Université
Caen
Normandie

Département d'informatique

Sokoban

Travail Personnel Approfondi

Maël Querré
Alexis Mortelier
Vincent De Menezes
Christina Williamson

10 avril 2018

Table des matières

Chapitre 1

Objectifs du projet

1.1 Description du concept

Sokoban est un jeu vidéo de puzzle inventé au Japon.

1.1.1 Règles du jeu [2]

Gardien d'entrepôt (divisé en cases carrées), le joueur doit ranger des caisses sur des cases cibles. Il peut se déplacer dans les quatre directions, et pousser (mais pas tirer) une seule caisse à la fois. Une fois toutes les caisses rangées (c'est parfois un vrai casse-tête), le niveau est réussi et le joueur passe au niveau suivant, plus difficile en général. L'idéal est de réussir avec le moins de coups possibles (déplacements et poussées).

1.2 Ce qu'il fallait faire

Dans un premier temps, il s'agit de réaliser un jeu jouable pour un humain avec importation de niveaux. Pour cela, des formats de données (comme .xsb, .sok ou .stb) dédiés peuvent être utilisés. Une interface graphique devra également être réalisée. Une fois ce travail préliminaire fini, il conviendra de proposer une fonctionnalité de résolution automatique de niveau (comme l'algorithme A*) et de permettre de faire jouer humain et ordinateur en parallèle. Enfin, une dernière étape consiste à rendre *anytime* l'algorithme de l'intelligence artificielle : cette dernière est obligée de jouer dès que l'humain fait un mouvement.

1.3 Ce qui existe déjà

Pour l'intelligence artificielle nous aurons besoin de calculer si le jeu est bloqué à chaque déplacement. Le Sokoban YASC ajoute aux fonctionnalités du Sokoban la possibilité de faire des retours en arrière mais surtout permettre d'éviter les situations où le jeu se retrouve bloqué.

Chapitre 2

Fonctionnalités implémentées

2.1 Description des fonctionnalités

2.1.1 Réalisation du jeu

Le jeu du Sokoban est un casse tête se jouant sur une grille. Chaque case de la grille possède un type d'élément, il en existe 6 :

- type vide
- type mur
- type caisse
- type joueur
- type cible
- type caisse sur cible
- type joueur sur cible

Pour modéliser ce problème nous avons décidé de séparer le jeu en 3 parties (plateau, élément du plateau, chargement des niveaux).

Plateau

Afin de générer un plateau nous avons eu l'idée de créer un tableau possédant les éléments du plateau. Dans cette classe nous établissons toute les règles nécessaires pour le fonctionnement du Sokoban. L'initialisation de la partie se fait par cette classe comme les méthodes de déplacements et Cette classe permet d'initialiser une partie et d'établir les déplacements sur le plateau, elle nous permet de detecter si la partie est finie. Il est nécessaire de lui fournir des informations afin qu'elle puisse simuler une partie, pour cela elle utilise la classe de chargement des niveaux pour obtenir les informations nécessaires pour lancer la partie.

Element du plateau

Les elements du plateau vont correspondre à nos types d'états, la classe va initialiser les 6 états vu précedemment. Elle a besoin de methodes de restitution afin qu'elle puisse communiquer avec le plateau. En effet, le plateau va interroger le type de la case lorsqu'on voudra agir sur celle-ci. Elle possède aussi une méthode permettant de changer son état, ce qui servira pour les déplacements demandés.

Cette classe va être aussi utilisé par la classe d'Intelligence Artificielle (voir 4). Celle-ci aura besoin d'autres informations que les types des cases afin que celle-ci puisse se reperer

sur le plateau. Cette classe doit restituer trois couches :
— couche de vérification
— couche de trajectoire
— couche de deadlock 9

La couche de vérification permet de vérifier chaque case du plateau en considérant le pi corral (page 10). La détection de pi corral nous permet de gagner du temps d'efficacité lors de la recherche dans les trois couches. Cette couche doit être réinitialisée à chaque utilisation de celle-ci, car, elle sert pour la détection de destination de caisse mais aussi lors du calcul des couches de trajectoires et de deadlock. Remarque : cette couche sert d'initialisation pour tous les autres calculs de l'intelligence artificielle.

