Structures de données et algorithmes II

Devoir de TP libre
2020 - 2021

Cadre du problème

Historiquement, il s'agit d'un palais de la mythologie grecque construit en Crète, par un architecte du nom de « Dédale » et servant de prison à la créature Minotaure. Il existe de nombreux jeux basés sur le principe d'un labyrinthe. Les scénarios de jeu sont variés : le joueur doit trouver la sortie, donc un chemin reliant le départ et l'arrivée, ou bien il doit récolter des trésors répartis dans un dédale constitué de passages enchevêtrés, etc. La figure ci-dessous présente le célèbre jeu Pac-Man édité par la société Namco dans les années 80. Comme on peut le voir, l'action se déroule dans un labyrinthe dans lequel il faut récolter des pastilles sans se heurter aux fantômes. Nous souhaitons modéliser un labyrinthe et automatiser la recherche de chemins.

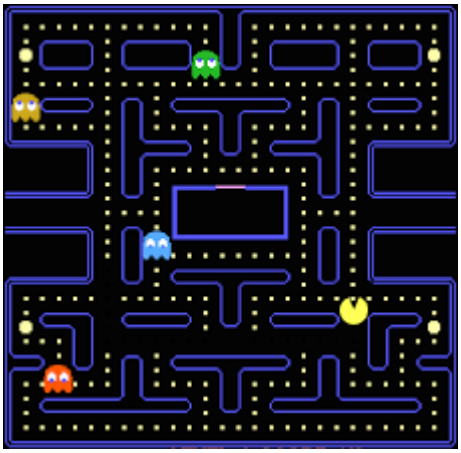


FIGURE 1 – Un exemple de labyrinthe utilisé pour un jeu vidéo.

1. Modélisation d'un labyrinthe

Une façon classique de coder un labyrinthe consiste à le représenter sous la forme de blocs en utilisant une matrice de cellules carrées contenant une valeur. Au plus simple, cette valeur est de type booléen : vrai signifie passage, faux signifie mur. On peut facilement dessiner un labyrinthe de ce type, tel que chaque bloc est peint soit blanc (passage) soit noir (mur) comme illustré par la figure ci-dessous, à gauche.

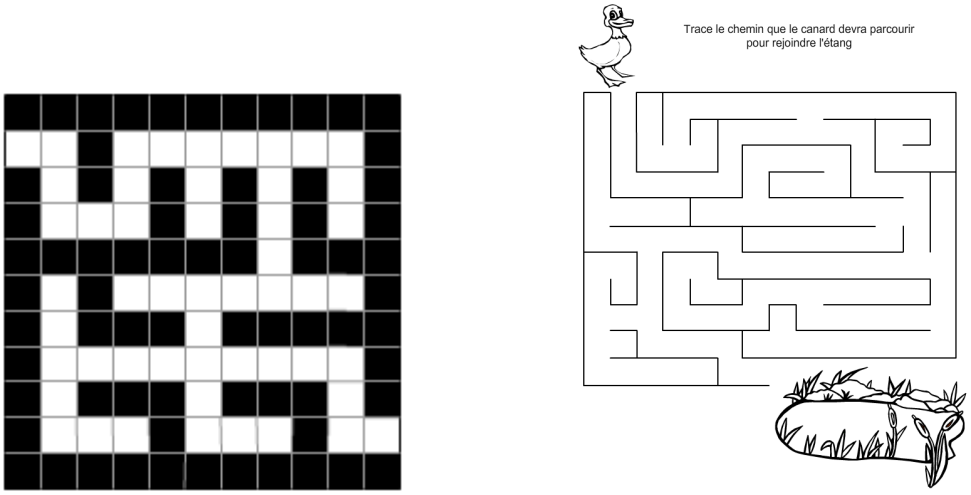


FIGURE 2 : A gauche, un labyrinthe constitué de blocs « booléens » (blanc/noir). A droite, un labyrinthe constitué de chemins et carrefours.

