses coordonnées (x=4, y=5). Un arrêt est une liaison entre deux nœuds et qui a un coût.



2. *OUTPUT :*

Le chemin trouvé par A* est un ensemble des nodes qui lient entre le node de départ et le node de fin avec des arrêts qui ont des coûts pondérés.

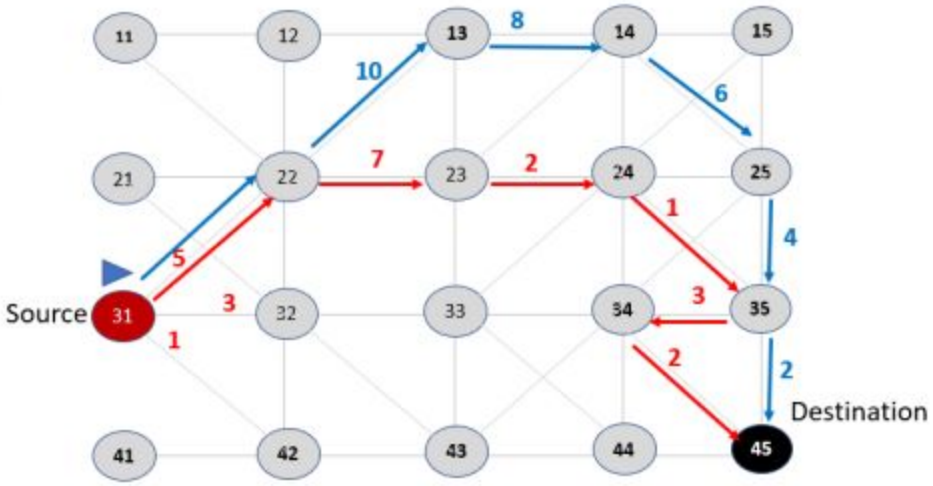


Figure 2

Structures de Données Utilisés:

Les structure de données utilisées dans la création de le graphe sont les suivants:

```
typedef struct t_NODE{
    unsigned int x,y;
    float f,g,h;
    unsigned int can_go;
    unsigned int opened;
    struct t_NODE *parent;
}t_NODE;

typedef struct t_GRAPH{
    unsigned int rows,cols;
    t_NODE **matrice;
}t_GRAPH;
```

Chaque sommet est représenté par *struct t_NODE* , ce type créé contient toutes les informations relative à un sommet tout seul tels que ses coordonnées [x,y], il contient aussi des information du sommet en relation avec l'algorithme A* tel que les valeur sur laquelle fonctionne l'algorithme (f,g,h) et il contient un type bool qui spécifie si le node permet le passage ou non (can_go) et si le sommet a été déjà traité par l'algorithme(opened).

Le type *t_NODE contient aussi un pointeur vers son parent qui nous sera utile lors de reconstruction du chemin .*

De plus la totalité de graph est représenté par une autre structure *struct t_GRAPH* qui contient le nombre des colonnes et ligne de la matrice qui représente l'espace ainsi que un tableau bidimensionnelle qui regroupe les sommets *t_NODE* , d'ailleur ce tableau est la seule copie des sommet, tous les autres structures de données n'ont que des références sur ce tableau.

Fonctions Utilisés:

```
void define_node(t_NODE * n, int x,
                 int y, int can_go)
{
    n->x=x;
    n->y=y;

    n->f = LARGE_NUMBER;
    n->g = LARGE_NUMBER;
    n->h = 0.0;
    n->can_go = can_go;
    n->opened = 2;
    n->parent = NULL;
}
```

La fonction *define node()* permet de créer un sommet qui a les coordonnées [x,y] et de spécifier si ce sommet admet le passage ou non.
Par défaut le node est marqué comme jamais visité, et ses valeur f,g sont tendus vers l'infini et sa heuristique vers 0.

