Projet TER de deuxième année de Licence

Résolveur de problèmes de vie ou de mort dans le Jeu de Go

Aliou Barry Mamadou
Carrio Florian
Huesca Victor
Rodriguez Julien
Soussi Wissem

Professeur Encadrant : Mr.Pompidor Pierre

Faculté des Sciences de Montpellier
Département Informatique Second Semestre de l'année Universitaire
2016-2017

Table des matières

Table des figures

Liste des Algorithmes

1 Introduction

Notre projet a pour but d'élaborer un algorithme capable de trouver, dans plusieurs situations d'une partie de Go si un groupe peut survire ou non. Pour cela nous avons besoin de tout construire depuis zéro, construire la structure de base du jeu, y implémenter les règles pour finir par créer l'algorithme nécessaire à la résolution d'un problème de vie ou de mort.

1.1 Présentation du Jeu de Go

1.1.1 Histoire du Go.

Le Jeu de Go est un jeu de plateau à 2 joueurs né en Chine il y a plusieurs milliers d'années, c'est le plus ancien jeu de stratégie combinatoire abstrait connu actuellement, on raconte qu'il fut créé pendant la période chinoise des Printemps et Automnes.

Malgré le fait qu'il soit très ancien, le Jeu de Go continue de profiter d'une grande notoriété dans les pays Asiatiques, notamment en Chine, en Corée et au Japon où il a vu naître sa forme actuelle au XVe Siècle pour beaucoup plus récemment s'exporter en Occident où nous l'avons découvert.

Il a subi ces dernières années une réelle attention de la part des grands de l'informatique notamment Facebook et Google qui se sont battus pendant un longue période pour confectionner une intelligence artificielle capable de battre n'importe quel humain au Go, c'est là que Google DeepMind a réussi avec Alpha Go en 2015 et a ainsi battu le meilleur joueur de Go dans les années 2000 Lee Sedol, sauf à la quatrième parti où celui-ci à vaincu la bête.

C'est à l'heure actuelle le plus grand rempart jamais franchi dans le domaine du DeepLearning. Cela explique l'engouement actuel autour de l'intelligence artificielle de beaucoup d'entreprises. Réussir ce challenge est alors un tournant symbolique, celui où les ordinateurs surpassent les humains dans une activité que l'on pensait propre à l'humain.

1.1.2 Règles du Jeu

Le Jeu de Go se joue à 2 joueurs, chacun d'eux pose à leur tour une pierre d'une couleur distincte sur un plateau quadrillé appelé Goban. Le but est de contrôler le plus vaste territoire, chacun se bat alors avec ses pierres pour envahir le territoire de l'autre tout en

protégeant les leur ! Il est aussi possible de capturer les groupes ennemis, nous faisant alors gagner des points que nous comptabilisons en plus des points rapportés par les territoires que nous possédons. Le gagnant est finalement celui qui obtient le plus de points.

1.2 Méthodologie de travail et environnement technique

1.2.1 Méthodologie de gestion des versions de codes : GIT

Ayant un module nous dispensant des cours sur cette plate-forme très pratique qu'est GIT, nous avons trouver ça intéressant d'utiliser celle-ci, et pour sûr nous ne l'avons pas regretter ! Chacun peut avancer en équipe et se passer le "témoin" à la manière d'une course de relais sauf que tout le monde peut avancer en même temps sur des points différents ! Malheureusement il y a aussi de légers désagréments, notamment lorsque tout le groupe ne code pas sur la même plate-forme. Cela engendre parfois des problèmes de compatibilité mais au final cela permet aussi une plus grande flexibilité sur le rendu final.

De plus, lorsque un membre fait des mauvaises modifications sans que nous nous en rendions compte, et que un peu plus tard tandis que nous buttions sur une erreur, nous réalisons alors que l'erreur venait de là... Heureusement dans sa globalité GIT fut un fabuleux outil que nous utiliserons sans doute pour nos futurs projets. Ainsi que la diversité des système d'exploitation permet aussi d'avoir une portabilité du programme et des débugages différents. L'intégralité de notre projet, ses avancées, etc sont disponibles via GIT à l'adresse suivante : www.github.com/Victor333Huesca/Jeu_de_Go

1.2.2 Choix des technologies

Le C++

Le choix s'est trouvé évident, cela faisait bientôt 2 ans que nous étudions ce langage, plus pour d'autres, d'autant plus que ce soit pour la programmation objet et l'optimisation mémoire celui-ci reste l'un des meilleurs pour exceller sur ces deux plans.

SFML

La SFML est une bibliothèque originellement destinée au langage C++ mais portée également vers d'autres langages divers et variés. Celle-ci permet -entre autre- un affichage graphique, des outils système, réseau et même audio.
Notre choix s'est porté sur celle-ci car elle est extrêmement bien documentée (en français en plus) et elle présente un bon rapport entre simplicité d'utilisation et possibilités apportées.

1.2.3 Diagramme de tâche prévisionnel Gantt

Afin de ne pas s'égarer dans le cœur du projet, nous nous sommes réunis pour planifier les étapes clef du projet. Nous nous somme servis de la méthode de Gantt qui est assez

intuitive.

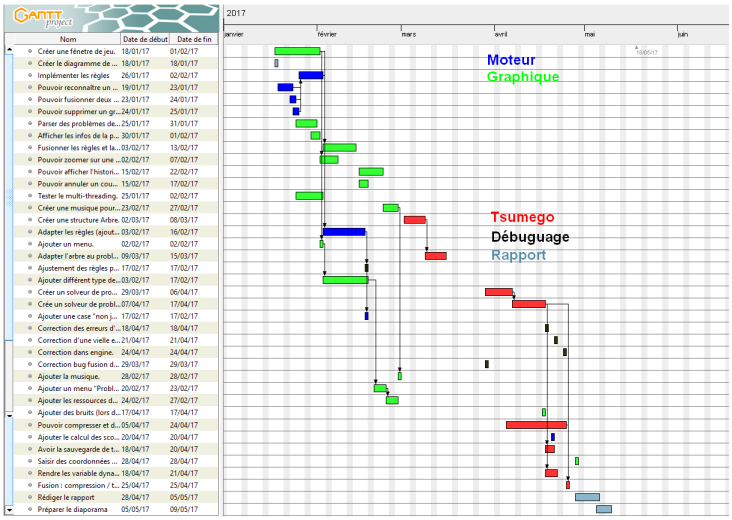

Figure 1: Gantt Prévisionnel

Évidemment dans un projet certaines parties prennent du retard et -plus rarement- de l'avance. Nous avons tout au long du projet tenu à jour notre planning pour le modifier en conséquence de notre avancement effectif.

Diagramme final

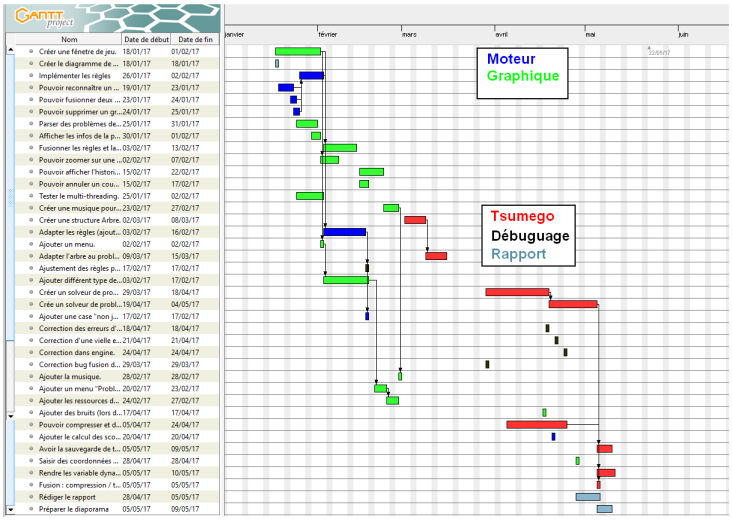

Figure 2: Gantt Final

2 Structures de données et algorithmes de base

Ici nous allons définir les règles du jeu à travers des algorithmes qui permettront d'analyser les situations probables du jeu afin de pouvoir vérifier la légalité des coups joués, ce qui nous rendra apte ensuite à créer l'IA du jeu.

2.1 Structure de données

2.1.1 Goban

Pour l'implantation du Goban, nous avons utilisé le paradigme objet du langage C++. Avec la notion de classe qu'introduit ce dernier, on a pu définir la classe Goban pour manipuler aisément ses données par le biais des fonctions membres.
Le Goban se présente comme un plateau de taille 19x19, composé d'état avec ses différentes valeurs.

