Université de Cergy-Pontoise

RAPPORT

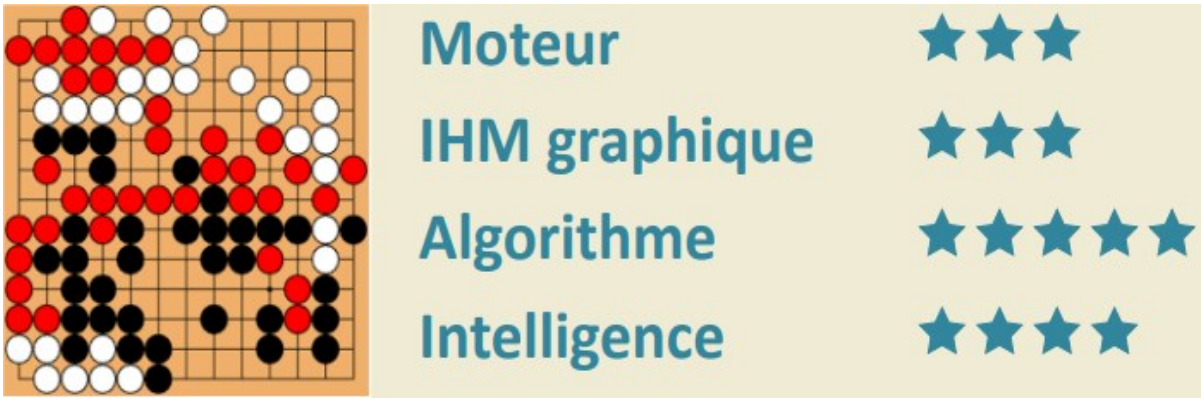
pour le projet Génie Logiciel
Licence d'Informatique deuxième année

sur le sujet

Jeu de Go

rédigé par

Afatchawo Junior - Chriqui Nathan - Da Cruz Mathis



Avril 2020

Table des matières

Table des figures

Liste des tableaux

Remerciements

Tout d'abord, nous tenons à remercier sincèrement M. Tianxiao Liu, qui, en tant que professeur de notre module de Génie Logiciel Projet, s'est toujours montré à l'écoute et disponible pour notre groupe tout au long de ce semestre. En effet grâce à sa dévotion en tant que superviseur, il nous a fournis beaucoup d'aide durant nos cours de TD, particulièrement utile en ce moment de confinement. Nous remercions également toute l'équipe des Professeurs de l'université de Cergy pour nous avoir suivie et soutenue jusqu'ici.

1 Introduction

Dans cette section, nous allons présenter brièvement l'objectif du projet

1.1 Contexte du projet

La motivation de ce projet est de faire connaitre le jeu de go à tous mais aussi de réinventer ce fameux jeu notamment en permettant une nouvelle approche à 3 joueurs.

Ce jeu était notre septième choix parce que l'idée général nous a bien plus : développer un jeu de plateau. De même l'aspect stratégique caractéristique au jeu de go nous a séduit.

1.2 Objectif du projet

Ce projet a pour objectif de réaliser un jeu de go en IHM graphique, l'utilisateur peut utiliser le logiciel pour apprendre et jouer à ce jeu de stratégie.

1.3 Organisation du rapport

2 Spécification du projet

Nous avons présenté l'objectif du projet dans la section 1. Dans cette section, nous présentons la spécification de notre logiciel réalisé. Ceci correspond principalement au document de spécification du projet (cahier des charges).

2.1 Notions de base et contraintes du projet

GO : Jeu de stratégie chinois (wéiqí) se pratiquant sur un goban (plateau quadrillé) de dimensions variables (9*9, 13*13, 19*19). Les joueurs disposent de pierres noires et blanches. Le jeu s'arrête une fois que les joueurs passent leur tour successivement.

Principe Général : Le but est de répartir le plateau entre les deux joueurs en dessinant des territoires, chaque intersection contenue dans un territoire valant un point. Noir commence en déposant une pierre de sa couleur sur une intersection du plateau. Puis, à tour de rôle, les joueurs posent une nouvelle pierre sur une intersection vide du goban. Le principe étant d'encercler les pierres adverses dans le but de les capturer.

