Année universitaire 2015-2016

Projet Tuteuré

Sujet : Soko'BanZai

Auteurs :
Guillaume DROUART
Marwan LAKRADI

Examinateurs :
Jean-Luc LAMBERT
Bruno ZANUTINNI

Diplôme préparé : L3 Informatique

Sommaire

1 Introduction :

1.1 Contexte

Les membres de l'équipe de recherche MAD (Modèles, Agents, Décision) travaillent sur des algorithmes d'intelligence artificielle et souhaiteraient pouvoir les tester sur le jeu Sokoban.

1.1.1 Objectifs du projet

L'équipe de recherche MAD souhaiteraient pouvoir, lors des démonstrations des différentes techniques d'intelligences artificielles (dans les lycées, collèges, expositions, etc.), visualiser graphiquement leurs fonctionnements et leurs vitesses d'exécution sur un jeu de Sokoban.

1.1.2 Équipe Modèles, Agents, Décision

L'équipe s'intéresse à des problématiques d'intelligence artificielle dans le contexte d'agents autonomes. Son activité peut être résumée ainsi :

Étudier et proposer des méthodes pour permettre à un groupe d'agents adaptatifs, en temps réel et sous contraintes de ressources, plongé dans un environnement dynamique, de prendre des décisions rationnelles pour la réalisation d'une mission.

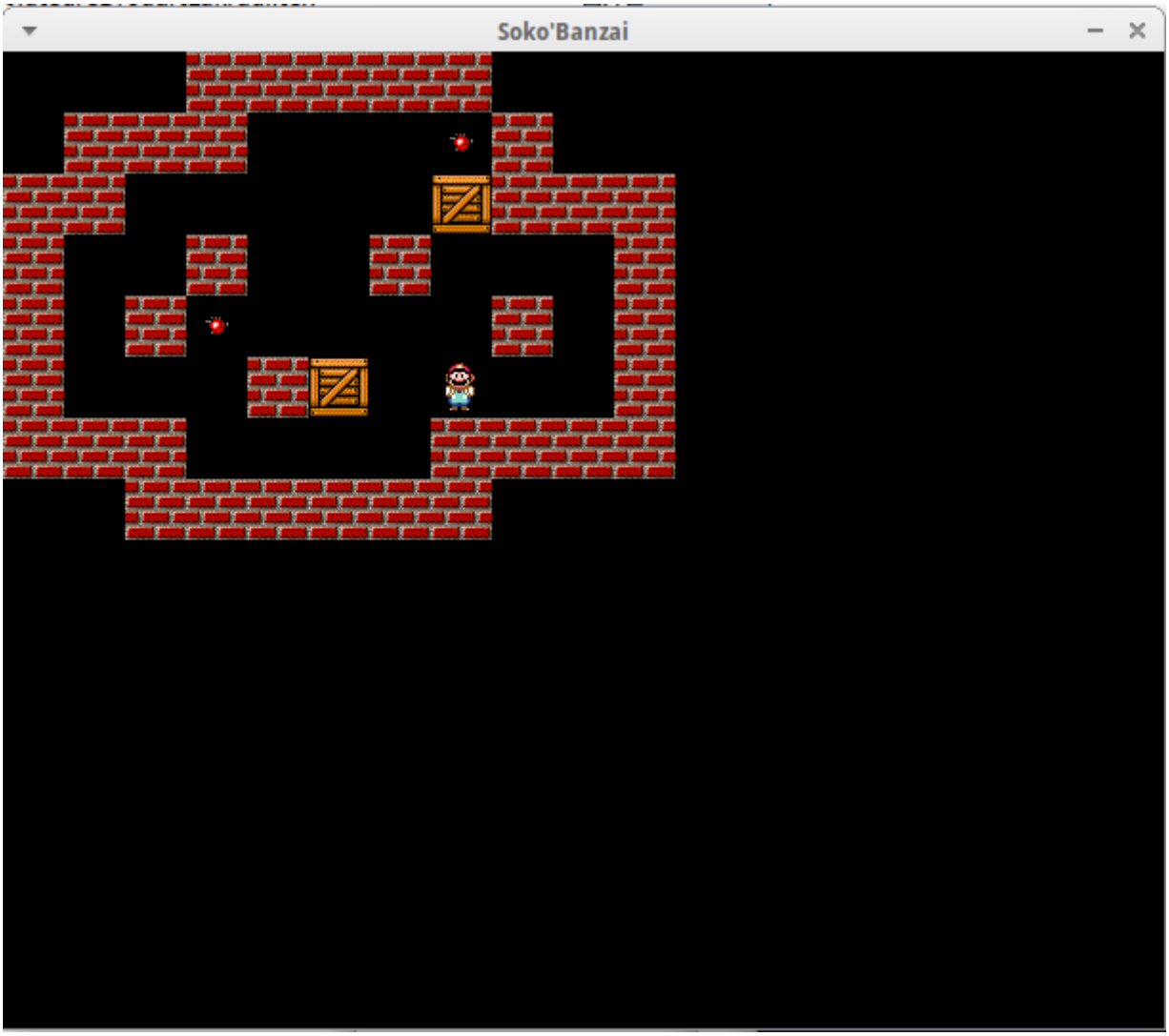
Ces activités sont organisées autour de trois axes :

— Modèles
— Agents
— Décision

Dans l'axe « Modèles », nous étudions les modèles, les algorithmes et la complexité du raisonnement, en particulier le raisonnement logique et le raisonnement sur le temps, l'espace ainsi que les préférences, et les problématiques de compilation de connaissances. Dans l'axe « Agents », on s'intéresse aux systèmes multi-agents, aux systèmes de réputation, à la formation de coalition, et à la modélisation ainsi qu'à la preuve de comportements. Dans l'axe « Décision », on s'intéresse aux modèles et algorithmes pour la décision.

1.1.3 Qu’est-ce que le Sokoban ?

Sokoban est un jeu vidéo de puzzle inventé au Japon. Ce nom désigne un garde d’entrepôt. 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, plus complexe et plus élaboré. L’idéal est de réussir avec le moins de coups possibles (déplacements et poussées).



1.2 Analyse du besoin

1.2.1 Fonctionnalités à réaliser

— Implémentation d'un Sokoban en MVP
— Implémentation d'un algorithme A*
— Mise en place d'un système de test de différents algorithmes sur le Sokoban
— Mise en place d'un générateur aléatoire de niveaux de Sokoban

1.2.2 Demande initiale

Demande sur le MVP :

Le code d'un Sokoban sans pattern a été proposé au client mais le client a refusé car le code n'était pas adaptable à ses algorithme.
Le choix d'implémentation d'un MVP (Modèle Vue Présentateur) a donc été mis en place.

Demande sur le A* :

Le projet en l'état a pris du retard mais l'implémentation d'un algorithme de résolution du Sokoban va bientôt être mis en place, le A* sera implémenté avec comme heuristiques une distance euclidienne ou une distance de Manhattan.

Demande sur la mise en place d'un générateur aléatoire de niveaux de Sokoban :

La mise en place d'un générateur aléatoire de niveaux de Sokoban n'a pas pu être effectuée.

1.2.3 Le cahier des charges

Notre cahier des charges se compose de trois parties :

— Une partie jeu où nous devons essayer de faire un jeu qui soit à la fois léger et qui permette une bonne jouabilité.
— Une partie interface graphique où nous devons essayer de faire l'interface au graphisme abouti au maximum de nos capacités avec le temps qu'il nous était, afin que le jeu attire l'oeil de toute personne intéressée.
— Mise en place d'un système de test de différent algorithme sur le Sokoban
— Une partie intelligence artificielle permettant au personnage de résoudre les différents niveaux de Sokoban tout seul.

