Quoridor

Filière informatique 1ère année
Projet informatique S6

Groupe 12828

Auteurs :
Arnaud GRAPPE
Aymane LAMHAMDI
Yves LEBARBIER
Dylan MACHADO

Encadrant :
M. Mihail POPOV

Directeur de projet :
M. David RENAULT

Contents

I Présentation du sujet **3**
I.1 Objectif du projet 3
I.2 Présentation des règles fondamentales 4

II Cadre de travail et organisation de l'équipe **5**
II.1 Cadre de travail 5
II.2 Organisation 5
II.3 Organisation du dépôt 5

III Structure du projet **6**
III.1 Organisation des fichiers 6
III.2 Compilation 7

IV Implémentation des éléments essentiels au projet **9**
IV.1 Graphe représentant le plateau de jeu 9
IV.2 Le joueur 11
IV.3 Structure de file 11
IV.4 Respect des règles du jeu 12

V Les stratégies développées **13**
V.1 Un premier client : Rush 13
V.2 Un client plus élaboré : Greedy 14
V.3 Une implémentation d'ouverture : Sidewall Opening 15

VI Retour d'expérience **16**

List of Figures

1 Exemple de partie de Quoridor 3

2 Structure du dépôt 6

3 Graphe carré 9

4 Graphe haché 9

5 Graphe torique 10

6 Graphe serpent 10

7 Score < 1, initial 16

8 Score < 1, coup joué 16

9 Score > 1 16

10 Présentation des deux comportements possibles du client *Greedy*, en rouge sur les deux figures 16

11 Situation initiale 17

12 Deuxième tour 17

13 Objectif de l'ouverture 17

14 Description de l'ouverture *Side-Wall* 17

I Présentation du sujet

I.1 Objectif du projet

A ce jour, il existe de nombreux jeux de société ou de stratégie qui ont été implémentés à titre lucratif, de loisir ou d'étude. En effet on retrouve chez certains informaticiens une motivation de les programmer pour améliorer les stratégies comme par exemple, rivaliser avec les meilleurs joueurs ou développer des ouvertures aux échecs.

Dans notre cas, nous nous intéressons dans le cadre de ce projet au jeu de société **Quoridor**. Le principe est simple : sur un plateau de jeu, un ensemble de joueurs (au minimum 2 et au maximum 4) s'affrontent et ont chacun pour objectif d'être le premier à amener leur pion sur le bord opposé du plateau, supposé carré. À chaque tour, ils peuvent choisir soit de déplacer leur pion sur une case voisine, soit de poser un mur dans une rigole afin de bloquer les déplacements entre deux cases. La figure 1 apporte une représentation des éléments du jeu de société tels que le plateau, les murs et les pions.

Ce projet a donc pour objectif de programmer le **Quoridor** sous la forme d'une interface client-serveur. Précisons enfin que seule la machine est capable de jouer suivant les stratégies développées.

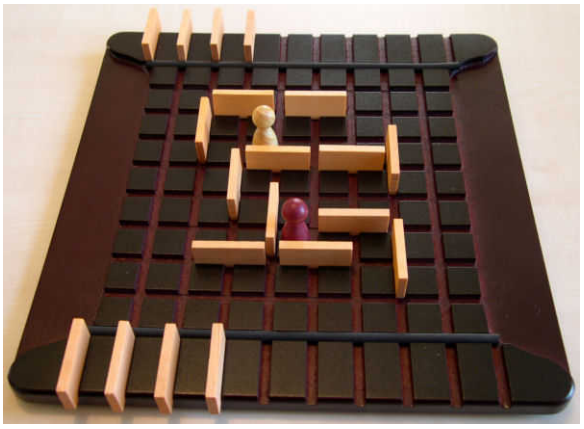

Figure 1: Exemple de partie de Quoridor

I.2 Présentation des règles fondamentales

Le jeu standard se joue sur un plateau carré de taille 9x9 composé de tuiles. Le nombre de murs donnés à chaque joueur en début de partie est le nombre d'arêtes du graphe divisé par 15, dont on prend la partie entière supérieure.

