

Projet Approche Objet

version revue – samedi 25 novembre 2017

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 niveau nécessairement plus difficile. Le labyrinthe sera généré automatiquement avec des chemins de plus en plus complexes où le joueur trouvera sur son passage des monstres  qui le poursuivront et qu'il devra éviter et des  qu'il devra ramasser.

¹En 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 , . Elle permettent par exemple de piéger les monstres dans des îlots du labyrinthe, le temps de rejoindre la sortie.

1. Cette dernière partie est laissée en option pour les étudiants qui voudraient augmenter la qualité du projet. Elle demande un travail en algorithmique pour que le jeu reste jouable.

2 Algorithmique

2.1 Labyrinthe

Le labyrinthe est un cadre de 16x16 cellules carrées. comme représenté 5

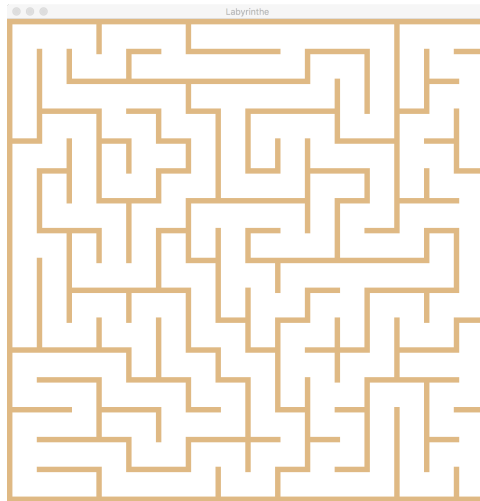


FIGURE 1 – Labyrinthe

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

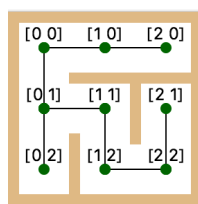


FIGURE 2 – Chemin

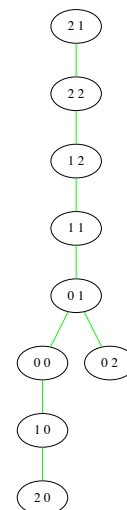


FIGURE 3 – Graphe

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 list of directions
  // directions in {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 $dy = |y_1 - y_2|$.

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 pourrions 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 (appelée 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

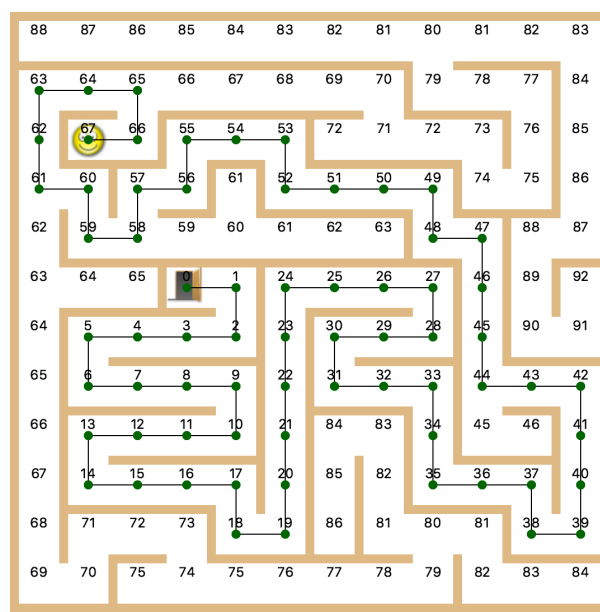


FIGURE 4 – Algorithme de Manhattan

Remarque : cet algorithme peut être aussi utilisé pour trouver les sommets les plus éloignés. Nous en donnons le code Java plus loin en annexe.

²Algorithme 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, \dots, 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, \dots) . 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, \dots, x_k) tel que $\forall i, m_{k+1} \neq x_i$, cela veut dire que le méchant croquera nécessairement le joueur.

