Rapport de Programmation : Reversi

Louis Coumau

Novembre 2019

Table des matières

Chapitre 1

Implementation

1.1 Interface generale

L'interface est importante, elle va nous permettre d'interagir avec le jeu. Il faut donc qu'elle soit la plus fiable possible, et qu'elle informe sur le conportement de ce de dernier tout en restant simple à comprendre.

1.1.1 Norme d'erreur

La gestion des erreurs et de son affichage se fera dans le fichier reversi.c. elles se présenteront de la facon suivante : `reversi: t: i` avec

- `t` le type de l'erreur
- `i` l'indication concernant l'erreur

Pour certaines des erreurs comme la lecture d'un fichier sans droit, il est renvoyé le numéro de l'erreur concernée.

1.1.2 Le mode verbose

Le mode verbose est le mode permettant de voir le comportement des joueurs et la disposition du plateau tout au long du jeu. Dès lors que l'option est donnée en argument, une variable booléenne global, initialement définit à faux, devient vrai. Lorsqu'une simple partie est lancé sans l'option, le jeux affiche les informations suivantes :

```
Welcome to this reversi game !
Black player (X) is human and white player (O) is human.
Black player start !

Playing...
```

Ensuite, dans le cas où le joueur est humain, le jeux va présenter le plateau et demander un coup.

À la fin de la partie, le plateau final et le score sont affichés :

```
Game over.

     A B C D E F G H
   1 O O O O O O O O
   2 O O X X O X X X
   3 X O X O X O X O
   4 X X O O O O X O
   5 X X O O O O X O
   6 X O X O O O O O
   7 X X O X X O O O
   8 X O O O O O O O

Score: 'X' = 22, 'O' = 42
Player 'O' win the game.
```

Lorsqu'un joueur aléatoire [1] joue, aucun détail sur son action est affiché. Ainsi, si deux joueur aléatoires joue l'un contre l'autre, les seuls informations affichées serons celles du début de partie et celle de fin de partie. Contrairement au mode verbose qui, à chaque fois qu'un joueur va poser un pion, et ce quelque soit le type de joueur, va préciser quel coup le joueur à joué et dessiner une limitation entre chaque tour.

Par exemple, voici un échantillon d'affichage dans le cas ou l'on fait jouer un joueur aléatoire contre un autre avec l'option verbose :

```
X player's turn.

     A B C D E F G H
   1 X * * * * O _ X
   2 O X O * O O X X
   3 * O X X * O O X
   4 _ * O X X O O X
   5 X X X O O X * X
   6 _ X * O O * O X
   7 _ _ * * X O * *
   8 _ _ _ _ * _ _ _

Score: 'X' = 19, 'O' = 17

X plays D2
========================================

O player's turn.
```

1. On définira et expliquera un peut plus tard ce type de joueur.

```
    A B C D E F G H
  1 X * * * * O * X
  2 O X X X X X X X
  3 _ O X X * O O X
  4 _ * O X X O O X
  5 X X X O O X * X
  6 * X * O O * O X
  7 _ _ _ * X O _ _
  8 _ _ _ _ * * _ _

Score: 'X' = 23, 'O' = 14

O plays C6
========================================
```

1.2 API board.c

1.2.1 Les bitboard

Les bitboards sont des entiers dont les bits de la valeur binaire représentent l'état d'une case d'un tableau. Étant donné que la taille du plateau sur lequel nous jouons varie entre 2×2 et 10×10, la taille d'un bitboard doit donc être d'au moins 100 bit pour être capable de représenter tout le plateau. Or, un type de valeur doit être une puissance de 2. Ainsi, nos bitboards seront de taille 128 bits.

Avant de commencer à utiliser de tels entiers, nous allons définir la convention suivante : la case 0,0 (la case tout en haut à gauche du plateau) sera représenté par le bit de poids faible.

Ci- dessous, un exemple de plateau et sa représentation binaire :

1	0	1	1
0	0	0	0
0	0	0	0
0	0	0	0

plateau de taille 4×4

...	0	0	0	0	1	1	0	1

Avec cette convention, l'initialisation d'un bitboard en fonction de la taille d'un plateau et de sa position est évidente.

1.2.2 Implementation des fonctions shifs

Les fonctions shifts on pour but de déplacer des pions sur le plateau dans une direction donné. Pour cela, nous allons utiliser l'opérateur [2] bit à bit » (resp. «) qui est le décalage des bit vers la droite (resp. gauche).

shift-north

Le déplacement vers le nord est de plus simple à implémenter car, comme vu plus haut, le bit de poids faible est en haut à gauche, il suffit donc de décaler les bits vers la droite (du bit de poids fort au bit de poids faible) autant de fois qu'il y a de case dans une ligne. Par exemple, prenons encore une fois le plateau de taille 4×4 :

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Disposition initiale du plateau. *On utilisera toujours cette disposition initiale.*

En y appliquant le shift-north, soit le décalage des bits vers la droite cela donne :

0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	0

Disposition après un shift-north

shift-south

La fonction shift-south est légèrement plus subtile car, si on décale les bits vers la droite, certains sont toujours présents en dehors du plateau. On va donc devoir utiliser une valeur permettant de retirer tout les bits dépassants du plateau (masque).

2. Pour manipuler les bitboard, il y a plusieurs types d'opération élémentaires : '&' qui est le 'ET' logique, '|' qui est le 'OU' logique *(On parlera d'ajout)*, '∼' qui est le 'NON' logique et le décalage expliqué ci-après.

0	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Disposition après un shift-south

0	0	0	0		0	0	0	0		0	0	0	0
1	0	0	0		0	0	0	0		1	0	0	0
0	1	0	0	& $\sim$	0	0	0	0	=	0	1	0	0
0	0	1	0		0	0	0	0		0	0	1	0
0	0	0	1		1	1	1	1		0	0	0	0

Calcul d'un shift-south

On notera que a& $\sim$ b permet de 'retirer' les bits de b à a

shift-west

De même que pour shift-south, lorsque l'on va décaler les bits d'un cran vers la droite (vers la gauche sur le plateau) on va devoir utiliser un masque.