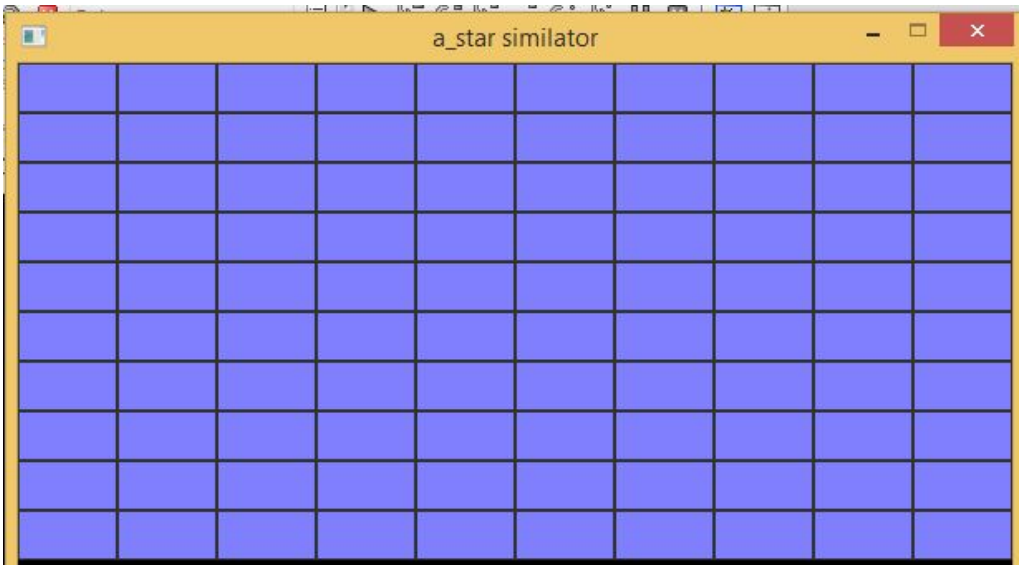
```
t_GRAPH * create_graph(int nb_col,int nb_row){
    t_NODE **nodes;
    t_NODE *nodes_row;
    t_NODE node;
    t_GRAPH *g=(t_GRAPH*)malloc(sizeof(t_GRAPH));
    nodes=(t_NODE**)malloc(sizeof(t_NODE*)*nb_row);
    g->rows=nb_row;
    g->cols=nb_col;

    int x,y;
    for(x=0;x<nb_row;x++){
        nodes_row=(t_NODE*)malloc(sizeof(t_NODE)*nb_col);
        nodes[x]=nodes_row;
        for(y=0;y<nb_col;y++){
            define_node(&node,x,y,1);
            nodes_row[y]=node;
        }
    }

    g->matrice=nodes;
    return g;
}
```

La fonction *create_graph()* reçoit le nombre des colonnes et lignes souhaité et procède à allouer l'espace nécessaire pour la matrice et pour chaque case elle crée un sommet et l'attache à la case convenable

voici un graphe de 10 lignes et 20 colonne généré à l'aide des structures et fonctions préalablement mentionnées.



IMPLÉMENTATION DE A*

Structures de Données Utilisés:

```
typedef struct PRIORITY_QUEUE{
    unsigned int size,maxsize;
    t_NODE **arr;    //tableau de r
}PRIORITY_QUEUE;
```

On a implémenté cette file de priorité parce que les éléments filés ayants une priorité à base de laquelle on doit enlever l'élément de le file, cette fonctionnalité nous servira ultérieurement pour qu'on puisse décider qu'elle sommet à entamer en fonction de f=g+h minimale dans la file

Fonctions Utilisés:

```
void push_to_queue(PRIORITY_QUEUE *q, t_NODE *n){

t_NODE * pop_from_queue(PRIORITY_QUEUE *q){

PRIORITY_QUEUE * create_queue(t_GRAPH * g){
```

Les fonction de base d'une file de priorité sont les suivantes : une pour créer la file et deux autres, une pour ajouter des éléments à la file et l'autre pour enlever de la tête de la file. On note qu'on a implémenter cette file de priorité à l'aide d'un heap(tableau) alors il faut à chaque ajout ou suppression vérifier que le caractéristique de la file est encore présente, sinon il faut la rétablir.

```
t_NODE * a_star(t_GRAPH * g, int start_x,int start_y,int end_x,int end_y){

    int row,col;
    t_NODE *n,*destination_node;
    PRIORITY_QUEUE *queue;


    //appliquez l'heuristic qui depend de la destination (end_x,end_y)
    destination_node=&(g->matrice[end_x][end_y]);
    for(row=0;row<g->rows;row++){
        for(col=0;col<g->cols;col++){
            n=&(g->matrice[row][col]);
            n->h=heuristic(n,destination_node);
        }
    }

    queue=create_queue(g);

    n=&(g->matrice[start_x][start_y]);
    push_to_queue(queue,n);


    while(n != destination_node && queue->size>0){
        n=pop_from_queue(queue);
        n->opened=0;

        update_all_adjacent_nodes(g,n,queue);
    }
    free(queue->arr);
    free(queue);
    return destination_node;
}
```