2. Partie qui demande un travail algorithmique supplémentaire.

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

javafx.animation.Timeline Pour produire des événements à temps régulier

4 Code fourni et aide

4.1 Labyrinthe parfait

Pour générer un labyrinthe parfait : il s'agit de produire un arbre où les sommets (x, y) représentent toutes les cellules du labyrinthe. Deux sommets sont reliés si A) ils sont voisins B) Une porte communique

Exemple de code :

```
public void buildRandomPath(Vertex vertex){
    // une liste aléatoire des 4 directions
    Vector<Directions> v = new Vector<Directions>();
    for (int i = 0; i < 4 ; ++i)
        v.add(Directions.values()[i]);
    Directions directions [] = new Directions[4];
    for (int i = 0; i < directions.length; ++i) {
        int index=random.nextInt(v.size());
        directions[i]=v.get(index);
        v.remove(index);
    }
    // pour chacune de ces directions , on avance en profondeur d'abord
    for (int i = 0 ; i<4; ++i){
        Directions dir = directions[i];
        if (vertex.inBorders(dir) && graph.doesntExist(vertex , dir)){
            int x = vertex.getX();
            int y = vertex.getY();
            int xt = 0, yt = 0;
            switch (dir){
                case NORTH: xt = x; yt = y-1; break;
                case SOUTH: xt = x; yt = y+1; break;
                case EAST: xt = x+1; yt = y; break;
                case WEST: xt = x-1; yt = y; break;
            }
            Vertex next = new Vertex(xt, yt, vertex.getNbr()+1);
            graph.addVertex(next);
            graph.addEdge(vertex , next);
            buildRandomPath(next);
        }
    }
}
```

4.2 Simplifier le jeu en ouvrant des portes

Pour cela, on ajoute des sauts entre deux sommets voisins du graphe (deux sommets sont voisins si $(|x_1 - x_2| = 0 \wedge |y_1 - y_2| = 1) \vee (|x_1 - x_2| = 1 \wedge |y_1 - y_2| = 0)$)

Exemple de code :

```
public void openDoorRandom() {
    // On essaie 1000 fois , apr'ès quoi on renonce
    for (int i=1 ; i<=1000 ; ++i){
        // On choisi un sommet au hasard
        Vertex vertex = graph.randomVertex();
        if (vertex!=null){
            // On choisi une direction au hasard (on devrait prendre seulement
            // celles qui correspondent 'a des murs...)
        }
    }
}
```

```

        Labyrinth.Directions dir =
            Directions.values()[random.nextInt(Directions.values().length)];
        if (isWall(vertex, dir)){
            Vertex vertex2 = graph.getVertexByDir(vertex, dir);
            if (vertex2!=null){
                Edge edge = graph.getEdge(vertex, vertex2);
                if (edge==null){
                    // On ajoute un saut entre ces sommets
                    graph.addEdge(vertex, vertex2,
                        new Edge(Type.OPENED_DOOR));
                    return;
                }
            }
        }
    }
}

```

4.3 Complicuer le jeu en fermant des portes

Pour cela, on supprime des sauts entre deux sommets voisins, ou plus exactement, on étiquette ces saut par `CLOSED_DOOR`

Exemple de code :

```

package model;

import org.jgrapht.graph.DefaultEdge;

public class Edge extends DefaultEdge implements Comparable<Edge> {

    public enum Type{
        OPENED_DOOR,
        CLOSED_DOOR,
        CORRIDOR;
    };

    private Type type;

    public Edge(Type type){
        super();
        this.type = type;
    }

    // default
    public Edge(){
        super();
        this.type = Type.CORRIDOR;
    }

    public Vertex getSource(){
        return (Vertex) super.getSource();
    }

    public Vertex getTarget(){
        return (Vertex) super.getTarget();
    }
}

```

```

    }

    public Type getType() {
        return type;
    }

    public void setType(Type type) {
        this.type = type;
    }

    @Override
    public int compareTo(Edge o) {
        int source = this.getSource().compareTo(o.getSource());
        if (source!=0)
            return source;
        else {
            return this.getTarget().compareTo(o.getTarget());
        }
    }
}

```

Exemple de code pour fermer une porte :

```

public void closeDoor(Edge edge){
    edge.setType(Edge.Type.CLOSED_DOOR);
}

public void closeDoorRandom(){
    Edge edge = graph.randomEdge();
    closeDoor(edge);
}

