Rapport de Soutenance n°2

StockMeat

Antoine Jouy, Evan Reynaud, Maxence Gatard

Table des matières

1. Introduction

1.1. Avancée du projet

Cela fait maintenant 8 semaines que nous travaillons sur le projet StockMeat. En effet, nous sommes satisfaits de l'avancement du projet. Même si nous avons dû modifier le planning des tâches à réaliser du au départ d'un des membres du groupe. Ainsi voici le planning que nous nous sommes fixés après la 1ère soutenance afin d'être certains de pouvoir terminer notre projet.

Tâches	Semaine du 20/02 1ère soutenance	Semaine du 17/04 2ème soutenance	Semaine du 29/05 3ème soutenance
Composantes de l'algorithme Min-Max	Pas commencé	En cours	Terminé
Interface graphique	En cours	Terminé	Terminé
Fonctionnement d'un jeu d'échec	En cours	Terminé	Terminé
Site Web	En cours	En cours	Terminé
Autres (Relier interface et jeu, ajouts de fonctionnalités sur l'interface)	Pas commencé	En cours	Terminé

Enfin, nous avons respecté toutes les tâches que nous devions réaliser à savoir le fonctionnement du jeu et l'interface graphique du jeu. Il ne nous restera plus qu'à relier les deux afin de pouvoir jouer au jeu puis de rajouter des fonctionnalités à l'interface comme par exemple pouvoir choisir la couleur de ses pions etc...). Ensuite nous avons pu commencer à nous intéresser réellement aux composantes de l'algorithme Min-Max et enfin à commencer à développer le Site Web.

2. Tâches effectuées

2.1. Structure du jeu - Antoine JOUY

La structure du jeu est maintenant terminée, nous allons détailler les nouvelles fonctions et les derniers changements.

2.1.1. Modification - Antoine JOUY

Nous avions utilisé une liste de pièces afin de parcourir plus efficacement les pièces en jeu. Cette dernière était triée pour que les pièces appartenant à une même couleur soient regroupées en dessous de l'index 16 et les autres à partir de l'index 16 jusqu'à 31. Cependant, lorsque nous voulions effectuer un déplacement classique, la liste était modifiée. Cela est dû à l'utilisation des pointeurs de structure dynamiquement alloués. Nous avons malheureusement perdu un temps non négligeable à régler ce problème. Dorénavant, la liste de pièces a été abandonnée, nous avons perdu une légère optimisation qui pourra être regagnée plus tard. Ainsi, les pièces se sont vu enlever le champ *index* qui permettait d'y accéder depuis la liste de pièces.

Par ailleurs, afin de faciliter le déroulement des appels des fonctions, le champ *realPlayerColor* leur a été ajouté. Il correspond à la couleur du vrai joueur (car l'autre camp sera contrôlé par l'IA). Il est très important d'avoir cette information car elle permet d'orienter l'échiquier vers le joueur, et de déterminer le sens de mouvement des pions. Nous avons aussi ajouté le champ *hasMoved* permettant de savoir si une pièce a bougé ou non (utile pour les roques).

Voici la nouvelle structure des pièces.

```
struct piece {
    int x;
    int y;
    enum Role role;
    enum Color color;
    int value;
    struct list *possibleMoves;
    int realPlayerColor;
    int hasMoved;
};
```

2.1.2. **Mouvement** - Antoine JOUY

```
int move(struct piece** board, struct piece* piece, int x , int y);
```

Cette fonction appelle *getMoves()* qui calcule les coups possibles de la pièce en paramètre. Si dans ces coups se trouve la case souhaitée (aux coordonnées *x,y*) alors la fonction va calculer si ce coup n'engendre pas un échec dans son camp. Pour se faire, la pièce est échangée avec la case aux coordonnées souhaitées avec la fonction *swap()*. Cette dernière prend en paramètre les deux pointeurs de structure de pièce et va échanger leurs places ainsi que leurs champs *x* et *y* qui représentent les coordonnées de la pièce sur le plateau. Une fois les deux cases échangées, la fonction *isCheck()* est appelée.

```
int isCheck(struct piece** board, int color, int index);
```

