École nationale
de la statistique
et de l'administration
économique

université
PARIS-SACLAY

Ensae 1re année

Projet informatique – Rapport

Dans le repère de Flamme d'Udùn, ou comment s'échapper d'un labyrinthe hanté.

Cédric Allain

Mercredi 24 mai 2017

Table des matières

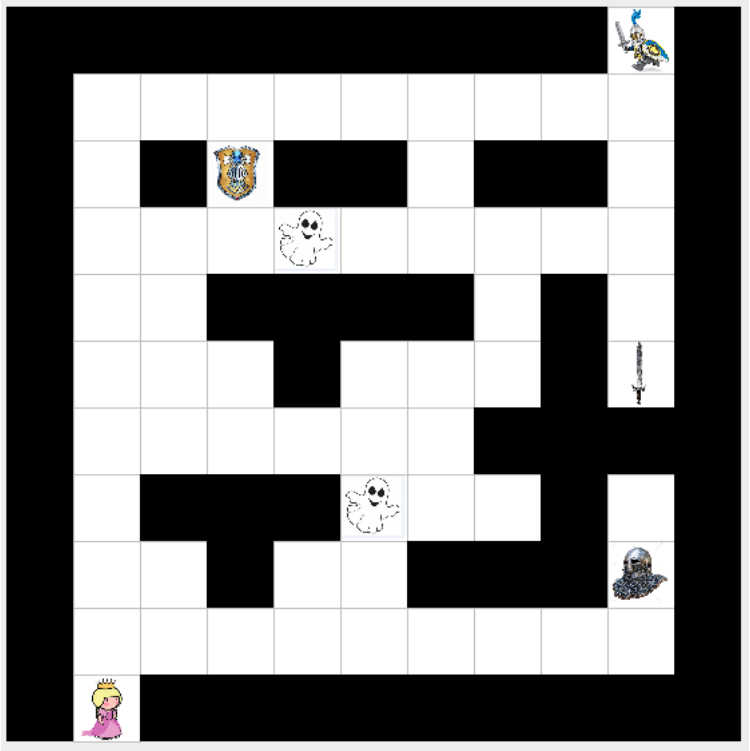

Figure 1 – Le labyrinthe avec les différents éléments.

Introduction

Mon projet consiste en l'élaboration d'un jeu en Java, avec une interface graphique adaptée.

Le but de ce jeu est de déplacer un valeureux chevalier dans les méandres d'un labyrinthe afin de trouver la sortie pour pouvoir libérer la princesse Cunégonde. Malheureusement, une vile sorcière fait garder la princesse par un féroce dragon, connu sous le nom de Flamme d'Udùn (prononcez Flammedoudoune, c'est tout de suite plus rassurant). Pour avoir une chance de vaincre le dragon, le chevalier doit récupérer différentes pièces d'armure éparpillées dans le labyrinthe. En effet, le chevalier étant – légèrement – tête en l'air, il a oublié au château des pièces importantes de son armure, telles que sa côte de mailles, son bouclier et son épée. Heureusement, les bonnes fées de la princesses sont là pour venir en aide au chevalier en déposant les affaires oubliées.

Le chevalier doit donc récupérer l'ensemble des pièces d'armure avant de retrouver la princesse. Cependant, la sorcière, ayant pris quelques précautions, a placé ses deux serviteurs – deux fantômes qui ont la capacité de traverser les murs intérieurs mais pas les murs d'enceinte – en patrouille dans le labyrinthe. Le chevalier doit donc éviter également ces derniers, sous peine de rejoindre Saint Pierre plus tôt que prévu.

Le joueur peut déplacer le chevalier dans les huit directions possibles, désignées comme des points cardinaux (Nord-Ouest, Nord, Nord-Est, Ouest, etc.). Il peut également faire patienter le chevalier sur une case, afin de pouvoir éviter les fantômes plus facilement.

1 Exécuter le programme, ou comment accéder au repère de Flamme d'Udùn

Pour réaliser ce projet, j'ai décidé d'utiliser Eclipse comme environnement de développement (IDE). Ne connaissant que cet IDE pour Java, la notice pour exécuter le programme ne concernera qu'Eclipse.

Tout d'abord, il faut télécharger Eclipse si ce n'est pas déjà le cas. Une fois Eclipse téléchargée, l'installation d'Eclipse est ce qu'il y a de plus classique : suivez simplement les différentes étapes. Une fois l'installation effectuée, lancez simplement l'applicaion. Il vous est alors demandé de choisir un emplacement pour le *workspace*. Ce dernier est simplement un dossier où se trouveront l'ensemble des projets Java.