Mais une telle représentation informatique limite beaucoup l'architecture du labyrinthe. Nous allons donc plutôt utiliser une représentation chainée, basée sur une liste d'adjacence, qui nous permettra de représenter un labyrinthe par un ensemble de carrefours et chemins (comme celui de la figure 2 à droite). Nous définissons la structure de données comme suit :

```
#define NDIR 4
typedef enum { NORD=0, EST, SUD, OUEST } Dir ;

typedef struct s_noeud {
     struct s_noeud     *voisins[NDIR];
     int                distances[NDIR];
} *noeud ;
```

Le labyrinthe est constitué d'un ensemble de nœuds (ce sont les carrefours) reliés entre eux par des passages horizontaux ou verticaux. En chaque nœud on peut se déplacer dans l'une des 4 directions cardinales, Nord, Est, Sud, Ouest (modélisé dans la structure par 4 pointeurs « voisins ») sous réserve qu'il existe un passage. Une direction ***d*** de type ***Dir*** ne peut être prise que si le pointeur ***voisins[d]*** est différent de nul. Si ce pointeur est différent de nul alors ***distances[d]*** indique la longueur du passage, càd la distance à parcourir en mètres pour arriver jusqu'au prochain carrefour (nœud). Un nœud est un cul-de-sac s'il ne permet de prendre qu'une seule direction : càd que tous les pointeurs ***voisins*** sont nuls sauf un seul. On suppose qu'il n'existe pas de nœud isolé, donc au moins un des pointeurs ***voisin*** est différent de nul.

Une condition importante permettant de préserver une structure de données cohérente est la suivante : soit une direction **d** et deux nœuds **n1** et **n2**, tels que **n1->voisin[d]=n2**, alors **nécessairement** :

n2->voisin[(d+2)%4] == n1 ET n1->distances[d] == n2->distances[(d+2)%4]

Notez que (d+2)%4 permet d'obtenir la direction opposée à d !

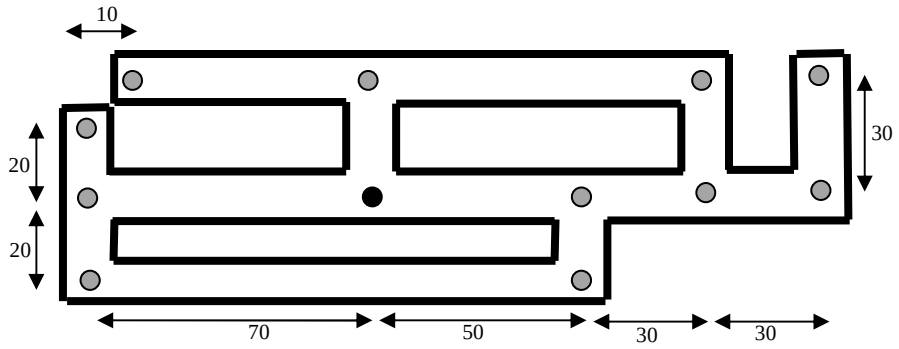
Cette condition modélise simplement le fait que s'il existe un passage de longueur ***l*** de ***n1*** vers ***n2*** alors il doit aussi exister un passage de ***n2*** vers ***n1*** ayant la même longueur.

La seconde condition est que le graphe formé par le labyrinthe doit être connexe : c'est-à-dire que tout nœud du labyrinthe peut être atteint à partir de n'importe quel autre nœud par au moins un chemin (un chemin est une suite de passages).

1.1 Ecrire les opérations de création de nœud et de passages suivantes :

```
noeud nouvnoeud() ;
void passage(noeud n1, noeud n2, Dir d, int dist);
```

1.2 En utilisant les deux opérations précédentes, écrire un code permettant de créer le labyrinthe suivant :



Sur cette figure, chaque point (gris ou noir) représente un carrefour (un nœud). Les flèches indiquent les distances en mètres. Il y a 12 nœuds, dont 3 sont des culs-de-sac.

2. Déterminer la taille ainsi que des positions au sein du labyrinthe

2.1 Partant d'un nœud donné, on souhaite déterminer la taille totale en X et en Y du labyrinthe. Dans l'exemple précédent, le labyrinthe a pour dimensions X=180 et Y=50. Ecrire un code qui calcule X et Y à partir d'un nœud quelconque :

```
void taille(noeud n, int &X, int &Y);
```

Notez que comme le labyrinthe est connexe, peu importe le nœud de départ. En d'autres termes, le résultat devra toujours être le même quelque soit le nœud ***n*** fourni en paramètre !

