Compte Rendu.
Reversi: Advanced Computer Player

GONZALEZ ZAMBRANO, Boris Antonio
Master 1 Informatique - Cryptologie et Sécurité Informatique
4TIN707U - Programmation
UNIVERSITÉ DE BORDEAUX

December 1, 2017

Contents

1 Chapitre 1

Avant de finir le projet nous avons fait quelques changements algorithmiques, mais le principal était le changement de la fonction heuristique, ce qui est le plus important car il nous donne de la puissance au moment d'implémenter la « Intelligence Artificielle » dans le jeu.

Après avoir faire ça nous avons changé les valeurs que nous utilisons pour exécuter l'algorithme minimax avec alpha – beta. Nous avons implémenté le *aspiration search* qui est une petite amélioration à l'appelle de la fonction minimax alpha - beta dans laquelle nous changeons les valeurs de alpha et beta. Normalement l'appel à cette fonction est ***minimax_ab(board, depth, player, alpha, beta);*** alpha vaut –INFINITE et beta vaut INFINITE, mais avec le *aspiration search* on n'utilise plus les valeurs d'infinité. Nous le remplaçons alors par une valeur qui va être une approximation du résultat attendu, et un « fenêtre » qui va être la déviation qu'on pense que cette valeur puisse avoir.

Avec *aspiration search* nous allons explorer moins nœuds puisque les limites de alpha et beta seront déjà posés dès le début. L'unique problème que pourrait arriver ce que les valeurs heuristiques ne soient pas dans nos hypothèses, dans ce cas la recherche va échouer. Mais comme valeurs nous avons utilisé 10 et comme fenêtre 5, puisqu'on pense qu'aucun nœud aura plus de 15 mouvements possibles. Avec cette implémentation nous allons diminuer le temps d'exécution.

1.1 La fonction Heuristique

Au début nous avions une fonction heuristique très simple qui calculait la différence de points, ou de pierres dans le tableau, entre le joueur et son adversaire. Mais cette heuristique n'est pas très efficace pour le jeu Reversi, car nous pouvons avoir une grande différence de points à notre faveur, mais si notre adversaire a plus mouvements possibles et avec une meilleure position, il peut gagner après quelques mouvements.

Les fonctionnalités les plus importantes pour le jeu Reversi sont la stabilité, la mobilité et la parité des disques ou pierres. En particulier on peut dire que les pierres stables ne peuvent pas être tournées par les adversaires, et que les coins sont les positions qui donnent une plus grande stabilité. Alors, avoir moins possibilités ou moins mouvements que notre adversaire est dangereux, car cela incrémente les possibilités de perdre un coin au futur. Les coins fonctions comme des ancres pour stabiliser les pierres qui sont proches d'elle.

C'est pour ça que nous avons modifié cette fonction heuristique pour qu'elle puisse calculer la quantité de mouvements possibles qu'aura chaque nœud étudié pour l'algorithme de recherche d'arbre. On alloue cette valeur à une variable qui va être les valeurs heuristiques des nœuds. Après, on teste si dans la liste de mouvements possibles il existe au moins un coin, et dans ce cas, on va augmenter la valeur heuristique pour chaque coin qu'on trouve dans les mouvements

possibles des nœuds. Nous avons pris en compte aussi les positions qui peuvent donner une opportunité à notre adversaire d'obtenir un coin. Nous l'appelons les frontières, qui sont les positions adjacentes aux coins. Avec la même stratégie d'avant, nous avons cherché des frontières dans les mouvements possibles de chaque nœud et nous diminuons la valeur heuristique pour chacune qu'on trouve.

+	-					-	+
-	-					-	-
-	-					-	-
+	-					-	+

2 Chapitre 2

Pour ce chapitre nous allons expliquer les techniques d'implémentation que nous avons utilisée pour développer le projet. Pour être un projet si long et un peu vaste nous devions connaître plusieurs techniques et outils de programmation, plus spécifiquement du langage C. Mais nous devons souligner les points les plus importants:

- Structure de données.
- Fonctions.
- Libraires.