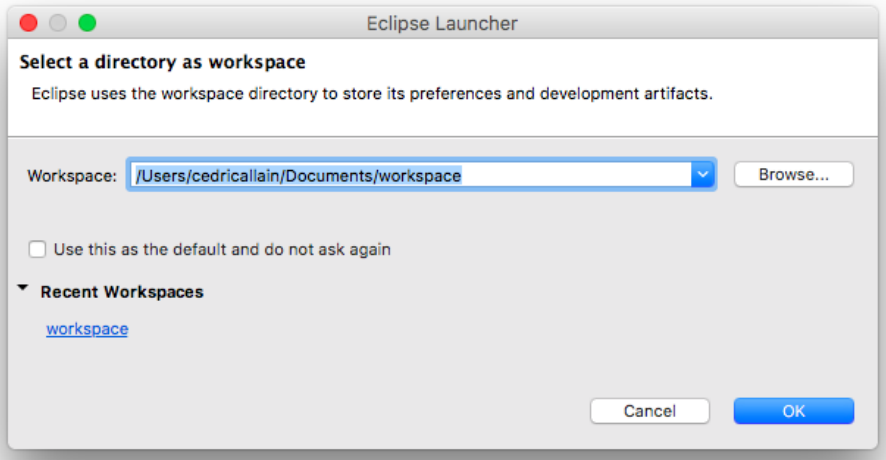

FIGURE 2 – Choisir l'emplacement du *workspace* lors du lancement d'Eclipse.

Une fois cet emplacement choisi, cliquez sur OK. Maintenant, il faut décompresser le dossier `Projet_informatique_1A_S2_Cedric_Allain.zip`, puis copier le dossier `Projet_1A_S2_v1` dans le dossier `workspace`.

Pour importer ce projet dans Eclipse, il faut retourner sur Eclipse puis aller dans `File -> Import`. Ensuite, selectionnez `General -> Existing Projects into Workspace`, cliquez sur Next, puis, à l'aide du bouton `Browse`, parcourez votre *workspace* puis sélectionnez le dossier `Projet_1A_S2_v1`, qui contient touts les éléments nécessaires à l'exécution du jeu. Finalement, cliquez sur `Finish` pour terminer l'importation du projet.

Pour lancer le projet, il suffit de double-cliquer sur la classe `Main.java`, qui se trouve dans la barre latérale `Package Explorer`, comme sur la figure 4. Finalement, la dernière étape consiste à lancer cette classe `Main.java`, en faisant `Run -> Run As -> 1 Java Applicaiton`. Le programme est maintenant exécuté, à vous de jouer !

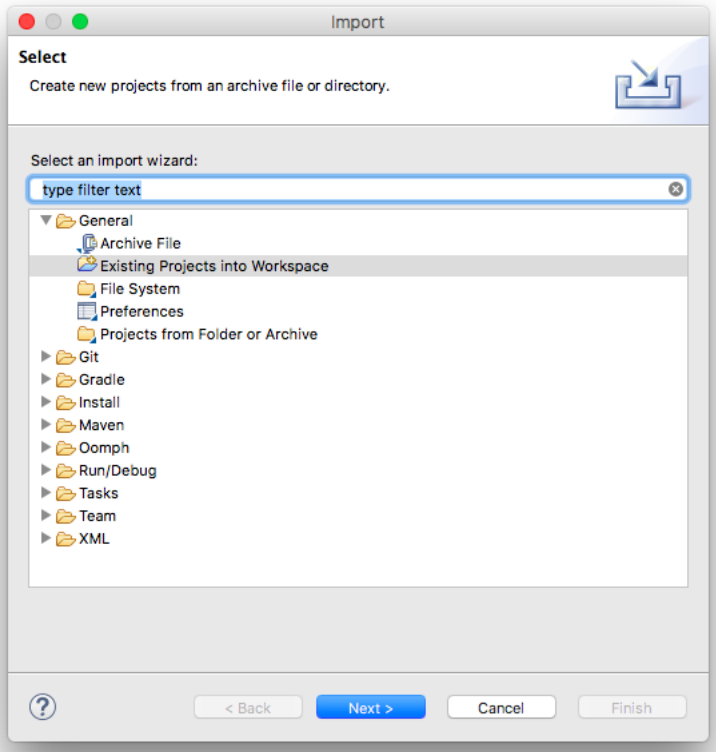

FIGURE 3 – Importer un projet dans Eclipse.