Accès aux états

Le goban est en alors un tableau d'états, dont la manipulation à l'aide des accesseurs permettent de modifié le Goban et simulé le Jeu.

2.1.2 États

L'état est une représentation d'une pierre dans le plan, avec ses coordonnées et une valeur qui peut être vide, noir, blanc, KO_Noir, KO_Blanc, ou simplement Non jouable (NJ).

2.1.3 Territoires d'un Goban : Les groupes

Un Groupe est une liste d'état voisin de même valeur se trouvant dans le Goban.

2.2 Algorithmes

2.2.1 Algorithme de Recherche

La recherche se fait dans le Goban, en deux temps. D'abord on recherche les groupes de couleurs noirs ainsi que les groupes de couleurs blanches. Ensuite on lance la fusion sur chacun des groupes. La recherche de Groupe consiste à parcourir tout le Goban en largeur

et vérifier si l'état courant appartient à un groupe déjà défini. Le cas échéant on rajoute la pierre à ce groupe et on passe au suivant, autrement on créer un nouveau groupe avec cette pierre.

```
Données : val : VAL , Array : Etat[]
Résultats : Définir les groupes de la couleur val dans Array
Variables : x, i, j : entier, groupe : Groupe
x ← 0
Si (val = BLANC) Alors
    val ← groupsWhite
Sinon
    val ← groupsBlack
Fin Si
Pour i de 0 à TGOBAN*TGOBAN faire
    Si (ValArray[i] = val) Alors
        j ← 0
        Tant que (j ≤ tailleGroups) faire
            Si (! (Array[i] ∈ groupe[j]))Alors
                Si (groupe[j] doitContenir Array[i]) Alors
                    groupe[j] ← ajout(Array[i])
                Fin Si
            Sinon
                j ← tailleGroups+1
            Fin Si
            j ← j + 1
        Fait
        Si (j = tailleGroups) Alors
            groupe[i] ← ajout(Array[i])
        Fin Si
    Fin Si
Fin Pour
```

Algorithme 2: Algorithme de recherche groupes

2.2.2 Algorithme de fusion

Pour un ensemble de groupe donnée, cet algorithme vérifie si une pierre d'un groupe est voisin d'une autre puis rajoute le deuxième groupe au premier et supprime le deuxième sinon il passe au suivant

```
Données : group : Groupe
Résultat : Fusionner les groupes de 'group' qui sont voisins
Variables : i, j : entier
Début :
Si (tailleGroup > 0) Alors
    Pour i de 0 à tailleGroup - 1 faire
        Pour i de i+1 à tailleGroup faire
            Si (groupe[i] estVoisin groupe[j]) Alors
                groupe[i] fusion groupe[j]
                supprimer(groupe[j])
            Fin Si
        Fin Pour
    Fin Pour
Fin Si
```

Algorithme 3: algorithme de fusion de Groupes

```
Données : goban : Goban
Résultat : definir les groupes du goban
Début :

rechercheGroups(NOIR, goban.array)
rechercheGroups(BLANC, goban.array)
fusionGroupes(groupesBlack)
fusionGroupes(groupesWhite)
```

Algorithme 4: Algorithme de recherche et de fusion de groupes

2.2.3 Algorithme de définition des libertés d'un groupe

Afin de calculer la liberté d'un groupe il faut d'abord calculer les libertés d'une pierre. L'algorithme de liberté s'applique de la façon suivante :

- Pour chaque pierre du groupe faire :
  - Vérifier si la pierre se trouve au centre ou dans le bord du goban.
  - Ajouter les états voisins de la pierre dans un tableau.
- Pour la liberté d'un groupe il suffit de voir s'il y a au moins un état dans le tableau des voisins qui est vide, le cas échéant renvoyer vraie sinon faux.

Dans l'algorithme il y a les fonctions suivantes :

— **taille(T)** : Soit un tableau d'éléments T la fonction renvoie la taille de T.

— **ajoute(e,t)** : Soit un élément e et un tableau t la fonction ajoute l'élément e en queue du t.

Données : gob : **Goban**, groupe : **Groupe** ;
Résultat : renvoie un tableau des libertés du groupe
Variables : l : **entier**,
Début :
Pour chaque pierre p de groupe **faire**
  Pour chaque voisin v de p **faire**
    Si (v = VIDE) **Alors**
      ajoute(v,l)
    Fin Si
  Fin Pour
Fin Pour
renvoyer l

Algorithme 5: Libertés d'un groupe

2.2.4 Algorithme de captures des groupes

A chaque pierre posée il faut vérifier si un ou plusieurs groupes adverses sont tués ou pas et mettre à jour le goban sur lequel se déroule le jeu.
L'algorithme d'élimination est lancé sur les groupes de l'adversaire par rapport à la dernière pierre posée. Pour chaque groupe du tableau des groupes de l'adversaire il faut vérifier différents éléments :

— Vérifier que le groupe ait des libertés.

— Si le groupe n'a pas de libertés alors il faudra l'éliminer du goban mais l'éliminer aussi du tableau des groupes.

```
Données : goban : Goban, groupes : Groupe[] ;
Résultat : Élimine les groupes sans libertés
Variables : Libertés : Etat[], j : entier, estLibre : booléan ;
Début :
Pour chaque groupe g de groupes faire
    estLlibre←0
    j←0
    libertés←libertés(g)
    Tant que (estLibre = faux et j < taille(libertés)) faire
        Si (g[j] = VIDE ou g[j] = KO ou g[j] = NJ) Alors
            estLibre←vraie
            i←i + 1
        Fin Si
    Fait
    Si (estLibre = faux) Alors
        eliminerDuGoban(g,goban)
        eliminerDuGroupes(g,Groupes)
    Fin Si
Fin Pour
```

Algorithme 6: Élimination des groupes

2.2.5 Algorithme d'élimination des groupes et intégration du KO

A chaque pierre posée il faut vérifier si un ou plusieurs groupes adversaires sont tués puis mettre à jour le goban sur lequel se déroule le jeu.
L'algorithme d'élimination est lancé sur les groupes de l'adversaire par rapport à la dernière pierre posée. Pour chaque groupe du tableau des groupes de l'adversaire il faudra vérifier différents critères :

— Vérifier que le groupe ait des libertés.
— Si le groupe n'a pas de libertés alors l'éliminer du goban mais l'éliminer aussi du tableau des groupes.

```
Données : goban : Goban, groupes : Groupe[] ;
Résultat : Élimine les groupes sans libertés
Variables : Libertés : Etat[], j : entier, estLibre : booléan ;
Début :
Pour chaque groupe g de groupes faire
    estLlibre←0
    j←0
    libertés←libertés(g)
    Tant que (estLibre = faux et j < taille(libertés)) faire
        Si (g[j] = VIDE ou g[j] = KO ou g[j] = NJ) Alors
            estLibre←vraie
            i←i + 1
        Fin Si
    Fait
    Si (estLibre = faux) Alors
        eliminerDuGoban(g,goban)
        eliminerDuGroupes(g,Groupes)
    Fin Si
Fin Pour
```

Algorithme 7: Élimination des groupes

Le KO est un cas particulier qui cause un blocage de la partie il est alors interdit par les règles du jeu. Un exemple de KO se trouve dans les figures 1 et 2 , on voit ici que les joueurs peuvent continuer à jouer à l'infini en mangeant les mêmes deux pierres.
Dans la structure de données créer un Etat peut avoir comme valeur **KOBLANC** qui désigne le KO blanc ou **KONOIR** qui désigne le KO noir.
Un KO retourne vide après un seul coup et une pierre ne peut pas se poser sur un KO

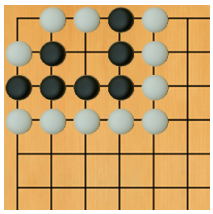

Figure 3: KO avant

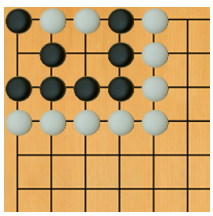

Figure 4: KO après

de la même couleur sauf si il mange un groupe qui est composé par plus d'une pierre.
Le KO devient, plutôt qu'un algorithme, une intégration de l'algorithme de l'élimination des groupes.

