

Rapport de Troisième Soutenance

Les Albatros

EPITA



Réalisé par

Félix Lena • Loïc Segundo • Theo Daronat • Alan Gueret

Fait le 4 Juin 2021

Table des matières

1	Introduction	4
2	Présentation du groupe	5
2.1	Formation du groupe	5
2.2	Alan Gueret	6
2.3	Loïc Segundo	6
2.4	Félix Lena	6
2.5	Théo Daronat	7
3	Organisation	8
3.1	Workflow	8
3.2	Répartition des tâches	8
3.3	Progression	9
3.4	Structure du projet	11
3.5	Visualisation du repos git	12
4	Avancement	13
4.1	Les utilitaires	13
4.1.1	La structure Game	13
4.1.2	Les listes chaînées	16
4.2	Les règles	17
4.2.1	L'aventure king_suicide	20
4.2.2	Échec et mat	22
4.2.3	Égalité	22
4.3	L'affichage	23
4.4	l'algorithme MinMax	25
4.4.1	Premiers essais	25
4.4.2	Les fonctions d'évaluation	25
4.4.3	Mise en place de l'algorithme	26
4.5	L'algorithme Alpha-Bêta ou l'enfer de l'élagage	28
4.5.1	La structure Tree	28

4.5.2	Conception et Mise en place	29
4.6	Ajustement de MinMax et d'Alpha-Bêta	31
4.6.1	Mouvements complexes	31
4.6.2	Modification de la structure Tree	32
4.7	Online	33
4.7.1	Le serveur	33
4.7.2	Le client	33
4.7.3	La communication c'est la clef!	34
4.7.4	Le jeu : options propre à la version en ligne	34
4.8	Le réseau de neurones	35
4.8.1	Réflexion et projet sur le réseau de neurones	35
4.8.2	Structure du réseau de neurones	36
4.8.3	Principe général	39
4.8.4	Le fonctionnement	40
4.8.5	Les outils nécessaires	40
4.8.6	Optimisation	41
4.8.7	L'adaptation au problème	42
4.9	Déroulement du jeu	43
4.10	Travaux inachevés	44
5	Conclusion	45
5.1	Expérience individuelle	45
5.1.1	Alan	45
5.1.2	Loïc	46
5.1.3	Félix	47
5.1.4	Théo	47
5.2	Conclusion	49
6	Annexe	50

Chapitre 1

Introduction

Nous voilà maintenant arrivés en S4, l'OCR étant terminé, il est maintenant temps pour nous, élèves, de se consacrer corps et âme dans un nouveau projet. Avec ce premier semestre nous avons réussi à accumuler un certain bagage. Normalement le langage C, notamment grâce au projet de S3 et aux travaux pratiques, ne nous fait plus peur et n'est plus une barrière. De ce fait, il nous sera envisageable de proposer des sujets et des idées plus complexes et plus profondes que ce que nous aurions proposé auparavant.

Chapitre 2

Présentation du groupe

2.1 Formation du groupe

Qui dit S4 dit également semestre à l'étranger. Enfin seulement pour ceux qui ont eu le courage et la chance de proposer une candidature dans une des nombreuses écoles et ayant été accepté. De plus, beaucoup de classes ont été fusionnées. De ce fait, il était quasiment impossible de recréer le même groupe que pour le projet OCR du S3.

Mais il y avait quand même des restes ! Les inséparables, Alan et Théo, savaient depuis un moment qu'ils seraient dans le même groupe, mais il manque encore deux personnes. Félix étant dans la même classe qu'eux et ayant déjà discuté à de nombreuses reprises entre eux, il fut naturel que ce dernier intègre le groupe. Maintenant il ne reste plus qu'un membre à trouver, mais qui cela pourrait-il bien être ? Se connaissant par le biais d'un ami commun et ayant appris qu'ils seraient dans la même classe, Loïc entra en contact avec le trio et ainsi intégra le groupe et permis à ce dernier d'être enfin complété.

Il a fallu ensuite décider du sujet sur lequel nous allions travailler durant tout un semestre. Il est donc hors de question de prendre un sujet qui ne plairait pas à tout le monde. Chacun proposa alors quelques idées et nous avons procédé à un vote, l'objectif étant de dégager les thèmes qui nous plairaient le plus. Au final, nous nous sommes aperçu que nous avons tous un intérêt pour le jeu d'échec. Seulement nous avons peur que cette idée soit rejetée car ne correspondant pas aux caractéristiques demandées ou quelle soit considérée comme trop simple. Nous avons donc pris la décision de créer notre cahier des charges sur ce sujet en nous mettant d'accord sur un autre sujet qui serait plus susceptible d'être accepté. Après un premier rejet nous demandant d'augmenter la difficulté sans changer de sujet, notre projet d'intelligence

artificielle de jeu d'échec à été accepté. Et c'est ainsi que nous avons pris le chemin nous menant vers les péripéties, s'annonçant pour le moins intéressantes, de notre nouveau projet.

2.2 Alan Gueret

Mes expériences de projets ont la plupart du temps été très bonnes et m'ont laissé de bons souvenirs. Les personnes de mon groupe sont des personnes que j'apprécie et qui me pousseront à donner le meilleur de moi-même. Nous sommes tous motivés et déterminés à mener ce projet à bien. J'ai déjà réalisé plusieurs projets avec Théo et je sais que c'est quelqu'un de travailleur. Mes deux autres coéquipiers sont tout aussi sérieux et je pense que l'on va réussir à garder une bonne ambiance de travail au sein du groupe. Grâce à tout ceci j'ose espérer que nous arriverons à finir notre projet dans les temps. Je sens que je vais apprendre beaucoup de nouvelles choses avec ce projet et avec ce groupe de travail.