0	0	0	1		0	0	0	1		0	0	0	0
1	0	0	0	& $\sim$	0	0	0	1	=	1	0	0	0
0	1	0	0		0	0	0	1		0	1	0	0
0	0	1	0		0	0	0	1		0	0	1	0

Calcul d'un shift-west

shift-est

Le fonctionnement de shift-est est quasiment identique à celui de shift-west :

0	1	0	0		1	0	0	0		0	1	0	0
0	0	1	0	& $\sim$	1	0	0	0	=	0	0	1	0
0	0	0	1		1	0	0	0		0	0	0	1
1	0	0	0		1	0	0	0		0	0	0	0

Calcul d'un shift-est

1.2.3 La fonction trace

Lorsque le joueur actuel [3] peut jouer, il pose un de ces pion sur la case correspondante à la position donné par ce joueur. Or, d'après la règle, tout les pions adverses se trouvant entourés à partir du pion précédemment posé se 'retournent' [4], et cela, quelque soit la direction. J'ai donc implémenté une fonction permettant de retrouver la 'trace' de tous les pions à retourner à partir du pion posé. Elle renvoie un bitboard qui n'a plus qu'à être retiré (ajouté) du board représentant les pions de l'adversaire (joueur actuel resp.).

L'algorithme est le suivant :

- pour toutes les directions,
  - tant que le pion rencontré est celui de l'adversaire
  - on ajoute ce pion à la trace
- on retourne la trace des pions à retirer

1.2.4 L'itérateur de coup à jouer

Un peu plus tard, lors de l'implémentation des joueur, nous aurons besoin de savoir si le joueur actuel peut jouer un coup, et si oui, lequel.

C'est pourquoi nous allons ecrire la fonction `board_next_move()` qui va nous renvoyer les coups possibles du joueur actuel, au fur et à mesure de son itération.

Tout d'abord, interessons nous au plateau suivant dans lequel les bits sont des coups possibles :

0	0	1	0
0	1	0	0
0	0	1	0
1	0	0	0

On souhaite récupérer le premier coup à jouer en partant de la case 0,0. Comment déterminer la position de ce premier coup ?

Déjà, on voit bien qu'il est simple de récupérer la coordonné de la colonne en utilisant l'opérateur modulo avec le nombre de zéro précédent. Ensuite, on veut que, suivant une action entre la taille du plateau et ce nombre de zéro, on ait 0 comme coordonné de la ligne.

On s'aperçoit que, pour i quelconque, si $i < size$, la valeur entière de $i/size = 0$, de même, si $size \leq i < 2 \times size$, la valeur entière de $i/size = 1$.

3. On définit le joueur actuel comme le joueur à qui c'est le tour de jouer. On notera que ce dernier peut ne pas pouvoir jouer.

4. On parle de pion qui se retourne lorsque ce dernier change de joueur.

Ainsi, nous pouvons déterminer les coordonnées de ce coup avec le nombre de zéro qui le précède.

Comment calculer le nombre de 0 avant le premier bit ?

Nous allons précéder de la manière suivante.
Reprenons le bitboard précédent :

1	0	1	0	0	0	0	1	0	0	1	0	0

On retire 1 ce qui donne :

1	0	1	0	0	0	0	1	0	0	0	1	1

On applique l'opération suivante :

1	0	1	0	0	0	0	1	0	0	1	0	0

$\wedge$ [5]

1	0	1	0	0	0	0	1	0	0	0	1	1

=

0	0	0	0	0	0	0	0	0	0	1	1	1

À ce stade, il ne nous reste plus qu'a utiliser la fonction `popcount()` pour compter le nombre de bits à 1 dans ce bitboard et retirer 1 au résultat.
Ainsi, nous avons comme formule générale :

```
nbr_tz [6] = bitboard_popcount((possibles_moves-1) ∧ possibles_moves)-1;
```

Il existe la fonction `__builtin_ctz()` du compilateur `gcc` qui fait exactement la meme chose que la formule ci dessus. Cependant, elle n'est pas adapté pour les entiers de 128 bits. De plus, meme si on l'utilise à la place de la formule sur des entiers uniquement de 64 bits, l'execution n'est pas plus rapide.

1.3 Les types de joueurs

Apres avoir codé les fonctions de `board.c`, nous avons à présent une API nous permettant de manipuler des plateaux de jeu. Il nous faut donc implémenter les différents types de joueurs.

5. Opérateur du 'OU EXCLUSIF'
6. number of trailing zeroes

1.3.1 Le joueur aléatoire

Le joueur aléatoire est le plus simple de tous les joueurs. son objectif est simplement, comme son nom l'indique, de jouer un coup de façon aléatoire.

Pour cela, nous allons nous aider des fonctions `board_count_player_moves(board)` qui renvoie le nombre de coup que peux jouer le joueur actuel et `board_next_moves(board)` qui est un littérateur renvoyant succesivement un coup possible du plus proche de la case 0,0 au plus proche de la case size-1,size-1.

L'implementation est la suivante :

- La première étape consiste à récupérer le nombre de coup possible dans `nbr_poss_moves` avec la fonction `board_count_player_moves(board)`.
- On génère un nombre aléatoire `r` que l'on réduit dans l'intervalle [0,`nbr_poss_moves`-1] avec le modulo.
- On avance littérateur des coups possibles `r-1` fois grâce à une boucle `for`
- Et on retourne le `r`-ieme coup possible.

La question que l'on se pose à présent est : comment générer ce nombre aléatoire `r` ?

Nous allons utiliser la fonction `random()`. Cependant, pour ce faire, il est nécessaire de mettre à jour la grène aléatoire.

Nous allons donc écrire une fonction `rand_init()` qui va vérifier grâce à une variable booléenne statique si la fonction a déjà été exécuté. Si oui, elle ne fait rien, sinon, elle met a jour la grène aléatoire avec `srandom(time(NULL) - getpid())` et met à vrai la variable de vérification.

A partir de maintenant, Nous appellerons 'random_ player' le joueur aléatoire.

1.3.2 Le joueur humain

Le joueur humain est plus complexe car il nécessite une interaction.

L'objectif de cette fonction est de renvoyer un coup donné. Elle doit donc vérifier si la suite de caractère donné est bien une position possible dans le plateau et si cette position est un coup possible.

