

# Projet semestre 1 Jeu de Go

BOËDA Gautier      LE ROUZIC Steven

ENSICAEN 2013-2014, 1A Info, Groupe TP 2

## Contents

<b>1</b>	<b>Environnement de développement</b>	<b>2</b>
<b>2</b>	<b>La libmatrice</b>	<b>2</b>
<b>3</b>	<b>Le jeu de Go</b>	<b>2</b>
3.1	Structures de données . . . . .	2
3.2	Algorithmes . . . . .	4
3.3	Intelligence Artificielle . . . . .	7
<b>4</b>	<b>Architecture du programme</b>	<b>8</b>
4.1	Les écrans et leurs états . . . . .	8
4.2	Les contextes . . . . .	9
4.3	Les interfaces . . . . .	9
4.4	Les événements . . . . .	9
<b>5</b>	<b>Autres éléments du programme</b>	<b>9</b>
5.1	Eléments d'interface graphique . . . . .	9
5.2	Les ramasse-miettes . . . . .	10

# 1 Environnement de développement

Le programme a été réalisé principalement sous Windows, avec l'IDE Eclipse et l'environnement MingW. Nous avons utilisé la librairie SDL 1.2 ainsi que ses add-ons `SDL_image` et `SDL_ttf`. La librairie matrice écrite en TP a été réécrite pour correspondre aux conventions établies pour ce projet. Nous avons utilisé Git comme système de versionnage afin de faciliter le travail depuis de multiples stations de travail, et GitHub comme dépôt central. (<http://github.com/stevenlr/ProjetGO>)

Chaque sous-projet était organisé comme ceci : un dossier `assets` contenant les images, les fontes et autres ressources, un dossier `lib` contenant les librairies nécessaires à la compilation, un dossier `include` contenant les fichiers d'en-tête des librairies et un dossier `src` contenant le code source et un dossier `include` contenant les en-têtes.

Des conventions d'écriture ont été adoptées : nous avons utilisé le CamelCase, le code était écrit le plus possible en français (à l'exception des mots `get` et `set`) et les prototypes de fonctions suivaient le style `Type_nomFonction` où `Type` est le type sur lequel opérait la fonction.

Les fichiers sources étant organisés en sous-dossiers, un fichier `Makefile` a été placé dans chaque dossier en appelant un autre commun à tous ceux-ci contenant les instructions de compilation pour chaque sous-dossier, ainsi que les autres `makefiles` des sous-dossiers. Tous les fichiers objets générés étaient alors placés dans un même dossier `obj` d'où ils étaient assemblés en l'exécutable final. **Les instructions de compilation sont spécifiées dans le fichier `README.md`.**

Ce rapport a été écrit en Markdown puis compilé en PDF grâce à l'utilitaire Pandoc.

## 2 La libmatrice

La librairie matrice écrite en TP a été entièrement réécrite afin de suivre les conventions établies, en particulier l'encapsulation des données. En interne, une représentation linéaire des données a été utilisée au lieu d'un tableau bidimensionnel.

## 3 Le jeu de Go

### 3.1 Structures de données

L'implémentation des types a été réalisée suivant les conventions du cours d'Algorithmique et, le cas échéant, selon les conventions données dans le document

Projet GO 2013.

Les types Liste, Pile, Position, Pion, Chaîne, Territoire, Plateau, Partie et Tutoriel sont, chacun, un pointeur sur la structure lui correspondant. Le type ElementListe est une structure. Le type Couleur est une énumération.

### **3.1.1 ElementListe**

Cette structure est définie dans Liste.c et correspond au type “cellule” vu en cours d’algorithmique. Ce type est très utilisé, on a donc créé un ramasse-miettes pour ce type.

### **3.1.2 Liste**

Ce type permet de définir les types Pile et Chaine utilisés par le Jeu de Go.

### **3.1.3 Pile**

Pile est un typedef de Liste. Ce type est utilisé pour les fonctions de détermination de Chaines et de Territoires.