2.1 Structure de données.

Pour gérer les « objets » du projet, nous avons utilisé des structures de données, avec lesquelles nous pouvons organiser les composants de chaque objet dans un même ensemble. Ça veut dire, que chaque instance des objets va avoir les mêmes éléments et nous allons le stocker avec la même structure. Pour faire ça nous avons créé une structure principale, appelé ***board_ t,*** que nous avons défini de la forme suivante:

```
struct board_ t
{
    size_ t size;
    stone_ t player;
    stone_ t **grid;
    moves_ t *moves;
} ;
```

Chacun des ***board_ t*** va avoir les mêmes éléments:

- **size**: La taille du board. Elle doit être entre 2 et 8.
- **player**: Ici on va avoir l'identificateur du joueur courant. C'est de type ***stone_ t***, qui est aussi une structure.
- **grid**: Ce sont les cases du board. Cet élément sera un pointeur de 2 dimensionnes, pour mieux gérer les lignes et colonnes. Il va être aussi du type ***stone_ t***.
- **moves**: Les mouvements qui va avoir le jouer dans le board. Il va être du type ***moves_ t.***

Cette structure est la plus importante du projet puisque c'est avec ces composants qu'on peut jouer. On implémente la plus part des fonctions qui sont en charge du jeu dans ***board_ t***, aussi c'est ici où on applique les mouvements pour jouer, c'est le cœur du jeu.

Autres structures que nous avons utilisés sont les suivantes:

```
struct moves_ t
{
    size_ t size;
    move_ t *list;
} ;
```

Nous avons créé ***moves_ t*** pour gérer les mouvements qui va avoir le joueur et on peut le résumer comme une liste qui va enregistrer les coordonnes des mouvements possibles.

```
struct node_ t
{
    int value;
    move_ t move;
} ;
```

Nous avons créé une dernière structure auxiliaire dans le module ***player*** appelé ***node_ t,*** qui va nous aider au moment d'implémenter les algorithmes de recherche d'arbre. Chaque nœud des arbres qu'on va créer au moment de chercher le meilleur mouvement possible va avoir un ***move*** ou un mouvement avec des coordonnes des lignes et colonnes dans le tableau, et aussi un ***value*** ou valeur numérique entière heuristique, que nous allons le donner après exécuter la fonction qui calcule l'heuristique. Cette fonction va nous aider au moment d'implémenter les algorithmes de recherche d'arbres.

2.2 Fonctions

Pour ce projet nous avons créé plusieurs fonctions (presque 50), certaines plus importantes que d'autres, mais ils font tous un travail nécessaire. Nous allons expliquer un peu les plus importantes et ceux qu'on a créés de maniéré auxiliaire pour nous aider au moment de l'implémentation et codification :

- ***board_ is_ move_ valid***: Nous allons donner un mouvement à cette fonction (coordonnes des lignes et colonnes), et cette fonction va nous dire si cet mouvement est valide, en fonction du joueur courant et les pierres de l'adversaire.

- ***board_ play***: C'est la fonction principale du mécanisme de jeu. On lui donne un mouvement (coordonnes des lignes et colonnes) et on le place dans ces coordonnes, en tournant les pierres de l'adversaire qui sont entre la pierre qui vient d'être placé et les autres pierres du joueur courant.

- ***board_ load***: Nous allons utiliser cette fonction pour charger un fichier existant, mais elle va accepter seulement si le fichier est un tableau avec le format du jeu. Elle a été créée pour charger des fichiers des tableaux sauvegardés, en nous donnant l'opportunité de quitter et continuer un match quand nous voulons.

- ***board_ set_ player***: Cette fonction est très simple, mais nous l'avons créé pour faciliter le fonctionnement du jeu, plus spécifiquement au moment de faire le changement de joueurs après ces mouvements, pour changer le joueur après avoir sauvegarder le jeu et avant de le quitter. Elle va avoir comme argument un tableau et un joueur spécifique, et elle va retourner le même tableau, mais nous allons changer le joueur courant par ce qui a été donné comme argument.