`- get_column()` Intéressons nous tout d'abord à la fonction `get_column(char current_char, size_t size)` qui permet de vérifier si le caractère `current_char` est bien compris entre A et J quelque soit la casse et renvoie sa valeur numérique correspondant à ce caractère. Sinon, elle renvoie size. Effectivement les plateaux que nous utilisons peuvent avoir une taille fixé de 2×2, 4×4, 6×6, 8×8 et 10×10 soit, au maximum, la colonne J.

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

- `get_move()` Ensuite, la fonction utilisé pour demander et récupérer les caractères est `move_t get_move(size_t size)`. Elle exécute les étapes suivantes :

On commence par la partie qui récupère la colonne :

- la variable `char current_char` prend la valeur `getchar()` et on définit `move_t move`.

- si ce caractère est `q` la fonction renvoie le coup `size+1,size+1`.

- sinon on récupère la valeur de `get_column(current_char, size)` dans `move.column` (on notera que, comme vu plus haut, `move.column = size` si le caractère est incorrecte)

Vient ensuite la partie permettant de récupérer le ou les caractères définissant la ligne correspond au coup. Effectivement, le coup donné peut varier entre 1 et 10 et peut donc contenir deux caractères

- on définit un tableau `char char_number[2]` de taille 2

- on récupère les caractères dans le tableau.

- `move.row` prend la valeur du tableau convertie en entier.

- `move.row` prend la valeur `size` si cette dernière est égale à 0 ou si elle est supérieur à `size`

- sinon, on décrémente `move.row` car on souhaite qu'elle soit dans l'intervalle [0,size-1].

Pour résumer, cette fonction renverra les coups `size,*` et/ou `*,size` si l'un des caractères ou l'une des positions sont incorrects et `size+1,size+1` si le caractère rencontré est q.

- `human_player()` Nous pouvons maintenant nous intéresser a la fonction principale.

Pour permettre la gestion de la répétition d'un coup, nous allons utiliser une boucle while et une variable bolléenne `error_case` initialisé à vrai. Cette variable redevient vrai dans les cas ou il est necessaire de redemander le coup, et faux sinon. Ainsi, la boucle s'arretera si aucun de ces cas n'est rencontrés. Cette derniere s'arretera aussi si le caractère entré est `q`. Car, comme vu plus haut, dans ce cas, la fonction `get_move()` renverra `size+1,size+1`. Il suffit donc de faire une condition sur uniquement d'une des deux composante de la position retournée.

Il est important de preciser que lorsque l'on a plus besoins de lire un caractère avec `get_char()`, il est necessaire de supprimer les valeurs encore stockées dans ce dernier. Pour ce faire, la fonction `clean_buffer()` va appeler `get_char()` jusqu'a ce que le caractère retourné par celui ci soit le caractère 'new line'

À partir de maintenant, nous appellerons 'human_player' le joueur utilisant la fonction human_player().

1.3.3 Implementation de l'algorithme Mini-Max

L'un des principaux objectifs de ce projet et de parvenir à créer un type de joueur capable de gagner quelque soit l'adversaire. Pour ce faire, l'une des methodes evidente est de parcourir toutes les combinaisons de coup possibles

afin de determiner, lequel permet de gagner dans tous les cas. Nous verrons plus tard qu'il il est impossible de parcourir toutes les possibilitées jusqu'à la fin du jeu, c'est pourquoi il est necessaire de fixer une profondeur.

L'algorithme minimax permet, en fonction du joueur pour lequel il joue et pour chaque configuration du jeu, de lui maximiser une heuristique et, a l'opposé, de minimiser cette heuristique pour le jouer adverse. Nous commencerons avec une heuristique simple qui ne fait que calculer la difference entre le score du joueur et celui de son adversaire.

Exemple avec un plateau de taille 4×4 : On suppose que le joueur X est minimax en profondeur 1.

	A	B	C	D
1	O	O	O	*
2	X	O	O	*
3	O	X	O	*
4	*	X		

FIGURE 1.1 – Joueur actuel : X ; Scores : X : 3 , O : 7

C'est au joueur X de jouer. L'algorithme minimax va donc maximiser l'heuristique des 4 coups possibles.

	A	B	C	D
1	O	O	O	X
2	X	O	X	
3	O	X	O	
4		X		

Coup D1

FIGURE 1.2 – Joueur actuel : O ; Scores : X : 5 , O : 6
Dans cette configuration, l'heuristique renvoyé est $5 - 6 = -1$

Parmis les 4 coups possible, la différence de score du coup D2 est supérieur à celle de tous les autres. Ainsi, le coup renvoyé par minimax à une profondeur 1 sur cette disposition de plateau est D3.

	A	B	C	D
1	O	O	O	
2	X	X	X	X
3	O	X	X	
4		X		

Coup D2

FIGURE 1.3 – Joueur actuel : O ; Scores : X : 7 , O : 4
Dans cette configuration, l'heuristique renvoyé est $7 - 4 = 3$

	A	B	C	D
1	O	O	O	
2	X	O	O	
3	O	X	X	X
4		X		

Coup D3

FIGURE 1.4 – Joueur actuel : O ; Scores : X : 5 , O : 6
Dans cette configuration, l'heuristique renvoyé est $5 - 6 = -1$

	A	B	C	D
1	O	O	O	
2	X	O	O	
3	X	X	O	
4	X	X		

Coup A4

FIGURE 1.5 – Joueur actuel : O ; Scores : X : 5 , O : 6
Dans cette configuration, l'heuristique renvoyé est $5 - 6 = -1$

`minimax_machine()`

Commençons à présent par implementer cette fonction. D'abord, intéressons nous à la fonction `static int minimax_machine(board_t *board, size_t`

`depth, disc_t player)` qui prend en paramètre variable :

- `board` qui est le plateau sur lequel on joue,
- `depth` qui est la profondeur d'observation,
- `player` qui est le joueur pour lequel on veut maximiser l'heuristique.

