



UNIVERSITÉ
CAEN
NORMANDIE

Puzzle Edition

Rapport de projet

VINCENT Leo 21805239

BELLEBON Alexandre 21808613

LEROY Clementine 21800424

DEROUIN Auréline 21806986

Table des matières

1	Présentation du projet	1
A	Présentation de l'application	1
2	Architecture du projet	2
A	Arborescence du projet	2
3	Aspects techniques	3
A	Librairie piecesPuzzle	3
B	Modèle : Le plateau	3
C	Controleur	4
i	Play	5
ii	Vue	6
iii	File	6
4	Conclusion	7
5	Ressources utiles et sources utilisés	8

1 Présentation du projet

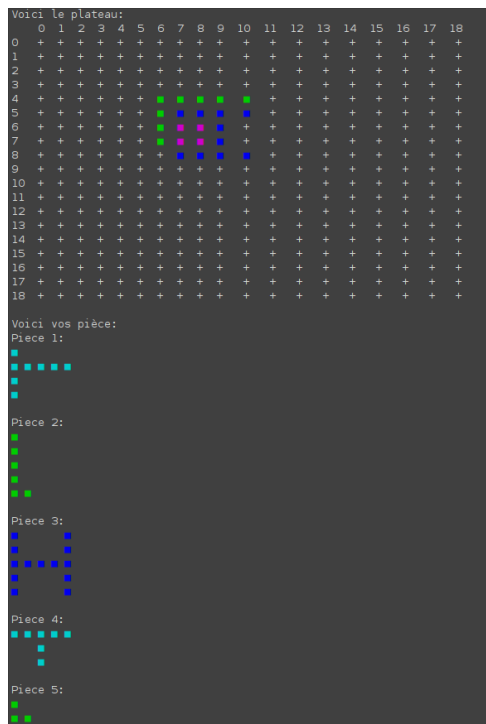
A Présentation de l'application

Puzzle Edition est une application de jeu, dotée d'une interface graphique, qui consiste à assembler des formes de sorte qu'elles occupent le moins de place possible. Cela s'apparente au tetrax, mais avec des modalités différentes. D'ailleurs, le joueur retrouvera certaines pièces de ce jeux. Ces pièces peuvent être placées, déplacées ou supprimées du plateau ou encore tournées.

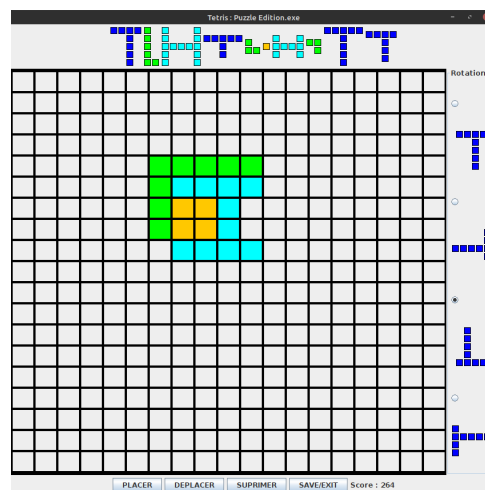
Le joueur peut créer ou charger une partie. La difficulté de la partie est mesurée en fonction de la taille du plateau choisie ainsi que le nombre de pièces générées. En effet, plus il y a de pièces générées, plus la partie est difficile, mais plus le score sera élevé. C'est pour cela qu'à chaque partie, il est possible soit de demander à obtenir une nouvelle configuration de départ, soit de charger une configuration déjà créée et sauvegardée.

Le but du joueur est de minimiser l'espace occupé par l'ensemble des pièces. Plus précisément, la fonction d'évaluation sera l'aire du plus petit rectangle (parallèle aux axes) recouvrant l'ensemble des pièces. Lorsque le joueur considère avoir terminé (ou lorsque le nombre maximum d'actions autorisées est atteint), il clique sur un bouton et son score est alors calculé.

Une option permet de faire jouer l'ordinateur sur une nouvelle partie, ou une partie déjà sauvegardée, permettant ainsi de comparer son score avec celui de l'ordinateur.



