Projet d'Algorithmique et Programmation: Jeu du Labyrinthe

Christian Artigues et Sophie Demassey

1 Présentation du jeu

L'objectif de ce projet est de réaliser un jeu en C++. L'énoncé du jeu est simple. Au cours d'une partie, 4 joueurs (au plus) démarrent aux 4 coins d'un labyrinthe fermé et doivent atteindre une cible située au centre du labyrinthe (voir écran 1). 4 touches (par exemple Z, W, S et D) permettent de se déplacer si c'est possible dans une des 4 directions haut, bas, droite, gauche. L'objectif pour chaque joueur est d'atteindre la cible en un minimum de coups (et avant qu'un des autres l'atteignent s'il n'est pas tout seul). Les joueurs jouent à tour de rôle dans un ordre tiré aléatoirement au départ. La partie s'arrête lorsqu'un des joueurs arrive à la cible ou lorsque les commandes spéciales suivantes sont entrées.

- A tout moment, on peut (si tous les joueurs sont d'accord) demander à l'ordinateur de terminer la partie en affichant les chemins restant à parcourir pour chacun des joueurs afin d'obtenir le meilleur score (Ecran 2).

- A tout moment, on peut (si tous les joueurs sont d'accord) demander à l'ordinateur d'afficher quels étaient les meilleurs chemins à partir de la position de départ (Ecran 3).

En cas de victoire d'un joueur, l'affichage de la meilleure solution à partir de la position initiale se fait automatiquement. Si les joueurs demandent l'affichage de la meilleure solution à partir de la position initiale, la partie est annulée. Si les joueurs demandent l'affichage de la meilleure solution à partir de la position courante, le gagnant est celui qui a le plus petit nombre de coups (ceci permet de faire des paris!).

Le jeu se déroule en n parties. A l'issue d'une partie, le joueur qui a gagné augmente son score de 1. Le gagnant est celui qui a le plus grand score à la fin des n parties.

2 Travail à réaliser

Le programme, à réaliser en C++, comporte plusieurs classes dont les suivantes:

- la classe `Labyrinthe` contient les caractéristiques d'un labyrinthe : une **matrice carrée de caractères** à allouer dynamiquement en fonction de la **taille** du labyrinthe. La **densité** du labyrinthe donne le pourcentage de cases libres. La **position** de la cible est également stockée. La classe contient également les méthodes suivantes :
  - un **constructeur** qui génère un labyrinthe à partir d'une taille et d'une densité. Ce constructeur alloue la place nécessaire à la matrice et la remplie aléatoirement, en respectant la densité. La cible est positionnée le plus près possible du centre en utilisant la fonction `plusProcheLibre` (voir ci-après).
  - un **destructeur** qui libère la mémoire allouée
  - les **accesseurs** habituels.
  - une méthode `plusProcheLibre` donnant les coordonnées de la case libre la plus proche de la case donnée en entrée.
  - une méthode `plusCourtChemin` calculant le plus court chemin d'une case à une autre dans le labyrinthe (voir paragraphe 3)
  - une méthode `affiche` qui affiche le labyrinthe à l'écran ainsi que les joueurs donnés en entrée de la fonction sous la forme d'une liste d'objets de classe Joueur (cf écrans 1,2 et 3 pour gérer l'affichage). Un indicateur, fourni en entrée, précise si on doit afficher en mode Jeu (écran 1), en mode Solution à partir de la position courante des joueurs (écran 2) ou en mode Solution à partir de la position initiale (écran 3).
- la classe `joueur` contient les caractéristiques d'un joueur : Son **identifiant**, son **score** (le nombre de parties gagnées depuis le début du jeu), sa **position**

initiale dans la partie en cours, sa **position courante** dans la partie en cours, le **nombre de déplacements** effectués depuis le début de la partie, le **meilleur chemin**[1] à partir de la position initiale jusqu'à la cible et le **meilleur chemin** à partir de la position courante. La classe `Joueur` possède les méthodes suivantes :

- un **constructeur** qui initialise l'identifiant et le score (passés en entrée).
- une fonction `calculePositionInitiale` qui initialise la position en fonction de l'identifiant et d'un labyrinthe passé en entrée. Le premier joueur est en haut à gauche, le deuxième joueur est en haut à droite, le troisième joueur est en bas à gauche, et le quatrième joueur est en bas à droite. On utilisera la fonction `plusProcheLibre` de la classe `Labyrinthe` pour trouver une position initiale libre le plus près possible de la case voulue.
- une fonction `calculeMeilleurCheminInit` qui calcule le meilleur chemin à partir de la position initiale vers la cible (en allouant la mémoire nécessaire au chemin puis en utilisant la fonction `plusCourtChemin` de la classe `labyrinthe`)dans un labyrinthe passé en entrée. Si un tel chemin n'existe pas la mémoire réservée pour le chemin est détruite et le pointeur correspondant est initialisé à NULL.
- un **destructeur** qui libère la mémoire éventuellement allouée aux deux chemins.
- les **accesseurs** habituels.
- des **méthodes de déplacement** du joueur (une pour les quatre directions possibles). Ces méthodes n'effectuent le déplacement que s'il est possible, à partir du labyrinthe passé en entrée et mettent à jour le score (+1 pour un déplacement réussi).
- une méthode `aGagne` qui teste si un joueur a gagné (il est sur la cible du labyrinthe passé en entrée).
- une méthode de **calcul du meilleur chemin** depuis la position courante. Cette méthode suppose que ce chemin existe toujours.