On va commencer par récupérer le joueur actuel. Ensuite, on continue si ce dernier n'est pas `EMPTY_DISC` (cas ou la partie est terminée) et si la profondeur n'est pas égale à 0. Si tel est le cas, on retourne l'heuristique.
A ce niveau, 2 cas se présentes

- Premier cas : C'est le cas ou le joueur actuel du plateau est le même que celui pour lequel on souhaite maximiser l'heuristique. On définit une variable `best_score` à `-INFINITY`[7] car on souhaite trouver le maximum parmi plusieurs valeur. Il faut donc que la première valeur de référence soit très petite afin de la remplacer dès la première comparaison. A ce stade, pour tout les coups possible du joueur, on va jouer le coup sur une copie `tmp_board` du même plateau pour ensuite récupérer dans `score` la valeur retourné par `minimax_machine(tmp_board, depth - 1, player)`. Apres avoir executé cette etape, `tmp_board` est inutile, on peut donc le liberer avec `board_free(tmp_board)`. Ensuite, on compare `best_score` avec `score` et si cette derniere est superieur à `best_score`, alors `best_score` prend la valeur de `score`.

- Deuxieme cas : Le deuxième cas, opposé au premier, n'est pas très différent de ce dernier car, comme on sait que le joueur actuel est l'adversaire, on va minimiser son heuristique et donc définir `best_score` à `INFINITY`. De plus, lors de la comparaison, `best_score` prend la valeur de `score` si `score` est inférieur à `best_score`.

On vient donc d'implementer une fonction permettant de renvoyer la meilleur heuristique jusqu'à une certaine profondeur.

`minimax_player`

Voyons maintenant `static move_t minimax_player(board_t *board, size_t depth)` qui, en fonction d'une disposition de jeu et d'une profondeur va nous renvoyer le meilleur coup possible d'apres l'heuristique.

Etant donné qu'il s'agit de la premiere execution a partir du moment ou le joueur souhaite connaitre le meilleur coup, on sait que la disposition du plateau et le joueur actuel n'ont pas changés. Il s'agit donc, dans cette fonction, de trouver la maximum parmi les valeurs de `minimax_machine()` avec comme parametre les plateaux sur lesquels on a joué un des coups possibles, pour chaque coup possible.

L'algorithme est le suivant :

- on récupère le joueur actuel dans `current_player` ;
- on definit `best_score` à `-INFINITY` et on declare `best_move` ;

7. On notera que, l'on fixe `INFINITY` à la borne maximal d'un integer, soir $2^{15} - 1 = 32767$.

- pour chaque coup possible `move` ;
  - si `best_move` n'a pas de valeur, ce dernier prend la valeur de `move` ;
  - on joue `move` dans `tmp_board`, une copie temporaire du plateau ;
  - on stocke la valeur renvoyé par `minimax_machine(tmp_board, depth - 1, current_player)` dans `best_score` ;
  - si `score` est supérieur à `best_score`, ce dernier prend la valeur `score` et `best_move` prend la valeur de `move` ;
  - on libère le plateau temporaire ;
- on renvoie `best_move`.

À partir de maintenant, Nous appellerons 'minimax_player' le joueur simulant l'agorithme Mini-Max.

1.3.4 Implementation de l'algorithme Alpha-Beta

L'algorithme Alpha-Beta est similaire à celui de Mini-Max. Il va parcourir les différentes configurations du plateau afin de renvoyer la meilleur. Seulement, contrairement à Mini-Max, l'algorithme Alpha-Beta ne parcourt que les branches pertinentes.

`alpha_beta_machine()`

Reprenons l'algorithme de `mini_max_machine()`, en y ajoutant deux conditions :

Soit `player`, le joueur pour qui l'algorithme doit renvoyer le coup.
- on récupère le joueur actuel dans `current_player` ;
- si la partie est terminé ou si la profondeur atteinte est 0 ;
  - on renvoie l'heuristique en fonction de `player`
- si `current_player` = `player`
  - pour chaque coup possible `move` ;
  - on joue `move` dans `tmp_board`, une copie temporaire du plateau ;
  - `best_score` = `alpha_beta_machine(tmp_board, depth - 1, alpha, beta, current_player)` ;
  - on libère le plateau temporaire ;
  - si `score` > `alpha`
    - `alpha` = `score` ;
    - si alpha >= beta
      - on stop la boucle
  - on renvoie `alpha`
- si `current_player` ≠ `player`
  - pour chaque coup possible `move` ;
  - on joue `move` dans `tmp_board`, une copie temporaire du plateau ;
  - `best_score` = `alpha_beta_machine(tmp_board, depth - 1, alpha, beta, current_player)` ;
  - on libère le plateau temporaire ;

- si `score` < `alpha`
    - `alpha` = `score` ;
    - si alpha >= beta
        - on stop la boucle
- on renvoie `beta`

`alpha_beta_player()`

- on récupère le joueur actuel dans `current_player` ;
- `best_score` = `-INFINITY`
- on définit `best_move` ;
- pour chaque coup possible `move` ;
    - si `best_move` n'a pas de valeur
        - `best_move` = `move`
    - on joue `move` dans `tmp_board`, une copie temporaire du plateau ;
    - `best_score` = `alpha_beta_machine(tmp_board, depth - 1, -INFINITY, INFINITY, current_player)` ;
    - si `score` > `best_score` ;
        - `best_score` = `score`
        - `best_move` = `move` ;
    - on libère le plateau temporaire ;
- on renvoie `best_move`.

1.3.5 Gestion des joueurs

Au fur et a mesure de l'avancement du projet, nous allons devoir ajouter et tester différents types de joueurs afin de déterminer lequel est le plus efficace. Il nous vient donc le problème suivant : Comment ajouter et utiliser facilement dans `reversi.c` un type de joueur implémenté dans `player.c` ?

Pour résoudre ce problème, nous allons créer une constante de préprocesseur `NBR_PLAY_FUNC` qui définit le nombre de type de joueur. Nous pouvons donc l'utiliser pour définir deux tableaux statiques de cette taille, le tableau `play_func` qui va contenir les pointeurs des fonctions simulant les types de joueurs, et le tableau `name_play_func` qui va contenir le nom de chaque type de joueur correspondant, à la même position de ce dernier.
Par exemple :

si `play_func` contient

joueur1	joueur2

`name_play_func` doit contenir

nom du joueur1	nom du joueur2

Cette méthode simplifie l'utilisation des fonctions `human_player` et `random_player` car comme la première correspond à l'option 0, il suffit de définir la fonction avec laquelle un joueur va joueur avec `play_func[0]`. De même avec `random_player` qui correspond à l'option 1. On peut donc, généraliser en récupérant la fonction à la même position que l'option donné.