La couche de trajectoire va attribuer à la case ciblé un nombre. Ce nombre permet de connaître le nombre de case entre la position d'origine et la case ciblé. Remarque : en utilisant une décrementation pour détecter la case voisine nous pouvons déduire le meilleur chemin entre la case ciblé et l'origine.

La couche de deadlock est une couche booléenne, elle permet de savoir si la case ciblé est une case gelée 9.

Chargement des niveaux

Le chargement des niveaux est une classe permettant d'obtenir a partir d'un fichier les informations nécessaires lors de la création d'un plateau. Elle doit donc restituer une liste de chaine de caractère afin que la classe plateau puisse initialiser le niveau. La classe possède deux méthodes, sauvegarde et chargement de niveau permettant à l'utilisateur de sauvegarder sa progression. Pour le fonctionnement de la classe plateau, cette classe doit pouvoir indiquer la largeur et la hauteur de la grille. Une méthode existe donc permettant d'obtenir toutes les informations nécessaire à l'initialisation d'un niveau.

Cette conceptualisation du jeu nous permet de faciliter l'acces aux informations pour l'intelligence artificielle mais aussi facilité la communication avec l'interface graphique. Lors du projet nous avons eu une nouvelle méthode d'approche afin de résoudre le probleme donné, celle-ci oublie le principe de tableau d'element du plateau. Elle est composée d'ensemble. Ces ensembles représente trois parmis six des types présentés, il y a donc un ensemble pour les caisses, les destinations et le curseur. Cette méthode est très efficace pour le déplacement via la superposition des types (une caisse sur une destination deviens un nouveau type dans l'univers du sokoban alors qu'avec cette méthode il y a juste les mêmes positions dans deux ensembles, caisse et destination). Mais cette méthode ne permet pas de simuler nos trois couches ce qui rends la vision de l'intelligence artificielle sur le plateau difficile, bien qu'avec cette nouvelle méthode d'autres problèmes peuvent être éviter.

2.1.2 Modélisation de l'intelligence artificielle

Avant de pouvoir faire fonctionner notre intelligence artificielle, il faut lui imposer des limites. Les limites seront calculées dans un premier temps par les *deadlocks*[1].

1. voir chapitre 3.1.2

D'autres limites peuvent être ajoutées comme la vérification d'un plateau déjà effectué ou bien une simple vérification de direction afin d'éviter que l'intelligence artificielle ne calcule trop de coups avant de s'apercevoir qu'elle aurait pu obtenir une meilleure situation à l'aide d'un guidage de direction.

L'intelligence artificielle ignore la notion de *deadlock*, pour cela il faut lui faire détecter les situations de *deadlocks* à l'aide de fonctions. Dans un premier temps, il est facile de détecter un simple *deadlock*, il suffit de partir des destinations puis de tirer une caisse sur toutes les caisses possibles du plateau. Une fois ce processus fini, toutes les cases qui n'ont pas été visitées par la caisse sont automatiquement des cases gelées [2]. Pour détecter les impasses, le principe est assez simple. Nous détectons l'entrée d'un tunnel et nous vérifions que pour toutes les cases du tunnel, il n'y ait pas d'obstacle comme un mur ou une caisse (en effet, il préférable de détecter aussi les cases gelées dans le tunnel, car un tunnel qui effectue un virage n'est pas reconnu comme impasse pour la mobilité du joueur mais ne pourra pas faire passer une caisse).

À chaque déplacement d'une caisse, nous vérifions si la caisse ne se trouve pas sur une situation de deadlock à l'aide d'un algorithme qui va détecter si une case est gelée sur un axe :

Algorithme 1 : Détection d'une situation de deadlock sur l'axe horizontal

Entrées : Des positions i et j correspondant à une coordonnée d'une case à vérifier
Sortie : Une valeur booléenne correspondant à l'axe bloqué

```
1  haut ← false;
2  bas ← false;
3  si la case en haut et en bas n'est pas un mur alors
4  |  si la case en haut ou en bas n'est pas une case gelée alors
5  |  |  si la case en haut est une caisse alors
6  |  |  |  haut ← true;
7  |  |  si la case en bas est une caisse alors
8  |  |  |  bas ← true;
9  |  |  si haut == false et bas == false alors
10 |  |  |  retourner false;
11 retourner true;
```

Pour savoir si notre case est une case gelée, nous lui attribuons une « couche » dans notre tableau d'objets. Cette couche booléenne nous permet de connaître l'état de la case.