Jeu de GO (logiciel) : Dans ce projet, on souhaite réaliser une application permettant de jouer au jeu de go aussi bien contre de « vrais » joueurs que contre une I.A. La particularité de notre jeu est que l'on fera des parties avec 3 joueurs.

Outils de développement :

1. Java
2. Eclipse
3. Latex

2.2 Fonctionnalités attendues du projet

L'utilisateur du logiciel doit pouvoir :

- Voir apparaitre le menu principal
- Décider de commencer une nouvelle partie ou quitter l'application
- Opter pour jouer face a des autres joueurs ou contre l'ordinateur
- Placer son pion sur le goban à l'aide la souris
- Visualiser le goban quadrillé ainsi que les hoshis
- Visualiser ses propres pierres à chaque nouveau coup (actualisation)
- Visualiser les pierres ennemies à chaque nouveau coup (actualisation)
- Visualiser la disparition de pierres capturées (actualisation)
- Visualiser le nombre de pierres adverses qu'il a capturé au cours de la partie
- Connaitre à tout moment à qui est le tour de jouer
- Nécessiter l'aide lui permettant de prendre connaissance du prochain coup qu'il pourrait faire.
- Utiliser le mégapion durant la partie
- Réinitialisé son dernier coup si jamais il se trompe

- Passer son tour : afin de manifester sa volonté de finir la partie
- Abandonner la partie : la partie s'arrête pour tout le monde et on retourne au menu principal.
- En fin de partie, visualiser s'il est gagnant ou non et son classement (score)
- Retourner au menu principal en fin de partie

Couleur	Pierres Capturées	Territoire	Total
Black	0	4	4
White	5	7	12
Red	8	3	11

TABLE 1 – Tableau des résultats

2.3 IHM

➢ Jeu

- Goban quadrillé (17*17) avec des hoshis (4) au centre de l'écran
- Pierres noires, blanches, rouges
- Méga pions : 1 seule utilisation par joueur par partie. Elimine les pierres ennemies voisines. De même couleur que les pierres de son joueur.
- Affichage de l'ordre des coups (exemple : « Au tour de rouge »)
- Affichage en temps réels des pierres capturées par chaque joueur (compteur pierres capturées)
- Boutons :
  - Resign : abandonner partie
  - Pass : passer son tour
  - Help : Affichage/Indication pédagogique où jouer le prochain coup
  - Confirm : confirmer choix du coup
  - Mégapion : Jouer le mégapion (utilisable seulement une fois => bouton grisé/inutilisable)

➢ Menu

- Boutons :
  - 3 adversaires réels
  - 2 adversaires réels 1 adversaire bot
  - Exit : Quitter

2.4 Spécificité

- Ordre de coups : Noir/Blanc/Rouge
- Actualisation du goban tour par tour.
- Comptage de points : Pour chaque joueur/couleur il y a deux compteurs. Un compteur comptant le nombre de pierres capturées et un autre comptant le nombre d'intersections occupées. Le score final du joueur est la somme de compteur pierre et compteur intersection. La couleur ayant le plus grand score final gagne la partie.
- Règles : Chinoises
- Le joueur sera interdit de jouer dans les cases présentant une situation dîtes de suicide ou de ko.

- Les pions morts seront comptés au profit du joueur « propriétaire » de la zone en question.
- La capture : Si une pierre ou chaine de pierre est encerclée par des pierres d'une ou plusieurs couleurs elles sont capturées. Les pierres capturées disparaissent de l'écran. Celle-ci seront comptabilisées au compteur de pierre du joueur ayant posé la pierre qui supprime la dernière liberté de cette chaine ou de cette pierre.
- Exceptions : Temps, Komi, Handicap ne sont pas prises en charge
- Robot ayant un seul niveau de difficulté : niveau 1. Il aura des tactiques de base mais sera peu agressif. Sa couleur rouge par défaut. (Plusieurs niveaux à développer si possible)
- Affichage pédagogique : Indique l'endroit où le joueur pourrait placer son pion au prochain tour.
- Méga pion : 1 seule utilisation par joueur par partie. Il élimine les pierres ennemies voisines. Attention cette action est valable uniquement lors du tour où la pierre est posée. Le mégapion disparait au tour suivant. Le nombre de pierres éliminées n'est pas pris en compte dans le compteur de pierres capturées (avantage purement stratégique). Il est utilisable partout.
- Fin de partie : Les 3 joueurs passent successivement.
- Affichage des points : Affichage du type (« noir gagnant ») suivis d'un tableau des scores.