2.3 Algorithme de définition des gobans fils d'un noeud

Pour chaque noeud de l'arbre on calcule tout les coups possibles du joueur et pour chaque coup on créé un goban. On aura de cette manière les gobans fils qui serons créé successivement à la base de ces premiers.
Le nombre de gobans créés sera intuitivement le nombre d'états vides dans le goban de référence. Dans l'algorithme il y a les fonctions suivantes :

- **jouer(goban,pierre)** : contrôle si par rapport aux règles du jeu on peut poser la pierre dans le goban.
  Si le coup est possible alors l'appliquer et renvoyer vraie, sinon faux.
- **ajoute(e,t)** : Ajoute l'élément e en queue du tableau t.
- **rechercheGroupes(e,t)** : Ajoute l'élément e en queue du tableau t.
- **ajoute(e,t)** : Ajoute l'élément e en queue du tableau t.

2.3.1 Algorithme du suicide

Le suicide est un coup avec lequel un joueur peut tuer un ou plusieurs groupe de sa propriété. Ce type de coup est illégal dans le jeu de go, il faudra donc que le programme détecte si un coup du joueur est un suicide ou pas. Pour cela on a créé l'algorithme suivant qui suit les étape suivantes :

- Vérifier si l'endroit où on va poser la pierre a des libertés afin de déterminer si c'est un suicide ou pas.
- Exécuter le coup sur un deuxième goban virtuel et lancer sur ce dernier la recherche de nouveaux groupes et l'élimination des groupes de l'adversaire relatif à la couleur de la pierre.
- Vérifier si le coup a tué un groupe adverse, indiquant alors que c'est pas un suicide.
- Enfin lancer sur le goban virtuel l'élimination des groupes de la même couleur que la pierre, puis vérifier si il tue un de ses groupes, prouvant alors le suicide.

Dans cet algorithme il y a 4 fonctions appelées qui auront les objectifs suivants :

1. **GroupesNoir(Goban)** : Soit une instance de la structure Goban passée en paramètre la fonction renvoie alors ses groupes noirs.
2. **GroupesBlanc(Goban)** : Soit une instance de la structure Goban passée en paramètre la fonction renvoie alors ses groupes blancs.

3. **PosePierreDansGoban(Pierre,Goban)** : Soit une pierre et un goban la fonction pose la pierre sur le goban aux coordonnées stockées dans la variable pierre en vérifiant que l'endroit dans le goban soit vide.
4. **EliminerGroupes(pierre, goban)** :Soit une pierre et un goban la fonction élimine les groupes sans libertés dans goban avec la même couleur que la pierre.
5. **EliminerGroupesAdversaire(pierre, goban)** :Soit une pierre et un goban la fonction élimine alors les groupes sans libertés dans goban avec la couleur opposée à la pierre initiale.

Il est important de préciser que la fonction de calcule des libertés d'une pierre sera nécessaire aussi.

Données : Gob : **Goban**, Pierre : **Etat** ;
Résultat : Renvoie vrai si Pierre cause un suicide dans le Gob, renvoie faux sinon.
Variables : GroupesAVérifier : **Groupe[]**, GroupesAdversaire : **Groupe[]**, GroupesAdversaireAprès : **Groupe[]** ;
Début :
Si (taille(libertés(Pierre)) > 0) **Alors**
  renvoyer faux
Fin Si
goban2 ← goban
Si (Couleur(Pierre) = BLANC) **Alors**
  GroupesAdversaire ← groupesNoir(goban2)
Sinon
  GroupesAdversaire ← groupesBlanc(goban2)
Fin Si
JouerPierreDansGoban(Pierre, goban2)
RechercheGroupes(goban2)
EliminerGroupesAdversaire(Pierre, goban)
Si (Couleur(Pierre) = BLANC) **Alors**
  GroupesAdversaireAprès ← groupesNoir(goban2)
  GroupesAVérifier ← groupesBlanc(goban2)
Sinon
  GroupesAdversaireAprès ← groupesBlanc(goban2)
  GroupesAVérifier ← groupesNoir(goban2)
Fin Si
Si (taille(GroupesAdversaireAprès) < taille(GroupesAdversaire)) **Alors**
  renvoyer faux
Sinon
  EliminerGroupes(Pierre, goban)
  Si ((Pierre = BLANC et taille(GroupesAVérifier) ≠ $taille(groupesBlanc(goban2))$)**Alors**
    renvoyer vraie
  Fin Si
  Si ((Pierre = NOIR et taille(GroupesAVérifier) ≠ $taille(groupesNoir(goban2))$)**Alors**
    renvoyer vraie
  Fin Si
Fin Si
renvoyer faux

Algorithme 8: Est-Suicide ?

3 Interface graphique

3.1 Le jeu

3.1.1 La fenêtre de Jeu

C'est un élément déterminant car il est ce que voit l'utilisateur est c'est par celui-ci est uniquement celui-ci que le programme communique avec ce dernier. Bien que central cet élément **ne gère pas les choses lui-même**, en effet il se contente de **communiquer les événements qu'il reçoit** à ses composantes que sont le *Plateau (le Goban)* et *la zone d'infos.*

La fenêtre de jeu fait donc principalement office **d'interface** et sa fonction se résume à savoir qui des *infos* ou du *plateau* est apte à *traiter l'information* reçue. C'est donc un des éléments situés au plus **haut niveau** avec les différents menus, cette catégorie d'objet est celle ayant en charge la **boucle principale** et s'affichant directement à l'écran.

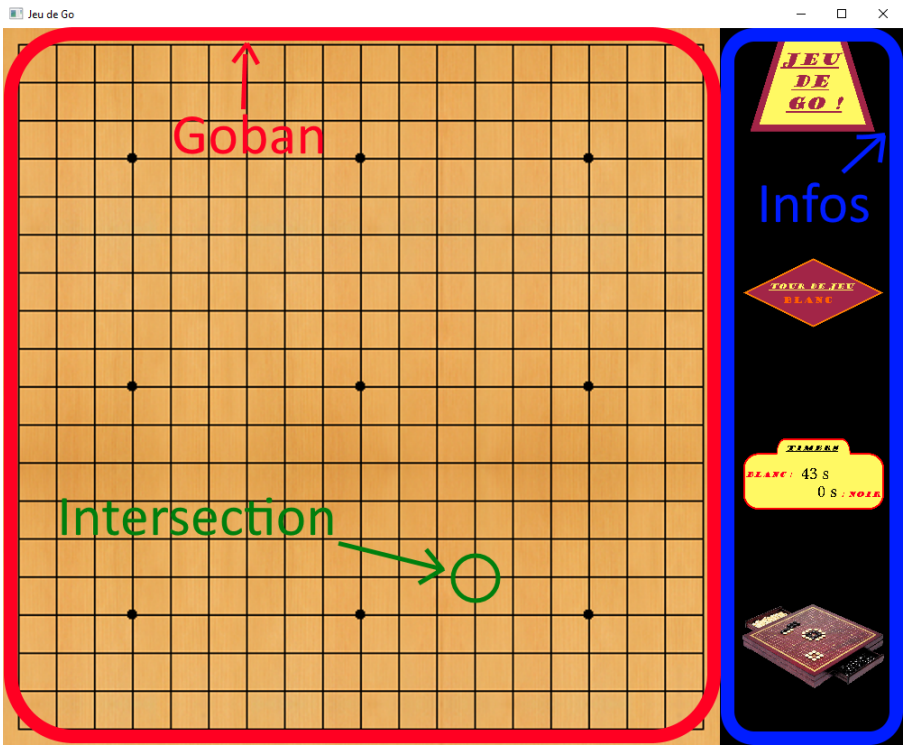

Figure 5: Fenêtre de Jeu

3.1.2 Une partie de Go

Le goban est composé tout d'abord d'un **visuel** -son apparence- qui peut éventuellement changer selon les goûts de l'utilisateur. On peut aussi souhaiter se concentrer simplement sur une partie du plateau et pour cela il convient d'autoriser un **zoom** afin de correspondre au mieux à la zone pertinente aux yeux de l'utilisateur.

De plus le plateau est l'élément qui **interface** avec le *'moteur'* du jeu, c'est à dire les différentes *règles du jeu*. Pour cela il convient de lui permettre de faire la jonction correctement entre le visuel -qui est destiné à l'utilisateur- et ce moteur. Évidemment ceci fonctionne dans les deux sens et **les actions utilisateur** (clavier, souris, ...) doivent être interceptées par cet élément pour **être transmises** aux *règles* qui lui dicteront en retour l'action à entreprendre (par exemple pour un feed-back).

