Sciences informatiques - Bachelier 1

Rapport de projet

HUYLENBROECK Florent
VERHIEST Simon

Titulaire: H. MELOT

UMONS – Année académique 2016-2017

Contents

Introduction

Dans le cadre de notre projet de programmation et algorithmique du deuxième quadrimestre de bachelier 1 en sciences informatiques, nous avons dû réaliser une application utilisant le langage JAVA. Le but de celle-ci étant de permettre à un utilisateur de jouer au jeu Sokoban grâce à une interface graphique, et de pouvoir importer et générer ses propres niveaux.

La difficulté principale de ce projet était selon nous l'implémentation d'un générateur de niveaux qui soient tous jouables (Un niveau *jouable* est un niveau qui est complètement réalisable).

Dans ce rapport, nous allons traiter du fonctionnement de notre binôme, de la conception de notre application et de son fonctionnement.

Le groupe

Notre groupe de projet était composé de deux personnes. L'association n'a pas été difficile car nous sommes cousins et vivons sous le même toit. Les deux membres de ce groupe sont Florent HUYLENBROECK(151203) et Simon VERHIEST(171679).

2.1 Répartition des tâches

Dans ce paragraphe uniquement, Florent sera la première personne.

Étant un élève redoublant, j'ai déjà du remettre un projet d'informatique en 2016. Ainsi, la répartition des tâches au sein de notre groupe s'est effectuée assez rapidement.

Dès l'annonce thème de notre projet, je me suis mis à travailler la partie logique de celui-ci, tandis que mon binôme s'apprêtait à en réaliser la partie graphique. Ainsi, une fois la base de la partie logique terminée (création du plateau, déplacement et vérification de fin de partie) nous avons commencé la partie visuelle du projet ensemble, car l'avancement du cours ne permettait pas à mon binôme de se lancer tout seul là-dedans. Nous avons donc commencé l'interface graphique de notre application à deux, de manière à apprendre les bases du package swing. L'interface graphique ainsi commencée, je me suis dirigé vers un élément supplémentaire au projet qui me tenait à cœur : un algorithme qui détecte les *deadlocks* (les états de jeu qui mènent à un blocage d'une ou plusieurs caisses). L'écriture de celui-ci était algorithmiquement intéressante et m'a tenu occupée jusqu'à ce que notre interface graphique soit terminée.

La dernière partie de ce projet fut la mise en place du générateur. Mis à part le facteur temps, celle-ci ne nous a pas posé problème. Nous avions pris le temps d'y réfléchir tout au long de l'écriture autres parties, et particulièrement lors de celle de l'algorithme de détection de deadlocks. Le fonctionnement du générateur est détaillé plus précisément à la section.

2.2 Les apports du projet

Selon nous, le projet est un cours intéressant de la première année de bachelier. Voici deux aspects positifs et un négatif que nous souhaitons soulever :

Apports positifs

Le déroulement du projet est intéressant de par le fait de devoir créer une application permettant de jouer à un jeu simpliste, en parallèle avec le cours d'algorithmique, ce qui permet d'approfondir notre apprentissage du langage JAVA

C'est aussi notre première réelle expérience en tant que programmeur. Nous avons un objectif, des consignes, et un délai à respecter. Cela nous apprend donc à nous organiser, à travailler en groupe et à communiquer.

Apports négatifs

Un point faible du projet est l'investissement que celui-ci demande. Même en y travaillant raisonnablement tous les jours depuis l'annonce de celui-ci, nous avons quand même dû y dédier plus d'heures que prévu ces deux dernières semaines, car nous allions manquer de temps. Le projet est un cours a 5 crédits, il équivaut donc à 150 heures de travail selon le système de crédits ECTS. Le problème ne réside pas dans le nombre d'heure mais dans le fait que la grande majorité de celles-ci soit du travail à domicile et non des cours. Mais nous sommes tout de même conscient qu'il s'agit d'un défi de la première année de bachelier, qu'il a été intéressant de relever

L'application

3.1 Nos choix de conception

Le pattern MVC

Cette manière de travailler est venue assez naturellement. Lors des premiers jours, nous nous sommes mis d'accord sur le fonctionnement de l'application, qui est de séparer la logique de l'interface graphique. Après renseignement nous avons découvert le type de design pattern appelé *MVC* (Modèle - Vue - Contrôleur) qui consiste à faire interagir entre elles 3 parties distinctes. Dans un premier temps, ce que nous appelons la logique nous a servi de "Modèle", ensuite notre interface graphique a fait office de "Vue" et pour finir, certains éléments de celle-ci ont servis de "Contrôleurs". Ces deux dernières parties ne sont pas exactement distinctes étant donné que certains contrôleurs sont en fait des classes internes à d'autre classes faisant parties de notre interface graphique. Nous jugeons cependant avoir respecté le design pattern MVC.

Utilisation de classes énumérées

Lors de la création de notre premier objet, l'objet plateau, l'idée de créer un objet correspondant à chaque type de case nous semblait un peu "lourde". Ainsi, en créant une classe énumérée, un même objet statique pouvait être réutilisé plusieurs fois et la classe pouvait en même temps servir à charger les images correspondant aux cellules du plateau. Ainsi, moins d'instanciation d'objets et moins de chargement d'images étaient nécessaires.

Utilisation de Swing de JAVA

Nous avons choisi d'utiliser Swing de JAVA au lieu de JavaFX pour 3 raisons. Premièrement la simplicité. Sachant que le bagage nécessaire à la partie logique du projet était déjà conséquent, nous ne souhaitions pas avoir à utiliser des méthodes trop compliquées pour notre interface graphique. Deuxièmement, car comme stipulé plus haut, un des membres du groupe avait déjà remis un projet, et avait utilisé Swing pour l'interface graphique de celui-ci. Et dernièrement, car au moment de commencer notre interface, la séance d'information dédiée à celle-ci n'avait pas encore été donnée. Nous étions en avance.