- ***minimax_ ab( ) et minimax( )***: Ces fonctions auxiliares ont été créées pour gérer la vraie implémentation des algorithmes de recherche d'arbres, en utilisant l'algorithme Minimax et Minimax avec l'élagage Alpha - Beta respectivement. On les envoie comme arguments un tableau, un entier positive pour representer la profondeur qu'on veut utiliser dans l'algortihme et le joueur courant du tableau. Pour Alpha - Beta nous l'envoyons de plus les valeurs qu'on veut utiliser pour Alpha et Beta respectivement. Nous faisons les appels de ces fonctions dès ***minimax_ player*** et ***minimax_ ab_ player*** respectivement dans lequel on reçoit les paramètres dès la fonction principale du jeu. Les deux fonctions retournent le meilleur noeud que les algorithmes peuvent trouver.

- ***version***: Une fonction très basique qui va imprimer la version du logiciel et quelques autres informations. Elle est statique et elle ne va rien retourner.

- ***playervalid***: Cette fonction a été créée pour savoir si le joueur courant a au moins un mouvement possible dans le tableau qu'on le donne comme argument. Nous l'utilisons dans le module principal ***reversi***, dans la fonction ***game.*** Elle va parcourir tout le tableau et va chercher un mouvement valide avec l'aide de la fonction ***board_ is_ move_ valid***. Si elle trouve un mouvement elle va retourner une booléenne ***true.***

- ***printgamemode:*** C'est une fonction qui va nous faciliter le travail au moment de faire l'impression de l'information de jeu au début de l'exécution. Elle va prendre comme argument le type de jeu que l'utilisateur a choisi, et elle va imprimer l'information concernant à la valeur du type de jeu. (tous les deux joueurs seront humains, les deux seront gérés par l'ordinateur, ou une combinaison d'humaine et ordinateur.)

2.3 Librairies

Nous avons utilisé plusieurs librairies, par exemple les plus importants et les plus connues ***stdio*** et ***stdlib,*** mais nous avons utilisé aussi des autres librairies comme les suivantes:

- ***err.h***: Nous avons implémenté cette librairie dans les modules ***board*** et ***reversi***, et on utilise les fonctions ***err*** et ***stderr,*** avec lesquelles nous gérons les erreurs dans le programme. Plus spécifiquement, si on doit imprimer un message d'erreur lorsqu'on trouve qu'un argument est incorrect au moment d'exécuter le programme et quand nous trouvons qu'une allocation de mémoire n'a pas été bien fait.

- ***string.h***: Nous avons utilisé plusieurs composants de cette librairie, comme le type de données ***size_ t*** que nous utilisons pour les variables qui vont avoir les tailles des tableaux et des listes des mouvements. ***NULL*** qui nous permet définir un pointeur vide ou nulle. ***strcat*** que nous avons utilisé pour faire une concaténation des chaînes au moment de transformer ce que nous avons dans un tableau ou une liste à une chaîne de caractères.

- ***getopt.h***: ***getopt.h*** forme partie d'une librairie plus grande qui s'appelle ***unistd.h,*** mais comme nous allons seulement utiliser une composante de cette fonction nous avons appelé seulement à ***getopt.h.*** Elle va nous aider avec le parseur que nous avons utilisé dans le main de ***reversi.c,*** avec la fonction ***getopt( )***.

3 Chapitre 3.

3.1 Qualité du code.

Pour assurer la qualité du code dans ce projet, nous avons pris en compte quelques mesures pour que le code puisse être plus compréhensible ou plus agréable au moment où une autre personne le lit. Par exemple, nous avons essayé de maintenir la clarté dans le code, car si le code est facile de comprendre, alors il serait facile de changer ou améliorer. Pour ça nous avons utilisé une seule structure du codage pour tout le code, une mémé format au moment d'utiliser les fonctions du langage et une bonne implémentation du *code style.*

Par exemple, au moment d'écrire une nouvelle fonction ou d'utiliser une *for*, *while*, etc., nous l'écrivons comme ça:

```
int foo (int a, int b)
{
        ...
        for ( i = 0 ; i < size ; i++ )
        {
                ...
        }
}
```

Ainsi, à notre avis, le code sera plus facile de lire. Nous avons pris en compte aussi le *code style* du C, ça veut dire que nous avons suivi les règles au moment d'écrire le code, en respectant la maniéré dans laquelle nous utilisons les indentations (de 2, 4 ou 6 espaces), les noms qu'on utilise pour les variables et fonctions (toujours en anglais et avec du sens avec le programme, qui décrit un peu son fonction), nous ne surpassons pas 80 caractères par ligne du code, en donnant des espaces au moment d'utiliser les opérateurs du langage (par exemple: *if (size == 4 | | size == 6)*), au moment de faire les # *include,* on les ordonne alphabétiquement et on les divise selon le type de librairie, et quelques autres règles.