```

4.4 Quelques prédicats pour détecter une porte ouverte, fermée, un couloir ou un mur...

A partir d'un sommet, en donnant une direction...

```

public boolean isWall(Vertex vertex, Directions dir) {
    Edge edge = graph.getEdge(vertex, dir);
    return (edge==null);
}

public boolean isClosed(Vertex vertex, Directions dir) {
    Edge edge = graph.getEdge(vertex, dir);
    return (edge==null || (edge.getType()==Edge.Type.CLOSED_DOOR));
}

public boolean isOpened(Vertex vertex, Directions dir) {
    Edge edge = graph.getEdge(vertex, dir);
    return ((edge!=null) && (edge.getType()!=Edge.Type.CLOSED_DOOR));
}

public boolean isClosedDoor(Vertex vertex, Directions dir) {
    Edge edge = graph.getEdge(vertex, dir);
}

```



```

    return (edge!=null && edge.getType()==Edge.Type.CLOSED_DOOR);
}

public boolean isOpenedDoor(Vertex vertex, Directions dir) {
    Edge edge = graph.getEdge(vertex, dir);
    return ((edge!=null) && ((edge.getType()==Edge.Type.OPENED_DOOR)));
}

```

4.5 Faire diriger un *Sprite* vers un autre avec l'algorithme de *Manhattan*

Les deux **Sprite** sont identifiés par les sommets qu'ils occupent (respectivement **source** et **target**).

On étiquette l'ensemble des sommets du graphe par un nombre représentant la distance de manhattan entre **source** et **target**.

```

public class Labyrinth {
    ...
    private void calculateManhattanDistance(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.getVertexByDir(actual, dir);
                    if (next.getNbr()==0){
                        next.setNbr(actual.getNbr()+1);
                        if (next!=source)
                            fifo.add(next);
                    }
                }
            }
        }
    }

    public void launchManhattan(Vertex source, Vertex target){
        for (Vertex vertex : graph.vertexSet())
            vertex.setNbr(0);
        calculateManhattanDistance(source, target);
    }
}

```

Ensuite, il suffit de faire avancer d'une case le poursuivant :

```

public class Sprite implements ISprite {
    ...
    public void move(Labyrinth labyrinth) {
        Vertex vertex = this.getVertex(Labyrinth.graph);
        for (Directions dir : Directions.values()) {
            Vertex next = Labyrinth.graph.getVertexByDir(vertex, dir);
            if (Labyrinth.graph.isConnected(vertex, next)
                && (next.getNbr()==vertex.getNbr()-1)){
                this.move(labyrinth, dir);
            }
        }
    }
}

```


5 Dessiner tout ça

Pour faire la partie graphique, nous nous remettons à la bibliothèque `javafx.*`.

5.1 Stage

Le programme principal étend la classe `Application` de `javafx`

Exemple de code pour `main` :

```
import controller.Controller;

import javafx.application.Application;
import javafx.stage.Stage;

public class Labyrinth extends Application {

    public static void main(String [] args) {
        launch();
    }

    @Override
    public void start(Stage stage) {
        Controller.makeInstance();
        Controller.start(stage);
    }

    @Override
    public void stop() {
        System.exit(0);
    }
}
```

5.2 Frame

Le dessin du cadre du jeu (sans les Sprites, ni les cloisons) ressemble à ceci :