### **3.1.4 Position**

Ce type est utilisé pour indiquer la position d’un pion, d’une case sur un plateau. Il est donc très utilisé. Ainsi, un ramasse-miettes a également été créé pour ce type.

### **3.1.5 Couleur**

C’est une énumération contenant les couleurs de cases/pions suivantes : VIDE, BLANC, NOIR, AURAVIDE, AURABLANC, AURANOIR. Les couleurs AURA nous permettent de mettre en évidence certains pions ou cases. L’utilisation de ces mises en évidence est faite dans le tutoriel.

### **3.1.6 Pion**

La structure Pion contient une couleur et une position. Son utilisation est limitée et on aurait pu en faire abstraction. En effet, son unique utilisation est faite dans la fonction Plateau\_capturerChaines.

### 3.1.7 Chaîne

La structure Chaîne contient une liste de positions et une couleur. Il correspond à une chaîne de pions d'une même couleur sur un plateau de jeu de GO. Le type Chaîne est utilisé ainsi par les fonctions Plateau\_determinerChaîne et Territoire\_determinerTerritoire.

### 3.1.8 Territoire

Territoire est un typedef de Chaîne. C'est une chaîne de cases vides uniquement.

### 3.1.9 Plateau

Ce type est un plateau de jeu de Go auquel on a ajouté des informations sur le nombre de pions de chaque couleur.

### 3.1.10 Partie

Ce type gère une partie de jeu de Go. Il stocke l'historique des plateaux, le komi, la taille du plateau, le handicap et le numéro du tour actuel. Il contient aussi les informations sur les joueurs, c'est-à-dire leur pseudonyme, et leur type (ordinateur ou humain).

### 3.1.11 Tutoriel

Ce type gère la partie tutoriel du jeu de Go. Il stocke une liste de plateaux-exemples ainsi qu'un commentaire pour chaque exemple.

## 3.2 Algorithmes

### 3.2.1 Déterminer les libertés

L'objectif est de déterminer les libertés de la chaîne passée en paramètre. Pour ceci, l'algorithme consiste à parcourir chaque pion de la chaîne. À chaque fois, on fait l'exécution suivante :

- On regarde s'il y a une case vide autour de ce pion (Au dessus, en dessous, à sa gauche ou à sa droite).
- Si une case vide est trouvée, on vérifie si cette dernière ne fait pas déjà partie de la liste de libertés en cours de détermination de cette chaîne.
- Si ce n'est pas le cas, on ajoute cette case à la liste de libertés.

Une fois le parcours terminé, on obtient ainsi la liste de libertés de la chaîne.

### 3.2.2 Déterminer territoire

L'objectif est de déterminer le territoire auquel la position passée en paramètre appartient. Cette position doit correspondre à une case vide du plateau. Pour ceci, l'algorithme consiste en l'utilisation d'une pile qui contient au départ la position passée en paramètre. À chaque fois on fait l'exécution suivante :

- On dépile une position de la pile et on vérifie s'il y a une case vide autour. (Au dessus, en dessous, à sa gauche ou à sa droite).
- Si une case vide est trouvée, on vérifie si cette dernière ne fait pas déjà partie du territoire en cours de détermination.
  - Si ce n'est pas le cas, on ajoute la position de cette case au territoire et on l'empile.
- Si une case non vide est trouvée, on vérifie que le territoire a une couleur d'un des deux joueurs.
  - Si ce n'est pas le cas, on lui donne la couleur du pion.
  - Si c'est le cas, et si la couleur n'est pas celle du territoire alors on déclare ce territoire neutre.
- On s'arrête une fois la pile vide.

Une fois le parcours terminé, on obtient ainsi un territoire auquel appartient la position passée en paramètre.

### 3.2.3 Déterminer séki

L'objectif est, à partir d'un territoire passé en paramètre, de savoir si ce dernier est un séki ou non. Ce territoire doit avoir la couleur VIDE et n'être constitué que de deux cases. Ce territoire étant de couleur VIDE, cela implique qu'il est au moins entourée par deux chaînes de couleur différentes. L'algorithme s'exécute donc ainsi :

- On détermine les chaînes qui entourent le territoire.
- Pour chaque chaîne, on vérifie que ses libertés appartiennent au territoire. Le cas échéant, on renvoie que ce n'est pas un séki.
- Si les libertés de chaque chaîne sont toutes comprises dans le territoire, alors ce dernier est un séki.