2 Réalisation

2.1 interface

3 classes principales différentes :

VUE PRINCIPALE :

Cette classe est directement reliée avec le présentateur, elle représente la vue générale de notre application, en outre, elle possède un "KeyListener" sur notre classe PanneauGrille, permettant d'écouter le déplacement du personnage.

Notre VuePrincipal instancie un nouveau modele, donc une nouvelle grille par le biais du présentateur lorsque le niveau est terminé. Nous passons ainsi au niveau suivant et la vue se met a jour en fonction du modele. En demande à la vue de la grille de notre classe PanneauGrille de se mettre a jour.

AFFICHAGE DE LA GRILLE DE JEU :

La classe nommé PanneauGrille est instanciée avec comme paramètre notre grille du modele, puis l'affichage est directement dessinée après un appel à la méthode : « public final void dessiner(Grille grille) » en parcourant la grille, nous récupérerons le contenu de chaque case, par la suite nous instancions une nouvelle VueCase associée a chaques différentes case de la grille, puis chacune des cases de la grille vaut la valeur de l'instance de vue case, enfin nous ajoutons un listener sur cette même instance. Lors de la mise a jour de la vue, il y a un appel à la methode "update" qui supprime tous le contenu de la grille actuel, et redessine la grille la vue correspondant à la grille passée en argument.

VUE DES CASES :

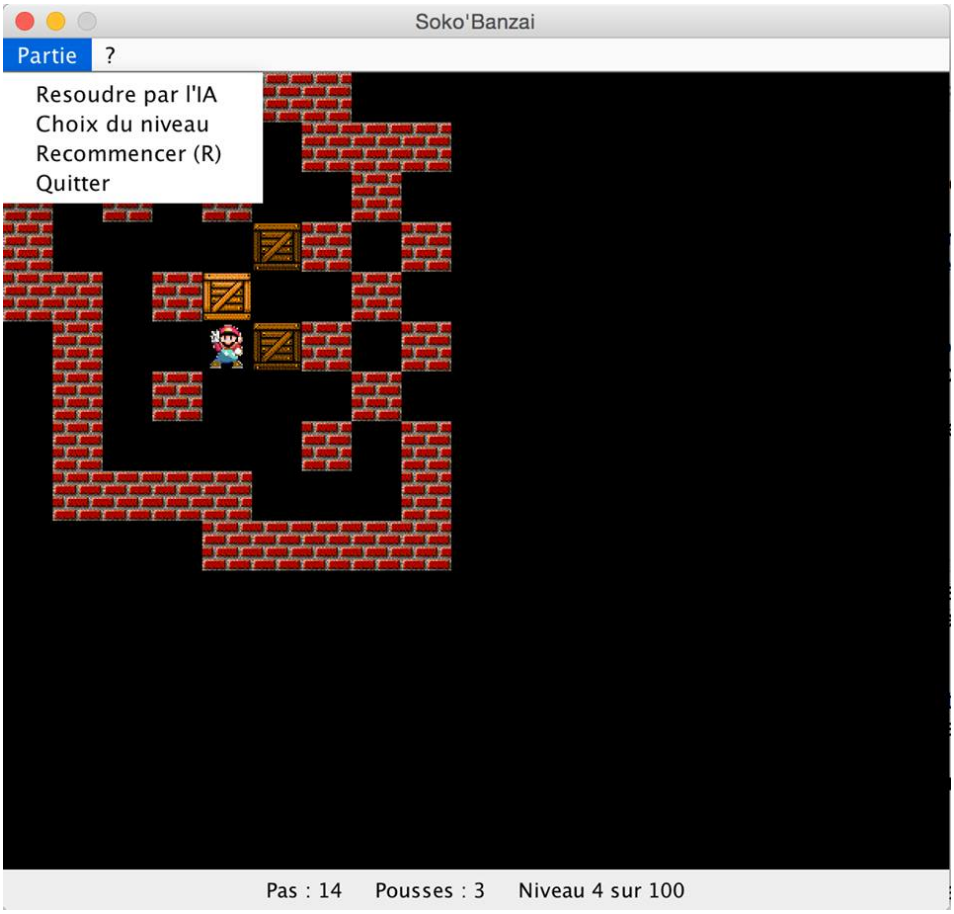
Vue graphique de chaque case de la grille. Notre classe possède un attribut de classe de type tableau ImageIcon[] appelé "sprite" représentant les sprites des différentes images de cases stocké dans le repertoire "ressources/sprites/" de notre application.

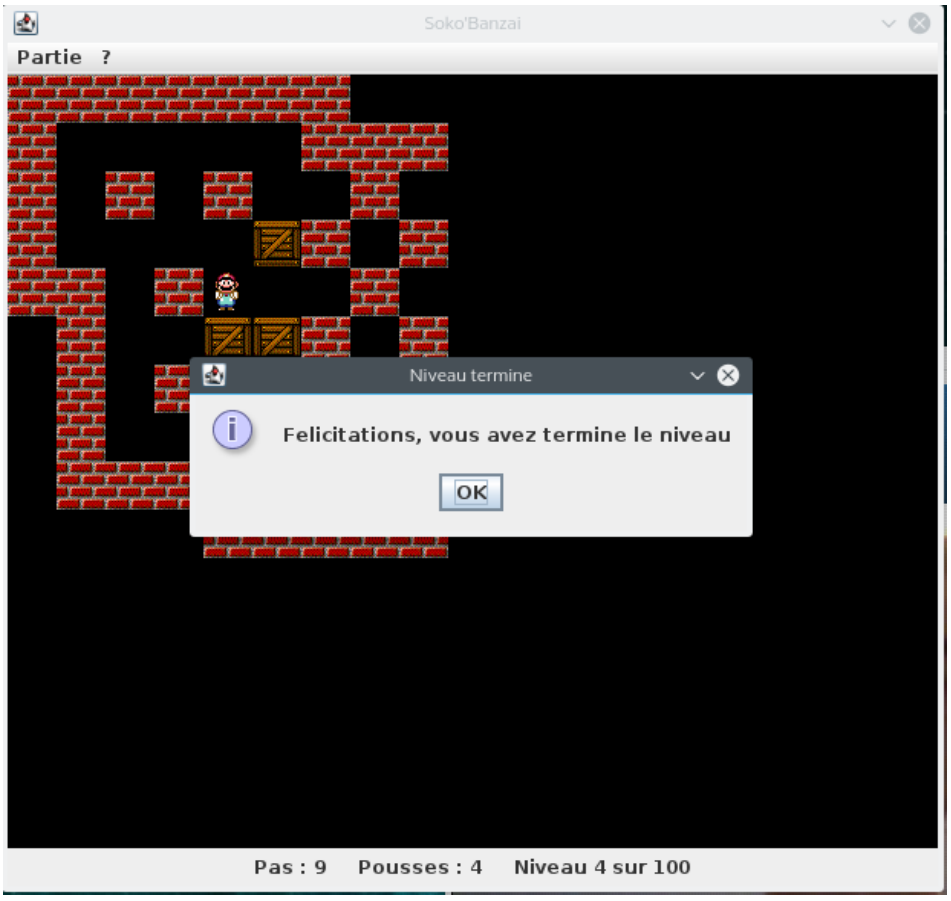
Notre vue de case est donc instanciée par le biais d'un attribut de type modele.Case associé à son image correspondante dans le tableau des sprites. En effet, il existe un type énumérée dans notre classe modele.Case permettant d'associé le statut d'une case a une image particulière grâce a la méthode java "ordinal()". Nous parcourons les enums en parallèles.

Notre classe implémentant l'interface « Observer » nous ajoutons a chacune de nos case un observer, en réalité, notre classe VueCases observe la classe modele.Case.

Nous possédons en conséquence la méthode "public void update(Observable observable, Object object)" qui est appelée quand l'objet observe a change (Nous changeons l'image de la case observée).