Coups possibles:

Initialement, les joueurs disposent d'un pion et d'un ensemble fini de murs. Le premier tour consiste à placer le pion de chaque joueur sur une des cases qui lui appartient dans le graphe, dans leur ligne de départ. Ensuite, à chaque tour de jeu, les joueurs peuvent choisir soit de déplacer leur pion, soit de placer un mur s'il leur en reste un.

Un joueur qui choisit de se déplacer n'a le droit de le faire que sur une position adjacente à la position de son pion. Si jamais la position visée est occupée par un pion adverse, il peut sauter par-dessus ce pion en se déplaçant de deux cases dans la même direction. Si ce dernier déplacement est impossible, le second déplacement peut se faire sur n'importe quelle arête voisine du pion adverse, sans revenir à sa position de départ.

Un joueur qui choisit de placer un mur ne peut le faire que s'il en dispose encore. Son placement est invalide s'il empêche l'un des joueurs de rejoindre la ligne d'arrivée, soit l'une des cases d'arrivée valide. Les murs ne peuvent pas se recouvrir.

II Cadre de travail et organisation de l'équipe

II.1 Cadre de travail

La réalisation de ce projet a débuté le vendredi 12 mars et elle s'est achevée le vendredi 21 mai. Contrairement au projet *Tilings* du premier semestre, nous avons pu nous réunir chaque semaine dans une salle informatique de l'ENSEIRB. Cette organisation nous a permis d'interagir plus facilement entre nous comme avec nos encadrants.

II.2 Organisation

Au début de chaque séance, nous passions en revue les réalisations de la semaine dernière. Cela nous permettait de définir les objectifs pour la séance, et de nous accorder sur une répartition du travail à effectuer. Nous avions d'abord commencé par réfléchir à l'organisation des fichiers ainsi qu'à la structure du Makefile. En parallèle, nous avions aussi débuté l'implémentation de la généralisation des différents types de graphe. S'en est suivi de l'ajout de la fonctionnalité de chargement dynamique de bibliothèques puis nous nous étions lancés sur l'écriture des éléments essentiels au projet tels que les structures et les fonctions nécessaires au jeu. Enfin, nous nous étions concentrés sur les stratégies à mettre en oeuvre.

II.3 Organisation du dépôt

Pour les besoins du projet, la figure 2 montre l'organisation des fichiers dans le dépôt.
Le fichier Makefile contient les règles suivantes:

- **build**: compile l'ensemble du code.
- **test**: compile les tests, sans les exécuter.
- **install**: copie les exécutables server, alltests et un des bibliothèques dynamiques à l'intérieur du répertoire `install`
- **clean**: supprime les fichiers compilés et installés.

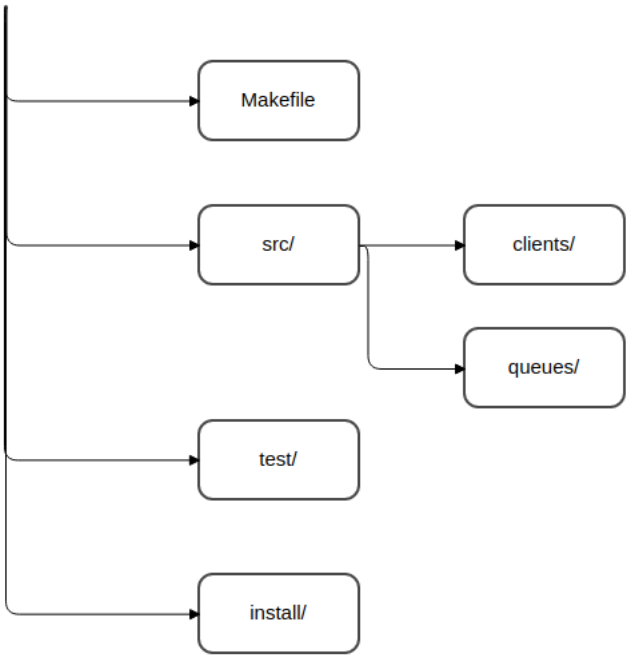

Figure 2: Structure du dépôt

III Structure du projet

Pour ce projet, la structuration des fichiers et du code est particulièrement importante, notamment pour pouvoir gérer l'interface client-serveur.