Bien qu'il fasse office de passerelle avec le moteur, il convient de ne pas résumer le goban simplement à cette fonction munie d'un visuel. En effet le goban représente avant tout **la zone de jeu** et celle-ci est composé d'un élément capital : *les intersections* où sont posées les pierres.

3.1.3 L'intersection

Dans un goban l'intersection joue un rôle majeur et chacune d'entre elles possède diverses informations qu'il est nécessaire de bien définir afin de pouvoir les représenter. Parmi ces informations on distingue deux types d'information : celles **propres à chaque intersection** et celles **communes à toutes les intersections**.

Pour ce qui est des informations communes, on trouve simplement **l'apparence des pierres**. Celle-ci doit en effet être la même indépendamment de l'intersection où se trouve la pierre. D'autres informations pourraient être relevées mais elles n'ont pas d'implication dans notre application. C'est notamment le cas du poids des pierres, de leur animation de pose / de prise, de leurs sons dans divers cas, etc...

Concernant les informations distinctes, celles-ci sont un peu plus nombreuses. Il faut retenir -entre autres- **la position** de chaque intersection ainsi que **sa 'valeur'** (i.e. si c'est une pierre noire, blanche ou s'il n'y a pas de pierre du tout). Pas d'autre information effectivement utilisée n'est à relever ici.

3.1.4 La zone d'infos

Si une partie de Go pourrait se résumer à un *plateau et des pierres*, certaines informations manquent cruellement à cette définition. C'est notamment le cas du **joueur actuel** car le jeu de Go se joue au tour par tour comme la très grande majorité des jeux de plateau. Mais aussi du **temps passé** par chaque joueur depuis le début de la partie. En effet

certaines règles imposent un temps de jeu limité par coup / global ; et lorsque ce n'est pas imposé, il est tout de même souhaitable de disposer visuellement de ces informations.

3.2 Les menus

3.2.1 Définition des menus

Lorsque l'on développe une **application graphique** il arrive un moment où il convient d'intégrer un menu à celle-ci afin de la rendre **navigable et configurable** par l'utilisateur. Malheureusement cette partie peut s'avérer très longue et fastidieuse avant d'arriver à un résultat convenable. Aussi il faut bien souvent choisir entre des menus lourds très personnalisables mais longs à intégrer et des menus plus génériques mais intégrables rapidement. Nous avons opté pour cette seconde méthode qui –bien qu'elle soit plus longue à préparer- permet de **rajouter facilement des menus, des items, etc...** Nous allons détailler ici les menus génériques tels qui *peuvent être intégrés* à n'importe quel autre programme.

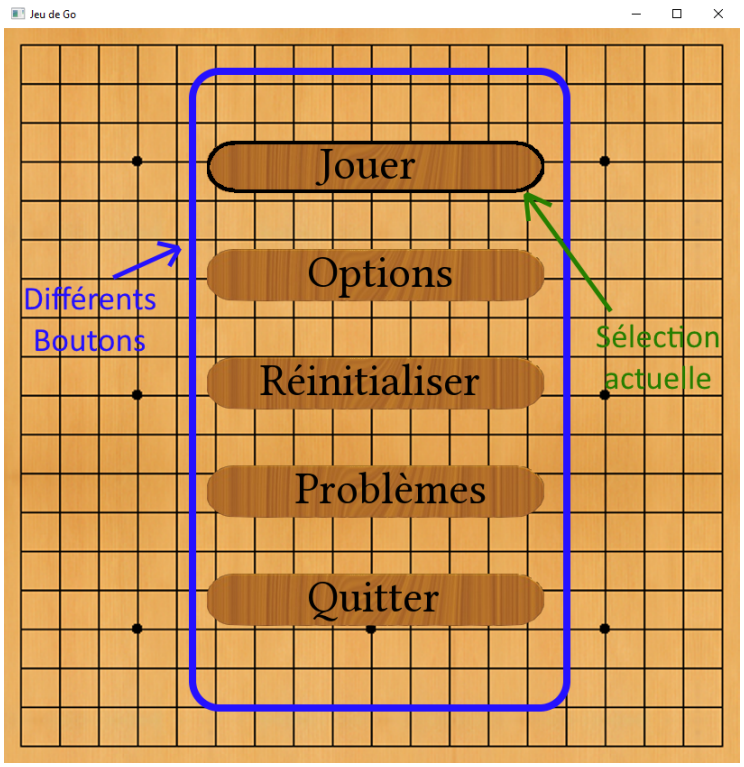

Figure 6: Menu où les boutons ont la même texture mais un texte différent

Pour cela nous devons identifier ce qu'est un menu et ce qu'il permet de réaliser. Tout d'abord un menu est un **élément visuel navigable** qui permet à *l'utilisateur* **d'interagir** avec le programme. Chaque menu est donc composé d'éléments visuels tels que **l'arrière-plan**. De plus chaque menu possède différents *boutons*. Comme dit ci-dessus les menus doivent être capables d'interagir avec l'utilisateur, notamment un menu doit

permettre de **naviguer** librement entre les différents *choix* qu'il propose ainsi que de mettre en valeur le *choix actuellement sélectionné* par l'utilisateur. En plus de pouvoir naviguer parmi différents choix, l'utilisateur peut souhaiter aussi pouvoir **valider** le choix actuellement sélectionné, il faut donc lui en donner l'opportunité.

Une fois que ces principes de bases sont posés on peut réfléchir à la façon dont les menus doivent être créés par le programmeur. Pour créer un menu il faut donc **une position**, éventuellement **menu précédent** et **un aspect** (l'arrière-plan). Le menu ne se suffisant pas à lui-même il faut permettre **l'ajout *d'items*** en son sein. Nous nous sommes arrêté ici car ces menus doivent répondre à des besoins basiques notre projet étant en temps limité.

3.2.2 Définition des items

Comme *un menu* ne peut se passer de boutons nous avons créé ceux-ci et, là aussi, un peu de réflexion a été de mise. Tout comme les menus, les boutons possèdent **un aspect**. En fait ils en possèdent même plusieurs selon **leur état**.

En effet les boutons possèdent différents états qui peuvent parfois se **cumuler**. Notamment ils peuvent être **sélectionnés**, **survolés** par la souris, **enfoncés** par un click, voir ne pas avoir d'état particulier, un peu à la manière des *liens HTML*.

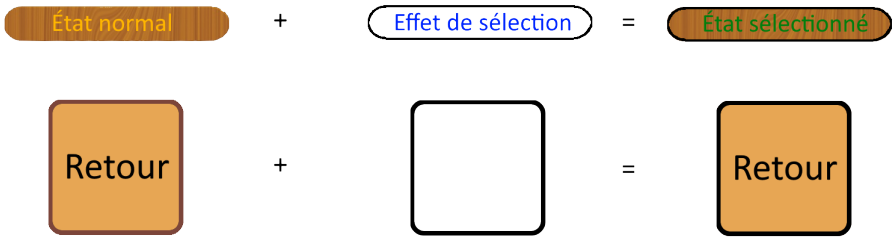

Figure 7: Application d'effets aux boutons

Ces états déterminent donc **l'affichage du bouton** et c'est pourquoi celui-ci possède *plusieurs aspects* même si un seul est affiché à la fois. Le bouton doit contenir en permanence son **état actuel** qui sera changé par le *menu auquel il appartient*. Enfin pour en finir avec l'aspect des boutons ceux-ci possèdent bien évidement **une position** relative au menu auquel ils appartiennent ainsi qu'**une zone de collision** permettant au *menu* de déterminer si l'utilisateur pointe sa souris sur tel ou tel bouton.

Enfin les boutons ne sont pas juste là pour décorer et leur fonction principale est de permettre **d'effectuer une action** lors de la sélection de ceux-ci. Une fois *validé* un bouton peut permettre des actions aussi *diverses* que couper la musique revenir au menu précédent, lancer une partie de Go ou même quitter le jeu.

3.2.3 Les cas spéciaux

Une fois les menus et boutons créés nous nous sommes rendu compte en les utilisant -pour créer les différents menus du jeu- que certaines tâches répétitives persistaient et que les menus pouvaient donc encore être améliorés.