### 3.2.4 Déterminer les yeux d'une chaîne

L'objectif est, à partir d'une chaîne, de déterminer les libertés de la chaîne qui sont des yeux. Pour cela, l'algorithme consiste à parcourir les libertés de la chaîne et vérifier pour chacune si c'est un oeil. À chaque fois, on fait l'exécution suivante :

- On regarde si la liberté est entourée par un pion de la couleur de la chaîne ou un bord. (Au dessus, en dessous, à gauche et à droite)
- On regarde si la liberté a au plus un pion de la couleur opposée ou vide adjacent en diagonale. (Les coins donc du carré entourant la liberté).
- Si les deux conditions sont vérifiées, c'est un oeil. Sinon ce n'en est pas un.

On obtient donc la liste des yeux d'une chaîne.

### 3.2.5 Déterminer une chaîne

L'objectif est de déterminer une chaîne à partir d'une position appartenant à cette chaîne. Pour ceci, l'algorithme consiste à utiliser une pile qui contient au départ la position passée en paramètre. A chaque fois on fait l'exécution suivante : - On dépile une position de la pile et on vérifie s'il y a un pion de la couleur de la chaîne autour. (Au dessus, en dessous, à sa gauche ou à sa droite) - Si un tel pion est trouvé, on vérifie si ce dernier ne fait pas déjà partie de la chaîne en cours de détermination. - Si ce n'est pas le cas, on ajoute la position de ce pion à la chaîne et on l'empile. - On s'arrête une fois la pile vide.

Une fois le parcours terminé, on obtient ainsi une chaîne auquel appartient la position passée en paramètre.

### 3.2.6 Réaliser une capture

L'objectif est de vérifier si la pose d'un pion entraîne une capture ou davantage et la ou les captures réalisées. Pour ceci, l'algorithme consiste à vérifier si la pose du pion entraîne la capture d'une des chaînes situées au dessus, en dessous, à gauche ou à droite. Pour chacune des 4 positions autour du pion posé, on fait l'exécution suivante :

- On vérifie si la position appartient déjà à une des chaînes à capturer.
- Si ce n'est pas le cas, on détermine la chaîne correspondant à cette position ainsi que ses libertés.
- Si cette chaîne n'a pas de liberté, on l'ajoute à la liste des chaînes à capturer.
- Une fois ces 4 positions étudiées, et si la liste des chaînes à capturer n'est pas vide, on réalise la capture de ces chaînes.

### 3.2.7 Déterminer les chaînes entourant un territoire

L'objectif est de déterminer les chaînes entourant un territoire. Pour cela, l'algorithme va parcourir toutes les positions du territoire. À chaque fois, il fait l'exécution suivante :

- Il verifie si au dessus, en dessous, à gauche ou à droite il y a un pion.
- Pour chaque pion trouvé, il va vérifier si ce pion n'appartient pas déjà à une des chaînes autour du territoire en cours de détermination.
- Si c'est le cas, il va déterminer la chaîne à laquelle ce pion appartient et l'ajouter à la liste des chaînes autour du territoire.

Une fois le parcours terminé, on obtient ainsi les chaînes qui entourent ce territoire.

### 3.2.8 Calcul du score

L'objectif est de calculer le score final de la partie à partir d'un plateau. Pour ceci, l'algorithme ajoute d'abord le komi au bon joueur puis parcourt le tableau. Pour chaque position, il fait l'exécution suivante :

- On verifie si la position a déjà été traitée. Si c'est le cas, on passe à la suivante.
- On verifie si la position correspond à un pion. Si c'est le cas, on ajoute 1 point au score du joueur de la couleur du pion et on ajoute la position aux positions déjà traitées.
- Sinon, la case est vide, on détermine alors le territoire qui lui est associé.
- On verifie si le territoire appartient à un des deux joueurs. Si c'est le cas, on ajoute le nombre de cases du territoire au joueur de la couleur du territoire.
- On ajoute toutes les positions des cases aux cases déjà traitées.