III.1 Organisation des fichiers

Ce projet, bien qu'exécutant tout le code sur une unique machine, a pour but de représenter une véritable interface client-serveur, avec un serveur central n'envoyant que les informations nécessaires aux différents clients. Ainsi le code source est séparé en 3 parties :

- une base commune nécessaire au serveur et aux clients, composé principalement de l'implémentation et de la gestion des matrices représentant le plateau de jeu,
- un fichier gérant l'exécution de la partie qui modélise donc le serveur;

- différents fichiers contenant le code des clients, ensuite compilés en librairies dynamiques importées par le serveur. Pour chaque client, il y a plusieurs fichiers où sont réparties les fonctions selon leur utilité, notamment si elles gèrent le joueur en lui-même ou si elles implémentent une stratégie.

Concernant les clients, il existe 2 types de fichiers : tout d'abord plusieurs fichiers implémentant les stratégies nécessaires comme par exemple le calcul du plus court chemin pour le client *Rush*, et ensuite les fichiers gérant directement le client, c'est-à-dire qu'ils implémentent les fonctions nécessaires au fonctionnement de la partie, comme l'initialisation du plateau local ou le choix d'un coup à jouer.

Enfin, il existe un certain nombre de fichiers de test correspondant chacun à un fichier du code source et testant les différentes fonctions de celui-ci.

III.2 Compilation

La compilation de ce projet est gérée par un Makefile contenant différentes règles permettant de compiler les fichiers, lancer une partie et tester le code.

Tout d'abord, tous les fichiers sont compilés en fichiers objets avec la commande `make build`. Ensuite, pour chaque client, ses fichiers et la base commune sont liés dans une librairie dynamique avec la commande `make install`. De plus, cette commande crée l'exécutable du serveur à partir du fichier objet correspondant et de la base commune. Finalement, cet exécutable prend en paramètre deux librairies dynamiques et à l'exécution, il fait jouer une partie aux deux clients correspondants et affiche le vainqueur de cette partie. De plus, cet exécutable prend des options : `-i` pour afficher l'état du plateau à chaque tour de jeu, `-m` qui, suivie d'un nombre, fixe la largeur du plateau à cette valeur et enfin `-t` qui permet, suivie de c, t, h ou s, de choisir la forme du plateau : carrée, torique, hachée, ou en forme de serpent.

Enfin, il existe une règle `make test` qui permet de compiler tous les fichiers de test relatifs au projet en un exécutable `alltests` contenu dans le dossier `install`.

Vérification des arguments d'entrée

Le programme exécutera une partie du jeu en prenant en arguments deux librairies dynamiques représentant les deux clients et des options décrites

dans la partie compilation.

```
1 void parse_opts(int argc, char* argv[], size_t *width, char *
    shape, int* interface);
2
3 void check_opts(size_t width, char shape);
4
5 void check_clients(int argc, char* argv[]);
6
7 void exit_message(char* message);
```

La fonction `parse_opts` analyse les paramètres optionnels `-m` et `-t`.

La fonction `check_opts` vérifie si les arguments passés en paramètre sont valides et s'il sont écrits suivant le bon format.

La fonction `check_clients` vérifie si les deux derniers arguments sont bien des bibliothèques dynamiques.

La fonction `exit_message` affiche le message passé en paramètre suivi d'un `EXIT_FAILURE`.

IV Implémentation des éléments essentiels au projet

Le projet repose essentiellement sur la structure de graphe utilisée mais aussi sur celle des éléments essentiels au déroulement d'une partie tels que le joueur. De plus, pour qu'une partie se joue suivant les règles du jeu, il est nécessaire d'écrire des fonctions vérifiant si elles sont respectées.