2.3 Loïc Segundo

Loïc Segundo, 20 ans, et libre comme l'air ! Tout semble se passer mieux que dans mes précédentes expériences en groupe. Comme beaucoup je vois ce projet comme un point de contrôle juste avant le cycle ingénieur, il me permet de voir si je suis au point ou non avec mes camarades. Bien que ce projet ne soit pas mon choix personnel, j'arrive à voir en quoi il est intéressant de travailler dessus au-delà de l'expérience en groupe. C'est la fin l'ambiance est toujours au TOP, tout s'est bien passé, tout le monde est content.

2.4 Félix Lena

Ça y est on a fini ! Après des semaines de travail régulier, nous avons enfin terminé le projet. Il y a eu de nombreuses complications et beaucoup de recherche de bugs, mais enfin nous pouvons rendre notre projet. J'en suis fière. J'étais l'initiateur de ce projet de jeu d'échec et ne regrette pas car on s'est bien amusé. Les connaissances en langage C apprises lors du S3 et S4 n'ont pas été superflues, je suis content d'avoir travaillé les TPs car ils m'ont permis d'être efficace tout au long du projet. J'avais un attrait plutôt pour le réseau de neurones initialement, mais le travail sur le réseau en ligne m'a bien amusé.



2.5 Théo Daronat

La première fois que j'ai codé, c'était à l'Epita. Mon expérience en programmation se limite donc aux travaux pratiques ainsi qu'au projet de S2 et de S3. Mais au fil du temps j'ai appris et j'ai dorénavant un peu plus confiance en mes capacités et j'essaierais d'être utile à l'équipe du mieux que je peux. J'ai déjà fait équipe avec Alan pour notre projet de jeux vidéo de S2 ainsi que pour le projet OCR du S3. Je sais déjà que je peux compter sur lui et que nous fonctionnons bien ensemble. Mais je sais également que mes autres camarades sont déterminés et persévérants, donc je suis amplement rassuré pour ce projet ! J'attends beaucoup de ce dernier, cela fait maintenant un bon moment que je réfléchissais à coder un jeu d'échec et que je voulais aborder d'autres conceptions de l'intelligence artificielle. Ce projet est la parfaite occasion pour m'y lancer !



Chapitre 3

Organisation

3.1 Workflow

Nous avons commencé ce projet en créant un repos git dont le nom vous a été donné sur la page de couverture de ce rapport. Ce repos nous accompagnera durant tout ce projet et nous permettra une mise en commun du code assez aisée, étant donné soit peu que nous sachions l'utiliser.

Nous avons également pris la décision, sous la requête avisée de Loïc, d'utiliser Trello. Ce dernier est un outil de gestion de projet en ligne. Il nous permettra de créer de nouvelles tâches à faire ainsi que d'y assigner des personnes. Ceci va nous permettre de faciliter grandement notre organisation ainsi que de fluidifier notre progression pour ce projet.

Nous arrivons aux termes de ce projet et notre organisation a tenu le coup ! Nous avons réussi à avoir un travail régulier et efficace, ce qui nous a grandement aidé à prendre du plaisir dans ce projet.

3.2 Répartition des tâches

	Jeu d'échecs	Intelligence Artificielle	P2P	Réseau de neurones	Site web
Loïc	x			x	x
Théo	x	x	x		x
Félix	x	x	x		x
Alan	x			x	x

3.3 Progression

	Jeu d'échec
Fait	<ul style="list-style-type: none">- Implémentation de la structure principale- Création des différents mouvement relatifs à chaque pièce- Restrictions de déplacement de certaines pièces lors de- Implémentation de la structure liste chaînée ainsi que ses principales fonctions de manipulation- Création de certaines restrictions de déplacement des pièces- Affichage du plateau en fonction de la couleur en train de jouer. certaines situations- Création de certaines règles tel que échec et mat ainsi que nul.

	Réseau serveur - client
Fait	<ul style="list-style-type: none">- Création du serveur- Création du client- Communication et transfert de données- Compatibilité avec le jeu- Implémentation dans la fonction principale du jeu

	MinMax
Fait	<ul style="list-style-type: none"> - Implémentation de l'algorithme principale - Création de la structure arbre - Les fonctions d'évaluations - Mettre en lien les différentes parties créées - Ajustement de certaines mécaniques de jeu.
	Alpha-Bêta
Fait	<ul style="list-style-type: none"> - Implémentation de l'algorithme principale - Utilisation de la structure arbre - Élagage - Ajustement de certaines mécaniques de jeu.