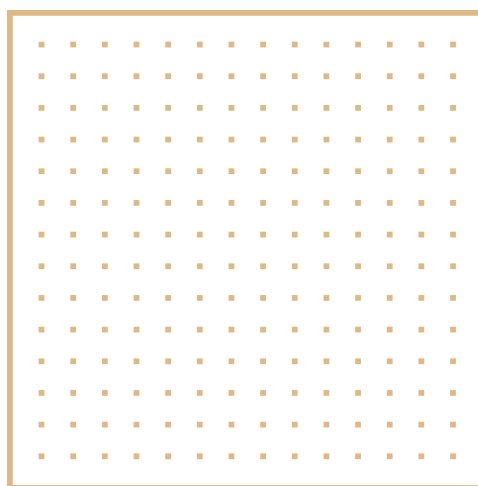


FIGURE 5 – Labyrinthe

exemple de code :

```
public class ViewFrame {

    static final int SPAN = 4; // Pixels for a unit
    static final int WALL = 2; // thickness of the walls (in units)
    static final int CELL = 9; // size of the cells (in units)
    public static final Paint WALLCOLOR = Color.BURLYWOOD;

    ...

    public static void drawFrame(Stage stage, int nbrX, int nbrY){
        scene = new Scene(pane,
            ((WALL + CELL) * nbrX + WALL) * SPAN,
            ((WALL + CELL) * nbrY + WALL) * SPAN);
        scene.setFill(SCENE_COLOR);

        Rectangle square;
        stage.setScene(scene);

        square = new Rectangle(0, 0,
            SPAN * (nbrX * (CELL+WALL) + WALL), WALL * SPAN);
        square.setFill(WALLCOLOR);
        pane.getChildren().add(square);

        square = new Rectangle(0, SPAN * (nbrY * (CELL+WALL)),
            SPAN * (nbrX * (CELL+WALL) + WALL), WALL * SPAN);
        square.setFill(WALLCOLOR);
        pane.getChildren().add(square);

        square = new Rectangle(0, 0,
            WALL * SPAN, SPAN * (nbrY * (CELL+WALL) + WALL));
        square.setFill(WALLCOLOR);
        pane.getChildren().add(square);

        square = new Rectangle(SPAN * (nbrX * (CELL+WALL)), 0,
            WALL * SPAN, SPAN * (nbrY * (CELL + WALL) + WALL));
        square.setFill(WALLCOLOR);
        pane.getChildren().add(square);

        for (int x = 0 ; x < nbrX-1 ; ++x){
            int offsetX = ((WALL+CELL) + (WALL+CELL) * x) * SPAN;
            for (int y = 0 ; y < nbrY-1 ; ++y){
                int offsetY = ((WALL+CELL) + (WALL+CELL) * y) * SPAN;
                square = new Rectangle(offsetX, offsetY,
                    WALL * SPAN, WALL * SPAN);
                square.setFill(WALLCOLOR);
                pane.getChildren().add(square);
            }
        }
    }
}
```

5.3 Dessiner une cloison

exemple de code :

```

public static void drawWall(int xs, int ys, int xt, int yt, Paint color){
    int x = 0, y = 0, xspan = 0, yspan = 0;
    if (ys==yt){
        x = ((WALL+CELL) + (WALL+CELL) * ((int)(xs+xt)/2)) * SPAN;
        y = (WALL + ys * (WALL+CELL)) * SPAN;
        xspan = WALL * SPAN;
        yspan = CELL * SPAN;
        Rectangle square = new Rectangle(x, y, xspan, yspan);
        square.setFill(color);
        pane.getChildren().add(square);
    }
    else if (xs==xt){
        x = (WALL + xs * (WALL+CELL)) * SPAN;
        y = ((WALL+CELL) + (WALL+CELL) * ((int)(ys+yt)/2)) * SPAN;;
        xspan = CELL * SPAN;
        yspan = WALL * SPAN;
        Rectangle square = new Rectangle(x, y, xspan, yspan);
        square.setFill(color);
        pane.getChildren().add(square);
    }
}
}

```

5.4 Dessiner un personnage ou un objet

exemple de code (à adapter en fonction de vos classes MVC) :

```

Image image = new Image( getClass().getResource("FILE.JPG").toExternalForm());
imageView = new ImageView(image);
Frame.pane.getChildren().add(this.imageView);
double xt = (int) ((ViewFrame.WALL + sprite.getX() * (ViewFrame.WALL+ViewFrame.CELL))
* ViewFrame.SPAN);
double yt = (int) ((ViewFrame.WALL + sprite.getY() * (ViewFrame.WALL+ViewFrame.CELL))
* ViewFrame.SPAN);
imageView.setX(xt);
imageView.setY(yt);

```