IV.1 Graphe représentant le plateau de jeu

La base du jeu **Quoridor** est le plateau de jeu. Il faut donc s'occuper de modéliser celui-ci avant d'implémenter le jeu. Pour ce faire, nous avons représenté le plateau comme un graphe dont les sommets sont les cases du plateau et où pour toutes cases adjacentes, une arrête relie les deux sommets correspondants. Ce graphe est alors implémenté à l'aide d'une matrice creuse de type `gsl_spmatrix_uint` de la bibliothèque GSL de taille $n \times n$ où n est le nombre de sommets du graphe, donc le nombre de cases du plateau. Cette matrice représente les arrêtes du graphe : si M est la matrice et i et j des numéros de sommet valides, alors la valeur de $M[i][j]$ donne l'information sur l'arrête allant du sommet i au sommet j. Si elle vaut 0, alors il n'y a pas d'arête, sinon il y a une arête et la direction de celle-ci est donnée par la valeur.

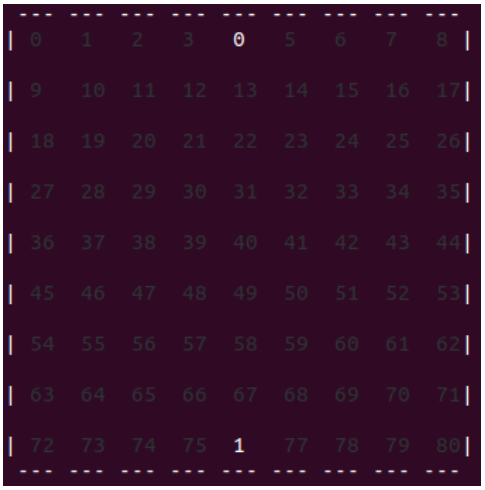

Figure 3: Graphe carré

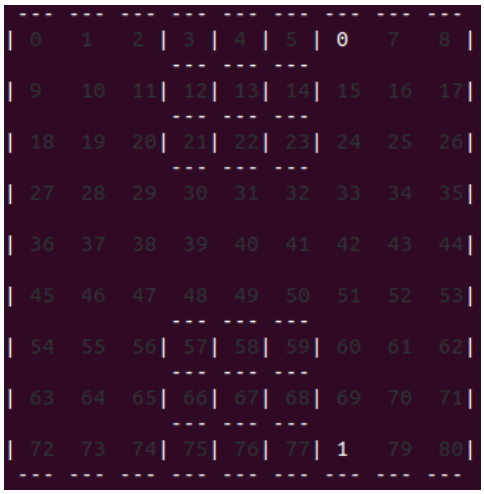

Figure 4: Graphe haché

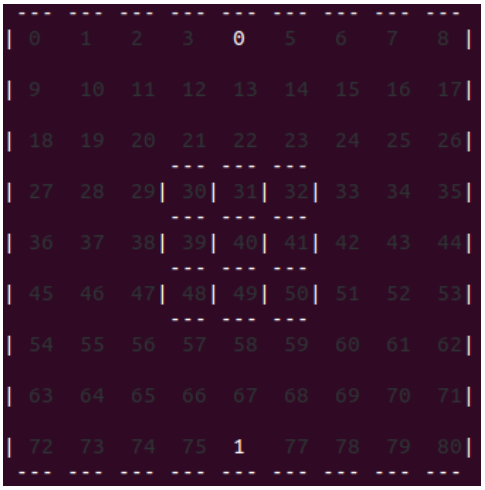

Figure 5: Graphe torique

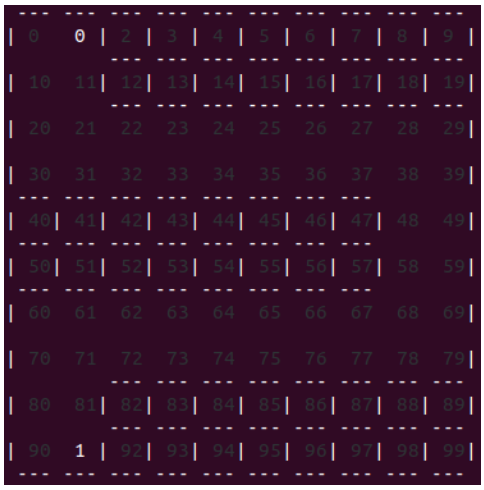

Figure 6: Graphe serpent

Ainsi, afin de regrouper ces informations sur le plateau, nous avons créé une structure `graph_t` qui contient le nombre de cases, la matrice contenant les informations sur les arêtes et une autre matrice recensant les positions appartenant à chacun des joueurs :

```
1 struct graph_t {
2     size_t num_vertices;
3     gsl_spmatrix_uint* t;
4     gsl_spmatrix_uint* o;
5 };
```