2.2 Vue des interfaces





2.3 modèle

PERSONNAGE :

Le personnage est instancié dans la classe Grille représentant notre plateau de jeu. Il est définit par ses mouvements et son orientation.
Le personnage possède donc des coordonnées (x, y) qui lui sont propre dans la grille ce qui permet de le situer, nous pouvons ensuite incrémenter ou décrémenter ces coordonnées en fonction du déplacement.
Par exemple, lorsque nous appuyons sur la flèche directionnelle de droite, nous appellons une méthode aDroite() qui elle-même va appeler la méthode deplacementVers(deltaX, deltaY) où deltaX vaut 0 et deltaY vaut +1, dans laquelle nous avons défini les éléments de façon à ce que le personnage ne puisse aller plus loin que le cadre de jeu, et qu'il ne soit pas en capacité de pousser 2 caisses en même temps.
La méthode aDroite() va donc appeler la methode deplacementVers(0, +1)incrémenter la coordonné « y », modifier l'orientation du personnage et par la suite assigne dans ses nouvelles coordonnées une nouvelle image associée à son orientation.
De plus la méthode deplacementVers(deltaX, deltaY) va effacer l'ancienne case où était le personnage.
Puis notifie ses changements à la vue par le biais de la méthode notifier(), car la classe Personnage hérite d'Observable.

GRILLE :

La grille est la classe principale du jeu, reliée avec le présentateur. La méthode principale permet de tester la victoire de l'utilisateur à chaque déplacement du personnage, l'utilisateur gagne lorsqu'il n'y a plus d'objectif et que le personnage n'est pas sur un objectif de la grille.

testVictoire() :

nous testons la valeur binaire d'un booléen égal à true (donc valant 1) ET binaire sur le fait que toutes les cases de la grille soient différentes d'un objectif ET logique d'un personnage sur objectif.
Dès lors que toutes les valeurs de bits sont égale a 1, la victoire est acquise.
Une fois la partie terminée, nous initialisons la grille du niveau suivant, et ainsi de suite.
Cependant, la grille ne peut être instanciée seulement si le niveau a été lu au préalable. En effet notre constructeur principal de la grille est : « public Grille(GestionNiveau gestionNiveau) » où le paramètre « gestionNiveau » nous permet de savoir sur quel numéro de niveau nous sommes. Nous pouvons ainsi initialiser la grille de jeu d'un niveau de sokoban en utilisant les fichiers .sok de niveau se trouvant dans le répertoire /ressources/niveau/

```
victoire = victoire & (this.cases[i][j].lireStatut() != TypeCases.Objectif) &&
                      (this.cases[i][j].lireStatut() != TypeCases.PersonnageSurObjectif);
```

GESTIONNIVEAU :

C'est la classe permettant de lire et de se déplacer dans les différents niveaux du jeu. Nous utilisons un simple BufferedReader sur notre fichier de niveau .sok lors de l'instanciation de notre classe. De plus, nous avons la possibilité d'aller au niveau suivant, précédent, ou de se rendre à un niveau voulu passé en paramètre de notre méthode « public void allerAuNiveau(int nbNiveau) ».

CHARGEURNIVEAU :

La classe qui permet ce chargement s'appelle ChargeurNiveau, celle-ci lit un fichier ".sok" qui est un simple fichier .txt mais appelé .sok pour Sokoban qu'il récupère via la classe Grille et le parse. Dès lors que la variable victoire passe à "true", la méthode partieTermine() est appelée, permettant ainsi d'incrémenter le niveau et d'initialiser la grille sur ce nouveau niveau (ChargeurNiveau charge alors le nouveau niveau).
Dans Grille la méthode suivante envoie au ChargeurNiveau le niveau qu'il doit lire :

```
public void initialiserGrille(int niveau) {
        try {
         gestionNiveau.allerAuNiveau(niveau);
         String fichierNiveau = "src/ressources/niveaux/lvl" + niveau + ".sok";
         //récupère l'instance du singleton et instancie le niveau sur la grille
         //avec le niveau associe
         ChargeurNiveau chargeurNiveau = ChargeurNiveau.getInstance();
         chargeurNiveau.initialiserNiveau(this, fichierNiveau);
        } catch ( IOException e ) {}
}
```

Le ChargeurNiveau lit le fichier ".sok" indiqué caractère par caractère, jusqu'à ce qu'il rencontre la lettre « A » où il doit s'arrêter. En effet, nous avons récupéré des niveaux pré-fait sur internet d'où l'importance de citer l'Auteur. Nous avons cependant modifié quelques niveaux, par conséquent nous avons dû changé le nom de l'auteur. Quand ChargeurNiveau lit ce fichier il créé une nouvelle « Case(TypeCases statut) » et transforme chaque caractère lu en type énuméré représentant le statut de cette case. Par exemple pour un "#" il instancie un TypeCase.Mur, un "@" représente TypeCase.Personnage...

Le ChargeurNiveau possède une sous classe « ChargeurNiveauSingleton » représentant notre singleton et possède un attribut d'instance statique qui ne sera chargée en mémoire que lorsque nous y ferons référence pour la première fois. Celle-ci possède une méthode permettant d'initialiser un niveau sur une grille à l'aide du nom du fichier, en parsant le fichier de niveau. Chaque élément du fichier instancie un certain type de case de notre classe Case en utilisant son type énuméré TypeCases.

CASE ET PERSONNAGE :

Les classes Case et Personnage correspondent à deux classes distinctes héritant de la classe Observable, elles sont ainsi observées par la vue représentant l'affichage graphique des cases.
Notre classe Case possède un enum permettant de différencier les divers types de cases. Une case est instanciée par la valeur de son enum ou par une case elle même en changeant le statut de celle-ci, pour cela il faut en changer le statut, afin de pouvoir le notifier à son observateur.
La classe Personnage, n'est en réalité qu'une classe servant à l'instanciation et au déplacement de celui-ci. En effet à chaque keyEvent tapé, le personnage se déplace et le notifie à son observateur, a chaque déplacement l'orientation du personnage est modifié pour permettre un rendu visuel un peu plus réaliste.

Remarque :

Au début de notre projet nous voulions que les caisses se déplacent 1 fois sur 10 en glissant d'une case de plus que lors du fonctionnement normal. Pour cela nous devions mettre en place une interface DeplacementParDefaut, puis une classe DeplacementGlissant qui reproduit un déplacement par défaut avec une chance de faire glisser une caisse d'une case de plus sur un côté ou en face du personnage. Cependant, nos classes de déplacement ne sont pas encore fonctionnelle (excepté celle du personnage qui représente un déplacement par défaut).

2.4 La conception technique

2.4.1 Pattern MVP

Modèle :

Les classes représentant les données manipulées à travers l'interface utilisateur.