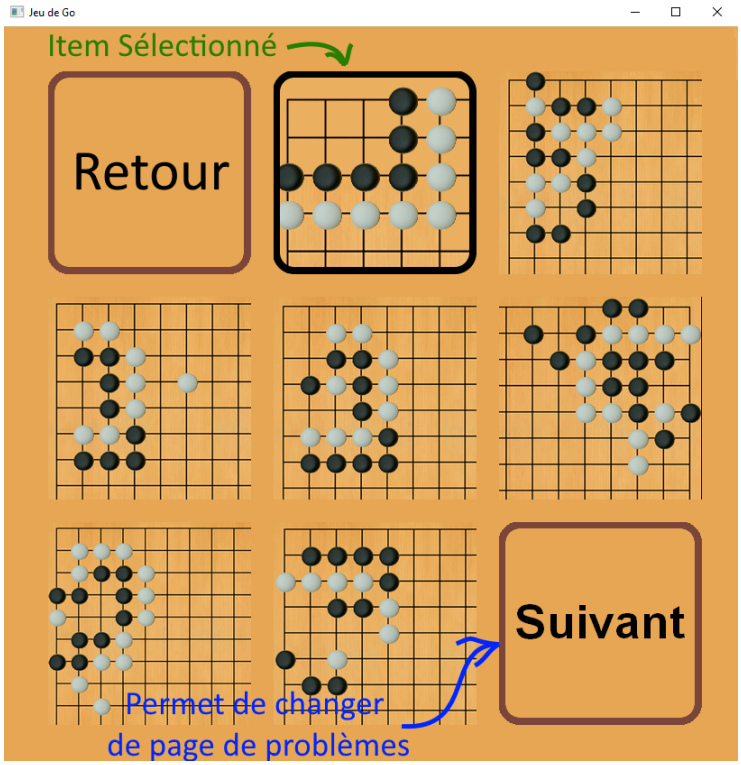

Figure 8: Menu où chaque bouton a sa propre texture

Par exemple dans le cas où *sélectionner un item* ne doit **pas changer son apparence** mais simplement lui appliquer une **mise en valeur** (par exemple un lisserait noir l'entourant dans notre cas). Pour régler ceci et afin **d'alléger la mémoire** dans ce genre de cas de figure assez courant nous avons choisi de dériver notre objet menu en une variante contenant -en plus de tout ce que possède déjà un menu- **l'effet à appliquer** au bouton sélectionné.

Nous avons aussi fait le choix de dériver à nouveau notre menu pour lui ajouter une autre fonctionnalité : **le texte**. En effet nous souhaitions créer des **boutons avec un texte** et nous n'avions besoin que d'une **police de caractère** pour tous les boutons du menu –nul besoin d'alourdir la mémoire avec la même police chargé X fois en mémoire là où une suffirait. Pour aller avec ces deux nouveaux menus nous avons de la même manière dérivé les boutons pour qu'ils puissent contenir soit leur *propre texture* pour le 1^{er} cas soit *leur texte* pour le 2^{nd} cas.

Tout ceci permet dans cette application un léger **gain mémoire** (dans notre exemple celui-ci est de d'un facteur 50) qui, dans le cas d'une application plus volumineuse avec de nombreux menus et surtout des ressources plus gourmandes en mémoires, permet d'atteindre des gains bien plus conséquent.

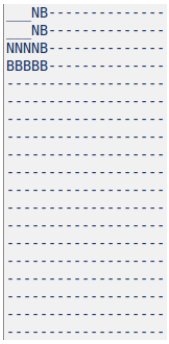

Figure 9: Le fichier .go du problème du 6 en coin

3.3 Les ressources

3.3.1 La musique

La musique a été réalisé à l'aide du logiciel "full studio 10 ". Elle essaie de traduire le plus possible l'univers du Go. Un univers asiatique et combatif.
Les bandes de sons concernant la prise d'un groupe et la pose d'une pierre ont été téléchargé sur un site de banque de son gratuit. Étant, l'un le bruit d'une explosion et l'autre le bruit d'un marteau sur du bois, ils ont été modifié et traité afin de mieux correspondre à nos besoins.

3.3.2 Les images

images quant à elles proviennent de diverses sources, pour la plupart elles ont été traités (détourage, filtres) avant d'être utilisés dans l'interface, certaines ont aussi été créés directement comme par exemple pour les menus.

4 Construction du résolveur de vie ou de mort : le Tsumego

Le Tsumego désigne un problème de go ou sa résolution. Ces problèmes peuvent avoir des formes différentes mais, mène à la même réflexion. **Est-ce que le groupe cible meurt ou vie ?** Tel est la question auquel notre programme doit répondre.

4.1 Création d'un problème : Structure et fonctionnement

Avant de résoudre un problème il faut le définir. Nous avons choisi avec le conseil de notre encadrant, de porter notre intérêt sur le problème du 6 en coin. Nous l'avons modélisé dans un fichier .go dont la structure a été expliqué plus bas (cf. partie sur le parseur). Enfin, à l'aide des algorithmes déjà implémentés sur le jeu nous avons toutes les informations nécessaire sur le comportement de tous les coups possibles dans une partie, en l'occurrence, dans le problème choisi. Il nous reste donc à choisir une structure de donnée puis un algorithme qui traitera cette dernière dans le but de répondre au problème.

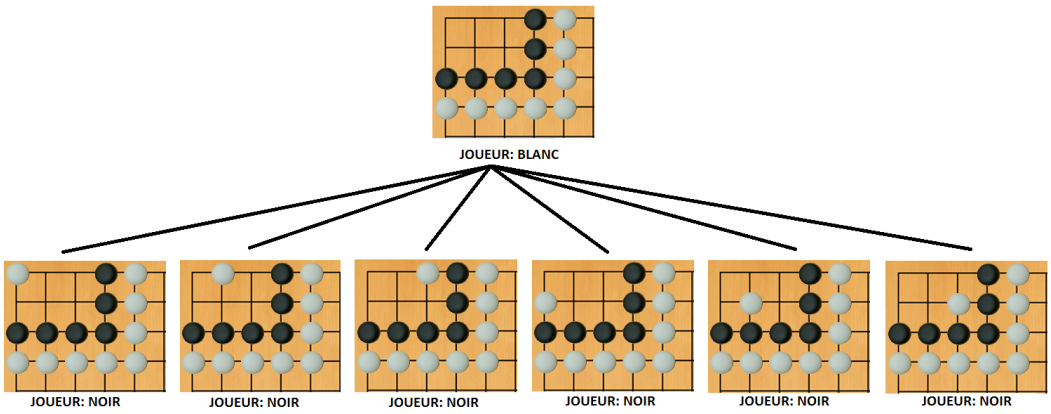

Figure 10: Les fils du premier nœud dans le problème du 6 en coin

Notre choix s'est arrêté sur une arborescence à n fils. Chaque nœuds de l'arborescence contient une valeur booléen dont le but est d'indiquer si ce nœud est gagnant. L'objet goban permet de connaître l'état de la partie à cet instant dans l'arbre. Chaque fils étant un état du jeu à l'instant T+1 (donc des gobans avec un coup supplémentaire). S'ajoute à cela une valeur, qui vaut blanc ou noir, afin de savoir qui doit jouer si on doit générer des fils à explorer ainsi qu'une variable renseignant le nombre de fils pour l'exploration.

4.2 Algorithmes

4.2.1 Algorithme de définition des fils d'un noeud

Pour chaque noeud de l'arbre on calcule tout les coups possibles du joueur et pour chaque coup on crée un goban. On aura de cette manière les gobans des fils qui seront créés successivement à la base de ces premiers.
Le nombre de gobans produit sera intuitivement le nombre d'états vides dans le goban de référence.

Voici les fonctions de l'algorithme :

- **jouer(Goban, VAL, x, y)** : Soit un Goban et un État la fonction contrôle si par rapport au règles du jeu on peut poser la pierre dans le goban.
  Si la pierre peut être posée alors la placer et renvoyer vraie, sinon renvoyer faux.
- **rechercheGroupes(g)** : Ajout de l'élément e en queue du tableau t.
- **itoc(i, goban)** : Soit un entier et un goban i il renvoie les coordonnées (x,y) de l'Etat pointé par i dans le goban.

```
Données : goban : Goban, value : VAL ;
Résultat : renvoyer un tableau de gobans
Variables : gob : Goban, x : entier, y : entier, lisGob : Goban[], Début :
Pour i de 0 à taille(goban) faire
    x←itoc(i, goban)[0]
    y←itoc(i, goban)[1]
    gob←goban
    Si (jouer(gob, value, x, y)) Alors
        rechercheGroupes(gob)
        EliminerGroupesAdversaire(value, gob)
        ajoute(gob, listGob)
    Fin Si
Fin Pour
renvoyer listGob
```

Algorithme 9: définition des gobans fils

4.2.2 Algorithme récursif du Tsumego

L'algorithme du Tsumego et celui qui va résoudre le problème de vie ou de mort choisie par le joueur. Le principe est de vérifier si un groupe de pierre arrive à s'échapper (voir la partie problèmes plus bas).