Après avoir choisi la représentation du plateau, il faut maintenant implémenter les fonctions liées à celui-ci, qui sont listées ci-dessous :

```
1 struct graph_t* graph__initialize(size_t m);
2
3 size_t vertex_neighbor(struct graph_t* g, size_t v, size_t dir);
4
5 int edge_exists(struct graph_t* g, size_t v1, size_t v2);
6
7 void place_wall(struct graph_t* g, struct edge_t wall[2]);
```

La fonction `graph__initialize` alloue la mémoire nécessaire à la création du plateau puis remplit la matrice par des arêtes afin d'obtenir un graphe carré. Ensuite, si une option de forme autre que carrée a été passée en paramètre, alors les arêtes des sommets ne faisant plus partie du graphe sont

supprimées. Ainsi cette fonction a une complexité en temps en $O(m)$ où m est le nombre de sommets du graphe carré.

La fonction `vertex_neighbor` permet, étant donnés un sommet et une direction, de donner en temps constant le voisin de ce sommet dans le direction donnée.

La fonction `edge_exists` permet, étant donnés 2 sommets, de savoir en temps constant s'il existe une arête entre ces 2 sommets. Cette fonction permet notamment de savoir si un déplacement entre deux cases est possible, elle est donc souvent appelée et il est important qu'elle s'exécute vite.

La fonction `place_wall` supprime dans le graphe des arêtes correspondant au mur donné en paramètre et placé. Cette suppression est effectuée en temps constant puisqu'elle consiste simplement à modifier la valeur des cases correspondantes dans la matrice.

IV.2 Le joueur

L'ensemble des fonctions communes entre les clients ont été implémentées sur le fichier `src/clients/clients_commons.c`. Nous définissons la structure *player* de la manière suivante:

```
1 struct player {
2   enum color_t id;
3   enum color_t opnt_id;
4   struct edge_t* walls;
5 };
```

L'*id* représente l'indice du joueur, *opnt_id* celui de son adversaire et *walls* est un pointeur vers un tableau de mur qui nous servira de stockage.

IV.3 Structure de file

Une structure de file (FIFO) sur les variables de type `size_t` est implémentée dans le fichier `src/queues/queue_uint.c`.

```
1 struct queue {
2     struct q_elem* front;
```

```
3       struct q_elem* back;
4       size_t length;
5   };
```

Celle-ci est accompagnée des fonctions usuelles sur les files: *enqueue*, *dequeue*, *peek*. Notons que la fonction `queue_squeeze` permet d'insérer un élément en tête de la file afin de s'en servir comme une pile. Cette structure et les fonctionnalités qui lui sont associées constituent une pierre angulaire de ce projet. Elles permettent de réaliser des parcours en largeur sur le graphe de jeu. Grâce à ceux-ci, il est possible de vérifier qu'un mur n'empêche pas un joueur de rejoindre l'arrivée, et de calculer des chemins de plus courte distance.

IV.4 Respect des règles du jeu

L'ensemble des règles du jeu sont vérifiées par les fonctions implémentées sur le fichier `src/rules.c`:

```
1  int move_is_licit(struct game_state* game, enum color_t p,
       size_t dest);
2
3  int path_exists_bfs(struct game_state* game, enum color_t p);
4
5  int wall_is_licit(struct game_state* game, struct edge_t e[2]);
6
7  int play_is_licit(struct game_state* game, struct move_t mv);
```

La fonction `move_is_licit` vérifie si le déplacement est valide.

La fonction `path_exists_bfs` vérifie si le joueur passé en paramètre peut atteindre la zone d'arrivée, soit la ligne de départ du joueur adverse. Ceci est valide s'il existe un chemin obtenu par un parcours en largeur du graphe du jeu.

