Introduction à la Programmation

—

Examen final sur machine

—

G1: monasse*(at)*imagine.enpc.fr G2: nicolas.audebert*(at)*onera.fr
G3: alexandre.boulch*(at)*enpc.fr G4: maxime.ferrera*(at)*onera.fr
G5: laurent.bulteau*(at)*u-pem.fr G6: pierre-alain.langlois*(at)*eleves.enpc.fr

—

11/01/19

—

1 Enoncé

1.1 Labyrinthe

Le but de l'exercice est de créer des labyrinthes aléatoires et de voir quelques méthodes de résolution. Un labyrinthe bien formé comporte une et une seule solution, c'est à dire un chemin partant du coin haut à gauche et se terminant en bas à droite sans traverser de mur.

Créez un projet Imagine++ avec un fichier contenant le `main`. Prenez le `CMakeLists.txt` d'un projet existant et adaptez-le.

Il est plus important de livrer un code clair (commenté et indenté) et qui compile sans warning, même s'il ne répond pas à toutes les questions. Pour cela, vérifiez à chaque étape que votre programme compile et se lance correctement. À la fin de l'examen, nettoyez votre code (indentation...) et vérifiez qu'il compile. Créez alors une archive portant votre nom et numéro de groupe.

La base

1. Les cases du labyrinthe occupent un carré de 10×10 pixels (utiliser une constante `zoom`) et on veut afficher le labyrinthe avec une marge de chaque coté, par exemple de 10 pixels (utiliser une constante), dans la fenêtre. Ouvrir une fenêtre suffisamment grande pour afficher un labyrinthe de 30×20 cases.
2. Dans un fichier à part, définir une classe `Mur` comportant deux bool (`hor` et `ver`). Le mur de la case (x, y) indique par `hor` si le chemin vers $(x, y-1)$ est muré (mur horizontal) et par `ver` si le chemin vers $(x-1, y)$ est muré (mur vertical). Le constructeur de `Mur` met ces champs à true.
3. Créer une classe `Labyrinthe` qui stocke un tableau de murs.
4. Définir un constructeur pour le labyrinthe prenant en paramètres les dimensions en nombres de cases. Les labyrinthes n'ont pas forcément tous la même taille, on utilise donc de l'allocation dynamique.

FIGURE 1 – Un labyrinthe initial à la construction (question 9, noter l'entrée en haut à gauche et la sortie en bas à droite), après l'avoir vidé (question 10), et après création (question 12).

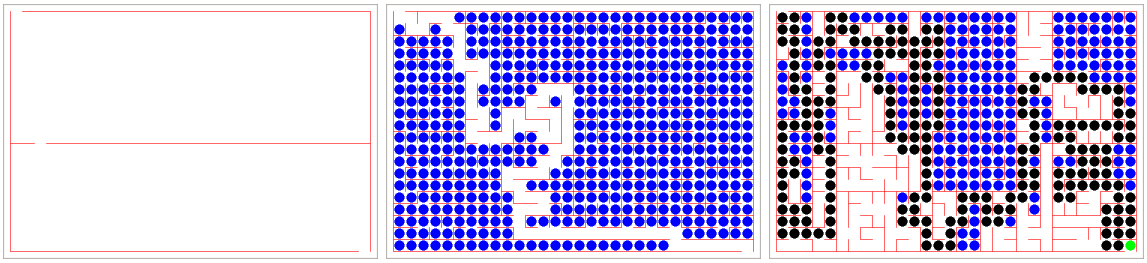

FIGURE 2 – Première séparation lors de la création du labyrinthe (question 11), résolution par bouchage des impasses (question 14) ou par exploration (question 16, chemin solution en noir).