Notre intelligence artificielle utilisera le principe de l'algorithme A* (voir page 9), dans un premier temps nous detectons les chemins pour atteindre les caisses. A l'aide d'une fonction qui va agir comme un radar (c'est à dire que la fonction va parcourir chaque case dans toute les directions depuis une position donnée). On remarque que cette fonction arrive à rentrer dans un tunnel puis à s'étendre afin de couvrir tout la zone du plateau. Grace à cette fonction nous pouvons déduire le meilleur chemin de position à une autre.

Une fois le chemin donnée pour l'intelligence artificielle afin de parvenir au caisse, il faut trouver le chemin le plus porche d'une caisse à une destination. Pour optimiser la recherche il faut garder en mémoire les plateaus deja simulé car il est possible d'obtenir plusieurs fois le même plateau (ce qui évite de potentiel bug de répétition).

2. voir chapitre 3.1.2

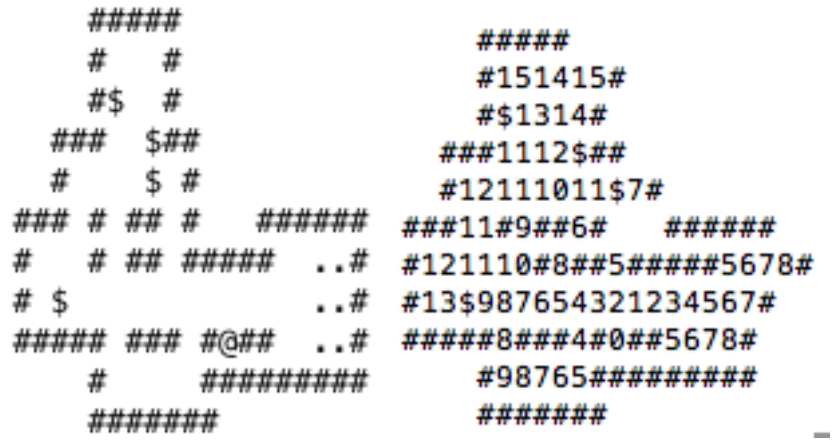

FIGURE 2.1 – Plateau rempli par des cases mesurant la distance entre une position donnée

2.1.3 Éditeur de niveau

Afin d'aller plus loin que ce qui était attendu, nous avons décidé de modéliser un éditeur de niveau. Afin de le rendre le plus efficace possible, nous avons décidé de le rendre totalement indépendant.

L'éditeur de niveau agit au plus bas possible, on interfère directement sur des chaînes. En effet, cela permet d'importer n'importe quel niveau déjà existant ou bien d'en créer un depuis zéro.

Dans l'éditeur, il est possible de :

— changer case par case les éléments du niveau ;
— augmenter la taille du niveau en ajoutant directement des lignes ou des colonnes du type de case demandé ;
— réduire la taille du niveau en supprimant des lignes ou colonnes à un index demandé.

L'interface graphique nous permet d'éditer le niveau d'une façon intuitive. Afin de choisir un élément, il suffit de cliquer sur un bouton représentant par une image le type de l'élément désiré. Pour changer une case, il suffit ensuite de cliquer sur celle à modifier. Pour l'ajout ou la suppression de lignes et de colonnes, des boutons sont implémentés afin d'agir rapidement et efficacement sur le niveau. Une fois le niveau réalisé, il peut être sauvegardé, au format `.xsb`, pour pourvoir le réutiliser plus tard.

2.1.4 Interface graphique

Lanceur de jeu

Au lancement du jeu, un lanceur (Figure 2.2) s'affiche afin de guider le joueur dans l'initialisation d'une partie.

Il est donc possible de lancer un nouveau jeu seul ou contre l'intelligence artificielle, de charger une partie ou un niveau, d'éditer un niveau soi-même (Figure 2.3) ou encore de voir les règles du jeu (Figure 2.4).