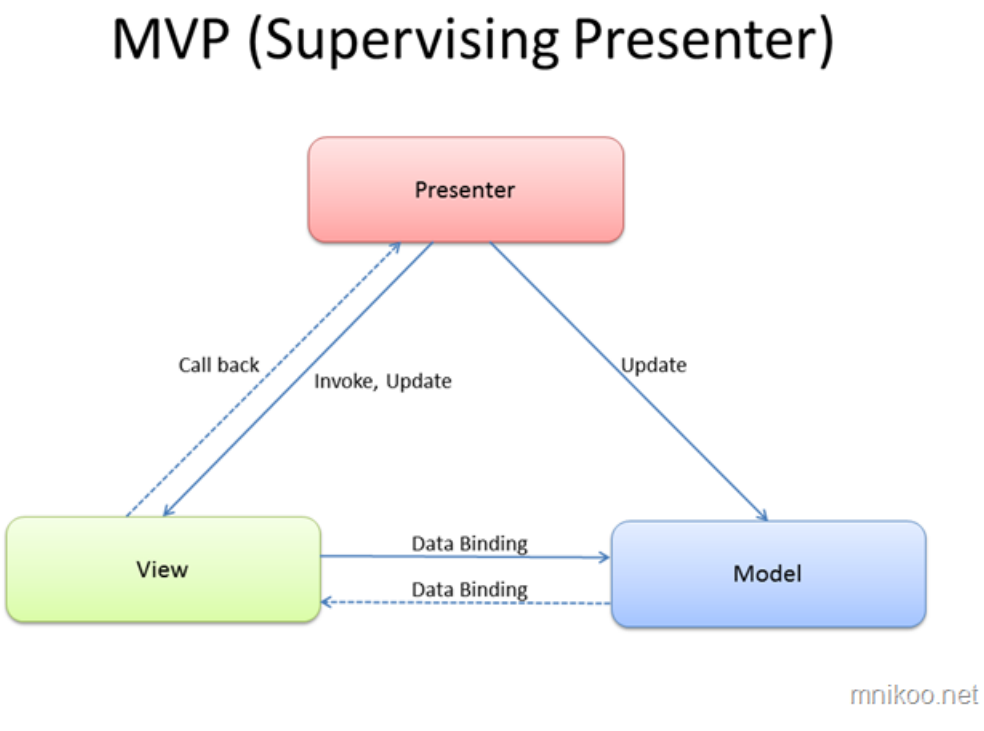
Vue :

Les classes présentant une vue des données à l'utilisateur.

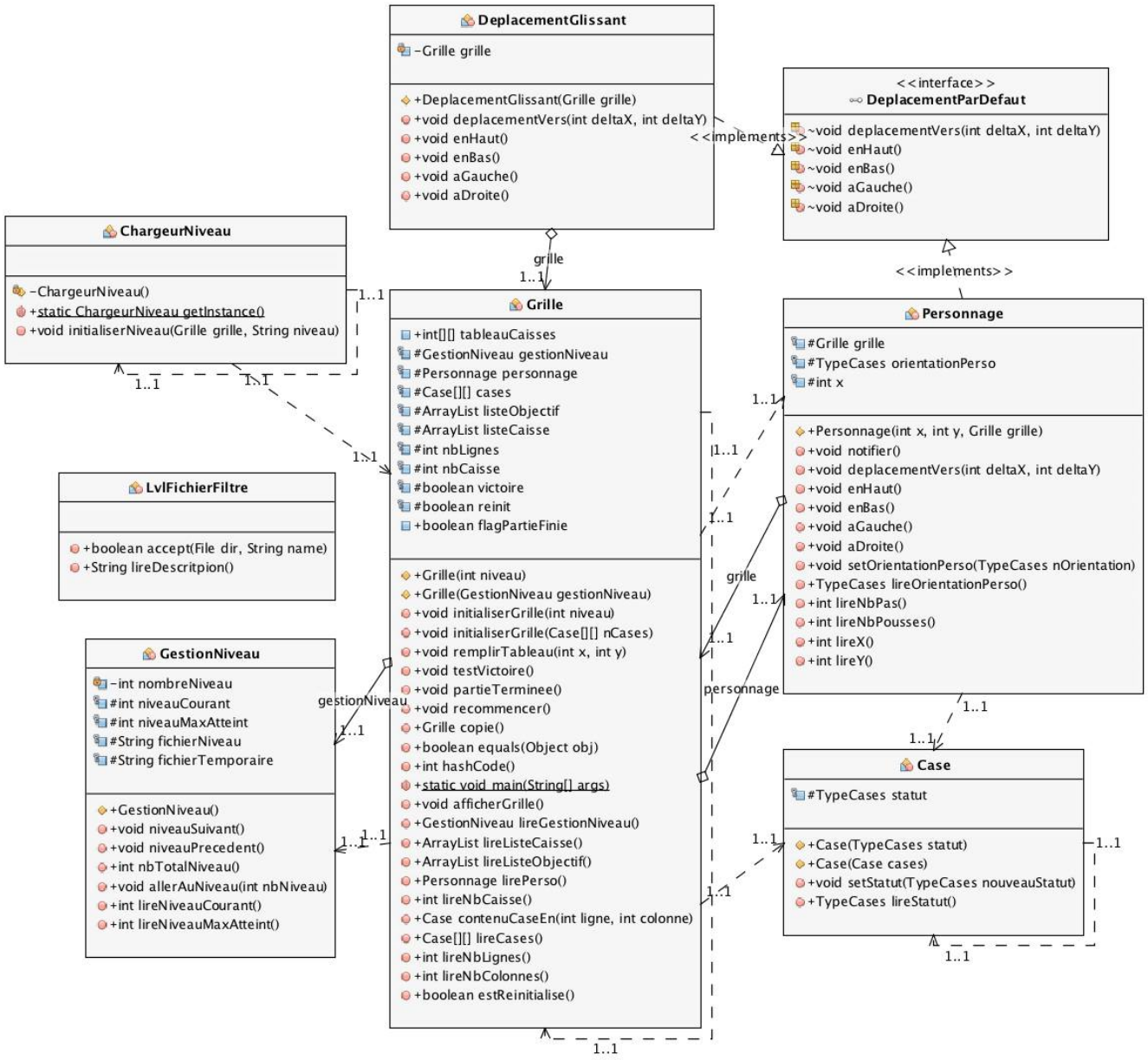
Présentateur :

Partie communicant avec les deux autres pour traduire et transmettre les commandes de l'utilisateur envoyée de la vue vers le modèle, elles communiquent également pour formater et afficher les données du modèle dans la vue.
Le principe est de découpler la vue et le modèle, en utilisant le présentateur comme intermédiaire.