La cible est représentée par une seule pierre appartenant au groupe ciblé. En effet, lorsqu'une pierre appartient à un groupe, celle-ci ne peut être prise si et seulement si le groupe entier est pris. Les données initiale du Tsumego seront donc une pierre, le goban -contenant le problème à résoudre-, l'état relatif à la cible et le joueur courant.

Si le premier joueur est de la même couleur que la cible alors son but sera de s'échapper. De la manière si le joueur est la couleur opposé il essayera de tuer la cible. Ainsi si le premier joueur gagne alors l'information du noeud sera "vraie", autrement faux.

Les fils d'un noeud auront comme joueur adverse le joueur père, cela implique que si le fils renvoie l'information "vraie" cela signifiera ainsi qu'il aura survécu, et qu'il est alors nécessaire d'essayer un autre coup, ainsi si ce cas se répète le Tsumego parcourra tout les fils jusqu'à en trouver un qui lui renverra un faux qui signifie qu'il le joueur adverse n'aura pas survécu, donc que le groupe aura été capturé, rimant avec victoire, sinon il continuera à développé l'arbre.

```
Données : A : Arbre, cible : Etat ;
Résultat : développer A pour résoudre le problème de vie ou de mort
Variables : i : entier,
Début :
[cas d'arrêt : A est une feuille]
Si (nbF(A) = 0) Alors
    Si (couleur(A) = couleur(cible) et enVie(cible,A) = vraie) Alors
        A.info = 1
    Sinon
        Si (couleur(A)≠ couleur(cible)etenVie(cible, A) = faux)Alors
            A.info = 1
        Sinon
            A.info = 0
        Fin Si
    Fin Si
    renvoyer ;
Fin Si
i←0
Tant que (i < nbF(A) et A.info = 0) faire
    Si (A.joueur = BLANC) Alors
        val←NOIR
    Sinon
        val←BLANC
    Fin Si
    [creation de l'arbre fils]
    A.filsA(GobansFils(A)[i], val)
    Si (enVie(cible,A.filsA)) Alors
        Tsumego(A.filsA, cible)
    Sinon
        [le coup a tué la cible et l'adversaire est le seul qui peut tuer]
        A.info = 1
        renvoyer ;
    Fin Si
    Si (A.filsA.info = 0) Alors
        A.info = 1
        renvoyer ;
    Fin Si
    i←i + 1
Fait
```

Algorithme 10: Tsumego

5 Optimisation & Parseur

5.1 Optimisation algorithmique

Nous avons utilisé deux types d'optimisation :

— La première est déjà implantée dans l'algo au-dessus, c'est ce qui distingue le tsumego brut, du tsumego normal. Elle se situe dans le while, c'est la condition A->Info == 0 ? : elle indique a l'algorithme de sortir de la boucle si on a tué le groupe, c'est ce qu'on appelle plus communément du **Pruning**.

— La deuxième a pour but de faire une économie mémoire dans notre tsumego, en effet au lieu de développer tout l'arbre des possibilités de coups possibles, ce qui ferait exploser la mémoire, nous développons qu'un "étage" de l'arbre, afin d'appliquer le tsumego sur chacun de ses éléments un à un en prenant bien soin d'effacer le fils précédent si la recherche développée à partir de celui-ci s'est révélée négative.

5.2 Optimisation mémoire

5.2.1 Format compression

Afin d'essayer de résoudre des problèmes plus conséquents, nous nous somme penché sur l'optimisation mémoire pour la fonction de Tsumego. En analysant les instances présentes dans le Tsumego on se rends compte que ce sont les différentes instances de Goban qui sont coûteuses en mémoire. Nous avons cherché comment résoudre ceci et avons opté pour une paire de fonctions : une pour compresser le goban et une pour le décompresser.

Il a fallu pour cela définir un format économe pour stocker le Goban en mémoire et pour cela nous avons tenté d'éliminer les informations superflues. Tout d'abord nous avons fait le constat que seuls les états des intersections du goban étaient utiles pour reconstruire l'entièreté de celui-ci. En effet les différents groupes peuvent être retrouvés via la fonction de recherche de groupes, quant aux scores de chaque joueur et l'historique, ces informations ne sont d'aucune utilité pour le Tsumego.

Ensuite nous avons cherché de la même manière à réduire l'espace mémoire de chaque intersection. Tout d'abord en éliminant les coordonnées de celles-ci car elles peuvent être trouvées automatiquement lors de la création d'un Goban. À ce Stade il ne nous restait plus qu'à trouver comment stocker les états. La façon la plus optimal étant de coder

chaque état sur un nombre de bits défini puis de coller les états les uns à la suite des autres.

De prime abord on pourrait penser qu'il nous faut coder jusqu'à 6 états différents (Blanc, Noir, Vide, KO_blanc, KO_noir, Non_Jouable) cf. partie sur l'état d'une intersection de Goban. Par conséquent il nous faudrait 3 bits ($6 < 8 = 2\hat{3}$) pour stocker n'importe quel état. Ors il se trouve que l'état Non_Jouable n'a pas à être codé car il n'est pas stocké (pour les raisons évoqués ci-dessus). De plus les deux états de KO peuvent être résumer en un seul à condition de garder l'information sur le joueur courant à chaque étape du Tsumego. Il ne reste donc plus que 4 état possibles ce qui permet de coder n'importe quel état sur 2 bits seulement.

Grâce à ce format compressé un goban de 9 par 9 intersections peut être stocké sur 20 octets et 2 bits. Lorsque que l'on sait que son penchant non compressé pèse à minima 1ko (1040 octets pour être exact) à taille équivalente, on se rends compte de l'efficacité de la compression.

$$9 * 9 = 81 intersections\,;\ 81 * 2 = 162 bits\,;\ 162 = 8 * 20 + 2$$

Mais pour pouvoir stocker le Goban sous cette forme il faut écrire deux fonctions permettant de passer de la version compressée à celle de base et réciproquement.

5.2.2 Compression du Goban

Pour la compression nous avons écrit l'algorithme qui suit. Cette version est assez bas niveau car nous parlons d'une compression binaire, néanmoins ce code a été simplifié pour faciliter sa compréhension. De même toutes les astuces pour économiser du temps de calcule sont passé sous silence. Nous espérons que le lecteur avisé ne s'offusquera pas de cet écart.

Entrées: goban : Goban à compresser (les cases inutiles sont marquées)
Sorties: Un tableau d'octets
$nb\_rev \leftarrow nbInterUtiles(goban)$;
$nb\_octets \leftarrow plafond(\frac{nb\_rev}{8})$;
$nb\_bits\_bouchon \leftarrow 8 - (nb\_rev \mod 8)$;
$act \leftarrow 0$;
$tmp \leftarrow 0$;
$comp \leftarrow tableau[nb\_octets]$
pour tout $inter \in goban$ **faire**
  $comp[cur] \leftarrow (comp[cur] << 2)$;
  $comp[cur] \leftarrow (comp[cur] + code(inter))$;
  si $tmp == 8$ **alors**
    $cur \leftarrow cur + 1$;
    $tmp \leftarrow 0$;
  finsi
fin pour
$comp[nb\_octets - 1] \leftarrow (comp[nb\_octets - 1] << nb\_bits\_bouchon)$;
Retourner *comp*.

Algorithme 1 : Algorithme de compression

Notations :
— A « N : décalage binaire N bits vers la gauche, appliqué à A ;
— A » N : décalage binaire N bits vers la droite, appliqué à A ;
— Ob10101011 : représentation d'un octet en binaire non signé (dans cet exemple 171) ;
— & : le ET bit à bit de deux entiers sous leur forme binaire ;

5.2.3 Décompression du Goban

Pour la décompression nous avons écrit l'algorithme qui suit. Cette version est assez bas niveau car nous parlons d'une compression binaire, néanmoins ce code a été simplifié pour faciliter sa compréhension. En particulier le cas du dernier octet compressé (celui pouvant contenir des bits de bourrage) a été omis pour éviter les doublons. Comme pour le cas de la compression, nous priions notre lecteur de ne pas s'offusquer de ces quelques imprécisions.