FIGURE 2.2 – Le lanceur du jeu

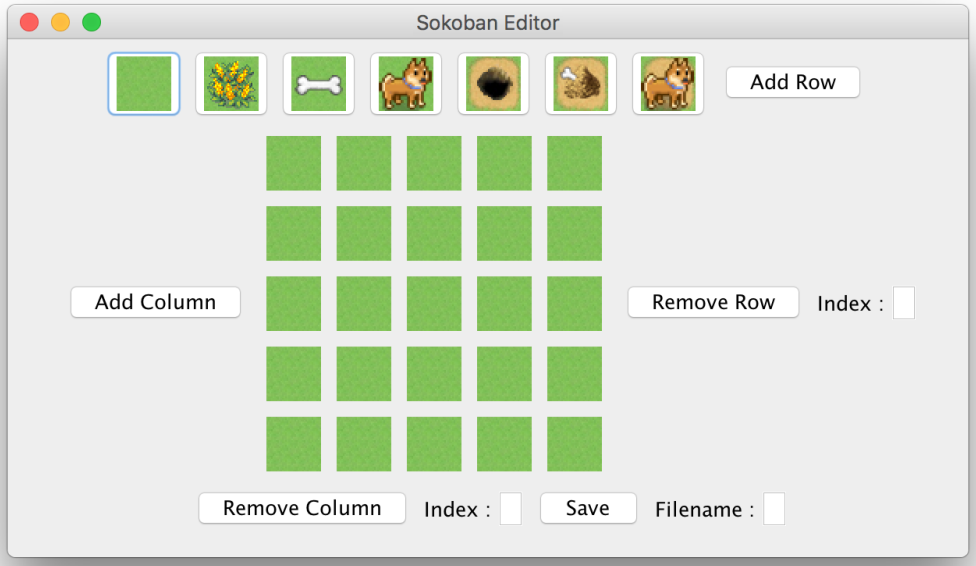

FIGURE 2.3 – L'éditeur de niveau

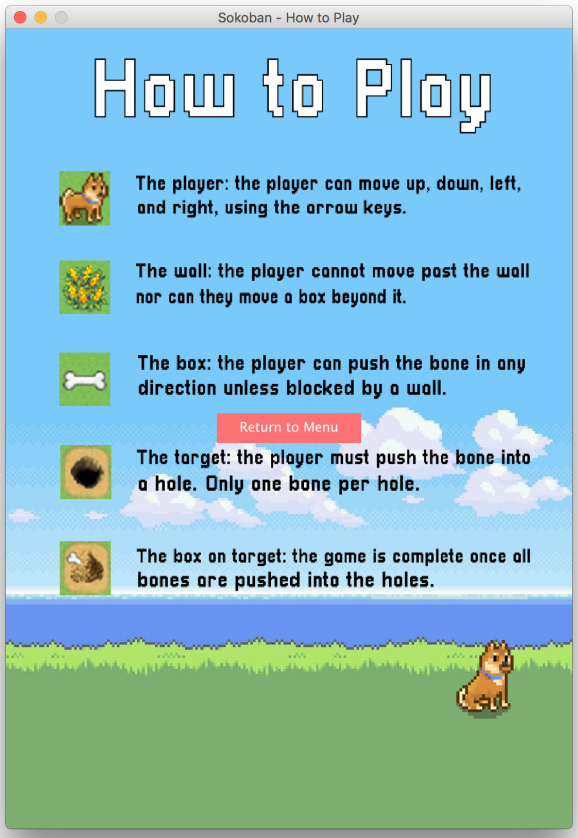

FIGURE 2.4 – La fenêtre des règles du jeu

2.2 Organisation du projet

Pour réaliser le projet nous avons réparti le travail en plusieurs parties. Deux élèves ont travaillé à temps plein sur l'interface graphique, les deux autres sur la réalisation du jeu de base. Une fois cette phase terminé, nous avons avancé avec tous les membres du projet sur l'intelligence artificielle. Lors de la finalisation de cette phase nous avons réfléchi à l'évolution du projet, grâce à l'implémentation d'options améliorant la qualité du jeu ou de l'intelligence artificielle.

Chapitre 3

Éléments techniques

3.1 Algorithmes

3.1.1 L'algorithme A*

L'algorithme A* [1] est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final. Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. C'est un algorithme simple, ne nécessitant pas de prétraitement, et ne consommant que peu de mémoire.