Vue console



Vue graphique

2 Architecture du projet

A Arborescence du projet

src : Arbo

3 Aspects techniques

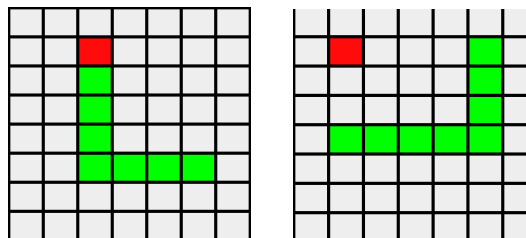
A Librairie piecesPuzzle

Pour la conception de ce projet, une librairie a été créée, référençant ainsi toutes les pièces possibles. Cette librairie comporte 4 pièces différentes. Ces quatre pièces ont toutes une grille créée à partir de la rotation définie. Cette grille et cette rotation est créée à l'aide de la méthode *createGrid()* qui prend en paramètre un numéro qui correspond à la rotation dans le sens horaire. Cette méthode change la largeur/longueur de la pièce, en fonction de celle donnée de la pièce de départ, permettant ainsi la création d'une nouvelle grille en prenant en compte la rotation. Chaque pièce a une grille composée de valeur boolean, permettant ainsi de créer la pièce voulue (valeur "true" pour indiquer que la case est pleine), en fonction de sa rotation, largeur, longueur. Elle est aussi composée de coordonnées, permettant, en fonction du jeu, de pouvoir situer la pièce dans le plateau.

B Modèle : Le plateau

La librairie piecesPuzzle est utilisée par notre modèle. PlateauPuzzle permet de créer un plateau ainsi que d'utiliser les pièces pour les ajouter, supprimer... Pour la création de notre plateau, nous avons utilisé une HashMap de coordonnées (ArrayList de Integer) en clé et d'une pièce, s'il y en a une, ou "null" en valeur. Cela permet donc d'avoir une lisibilité absolue sur le plateau, ainsi que sur les différentes pièces présentes, sans être obligé d'avoir une hashmap triée. La HashMap est directement créée dès l'appel à cette classe.

Le plateau peut donc créer, placer, supprimer, déplacer, tourner la pièce sur le plateau. Pour la plupart des différentes méthodes, un appel de la méthode *validePlacement()* est nécessaire afin de savoir si la pièce, reçue par la méthode, peut être placée dans la HashMap. Pour cela, dès que la valeur de la grille parcourue est de boolean *true*, elle regarde si la valeur de la clé coordonnées du HashMap (en fonction des coordonnées reçues par la méthode). Cela permet donc de pouvoir placer des pièces côte à côte, en ne plaçant que les cases "true" de la pièce dans la HashMap. Les pièces ont donc des cases "true" et "false" en fonction de comment elles sont créées. Nous avons choisi une case de référence afin d'avoir les coordonnées de la pièce s'il faut, par exemple, la déplacer ou la faire tourner, ou encore, si on sauvegarde la partie. Cette case de référence, nous l'avons choisie en haut à gauche de la pièce. Sauf que cela posait problème si le joueur voulait poser la pièce à l'endroit où il a cliqué/indiqué les coordonnées (voir image??). En effet, la case est toujours en haut à gauche, même si cette case est en false.



Légende :

Rouge : coordonnées où le joueur veut placer la pièce, et indique aussi la case de référence

Vert : pièce L en rotation 0 (img 1) et rotation 3 (img 2)

Pour régler ce problème, nous avons rajouté un changement de coordonnées, dès que le joueur indiquait, par un clic ou par indication de coordonnées en console, la case où il voulait placer la pièce.

Tant que les coordonnées Y de la grille de la pièce était false, les coordonnées (*coo*) en position Y était diminuer de -1, permettant ainsi de placer la grille de la pièce a partir des coordonnées indiqué -1, si la case était "true" .

Algorithm 1: finished() :boolean

```

1 posY ← 0;
2 xx ← 0;
3 yy ← 0;
4 while !piece.getGrid()[xx][yy + posY] do
5   |   coo.set(1, coo.get(1) - 1);
6   |   posY ← posY + 1;
7 end
8 return true
  
```

Cela permet de changer la "case", donc les coordonnées indiquer par le joueur, jusqu'à ce que la valeur des coordonnées de la grille de la pièce est égal a "true".