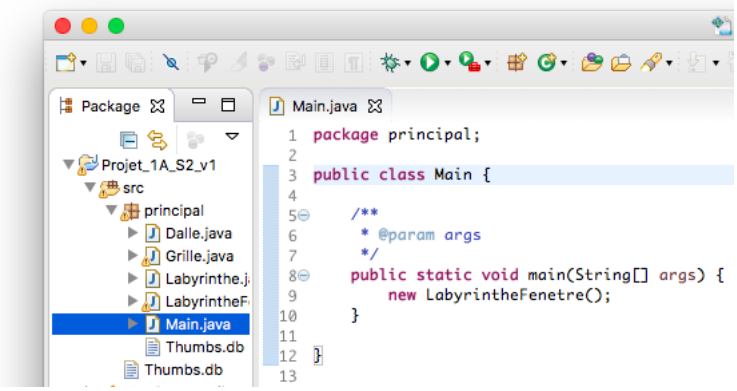

FIGURE 4 – Lancer le programme du jeu.

2 Déroulement du jeu

Lorsque le programme est exécuté, une boîte de dialogue s'ouvre et explique au joueur le contexte de l'histoire ainsi que le but et les règles du jeu. Ensuite, une seconde boîte de dialogue s'ouvre et invite le joueur à choisir un labyrinthe. En effet, il existe deux configurations possibles du labyrinthe et le joueur choisi entre ces deux versions à l'aide d'un menu déroulant, en haut à droite de la fenêtre de jeu.

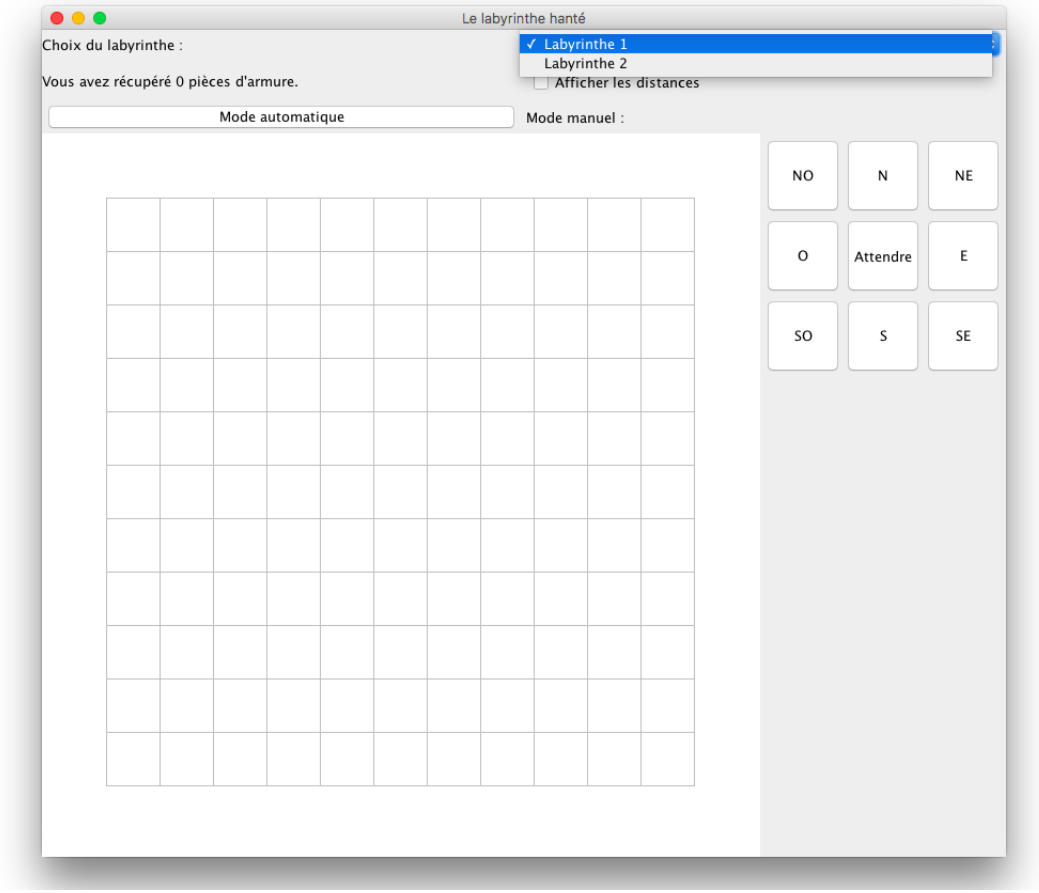

FIGURE 5 – Le joueur doit choisir une version du labyrinthe avant de pouvoir jouer.