Une fois le plateau parcouru, on obtient le score de chaque joueur.

## 3.3 Intelligence Artificielle

Une intelligence artificielle relativement simple (mais non naïve) a été implémentée. Son exécution se déroule en plusieurs phases : la phase de passage de tour, de défense, d'attaque et de jeu naïf.

### 3.3.1 Passage de tour

Si l'adversaire vient de passer son tour, alors l'ordinateur a une chance sur quatre de faire de même. Cela permet, lorsqu'un humain joue contre l'ordinateur, de terminer le jeu dans des temps raisonnables si le joueur décide qu'il ne peut plus rien faire qui pourrait l'avantager. En effet, cette IA ne réfléchira pas directement à la possibilité de passer son tour si elle n'a plus l'avantage nulle part sur le plateau.

Si l'ordinateur n'a pas décidé de passer, alors la phase de défense commence.

### 3.3.2 Phase de défense

L'ordinateur va commencer par vérifier s'il est en danger quelque part sur le goban : si une de ses chaînes ne possède plus qu'une liberté. Si c'est le cas, il va placer un pion sur cette liberté afin d'essayer de se libérer. Le système de placement de pion permet de s'assurer que si le coup joué résulte en une chaîne n'ayant plus de libertés pour l'ordinateur lui-même, alors le coup est annulé, car illégal.

Si l'ordinateur n'a pas joué défensivement, il va passer à l'offensive.

### 3.3.3 Phase d'attaque

L'ordinateur va alors chercher sur le goban toutes les chaînes ennemies et sélectionner celle possédant le moins de libertés. S'il en trouve une, il va placer un pion sur une de ses libertés.

Si l'ordinateur n'a pas pu jouer de coup offensif, alors il va passer à la phase naïve.

### 3.3.4 Phase naïve

Le dernier recours de l'ordinateur est alors de placer un pion au hasard sur les cases restantes du goban. S'il n'a pas réussi à en placer, alors il passe son tour.

## 4 Architecture du programme

Le programme a été structuré de façon à rendre l'implémentation des contextes console et graphique la plus séparée possible de la logique du jeu et à éliminer le maximum de redondances.

### 4.1 Les écrans et leurs états

Le programme fonctionne comme une machine à états, chaque état étant un écran. Les écrans sont : le menu, le tutoriel, les options de partie et le jeu. Chaque écran est donc une partie différente du programme qui se caractérise par une logique, des entrées et des sorties différentes. Ainsi chaque écran possède sa fonction logique (exemple `EcranJeu_main`) et des fonctions d'interface dont nous reparlerons plus tard. Chaque écran possède également ses états, c'est-à-dire des variables statiques à l'écran en cours qui définissent par exemple si on doit continuer à exécuter l'écran actuel, dans quelle partie on se trouve actuellement, etc. Les états sont initialisés à l'initialisation de l'écran. Lorsque le programme doit passer à un autre écran, l'écran courant initialise l'écran



suivant avec éventuellement les paramètres associés, puis s’auto-détruit. Le gestionnaire d’écrans est alors notifié du changement et, à moins que la fermeture de l’application soit demandée, va lancer la fonction `main` du nouvel écran.

## 4.2 Les contextes

Etant donné le besoin de pouvoir exécuter le programme à la fois en console et en mode graphique, il a fallu séparer au maximum ce qui était spécifique au contexte d’exécution de la logique du jeu, afin d’éviter toute redondance. Ainsi deux contextes sont définis : le contexte graphique et le contexte console. Au début du programme, le contexte d’exécution est choisi et initialisé. L’initialisation permet par exemple dans le cas du contexte graphique d’ouvrir la fenêtre, de charger les textures, de créer les boutons, etc. De même, à la fin du programme le contexte est détruit. Une structure similaire aux états pour les écrans contient les variables utiles à l’exécution du contexte dans la suite de l’application. Par exemple, le contexte graphique contient le pointeur `SDL_Surface` représentant l’écran.