La fonction *a_star()* reçoit un graphe et les coordonnées du point de départ et d'arrivé, par la suite on parcourt la matrice entière pour attribuer la valeur de l'heuristique à chaque node, puis on crée un file de priorité et enfile le sommet de départ. Tant que le sommet défilé est différent de celui du destination et la taille de la file n'est pas vide on actualise les valeur f,g des voisins du sommet défilé. le sommet défilé est marqué comme visité et le voisin qui a un coût estimé vers l'arrivé minimale et choisi pour être enfilé.

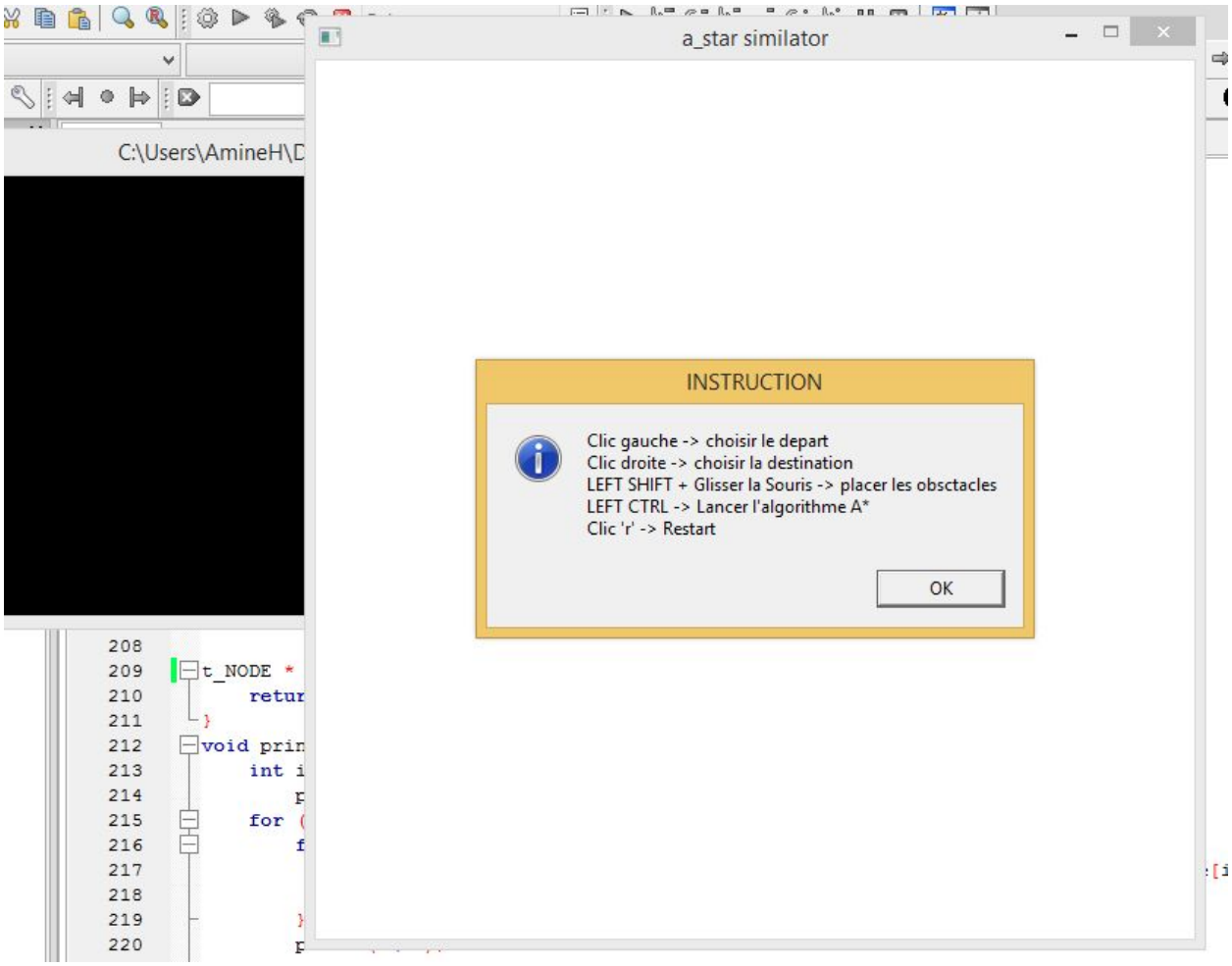
Vers la fin en renvoie le sommet destination.

```
t_NODE * reconstruct_path(t_NODE * start, t_NODE * end){
    return recursiveReverseLL(end);
}
```