3 Conception et réalisation du projet

3.1 Architecture globale du logiciel

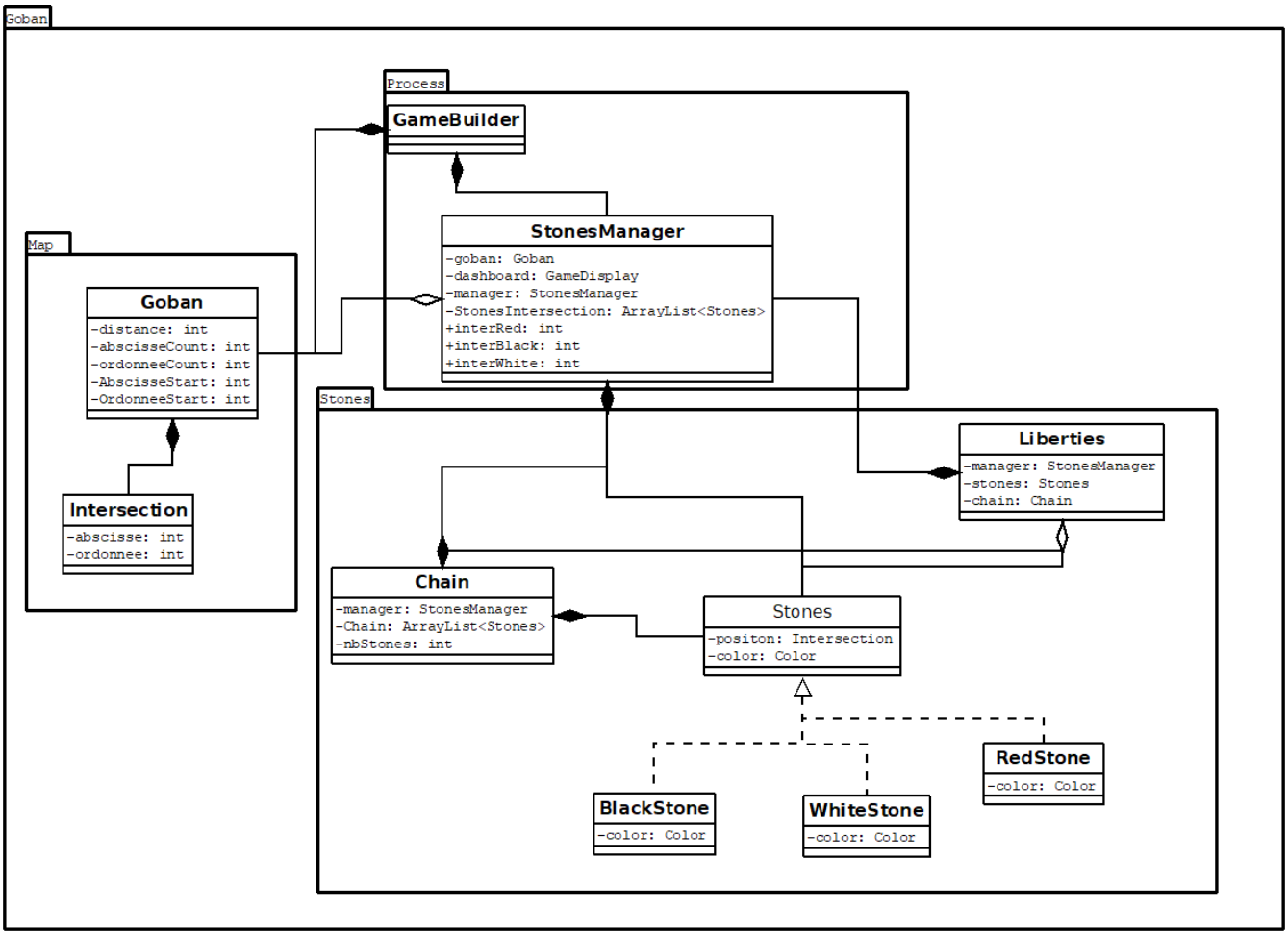


FIGURE 1 – UML simplifié du logiciel

3.2 Conception des classes de données

3.3 Conception des traitements (processus)

Cette section comporte les algorithmes principaux de notre projet qui ont permis l'implémentation des règles du jeu de go ainsi que de prendre en charge certaines exceptions. Ces algorithmes sont volontairement simplifiés afin de faciliter au mieux la compréhension.

Voici une liste de fonctions externes utilisées dans ces algorithmes :

- getIntersection() : renvoie l'intersection sur laquelle est placée la pierre
- ListStonesIntersection : liste des pierres présentes sur le goban
- getAbscise() et getOrdonnée : renvoie respectivement l'abscisse et l'ordonnée d'une intersection
- isExist() : vérifie si l'élément existe dans la liste en question
- isEnnemy(stones) : verifie si deux pierres sont ennemies
- isOnBorder(intersection) : verifie si l'intersection est sur un bord du goban

- isOnCorner(intersection) : verifie si l'intersection est dans un coin du goban
- countAllied(intersection) : renvoie le nombre d'alliés présent autour de ma pierre (sur les intersections voisines)

Voci une liste de variables récurentes utilisées dans ces algorithmes :

- find : intersection servant d'intermédiaire pour savoir si l'intersection initial existe
- supp : liste de pierres capturées
- chain : liste de pierres d'une même chaine de pierres
- chains : liste de toutes les chaines de pierres présentes sur le goban
- buttonMegapion : état du bouton Mégastone (1=pressé sinon 0)
- voisines : listes des pierres présentent sur les intersections voisines de mon intersection

Enfin les notations du type //INTERSECTION VOSINE DE GAUCHE indique que le phénomène se répète ensuite aux autres intersections (droite, haut, bas).

3.3.1 Fonction isOccupied(intersection)

Dans 1, on vérifie si une intersection donnée du goban est occupée ou non, cette fonction determinera si on pourra jouer notre prochain coup à l'intersection sélectionner avec la souris.