- une classe `Jeu` qui possède **une liste de joueurs**, un **numéro** de partie courante et le nombre de partie à effectuer. Elle a les méthodes suivantes :
  - un **constructeur** qui crée la liste des joueurs (entrée au clavier), initialise le nombre de parties (entrée au clavier) et le numéro de partie.
  - un **destructeur** qui détruit la liste des joueurs (et les objets joueurs également),
  - une méthode `jouePartie` qui joue une partie et renvoie 0 si le jeu est fini et 1 s'il reste encore des parties à jouer. La méthode crée un labyrinthe à partir d'une taille et d'une densité entrées au clavier. Elle donne également la position initiale des joueurs dans ce labyrinthe et initialise leur nombre de coups à 0. Elle lance le calcul des meilleurs chemins depuis la position initiale pour chaque joueur. Si un des chemins ne peut être trouvé, le labyrinthe est détruit, un nouveau est créé et le processus est répété jusqu'à ce qu'un labyrinthe "jouable" soit trouvé[2]. Elle se met ensuite en attente des commandes des joueurs. Référez-vous à la présentation du Jeu (paragraphe 1) pour les cas d'arrêts de la partie. A la fin de la partie le score des joueurs est mis à jour et le nombre de parties est incrémenté. On n'oubliera pas de détruire la mémoire réservée pour le labyrinthe si cette destruction n'est pas déclenchée automatiquement.
  - une méthode **d'affichage du score**

3 Calcul des plus courts chemins

La méthode de plus court chemin dans un labyrinthe entre deux cases données `plusCourtChemin` pourra être implémentée comme suit. Pour plus de simplicité les cases sont numérotées de 0 à $nc = t^2$-1 où t est la taille du labyrinthe. Pour mémoriser un chemin on utilise un tableau C de nc éléments où $C[i]$ désigne la case prédecesseur de i dans le plus court chemin. Par définition si une case i n'est pas dans le chemin on a $C[i] = -1$ et si une case est la première du chemin on a $C[i] = i$. Considérons le chemin suivant (joueur 1) dans un labyrinthe à 4 cases:

```
  0 1 2 3
0 1 1
1   1 C
2
3
```

Les cases sont numérotées de 0 à 15. et le tableau C qui code ce chemin est tel que C[6]=5; C[5]=1; C[1]=0; C[0]=0. Les autres cases sont telles que C[2]=C[3]=C[4]=C[7]=...C[15]=-1.

Voici maintenant l'algorithme qui permet de trouver le plus cours chemin d'une case a à une case b. Il s'agit d'explorer le labyrinthe *en largeur* à partir de la case de départ. L'algorithme utilise une structure de file dont nous expliquons le principe à la fin du paragraphe.

[1] pour le stockage des chemins, on peut s'inspirer du paragraphe 3 concernant la fonction plusCourtChemin

[2] Attention à donner une densité assez importante pour qu'il existe assez fréquemment des solutions!

1. Pour toute case i, initialiser $C[i]$ à -1 et initialiser le marquage à `faux`.
2. initialiser $C[a] = a$ et mrquer a.
3. ajouter a à la file F (initialement vide).
4. Répéter
   (a) récupérer la case k dans F et l'enlever de F.
   (b) Pour toute case l accessible depuis k et **non marquée**
      i. marquer l.
      ii. ajouter l à la file F.
      iii. mise à jour du chemin : $C[l] = k$.
5. tant que b n'est pas marquée et F n'est pas vide.

A la fin de l'algorithme le tableau C décrit le plus court chemin. En partant de la case de fin b, on peut "remonter" le long du chemin jusqu'à rencontrer la case de départ a et compter ainsi le nombre de coups. Si à la fin de l'algorithme la case b n'est pas marquée, c'est qu'il n'existe pas de chemins de a à b.

La file F sera décrite par un objet de classe file. La file sera codée par un tableau d'entier (les cases) ainsi qu'un indice de début (tète) et de fin (queue) de la file.