Cette fonction fait appel à une autre fonction qui renverse l'ordre *Fils->parent* vers *Parent->Fils* pour qu'on puisse démarrer du ***t_NODE*** appelé départ et suivre le pointeur nommé parent dans la structure ***t_NODE*** jusqu'à la fin où on trouvera le sommet d'arrivée

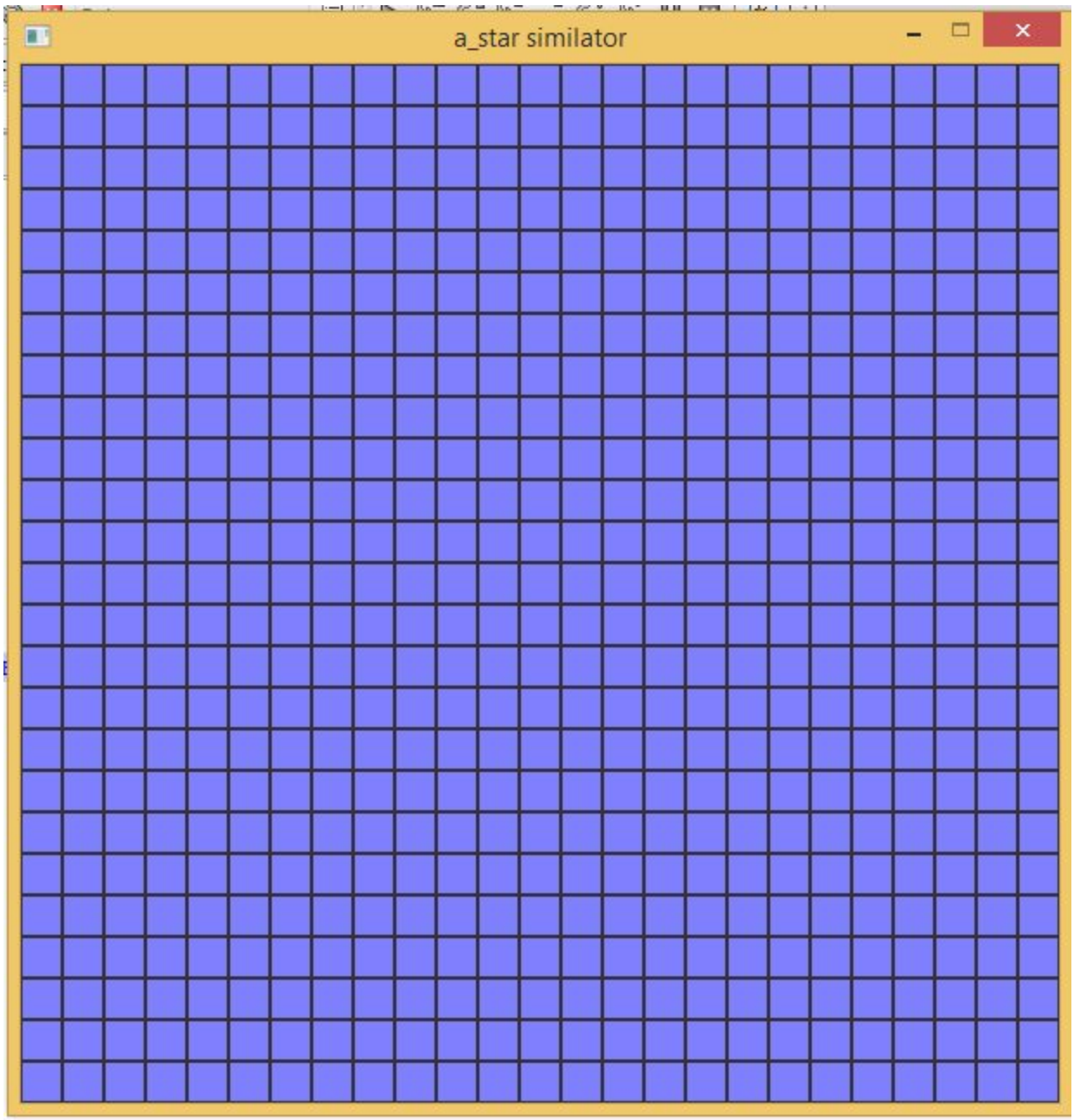
SIMULATION DE PROGRAMME

Le programme commence par affichage des instructions d'utilisation :