Il reste simplement à vérifier que l'option donné à `-b` ou `-w` est bien comprise entre 0 et `NBR_PLAY_FUNC-1`.

1.3.6 Simulations et comparaisons

Fonctions de simulation

Comme vu précédemment, nous pourrons facilement ajouter un type de joueur afin de pouvoir le tester. Cependant, les fonctions contenues dans le tableau doivent toute avoir la même signature. Or, nous savons que les fonctions `human_player()` et `random_player()` ne prennent qu'un seul argument en paramtre contrairement aux fonctions `minimax_player()` et `alpha_beta_player()` qui nécessitent une profondeur et donc un paramètre en plus. Pour contrer ce problème nous allons tout simplement créer, pour chaque fonction de ce genre, une fonction de simulation qui exécutera celles ci avec une profondeur prédéfinit dans `player.h`.

Par exemple :

```
move_t alpha_beta_player(board_t *board, size_t depth)
```

Aura comme fonction de simulation :

```
1 move_t simul_alpha_beta_player(board_t *board) {
2   return alpha_beta_player(board, DEPTH_ALPHABETA);
3 }
```

Comment tester un joueur ?

Maintenant que nous avons différents type de joueur, on souhaite les tester. L'objectif est de lancer un certain nombre de fois une partie et de récupérer le résultat final afin de pouvoir le conserver. Pour cela, nous allons utiliser le langage python, qui permet, grâce a la bibliothèque `subprocess`, de lancer un processus. De plus, avec cette commande, on peut ne pas afficher le résultat d'un exécution et simplement la récupérer dans une variable.

L'algorithme est le suivant :

- Soit `n` l'entrée de la fonction définissant de nombre de tests à faire,
- Soit `nbr_win_x`/`nbr_win_o` initialisé à 0, définissant le nombre de fois que le joueur X/O gagne, respectivement,

- Soit `p` le pourcentage d'avancement, initialisé à 0
- Pour tout $i \in [0, n]$,
  - on lance `./reversi` avec les option que l'on souhaite, et on récupère la chaîne de caractère renvoyé dans `stdout`
  - si `"Player 'X' win the game."`/`"Player 'O' win the game."` existe dans `stdout`, on incrémente `nbr_win_x`/`nbr_win_o`, respectivement,
  - si $i \times 80/n$ est supérieur à `p`, on écrit '#' et le pourcentage est mis à jour avec le calcul précédent. [8]
- a la fin des n exécutions, on écrit :

`n` tests
(`nbr_win_x` $\times$ 100 / `n`) % of success for X
(`nbr_win_o` $\times$ 100 / `n`) % of success for O

On aurait pu appliquer l'exécution de plusieurs parties directement dans un fichier de `test.c` en incluant les différentes API. Or, dans ce cas, on aurait quand même eu, a chaque fin de partie, un affichage du plateau final, ce qui sur plusieurs centaine de parties, pourrait ralentir le test.

Pour évaluer la fiabilité d'un joueur, nous regarderons son taux de succès contre le joueur aléatoire. Effectivement, si on fait jouer un joueur contre un autre sans aléa, la stratégie reste la même et donc la partie est déterministe.

Intervalle de fluctuation

À présent, une nouvelle question se pose : Sur combien de parties allons nous avoir des statistiques fiables ?

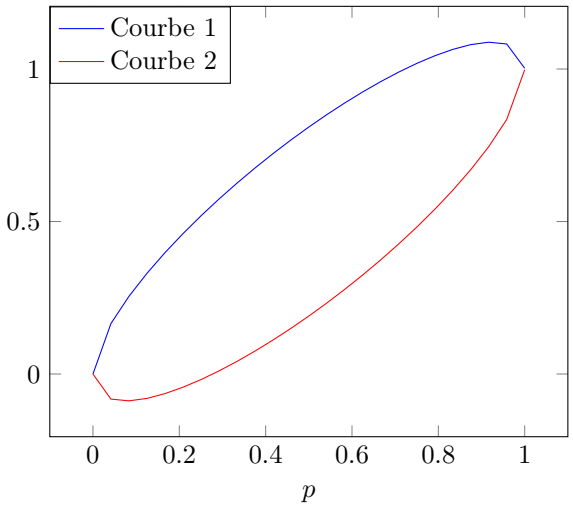
Pour ça, nous allons utiliser la formule de l'intervalle de confiance à un niveau de 95% qui va nous permettre de déterminer la fiabilité de notre valeur.

$$\left[f(p,n) = p - 1.96 \times \sqrt{\frac{p \times (1-p)}{n}}, g(p,n) = p + 1.96 \times \sqrt{\frac{p \times (1-p)}{n}} \right] \quad (1.1)$$

Avec p la valeur obtenue et n le nombre de parties testés.

Regardons quelques courbes :

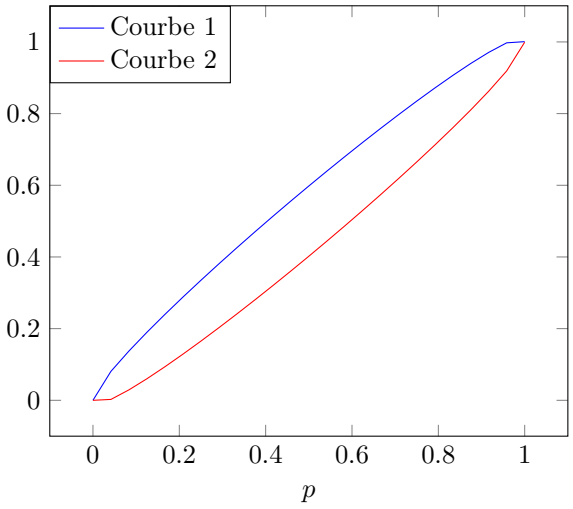
8. Cette formule est utilisé pour calculer un pourcentage. Dans ce cas, à chaque pourcent obtenue, on écrit un caractère. Cependant, le terminal sur lequel on lance cette fonction peut, en général, contenir en largeur 80 caractères, d'ou la formule.