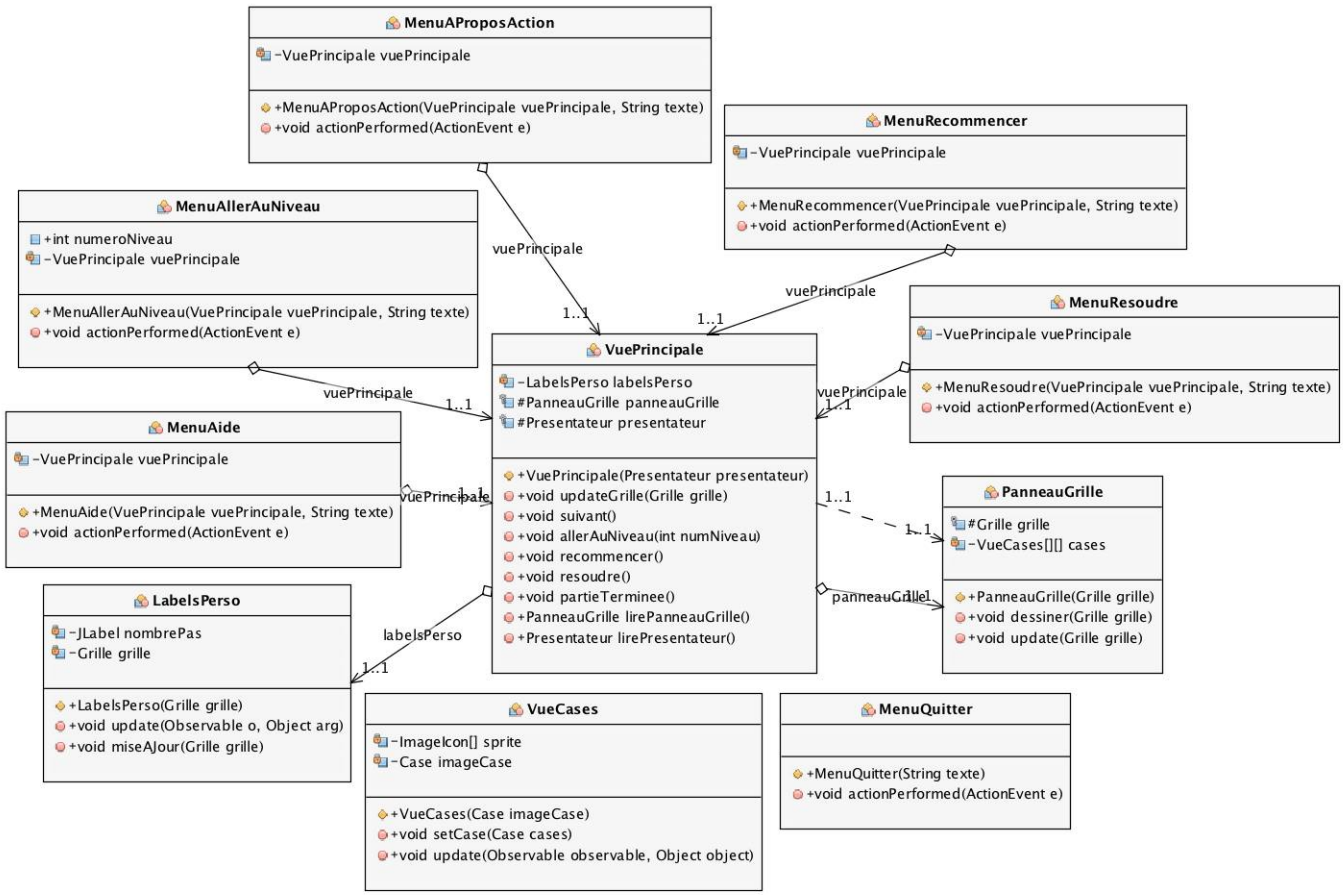
En effet, lorsque l'utilisateur joue à notre jeu, il démarre le Présentateur, qui instancie un niveau, puis une grille sur ce niveau et enfin la vue associée à la grille du modèle. A chaque fois que l'utilisateur effectue une action, il modifie le modèle (représentant notre classe Grille dans le Présentateur) suite à quoi, le présentateur signale à la vue principale que le modèle a changé, elle-même va signaler à la vue de notre grille de se modifier en conséquence.
Dès lors que le modèle indique que la partie est finie (c'est-à-dire victoire de l'utilisateur, changement de niveau, ou réinitialisation du niveau) la vue indique au présentateur de créer un nouveau modèle du niveau en conséquence.
Pour l'instant, notre présentateur possède une boucle infinie ne s'arrêtant que si il y a eu victoire de l'utilisateur ou si la méthode « recommencer » à été utilisée, permettant ainsi de réinitialiser le niveau courant.



2.4.2 Diagramme Modele



2.4.3 Diagramme Vue



3 Intelligence Artificielle

3.1 Qu'est-ce que l'intelligence artificielle ?

3.1.1 Définition

Le terme « intelligence artificielle », créé par John McCarthy, est souvent abrégé par le sigle "I.A." (ou "A.I." en anglais, pour Artificial Intelligence).
Il est défini par l'un de ses créateurs, Marvin Lee Minsky, comme "la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains car elles demandent des processus mentaux de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique" Nous y trouvons donc le côté "artificiel" atteint par l'usage des ordinateurs ou de processus électroniques élaborés et le côté "intelligence" associé à son but d'imiter le comportement.
Cette imitation peut se faire dans le raisonnement, par exemple dans les jeux ou la pratique des mathématiques, dans la compréhension des langues naturelles, dans la perception : visuelle (interprétation des images et des scènes), auditive (compréhension du langage parlé) ou par d'autres capteurs, dans la commande d'un robot dans un milieu inconnu ou hostile. Même si elles respectent globalement la définition de Minsky, il existe un certain nombre de définitions différentes de l'IA qui varient sur deux points fondamentaux :
Les définitions qui lient la définition de l'IA à un aspect humain de l'intelligence, et celles qui la lient à un modèle idéal d'intelligence, non forcément humaine, nommée rationalité.
Les définitions qui insistent sur le fait que l'IA a pour but d'avoir toutes les apparences de l'intelligence (humaine ou rationnelle), et celles qui insistent sur le fait que le fonctionnement interne du système d'IA doit ressembler également à celui de l'être humain et être au moins aussi rationnel.

3.1.2 Historique

L'origine de l'intelligence artificielle se trouve probablement dans l'article d'Alan Turing "Computing Machinery and Intelligence" (Mind, octobre 1950), où Turing explore le problème et propose une expérience maintenant connue sous le nom de test de Turing dans une tentative de définition d'un standard permettant de qualifier une machine de "consciente".
Il développe cette idée dans plusieurs forums, dans la conférence "L'intelligence de la machine, une idée hérétique", dans la conférence qu'il donne à la BBC e programme le 15 mai 1951 "Les calculateurs numériques peuvent-ils penser ?" ou la discussion avec M.H.A. Newman, Sir Geoffrey Jefferson et R.B. Braithwaite les 14 et 23 janvier 1952 sur le thème "Les ordinateurs peuvent-ils penser ?".
On considère que l'intelligence artificielle, en tant que domaine de recherche, a été créée à la conférence qui s'est tenue sur le campus de Dartmouth College pendant l'été 1956 à laquelle assistaient ceux qui vont marquer la discipline.
Ensuite l'intelligence artificielle se développe surtout aux États-Unis à l'université Stanford sous l'impulsion de John McCarthy, au MIT sous celle de Marvin Minsky, à l'université Carnegie-Mellon sous celle de Allen Newell et Herbert Simon et à l'université d'Édimbourg sous celle de Donald Michie. En France, l'un des pionniers est Jacques Pitrat.

3.2 L'algorithme A *

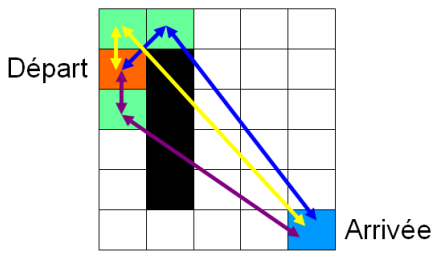
L'algorithme de recherche A* (qui se prononce A étoile, ou A star à l'anglaise) est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final tous deux donnés. De par sa simplicité il est souvent présenté comme exemple typique d'algorithme de planification, domaine de l'intelligence artificielle. L'algorithme A* a été créé pour que la première solution trouvée soit l'une des meilleures, c'est pourquoi il est célèbre dans des applications comme les jeux vidéo privilégiant la vitesse de calcul sur l'exactitude des résultats. Cet algorithme a été proposé pour la première fois par Peter E. Hart, Nils John Nilsson et Bertram Raphael. Il s'agit d'une extension de l'algorithme de Dijkstra de 1959.

Présentation :

L'algorithme A* 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.

