Master Informatique — Approche Objet – 4TIN706U
Université Bordeaux — 2017 – 2018

Projet Approche Objet

Un jeu de plateau *Labyrinthe*

Le but de cet exercice est de mettre en œuvre les éléments acquis lors du cours *Approche Objet*. Pour cela, nous demandons à des groupes de 4 personnes de réaliser un jeu de plateau de type *PacMan*.

Ce projet doit absolument :

— Être documenté par un dossier PDF de quelques pages
— Être documenté sous la forme d'un hypertexte produit par *javadoc*
— Appliquer les *Design Patterns* utiles
— Appliquer les tests unitaires
— Livrer un fichier *labyrinth.jar* qui contient le jeu utilisable immédiatement
— Livrer l'ensemble des sources sous une archive *labyrinth-dev.jar*

Chaque membre du projet doit explicitement dire les parties qu'il a réalisé seul et celles qu'il a réalisé en commun avec d'autres membres du projet, voir d'autres camarades de promotion ou extérieurs à la formation.

Tous les éléments qui ont été trouvés dans la littérature devront être identifiés comme tels et avoir un lien vers les références bibliographiques. Ceci vaut pour le code trouvé sur le *net*.

1 Le jeu – cahier des besoins

Labyrinth est un jeu de plateau où le joueur est représenté par un petit  qui doit trouver la  dans un labyrinthe complexe. Il se déplace vers la droite, vers la gauche, vers le haut ou le bas grâce aux flèches du clavier (nous dirons *EAST, WEST, NORTH, SOUTH*) et doit terminer chaque niveau pour commencer un nouveau nécessairement plus difficile. Le labyrinthe sera généré automatiquement avec des chemins de plus en plus complexes ou le joueur trouvera sur son passage des monstres  qui le poursuivront et qu'il devra éviter et des  qu'il devra ramasser.

En montant dans les niveaux, le joueur sera empêché de passer par des portes closes. Ces portes s'ouvrent et se ferment grâce à des interrupteurs , .

2 Algorithmique

2.1 Labyrinthe

Le labyrinthe est un cadre de 16x16 cellules carrées. comme représenté 1

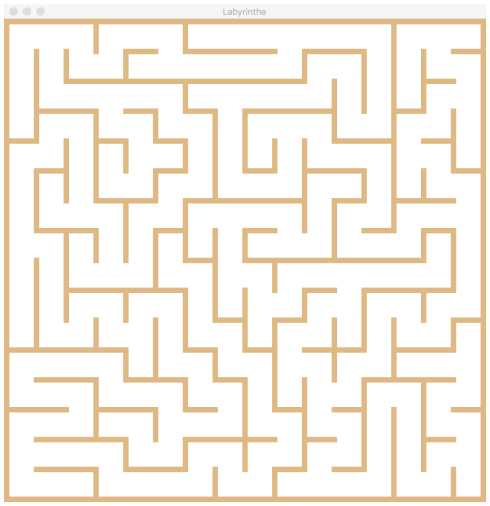

FIGURE 1 – Labyrinthe

Nous le modéliserons sous la forme d'un graphe correspondant à une surface connexe. Les cellules du labyrinthe sont les sommets du graphe et une communication entre deux cellules est un arc entre les deux sommets. Si le graphe est connexe, on dit que le labyrinthe est parfait. Ceci signifie qu'il existe toujours au moins un chemin entre deux cellules.

Par exemple, le labyrinthe 2 est représentée par le graphe 3

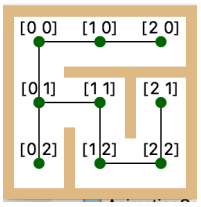

FIGURE 2 – Chemin

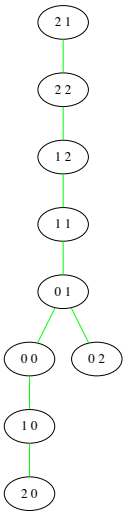

FIGURE 3 – Graphe

Algorithme de création d'un labyrinthe

Pour construire un labyrinthe parfait, il suffit de produire un chemin depuis une cellule donnée qui couvrira successivement toutes les cellules.

Listing 1 – Build labyrinth

```
Given
graph: empty directed acyclic graph
add v = (x,y) a vertex

Result: graph

procedure buildLabyrinth(v)
   (c_1, c_2, c_3, c_4) = random draw of {North, East, South, West}
   for i=1 to 4
   if (c_i = North)
      if (x,y-1) vertex doesn't exist in graph and y>TOP_BORDER
          add v' = (x, y-1) vertex to graph
          add v -> v' edge to graph
          buildLabyrinth(v');
   elseif (c_i = East)
      if (x+1,y) vertex doesn't exist in graph and y<RIGHT_BORDER
          add v' = (x+1, y) vertex to graph
          add v -> v' edge to graph
          buildLabyrinth(v');
   elseif (c_i = South)
      if (x,y+1) vertex doesn't exist in graph and y<SOUTH_BORDER
          add v' = (x, y+1) vertex to graph
          add v -> v' edge to graph
          buildLabyrinth(v');
   elseif (c_i = West)
      if (x-1,y) vertex doesn't exist in graph and y>WEST_BORDER
          add v' = (x-1, y) vertex to graph
          add v -> v' edge to graph
          buildLabyrinth(v');
```

Algorithme pour produire un labyrinthe plus complexe

Pour produire un labyrinthe qui contient un îlot, c'est-à-dire une zone non accessible depuis un ensemble de cellules, il suffit de supprimer un arc dans le graphe connexe. Pratiquement, on ne supprimera pas réellement cet arc, mais on le marquera par une étiquette permettant d'identifier une porte momentanément fermée.