Pour $n = 10$, Courbe 1 $= g(p, n)$, Courbe 2 $= f(p, n)$

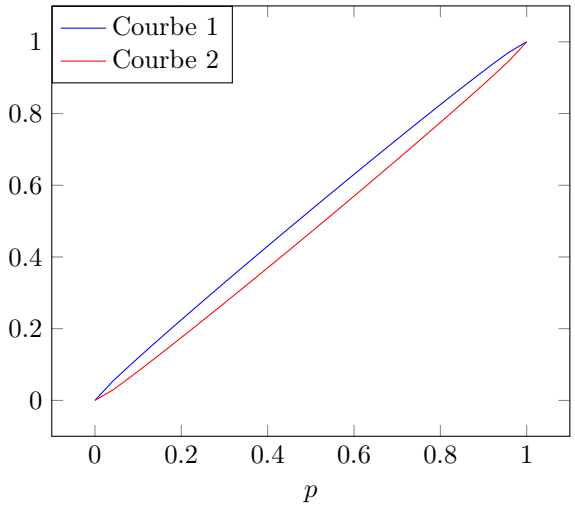
On observe dans ce graphique que les deux courbes d'intervalles sont très espacées lorsque $p = 0.5$ ce qui signifie que la marge d'erreur est importante. Soit, d'après le calcul quand p=0.5 et n=10, p peut être compris en [0.19,0.8] avec une probabilité de 0.95%

Évidemment, faire des tests sur 10 parties n'est pas représentatif car on pourrait très bien avoir un bon joueur qui échoue sur les 5 parties dans les 10 premières et qui n'échoue que très peu après.



Pour $n = 100$, Courbe 1 $= g(p, n)$, Courbe 2 $= f(p, n)$

Voyons maintenant pour $n = 100$: $p \in [0.402, 0.598]$ avec un probabilité de 95%



Pour $n = 1000$, Courbe 1 $= g(p, n)$, Courbe 2 $= f(p, n)$

Sur 1000 parties jouées, la marge d'erreur est plutôt bonne car, même dans le pire des cas ou p=50%, $p \in [0.47, 0.53]$ avec une probabilité de 95% soit $\pm 3\%$ de marge d'erreur.

Le temps

Finalement, estimer le taux de gain d'un joueur sur 1000 parties se rapproche raisonnablement de la réalité. Mais, peut on réellement lancer 1000 parties ?

Pour le joueur aléatoire, une partie se fait instantanément, mais regardons plutôt le temps d'exécution de 100 parties [9] pour Mini-Max et Alpha-Beta en fonction de la profondeur [10].

La figures en annexe contient deux graphiques représentant le temps *(axe des ordonnées)* d'exécution de 100 parties en fonction de la profondeur *(axe des abscisse)*. Afin de comparer facilement, les deux graphiques sont à la même échelle de temps. On s'aperçoit que, le temps moyen d'une partie joué avec Mini-Max contre random_player crois plus rapidement que celui d'une partie joué avec alpha-beta contre random_player quand la profondeur augmente. Cependant il reste en temps exponentiel.

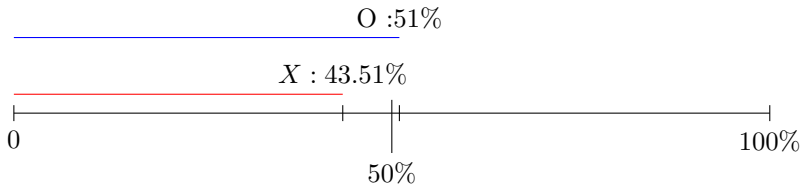
9. Dans ce cas, il ne s'agit que de mesurer le temps d'exécution d'une partie et ce, pour plusieurs profondeurs. 100 est donc raisonnable pour atteindre la profondeur 5 avec l'algorithme Mini-Max

10. Les tests sont fait avec un plateau de taille 8×8

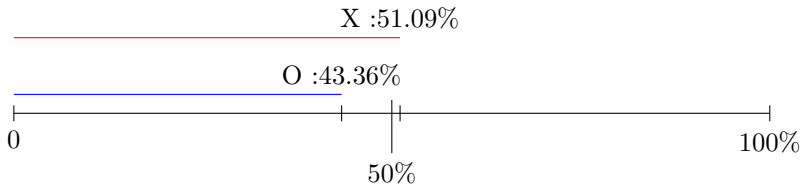
Uniformité de random_player

On voudrait voir à present si random_player est uniforme, c'est a dire si les deux joueurs on la meme chance de gagner.

Faisons le test sur 10000 parties avec X le premier à jouer. *On prendra la couleur rouge pour représenter le joueur X et bleu pour le joueur O.*



Maintenant, faisons le test toujours sur 10000 parties avec le joueur O qui commence.



On s'aperçoit après observation qu'un joueur à plus de chance de gagner si il n'est pas le premier à joueur

Chapitre 2

Ameliorations algorithmiques

2.1 La gestions des bitboards

2.1.1 Les shifts

Les déplacements (shifts) de pions sont la base du fonctionnement du jeu. Ce sont les fonctions les plus appelées pendant une partie car elles permettent non seulement de déterminer les coups possibles mais en plus de trouver quels pions peuvent êtres retournées après qu'un autre soit posé. Il faut donc minimiser les calculs au maximum dans ces fonctions. Pour cela, j'ai utiliser des condition (switch) sur la taille du plateau afin de ne pas à avoir à calculer le masque en fonction de la taille. De plus, comme il y a un masque différent pour chaque taille, je les ai stockés dans des préprocesseur.

2.1.2 Test du popcount

Au même titre que la fonction native de gcc `__builtin_ctz()` vu dans la partie traitant sur `board_next_move()`, il existe la fonction `__builtin_popcount()`. J'ai donc fait quelques test de comparaison entre ces deux fonctions. La première différence notable est que `__builtin_popcount()` ne peut prendre que des entiers jusqu'à 8bit. Effectivement, il existe la même fonction `__builtin_popcountll()` qui peut prendre en entré des entiers de 64 bit mais les entiers avec lesquels nous travaillons sont de 128 bit. On va donc devoir joueur sur la taille d'un board si on utilise cette fonction. C'est une contrainte.

Finalement, on obtient le même résultat qu'avec `__builtin_ctz()`, c'est a dire aucun changement notable au niveau du temps d'execution.