Algorithm 1 Vérifie si *intersection* appartient à *ListIntersection*

```
find ← null
for i : ListStonesIntersection do
  if i.getIntersection() == intersection then
    find ← i.getIntersection()
  end if
end for
return find == intersection
```

3.3.2 Fonction countLiberties(intersection)

Dans 2, on calcule le nombre de liberté d'une pierre. On part du postulat que notre pierre possède 4 libertés puis on décrémente d'une liberté a chaque fois qu'une liberté est "morte". A noter que dans cet algo, on considère qu'une intersection voisine occupée par une pierre alliée est une liberté

Algorithm 2 Retourne le nombre de *libertes* d'une pierre

```
liberties ← 4
//INTERSECTION VOSINE DE GAUCHE
if intersection.getAbscisse! = 0 then
  Intersectionleft
  if isExist(intersection) ET isOccupied(intersection) ET isEnnemy(intersection) then
    liberties ← liberties − 1
  end if
end if
//INTERSECTION VOSINE DE DROITE
//INTERSECTION VOSINE DU HAUT
//INTERSECTION VOSINE DU BAS
if isOnCorner(intersection) then
  liberties ← liberties − 2
end if
if isOnBorder(intersection) then
  liberties ← liberties − 1
end if
return liberties
```

3.3.3 Fonction isCaptured()

Dans 3, on parcours le goban et on capture les pierres possédants 0 liberté. A noter que cet Algo s'appplique uniquement aux pierres isolés (!= chaine).

Algorithm 3 Si $countLiberties(intersection) == 0$ alors pierre capturé

```
List < Stones > supp
for intersection : ListStonesIntersection do
  if countLiberties(intersection.getIntersection()) == 0 then
    supp.add(stones)
  end if
end for
for stones : supp do
  ListStonesIntersection.remove(stones)
end for
```

3.3.4 Fonction isDependent(stones)

Dans 4, on vérifie si une pierre est dépendante c-à-d si son nombre de libertés est égal à son nombre de pierres alliées voisines (ces dernières n'étant pas décomptabilisées du comptage des libertés) Cet Algo permet de savoir si une pierre d'une chaine est dépendante aux status des autres pierres de la chaine.