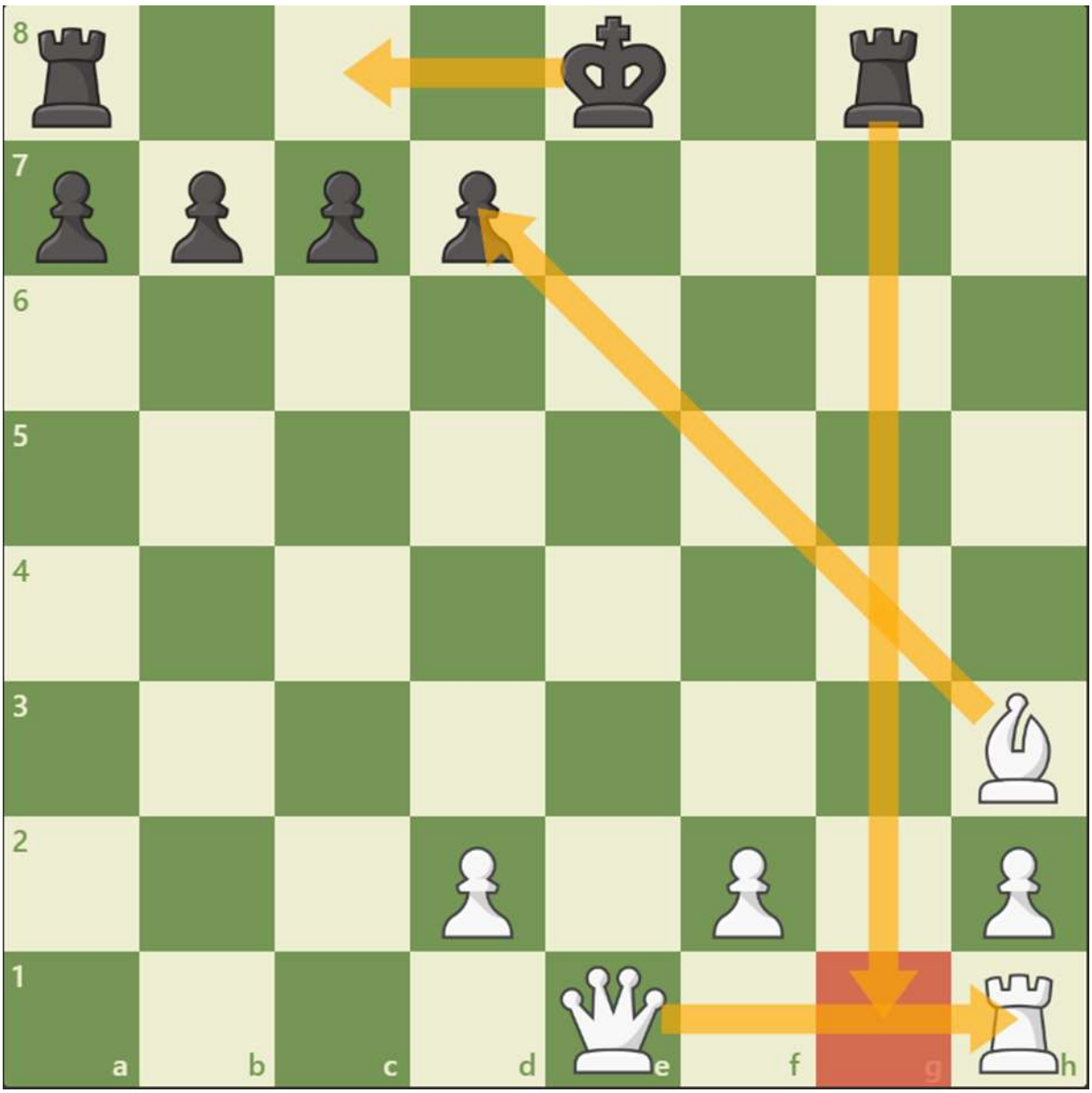
Cette fonction parcourt toutes les pièces adverses et vérifient si le roi du camp de la pièce jouée est menacé. L'index de la pièce mettant en échec le roi est retourné si c'est le cas, -1 sinon.