Nous essayons aussi d'être consistant, si au moment du codage nous devons faire quelque chose similaire à une autre qui est déjà existant, nous le faisons d'une maniéré similaire ou égal, en utilisant des noms similaires, les mémés paramètres et avec une structure du code pareil.

Nous avons aussi documenté le code. Pour chaque fonction nous avons écrit une petite description de ce qu'elle fait, ces paramétrés et ce qu'elle retourne (dans le cas qu'elle retourne quelque chose). Par exemple pour la fonction *board_ init* nous avons écrit:

/ this function creates and initialize the board with the 4 initial stones placed in the center, we use the parameter size to create the board and in the end we return it already initialized */*

*board_ t *board_ init (size_ t size)*

{

...

}

3.2 Modularité du code

La modularité du code n'est pas importante pour l'ordinateur ou pour le compilateur, puisqu'il peut exécuter le logiciel même si nous avons tous les instructions dans le mémé *.c*, mais nous utilisons les outils de modularité pour nous, pour les développeurs et pour ceux qui vont voir et lire notre code.

Si on prend en compte la modularité du code, on pourra nous concentrer dans une petite partie du code au moment du codage, ainsi sera plus facile de faire le programme et dans le cas qu'on a des erreurs, comme ça sera plus facile aussi de les trouver et *debugger*. Pour ça nous avons implémenté dans notre programme quatre modules, *reversi, board, player,* et *moves.* Chacun a son propre *.h*, ou *header,* dans lequel on définit les fonctions et les structures que nous allons utiliser dans ce module.

Ils ont aussi ses propres *.c*, qui sont le cœur des modules. Ici on implémente et on utilise ce qu'on a défini dans les *.h*. Dedans chaque *.c* nous avons des fonctions, dans lesquelles nous divisons le travail et les processus que nous devons faire pour que le programme marche comme il doit. Chaque fonction a un travail différent, et ils peuvent être appellés et utilisés pour des autres fonctions, dans les mémés *.c* ou dans une autre. Avec cette *feature* nous pouvons dire que notre programme est modulaire et récursive.

Nous avons implémenté aussi une maniéré de « cacher » les structures que nous avons utilisé dans le programme. Par exemple, la vraie structure de *board_ t* seulement peut être vu par le module *board*, et pas pour les autres modules, ni pour le module principal *reversi*, qui est le module qui va à avoir contact avec l'utilisateur. Ainsi on peut assurer la sécurité de la structure, l'utilisateur peut avoir accès aux éléments de la structure, mais il ne peut pas voir comment ça marche ni comment elle est composée. Les autres modules peuvent aussi avoir accès aux éléments des structures mais seulement à travers des fonctions qui vont retourner ce qu'ils demandent. Pour faire ça nous devons faire l'inclusion des modules avant de commencer à coder dans les *.c* et les *.h*, avec l'instruction # *include* qui va nous permettre d'utiliser les fonctions qui sont dans un autre module.

Pour faire ça on a implémenté la fonctionnalité *typedef struct* dans le *.h* qui va avoir la structure dedans son *.c.* Par exemple:

typedef struct board_ t board_ t;

dans *board.h*, et:

```
struct board_ t board
{
    size_ t size;
    ...
} ;
```

dans *board.c* et *bitboard.c.*

Évidemment chaque module a un *.o* ou programme objet après la compilation, qui est le programme en langage d'ordinateur qui va être lu par lui-même avant d'exécuter le programme. Au moment d'implémenter la modularité dans notre code on peut avoir des avantages, par exemple nous avons réutilisé les modules au moment de faire *bitboard.c* qui est une implémentation une peu différent de *board.c* mais avec la mémé structure et avec la mémé fonctionnalité, en fait nous avons utilisé le même *.h* pour le deux. Ça nous permet de ne pas avoir besoin d'écrire le mémé code plusieurs fois, en réutilisant les fonctions qu'on a déjà codé, puisqu'on peut l'appeler toutes les fois que nous voulons.