2.2 Ecrire une opération permettant de déterminer la position X et Y (en mètres) d'un nœud. Par exemple, le point noir de la figure précédente est positionné en X=70 et Y=30 (en partant de la gauche vers la droite et du haut vers le bas) :

```
void position(noeud n, int &X, int &Y);
```

3. Longueur des passages dans le labyrinthe

3.1 On souhaite déterminer la somme des distances de tous les passages du labyrinthe, donc la longueur totale du labyrinthe. Pour l'exemple de la figure, il y a 13 passages (7 horizontaux et 6 verticaux) entre les noeuds, formant une longueur totale de 590m.

```
int longueur(noeud n);
```

4. Créer une image de labyrinthe

Dans cet exercice on se propose de créer des fichiers d'images permettant d'afficher des labyrinthes. Il existe de très nombreux formats d'images. Nous utilisons ici le format pgm. Il fait partie d'une famille de formats (ppm, pgm, pbm) décrits plus en détail sur ce site https://fr.wikipedia.org/wiki/Portable_pixmap.

— le format pbm pour les images en noir et blanc ;
— le format pgm pour les images en niveaux de gris ;
— le format ppm pour les images en couleur.

L'avantage de ces formats est qu'il permet de représenter une image sous la forme d'un fichier texte. On peut donc le lire et le modifier *avec un simple éditeur de texte*.

Dans ce qui suit nous utilisons exclusivement le format pgm. Un fichier au format pgm :
— commence toujours par le mot P2 suivi d'un espace ou d'un retour à la ligne. Ces deux caractères sont appelés le magic number et permettent aux applications de lecture et d'édition d'image de reconnaître une image au format pgm
— doit ensuite comporter deux entiers représentant respectivement la largeur et la hauteur de l'image en pixels ;
— doit encore comporter un entier représentant l'intensité du blanc (généralement 255) ;
— tous les entiers qui suivent sont interprétés comme la valeur des pixels de l'image.

Par exemple le fichier pgm suivant :

```
P2
10 5
255
127 127 127 127 127 127 127 127 127 127 0 0 0 0 0 0 0 0 0 0
255 255 255 255 255
255 255 255 255 255 200 180 160 140 120 100 80 60 40 20
```

représente une image de 10 pixels de large et 5 pixels de haut. Le blanc est représenté par 255. Les trois premières lignes sont appelées l'entête du fichier image. Ensuite viennent une suite d'entiers représentant l'intensité de tous les pixels. La première valeur (ici 127) représente le pixel en haut à gauche de l'image. Ensuite viennent les pixels de la première ligne puis ceux de la deuxième ligne et ainsi de suite. L'image ci-dessus comporte une ligne de pixels (10 pixels) à 127, c'est à dire gris moyen. La deuxième ligne (10 pixels suivants) contient des valeurs à 0 (donc noir). Les 10 pixels suivants sont à 255 (donc blanc), etc.

Pour vous convaincre, lancer un éditeur de texte. Ouvrir un nouveau document et copier-coller le contenu ci-dessus dans ce fichier. L'enregistrer avec un nom *ayant une extension .pgm* (supposons que vous l'ayez appelée *essai.pgm*). Vous avez créé très simplement une image. Pour la voir, vous pouvez trouver le fichier dans un navigateur de fichier et double-cliquer dessus. L'image s'affichera.

Connaissant à présent le format de fichier pgm, écrire un programme permettant d'enregistrer le labyrinthe créé dans l'exercice précédent dans un fichier image.

Attention un pixel d'image ne correspond pas nécessairement à un mur. En effet, nous utilisons une échelle de grandeur telle que chaque mètre carré correspond à une taille de N×N pixels, et chaque passage a une épaisseur de M<N pixels dans l'image, les grandeurs N et M étant des paramètres de la fonction d'enregistrement. Un N grand (par exemple N=10) permettra de faire de « grandes » images même pour des labyrinthes très petits.

Indication : une façon simple de procéder consiste à initialiser une image avec des pixels tous noirs. Cela représente les murs. On peut ensuite « creuser » des passages d'épaisseur M pixels en suivant tous les chemins possibles et en modifiant les pixels rencontrés. Rappel : l'image doit avoir pour taille NX × NY pixels, X et Y étant la taille totale du labyrinthe et N étant le nombre de pixels par mètre.