Ainsi, si la valeur retournée est positive, la fonction move() réinitialise le mouvement et retourne 0. Sinon, elle vérifie si une pièce a été mangée par le coup. Comme nos cases sont des pointeurs de structures et qu'elles ont été échangées, il faut regarder le rôle de la case d'où partait le coup. Si ce rôle est différent de 0 (valeur d'une case vide) alors il faut mettre à jour cette case car la pièce a été mangée. Le champ indiquant si la pièce a été bougée est ensuite mis à 1 et la fonction retourne 1.

2.1.3. **Roque** - Antoine JOUY

Avant de pouvoir faire un roque, il faut vérifier 4 choses :

- Le roi et la tour n'ont jamais bougé.
- Le roi ne doit pas être en échec (on ne peut pas fuir un échec avec un roque).
- Le roi ne peut pas traverser de case attaquée par l'adversaire.
- Il ne doit y avoir aucune pièce entre le roi et la tour.



Par exemple, les noirs peuvent roques car le fou est bloqué par un pion. Cependant, les blancs ne peuvent pas car la tour menace.

Ainsi les fonctions *canShortCastle()* et *canLongCastle()* vérifient ces conditions.

```
int canShortCastle(struct piece** board,struct piece* piece);

int canLongCastle(struct piece** board, struct piece* piece);
```

Elles font exactement la même chose, simplement, la tour prise en compte n'est pas la même.

Ces fonctions sont respectivement appelées par *shortCastle()* ou *longCastle()* et si le roque est possible, alors le roi et la tour sont échangés via la fonction *swap()*.

2.1.4. **Promotion** - Antoine JOUY

Un pion peut être promu en n'importe quelle pièce si ce dernier se trouve sur la dernière ligne adverse.

```
void promotion(struct piece* piece, int role);

int canPromote(struct piece* piece);
```

La fonction *canPromote()* vérifie que la pièce en paramètre est bien un pion et qu'il se trouve aux bonnes coordonnées, si c'est le cas elle retourne 1, sinon 0. Cette fonction est appelée par *promotion(),* si la valeur retournée est 1, alors grâce à un *switch()*, la pièce en paramètre va être mise à jour avec le rôle souhaité, sinon rien ne se passe.

```
void promotion(struct piece* piece, int role)
{
    if(canPromote(piece))
    {
        switch(role)
        {
            case(QUEEN):
                piece->role = QUEEN;
                piece->value = 9;
                break;
            case(ROOK):
                piece->role = ROOK;
                piece->value = 5;
                break;
            case(BISHOP):
                piece->role = BISHOP;
                piece->value = 3;
                break;
            case(KNIGHT):
                piece->role = KNIGHT;
                piece->value = 3;
                break;
        }
    }
}
```

2.1.5. **Echec et mat** - Antoine JOUY

```
int checkMate(struct piece** board,int KingColor);

int cannotProtectKing(struct piece** board,int dangerousmoves[8], int KingColor, int cpt);
```