Pour faire l'union entre tous les parties et tous les modules du programme nous avons utilisé un *Makefile*, qui est un fichier spécial qui contient des instructions *shell,* et qui nous exécutons avec le commande *make*. Avec cette commande, nous allons exécuter les instructions qui sont dans le *Makefile.* Dans notre *Makefile* nous avons mis tous les *.c* et *.h* qui nous devions utiliser pour créer chaque exécutable que nous voulons. Par exemple pour créer l'exécutable reversi-1 nous avions l'instruction suivante:

reversi-1: reversi.o board.o moves.o player.o

$ (CC) $ (CFLAGS) -o $ @ $ ^ $ (LDFLAGS)

Ainsi on peut faire l'union entre tous les modules au moment de compilation (on peut regarder que pour créer l'exécutable *reversi-1* nous avions besoin des modules *reversi, board, moves* et *player).*

Nous devons remarquer aussi la maniéré dans laquelle nous avons ordonné le fichier, car nous avons utilisé un fichier qui s'appelle *include/* et dedans nous avons tous les *headers,* et un autre appelé *src/* avec tous les .c.

3.3 Extra tests.

Pour assurer que tout le programme marche comme il doit, nous avons fait de petits tests pour chaque fonction dans tous les *.c*, et avec tous les entrés possibles, pour voir le comportement du code et s'il y avait des erreurs ou bugs. Par exemple nous avons essayé d'appeler la fonction

board_ init(size_ t size) avec différentes tailles comme argument. Au moment de jouer nous avons testé que le *parser* dans la fonction *human_ player* n'accepte que les entrées possibles, (a5, a 5, A5, Q, q, etc.) si et seulement si l'entrée est valide dans cette instance du jeu. Avec *board_ save* et *board_ load* nous avons essayé de sauvegarder et de charger tout type de *board,* etc.

Quand nous étions proche de finir le projet, nous avons testé le programme en jouant avec des personnes que ne connaît pas le logiciel, et nous leur avons demandé de comment améliorer la fonctionnalité et l'interface du projet, et des opinions pour savoir si le programme était amical et facile d'utiliser pour l'utilisateur.

3.4 Utilisation des outils pour améliorer le code.

Pour augmenter l'efficacité de notre programme nous avons utilisé quelques outils qui nous permettrons de réduire les erreurs de mémoire au moment d'utiliser mémoire dynamique, et aussi pour diminuer les temps d'exécution.

A notre avis l'outil le plus important était *Valgrind,* qui nous permettre debouger les problèmes de mémoire dans le programme et la performance du code. Avec *valgrind* nous pouvons vérifier des points importants dans le fonctionnement du programme, comme l'utilisation de mémoire qui n'a pas été initialisé, la lecture / écriture de mémoire qui a été déjà libéré, la lecture / écriture hors des limites des blocs de mémoire, erreurs de segmentation, fuites de mémoire, etc.

Mais *valgrind* a une désavantage très important, quand nous exécutons un programme en utilisant *valgrind* l'exécution sera notablement beaucoup moins vite, et comme l'outil est en train de vérifier la mémoire pour chaque instruction du code, la consommation de ressources est plusieurs fois plus grand. C'est pour ça que nous avons utilisé cet outil seulement au moment du codage, pour trouver des erreurs que nous pouvions avoir fait, et nous ne l'utilisons pas pour la compilation normal du programme.

Pour notre projet nous utilisons plusieurs fois la mémoire dynamique, car nous utilisons des *malloc* et *calloc*, qui sont des fonctions qui allument un espace de mémoire selon la taille qu'on les demande, mais chaque fois que nous allumons un espace de mémoire, nous devons le libérer après l'utiliser, sinon nous aurions des fuites de mémoire, et plusieurs fois dans notre code nous avions des fuites de mémoire, mais avec l'aide de *valgrind* nous pouvions les fixer.

