Projet Algo 5 - Partie 3

Licence Informatique - 2nde année

Année 2022-2023

Cette dernière partie du projet va consister à construire un graphe d'exploration du labyrinthe, puis à utiliser celui-ci pour pouvoir trouver un chemin entre un point de départ et un point de sortie.

Préliminaires

Vous pourrez partir des fichiers sources correspondant à la correction de la partie 2 du projet, afin de disposer des fonctionnalités opérationnelles qui seront utiles pour votre application. On précise que l'application peut être lancée en fournissant les dimensions de la grille sur la ligne de commande.

Exemple La ligne de commande ci-dessous permet de créer une grille de largeur 20 et de hauteur 15.

```
./laby  20  15
```

Par défaut (lorsque les valeurs ne sont pas précisées) la grille sera de taille 10×10.

Structure de données

La représentation du graphe associé au labyrinthe va s'effectuer sous forme d'une matrice d'adjacence, similaire à ce que vous avez déjà utilisé en TP. La structure de données qui vous est fournie est la suivante, disponible dans le fichier `types.hpp` :

```
/*
 * structure d'un maillon de la liste chaînée associée à chaque ligne de la matrice
 */
struct Maillon {
  int l, c; // coordonnées du noeud dans la grille du labyrinthe
  Maillon *suiv;// élément suivant  sur la ligne
};

/*
 * structure représentant une matrice d'adjacence sous forme
 * de matrice creuse. La structure contient un tableau de lignes, chaque
 * ligne étant représentée par une liste chaînée de "maillons".
 */
struct MatriceAdjacence {
  int ordre; // nombre de sommets du graphe
  Maillon* *lignes; // tableau représentant les lignes de la matrice
};
```

Ici, chaque sommet du graphe sera une case de votre labyrinthe, qui sera identifiée par ses coordonnées dans la grille support de celui-ci. Ses sommets voisins seront les cases voisines qui sont accessibles, c'est à dire pour lesquelles un mur de séparation n'est pas présent. La matrice d'adjacence aura donc une ligne par case de la grille et chaque maillon associé à la ligne contiendra les coordonnées de la case voisine accesssible. Un exemple vous est donné dans la figure 1 ci-après.

```
noeud 0 (0,0)  : (0,1)(1,0)
noeud 1 (1,0)  : (0,0)(2,0)
noeud 2 (2,0)  : (2,1)(1,0)
noeud 3 (3,0)  : (3,1)
noeud 4 (0,1)  : (0,2)(0,0)
noeud 5 (1,1)  : (1,2)
noeud 6 (2,1)  : (2,2)(2,0)(3,1)
noeud 7 (3,1)  : (3,2)(3,0)(2,1)
noeud 8 (0,2)  : (0,1)
noeud 9 (1,2)  : (1,1)(2,2)
noeud 10 (2,2) : (2,1)(1,2)
noeud 11 (3,2) : (3,1)
```

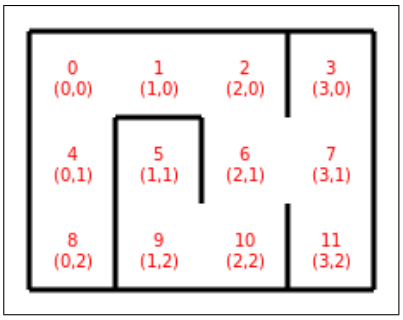

FIGURE 1 – Un exemple d'affichage d'un graphe d'adjacence (à gauche) issu du labyrinthe de droite.

Création de la matrice d'adjacence

En vous inspirant de ce qui a été vu en TP, ajoutez les fonctions suivantes à votre application :

- `void creerMatrice(MatriceAdjacence &m, int taille)`
- `void effacerMatrice(MatriceAdjacence &mat)`
- `void afficherMatrice(MatriceAdjacence &mat)`
- `void remplirMatrice(MatriceAdjacence &mat, const labyrinthe &lab)`

Les trois premières fonctions sont très similaires à ce que vous avez déjà utilisé. En ce qui concerne la quatrième, elle a pour objectif de créer, pour chaque ligne de la matrice, tous les maillons correspondant aux cellules voisines de la cellule correspondant à cette ligne. Elle consiste donc à parcourir chaque case de la grille et à déterminer quelles sont les cellules voisines accessibles. Lorsqu'une cellule voisine est accessible, on la rajoute à la liste chaînée de la ligne correspondante. Vous pourrez vérifier que tout fonctionne bien en faisant afficher un petit labyrinthe au format svg (utilisez une valeur différente pour la largeur et la hauteur de la grille, afin de vérifier que les calculs d'indice sont corrects), puis le contenu de la matrice d'adjacence (cf figure 1).

Dessin du graphe

Complétez votre application avec la fonction suivante :

```
void dessinerMatrice(const MatriceAdjacence &mat, const labyrinthe &laby, const string &nom)
```

dont l'objectif est de dessiner, dans un fichier au format *SVG*, le labyrinthe et son graphe de parcours (voir figure 2). Ses paramètres sont :

- `mat` la matrice d'adjacence construite à la question précédente, qui représente le graphe de parcours ;
- `laby` le labyrinthe qui correspond à la matrice d'adjacence ;
- `nom` le nom du fichier au format *SVG* dans lequel le dessin sera sauvegardé.

Pour réduire la taille des fichiers *SVG*, il est demandé de ne pas tracer 2 fois le même segment lorsque deux sommets sont connectés par une arête commune.