Algorithm 4 vérifie si une pierre est dépendante

```
return countLiberties(stones.getIntersection()) == countAllied(stones.getIntersection())
```

3.3.5 Fonction parcourChain(stones, chain)

Dans 5, permet de parcourir récursivement une chaine pierre par pierre et les ajouter une par une dans une liste de pierres représentant la chaine.

Algorithm 5 parcour une chaine

```
while !isExist(stones, chain) do
  chain.add(stones)
  //INTERSECTION VOSINE DE GAUCHE
  if intersection.getAbscisse! = 0 then
    Stonesleft
    if isExist(stones.getIntersection()) ET isOccupied(stones.getIntersection()) ET
    !isEnnemy(stones) then
      parcourChain(left, chain)
    end if
  end if
  //INTERSECTION VOSINE DE DROITE
  //INTERSECTION VOSINE DU HAUT
  //INTERSECTION VOSINE DU BAS
end while
```

3.3.6 Fonction isCapturedChain()

Dans 6, on parcours le goban, en regroupant chaque pierre dans une liste de pierres (chaine) et chaque chaine dans une liste de chaines puis on capture les chaines dont toutes les pierres sont dépendantes. Effectivement, une chaine dont l'intégralité des pierres sont dépendantes est une chaine capturée (puisqu'aucune des pierres la composant n'est voisine avec une intersection inoccupée => libertéChaine==0).

Algorithm 6 Pour toutes chaines, si *nbStones* == *dependentStones* alors chaine capturée

```
< List < Stones > supp
List < List < Stones >> chains
for stones : ListStonesIntersection do
  < List < Stones > chain
  if isDependent(stones) ET !isExist(stones, chain) then
    parcourChain(stones, chain)
  end if
  chains.add(chain)
end for
for chain : chains do
  count ← 0
  dependent ← 0
  for stones : chain do
    if isDependent(stones) then
      dependent + +
    end if
    count + +
  end for
  if count > 1 ET dependent == count then
    for stones : chain do
      supp.add(stones)
    end for
  end if
end for
for stones : supp do
  ListStonesIntersection.remove(stones)
end for
```

3.3.7 Fonction catchStones(stones)

Dans 7, on vérifie si une pierre voisine (=stones) possède uniquement 1 libertée et est une pierre ennemie (vérifie si elle est capturable ou non). Cet algo, s'avérera nécessaire dans le cas ou une intersection s'avérerait être une situation de suicide pour la pierre qu'on allait jouer mais qu'elle permet la capture d'une pierre ennemie (annule la situation de suicide).

Algorithm 7 vérifie si une pierre voisine est potentiellement capturable

```
return countLiberties(stones.getIntersection()) == 1 ET isEnnemy(stones)
```

3.3.8 Fonction catchChain(stones)

Dans 8, on vérifie si une chaine voisine (=stones) possède uniquement 1 liberté et est une chaine ennemie (vérifie si elle est capturable ou non). Cet algo, s'avérera nécessaire dans le cas où une intersection s'avérerait être une situation de suicide pour la pierre qu'on allait jouer mais qu'elle permet la capture d'une chaine ennemie (annule la situation de suicide).

Algorithm 8 vérifie si une chaine voisine est potentiellement capturable

return $countLiberties(stones.getIntersection()) == countAllied(stones.getIntersection()) + 1$ ET $isEnnemy(stones)$

3.3.9 Fonction suicideChain(stones)

Dans 9, on vérifie si une chaine voisine (=stones) possède uniquement 1 libertée et est une chaine alliée (vérifie si on s'apprête pas à suicider une chaine alliée existante en la privant de sa dernière liberté). Cet algo, s'avérera nécessaire dans le cas ou une intersection s'avérerait être une situation de suicide pour une de nos chaines.