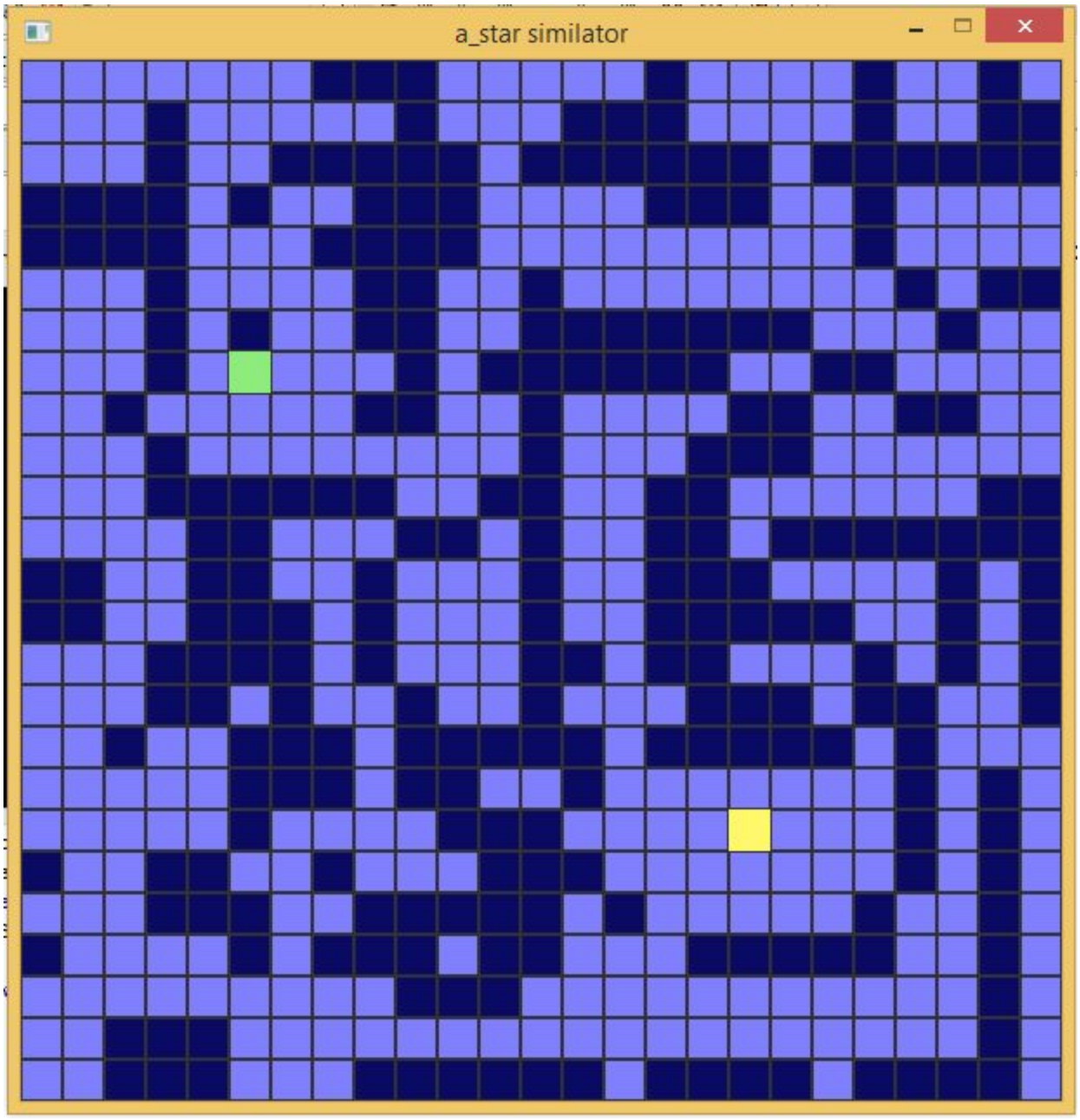


Clic gauche	choisir le départ
Clic droite	choisir la destination
LEFT SHIFT + Glisser la Souris	placer les obstacles
LEFT CTRL	Lancer l'algorithme A*
Clic 'r'	Restart