3.1.2 Jeu Bloqué et Deadlock

Il existe plusieurs façons d'obtenir un jeu bloqué, le plus courant est lorsqu'une caisse est poussée sur une case *deadlock* mais il existe aussi le cas des tunnels, des diagonales. Le cas le plus compliqué à détecter est celui des bipartites.

Une case *deadlock* est une case interdite aux caisses, en effet, si une caisse est poussée sur une case *deadlock*, le jeu sera forcément bloqué et aucune résolution du niveau ne sera possible.

Une case bloquée est une case où l'un des axes horizontal ou vertical est bloqué. Le jeu n'est pas automatiquement bloqué car si une destination se situe sur un axe non bloqué d'une case bloquée, alors le jeu n'est pas bloqué car une caisse pourrait être poussée à cette destination.

Une case gelée est une case où les deux axes sont bloqués.

Positionner une caisse sur une case gelée entrainera un jeu bloqué.

Un tunnel peut-être composé de mur ou de boite. Un tunnel peut devenir une impasse si au bout de celui-ci se trouve un mur ou une caisse.

Les blocages de diagonales fermé dans le cas le plus fréquents apparait lorsque 3 caisses sont positionné à coté de façon a former un "L". On appelle cette formation dans les thermes du Sokoban une diagonale, pour que cette diagonale soit fermé il suffit que la derniere case du carré ne soit pas acessible dans ce cas on dit que c'est une diagonale fermé.

On peut dire que chaque caisse possedent une position d'arriver, dans certain cas échanger de destination certaines caisse ne posera pas de probleme, sauf une caisse possède qu'une seule possibilité de destination alors le jeu se trouve bloqué par le type de deadlock « bipartite ». Pour éviter ce genre de problème il faut connaitre les destinations possibles pour chaque caisse, et vérifier que chaque caisse possède bien au moins une destination, si ce n'est pas le cas alors c'est un blocage bipartite.

Les caisses peuvent rendre certaines zone bloqué voir même gelée, il existe certaine cas qui lorsqu'on dépose un caisse sur une destination celle-ci gêle les cases voisines ce qui peut bloquer encore une fois le jeu.

Remarque : Chaques coups doivent être calculé à l'avance en effet déposer une caisse peut s'avéré être un piège (entre les blocages de diagonale, celui des case gelêe), il est préférable de bien s'assurer que chaque coups ne créer une situation de blocage.

Le corral est une zone que le joueur ne peut pas atteindre.

3.2 Structures de données

Pour représenter les données du jeu, nous avons utilisé un tableau de cases possédant chacune un type (case vide, mur, caisse, joueur, cible, caisse sur cible, joueur sur cible).