Le labyrinthe sélectionné se charge alors dans la fenêtre graphique. Pour chaque version du labyrinthe, la configuration des murs à l'intérieur ainsi que la position de du chevalier et de la princesse sont déterminées. En revanche, les différents éléments (fantômes et pièces d'armures) sont placés aléatoirement dans le labyrinthe, suivant quelques règles. En effet, les pièces d'armures ne peuvent pas se placer sur un mur ou sur une autre pièce d'armure. De la même façon, les fantômes ne se placent pas sur un mur (même s'ils ont la capacité de traverser les murs intérieurs), de façon à ce que le joueur puisse les voir dès le premier tour. De plus, pour faciliter le début du jeu, les fantômes ne peuvent se placer à moins de quatre cases de la position initiale du chevalier. L'interface de jeu auquel fait face l'utilisateur correspond à la figure 6.

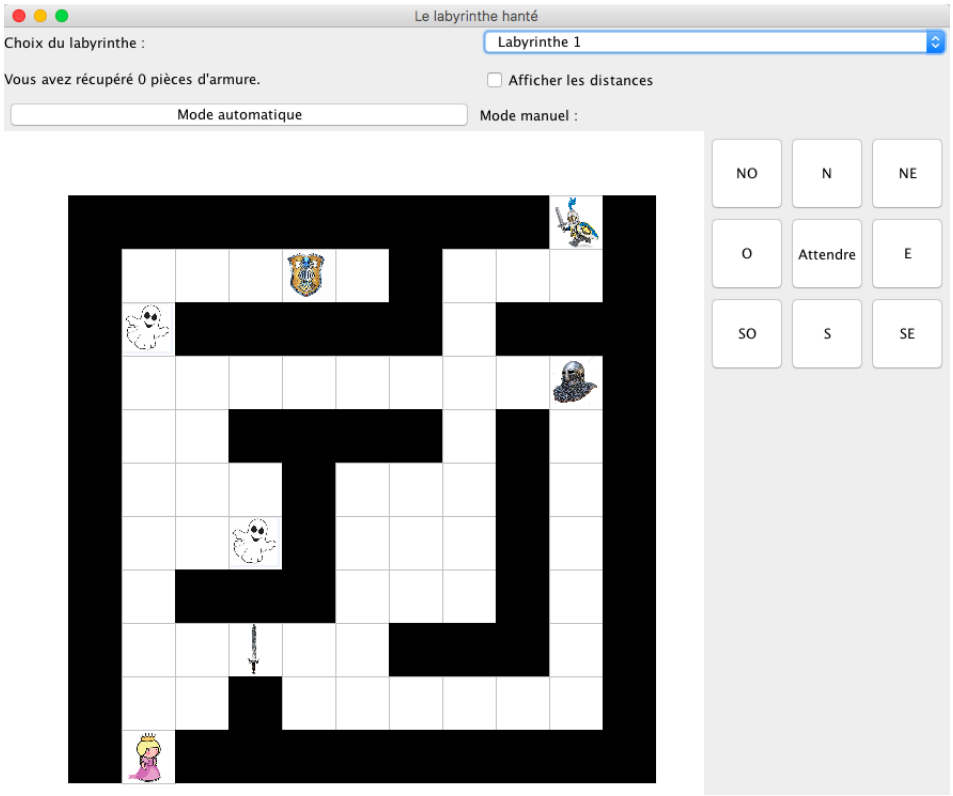

FIGURE 6 – L'interface de jeu, initialisé avec la labyrinthe 1. Les fantômes et les pièces d'armure sont disposés aléatoirement.

Une remarque s'impose concernant l'initialisation des éléments du labyrinthe (pièces d'armure et fantômes). Si, en initialisant le labyrinthe, le joueur n'apprécie pas la façon dont sont disposés les éléments – cela peut être le cas si les pièces d'armures sont très proches les unes des autres par exemple –, il est tout a fait possible d'initialiser de nouveau le labyrinthe en le sélectionnant de nouveau dans la liste déroulante. Les différents éléments changeront alors de place, tandis que la configuration interne (les murs intérieurs, la position du chevalier et de la princesse) resteront les mêmes, ces derniers étant intrinsèques à la version du labyrinthe choisie.

L'initialisation des pièces d'armures et des fantômes se fait respectivement à l'aide des méthodes `initArmure()` et `initFantome()`.

```
public int[] initFantome(){
  int x = -1;
  int y = -1;
  int[] pos = {x, y};
  boolean test = false;
  // On ne s'arrête que lorsqu'on a trouvé une case qui peut accueillir un fantôme.
  while(!test){
    test = true;

    // On tire au hasard une case dans le labyrinthe
    x = (int) (1 + Math.random() * (dim - 2));
    y = (int) (1 + Math.random() * (dim - 2));

    // On doit vérifier si cettte case ne correspond pas à un mur ou a un autre fantôme.
    Dalle dalleTest = dalles[x][y];
    // Le fantôme ne peut pas être à moins de 3 cases du chevalier lors de l'initialisation.
    calculDistance(posChevalier);
    if(dalleTest.isMur() || dalleTest.isFantome() || dalleTest.getDistance() < 3)test = false;
  }

  pos[0] = x;
  pos[1] = y;
  dalles[x][y].setFantome(true);
  listeFantomes[nbFantomesPosees] = pos;
  nbFantomesPosees++;

  return pos;
}
```

Listing 1 – La méthode `initFantome()` qui renvoie une case libre où placer un fantôme.