Une file est un ensemble géré selon le principe FIFO (premier entré premier sorti). La classe file aura une méthode `ajouter` qui permet d'ajouter un élément en queue de file, une méthode `enlever` qui permet de récupérer l'élément en tète de la file et de l'enlever de la file ainsi qu'une méthode `estVide` .

4 Questions

Vous concevrez les classes C++ permettant de modéliser le jeu du labyrinthe et vous implémenterez les différentes méthodes. Vous écrirez un programme C++ utilisant ces classes. Ecrivez le plus complètement possible les définitions de vos classes avant de vous lancer dans la programmation des corps des fonctions membres.

5 Questions Facultatives (faites preuve d'imagination!)

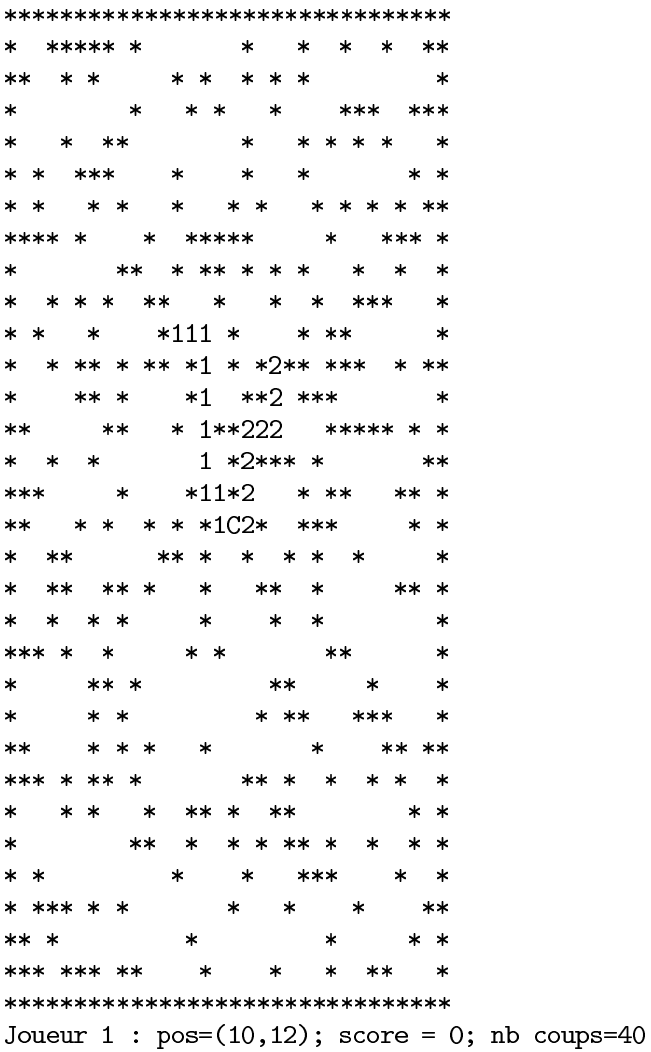
- Vous pouvez gérer une simulation de jeu. A partir des chemins optimaux précalculés, vous gérez automatiquement le déplacement des joueurs en les éloignant aléatoirement de ce chemin.
- Dans le cas où les chemins n'existent pas, au lieu de détruire le labyrinthe et de le reconstruire aléatoirement, vous pouvez essayez de libérer un minimum de cases occupées pour que le chemin vers la cible existe, ce qui permet de générer des labyrinthes "jouables" à faible densité.
- Vous pouvez faire interagir les joueurs : ils peuvent se bloquer la route, etc.

6 Ecrans

Ecran 1

```
********************************
*1 ***** *       *   *  *  * 2**
**  * *     * *  * * *         *
*        *   * *   *    ***  ***
*   *  **        *   * * * *   *
* *  ***    *    *   *       * *
* *   * *   *   * *   * * * * **
**** *    *  *****     *   *** *
*       **  * ** * * *   *  *  *
*  * * *  **   *   *  *  ***   *
* *   *    *    *    * **      *
*  * ** * ** *  * * ** ***  * **
*    ** *    *   **  ***       *
**     **   *  **      ***** * *
*  *  *         * *** *       **
***     *    *  *    * **   ** *
**   * *  * * * C *  ***     * *
*  **      ** *  *  * *  *     *
*  **  ** *   *   **  *     ** *
*  *  * *     *    *  *        *
*** *  *     * *       **      *
*     ** *         **     *    *
*     * *         * **   ***   *
**    * * *   *       *    ** **
*** * ** *       ** *  *  * *  *
*   * *   *  ** *  **        * *
*        **  *  * * ** *  *  * *
* *         *    *   ***    *  *
* *** * *       *   *    *    **
** *         *         *     * *
*** *** **    *    *   *  **   *
********************************
Joueur 1 : pos=(1,1); score=0; nb coups=0
Joueur 2 : pos=(1,29); score=0; nb coups=0
```

Ecran 2



Ecran 3