Pris en compte des coordonnées	Boucle while	Application de l'ajout ou déplacement de la pièce

Légende :

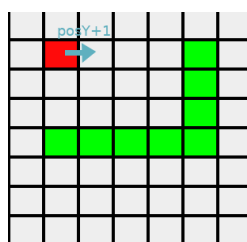
Rouge : Case de référence et variable "coo" qui, sur l'image 1, sont les coordonnées renseigner par le joueur (modifier par la suite dans la while)

Flèche : coo modifier dans la while

Vert/jaune : Pièce "fantome" où la pièce devrait être normalement sans cette while

Cela permet de changer la "case", donc les coordonnées indiquer par le joueur, jusqu'à ce que la valeur des coordonnées de la grille de la pièce est égal a "true".

Pour cela, pendant que la valeurs des coordonnées Y de la variable *coo* diminuent, le parcours de la grille se fait jusqu'à ce que elle rencontre la case plus en haut a gauche possible de la pièce. Donc elle ajoute +1 a la variable *posY*.



Légende :

Rouge : Case de référence

Flèche : vérification de la prochaine case si sa valeur est "true"

Vert : pièce L rotation 3

C Controleur

Le controleur est séparés en plusieurs classes :

classe PlayMenu : Permet de choisir entre la vue console et la vue graphique.

classe Play : Fait appel aux méthodes du modèle, et appels les méthodes de "choix" en fonction de la vue/joueur actuel.

classe **InterfacePlay** : Méthodes communes aux choix des joueur : ia ou joueur.

classe **PlayJoueur** : Méthodes de choix pour joueur Console, grâce aux Scanner.

classe **PlayIa** : Méthodes de choix pour ia.

enum **EnumAction** : Enumère les actions possibles en jeu.

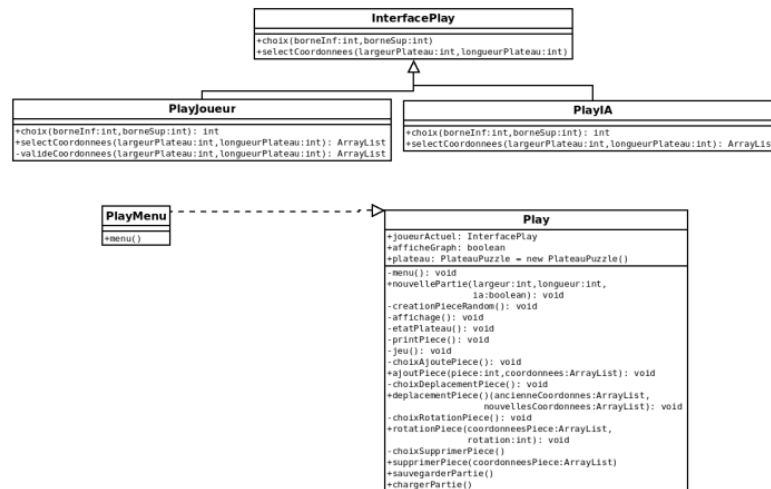


FIGURE 3.1 – Controleur

Cette séparation permet simplifier et séparer les choix de jeu. Cela évite la redondance de code.

i Play

En vue console, les choix se font grâce à la variable *joueurActuel* qui est une instance de *InterfacePlay*. Cela permet, en fonction du choix du joueur (option ia ou non), d'appeler les mêmes méthodes de choix, permettant ensuite d'appeler les méthodes communes à toutes les vues pour accomplir l'action prévu (placer, déplacer...). Tout cela est contrôlé par la méthode *jeu()*.



FIGURE 3.2 – Explication choix en fonction du joueur - vue console

En vue graphique, les choix se font grâce à des actions sur des cases/boutons, détectées par des Observateur/Observable voir...partie graph

Joueur Ia

Une méthode *jeuIa()* est appelée dès que le joueur choisit de regarder l'ia jouer une partie chargée ou une nouvelle partie (en vue graphique ou console). Cette option permet donc de savoir la meilleure composition de jeu. Cette méthode permet de lancer des *fourmis* (joueur ia qui joue une partie avec des choix random et guidés) afin de faire un certain nombre de parties. Le meilleur score et le meilleur plateau est enregistré. Cela permet donc d'avoir plusieurs possibilités aléatoires, et d'essayer de trouver la meilleure composition possible. Cela évite donc de n'avoir qu'une possibilité qui regarde jusqu'à une certaine profondeur.