Algorithm 9 vérifie si une chaine voisine est potentiellement "suicidable"

return $countLiberties(stones.getIntersection()) == countAllied(stones.getIntersection()) + 1$ ET $!isEnnemy(stones)$

3.3.10 Fonction megapion(stones, buttonMegapion)

Dans 10, on active la capacité spéciale du mégapion. En effet, si l'on à pressé le bouton mégastone alors quand on jouera notre pierre, toute pierre voisine et ennemie sera supprimé du goban.

Algorithm 10 Si $buttonMegapion == 1$ alors activation du mégapion

```
List < Stones > supp
if buttonMegapion == 1 then
  List < Stones > voisines
  for intersection : voisines do
    if isOccupied(stones.getIntersection) ET isEnnemy(stones) then
      supp.add(stones)
    end if
  end for
  for stones : supp do
    ListStonesIntersection.remove(stones)
  end for
end if
```

3.4 Conception de l'IHM graphique

- ➢ Menu
- ➢ Fenêtre Principale
- ➢ Fin du Jeu

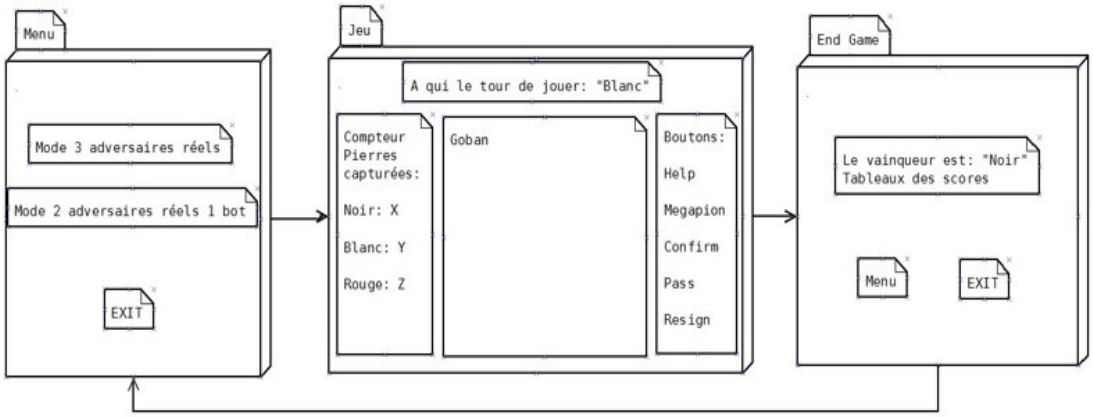


FIGURE 2 – Organisation des différentes fenêtres de l'IHM graphique

4 Manuel utilisateur

4.1 Menu

Dans la fenêtre d'accueil, les joueurs ont 3 options :

- Mode 3 adversaires réels : Lance le jeu et nous envoie vers la fenêtre principal. La partie se fera entre 3 joueurs réels (humains) tour par tour sur un même ordinateur (offline).
- Mode 2 adversaires réels, 1 bot : Lance le jeu et nous envoie vers la fenêtre principal. La partie se fera entre 2 joueurs réels (humains) et 1 robot (la machine) tour par tour sur un même ordinateur (offline). Attention, quand viendra le tour de rouge (robot), les joueurs humains seront priés de cliqués une fois sur le goban, pour que le robot joue et que la partie puisse se poursuivre. De même, ils appuieront sur le bouton "pass" à la place du robot (voir 4.2 Fenêtre principale).
- Exit : Quitte le logiciel.

FIGURE 3 – Menu

4.2 Fenêtre principale

Dans la fenêtre principale, les joueurs ont accès à 4 sections différentes :