Intuition :

Commençons par un exemple de motivation. Nous nous trouvons à l'intersection A et nous voulons nous rendre à l'intersection B dont nous savons qu'elle se trouve au nord de notre position actuelle. Pour ce cas classique, le graphe sur lequel l'algorithme va travailler représente la carte, où ses arcs représentent les chemins et ses nœuds les intersections des chemins.
Si nous faisions une recherche en largeur comme le réalise l'algorithme de Dijkstra, nous rechercherions tous les points dans un rayon circulaire fixe, augmentant graduellement ce cercle pour rechercher des intersections de plus en plus loin de notre point de départ. Ceci pourrait être une stratégie efficace si nous ne savons pas où se trouve notre destination, comme la police recherchant un criminel en planque.
Cependant, c'est une perte de temps si noud connaîssons plus d'informations sur notre destination. Une meilleure stratégie est d'explorer à chaque intersection la première directement qui va vers le nord, car le chemin le plus court est la ligne droite. Tant que la route le permet, nous continuons à avancer en prenant les chemins se rapprochant le plus de l'intersection B. Certainement devrons-nous revenir en arrière de temps en temps, mais sur les cartes typiques c'est une stratégie beaucoup plus rapide. D'ailleurs, souvent cette stratégie trouvera le meilleur itinéraire, comme la recherche en largeur le ferait. C'est l'essence de la recherche de chemin A*. Un labyrinthe simple, où l'algorithme A* ne sera pas très efficace
Toutefois, comme pour tous les algorithmes de recherche de chemin, leur efficacité dépend fortement du problème que nous souhaitons résoudre (c'est-à-dire : du graphe). Ainsi l'algorithme A* ne garantit pas de trouver un itinéraire plus rapidement qu'un autre algorithme, et dans un labyrinthe, la seule manière d'atteindre la destination pourrait être à l'opposé de la position de la destination, et les nœuds les plus proches de cette destination pourraient ne pas être sur le chemin le plus court, ce qui peut coûter beaucoup de temps de calcul.



3.3 Notre IA

Nous possédons un package nommé « ia », contenant certaines classes abstraites ou interfaces, permettant ainsi d'implémenter n'importe quel type d'algorithme à n'importe quel type de problème. En effet, après avoir discuté avec notre client, nous avons implémenté un code en types génériques.

Cependant, la gestion d'exception n'ayant pas encore était implémentée, faute de temps, nous supposerons que l'utilisateur ne se trompe pas de type pour que la compilation ne retourne pas d'erreurs.

Concernant les interfaces génériques, nous possédons :

— HeuristiqueAbstrait <E>
— ProblemeAbstrait <E, A>
— AlgorithmeAbstrait <E, A>

Dans notre cas :

— Le type E représente une grille, un état du jeu.
— Le type A représente une action applicable au jeu, un déplacement du personnage.

HeuristiqueAbstrait <E> :

Interface permettant d'ajouter une nouvelle heuristique admissible à appliquer sur un état E. Nous avons pour l'instant 2 méthodes heuristiques. Une permettant de calculer la distance Manhattan entre une caisse et un objectif, et une autre permettant de calculer la distance Manhattan entre le personnage et une caisse.

ProblemeAbstrait <E, A> :

Définit 4 méthodes, dont 3 sont des accesseurs permettant de : savoir si un état E est terminal (victoire du personnage.), faire la lecture de l'état courant du problème et enfin connaître la liste des actions de l'état courant du problème.
Sa 4ème méthode est « E successeur(E état, A action) », comme son nom l'indique, elle renvoie l'état successeur d'un état donné ayant une certaine action effectuée sur celui-ci.

AlgorithmeAbstrait <E, A> :

Possède une unique méthode
"ArrayList<E> resoudre(ProblemeAbstrait <E, A> probleme, HeuristiqueAbstrait<E> heuristique) ;"
qui permet de résoudre un problème donné avec une certaine heuristique, à condition que cette heuristique soit « admissible ».

PASSONS MAINTENANT AUX CLASSES CONCRÈTES DE NOTRE IMPLÉMENTATION D'INTELLIGENCE ARTIFICIELLE ; NOUS POSSÉDONS 4 CLASSES CONCRÈTES, ET 1 TYPE ÉNUMÉRÉ FORTEMENT TYPÉ :

ActionSokoban :

Celle-ci est la plus simple, c'est notre type énuméré qui représente tout simplement les 4 actions possibles d'un personnage sur une grille à savoir, les 4 déplacements : Haut, Bas, Gauche et Droite.

ProblemeSokoban :

Cette classe, implémente l'interface ProblemeAbstrait <Grille, ActionSokoban>, un problème s'instancie simplement à l'aide d'une grille, et lors de son instanciation, celui-ci créé un EnumSet « actions » de toutes les valeurs contenues dans le type énuméré « ActionSokoban ».
Puis cette classe affecte à la variable « listeActions » étant une List<ActionSokoban>, une nouvelle ArrayList de « actions » :

"EnumSet < ActionSokoban > actions = EnumSet.allOf(ActionSokoban.class) ;"
"ProblemeSokoban.listeActions = new ArrayList(actions) ;"

Cette classe redéfinit la méthode principale appelée « successeur », cette méthode prend en argument une grille à un certain état, et une action.

Lors d'un appel à « successeur » la grille passée en paramètre est copiée, afin qu'elle ne soit pas modifiée directement, puis nous appliquons l'action du personnage sur la copie de la grille et nous retournons la copie modifiée. Cette méthode permet donc de passer d'un état à un autre sans modifier l'état initial.
De plus, ProblemeSokoban redéfinit la méthode permettant de savoir si une grille est terminale ou non (si le personnage à réussi à placer toutes les caisses sur les objectifs.).

HeuristiqueSokoban :

Celle-ci est notre classe qui implémente HeuristiqueAbstraite, nous avons décidé d'utiliser une heuristique permettant de connaitre la plus courte distance Manhattan entre le personnage et les caisses, additionnée à la plus courte distance Manhattan entre les caisses et les objectifs.

La distance Manhattan représente le nombre de cases, en horizontal et en vertical entre la source et la destination.
Malheureusement, nos heuristique ne sont pas réellement au point.

En effet dès lors qu'il existe plusieurs caisses (donc plusieurs objectifs) sur une grille, notre heuristique prend en compte seulement la dernière caisse traitée et le dernier objectif, nous n'arrivons pas à stocker toutes les valeurs de coordonnées (x,y) des différentes caisses et objectifs présents sur la grille.

GrilleAvecValeur :

Cette classe n'est en réalité rien de plus qu'une grille « possède un attribut Grille grille », avec en plus des valeurs utilisées dans les différents algorithmes de recherche de plus court chemin à savoir la fonction "g", où " g(etatCourant)" est le coût du meilleur chemin connu pour atteindre etatCourant depuis l'état initial ; étant donné que comme nous avons un terrain toujours identique, le coût d'un déplacement du personnage est constant et vaut 1. Donc pour passer d'une grille à une autre en faisant un déplacement de personnage nous faisons simplement "g+1".

La seconde valeur est nommée "h" et représente la fonction heuristique (prédictions), cette fonction nous permet d'évaluer un coût vers la destination INFERIEUR au coût réel (encore inconnu).
A ce titre, A* est un algorithme optimiste.
Notre "h" représente donc un entier, d'estimation des distances entre la grille initiale et le grille terminale.
Donc pour trouver la valeur de la variable "h" nous devrons additionner la valeur de retour des deux méthodes de calcul de distance d'une instance de notre classe vu précédemment "HeuristiqueSokoban" sur une grille à un moment donné.

Cette classe servant notamment de référence de comparaison, c'est pourquoi elle implémente l'interface "Comparable" de l'API Java, nous redéfinissons donc sa méthode "compareTo(Object obj)".

Nous comparons donc deux grilles avec valeurs entre la valeur de l'heuristique de la grille actuel avec celle de la grille « obj » passée en argument, si les valeurs des deux « h » sont égales nous retournons 0, si la valeur du « h » de la grille actuel est supérieure à la grille « obj » nous retournons 1, sinon nous retournons -1.

Ceci nous permet de trier l'ordre de priorité des différentes GrilleAvecValeur dans une PriorityQueue de l'algorithme A*.

A propos de la PriorityQueue, nous avons eux beaucoup de problèmes sur le fait que la PriorityQueue par défaut compare l'égalité des pointeurs, or nous voulions comparer l'égalité de deux grille, cet-à-dire l'égalité des positions du personnage et des caisses.

Nous avons donc du redéfinir les méthodes "equals(Object obj)" et "hashCode()" de la class "Object" dont toutes les classes hérite par défaut.