La fonction `wall_is_licit` vérifie si le placement d'un mur est possible, ce dernier est valide si le mur n'empêche pas les joueurs d'atteindre leur zone d'arrivée.

La fonction `play_is_licit` généralise les deux fonctions définies précédemment, il permet de valider une action qui consiste soit au déplacement du pion joueur, soit au placement d'un mur.

V Les stratégies développées

A ce stade du projet, toutes les fonctions nécessaires au bon déroulement d'une partie ont été implémentées et il est donc possible de s'intéresser aux stratégies.

V.1 Un premier client : Rush

Nous présentons dans cette section le premier client développé qui nous a ainsi permis de lancer nos premiers tests de parties. L'idée de ce client est simple et consiste à toujours avancer afin de diminuer la distance avec la zone adverse d'où le nom de *Rush*. Ce client ne pose donc jamais de mur, ce qui n'est clairement pas la meilleure des stratégies. Elle repose essentiellement sur la recherche du plus court chemin qui peut ne pas être unique. Pour cela, on commence par implémenter une fonction `shortest_path` qui effectue un parcours en largeur à partir de la position du joueur et qui renvoie une file contenant la suite des sommets d'un des plus courts chemins ou bien une file vide dans le cas où le joueur se situe déjà dans la ligne d'arrivée. Un pseudo-algorithme de la fonction `shortest_path` est donné en (V.1). Il est important de préciser quelques points :

- Le chemin que renvoie la fonction est construit de telle sorte à ce que le joueur se déplacera toujours en priorité vers le haut si le coup est autorisé.
- En pratique, nous utilisons une unique structure `queue` pour implémenter la file. La fonction `queue_squeeze` permet alors d'insérer les éléments en tête de file et ainsi considérer la file comme une pile.

Enfin, la stratégie est implémentée dans le fichier `strat_rush.c` et plus précisément dans la fonction `rush_move` qui utilise donc `shortest_path`. Cette fonction renvoie le coup qui permettra au joueur de se rapprocher le plus possible de la zone d'arrivée.

j: joueur, g : graphe une file

$v \longleftarrow sommetJoueur()$
$distance \longleftarrow creationTableau(g.nombreSommets)$
$q \longleftarrow fileVide()$

distance[v] = 0

estDansZoneAdverse(j)
Retourner q

$q \longleftarrow ParcoursLargeur(q, g, j, v, distance)$

$chemin \longleftarrow pileVide()$
$empiler(chemin, v)$

distance[v] > 1
voisin w de v
distance[w] < distance[v]
suivant $\longleftarrow wempiler(suivant, chemin)$
$v \longleftarrow suivant$

Retourner chemin
Algorithme qui renvoie un plus court chemin entre la position du joueur et la zone d'arrivée

V.2 Un client plus élaboré : Greedy

Le client présenté précédemment implémente les bases d'une stratégie viable, c'est-à-dire rejoindre l'autre côté du plateau. Cependant, il est incapable de placer des murs, qui font tout l'intérêt du jeu. Dans cette section, nous présentons le client *Greedy* qui reprend la stratégie *Rush* pour se déplacer et place des murs pour ralentir la progression de son adversaire.

Le client en lui-même est implémenté dans le fichier `cl_greedy.c`, lequel utilise les fonctions de `strat_greedy.c` et `shortest_path.c`. Cette stratégie consiste à évaluer l'avantage d'un joueur en calculant un score défini suivant la formule (1).

$$score = |chemin(joueur\_adverse)| - |chemin(joueur)| \qquad (1)$$

Précisons que $|chemin(joueur)|$ représente la distance du plus court chemin qu'il reste au joueur à parcourir. Cette valeur est calculée par la fonction `greedy_score` qui prend en argument l'état de la partie et l'identifiant d'un joueur.

En fonction de la valeur de *score*, le client adopte deux comportements différents:

- si $score > 1$, le client considère qu'il a l'avantage de la distance et joue un coup selon la stratégie *Rush*. Un exemple d'une telle situation est présenté à la figure 9
- si $score < 1$, le client cherche à placer un mur qui va augmenter le plus possible la valeur de *score* (cf figures 7 et 8). S'il n'existe pas un tel mur, le client se déplace selon la stratégie *Rush*.

