Projet développement logiciel
Trax

Adel Mhiri

Novembre 2018

Table des matières

1 Introduction

Inventé en 1980 par le néo-zélandais David Smith, Trax est un jeu de plateau joué et vendu dans de nombreux pays. Les règles de ce jeu sont suffisamment simples pour être apprises en 5 minutes : placez les tuiles adjacentes à celles déjà en jeu de manière à ce que les couleurs des pistes correspondent. L'objectif est d'obtenir une boucle ou une ligne de votre couleur tout en essayant d'arrêter votre adversaire dans sa couleur. Ce qui donne à Trax sa profondeur stratégique, c'est la règle du jeu forcé qui permet (ou même oblige) de jouer plusieurs tuiles dans un tour. Ainsi au travers de ces règles en apparence simples il est possible de faire émerger des situations d'une grande complexité. (voir Annexe 1)
Cette ambiguïté est d'ailleurs des raisons qui peuvent expliquer son succès, amenant certains à le décrire comme "lutte imaginative et divertissante" ou "le meilleur jeu de stratégie jamais inventé".

Trax présente tout de même un inconvénient : il est difficile de le tester gratuitement avec une version en ligne. Dans le cadre de mon projet développement logiciel, je tenterai donc de remédier à cela via la programmation de ce jeu à l'aide de java. Ce rapport vous détaillera les différentes étapes de programmation, de la modélisation à l'implémentassions de l'intelligence artificielle, qui m'ont permis d'arriver au code final.

2 Modélisation et énumération des différents aspects du jeu

Dans un premier temps, je me suis attaché à modéliser chaque facette du jeu afin d le rendre programmable en java.

2.1 les tuiles

Comme annoncé en introduction, le jeu Trax est un jeu de plateau dont le principe est de tracer une boucle en déposant des tuiles adjacentes à celles déjà en jeu de manière à ce que les couleurs des pistes correspondent sur le plateau. Dans la version plateau de base, il n'était utilisé qu'une seule tuile avec une partie recto et verso différente.

FIGURE 1 – Jeu original

En considérants les différentes bouts de chemin qu'il est possible de créer au regard des règles du jeu, on peut différencier 6 tuiles distinctes.

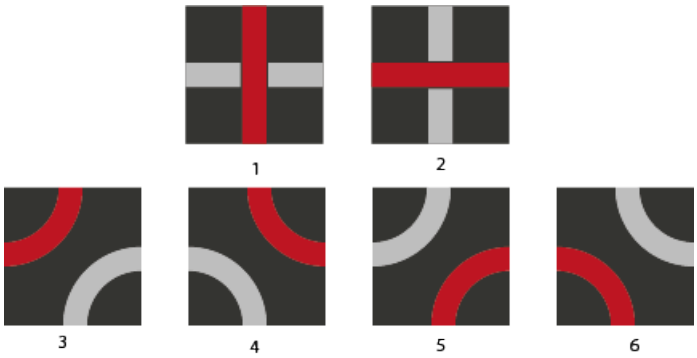


FIGURE 2 – Tuiles accessibles

Au regard de ce champs des possibles, on comprend bien le choix original du constructeur. En effet, les tuiles 2 est obtenue par une rotation de 90 de la tuile 1 et les tuiles 4,5,6 s'obtienne également par rotation de la tuile 3.

Néanmoins tirer partie de cette propriété pour une première modélisation m'a parue trop complexe. J'ai donc préférer m'attarder sur les faces des différentes tuiles. Chaque tuile étant un carré, elle comporte 4 faces qui sont modélisées informatiquement selon le modèle de la Figure 3.

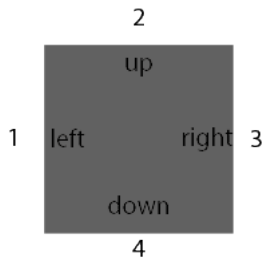


FIGURE 3 – Modélisation des 4 directions

A chaque face est associées une couleur. Les tuiles sont construites de sorte que deux de ses faces sont de la couleur du joueur 1 (dans cette étude on parlera de rouge) et les deux autres de la couleur du joueur 2 (on parlera de blanc). Ainsi si on sait où sont placées les faces rouges, on peut en déduire les faces blanche. Pour éviter une redondance d'information, il est donc possible de marquer les différentes tuiles avec uniquement la position des faces rouges. Cela donne en pratique la transcription suivante :