La méthode "equals(Object obj)" test si l'objet passé en argument est une instance de GrilleAvecValeur. Si l'argument n'est pas une instance de GrilleAvecValeur, nous retournons "False ;" sinon nous retournons la valeur du booléen de l'attribut grille actuel "equals" (appel à la méthode "equals(Object obj)" redéfinit dans la classe Grille du modèle, d'où l'intérêt du cast pour empêcher une exception) à l'attribut "grille" de la grille avec valeur passée en paramètre :

"return this.grille.equals(((GrilleAvecValeur) obj).grille) ;"

Puis la méthode "hashCode()", nous hashons simplement l'attribut this.grille car seulement celui-ci doit être unique : hash = 29 * hash + Objects.hashCode(this.grille) ; où hash vaut initialement 5.

Aetoile :

Passons maintenant à notre classe principale de l'algorithme de résolution du plus court chemin, l'algorithme A étoile (A star).

Celui-ci implémente donc l'interface "AlgorithmeAbstrait <E, A>", puis s'instancie par le biais d'un état E, et d'une List<A>, où l'état E est une grille, et la liste<A> est la ArrayList d'ActionSokoban vu précédemment.

Notre classe possède donc deux accesseurs : un sur la grille et un sur la liste d'action, nous redéfinissons la méthode considérée comme principale d' "AlgorithmeAbstrait" nommée

"public ArrayList<E> resoudre(ProblemeAbstrait<E, A> probleme, HeuristiqueAbstrait<E> heuristique)".

Cette méthode retourne une ArrayList<E> qui est une liste contenant le chemin de l'état terminal à l'état initial si il existe une solution, ou retourne une liste vide sinon, si c'est le cas cela signifie qu'il n'existe pas de chemin possible de l'état source à l'état terminal.

Décrivons maintenant notre méthode de résolution, nous maintenons deux listes différentes.

Nous avons choisis d'implémenter une file à priorité dite "ouverte" de type PriorityQueue, qui représente les états à explorer, puis une liste doublement chainée dite "fermée" de type LinkedList, qui représente ici les états visités considéré prioritaire par la PriorityQueue.

Premièrement, nous instancions un état initial appelé "etatInitial", un état source qui représente l'état courant du problème et non la grille de départ instanciée lors du démarrage d'un niveau.

Cela permet en effet à un joueur lorsqu'il s'est déjà déplacé, de demander un "indice" à l'algorithme permettant aisni à l'utilisateur de savoir si oui ou non il existe une solution à partir de l'état de la grille sur laquelle il se trouve.

Ensuite, nous instancions une grille avec valeur sur la grille de l'état initial précédent, avec un coup "g" de 0, et appelons "h" la valeur où :

"int h = ((heuristique.distanceManhattanPersonnageCaisse((E) etatInitial))
+ (heuristique.distanceManhattanCaisseObjectif((E) etatInitial))) ;".

Puis nous ajoutons cette grille avec valeur à la liste ouverte.

Nous commencons alors la boucle de recherche, tant que la file à priorité n'est pas vide, nous récupèrons la tête de liste, appelons le "etatCourant" il représente l'état avec la fonction F minimale, cet-à-dire avec la somme de "g" et de "h" la plus petite.

Nous supprimoms etatCourant de cette liste ouverte et nous l'ajoutons à la liste fermée.

Nous testons si le problème de l'état courant est terminal :

Si oui :

Nous récupèrons alors la dernière valeur de la liste fermée et nous l'assignons dans une variable de type Grille appelons le "noeudCourant", ensuite tant que "noeudCourant" est différent de la grille de départ, nous l'ajoutons à l'ArrayList "chemin", cette liste nous permettra plus tard de reconstruire le chemin de la source à l'état final. Puis nous assignons le père du noeudCourant au nouveau noeudCourant. Une fois terminé, nous retournons la liste "chemin" possédant ainsi le trajet du personnage de l'état initial à l'état terminale.

Sinon :

Pour chaque actions du problème qui sont applicable à l'état courant, nous génèrons les 4 successeurs de l'état courant, lorsque nous faisons un appel à la méthode « successeur » de la classe ProblemeSokoban. Ensuite nous mettons à jour notre heuristique en calculant les nouvelles distances sur les 4 états successeurs.

Nous incrémentons la fonction "g" de 1 puisque nous avons un coût uniforme peu importe le déplacement.

Puis, nous instancions 4 nouvelles grilles avec valeur en prenant comme paramètres les grilles successeurs, puis les nouvelles valeur de "g" et de "h".

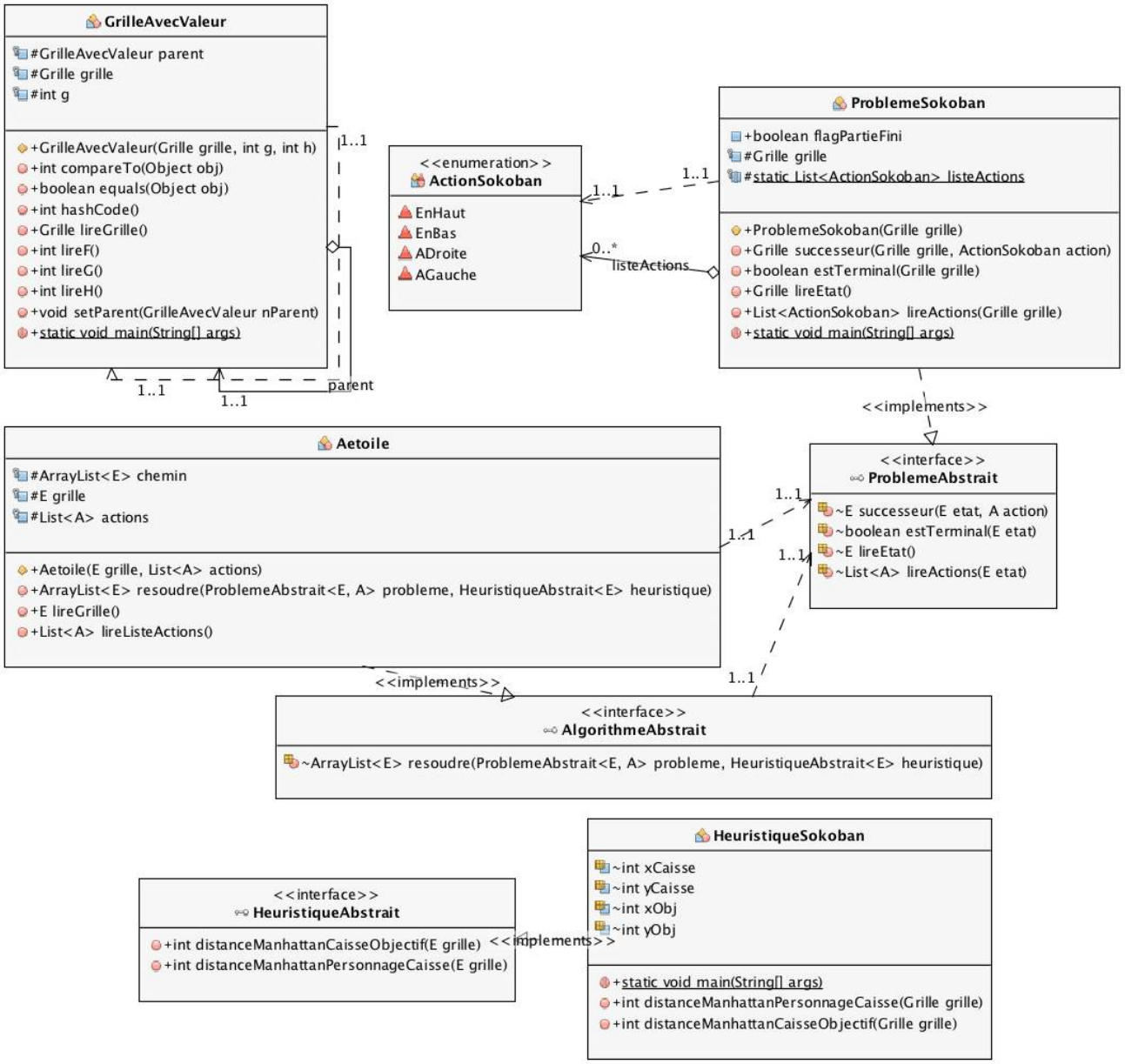
Enfin nous réalisons plusieurs tests, d'abord si une des grilles avec valeur successeurs est égale à l'état courant, nous continuons. Nous vérifions ensuite que la liste ouverte et la liste fermée ne contiennent pas un des états successeurs (ici, comme vu précédemment, nous ne comparons pas les pointeurs, mais bien les positions du personnage et des caisses dans la grille !) nous assignons à la fonction "g" de la grille avec valeurs successeurs la nouvelle valeur de "g".