L'utilisateur peut maintenant déplacer le chevalier dans le labyrinthe à l'aide des boutons de direction. Les directions sont définies par les points cardinaux qu'elles représentent ; pour descendre d'une case, l'utilisateur utilisera donc le bouton S (Sud). J'ai fais ici l'hypothèse que

le chevalier pouvait se déplacer en diagonale. De plus, si le joueur veut « passer son tour », c'est-à-dire s'il veut rester sur la même case mais que les fantômes se déplacent – cette situation peut arriver dans le cas où le joueur se trouverait coincé par un fantôme et voudrait alors attendre que ce dernier s'en aille –, le joueur peut utiliser le bouton `Attendre`, prévu à cet effet.

Le jeu se termine dès lors que le joueur a récolté l'ensemble des pièces d'armure puis rejoint la princesse. Le joueur est alors invité à joueur une nouvelle partie.

À tout moment, le joueur peut changer de labyrinthe (ou initialiser de nouveau celui avec lequel il joue). Dans ce cas, le joueur revient au début du labyrinthe, ses progrès sont perdus et tous les éléments sont initialisés de nouveau. De plus, si au bout d'un moment le joueur ne veux plus jouer mais que l'idée de laisser une princesse aux main d'un dragon ne l'enchante guère, il a la possibilité d'activer le mode automatique, simplement en pressant le bouton éponyme. Le chevalier se déplacera alors de lui même vers la princesse, ramassant au passage les différentes pièces d'armures qu'il lui manque.

3 Le mode automatique, ou la générosité des bonnes fées

Comme évoqué précédemment, le joueur peut activer un mode automatique. En effet, le chevalier – tête en l'air ne l'oublions pas – peut avoir du mal à s'orienter dans le dédale. Heureusement, il peut compter sur l'aide des bonnes fées de la princesse ! Ces dernières, lorsque sollicitées, peuvent guider le chevalier à travers le labyrinthe afin qu'il puisse récupérer son armure et aller délivrer la princesse.

Ce mode, lorsqu'il est activé, fait appel à la méthode `goAuto`, qui fait elle-même appel à deux autres méthodes, `nextArmure()` et `avanceChevalierAuto()`. Le principe est le suivant. Tant que le chevalier ne possède pas la totalité des pièces d'armure, il se déplace en direction de la pièce d'armure qui lui manque la plus proche de lui, la position de cette dernière est renvoyée par la méthode `nextArmure()`. Bien entendu, si le joueur active le mode automatique une fois qu'il a récolté l'ensemble des pièces d'armure, on passe directement à la seconde partie de la méthode `goAuto()`, à savoir déplacer le chevalier jusqu'à la princesse pour terminer la partie.

La méthode `avanceChevalierAuto()` quant à elle, permet de déplacer le chevalier d'une case en direction d'un objectif déterminé, bien entendu en évitant les murs et les fantômes. Ici, les différents objectifs sont donc, dans le cadre de la méthode `goAuto`, les pièces d'armures restantes, à chaque fois celle qui est la plus proche de la position du chevalier, puis la princesse, une fois l'armure constituée.

La distance d'une case à un objectif, à une « case cible » en quelque sorte, est calculée à l'aide de la méthode `calculDistance()`. Cette méthode est probablement la méthode qui a demandé le plus de réflexion. Il fallait en effet trouver dans un premier temps un moyen de calculer cette distance, en prenant en compte les murs et les éventuels fantômes. Il fallait trouver ensuite une façon de stocker l'information qui doit être accessible facilement.

Avec la façon dont j'ai choisi d'implémenter mon code, la seconde partie du problème a été résolue assez facilement. En effet, j'ai créé une classe Labyrinthe, dont l'élément principal est une tableau en deux dimensions, composé de `Dalle`(s). La classe `Dalle` est une classe d'objet représentant une dalle du labyrinthe. Toutes les informations relatives à une dalle – est-ce que la dalle correspond à un mur ? est-ce qu'une pièces d'armure est déposée sur cette dalle ? est-ce que la dalle accueille un fantôme ? etc. – sont passées en variables de cette classe.

```
1 // Cette classe défini l'élément de base d'un labyrinthe : une dalle.
2 public class Dalle {
3
4   // ------ Variables ------
5   private boolean mur = false, armure = false, fantome = false, visite =
      false, odeur = false;
6   private int distance = -1;
7
8   [...]
9 }
```

Listing 2 – Les variables de la classe `Dalle` et leur valeur initiale.