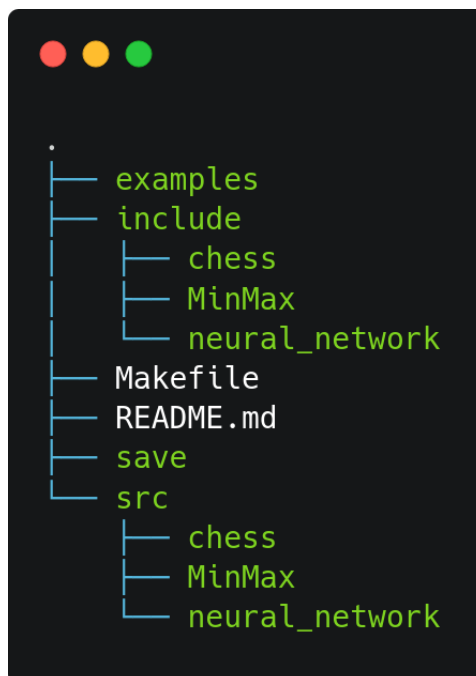
	Réseau de neurones
Fait	<ul style="list-style-type: none"> - Création des différentes structures - Création des fonctions essentielles - Algorithme de sélection génétique - Optimisation de l'apprentissage - Ajuster le réseau de neurones pour qu'il puisse utiliser les fonctions d'évaluation de MinMax

3.4 Structure du projet

Pour ce qui est de la structure du projet, nous avons décidé d'améliorer. Une pour les fichiers headers dans le dossier "include" et une autre pour les fichiers sources dans le dossier "src". Dans ces dossiers, il y a des également dossiers qui permettent de séparer les différentes parties du projet.

Nous avons également un Makefile qui nous simplifie la compilation du projet. Ce Makefile nous permet d'avoir plusieurs fichiers tests qui sont dans le dossiers "examples" et de les compiler indépendamment.

Voici la structure principale du projet :

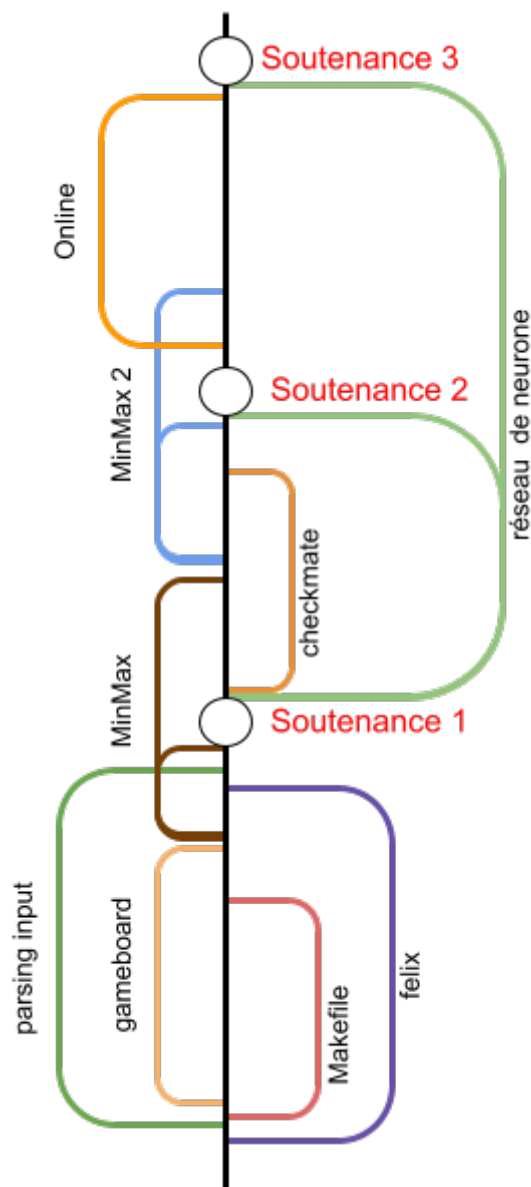


Vous pouvez voir la structure complète du projet en annexe de ce rapport.

La structure actuelle du projet n'est pas définitive mais elle se rapproche tout de même de l'aspect final qu'elle devrait avoir. Nous trouvons cette architecture bien pratique. De plus, notre Makefile a été fait dans l'optique de faciliter la création de nouveaux main ainsi que de nouveaux fichiers .c et .h

3.5 Visualisation du repos git

Voici un schéma des branches principales que nous avons créées au cours de notre projet :



Vous pouvez retrouver l'évolution plus détaillé en annexe.

Chapitre 4

Avancement

4.1 Les utilitaires

4.1.1 La structure Game

Nous avons commencé ce projet en construisant les différentes structures sur lesquelles va reposer notre jeu d'échec. Nous avons donc créé la structure Game.

```
typedef struct Game
{
    Piece *board[64];
    Piece blacks[16]; //every black pieces
    Piece whites[16];
}Game;
```

Comme vous pouvez l'observer, nous avons décidé d'utiliser un tableau du type Piece d'une taille de 64. Mais lors de la conception de la structure, nous nous sommes dit qu'il serait également intéressant d'avoir deux autres tableaux de taille réduite contenant uniquement les pièces de chaque couleur. Étant donné qu'il nous faudra régulièrement accéder aux différentes pièces, il est bien plus intéressant d'avoir des listes séparés qu'une seule liste de 64 à parcourir case par case. Alors pourquoi ne pas seulement utiliser ces deux tableaux? Et bien parce que la représentation du board

est également intéressante pour savoir s'il y a des pièces entre deux pièces données, il est plus intéressant de parcourir 5 cases de l'échiquier que de parcourir les deux listes de 16 pièces.

Donc nous avons également implémenté une structure Piece

```
typedef struct Piece
{
    enum pieces_types type;
    enum pieces_colors color;
    int alive;
    int moved;
    int x;
    int y;
}Piece;
```

Cette structure contient plusieurs informations : le type de la pièce (si c'est un cavalier, un roi, un pion, etc...), sa couleur (noir ou blanc), le fait qu'elle soit en vie ou non, le fait qu'elle ait déjà bougé ainsi que sa position dans l'échiquier.