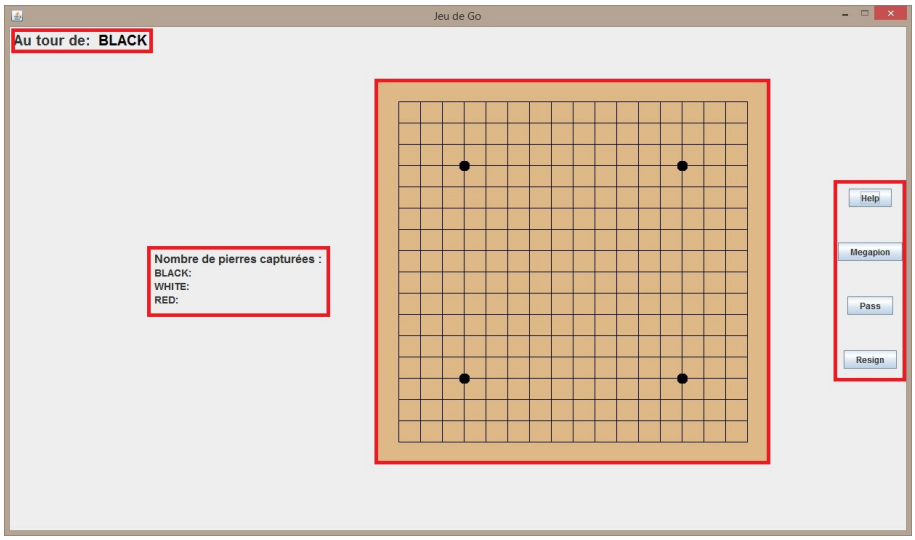


FIGURE 4 – Fenêtre principale

1. Affichage tour par tour : Mis à jour à chaque coup joué, il indique tout simplement la couleur de pierres (=joueur) qui doit être joué pour ce tour.

FIGURE 5 – Affichage dynamique tour par tour

2. Affichage du nombre de pierres capturées par chaque couleur : Lorsqu'une chaine de pierre ou bien une pierre est capturée, le compteur du captureur est incrémenté par le nombre de pierres capturées.

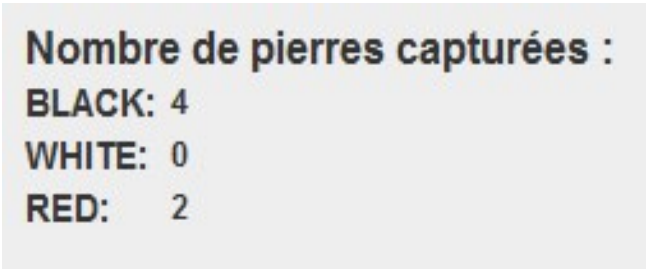


FIGURE 6 – Compteur de pierres capturées

3. Goban : Il s'agit du coeur de notre fenêtre, c'est la section qui va être le plus utilisée. Chacun leur tour, les joueurs vont cliquer sur l'intersection sur laquelle ils veulent jouer, et leur pierre s'affichera sur le goban si le coup respecte bien les règles.

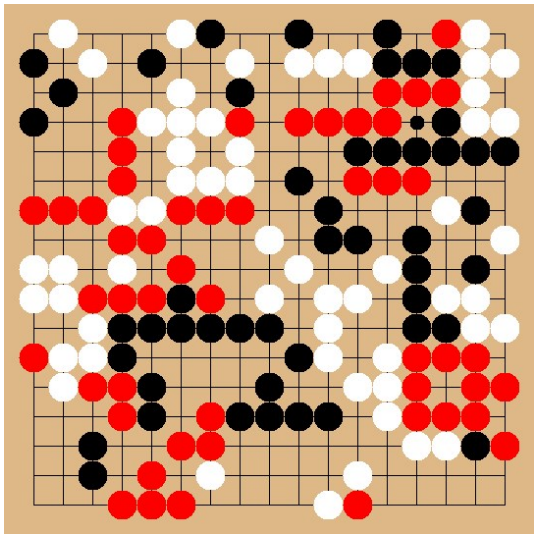


FIGURE 7 – Goban

4. Boutons :

   - Help : si un joueur ne sait pas où jouer il peut activer à tout moment ce bouton en cliquant une fois sur ce bouton puis en cliquant sur le goban. Alors l'IA jouera à sa place un coup en accord avec les règles du jeu de go.
   - Megapion : active le mode megapion. La pierre que l'on jouera sera alors un mégapion (rappel : avantage purement stratégique donc pas d'incrémentation dans le compteur des pierres).
   - Pass : si les 3 joueurs appuient successivement sur ce bouton, la partie est terminée (manière classique de mettre fin à la partie selon les règles du jeu de go =accord commun). Renvoie vers la fenêtre de fin de jeu.
   - Resign : Interruption imédiat de la partie. Renvoie vers la fenêtre de fin de jeu.