Lors de l'initialisation, toutes les dalles du labyrinthe ont donc une « distance » égale à -1. Cette valeur par défaut est remise à toutes les cases à chaque fois que l'on appelle la méthode `calculDistance()`, à l'aide de la méthode `resetDistances()`. Le principe de la méthode `calculDistance()` est le suivant. La distance de la case que l'on désigne comme étant la « case objective » (la position de cette dernière est désignée par `pos`, un vecteur qui contient les coordonnées x et y de la case) est initialisée à 0. Ensuite, l'algorithme parcours l'ensemble des dalles du labyrinthe et attribut une distance de 1 aux dalles dans le voisinage de la « case objective », si bien évidemment ce ne sont ni des murs, ni des fantômes.

De le même façon, pour chaque dalle du labyrinthe qui ne soit ni un mur ni un fantôme, la distance des dalles du voisinage est augmentée de 1, sauf si la distance sur ces dalles est inférieure ou égale à la distance de la case sur laquelle on est, car cela signifie que les dalles du voisinage sont au moins aussi proche de l'objectif que la case sur laquelle on est. Bien entendu, le cas où la distance sur les cases du voisinage est égale à -1 est exclu de la condition précédente. En effet, une distance de -1 ne signifie pas que la case est proche de l'objectif, mais que la distance n'a jamais changé. Finalement, l'algorithme s'arrête lorsque qu'aucun changement n'est effectué.

```
1  public void calculDistance(int[] pos){
2    resetDistances();
3    dalles[pos[0]][pos[1]].setDistance(0);
4    boolean fini = false;
5    while(!fini){
6      fini = true;
7      for (int i = 0; i < dim; i++) {
8        for (int j = 0; j < dim; j++) {
9          if(!dalles[i][j].isMur() && dalles[i][j].getDistance() != -1 && !
       dalles[i][j].isFantome()){
10           for (int k = -1; k < 2; k++) {
11             for (int l = -1; l < 2; l++) {
12               int[] posTest = {i+k, j+l};
13               if(estDedans(posTest) && !dalles[i+k][j+l].isMur() && !dalles
       [i+k][j+l].isFantome()){
14                 if(dalles[i+k][j+l].getDistance() > dalles[i][j].
       getDistance() + 1 || dalles[i+k][j+l].getDistance() == -1){
15                   dalles[i+k][j+l].setDistance(dalles[i][j].getDistance() +
        1);
16                   fini = false;
17                 }
18               }
19             }
20           }
21         }
22       }
23     }
24   }
25 }
```

Listing 3 – L'algorithme de calcul des distance à une case cible.

On peut conclure cette section en évoquant le coût de la méthode `goAuto()`. Cette méthode est à coût variable. En effet, le nombre de boucles effectuées lors de l'appel de la méthode varie en fonction de l'avancement du joueur dans la résolution du labyrinthe. Si le joueur lance le mode automatique après avoir ramassé l'ensemble des pièces d'armure, le coût de `goAuto()` sera beaucoup moins important comparé à la situation où le mode automatique est lancé dès le tout début de la partie.

Une remarque toutefois : si l'utilisateur active le mode automatique dans l'état actuel du projet, il ne verra pas le chevalier se déplacer tout seul. En effet, il ne verra que l'état final, c'est-à-dire la situation où le chevalier a récupéré l'ensemble des pièces d'armures et a rejoint la princesse. N'y voyait pas là une entourloupe de ma part, l'algorithme fonctionne vraiment et le chevalier se déplace bien de lui-même à travers le dédale, son parcours n'est simplement pas affiché à l'écran. Malgré plusieurs tests et recherches, je n'arrive pas à résoudre ce problème d'affichage.