Je pense que le fait de stocker le type ainsi que la couleur et la position actuelle de la pièce est assez évident. Mais pourquoi voulons nous stocker le fait que la pièce soit en vie ainsi que le fait que la pièce ait déjà bougé ? Et bien pour des raisons pratiques. En effet, lorsque nous voudrons user de l'algorithme minmax, il sera plus simple de faire revivre une pièce en changeant un int qu'en la copiant au préalable au cas où elle serait tué lors du tour. Et pour ce qui est de moved, il est intéressant pour certains mouvements tels que le premier déplacement des pions ou le rock avec le roi.

Comme je viens de l'évoquer, et vous l'avez sûrement vu, nous avons créé deux enums nous permettant de définir dans un premier temps le type de la pièce et dans un second sa couleur. Vous vous en doutez sûrement, ils sont extrêmement basiques. Et en même temps, j'aimerais vous demander pourquoi faire compliquer quand on peut faire simple !

```
enum pieces_types
{
    PAWN,
    ROOK,
    KNIGHT,
    BISHOP, //bishop -> fou
    QUEEN,
    KING
};
```

```
enum pieces_colors
{
    BLACK=0,
    WHITE=1
};
```

Comme vous pouvez le voir, nous avons défini une valeur à la couleur BLACK ainsi qu'à la couleur WHITE, et cela pour une bonne raison ! En effet c'est plus ou moins dans la même réflexion que pour notre ami 'alive'. Nous allons devoir très régulièrement vérifier dans Minmax si nous sommes en présence d'un coup allié ou d'un coup ennemi. Le score résultant dans ce cas est le même au signe près. Avoir donné des valeurs aux deux couleurs nous permet d'éviter des 'if' un peu lourds en les remplaçant par un simple calcul.

4.1.2 Les listes chaînées

Pour gérer les listes de coups nous avons deux choix, une implémentation statique ou dynamique. L'implémentation statique implique de réserver un tableau de taille prédéfinie. La taille aurait été le nombre de coups maximal pour une pièce donnée; par exemple 4 pour le pion ou 8 pour le roi. Mais cela aurait été une occupation de la mémoire évitable avec les listes chaînées. Le principe est donc simple : une structure pointant vers l'élément suivant et qui stocke les coordonnées d'un coup. Au début de cette liste on place une sentinelle, un élément invariable et marquant le début de la liste simplifiant la manipulation des listes.

La structure `Move_list` nous permet de stocker les informations relatives à la position après déplacement que peut prendre une pièce.

```
typedef struct Move_list
{
    struct Move_list *next;
    int x;
    int y;
}Move_list;
```

Nous avons donc également implémenté plusieurs fonctions nous permettant de manipuler plus facilement cette liste chaînée.

Pour cela nous avons créé les fonctions suivantes :

- `init_list()` qui nous permet d'initialiser une nouvelle liste chaînée.
 - `is_empty()` retourne une erreur dans le cas où la sentinelle de la liste est non initialisée, sinon elle retourne 1 dans le cas où la liste est vide, sinon elle retourne 0.
 - `add_list()` va créer une nouvelle `Move_list` contenant les coordonnées passées en paramètre et qui l'ajoutera en tête de la liste chaînée donnée.
 - `pop_list()` donne aux pointeurs passés en paramètre les valeurs contenues dans le premier élément de la liste qui sera ensuite free. Dans le cas où la liste ne contient plus d'élément autre que la sentinelle, la fonction renvoie 0, sinon elle renvoie 1.

- `in_list()` retourne 1 si un élément de la liste chaînée possède les mêmes informations que celles passées en paramètre, sinon retourne 0.
- `free_list()` permet de free toute la liste chaînée et non pas qu'un seul élément.
- `display_list()` permet d'afficher le contenu de toute la liste chaînée dans le terminal.

```
Move_list* init_list();

int is_empty(Move_list* list);

void add_list(Move_list* list, int x, int y);

int pop_list(Move_list* list, int* x, int* y);

int in_list(Move_list* list, int x, int y);

void free_list(Move_list* list);

void display_list(Move_list* l);
```

4.2 Les règles

Bon, c'est bien beau d'avoir des listes chaînées, mais il s'agirait de les utiliser! Nous avons créé une fonction permettant de créer une liste contenant tous les mouvements possibles d'une pièce dans l'état actuel du jeu. Pour cela nous nous sommes munis de plusieurs fonctions permettant d'obtenir les cases accessibles à partir de la position de la pièce. Ceci pour nous éviter une fonction avec un switch faisant approximativement 200 lignes. Nous avons donc une fonction par type de pièce, même si pour la reine nous avons simplement fait appel aux fonctions donnant les déplacements du fou et de la tour.

```

Move_list* get_knight_moves(Game* g, Piece* p);
Move_list* get_pawn_moves(Game* g, Piece* p);
Move_list* get_rook_moves(Game* g, Piece* p);
Move_list* get_king_moves(Game* g, Piece* p);
Move_list* get_bishop_moves(Game* g, Piece* p);
Move_list* get_queen_moves(Game* g, Piece* p);
Move_list* get_moves(Game* g, int x, int y);