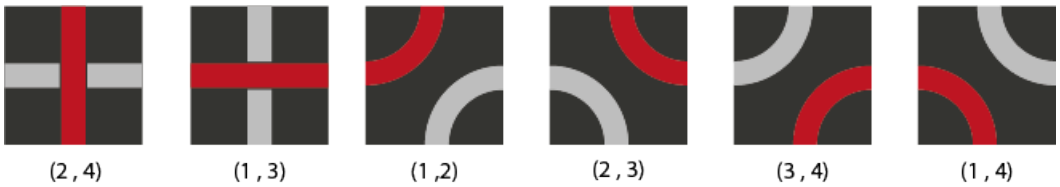


FIGURE 4 – Transcription des 6 tuiles du jeu

Par ailleurs, la taille du plateau dans le jeu originel est potentiellement infinie. Néanmoins comme on se place dans le domaine informatique, il nous faut nous placer dans un espace de dimension fini. Afin d'éviter les possibles match nuls, il a été décidé de faire boucler le plateau. C'est à dire que la case la colonne la plus à droite du plateau est virtuellement reliée à la colonne la plus à gauche du plateau. Il en va de même pour les lignes.

2.2 les contraintes sur le placement des tuiles

Afin de pouvoir pouvoir placer une tuile, on dénombre bon nombre de contraintes. Parmi elles, on distingue :

— Avoir les faces adjacentes de même couleur. Ceci se traduit par une contrainte sur ce que peut ou ne peut contenir une nouvelle tuile arrivant sur le plateau

— Une nouvelle tuile doit se trouver sur un emplacement vide. Il a été choisi de marquer par le couple (0,0) les cases dépourvues de tuiles.

— Une nouvelle tuile doit être située à la frontière. Pour vérifier ceci, il suffit de s'assurer que la case visée est adjacente à une tuile déjà placée sur le plateau.

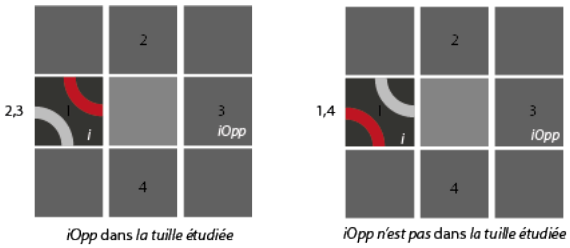


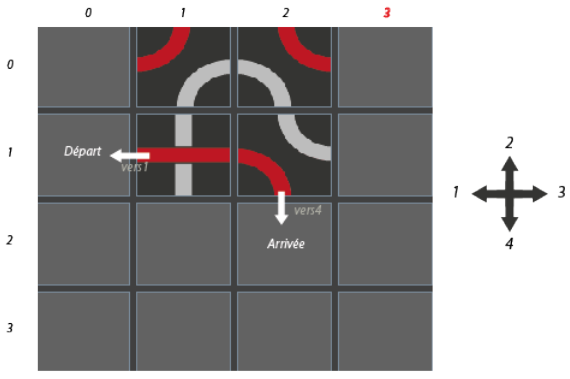
FIGURE 5 – contraintes imposées sur les couleurs adjacentes

— Il doit être impossible de placer autour d'une case vide trois tuiles présentant la même face. Pour vérifier cette dernière contrainte, il nous faut constamment s'assurer que la case visée a moins de deux faces voisines rouges et deux faces voisines blanches.

2.3 les chemins

Un chemin correspond ici à une succession de tuiles présentant des faces de même couleur contiguës les unes par rapport aux autres. Un chemin se définit donc par un point de départ, un point d'arrivée et un tracé. Lorsque le point d'arrivée rencontre le point de départ, on dit que notre chemin boucle. C'est ce dernier aspect qui nous intéresse ici puisqu'il marque la fin du jeu et la victoire de l'un des deux joueurs.