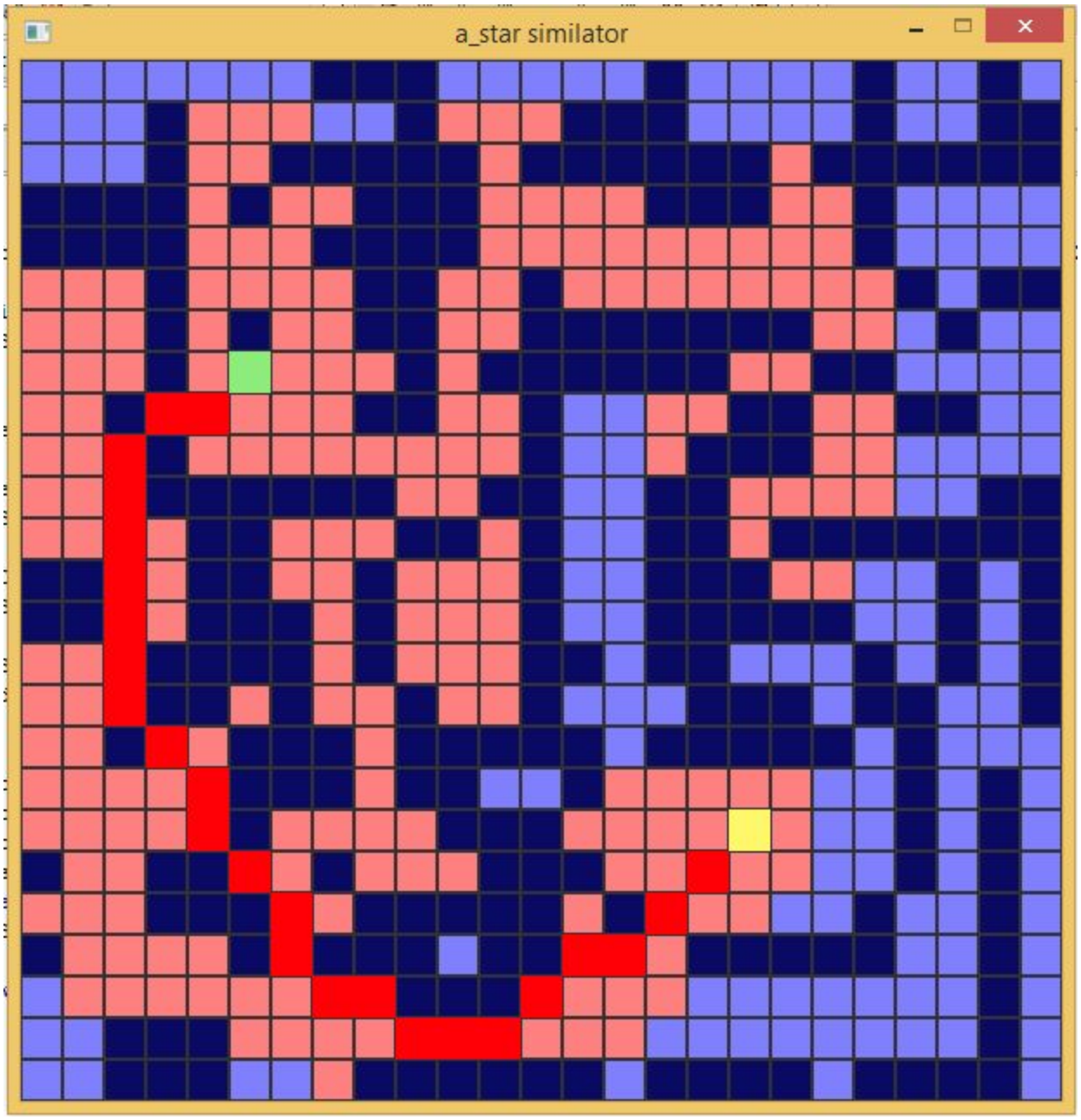
Ensuite la matrice s’affiche:



Maintenant l'utilisateur peut choisir le point de **départ** et d'**arrivée** ainsi que placer les **obstacles** :



Finalement avec un clic sur le bouton CTRL GAUCHE on lance l'algorithme :



les zones en **rouge** représentent le chemin trouvé et les zones en **rose** représentent les zones traité par l'algorithme.