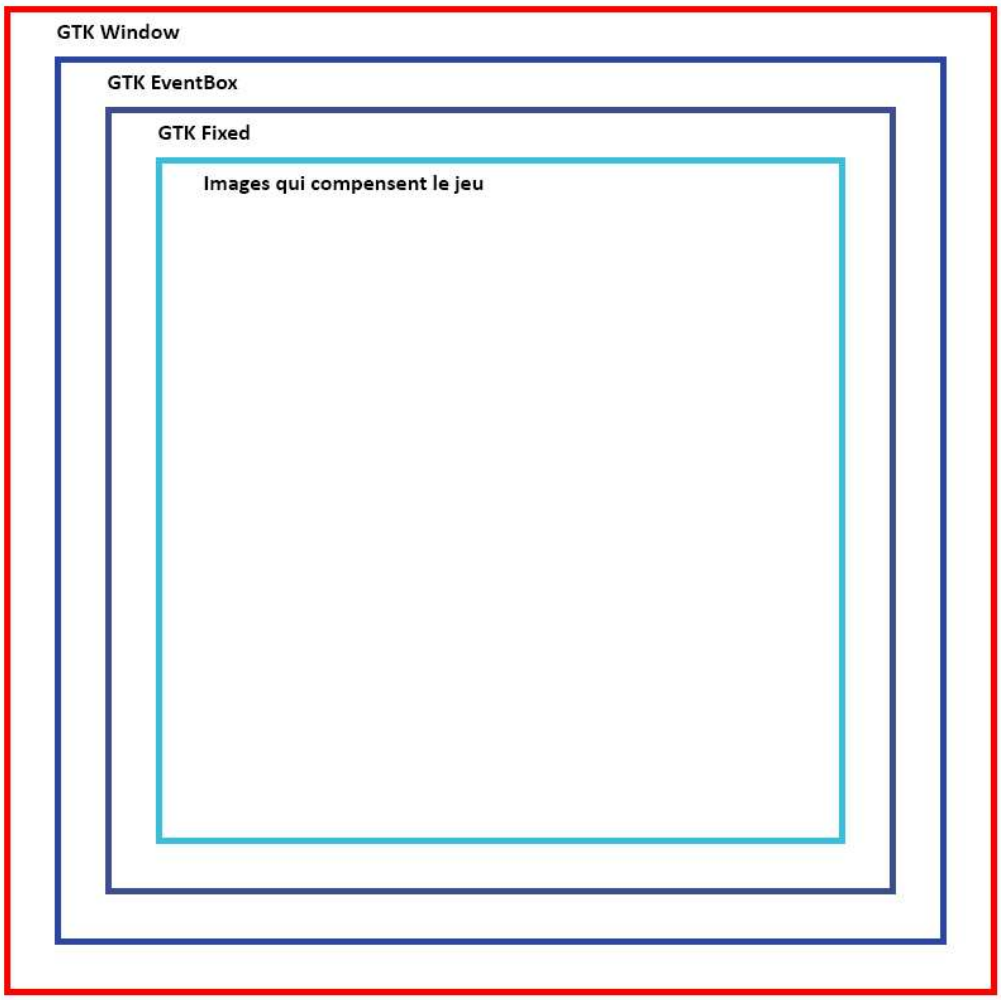
La fonction *checkMate()* ne sera appelée qu'en cas d'échec. La couleur du roi impliqué lui sera donnée en paramètre. Elle va vérifier pour tous les mouvements possibles du roi si ce sont des mouvements échappatoires. Si un mouvement est trouvé, elle retourne 0, sinon, toutes les coordonnées des cases vides autour du roi sont stockées dans un tableau d'*int* de taille 8 (*dangerousmoves*) donné en paramètre de la fonction *cannotProtectKing()*. Cette fonction va vérifier que le roi ne peut pas être protégé par une des pièces de son camp. Elle va parcourir toutes ses pièces, vérifier si une pièce peut s'interposer ou non. Pour ce faire, pour chaque coup, elle va parcourir le tableau *dangerousmoves* et vérifier si des coordonnées correspondent avec les coordonnées du coup possible. Si c'est le cas, la case est mise à -1.

Une fois que toutes les pièces ont été parcourues, elle va vérifier si toutes les cases du tableau *dangerousmoves* sont à -1. Si une case n'a pas cette valeur, le roi ne peut donc pas être protégé et la fonction retourne 1, sinon le roi est protégé et elle retourne 0. Cette même valeur va être retournée par la fonction *checkMate()*.

2.2. **Interface graphique** - Maxence Gatard

En ce qui concerne l'interface graphique, nous avons rencontré de nombreuses difficultées. Nous sommes partis totalement à l'opposé de ce que nous pensions faire au départ, à savoir soit utilisé des Gtk Grids afin d'avoir un côté pratique pour développer l'interface ou soit utiliser des Gtk DrawingArea afin de pouvoir afficher les pions et que ce soit beaucoup plus esthétique. Finalement, nous avons donc réussi à lier le côté simple de développement mais aussi le côté esthétique en utilisant un Gtk Fixed et un Gtk Event Box.

Voici comment se décompose l'interface :



Les éléments sont tous imbriqués les uns dans les autres sans les marges de l'image qui sont là juste dans le but de pouvoir bien visualiser. Le GTK EventBox nous permet donc de pouvoir récupérer les coordonnées de

chacun des clics de l'utilisateur. (Par exemple s'il clique en haut à gauche on aura un x = 0 et un y = 0). Quant à lui, le GTK Fixed va nous permettre de pouvoir positionner des GTKWidget donc des images, des boutons à des coordonnées fixes, ce qui nous permettra de pouvoir déplacer nos pions correctement.