La recherche du meilleur mur consiste simplement à trouver le meilleur score pour chaque mur possible sur le plateau. Notons que cette opération présente une complexité importante: pour un plateau carré de largeur m et de n cases, il existe $2(m-1)^2$ (un peu moins de $2n$) murs différents. Le placement de chaque mur nécessite de vérifier que celui-ci est bien licite, c'est-à-dire qu'il n'isole aucun des deux joueurs. Enfin, pour chaque joueur, il faut calculer la distance minimale qui le sépare de sa zone d'arrivée. Ces calculs correspondent à quatre parcours en largeur du graphe pour chaque mur, soit une complexité totale en $O(n^2)$.

V.3 Une implémentation d'ouverture : Sidewall Opening

Une des stratégies les plus populaires consiste à suivre un plan précis durant les premiers tours, c'est ce que l'on appelle une ouverture. Ainsi, nous avons implémenté un client qui réalise une ouverture appelée *sidewall opening*.

Celle-ci consiste à poser un mur vertical à côté de l'adversaire après son premier déplacement afin d'ensuite le bloquer dans une boîte, ce qui permet de prendre l'avantage sur l'adversaire. Une fois les premiers tours passés et l'ouverture complétée, ce client va reprendre notre stratégie la plus efficace, c'est-à-dire celle du client *Greedy*. Pour réaliser ceci, il suffit de prévoir des coups précis en fonction du numéro du tour, information auquel le joueur a accès dans sa copie de l'état du jeu. Ainsi, tant que les coups sont valides, les premiers tours seront toujours les mêmes.

Cependant, cette implémentation est peu viable : le client fait toujours les mêmes coups en début de partie et ne s'adapte pas aux coups de l'adversaire. Ainsi, il peut très bien continuer à créer une boîte sur la droite du plateau alors que l'adversaire part sur la gauche.

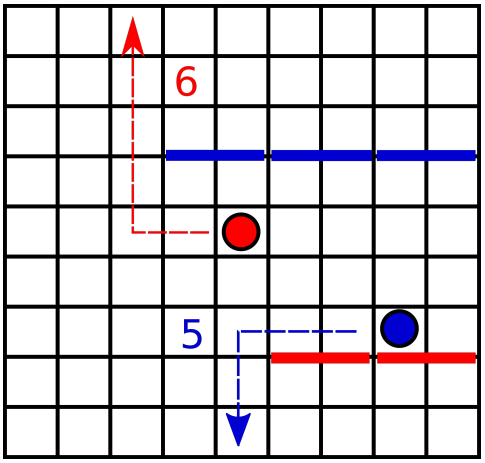

Figure 7: Score < 1, initial

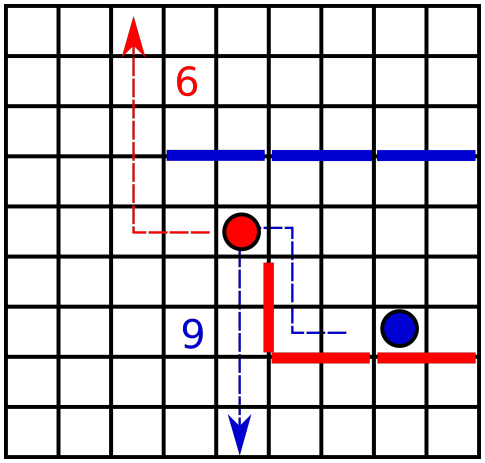

Figure 8: Score < 1, coup joué

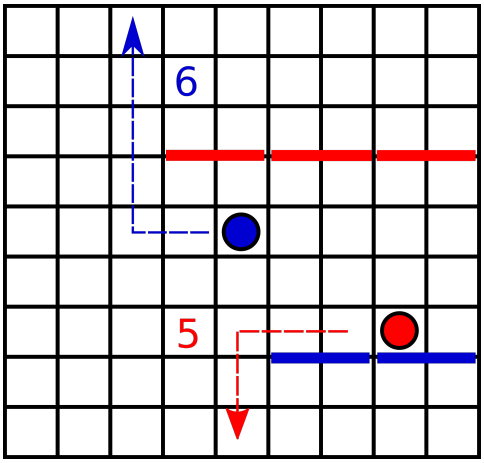

Figure 9: Score > 1

Figure 10: Présentation des deux comportements possibles du client *Greedy*, en rouge sur les deux figures