```

Mais avant cela il faut obtenir l'input de l'utilisateur qui sera sous forme de coordonnée comme par exemple : "C2" ou "G7". Pour cela nous avons une fonction `can_i_go()` permettant de faire le parsing de l'entrée (géré par une autre fonction). Et tant que l'input n'est pas valide, le programme redemande à l'utilisateur de donner une nouvelle entrée. Ensuite, `can_i_go()` va obtenir le type de la pièce se trouvant à la position donnée. si la position indique un emplacement non valide comme une case vide ou une case où se situe une pièce ennemie, la fonction demande une nouvelle entrée à l'utilisateur.

```

void can_i_go(Game *game, int *x, int *y, Move_list **li, enum pieces_colors c);

```

```

int get_piece(Game* g, int x, int y, Piece** p);

```

Eh bien nous avons tous les déplacements! Cela n'était pas très compliqué. A moins que... Nous avons oublié un détail! Certes une pièce peut se déplacer sur les cases auxquelles elle a accès, hormis quand elle défend le roi d'un échec! Dans ce cas, il est impossible de déplacer la dite pièce. Pour cela nous regardons avec un calcul si notre pièce est dans l'axe du roi. Si c'est le cas, nous regardons s'il n'y a pas de pièces entre notre pièce et le roi. Après cela, nous regardons dans la direction opposée pour voir s'il y a une pièce ennemie. S'il y a une pièce ennemie et si elle est d'un type qui mettrait le roi en danger alors nous renvoyons une valeur indiquant que cette pièce ne peut pas bouger.

```
int is_treason(Game *g, Piece *p);
```

Nous venons de parler des pièces qui devaient protéger le roi. Or la tâche de ce dernier est si importante que le suicide lui est interdit! Et donc cela implique que tous les déplacements qui impliquent une mise en échec du roi doivent être supprimés. Pour cela nous avons la fonction `king_suicide()` qui va supprimer les déplacements possibles de la `Move_list` du roi qui coïncident avec un déplacement possible des pièces de l'adversaire.

```
void king_suicide(Game *g, Piece *p, Move_list *king_moves);
```

Nous avons également une fonction qui permet de nous informer si le roi est en échec.

Il ne nous reste à faire que la fonction d'échec et mat ainsi que nul. Ce qui serait pratique pour pouvoir finir une partie!

4.2.1 L'aventure king_suicide

J'aimerais vous conter l'histoire d'un jeune étudiant en informatique qui pensait avoir eu une idée fort pratique et simple à réaliser, mais qui se trompait. Durant les deux semaines de vacances, que nous avons mis à profit pour avancer sur le projet, il ne s'est pas passé une seule journée sans que cette magnifique fonction qu'est king_suicide ne soit modifié.

Pour faire simple, l'objectif de cette fonction est de faire en sorte de restreindre les déplacements du roi lorsqu'il est sélectionné.

Dans un premier temps, l'idée était de regarder si certains des déplacements du roi coïncidaient avec les déplacements des pièces ennemies.

Sauf que oui, mais non. Certes, le roi ne peut pas se déplacer sur une case qui est menacé par un ennemi. Mais lorsqu'il est en échec, c'est à dire directement menacé, il ne peut pas se déplacer dans l'axe de la pièce ennemie. Prenons un exemple, le roi est mis en échec par une tour.

De ce fait, en l'état actuel de l'algorithme, nous retirons le mouvement du roi vers la tour car c'est un mouvement possible de la tour. Seulement le roi n'a également pas le droit d'aller dans le sens opposé. Étant menacé, ce dernier n'a pas le droit de rester dans l'axe de la tour.

Donc à l'aide d'un petit calcul de la différence de position du roi et de la pièce ennemi on peut savoir, en fonction du type de la pièce, si elle est dans l'axe du roi.

Bien, ça n'était pas si compliqué que ça au final. A moins que... En effet, maintenant le roi voit ses déplacements restreints par le fait que la pièce soit dans l'axe, mais il y a un autre problème. Lors d'un test avec nos IA, nous nous sommes rendu compte de quelque chose. Nous avons pu nous échapper d'un échec et mat en mangeant une pièce qui était protégé par une autre pièce ennemi. C'est un mouvement que nous n'aurions pas normalement dû faire. Et bien c'est reparti pour un tour!

Cette fois-ci la solution fut plus simple, en plus de récupérer la liste des mouvements de la pièce de l'adversaire, nous allons récupérer sa liste de défense et la comparer à la liste de mouvements du roi, et cela en même temps.

Bien, bien, bien. Est-il encore nécessaire de créer un prétendu suspense avant d'annoncer que nous avons fait face à un nouveau problème ? En effet par soucis d'optimisation, lorsqu'une pièce était dans l'axe du roi, nous supprimions que deux mouvements s'ils existaient : ceux se situant sur l'axe former avec le roi et la pièce ennemi.

Seulement il existe un cas particulier à ce concept que nous n'avions pas imaginé. Si la tour est collé au roi, que ce passe-t-il à votre avis ? Notre algorithme va définir que la tour est dans l'axe du roi, il va chercher le premier mouvement que le roi et la tour ont en commun pour le supprimer et va supprimer le mouvement opposé puis quitter.

Hors ce n'est pas ce que nous voulons ! Il a donc fallut créer un cas particulier lorsque la tour est sur une case adjacente au roi. Dans ce cas précis la tour n'est plus considéré dans l'axe du roi et en plus de chercher à supprimer tous les mouvement en commun du roi avec la tour, nous cherchons également à supprimer le mouvement du roi qui correspond à l'opposé de la position de la tour par rapport au roi.