Entrées: model : Goban où les case inutiles sont marquées,
comp : tableau d'octet (goban compresssé)
Sorties: Goban décompressé
$goban \leftarrow creerGoban(model)$;
$nb\_rev \leftarrow nbInterUtiles(goban)$;
$nb\_octets \leftarrow plafond(\frac{nb\_rev}{8})$;
$nb\_bits\_bouchon \leftarrow 8 - (nb\_rev \mod 8)$;
$act \leftarrow -1$;
$masque \leftarrow 0b11111100$;
pour tout $i \in [0, nb\_octets]$ **faire**
$oct\_act \leftarrow comp[i]$;
pour tout $j \in [0, 4]$ **faire**
$val \leftarrow ((oct\_act \& masque) >> 6)$;
$oct\_act \leftarrow oct\_act << 2$;
$act \leftarrow indiceProchaineItersectionUtile(model)$;
$goban[cur] \leftarrow tmp$;
fin pour
fin pour
Retourner *goban*.

Algorithme 2 : Algorithme de décompression

Notations :
— A « N : décalage binaire N bits vers la gauche, appliqué à A ;
— A » N : décalage binaire N bits vers la droite, appliqué à A ;
— Ob10101011 : représentation d'un octet en binaire non signé (dans cet exemple 171) ;
— & : le ET bit à bit de deux entiers sous leur forme binaire ;

5.3 Le parseur

Afin d'implémenter plus facilement tout les problèmes les plus connus et complexes du Jeu de Go afin d'utiliser le Tsumego dessus, nous avons décidé de créer un parseur afin d'importer facilement les Goban nécessaires.

Nous avons défini une nouvelle extension .go, qui n'est au final qu'un fichier texte de quatre caractères différents :
— - " N " : Pour les pierres noires
— - " B " : Pour les pierres blanches
— - " - " : Pour les zones non jouables
— - " _ " : Pour les zones vides jouables

Puis on lit le fichier dans lequel chaque ligne correspond à une ligne du Goban pour remplir celui-ci petit à petit.

```
Données : fichier : String ;
Résultat : Rend un Goban parsé
Variables : goban : Goban, y : entier, x : entier, piece : char ;
Début :
Si (regex(".go",fichier)) Alors
    ouvrir file(fichier) ;
    Si (file) Alors
        x←0
        y←0
        Tant que (file.get(piece)) faire
            Si (piece = 'B') Alors
                goban.coord(x , y).setVal(Blanc)
            Fin Si
            Si (piece = 'N' ) Alors
                goban.coord(x , y).setVal(Noir)
            Fin Si
            Si (piece = '-' ) Alors
                goban.coord(x , y).setVal(Vide)
            Fin Si
            Si (piece = '_' ) Alors
                goban.coord(x , y).setVal(Non Jouable)
            Fin Si
            Si (piece = 'B' ) Alors
                goban.coord(x , y).setVal(Blanc)
            Fin Si
            Si (piece = '/n' ) Alors
                y++
            Sinon
                "Caractère lu inconnu à la position (" x ", " y ")!"
            Fin Si
            x < TGOBAN ? x++ : x←0

        Fait
    Sinon
        "Impossible d'ouvrir le fichier ! "
    Fin Si
Sinon
    "Fichier introuvable"
Fin Si
Renvoyer goban
```

Algorithme 11: Parseur

6 Conclusion

6.1 Bilan

Nous sommes heureux d'annoncer que le Tsumego fonctionne pour le problème du 6 dans le coins, il détermine donc si le groupe va survivre ou pas. Du début jusqu'à la fin de ce projet les membres sont restés actifs ce qui nous à permis d'arriver à faire fonctionner notre algorithme de solution sur certains problèmes de Tsumego. C'est une réussite personnelle pour chaque membre car ce sujet de TER nous paraissait insurmontable en vu des échecs des précédentes années. Mais ce projet nous a intéressé et au delà du défi de réussir il nous a permis de nous enrichir autant techniquement que socialement. Au fur et à mesure du temps nous avons appris à nous connaître et à nous adapter avec le caractère des uns, parfois solitaire, et avec celui des autres. Et c'est ce que nous noterons de plus important lors d'un travail de groupe c'est de pouvoir s'adapter pour maintenir la cohésion.

6.2 Perspectives

Nous aurions pu améliorer la performance du Tsumego en éliminant les gobans des fils. Ceci permettrait une économie de mémoire lors de l'appel récursif du tsumego. En effet nous n'aurions pas à recalculer l'intégralité des fils mais seulement ceux qui nous intéressent.

De plus nous avions la possibilité d'utiliser le résultat du Tsumego pour créer une intelligence artificielle. Celle-ci aurait dû être capable de jouer contre une personne sur des problèmes de petite taille. Nous disposons de deux possibilités pour mettre en place cette IA. La première serait de lancer le Tsumego une seule fois puis d'utiliser l'arbre développé pour répondre aux coups du joueur. La seconde, plus économe en mémoire, serait l'utilisation d'une version alternative du Tsumego gardant en mémoire seulement le coup gagnant. Elle lancerait alors le Tsumego après chaque coup du joueur.

Un tout autre cas de figure est aussi envisageable : ajouter à notre application une fonctionnalité de jeu en ligne. Cette possibilité permettrait de nous initier à la communication internet d'une application et serait un prolongement intéressant pédagogiquement parlant. De plus en poursuivant ce projet nous pourrions envisager de le publier en ligne et permettre à des amateurs de Go de jouer sur un logiciel libre et gratuit !

6.3 Apport personnels du projet

Effectuer ce projet fut, pour chacun de nous, un gain en connaissances énorme. Il nous a permis d'en apprendre plus dans de nombreux domaines ainsi que de se perfectionner dans ceux que nous connaissions déjà. Cet apport ne s'est pas cantonné à l'informatique mais il a touché d'autre niveaux tels que la mise en page en Latex, dans le graphisme, la musique pour certains, la rédaction de rapport, etc.
Au final les projets TER sont un apport considérable dans nos études. Ils nous mettent face à la réalité du travail en groupe et nous apporte de nombreuses compétences. C'est l'un des points fort de notre formation et cela fait même parfois émerger des vocations pour certains.

7 Remerciements

Nous tenons à remercier notre professeur référent pour ce TER : M. Pompidor qui a su se montrer présent pour nous tout au long de ce projet. Son soutient et ses précieux conseils nous ont été d'une aide précieuse.

Nous souhaiterions aussi remercier M. Dicky –enseignant d'algorithme au second semestre- pour les quelques astuces qu'il nous a fourni. La rigueur de son travail dans l'UE HLIN401 a été pour nous un exemple tout au long du projet.

Enfin nous remercions l'intégralité de nos enseignant(e)s qui de par leurs cours nous ont d'une certaine manière aidé dans ce projet.

8 Annexes

8.1 Implémentation de la partie graphique

Pour implémenter une fenêtre de jeu, il fallait répondre au problèmes suivant :
Avoir un objet qui représente :

- une intersection dans un goban.
- un goban.
- les informations de la partie.
- la fenêtre de jeu.

8.1.1 Le Jeu

L'intersection

Pour répondre à ce problème nous avons implémenter une classe nommée : "Square". Cette objet hérite de la classe SFML "drawable" afin d'être "dessinable". Un objet square représente une intersection. Nous avons donc un couple de deux entiers pour les coordonnées (constante). Une valeur (Black, white ou None) pour savoir quoi afficher.

Enfin en attribut de classe, en variable static, les sf : :textures$\hat{1}$ d'images à afficher (pierre noir ou blanche).
On remarquera que les coordonnées n'apparaissent pas dans les attributs de la classe Square tout simplement car elles sont inutiles. En fait l'objet sf : :sprite$\hat{2}$ qui dessinera la pierre ou rien, contient déjà des coordonnées (relative à la fenêtre).
De plus, pour la méthode "draw", qui est définis plus haut dans la hiérarchie (dans Drawable), sera virtuelle pour pouvoir être remplacée par celle définie dans Square.

Le Goban

La classe Board, qui hérite aussi de drawable, représente un goban. Elle a donc comme attribut un objet Goban afin de manipuler toute les méthodes liées aux règles, à la recherche de groupes et à leurs manipulations.

Pour ce qui concerne l'affichage, nous aurons aussi en attribut un tableau d'objets "square" (qui sera dynamique). Une image, pour représenter graphiquement le goban. Un objet sf : :view$\hat{3}$ afin de pouvoir zoomer sur une partie du goban.

Enfin, quatre variables pour les effets sonores, eux aussi présents dans la bibliothèque SFML.
Un sf : :SoundBuffer[4] et un sf : :Sound[5] pour chaque son. Un lors de la pose d'une pierre, l'autre lors de la capture d'un groupe.