Fonctionnement du générateur

Notre générateur fonctionne en 3 étapes, si l'une d'entre elle ne rentre pas dans les critères de génération, le processus reprend du début.

La première étape consiste à créer un plateau vide (ne contenant que des cases sol (FLOOR) et des cases mur (WALL)). Pour ce faire, des *templates* (Figures représentant des sections de 3x3

contenant des cases FLOOR et WALL) sont utilisés. Cette méthode est inspirée du travail de Joshua Taylor et Ian Parberry[1]. Ensuite, le niveau créé est parcouru afin de vérifier que l'aire du sol est continue.

L'étape suivante est le placement du joueur et des goals. Pour cette étape, un numéro aléatoire est généré, correspondant à l'un des 4 coins du plateau. Le joueur est placé dans le coin opposé tandis que les goals sont placés sur les cases les plus proches possible du coin, mais atteignable par le joueur.

La dernière étape consiste à tirer les caisses depuis chaque goal, de stocker leur position dans une liste, et de placer un mur a la place de celle-ci dans un premier temps, pour éviter tout conflit.

Choix d'éléments supplémentaires

En ce qui concerne les éléments supplémentaires au projet (Ce qui ne figure pas dans la liste des éléments indispensables pour que le projet soit corrigé), nous avons choisi de n'ajouter qu'une seule fonctionnalité, mais qui demandait une certaine recherche algorithmique. Il s'agit de la recherche de deadlocks. Les fonctionnalités telles que le compteur de tour, le chronomètre, tirer une caisse, etc.. nous semblaient plutôt "gadget" qu' intéressante dans le cadre de notre apprentissage.

3.2 Les points forts de notre application

Le plus gros point fort de notre application est selon nous sa simplicité et sa légèreté. En effet, nous avons effectué beaucoup de petites optimisations de manière à ce que notre application tourne le plus vite possible. Par exemple, notre plateau contient une liste des cases ayant subi une modification, et seules celles-ci sont mise à jour par notre interface graphique. Mais aussi le fait que nous utilisions des classes énumérées (voir section précédente) à la place d'instancier un objet par case du plateau. Ou encore le fait de lancer nos algorithmes de fin de partie uniquement lorsqu'une caisse a été déplacée. Ce sont beaucoup de petits détails, mais cela contribue beaucoup à la rapidité d'exécution de l'application. Et pour ce qui est de la simplicité, nous avons opté pour des menus clairs, contenant des options simples qui rendent notre Sokoban très intuitif à utiliser.

3.3 Les points faibles de notre application

Un aspect du projet que nous n'avons pas énormément travaillé est l'aspect graphique. Nous avons choisi deux couleurs qui allaient bien ensemble pour les menus, gardé les boutons basiques des Swing, ainsi que le FileChooser. Nous avons préféré consacrer du temps à la création de belles méthodes qu'à dessiner de beaux graphismes.

[1] Joshua Taylor and Ian Parberry, Procedural Generation of Sokoban Levels, larc.unt.edu/techreports/LARC-2011-01.pdf, 2011, consulté en Avril 2017

3.4 Bugs connus

Un bug connu est que parfois, la génération aléatoire d'un niveau de difficulté "élite" va engendrer une boucle infinie, malgré la limite d'itération intégrée au cette partie du programme (en décrémentant un compteur). Ainsi, cette méthode de génération prendra parfois plusieurs minutes, et il sera nécessaire de redémarrer l'application complètement. Cette erreur ne devrait cependant pas se produire lors de la génération de niveaux d'autres difficultés.

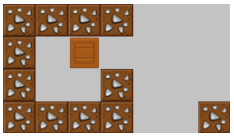

Figure 3.1: Exemple de situation non-jouable générée

Un autre bug connu lors de la génération de niveau, est que de, malgré la vérification des deadlocks, deux caisses se génèrent de manière a ne pas pouvoir être déplacée dans bloquer l'une des deux.

Le dernier bug connu survient aussi lors de la génération de niveau.
Il s'agit d'une *ArrayOutOfBoundException* (Erreur lorsqu'on parcours un tableau hors des limites qu'on lui a définies à son instanciation) générée par l'algorithme de recherche de chemin. Malgré nos essais pour la capturer, nous ne sommes pas parvenus à trouver la source de cette erreur au sein de notre méthode.

3.5 Guide d'utilisateur

L'interface graphique de l'application étant très intuitives, nous ne rentrerons pas dans les détails concernant l'utilisation de celle-ci. La seule option ne se trouvant pas dans les menus est celle de sauvegarde. Celle-ci se trouve dans le menu "Files" de la barre de menu lorsqu'on joue une partie de Sokoban.

```
\bin>java ApplyMov ../resources/input.xsb ../resources/mov.mov ../resources/output.xsb
```

Figure 3.2: Exemple d'appel à la commande ApplyMov

Pour appliquer un fichier .mov a un fichier .xsb, l'utilisateur est invité à se placer dans le répertoire "bin" (accessible via le chemin /code/bin depuis l'emplacement du fichier build.xml) et à utiliser la méthode "ApplyMov". Cette méthode prend 3 arguments : Le chemin relatif du fichier .xsb, le chemin absolu du fichier.mov, le chemin relatif du fichier de sortie. Voici à quoi ressemble un appel à cette commande.

Conclusion

En conclusion, nous pensons avoir bien mené notre projet. Tous les éléments imposés sont présents, et nous avons pris le temps d'ajouter un élément supplémentaire de taille. Ce projet nous a permis d'approfondir grandement nos connaissances en JAVA. Il a été agréable à réaliser et nous espérons pouvoir le défendre oralement.