Les variables associées à un chemin sont donc :
— la position du début du chemin (de type int[]) ;
— la direction de ce chemins à l'entrée (de type int) ;
— la position de fin du chemin (de type int[]) ;
— a direction de ce chemins à la fin (de type int) ;
— un score qui correspond au nombre de tuiles traversées par le chemin (de type int).




```
Liste des chemins :
     (rouge): la boucle démare à 1,0 avec dir 1et finit à 2,2 et dir 4, score 2
     (rouge): la boucle démare à 0,0 avec dir 1et finit à 3,1 et dir 2, score 1
     (rouge): la boucle démare à 3,2 avec dir 2et finit à 0,3 et dir 3, score 1
     (blanc): la boucle démare à 1,3 avec dir 3et finit à 2,1 et dir 4, score 4
```

FIGURE 6 – Exemple de chemins

3 Implémentations Java du jeu

A présents que nous venons de voir comment il était possible de modéliser les différentes facettes du jeu, il est temps de passer à son implémentations Java. Après avoir étudié le diagramme de classe utilisé, je reviendrais un peu plus en détail sur l'aspect interface graphique du jeu.

3.1 Diagramme de classe

Pour l'implémentations de ce jeu, je suis partis sur une structure similaire à celle implémentée l'année dernière avec le jeu 2048 (développé dans le cadre d'études de Laboratoire de Génie Logiciel). Ainsi mon diagramme de classe comporte 4 packages :

— *game* : comporte l'ensemble des briques élémentaires du jeu ;
— *control* : permet de choisir parmi les différents mode de jeu et gère le déroulement d'un tour ;
— *hmi* : correspond à toute la partie interaction homme-machine ;
— *Trax* : comporte le Launcher du jeu.

Le diagramme d'état suivant réalise un état des lieux des différentes classes implémentées dans chaque packages. Pour des questions de lisibilité, il a été décidé de ne pas indiquer toutes les méthodes et attributs.

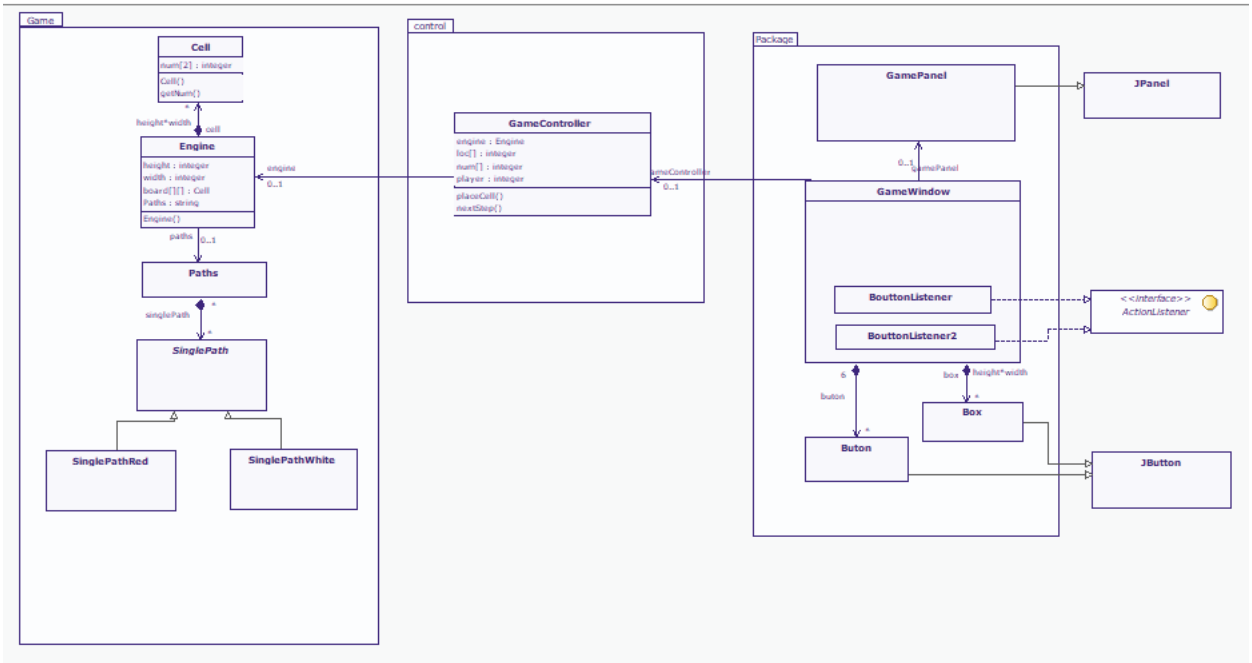


FIGURE 7 – Diagramme de Classe

Dans ce diagramme de classe, on remarque donc que toute la partie modélisation se trouve implémentée dans le package *game*. Le package *control* s'assure du bon déroulement de chaque tour. Ce package sera vu un peu plus en détail dans la partie traitant de l'intelligence artificielle. Quant au package hmi nous allons voir un peu plus en détail à quoi il correspond.