VI Retour d'expérience

Dans le cadre des projets de programmation, nous avons donc réalisé un programme du jeu *Quoridor*. Ce projet nous a permis de consolider les notions acquises en programmation impérative au cours du premier semestre, et de mettre en pratique celles vues lors du deuxième : la création et l'utilisation de bibliothèques dynamiques en C, l'écriture de tests unitaires, la recherche de fuites mémoire avec Valgrind, etc. Il nous a aussi offert l'occasion de nous familiariser avec des outils comme Git et d'apprendre à travailler en effectifs

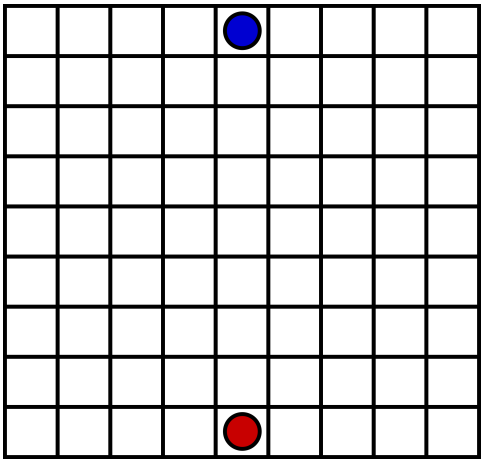

Figure 11: Situation initiale

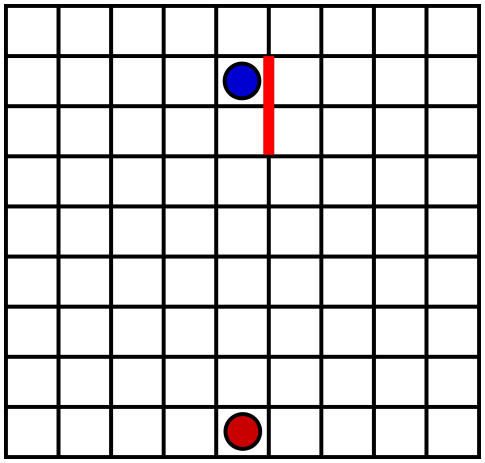

Figure 12: Deuxième tour

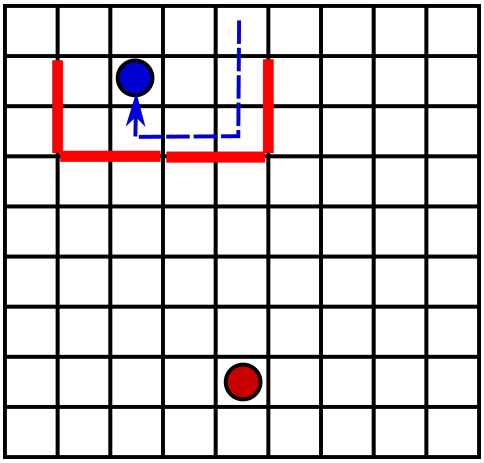

Figure 13: Objectif de l'ouverture

Figure 14: Description de l'ouverture *Side-Wall*

plus conséquents en comparaison avec le projet *Tilings*.
De plus, nous avons pu mettre en oeuvre des notions pratiques étudiées durant cette première année, comme l'implémentation de types abstraits de données. Par ailleurs, nous avons pu concrétiser des notions théoriques vues en algorithmique des graphes.
Enfin, ce projet aura servi d'introduction au développement d'une interface client-serveur et à la création de programmes capables de jouer à des jeux de société. Aussi, notre professeur encadrant nous a donné de précieux conseils et nous l'en remercions.