Attention, s'il y a w cases en largeur, il y a $w-1$ murs potentiels entre les cases, plus les murs aux extrémités extérieures, donc $w+1$ murs verticaux par ligne. Idem en hauteur. On utilise $(w+1)\times(h+1)$ murs, même si c'est plus que strictement nécessaire. *On prendra garde dans toute la suite que le nombre de murs et celui de cases ne coïncident pas.*

5. Définir un destructeur pour le labyrinthe.
6. Dans le constucteur, démurer les murs horizontaux de la dernière colonne (puisque ces cases n'existent pas) et verticaux de la dernière ligne. Noter que la dernière case du tableau a ses deux murs à false.
7. Libérer un accès en haut pour la case haut-gauche et une sortie en bas pour la case bas-droite.
8. Créer une méthode d'affichage, prenant en paramètre une marge de pixels pour ne pas afficher directement au bord de la fenêtre. Les murs sont représentés par un segment de longueur `zoom`.
9. Faire afficher un labyrinthe construit ainsi.
10. Créer une méthode `vide` enlevant tous les murs intérieurs et faire afficher un labyrinthe vide.

Création de labyrinthe

Pour créer un labyrinthe, on commence par vider ses murs intérieurs. Puis on sépare par un mur horizontal ou vertical sur toute la longueur. Pour choisir, on compte le nombre de murs intérieurs horizontaux et verticaux, et on en tire un au hasard. Ainsi on a tendance à couper dans la dimension la plus grande. On ménage une ouverture d'emplacement tiré aléatoirement dans ce grand mur. Cela ménage un passage (et un seul) entre les deux sous-rectangles. On procède de même sur chacun des deux et on s'arrête quand toutes les parties n'ont plus qu'une largeur ou hauteur d'une case.

11. Programmer cet algorithme dans une méthode `genere` de `Labyrinthe`. Pour debugger, on pourra à chaque coupure réafficher et attendre la suite par un clic souris. Une fois satisfait, on enlève.
12. Afficher un labyrinthe aléatoire.

Méthodes de résolution

13. Une méthode de résolution consiste à "boucher" les impasses. Une impasse est une case ayant exactement une case voisine accessible (non séparée par un mur), c'est-à-dire entourée de 3 murs. Écrire une méthode `bouche_impasse` qui (a) indique si une case passée en paramètre est une impasse, (b) si c'est le cas donne la case voisine accessible et de plus (c) ferme ce 4ème mur.
14. L'algorithme de résolution consiste à parcourir les cases du labyrinthe. Dès qu'une case est une impasse, on la bouche, la case voisine est peut-être elle aussi une nouvelle impasse et alors on la bouche, ainsi jusqu'à aboutir à une jonction. On reprend alors le parcours des cases. Les cases restantes sont la solution du labyrinthe. Programmer cet algorithme dans une fonction `resolution_impasses` qui marque les impasses bouchées par un cercle bleu.
    Cet algorithme est efficace mais requiert une vue d'ensemble du labyrinthe. Elle n'aide pas le pauvre rat de laboratoire à trouver la sortie. C'est l'objet de la fin du devoir.
15. Écrire une méthode `adjacent` de `Labyrinthe` qui, étant données une case et une direction de voisine $(dx, dy) \in \{-1, 0, 1\}^2 \setminus \{(0, 0)\}$ et $dx \times dy = 0$, indique si ces cases ne sont pas séparées par un mur.
16. L'algorithme utilise une fonction récursive gardant la trace des cases explorées. Étant données une case courante et la case voisine d'où on vient, marquer la case origine avec un cercle noir, et case courante avec un cercle vert, et regarde si une des voisines *accessibles* et *non encore explorée* permet de sortir (appel récursif). Sinon, on marque la case courante en bleu (chemin aboutissant à une impasse) et on revient à la case origine. Implémenter cet algorithme (ça peut vous rappeler quelque chose ?). Pour parcourir les voisins, on peut initialiser $(dx, dy) = (0, 1)$ et utiliser $(dx, dy) \leftarrow (-dy, dx)$, cela permet une boucle économisant du code à rallonge. Noter que cet algorithme peut trouver la sortie avant d'avoir tout exploré, comme dans la figure.