5. Recherche de trésors dans le labyrinthe

A présent, nous allons placer des objets dans le labyrinthe. Comme les pastilles pour Pac-Man. Pour cela nous devons modifier la structure de données, de façon à pouvoir ajouter à chaque passage une liste de trésors (ou objets précieux). Chaque trésor a une valeur donnée par un float.

Nous utilisons le codage suivant : à chaque passage entre deux noeuds, on associe un tableau de ***l*** valeurs, ***l*** étant la distance en mètres du passage (une valeur par mètre). Si la valeur vaut 0.0 alors il n'y a rien à cet emplacement, sinon c'est un trésor dont la valeur est donnée par le float :

```
typedef struct s_noeud {
     struct s_noeud     *voisins[NDIR];
     int                distances[NDIR];
     float              *tresors[NDIR];
} *noeud ;
```

Pour chaque direction ***d***, ***tresor[d]*** est un pointeur de float, qui donne l'adresse d'un tableau contenant exactement ***distances[d]*** éléments. En d'autres termes, si l'on reprend l'exemple de Pac-Man, chaque pastille correspond à un float dans notre cas. Et il y a autant de pastilles (de floats) que de mètres sur un chemin.

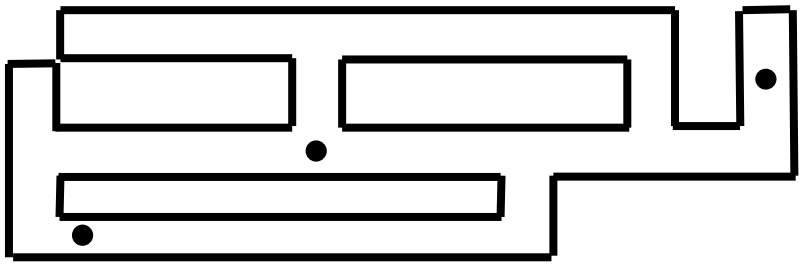
Il faut à présent réécrire l'opération de création de passage :

```
void passage(noeud n1, noeud n2, Dir d, int dist, float tr[]);
```

Une précondition est que la taille du tableau ***tr[]*** doit exactement valoir ***dist*** (un float par mètre). Attention à ne pas oublier d'initialiser le passage inverse de ***n2*** vers ***n1*** !

Noter qu'un trésor peut éventuellement se trouver exactement sur un nœud. La figure ci-dessous montre un labyrinthe dans lequel 3 trésors d'une valeur de 1000.0 chacun sont placés (points noirs). Un des trésors se trouve exactement sur un nœud.

Ecrire un code permettant de créer ce labyrinthe (en utilisant la fonction ***passage*** modifiée).



Ecrire un code qui, partant d'un nœud donné, explore le labyrinthe et renvoie la valeur totale de tous les trésors qui s'y trouvent. Pour le cas de l'exemple, la fonction renverra 3000.0, étant donné qu'il y a trois trésors d'une valeur 1000.0 chacun.

```
float valeurTotalTresors(noeud n);
```

Ecrire un code qui, partant d'un nœud donné, renvoie la valeur du trésor **le plus proche**, càd le trésor pour lequel il faut parcourir la distance en mètre la plus petite, en partant du nœud fournit en paramètre.

```
float valeurPlusProcheTresors(noeud n);
```

Attention: pour atteindre un trésor, on est pas forcé de traverser l'intégralité d'un passage. Par exemple si le trésor se trouve au milieu d'un passage, on ne parcourt que la moitié de la distance de ce passage. Il faut donc déterminer **la distance exacte** à parcourir en mètre pour atteindre un trésor, et ainsi déterminer le plus proche.

Remarques

Le TP libre est à déposer pour le **lundi 3 mai 2021**. Le fichier archive contiendra :

- Les codes sources commentés du programme.
- Une image pgm de labyrinthe.
- Un document word/rtf/pdf illustrant et documentant la réalisation (max. 5 pages).

Le projet est à réaliser seul.

La note est fonction de la quantité et qualité ***du travail personnel*** réalisé.