## 4.3 Les interfaces

Pour pouvoir interagir en entrée et en sortie avec l’utilisateur, nous avons défini des interfaces. Chaque écran possède une interface de sortie et une d’entrée pour chacun des contextes. Ces fonctions ont connaissance des états mais ne peuvent pas les modifier. En effet, elles ne gèrent strictement que les entrées et sorties et ne doivent en aucun cas avoir conscience de la logique du jeu.

## 4.4 Les événements

Afin donc de pouvoir faire évoluer les états de jeu, les interfaces appellent des fonctions événements. Chaque écran possède ses propres fonctions d’événements. Par exemple pour l’écran de jeu : un événement pour la pose d’un pion, un pour l’enregistrement de la partie, un pour le retour à un tour précédent, etc. Ces fonctions sont alors le cœur de la logique du jeu.

# 5 Autres éléments du programme

## 5.1 Eléments d’interface graphique

Afin de faciliter les entrées utilisateurs, des structures ont été créées pour faciliter l’affichage et la gestion des éléments d’interface graphique. Chaque élément est instancié lors de la création du contexte graphique et stocké dans une variable globale afin de pouvoir être utilisé plus tard.

### 5.1.1 Les boutons

Le premier élément d'interface graphique est le bouton. Il lui est spécifié une taille, une position, un texte, une couleur de fond et une couleur de texte. Une fois créé il est possible lors d'un clique de souris de tester si celui-ci était sur le bouton afin d'engendrer d'autres événements.

### 5.1.2 Les boutons à choix multiple

Le second élément est un bouton à choix multiple. Il permet à l'utilisateur de choisir une option parmi une liste proposée. Il lui est spécifié une taille par choix, une position, des couleurs de fond pour l'état sélectionné et normal d'un choix et la couleur du texte. Les différents choix (au nombre maximal de 3) sont ensuite ajoutés. Une fois créé, on peut lors d'un clique de souris tester si celui-ci était sur le bouton à choix multiple et si c'est le cas, déterminer quelle option était choisie. Il est alors possible de récupérer le numéro de l'option choisie. Ces boutons sont utilisés lorsque l'utilisateur a un choix fini. Exemple : humain ou ordinateur, taille du plateau.

### 5.1.3 Les champs de texte

Enfin, le champ de texte permet à l'utilisateur d'entrer du texte au clavier après l'avoir sélectionné à la souris, comme avec une interface graphique classique. Une barre indique que le champ est prêt à être utilisé. La taille du texte contenu est limitée, il n'est possible d'effacer des caractères depuis la fin uniquement et certains caractères spéciaux ne sont pas acceptés. Ces champs sont utilisés pour entrer le nom des joueurs.

Deux autres utilitaires plus simples existent : un pour gérer les textures (chargées lors de la création du contexte graphique) et un pour écrire du texte à l'écran, permettant de gérer le style et l'alignement du texte.

## 5.2 Les ramasse-miettes

Afin de limiter le nombre d'allocations et de désallocations par unité de temps pour les structures qui en demandent beaucoup telles que `ElementListe` et `Position`, nous avons implémenté un ramasse-miettes très simple nous permettant de recycler les précédentes instances de ces types dont voici le fonctionnement. Chaque type à recycler possède son ramasse-miettes.

Un ramasse-miettes est configuré avec le nombre maximal d'instances recyclées stockées et la taille d'une structure à allouer. Au début, aucun pointeur n'est recyclé. Lors de l'allocation d'un élément, le ramasse-miettes regarde si on a des pointeurs disponibles à recycler. Si c'est le cas, il retourne celui au dessus de

la pile des pointeurs recyclés, dans le cas contraire il alloue un élément de la taille spécifiée à la création du ramasse-miettes. Lors de la désallocation, si la pile d'éléments recyclés n'est pas pleine, il place le pointeur en haut de la pile et dans le cas contraire désalloue l'élément. À la fin de l'exécution du programme, tous les pointeurs recyclés restants dans la pile sont désalloués.