Puis nous ajoutons ce successeur à la liste ouverte.

Sinon, si la valeur de la fonction "g" du successeurs est supérieure au nouveau "g", alors nous assignons au "g" du successeur la nouvelle valeur, enfin nous vérifions que la liste fermée contient bien cet état successeur, ensuite nous le supprimons de la liste fermée, et nous l'ajoutons à la liste ouverte.

Après tout cela, si l'algorithme ne trouve pas de solution, c'est qu'il n'en n'existe pas, notre méthode ne retourne alors pas d'erreur mais une liste vide.

3.4 Diagramme IA



4 Description du déroulement

4.1 Grandes étapes et plannings

BACKLOG				
N°	Story	Priorité	Lots	Etat
1	L'utilisateur peut démarrer une partie grâce à une interface graphique	1	1	Terminé
2	L'utilisateur peut se déplacer à l'aide des touches du clavier	2		Terminé
3	L'utilisateur peut choisir un niveau plus ou moins élevé	4		Terminé
4	Le jeu informe l'utilisateur de son échec ou de sa réussite	3		Terminé
5	L'utilisateur peut observer une partie, avec un niveau d'intelligence artificielle basique, se jouer seule	5	2	Terminé
6	L'utilisateur peut observer une partie, avec un niveau d'intelligence artificielle élevé, se jouer seule	6	3	**En cours**
7	L'utilisateur peut jouer une partie avec les règles des « boites glissantes »	7	4	A venir
8	L'utilisateur peut observer une partie où une intelligence artificielle fonctionne sur le jeu avec les règles des « boites glissantes »	8		A venir
9	L'utilisateur peut générer un niveau aléatoire et jouer lui-même, ou bien faire jouer une intelligence artificielle	9	5	A venir

4.2 Problèmes rencontrés et solutions apportées

Tout d'abord nous avions eux beaucoup de problèmes sur le fait que le code d'un jeu Sokoban récupéré en open source était réellement mal codé, et non orienté objet.
C'est pourquoi nous avons du programmer tout un jeu de Sokoban en utilisant les patrons de conception appropriés ainsi que rendre le code plus extensible, plus modulaire.
Cela nous a en effet pris du temps, nous avons donc eu du retard sur nos jalons initiaux. Cependant grâce aux cours sur la programmation orientée-objet et aux expériences acquises au cours de nos études en génie logiciel nous avons pu résoudre ce problème de la meilleure des manières possible.
Notre second problème majeur fut l'implémentation de l'algorithme A étoile. En effet notre client nous ayant conseillé de programmer la partie intelligence à l'aide des types génériques, nous avons dû lire les documentions et apprendre à les utiliser pour ensuite les appliquer à notre jeu de Sokoban.

Une fois le problème des types génériques résolus, la traduction du pseudo-code d'un algorithme A étoile en java a été plus difficile que prévu étant donné la simplicité du pseudo-code et du fonctionnement de l'algorithme.

Le fait que l'algorithme possède 2 listes, dont une file à priorité : la PriorityQueue en java, nous avons donc redéfinit la méthode « compareTo() » pour pouvoir comparer 2 différentes grilles avec valeur et les placer ainsi dans la liste en fonction de la meilleurs grille avec valeur.
Cependant nous avons mit beaucoup de temps à savoir que la méthode « contains() » de la PriorityQueue compare les pointeurs, et non l'égalité des coordonnées des différents objet sur cette grille, ce que nous voulions comparer a la base.

Après moult recherches sur les PriorityQueue, et une discussion avec notre client, nous avons compris que nous devions redéfinir les méthodes « equals() » et « hashcode() » de la classe Object, pour pouvoir ainsi comparer les coordonnées du personnage et le statut des cases de la grille.
Nous avons du d'abord redéfinir ces 2 méthodes dans la classe Grille du modèle, puis ensuite dans la classe GrilleAvecValeur du package « ia » permettant de comparer 2 grilles du point de vue du modèle.

Pour revenir sur la partie de l'implémentation de l'algorithme A étoile, notre réel problème à était lors de la génération des 4 successeurs d'un état. En effet, lors de l'appel de la méthode « successeur() » de la classe ProblemeSokoban, nous devions faire une copie de la grille passée en paramètre.
0Nous avons eu alors un soucis de copie profonde (« deepcopy »), c'est un problème récurrent en programmation. Au départ nous copions seulement les pointeurs sur la grille et non tous les objets eux-même contenu dans la grille, nous avions essayé d'utiliser la méthode clone de l'interface Cloneable, cependant le même problème était toujours présent.
C'est pourquoi nous avons créé une méthode nommée « copie » qui permet la copie en profondeur de chaque différents types de cases et du personnage présent sur la grille actuel, puis assignes toutes ces valeurs dans une nouvelle instance de grille appelée « copieGrille » et retourne la grille copiée. Permettant ainsi d'effectuer les actions du personnage non pas sur la grille actuel (car nous ne pourrions pas revenir en arrière) mais sur une copie de celle-ci.
Le problème encore en cours est qu'une fois la résolution du problème par l'algorithme A* avec le chemin le plus court, nous n'arrivons pas à le relié à la vue permettant ainsi à l'utilisateur d'observer le personnage pas à pas résoudre le niveau courant.

4.3 Gestion du projet : organisation et répartition des tâches

4.3.1 Forge SVN

Il s'agit d'un système permettant de sauvegarder les différentes versions du programme.
Il permet aussi de suivre l'évolution et les modifications effectuées par un autre utilisateur. Ce système comprend de nombreuses autres fonctionnalités, comme par exemple de faire des demandes aux autres utilisateurs du projet.
C'est donc un excellent moyen de gestion de projets de groupe.

4.3.2 Organisation et répartition des tâches

Avant d'avoir commencé à répartir les tâches, nous nous sommes d'abord fixé les objectifs que nous avions décidé d'atteindre bien qu'ils n'aient pas tous été atteints.

Nous avons fait une répartition selon les différents modules, leurs complexités, et les différentes compétences de chacun.

Chacun de nous devait s'occuper d'une ou deux parties, indépendamment les unes des autres.

5 Bilan et Conclusion

Notre projet fut très intéressant à mettre en place et notre équipe a dans l'ensemble bien fonctionné.

Du point de vue technique, ce projet nous a permis d'enrichir nos connaissances sur le java et l'intelligence artificielle et nous a fait découvrir des outils très intéressants qui se sont avérés très utiles pour la conception et la réalisation du projet.

Nous avons ainsi pu développer notre technique de recherche d'informations ainsi que notre esprit d'initiative et notre esprit d'équipe.

6 Sources et Documentations

https ://www.greyc.fr/fr/mad

http ://www.gamedev.net/page/index.html

http ://www.redblobgames.com/pathfinding/a-star/implementation.html