2.2 L'heuristique

Intéressons nous maintenant à l'heuristique. C'est la partie essentiel pour permettre au joueur utilisant cette heuristique d'améliorer son taux de succès. Pour l'instant, elle ne fait que calculer le score en fonction d'un joueur. On rappel que nous utilisons Alpha-Beta, donc la valeur de l'heuristique calculé sera maximiser pour le joueur pour qui l'algorithme joue, et minimiser pour son adversaire. Ainsi, nous allons pouvoir manipuler ce score afin d'améliorer les chances de succès.

2.2.1 Gestion des coins

L'une des stratégie efficace du reversi est de jouer en priorité dans les coins car l'adversaire ne pourra jamais encadrer le pion se trouvant à cette position. A l'opposé, if faut éviter de jouer sur les cases voisines aux coins car cela ouvre un chemin potentiel à l'adversaire pour jouer dans un coin.

- heuristique 1 : Calcul de la différence de score en retirant le score de l'adversaire à celui du joueur.

- heuristique 2 : Le principe est simple : on vérifie si on a déjà joué dans l'un des quatre coins possibles. Si c'est le cas, on ajoute 5 au score [1]. Sinon, on vérifie si le joueur adverse à joué dans l'un des coins, si c'est le cas on retire 5 au score.

Voyons ce que donnent les statistiques avec cette nouvelle heuristique sur 1000 parties : On voit dans la figure 2.1 que l'heuristique 2 est plus efficace [2] que l'heuristique 1.

Une autre question se pose : Quel valeur à ajouter et à soustraire faut il choisir afin de maximiser le taux de succès ? Remarque : le resultat est different suivant la profondeur.

On va faire différents test à profondeur 3 a une profondeur de 3 :

- sans modifications 86.4% de succes sur 1000 tests
- en ajoutant 10 au score si un move dans un coin est possible 51.4% de succes sur 1000 tests
- alors que si l'on retire 10 au score si un move dans un coin est possible 95.3% de succes sur 1000 tests

- heuristique 3 : Dans notre nouvelle heuristique, nous avons mis en oeuvre la statégie de priorité des coins. Maintenant ajoutons à ça la stratégie qui consiste à eviter de joueur dans les cases voisines aux coins. Le principe est le suivant :

1. On prend 5 de façon arbitraire.
2. On parlera de meilleur efficacité d'une heuristique A à une heuritique B lorsqu'un joueur qui l'utilise à un meilleur pourcentage de réussite qu'avec la B

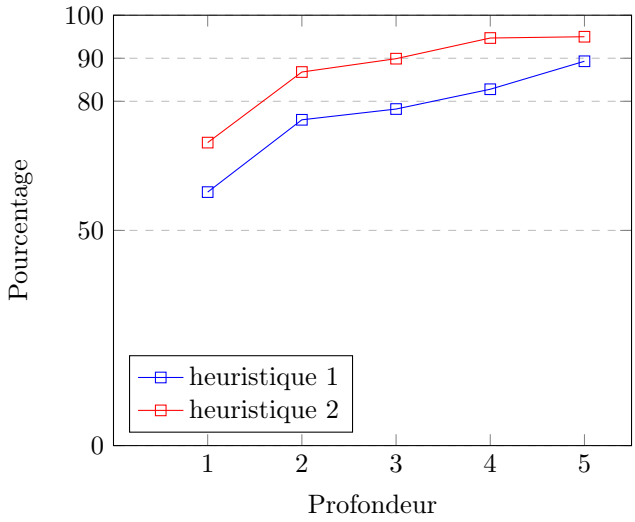


FIGURE 2.1 – Données et graphe de comparaison entre heuristique 1 et 2

Si on a déjà joué dans l'une de ces cases, on retire 20 au score.
Nous avons donc comme résultats :

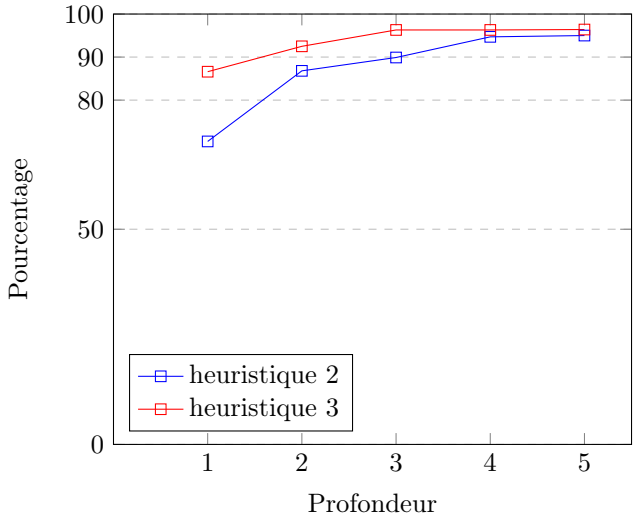


FIGURE 2.2 – Données et graphe de comparaison entre heuristique 2 et 3

2.2.2 Gestion de la parité

On a vu plus haut, que lorsque l'on a comparé deux test qui faisait jouer deux random_player, on s'est aperçu que le joueur jouant en deuxième avait plus de chance de gagner. En réalité, c'est tout simplement parce qu'il a plus de chance de terminer la partie. Effectivement, si les joueur joue à tour de rôle, le deuxième à avoir posé son premier pion terminera la partie et aura un avantage sur le plateau. C'est la parité. Ainsi, si nous somme le premier à devoir jouer, il faudra faire en sorte que l'adversaire ne puisse pas jouer pour passer son tour au moins une fois, et donc avoir l'avantage.

L'algorithme est le suivant :

Lorsque l'on parcourt l'arbre les possibilités de jeux, si l'on à atteint la fin du jeux, on vérifie si nous somme le dernier joueur à avoir posé un pion, dans ce cas favorable, on va augmenter le score de l'heuristique. Sinon, cela signifie c'est l'adversaire qui à posé le dernier pion, il faut donc baisser le score.

Chapitre 3

Mesures prises tout au long du projet

Afin de mener à bien ce projet, il a fallut utiliser quelques outils.

3.1 Clang format

Le clang format est maintenant, à mon sens, un outils nécessaire à la rédaction de code. Effectivement il permet de gérer le 'coding style' c'est a dire la norme à utiliser afin d'avoir un code lisible, cohérent et surtout, de respecter les mêmes règles de codage. Il gère aussi le dépassement des 80 caractères, ce qui permet de ne plus s'en soucier.