FIGURE 8 – Boutons

4.3 Fin de Jeu

FIGURE 9 – Fin du Jeu

A la fin de la partie, un tableau de score est affiché, il est constitué de 4 colonnes :

- Color : Les 3 couleurs représentants chaque joueur. Ainsi une ligne correspond à 1 joueur.
- CapturedStones : Nombre de pierres capturées par chaque joueur.
- Intersections : Nombre de pierres restantes pour chaque joueur.
- Total : Somme de CapturedStones et Intersections.

Color	Captured Stones	Intersections	Total
Black	33	21	54
White	61	36	97
Red	19	44	63

FIGURE 10 – Tableau des scores

Le joueur ayant le total le plus élevé remporte la partie !

Enfin dans la fenêtre de fin de jeu, les joueurs ont 2 options :

- Menu : Renvoie au Menu (fenêtre d'accueil).
- Exit : Quitte le logiciel.

5 Déroulement du projet

Dans cette section, nous décrivons comment le projet a été réalisé en équipe : la répartition des tâches, la synchronisation du travail en membres de l'équipe, etc.

5.1 Réalisation du projet par étapes

Etapes d'avancemements :

1. Spécifications du projet : Documentation, cahier des charges et répartition des tâches.
2. Conception des classes de données : UML des classes de données, premières classes.
3. Conception de l'IHM graphique : Construction du goban et affichage en plusieurs fenêtres.
4. Conception du noyau fonctionnel : Conceptions des classes et approfondissements...
5. Algorithmes principaux : Règles du jeu, exceptions, résolution des problèmes rencontrés.
6. Robot : Réalisation d'un robot respectant les règles du jeu de go.
7. Tests du logiciel : Réalisation des classes de tests et log.
8. Latex et documentation du projet : Rédaction du rapport et réalisation de la vidéo de démonstration.
9. Préparation à la soutenance finale : Préparation pour l'oral

5.2 Répartition des tâches entre membres de l'équipe

Les différents membres de l'équipes ont sû faire preuves de cohésion et d'entraide lors de la conception du projet. Cependant, nous nous sommes répartis les tâches afin de respecter au mieux le délai imposé :

- Mathis : Implémentaion des règles du jeu de go (Algorithmie), documentation latex...
- Junior : Implémentation du robot (IA), classes de tests, conceptions de classes...
- Nathan : Interface graphique (IHM), navigations entre fenêtres, affichages...

6 Conclusion et perspectives

Dans cette section, nous résumons la réalisation du projet et nous présentons également les extensions et améliorations possibles du projet.

6.1 Résumé du travail réalisé

Le cahier des charges est globalement respecté(point de vue moteur, IHM, algo et IA). En revanche nous ne sommes pas parvenus à dissocier parfaitement dans des classes différentes les différentes fonctions (algorithmes) implémentant les règles d'où la surcharge de la classe GameFrame.

6.2 Améliorations possibles du projet

Optimisation de la classe GameFrame et meilleurs répartitions des fonctions dans les classes, IA plus autonomes et plus intelligente, pris en charge de l'ensemble des exceptions, Interface graphique avec un meilleur design...

6.3 Derniers mots

Nous avons trouvé très intéressant de travailler sur la conception d'un jeu de plateau et de stratégie. Au quotidien nous jouons beaucoup à des jeux qui demandent une approche stratégique que ce soit des jeux plus contemporains comme League of Legends ou bien plus traditionnel comme les échecs. Donc ce projet nous a permis de savoir comment tout cela fonctionne vraiment, ceci est très intéressant dans un cadre professionnel comme personnel Et cela nous a également permis d'évoluer dans le monde de la programmation et d'appliquer les design pattern de développement d'un logiciel en Java. De plus la répartition des tâches personnelles et les séances de travail en commun nous ont montré l'importance du travail en équipes. Nous en sommes très fiers.

Références

[HLR02] L. M. Haas, E. T. Lin, and M. A. Roth. Data integration through database federation. *IBM Syst. J.*, 41(4) :578–596, 2002.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.