3.2 Interface homme-machine

L'implémentations de cette interface s'effectue via le package hmi du diagramme de classe.

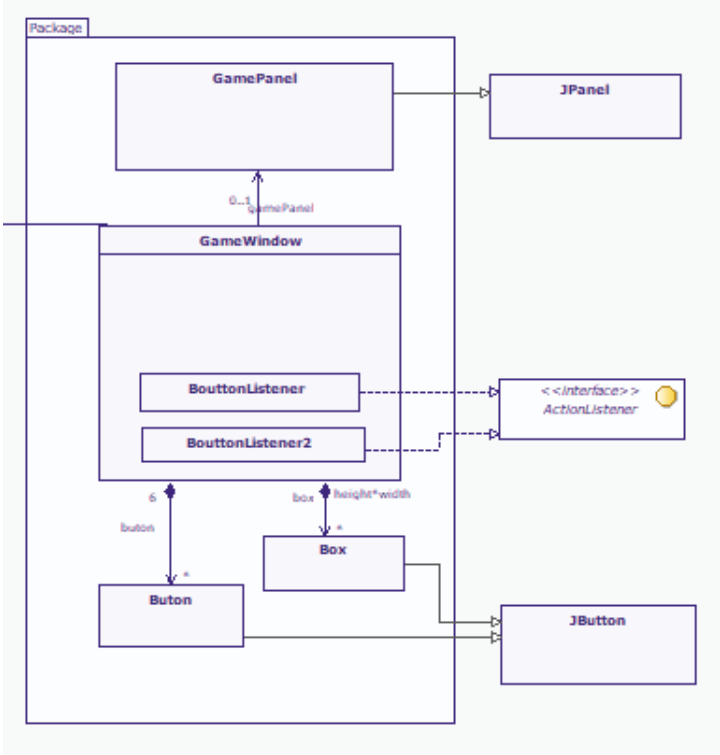


FIGURE 8 – Diagramme de Classe du package hmi

On peut distinguer dans ce package la partie affichage graphique, assurée par les classes héritant de JButton et JPanel, et la partie saisie de données utilisateurs, assurée par les classes BouttonListener et BouttonListener2. Le tout est coordonnée dans le constructeur de la classe GameWindow.

Concernant la partie interaction avec l'utilisateur, il a été choisit d'utiliser deux types de boutons. Le premier type bouton implémenté par la classe Box correspond aux différentes cases du jeu et permet d'obtenir la position souhaitée par le joueur. Le second type de bouton implémenté par la classe Button correspond aux différentes tuiles que le joueur peut placer sur le plateau. Appuyer sur un tel bouton permet de connaître le type de tuile que le joueur souhaite affecter à son prochain mouvement.

4 Implémentations du mode automatique

Le dernier aspect du projet développement logiciel fut la conception d'un mode de jeu automatique. Pour se faire, il a été nécessaire de passer par plusieurs étapes intermédiaires. Au final, le mode de jeu est choisit depuis le Launcher suivant la transcription suivante : 0=manuel, 1=auto, 2=random.

4.1 Mode player vs player

Dans ce mode de jeu , deux utilisateurs s'affrontent chacun leur tour jusqu'à ce que l'un d'eux remporte la partie. La Figure suivante fait état du déroulement d'une partie entre deux joueurs.