Et cette fois-ci c'est fini, pour de vrai ! Nous n'avons plus eu besoin d'y retoucher. La bataille fut longue, mais nous l'avons remporté !

4.2.2 Échec et mat

Le principe de la règle d'échec et mat est facile à comprendre mais est déjà plus difficile à visualiser sur un vrai plateau de jeu suivant les situations. Ce qui rend cette règle difficile à détecter est surtout le fait que les mouvements de chaque pièces peuvent empêcher l'échec et mat.

Tout d'abord on commence par tester si l'on est dans une situation d'échec, si nous ne sommes pas dans cette situation, on peut directement arrêter le test. Il faut ensuite tester après chaque coups de chacune des pièces de l'équipe du roi qui est potentiellement en échec et mat si il y a échecs. Si une pièce peut empêcher l'échec et mat, cette fonction s'arrête immédiatement. Nous savons que cette fonction est très coûteuse, de plus cette fonction sera lancé à chaque tours pour détecter un échec et mat dès qu'il se produit.

4.2.3 Égalité

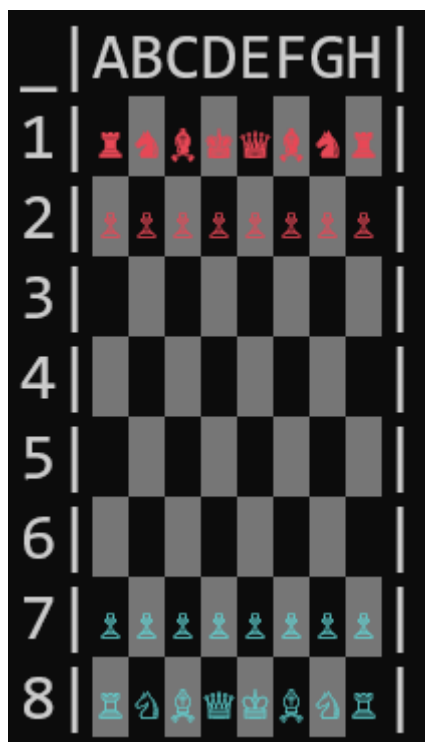
L'égalité intervient lorsqu'un joueur ne peut pas jouer lors de son tour. Cette égalité intervient même si l'un des deux joueurs a un avantage important sur l'autre.

Pour tester si nous sommes dans une situation d'égalité, il faut savoir si un joueur a la possibilité de bouger une pièce lors de son tour. Si ce n'est pas le cas, la partie s'arrête avec une égalité.

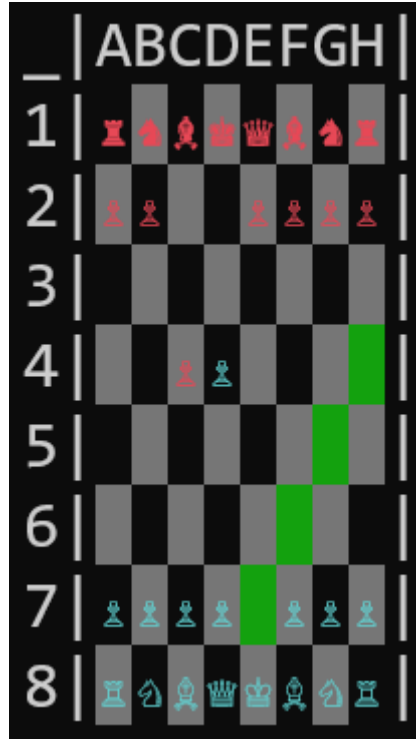
4.3 L'affichage

La fonction qui permet d'afficher le plateau de jeu n'est pas très complexe puisque le plateau est stocké dans un tableau. Il nous suffit de parcourir ce tableau en affichant la pièce présente dans chacune des cases tout en marquant si cette case est noire ou blanche. Lorsque c'est le tour de l'autre joueur on tourne le plateau de jeu en parcourant le tableau à l'envers.

Pour pouvoir afficher le tableau d'échec ainsi que les pièces proprement dans la console, nous avons utilisé des caractères Unicode et des couleurs. Au début, nous ne savions pas que nous pouvions afficher des caractères en couleurs dans le terminal et nous pensions que ne pouvions utiliser que des caractères Unicode. Mais après quelques recherches, nous avons appris à mettre ces caractères en couleurs grâce aux caractères d'échappement ANSI. Après avoir fait plusieurs essais sur des terminaux différents et avec des couleurs différentes, nous sommes arrivés au résultat suivant.



Une fois que ceci a été réalisé, il fallait maintenant afficher les déplacements possibles de la pièce sélectionnée. Pour se faire nous avons implémenté une fonction qui prend la liste des coups possibles d'une pièce et nous créons un tableau de 64 bit stockés dans un "unsigned long" où un 1 est un coup possible. Cette méthode nous évite d'avoir à parcourir la liste des coups possibles à chaque nouvelle case.



4.4 l'algorithme MinMax

4.4.1 Premiers essais

Le groupe ayant été chargé de travailler sur l'algorithme MinMax a, dans un premier temps, voulu mettre en place un algorithme basique pour bien comprendre son fonctionnement et voir à quoi ce dernier pourrait ressembler.

Cette première ébauche ne comprenait pas de structure particulière. L'objectif était de faire des appels récurifs sur chaque coup possible dans une situation jusqu'à une certaine profondeur. Chaque coup joué par la couleur du joueur rapportait un certain nombre de points. Le tour suivant (celui de l'adversaire) nous soustrayons le nombre de point équivalent au meilleur coup qu'il peut faire. Tous cela remontait dans la fonction d'appel qui itérait sur chaque possibilités pour une couleur et retournait ainsi le meilleur coup à jouer en fonctions des scores retournés.