Chapitre 4

Architecture du projet

4.1 Diagramme de classes

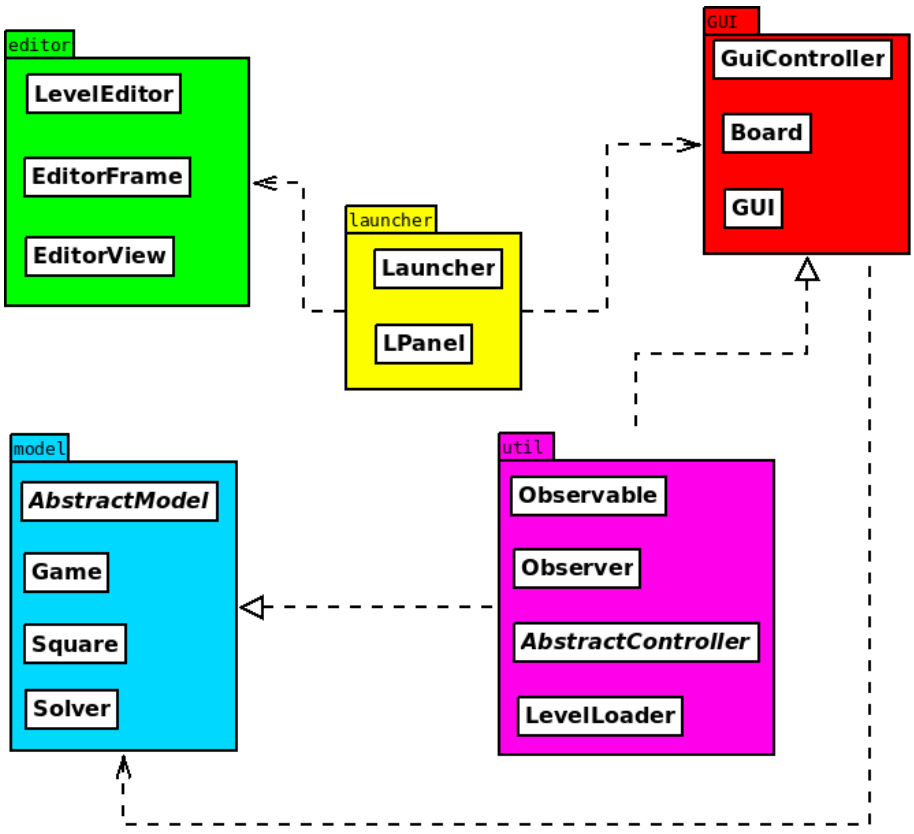

FIGURE 4.1 – Principaux packages du projet

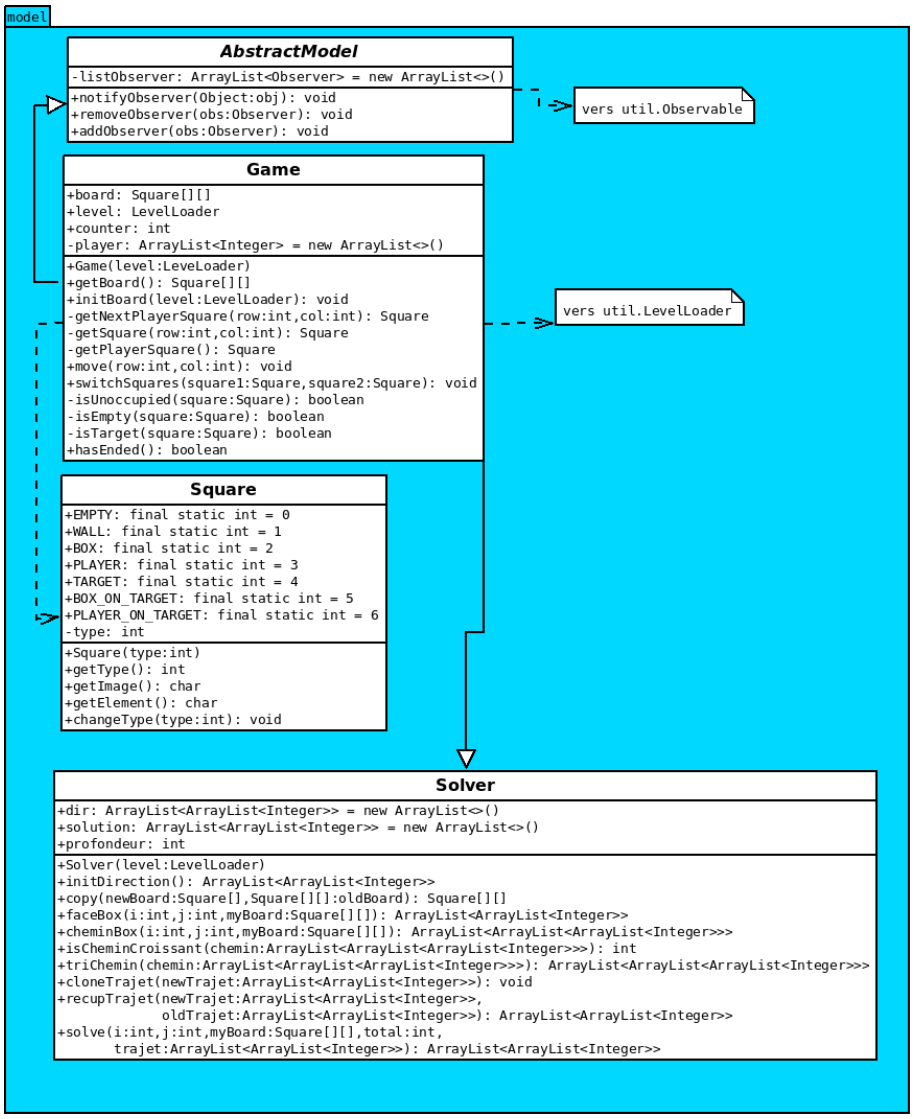

FIGURE 4.2 – Diagramme de classes du package `model`

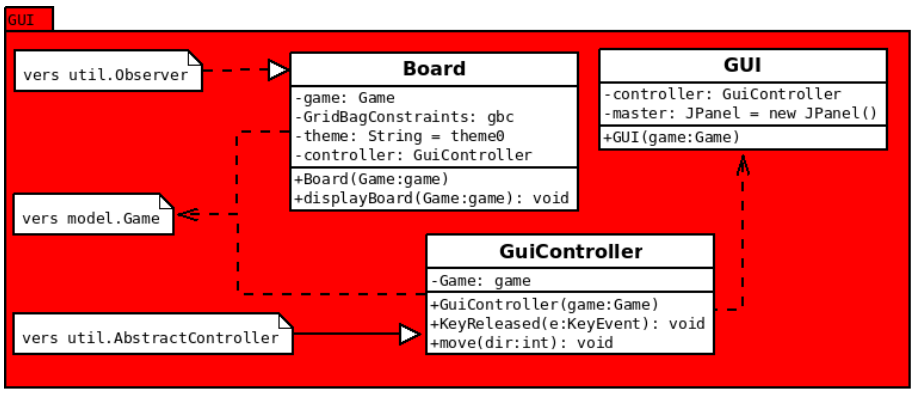