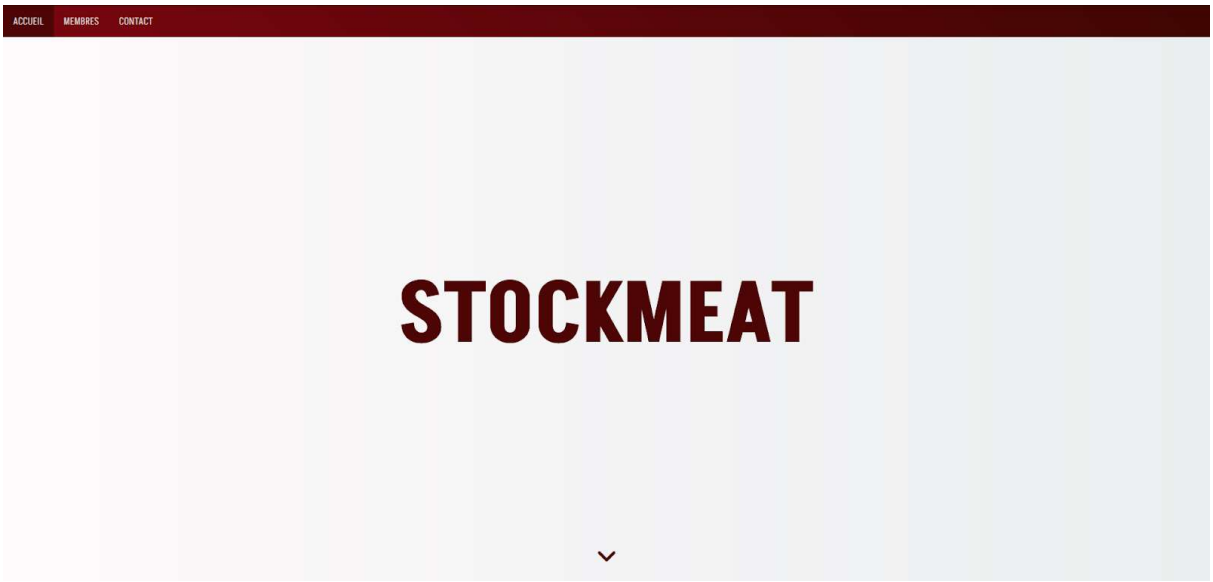
Pour ce qui est des "assets", nous avons récupéré celles du site chess.com afin de pouvoir tester notre code mais en vue de la prochaine soutenance nous en utiliserons d'autres que nous créerons.

Voici à quoi ressemble l'interface :

2.3. **Site Web** - Maxence Gatard

Le site web n'en est qu'à ses débuts, en effet nous n'avons pas encore eu le temps de le finaliser et de l'alimenter des différents éléments mais la maquette de base est ce qui se rapproche le plus du résultat final en termes de design. Par ailleurs, la version téléphone est aussi et le site sera totalement terminé en vue de la dernière soutenance, il nous faudra rajouter la page de la présentation des membres ainsi qu'une page pour télécharger les différents rapports de soutenance et aussi le jeu.

Voici la maquette :



lien du site : https://le-jy.github.io/StockMeatWeb/

3. Prochaine soutenance

Pour la prochaine soutenance, Maxence et Antoine devront relier l'interface graphique et le jeu tout en gardant le site web à jour. Evan commencera la mise en place de l'algorithme min-max et sera rejoint par Antoine et Maxence une fois leur tâche effectuée.

4. Conclusion

Nous sommes satisfait du travail accompli depuis la dernière soutenance. Nous avons dû faire face à un abandon d'un des membres du groupe ce qui a provoqué un changement des deadlines initiales. Nous ne pensions pas que mettre en place le jeu d'échecs en lui-même allait être aussi long et difficile. Cependant, nous avons toujours réussi à corriger les bugs rencontrés et nous avons vu notre capacité d'analyse s'améliorer à vue d'œil.