Sachant que la seule façon de gagner des points était de manger une pièce, il manquait quelques cases à notre magnifique intelligence artificielle ! Mais après tout, l'objectif n'était pas de créer une IA parfaite du premier coup ! Fort de cette expérience, nous nous sommes penchés sur comment la rendre plus pertinente.

4.4.2 Les fonctions d'évaluation

Il y a plusieurs facteurs importants pour juger une situation aux échecs bien que cela se résume assez facilement :

- Les menaces et les défenses : globalement nous allons calculer pour chaque pièce combien et quel type de pièce elle défend et attaque. Mais nous avons vite réalisé que nous ne pouvons pas simplement faire une somme brute de la valeur des pièces. Cela entraîne des mouvements absurdes de la part de l'IA tel que des attaques suicidaires ou une attitude très passive. Nous n'avons donc pas encore trouvé les bons facteurs pour pondérer la défense et l'attaque mais nous y travaillons sérieusement.
- La position : rien de très impressionnant dans le principe. Seulement nous allons l'exploiter de deux manières différentes. La première sera utilisé par l'IA, le fonctionnement est simple : une pièce au centre du plateau impose une plus grande influence que sur les bords. De ce fait l'échiquier est composé de 4 cercles concentriques.
L'autre méthode que nous utilisons dans la fonction d'évaluation de MinMax

est constituée de plusieurs matrices, une pour chaque type de pièce. Ces matrices indiquent des valeurs pour chaque case de l'échiquier, donc l'IA sera poussé à jouer chaque pièce vers une position plus favorable. Ces valeurs ne dépassent pas 5 (que ce soit en positif ou en négatif). De ce fait, la valeur de la position aura plus d'impact en début de parti lorsque le jeu n'est pas encore ouvert et qu'il n'est pas possible de menacer l'adversaire. Et cela permet de ne pas trop impacter la décision de manger ou non une pièce ennemi car la plus petite valeur, celle du pion, est de 10.

4.4.3 Mise en place de l'algorithme

Le principe de MinMax est assez basique et nous avons essayé de l'appliquer tel quel. J'emploie le mot essayer car ça n'a pas été aussi simple que prévu de le mettre en place! En effet, même si sur le papier il n'y a rien de plus simple : parcourir tous les coups possibles, réitérer sur un coup de l'adversaire si nous ne sommes pas à la profondeur souhaitée sinon évaluer.

Ce n'est pas l'itération sur chacun des coups qui fût compliqué car nous avons déjà de multiples fonctions qui nous facilitaient la tâche. Ce n'est pas non plus la remontée des coups et le fait de revenir dans un état antérieur du jeu qui nous a posé des problèmes.

En fait ce qui nous a posé quelques soucis c'est l'utilisation de nos fonctions d'évaluations. Ces dernières n'avaient aucun problème, mais c'est la façon dont nous gérons le score qui était mauvaise. Et il nous a fallu beaucoup de temps avant de comprendre cela!

Imaginez, vous testez enfin votre MinMax fraîchement codé avec vos nouvelles fonctions d'évaluations. Vous essayez à une profondeur de 1 et tout se passe bien, ce qui est en soit logique sinon ce serait directement les fonctions d'évaluations qui poseraient problème.

Fort de ce premier test, vous décidez d'essayer votre algorithme à une profondeur de 2, et il fonctionne! C'est bien le meilleur coup qui est joué!

Vous vous sentez inarrêtable et décidez d'essayer avec une profondeur de 3! D'un coup toute votre joie retombe, votre IA s'est mise en tête de jouer les pires coups possibles. Vous observez, d'un regard vide de sens, les tours noire et blanche faire des aller-retour entre les fou et le bord de l'échiquier.

Dans un élan de désespoir vous décidez de tester avec une profondeur de 4 en priant pour un résultat différent du précédent. Et c'est sans grands étonnements que vous faites face au ballet des deux tours.

Après plusieurs heures réparties sur deux journées, car à force de nous casser le crâne sur le pourquoi du comment durant des heures, il nous fallait prendre du recul. Mais nous avons fini par résoudre nos problèmes et notre algorithme MinMax flambant neuf, et ayant flambé nos neurones, vit le jour !

4.5 L'algorithme Alpha-Bêta ou l'enfer de l'élagage

4.5.1 La structure Tree

La structure Tree permet de simuler le comportement des arbres généraux et ainsi stocker dans le mémoire les résultats des évaluations précédentes. De cette façon nous allons limiter le nombre de calculs d'une fois sur l'autre en ne recalculant pas ce qui a déjà été fait. Chaque structure représente un coup, nous stockons donc la position où le coup mène ainsi que la position initiale de la pièce bougée. Il nous n'est pas utile de stocker quelle pièce bouge car cela est fait dans le parcours de la fonction principale. Nous gardons également dans chaque structure le fils et le frère de chaque arbre. C'est donc une forme de liste chaînée avec deux sorties. Pour calculer une suite de coup à partir du second tour, il suffira de :

- Sélectionner quel coup a jouer l'adversaire
- Regarder quel coup a été le mieux évalué parmi les suivants de celui-ci
- Libérer la mémoire de tous les autres coups qui ne sont plus utiles.
- Pour chaque feuille de l'arbre, calculer les nouvelles feuilles.
- Retourner le maximum.

```
typedef struct Tree
{
    struct Tree* child;
    struct Tree* sibling;
    int score;
    int sum;
    double max;
    int pos;
    int old_pos;
}Tree;
```

Comme vous avez sûrement pu le remarquer, il y a également deux valeurs qui sont stockées en plus du max qui sont le score et la somme. Le score est la valeur de la pièce mangée avec le déplacement, s'il n'y en a pas la valeur est 0. Bien évidemment, si le coup actuel est celui d'un adversaire, la valeur de la pièce mangée est soustraite au score et on obtient ainsi un score négatif. La valeur 'sum' est l'addition du score actuel du noeud avec le score de tous ses ancêtres. Cela nous sera utile pour l'élagage de l'arbre.