FIGURE 4.3 – Diagramme de classes du package `gui` (graphical user interface)

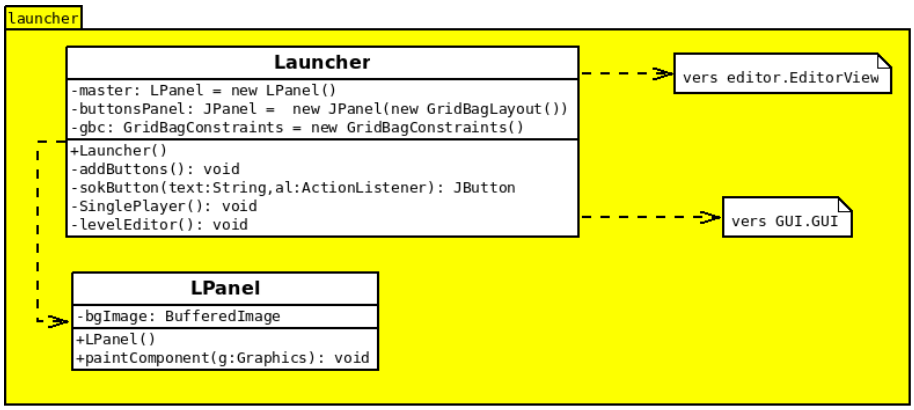

FIGURE 4.4 – Diagramme de classes du package `launcher` (lanceur du jeu)

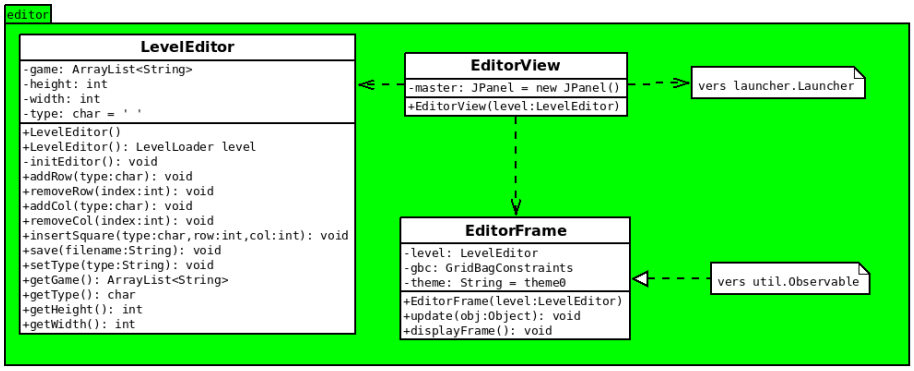

FIGURE 4.5 – Diagramme de classes du package `editor` (éditeur de niveau)