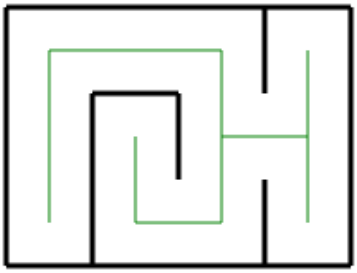

FIGURE 2 – Aperçu du graphe du labyrinthe présenté en figure 1.

Recherche d'un chemin dans le labyrinthe

Dans cette dernière partie, vous allez coder les différentes fonctions permettant de rechercher un chemin entre deux cases du labyrinthe et d'afficher le résultat au format *SVG*.

Saisie des coordonnées

Développez et testez le code de la fonction suivante, qui permet de saisir les coordonnées de la case de départ (`deb`) et celles de la case d'arrivée (`fin`) dans le labyrinthe :

```
void saisirCoordonnees(coordonnee &deb, coordonnee &fin, int largeur, int hauteur);
```

Le type `coordonnee` est défini dans le fichier `types.hpp` de la manière suivante :

```
struct coordonnee {
  int x, y; // abscisse et ordonnée d'une case du labyrinthe
};
```

En sortie de la fonction, les coordonnées doivent obligatoirement être comprises entre 0 et $(largeur-1)$ pour leur abscisse (largeur du labyrinthe) et entre 0 et $(hauteur-1)$ pour leur ordonnée (hauteur du labyrinthe). Cette fonction sera appelée après construction du graphe de parcours, pour demander les cases de départ et d'arrivée dans le labyrinthe.

Calcul des chemins

Pour calculer le chemin entre un sommet de départ et un sommet d'arrivée, vous allez utiliser le parcours en largeur qui a été vu en cours et en TP. Celui-ci permettra de trouver le chemin le plus court dans le graphe entre les deux cases, votre graphe étant un graphe non orienté et non pondéré. Ce dernier sera utilisé dans la fonction suivante :

```
chemin calculerChemin(const MatriceAdjacence &mat, coordonnee deb, coordonnee fin, int largeur);
```

avec :

- `mat` la matrice d'adjacence représentant le graphe de parcours du labyrinthe ;
- `deb` les coordonnées de la case de départ dans la grille sous-jacente au labyrinthe ;
- `fin` les coordonnées de la case d'arrivée dans cette même grille ;
- `largeur` la largeur de la grille, qui sera nécessaire pour transformer des coordonnées en indice de sommet du graphe et inversement.

Cette fonction réalisera les tâches suivantes :

- création des tableaux nécessaires au parcours en largeur ;
- remplissage des tableaux en utilisant ce parcours (attention, la fonction codée en tp nécessitera quelques ajustements ...) ;
- création du chemin entre le point de départ et le point d'arrivée à partir des tableaux créés. Ce chemin créé sera retourné par la fonction et aura la structure de données suivante :

```
struct Chemin {
  int lg; // longueur du chemin (nb de cases du tableau etape)
  coordonnee *etape;// les coordonnées des différents sommets à parcourir
};
```

On précise que le champ `etape` devra être alloué dynamiquement lorsque la taille du chemin sera connue (*cf* le tableau des distances créé dans le parcours en largeur ...).

Vous incluerez dans votre application les fonctions qui sont nécessaires au parcours en largeur, ainsi que les types qui devront être utilisés dans le fichier `types.hpp`.

Affichage de la solution

Développez enfin la fonction suivante :

```
void dessinerSolution(const labyrinthe &laby, const chemin &ch, const string &nomFichier);
```

qui a pour objectif de dessiner le labyrinthe (`laby`) et le chemin qui a été trouvé (`ch`) pour aller de la

case de départ à la case d'arrivée. Le dessin sera sauvegardé dans le fichier au format *SVG* dont le nom sera passé en troisième paramètre.

Version finale

Dans sa version finale, votre application devra générer trois fichiers `SVG` :
— un fichier nommé `laby.svg` qui représentera le labyrinthe *vierge* ;
— un fichier nommé `graphe.svg` qui montre le graphe de parcours de ce labyrinthe ;
— un fichier nommé `solution.svg` qui montrera le chemin à parcourir dans le labyrinthe entre les deux cases sélectionnées par l'utilisateur.

La figure 3 ci-dessous illustre ces trois types de sortie, sur un labyrinthe de taille 20×15.

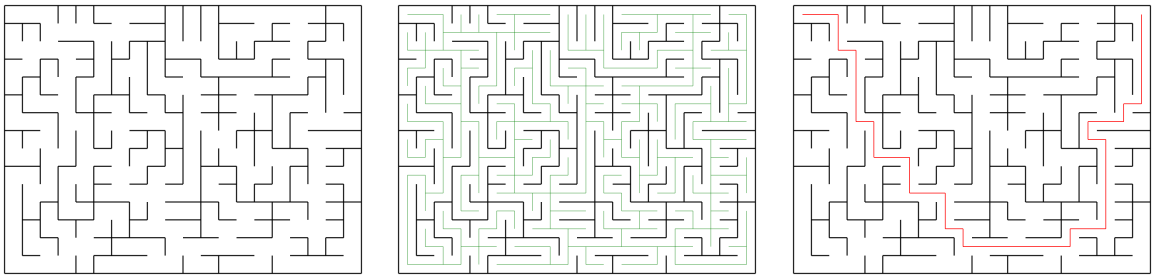

FIGURE 3 – Les trois sorties requises pour l'application : à gauche, le labyrinthe vierge ; au centre, le labyrinthe et son graphe de parcours ; à droite, le labyrinthe et le chemin le plus court entre les case $(0, 0)$ et $(19, 0)$.