Il existe cependant un moyen de voir la méthode `goAuto()` en action. Pour cela il faut modifier quelque peu la méthode `LabyrintheFenetre()`. Au début de cette fonction, il faut mettre en commentaire la commande `laby.init0()` et activer les commandes `laby.setNumLaby(2)` et `laby.reset()`, ainsi que la commande `goAuto()` à la fin de la même méthode.

```
1  public LabyrintheFenetre(){
2    super("LabyrintheFenetre");
3
4    // Initialisation du labyrinthe
5    //laby.init0();
6
7    // Pour voir le chevalier bouger tour à tour dans le labyrinthe,
       initialisez avec un labyrinthe (1 ou 2 en param tre), puis à la fin de
       cette méthode, appelez goAuto()
8    laby.setNumLaby(2);
9    laby.reset();
10
11   [...]
12
13   goAuto();
14 }
```

Listing 4 – La méthode `LabyrintheFenetre()` modifiée de façon à voir `goAuto()` en action

En résumé, ce changement permet de d'initialiser directement une version du labyrinthe (dans notre exemple, c'est la seconde version du labyrinthe qui est initialisée), puis de lancer directement la méthode `goAuto()`. L'utilisateur verra alors le chevalier se déplacer tout seul, jusqu'à finir la partie. Les messages de fin s'affichent alors, et le joueur est invité à jouer une autre partie. Une fois cette première partie – automatique – finie, on se retrouve dans la situation « normale », décrite dans la section précédente.

4 Structure du code

Le projet est constitué de 5 classes, chacune ayant un rôle particulier. La classe `Dalle` a déjà été présentée précédemment.

4.1 La classe `Labyrinthe`

La classe `Labyrinthe` sert à définir un `labyrinthe` comme étant un tableau à deux dimensions où chaque case du tableau est une `Dalle`. Cette classe regroupe l'ensemble des variables et méthodes liées à l'objet `labyrinthe`.

Les variables présentes dans cette classe sont entre autres `afficheDistances`, un booléen qui indique si on doit afficher ou non les distances des dalles constituant le labyrinthe, `listeArmures` et `listeFantomes`, des tableaux à deux dimensions d'entiers qui stockent respectivement les positions – une coordonnée en x et une coordonnée en y – des différentes pièces d'armures et des fantômes.

Parmi les méthodes propres à la classe `Labyrinthe`, on retrouve les méthodes `calculDistance()` et `avanceChevalierAuto()`. On retrouve aussi des méthodes qui ne sont utilisées qu'en tant que « sous-méthodes » dans des méthodes plus importantes. Par exemple, il y a la méthode `posEgales()`, qui prend en paramètre deux positions – une position est un couple d'entiers qui représente les deux coordonnées d'un élément dans le labyrinthe – et retourne un booléen : vrai si les deux positions sont égales (i.e. les deux positions correspondent à la même dalle du labyrinthe) et faux sinon.

4.2 La classe `Grille`

Cette classe sert exclusivement à la représentation graphique d'un labyrinthe, ce dernier étant passé en paramètre lors de l'appel de cette classe. Comprendre le fonctionnement d'une telle classe m'a demandé du temps, car c'était la première fois que je devais utiliser le *package* `java.awt.Graphics` et l'objet `Graphics`.

C'est dans cette classe que les paramètres qui vont servir à la représentation du labyrinthe sont stockés. Par exemple, on y trouve la variable `TAILLE_CASE`, un entier qui stocke la dimension d'une case. Si un jour quelqu'un veut reprendre ce projet et ajouter un « zoom » qui servirait à agrandir/réduire la taille du labyrinthe, c'est sur cette variable `TAILLE_CASE` qu'il faudra agir.

C'est également dans cette classe que l'on décide de représenter le chevalier, la princesse, les fantômes et les pièces d'armures par les images de notre choix, comme on peut le voir dans la figure 1. Les différentes images se trouvent à la racine du projet, c'est à dire dans le dossier `Projet_1A_S2_v1`. Si l'utilisateur décide de changer le visuel des éléments, ce sont ces images qu'il se doit de changer.

4.3 La classe `LabyrintheFenetre`

Cette classe est assez complexe. C'est dans celle-ci que tout se déroule. En effet, l'appel de cette classe est l'unique action effectuée par la classe `Main`. Cette dernière est l'unique classe du projet qui est exécutée lorsque l'utilisateur lance le jeu.

```
package principal;

public class Main {

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

}
```

Listing 5 – La classe `Main` qui ne sert qu'à appeler la classe `LabyrintheFenetre`.

C'est dans la classe `LabyrintheFenetre` qu'est définie la fenêtre graphique que le joueur voit apparaître lors du lancement du jeu. Les différents boutons, l'agencement des éléments constituant la fenêtre, les différents messages qui apparaissent, les actions qui s'effectuent lorsqu'on clique sur les boutons, etc., tout cela est codé dans cette classe.

La méthode de la classe qui permet de faire le lien entre une action faite sur un bouton (clic, choix dans un menu déroulant, etc.) et l'algorithme qui est lancé suite à cette action est la méthode `actionPerformed()`.

```
public void actionPerformed(ActionEvent arg0) {
    if(arg0.getSource() == numLabyCB){
      laby.setNumLaby(numLabyCB.getSelectedIndex() + 1);
      laby.reset();
    }

    [...]
}
```

Listing 6 – Extrait de la classe `actionPerformed`.