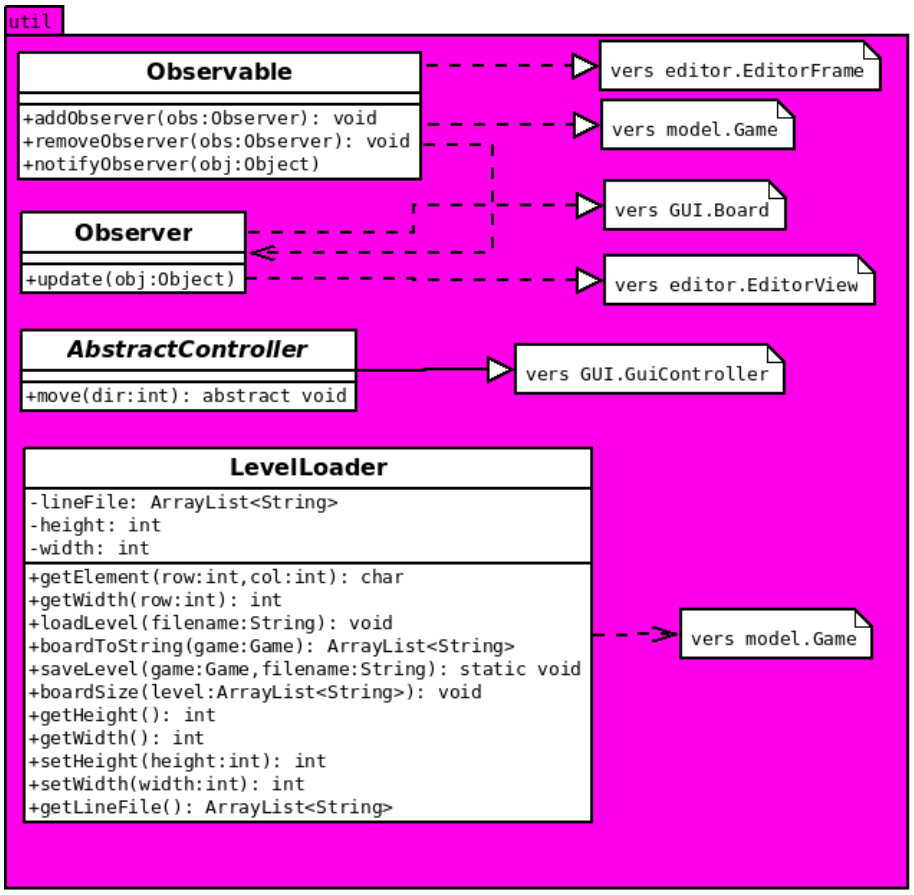

FIGURE 4.6 – Diagramme de classes du package `util` (utilitaires au MVC)

Chapitre 5

Conclusion

5.1 Récapitulatif des fonctionnalités principales

Un plateau permettant de jouer au jeu Sokoban avec toutes les règles, les fonctions de déplacement et d'initialisation lui permet de proposer au joueur une version du jeu jouable sous console.

Une intelligence artificielle permettant de résoudre des niveaux de Sokoban. Elle utilise des fonctions qui lui permettent de se repérer sur le plateau mais aussi des fonctions qui permettent de limiter les coups inutiles grâce à la détection de *deadlock*.

5.2 Propositions d'améliorations

Une seconde version du jeu a été modélisée mais, par manque de temps, nous n'avons pas pu intégrer cette version avec l'intelligence artificielle. Pour l'intelligence artificielle, une amélioration de recherche entre les caisses et les destinations permettraient de résoudre le bug de cycle. L'ajout d'un niveau en coopération pourrait rendre le jeu plus riche grâce a des niveaux dynamiques, ce qui permettrait d'avoir la possibilité de jouer sur le même plateau avec 2 curseurs.

Bibliographie

[1] Algorithme A*. Algorithme A* — Wikipedia, l'encyclopédie libre, 2018. [accédé le 20-février-2018].

[2] Sokoban. Sokoban — Wikipedia, l'encyclopédie libre, 2018. [accédé le 20-février-2018].