La fourmis appelle donc la méthode *jeu()*, commune si le joueur actuel est le joueur en vue console. Cette méthode permet de choisir l'action que le joueur va effectuer. C'est un choix, donc cette méthode appelle l'attribut *joueurActuel*, permettant de faire un choix random pour l'ia. Pour éviter d'avoir un simple choix aléatoire où toutes les actions ont le même niveau de priorité, une liste d'actions a été créée avec plusieurs mêmes actions à l'intérieur, permettant de signifier qu'une action pour "avancer" (placer et déplacer) dans le jeu est plus importante que de "rester sur place" (rotation dans le plateau et rotation des pièces à jouer) et est plus importante qu'une action qui "recule" l'avancement du jeu (supprimer une pièce). Une fois l'action choisie, d'autres méthodes de choix sont appelées en fonction du choix de l'action.

Pour les actions pour "avancer" dans le jeu, les choix ne sont pas de simple random. En effet, si la fourmis a choisi l'action de PLACER ou DEPLACER une pièce, la méthode *choixDeplacement()* ou *choixAjout()* est appelée. Ces deux méthodes prennent en compte la largeur, la longueur et le plateau. Pour *choixDeplacement()*, l'ia choisit d'abord une pièce au hasard sur le plateau, grâce à la méthode *selectPiece()* (méthode commune de l'InterfacePlay) qui sélectionne une pièce dans la liste des pièces à placer. Les coordonnées récupérées sont les coordonnées de la case de référence de la pièce (cf : voir ??). Une boucle while, presque similaire à ici¹, est donc nécessaire pour avoir les coordonnées qui auraient pu être cliquées ou indiquées par un véritable joueur. Une fois les coordonnées obtenues, la pièce est supprimée afin d'exercer une simulation de possibilité sans avoir une pièce sur ces cases dans le plateau. À partir de là, *choixDeplacement()* exerce comme *choixAjout()*. Elles créent une copie du plateau actuel, puis parcourent tout le plateau à l'aide des coordonnées placées en clé du HashMap. Si la coordonnée est vide et au moins une pièce est située à 1 ou 2 cases autour de celle-ci, alors un test de placement est fait (grâce à la méthode *validePlacement()*^B). Si ce test est réussi, un deuxième test est nécessaire : faire un ajout de la pièce aux coordonnées et calculer le score grâce aux méthodes dans la classe *PlateauPuzzle*. Si le score est supérieur au score du test précédent, une sauvegarde du score et des coordonnées est faite. Sinon, cela passe au test pour les coordonnées suivantes. Le *validePlacement()* est déjà appelé dans la méthode d'ajout ou de déplacement, mais cela diminue de temps d'exécution en vérifiant le placement avant d'appeler la méthode d'ajout qui vérifie le placement et qui place si c'est possible.

Pour les autres actions (Rotation, Supprimer), le choix est défini en aléatoire.

Une fois que toutes les fourmis ont fini leurs parties, *afficheIa()* affiche le plateau ayant obtenu le meilleur score. L'affichage déplace d'abord les pièces déjà placées sur le plateau initial (si c'est une partie chargée), puis place ensuite les pièces à jouer en fonction du plateau obtenu par les fourmis. Pour cela, elle utilise les méthodes communes pour effectuer des actions.

joueur Graphique

Si la vue graphique est choisie, une *InterfaceGraphique*, une *MouseCliquer* ainsi qu'un *ActionGraphique* sont créés à partir du constructeur de la classe *Play()*, permettant ainsi de s'afficher la Vue. Ensuite, la méthode *menuGraph()* est appelée, permettant d'appeler *jeuIa()* ou *jeuVue* en fonction de ce que veut le joueur. Le choix se fait grâce au

ii Vue

iii File

La sauvegarde/chargement du fichier se fait grâce aux Json.

4 Conclusion

Ce projet était intéressant dans l'ensemble. Nous en sommes satisfaits, même si l'application peut encore être améliorée, nous avons une vue console et graphique fonctionnelle ainsi qu'un joueur IA un peu plus performant que de simple choix aléatoire. Nous avons respecté, dans l'ensemble, ce qui nous a été demandé, même si nous n'avons pas implémenté de pattern observateur/observable, nous avons un MVC correct et respecté.

5 Ressources utiles et sources utilisés