L'extrait de la classe `actionPerformed` présenté ci-dessus montre ce qu'il se passe lorsqu'on choisit une version du labyrinthe dans la liste déroulante `numLabyCB`, cette dernière étant initialisée dans la méthode `LabyrintheFenetre()`. On voit que lorsque l'utilisateur choisi une version du labyrinthe, la variable `numLaby` du labyrinthe que l'on considère (ici `laby`) est mise à jour. Ensuite, la méthode `reset()` de la classe `Labyrinthe` est appelée. Cette dernière permet d'initialiser la version du labyrinthe choisie.

C'est également dans cette classe qu'est défini la méthode `goAuto()`.

5 Quelques prolongements possibles, ou comment écrire la suite de l'histoire

Dans cette dernière partie, je propose des prolongements qui pourraient être intéressants, dans l'objectif de rendre le jeu plus attrayant pour le joueur. Par soucis de temps et de complexité, il m'a été impossible d'implémenter ces améliorations.

5.1 Initialisation et création du labyrinthe

Comme évoqué précédemment, lorsque le joueur lance une partie, il doit choisir une version du labyrinthe. Bien que la position des pièces d'armures et des fantômes soit aléatoire, la disposition des murs à l'intérieur du labyrinthe est propre à chaque labyrinthe. Une première amélioration du jeu serait de réussir à initialiser le labyrinthe de façon complètement aléatoire

Par exemple, il est possible d'imaginer que, lors du lancement du jeu, l'utilisateur puisse choisir la dimension du labyrinthe – soit dans une liste de taille (petit, moyen, grand, géant), ou directement en rentrant les dimensions –, puis que la disposition des murs se fasse de façon aléatoire. Lors de l'initialisation, il faut cependant faire attention à respecter quelques règles. Par exemple, il faut que sur les bords il y ait bien des murs, qu'il y ait bien une entrée et une sortie et surtout, qu'il existe bien un chemin entre l'entrée et la sortie.

Une deuxième amélioration que l'on peut proposer est celle d'un outil de création de labyrinthe. L'utilisateur pourrait créer lui-même le labyrinthe dans lequel il souhaite évoluer en « construisant » les murs, puis en plaçant les différents éléments, soit manuellement, soit en choisissant une option qui les place de façon aléatoire. Le labyrinthe ainsi créé pourra être sauvegardé afin que l'utilisateur puisse le retrouver lors de ses prochaines parties.

5.2 État d'esprit des fantômes

Dans la version actuelle du jeu, le nombre de fantôme est fixé à deux, et leurs déplacements sont prévisibles ; un fantôme se déplace sur sa colonne et le second se déplace sur sa ligne. Une amélioration que l'on pourrait apporter serait tout d'abord de pouvoir choisir le nombre de fantômes dans le labyrinthe.

Une autre amélioration intéressante est de pouvoir modifier « l'état d'esprit des fantômes ». Par cela, j'entends leur mode et leur logique de déplacement. On pourrait alors différencier 5 états d'esprit, que l'on peut classer en cinq niveaux de difficulté croissante, comme suit :

- pas de fantômes du tout ;
- les fantômes sont immobiles ;
- les fantômes se déplacent de façon prévisible, sur le modèle de la version actuelle ;
- les fantômes se déplacent de façon aléatoire, dans les 8 directions possibles et peuvent également attendre sur leur case, à l'instar du chevalier ;
- les fantômes se déplacent de façon à se diriger en direction du chevalier.

L'utilisateur pourra choisir entre ces différents niveaux de difficultés à l'aide d'un menu déroulant, de la même manière qu'il choisit la version du labyrinthe.

5.3 Contrôler le chevalier à l'aide du clavier

Le fait de devoir déplacer le chevalier à l'aide bouton virtuels est loin de l'aspect intuitif que les utilisateurs pourraient avoir pour ce type de jeu, à savoir déplacer le chevalier à l'aide des flèches du clavier. Faire une telle amélioration rendrait l'expérience de jeu bien meilleure. Cependant, utiliser ce mode de déplacement – seulement quatre directions possibles – implique de coder à nouveau toutes les fonctions liées au déplacement et au calcul des distances. En effet, dans la version actuelle, pour une case centrale, une case ayant un côté adjacent et une case n'ayant qu'un seul coin en commun avec la case centrale sont toutes deux considérées comme étant à la même distance de la case centrale.

Mise à part cette phase de recodage, lier les flèches du clavier à un déplacement du chevalier ne me semble pas très compliqué.

Annexes

A Tables des figures et des algorithmes (listings)

Table des figures

Listings