3.2 Structuration

Afin de structurer correctement mon code, j'ai utilisé quelques norme. L'une d'entre elles est utilisé dans `board.h`, elle consiste à trier, par type de retour, les fonctions non `static` de `board.c`. Une autre consite à séparer chaque type de joueur dans player.c par un bloc de commentaire afin de bien les différencier.

Par exemple :

```
1 /*****************************************************************/
2 /************************ MINIMAX PLAYER PART ************************/
3 /*****************************************************************/
4
5 static int minimax_machine(board_t *board, size_t depth, disc_t player) {...}
6 static move_t minimax_player(board_t *board, size_t depth) {...}
7 move_t simul_minimax_player(board_t *board) {...}
```

```
8
9
10 /*****************************************************************************/
11 /************************ RANDOM PLAYER PART *********************************/
12 /*****************************************************************************/
13
14 static void rand_init(void) {...}
15 move_t random_player(board_t *board) {...}
```

3.3 Tests

Comme on l'a vu dans la partie traitant sur les statistiques des différentes heuristiques, j'ai utilisé le langages python pour certains test. Le premier test était celui du parser de fichier (`file_parser()`) qui à été, à mon sens, la fonction la plus compliqué du projet. Elle permet au jeu d'être capable de renvoyer un coup quelque soit la disposition d'un plateau passé en argument et donc de pouvoir jouer contre un autre adversaire (le mode contest). Pour nous aider à vérifier les erreurs de cette fonction, nous avions des fichiers contenant différents types de plateau. Comme le nom de ces fichier permettait de savoir si le plateau était correct ou nom, j'ai utilisé un algorithme python permettant de tester chaque fichier et de comparer la sortie (error/success) avec le mot clés contenue dans le nom (fail/pass)

3.4 Le temps

Pour optimiser au maximum les fonctions, dès lors qu'une autre solution d'implémentation me venait à l'esprit, j'utilisais la commande `time` en exécutant plusieurs fois la fonction testé afin de voir si une implémentation était plus rapide qu'une autre.

3.5 Utilisation de gdb

`gdb` m'a été très utile lors de la manipulation des bitboards. Par exemple, nous avons vu plus haut qu'il était nécessaire d'avoir des masque afin de pourvoir faire des déplacements (shift). Or, ces masque peuvent aller jusqu'a 128bits, et il n'est pas possible d'afficher de tels valeurs avec un `print`. `gdb` m'a donc permis de pouvoir consulter ces valeurs et en plus de ça, de pouvoir consulter leur valeur binaire.

3.6 Valgrind

Valgrind m'a aussi été utile pour gérer les fuites mémoires provoqué par la libération du board temporaire après la comparaison de alpha/beta avec le

score dans la fonction `alpha_beta_machine()`. Effectivement, la libération ne se faisait jamais lorsque alpha était supérieur à bêta car, dans ce cas, on sortait de la boucle et donc la libération n'était pas atteinte.

utilisation de valgrind pour detecter les fuites dans alpha-beta à cause de la liberation du board apres avoir retourné alpha/beta

3.7 Utilisation gnuplot

Gnuplot est un outils permettant d'interpréter des données textuel sous forme graphique. Il m'a été très utile pour comparer les types de joueurs et les types d'heuristiques afin de déterminer lequel était le plus efficace.

Annexe A

Comparaison de Mini-Max et Alpha-Beta

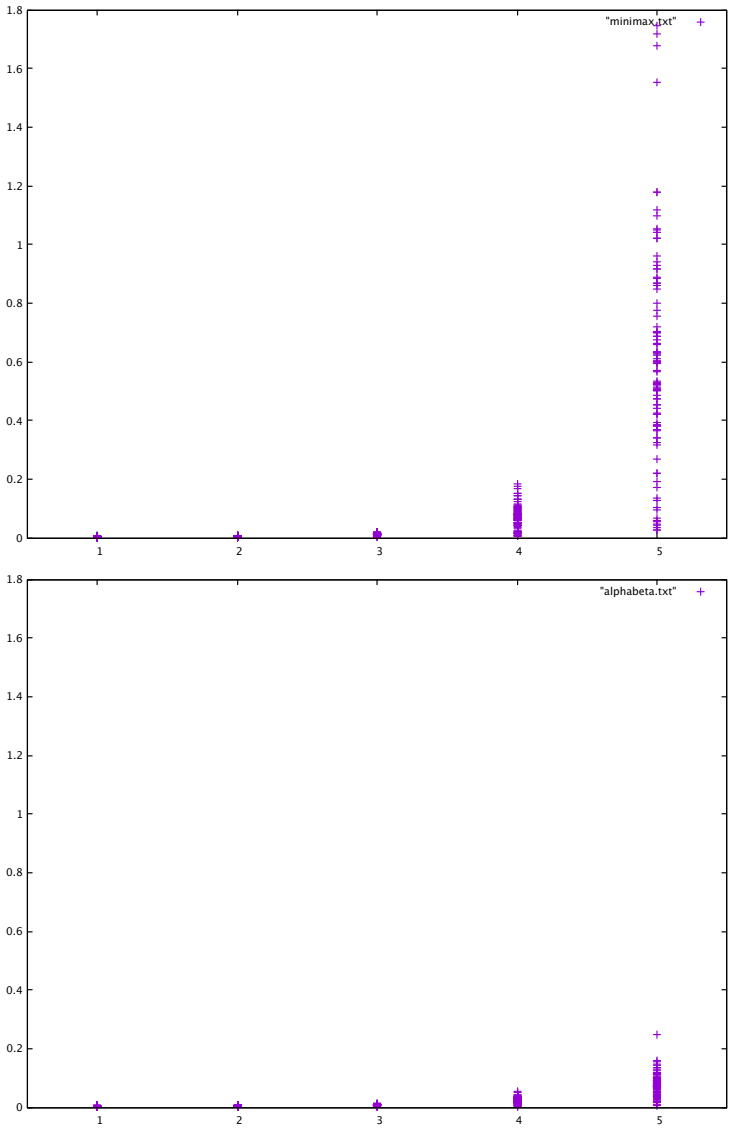


FIGURE A.1 – Mini-Max & Alpha-Beta