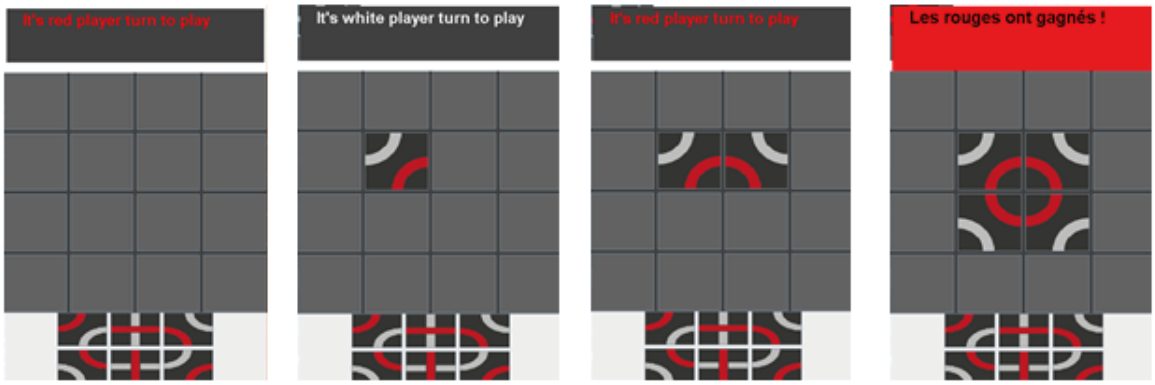


FIGURE 9 – Exemple de partie type

4.2 Mode player vs random

Dans ce mode de jeu, l'utilisateur affronte un joueur jouant aléatoirement à chaque tour. L'implémentations d'un tel mode de jeu a été rendue nécessaire pour vérifier que les méthodes d'explorations des cases disponibles étaient bien fonctionnelles.

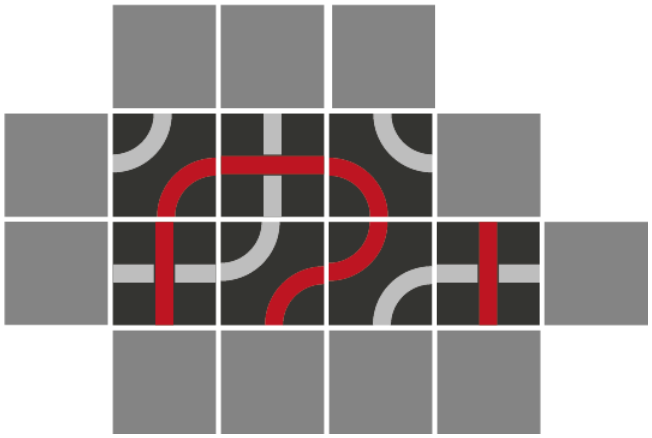


FIGURE 10 – Exemple de cases autorisées

4.3 Mode player vs auto

Dans ce mode de jeu, l'utilisateur affronte un joueur dont les choix sont déterminés selon un algorithme basé sur le principe de Monte Carlo. Le principe d'un tel algorithme est de jouer pour chaque combinaison (position possible, tuile possible) n parties jusqu'à la fin. A partir des résultats de ces n simulations, on détermine via un système de score la solution optimisant ses chances de victoires.La complexité apparemment élevée d'un tel algorithme va donc être un facteur limitant à son implémentations.

Dans l'implémentations de mon algorithme, la complexité des opérations étaient telle que je ne pouvais que difficilement dépasser n=50. Pour remédier à cela, j'ai également rajouté une dimension temporelle, en imposant comme règle que l'algorithme devait privilégier les solutions lui permettant une victoire rapide. Cela se traduit par des parties qui se terminent par des match nuls si personne n'arrive à gagner au bout de 20 tours de jeu. De plus le score d'une combinaison est fonction de deux paramètres :

- *s win* qui est incrémenté du nombre totale d'itérations (ici 20) moins le nombre d'itérations effectuées en cas de victoire
- *urg* qui correspond à l'urgence de la case. Il est incrémenté de 2*n si la partie se finit tout de suite après placement de la combinaison étudiée et de n si la partie se finit après un tour de jeu de l'adversaire.

Tout ceci permet l'implémentations d'un algorithme satisfaisant qui réussis à me battre quasi-systématiquement avec un n comprit entre 10 et 100 (20 étant un bonne valeur pour n).

5 Conclusion

Malgré un début difficile, je suis donc parvenu à répondre à la demande principale de ce projet de développement logiciel à savoir pouvoir s'essayer au jeu Trax. Bien sur ce jeu n'est pas parfait et présente encore bon nombre de défauts parmi lesquels des bug graphiques, un problème de complexité lors de l'exécution du mode automatique... Néanmoins il reste parfaitement fonctionnel en mode player vs player.

De nombreux axes d'améliorations sont à explorer. Par exemple, il serait intéressant d'améliorer le mode auto en basant l'algorithme sur une méthode d'apprentissage plus efficace. Par ailleurs, il peut être intéressant de permettre un choix de la taille du plateau et du mode de jeu directement depuis la fenêtre du jeu.