Pour ajouter une porte d'une cellule à une autre, il suffit d'ajouter un arc entre deux sommets du graphe tels qu'il sont voisins.

Pour savoir si deux cellules (x_1, y_1), (y_1, y_2) sont voisines, il suffit de vérifier

$$(d_x = 0 \wedge d_y = 1) \vee (d_x = 1 \wedge d_y = 0)$$

où $dx = |x_1 - x_2|$ et $d_y = |y_1 - y_2|$.

Algorithme pour placer la porte, les bonbons et les méchants

Il convient de placer la porte la plus éloignée possible du joueur. Les méchants doivent s'approcher dangereusement au fil du jeu, etc.

Pour connaître la longueur d'un chemin entre deux cellules et pour permettre de déplacer un méchant en direction du joueur, nous pourrons utiliser l'algorithme dit de Manhattan :

Algorithme de Manhattan

Il porte le nom de ce quartier de New York qui est construit avec des rues perpendiculaires ; telles qu'une distance d'un point A à un point B (appellée distance de Manhattan) est $|x_B - x_A| + |y_b - y_a|$. L'algorithme de Manhattan est une méthode très simple pour trouver le chemin le plus court tracé sur ces rues. Cet algorithme est par exemple employé lors du tracé des pistes d'un circuit imprimé.

L'idée est d'étiqueter le graphe par un nombre croissant selon un parcours en largeur d'un point A à un point B. Lorsque le point B est atteint, il suffit alors de décompter du point B au point A et l'on découvre le chemin le plus court.

Nous illustrons cela par la figure 4

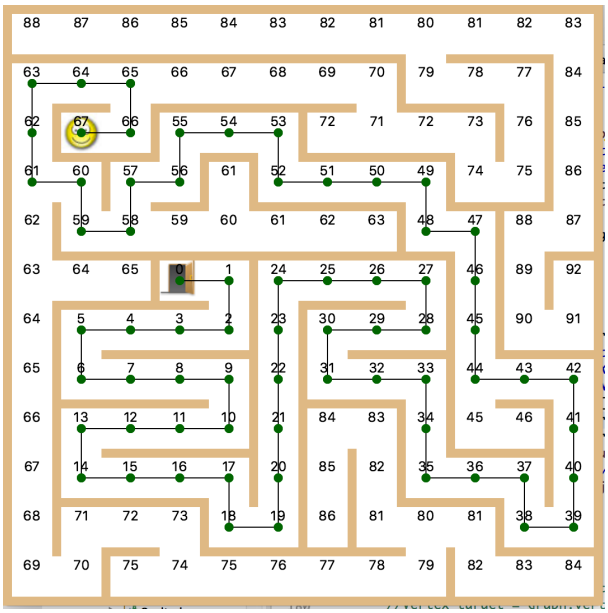

FIGURE 4 – Algorithme de Manhattan

Listing 2 – Algorithme de Manhattan implémenté en Java

```
private void privateManhattan(Vertex source, Vertex target){
        Queue<Vertex> fifo = new ArrayDeque<Vertex>();
        target.setNbr(1);
        fifo.add(target);
        while(!fifo.isEmpty()){
                Vertex actual = fifo.remove();
                for (Directions dir : Directions.values()) {
                        if (this.isOpened(actual, dir)){
                                Vertex next = graph.getVertex(actual, dir);
                                if (next.getNbr()==0){
                                        next.setNbr(actual.getNbr()+1);
                                        if (next!=source)
                                                fifo.add(next);
                                }
                        }
```

```
                    }
                }
            }
```

Remarque : cet algorithme peut être aussi utilisé pour trouver les sommets les plus éloignés.

Algorithme pour produire un jeu réalisable

Si, sur le chemin vers la porte, on rencontre systématiquement un méchant, il n'est pas possible de terminer le niveau. Pour éviter cette situation, il convient de produire des chemins supplémentaires.

Un chemin est modélisé par une séquence $(x_1, x_2, \ldots, x_k)$ où x_1 est le sommet où se trouve le départ et x_k l'arrivée.

Un méchant se déplace vers le joueur en suivant un chemin de Manhattan $(m_1, m_2, \ldots)$. Le méchant se trouve à l'instant t_0 en m_1, il se trouvera à la position m_{k+1} à un instant $t_0 + k$.

On estime donc cette position future m_{k+1}. Et s'il n'existe pas de chemin $(x_1, x_2, \ldots, x_k)$ tel que $\forall i, m_{k+1} \neq x_i$, cela veut dire que le méchant croisera nécessairement le joueur.

Dans ce cas, il faut produire un arc nouveau ou déplacer le méchant dans une position moins critique.

3 Bibliothèques retenues

Pour simplifier l'écriture du code Java, nous demandons aux étudiants d'utiliser les bibliothèques suivantes :

org.jgrapht.* Permet de manipuler simplement des graphes.

javafx.event.* Pour créer une interface homme machine à l'aide du clavier de l'ordinateur

javafx.scene.*, javafx.stage.* Pour créer une interface homme machine graphique

java.util.Timer Pour produire des événements à temps régulier

4 Code fourni

Nous produisons le code qui permet l'affichage du cadre du jeu, images, etc.