Cet objet doit pouvoir interagir avec l'utilisateur. Poser des pierres dans le respect des règles du jeu. Pour se faire nous utilisons une méthode "click" qui vérifiera que le bouton pressé est bien le gauche grâce à la variable "event" de type "sf : :Mouse : :Button"[6]. Ensuite, on vérifie que les coordonnées du pointeur sont bien dans la zone de jeu et valide. Enfin on pose la pierre dans le goban en relançant la méthode de recherche de groupe pour mettre à jour les groupes. On joue un son lors de la pose, un autre si un groupe est éliminé. Enfin on applique la méthode load qui va convertir le goban en tableau de square afin de dessiner le nouveau goban.

Les Informations

La classe infos nous permet d'afficher les informations sur la partie en cours : le joueur actuel (blanc ou noir) et le temps de jeu pris pour chaque joueur.

Pour le temps nous utilisons une autre classe nommée "Timer" qui hérite des classes SFML, sf : :Clock et sf : :Text. Les attributs sont naturellement, un objet sf : :Time, un "sf : :Font" pour charger une police d'écriture et un booléen pour savoir si le temps est en "pause" ou non. Pour les mêmes raison que les objets square, lors de l'affichage du temps, on utilise un objet sf : :Text qui à une position, une sf : :Color[7], une taille de caractère etc.. Il suffit donc de les définir lors de l'appel du constructeur.

Nous dessinons donc chaque temps (blanc et noir) et à chaque changement de joueurs la méthode "pause()" est appelée. La méthode "setCurPlayer" permet en passant en paramètre la Square : :Value (blanc ou noir) de savoir quel joueur doit jouer.

La fenêtre de Jeu

La classe Game_Window qui hérite de Screen, va avoir le rôle de chef d'orchestre lors du lancement d'une partie de Go. Elle a comme attribut une board, un infos et une value car a la valeur du joueur courant.

Quelques mots sur la classe "Screen". Elle hérite de "sf : :Drawable". On y retrouve deux méthodes virtuelles publics : "draw", pour que tout ce qui hérite de "Screen" puisse redéfinir cette méthode (tout comme dans square). Et "Run", ce sera la méthode qui sera utiliser pour gérer les intersections et les évènements courants (un clique, un appuie sur un touche du clavier, etc..). Cette méthode retourne un "Screens". C'est une information afin de pouvoir gérer les différente fenêtre.

Dans Game_Window nous retrouverons donc la méthode "Run", qui va s'occuper d'aiguiller les évènements à l'aide d'un "switch". Pour ce qui concerne le clique on va appeler la méthode "click". Elle fera un bref appel à la méthode "click" définie dans la classe board en transmettant les données, tel que la position et le joueur actuel qui sera modifiée lors de cette exécution. Bien évidemment, la méthode click de board étant un booléen, il sera utilisé pour le changement de tour du joueur (si le clique était autre chose que poser une pierre). Pour les évènements clavier on appel la fonction "keyPressed", qui en fonction de la touche exécutera une action, par exemple un "ctrl+Z" retire le coup qui vient d'être joué un "Escape" mets en pause le jeu et ouvre le menu de pause (nous expliquerons son fonctionnement plus tard).

8.1.2 Les menus

Pour implémenter les menus nous avions besoin d'une image de fond et de plusieurs boutons afin de pouvoir naviguer parmi différentes fenêtres comme le jeu, les problèmes, les options...
Pour y répondre nous avons créé deux classes. Une étant le menu "Menu" lui même et une autre gérant les boutons : "Choice".

Le Menu

La classe "Menu" qui hérite de screen (vu précédemment) à pour attributs une sf : :Texture qui sera la texture de l'arrière plan. Un sf : :Sprite pour dessiner à l'aide de la méthode "draw" la texture. Enfin une variable de type Screens (énumération). On note aussi la présence de deux attributs accessibles par la classe et tout ses héritiers (protected), ce sont les objets "Choice" du menu, rassemblés dans un vecteur et un pointeur pour le Choice courant. Pour les méthodes, draw, Run, click auront les mêmes logiques que précédemment.

Une méthode addItem est présente, elle prend en paramètre un objet Choice qui sera ajouté au vecteur. D'ailleurs la présence de std : :reference_wrapper" permet d'insérer les éléments par référence dans le "conteneur" ici le vecteur.

Les Boutons

La classe "Choice qui hérite de sf : :Drawable aura comme attributs tout ce qui est nécessaire pour afficher un bouton (deux pointeurs sf : :Texture et deux sf : :Sprite). Pourquoi deux pointeurs sf : :Texture ? Tout simplement car nous voulions afficher une image différente lors de la sélection d'un bouton (ici ce sera un léger liseré noir qui entourera le bouton).
Pour savoir si ce dernier est sélectionné, nous avons un booléen "selected" pour renseigner cette information.

Enfin, nous trouverons l'attribut sf : :Text pour afficher le nom du bouton. Mais que fait ce bouton ? il y a un attribut "std : :function<....". En fait c'est cette fonction (appelée lambda fonction) qui s'exécutera. Elle renvoi un entier, plus précisément un "Screens".

Les menus spéciaux

Ces deux classes héritent de la classe "Menu". Ce choix particulier de créer deux autres classes de menus réponds à une vision des choses. Nous nous sommes posés la question : Vaut-il mieux instancier chaque images pour chaque boutons ou bien créer des menus avec des boutons identiques ? Pour nous, un menu affiche une ensemble de choix (les choices). Ces choix sont graphiquement identiques : même taille pour les boutons, une image et une sélection unique pour chaque bouton.
Ayant le menu "Problème" qui devait contenir des choix graphiquement différents des autres menus nous avons donc créé une autre classe de menus.

Leurs fonctionnements techniques sont identiques à ceux de leur père, menu. Cependant menu_miniature est plus léger en termes de variables. Par exemple, il n'y a pas de "sf : :Text" car chaque bouton affiche des images de problèmes à charger, l'utilité d'un texte était donc superflu.

Les boutons spéciaux

Ces deux classes héritent de Choice, comme nous l'avons expliqué pour la partie précédente le choice_miniature (qui concerne le menu "problème") contiendra une référence de "sf : :Texture" pour chaque image de problèmes à dessiner. L'autre classe, choice_simple contiendra un sf : :Text et un sf : :Font en plus des attributs de la classe mère (même image pour tout les boutons).

8.1.3 Le Programme Principal

Le programme principal (le main) doit gérer toutes les fenêtres de notre programme. On remarquera la présence d'une macro qui active le multi-threading. Nous nous intéresserons plus particulièrement à la fonction "renderingThread" qui va gérer les différents "screens" en fonction de ce que demande l'utilisateur. Ceci est géré grâce a un "while" qui vérifie que la fenêtre courante est différente de l'appel de fermeture. Toute les fenêtres seront présentes dans un "std : :vector" d'objet "Screen" vu dans la partie I. Ajoutons que ces menus seront dessinés en effaçant la fenêtre courante à l'aide de la méthode ".clear" sur l'objet window qui est de type "sf : :RenderWindow". Ensuite avec la méthode "draw" nous dessinerons la nouvelle fenêtre.

Pour plus de clarté nous avons séparé la gestion de ces menus avec leurs déclarations qui est décrite dans un objet "go_solver". Cet objet contient donc comme attributs un

"std : :vector" de pointeur sur des screens, un "Screens" pour savoir quel est la fenêtre courante.
Un "std : :vector" pour la musique, si nous avons plusieurs piste par exemple. Une musique courante aussi. Une fenêtre de jeu spécifique à l'objet Game_window" que nous avons vu qui sera appelée en cas de début de jeu. Enfin un "std : :thread" pour le lancement du tsumego automatique afin de pouvoir interagir dans le programme pendant que le traitement s'effectue (qui peu être assez long).
Cela permet entre autre, de fermer les programmes "proprement" à partir des boutons dédiés.
La variable "target_tsumego" permet de renseigner les informations concernant la cible à tuer (plus de précision sur la partie tsumego).

8.1.4 Glossaire

1. Une texture est un objet qui réfère à une image.
2. Une sprite est un objet qui est dessinable.
3. Une view et une vue.
4. SoundBuffer est un objet de la classe SFML.
5. Sound aussi
6. Énumération dans SFML des évènement de souris.
7. Ce type présent dans la SFML utilise une méthode qui n'a pas la même écriture sur Linux et sur Windows, d'où la présence d'une macro définissant setFillColor en setColor si nous somme sur Windows.