Par exemple nous utilisons l'instruction suivante:

valgrind - -leak-check=full ./reversi-1

pour vérifier si nous avons des erreurs de mémoire dans l'exécutable *reversi-1*, et au moment de finir l'exécution du programme, si nous n'avons pas des erreurs nous aurons cet message:

HEAP SUMMARY:

in use at exit: 30,620 bytes in 1 blocks

total heap usage: 17 allocs, 16 frees, 35,868 bytes allocated

LEAK SUMMARY:

definitely lost: 0 bytes in 0 blocks

indirectly lost: 0 bytes in 0 blocks

possibly lost: 0 bytes in 0 blocks

still reachable: 30,620 bytes in 1 blocks

suppressed: 0 bytes in 0 blocks

Reachable blocks (those to which a pointer was found) are not shown.

To see them, rerun with: –leak-check=full –show-leak-kinds=all

For counts of detected and suppressed errors, rerun with: -v

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Mais dans le cas que nous avons quelque erreur, *valgrind* va nous indiquer où est-ce qu'il se trouve, dans quel fonction et dans quel ligne, et il va nous montrer une trace de l'erreur. Par exemple:

28 (64 direct, 64 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 3

at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_ memcheck-amd64-linux.so)

by 0x4039F8: moves_ alloc (moves.c:28)

by 0x40387E: board_ get_ moves (board.c:1027)

by 0x4044B8: minimax_ ab_ player (player.c:267)

by 0x403D7A: random_ player (player.c:26)

by 0x400EFE: game (reversi.c:85)

by 0x40142B: main (reversi.c:320)

Ce message veut nous dire que nous avons une petite fuite de mémoire de 28 bytes dans une fonction *moves_ alloc* dans la ligne 28 du *moves.c* et qui a été appelé dès la fonction *main* de *reversi.c* dans la ligne 320, en passant aussi pour *player.c, board.c* et en arrivant à *moves.c.*

Un autre outil que nous avons utilisé était *time*, qui nous permettrons de connaître le temps qui prent le programme en exécuter des instructions. Au moment d'implémenter « l'intelligence artificielle », ou les algorithmes *minimax* pour trouver le meilleur mouvement, nous devons trouver un code qui nous permet ou une implémentation de ces algorithmes pour qu'ils puissent trouver une solution dans le plus petit espace de temps possible. c'est pour ça que nous avons utilisé l'outil *time.* Par exemple nous utilisons l'instruction:

time ./reversi-1

Et après faire quelques mouvements dans le joue, nous aurions une sortie comme ce la:

real 0m0.092s

user 0m0.088s

sys 0m0.000s

Nous avons utilisé aussi *gprof* ou le profilage du code. Avec cet outil nous pouvons connaître les parties du code qui prennent et consomment plus de temps, ainsi nous pouvons améliorer la maniéré dans laquelle le code fonctionne et comme ça le programme pourra s'exécuter plus rapide et efficace.

Pour exécuter *gprof* nous devions changer l'instruction pour compiler le programme, en ajoutant *-pg* à l'instruction. Par exemple dans notre *Makefile* nous devons avoir l'instruction suivante

CFLAGS = -std=c11 -Wall -pg -Wextra -g # -O2

Avec ça, au moment d'exécuter le code, il va créer un fichier qui s'appelle *gmon.out*, avec des statistiques de comment le programme a été exécuté. Nous pouvons le transformer en format .txt pour le lire. Dans cette fichiers nous aurons une analyse exhaustive de la performance du code. Par exemple:

% time	cumulative seconds	self calls	self Ts/call	total name
0.00	0.00	7173	0.00	board_ is_ move_ valid
0.00	0.00	88	0.00	board_ count_ moves
0.00	0.00	46	0.00	moves_ set
0.00	0.00	19	0.00	board_ get_ size
0.00	0.00	12	0.00	board_ get_ player
0.00	0.00	12	0.00	board_ play
0.00	0.00	12	0.00	board_ score
0.00	0.00	12	0.00	playervalid
0.00	0.00	7	0.00	moves_ alloc