4.5.2 Conception et Mise en place

Certes alpha-bêta est une optimisation de minmax, mais nous voulions garder minmax tel quel, l'objectif étant de montrer l'évolution de l'IA via l'optimisation et observer les différences qu'elles auraient en plus du temps de calcul. C'est pour cela qu'au lieu d'intégrer notre structure Tree dans Minmax nous avons décidé de reconstruire et repenser minmax autour de cette structure.

Bon nous n'allons pas nous le cacher, on a grandement recopié ce que nous avons déjà fait. En même temps pourquoi s'embêter à tout recréer !

Au début nous avons deux valeurs dans la structure indiquant le score : score et max. Mais comme vous pouvez le constater dans la partie précédente, il y a une valeur en plus et nous allons voir pourquoi.

Le principe de ces deux valeurs n'est pas bien compliqué. Nous voyons les choses de la manière suivante : la valeur score prendra la valeur du score du père plus celle de la pièce mangée lors du coup, elle prend la valeur 0 dans le cas où elle ne mange aucune pièce, et la valeur max est la valeur de l'évaluation en feuille qui va remonter jusqu'à la racine.

Cette version est presque intéressante. Nous avons testé Alpha-Bêta en faisant jouer deux IA Alpha-Bêta qui possédaient deux arbres différents. Tout cela pour dire que sur le premier coup de chaque IA, alpha-bêta marchait du tonnerre ! Mais après ces deux premiers coup, l'algorithme commençait à avoir un score délirant.

En fait le problème n'est pas bien complexe, lorsque qu'on itérait une nouvelle fois avec Alpha-Bêta, la valeur score des noeuds était erronée. Étant donné que le score est le cumul du score du mouvement et de celui de tous ses ancêtres, il est normal d'avoir des valeurs qui deviennent très grandes ! Mais ce n'est pas ce qu'on veut. De ce fait, il nous faut réinitialiser la valeur score sur la valeur de la pièce mangée. Or pour cela il nous faut recalculer le coup, et cela est hors de question !

C'est pour cela que nous avons créé la valeur sum qui est en faite l'ancienne valeur score, donc le score de la pièce mangé plus le score du parent. Et la valeur score devient uniquement la valeur de la pièce mangé. Comme cela lors des prochaines itérations, nous n'aurons pas besoin de recalculer entièrement le coup, mais simplement de faire une addition.

Si je raconte tout ça, ce n'est pas entièrement pour rien ! L'élagage de l'arbre s'applique à la descente et nous nous servons du score et plus précisément de la valeur

sum pour définir quand est-ce que nous arrêtons d'explorer un chemin. Cette condition est appliquée avant de faire un nouvel appel récursif de la fonction. Cette valeur d'élagage est définie est tant que valeur global, ce qui nous permet de plus facilement faire des modifications.

Parce qu'en effet, pour trouver une bonne valeur d'élagage il nous faut faire des tests! Au final nous avons décidé de définir la valeur comme légèrement inférieur à moins la valeur du fou.

Il nous reste maintenant qu'à mettre en place le test final pour notre algorithme, nous allons l'affronter! Après création de la nouvelle fonction `human_vs_IA`, nous avons pu lancer le test. Malheureusement pour l'IA, nous sommes deux cerveaux contre elle! Et malheureusement pour nous, cela n'a pas suffi... Ce sentiment d'amertume et de fierté est assez particulier!

4.6 Ajustement de MinMax et d'Alpha-Bêta

4.6.1 Mouvements complexes

Lors de notre recherche de bug, en poussant notre IA vers des chemins qu'elle ne pourrait pas supporter, nous nous sommes rendu compte que cette dernière faisait des mouvements étranges.

En faite, cela était dû au petit et grand rock. Lorsque nous restituions les différents coups joués par Minmax ou Alpha-Bêta, nous appliquions un déplacement inverse. La pièce revient sur à sa position précédente et la pièce mangée, s'il y en a une, est remise là où elle était. Seulement lors d'un rock il y a un déplacement sous-jacent, celui d'une des tours. Or ce déplacement n'est pas restitué. Nous nous retrouvons donc avec des déplacements fantômes ou résiduels, appelez les comme vous le souhaitez, ce qui est sûr c'est qu'il sont gênant!

Seulement nous n'avons pas d'autres moyens efficaces pour effectuer ce déplacement. Nous avons donc décidé de retirer la possibilité à l'IA de faire un rock. Pour cela il nous a suffi de rajouter un paramètre à la fonction qui nous donne les différents mouvements possibles d'une pièce.

4.6.2 Modification de la structure Tree

Mais nous ne voulons pas retirer cette possibilité à l'humain qui joue contre l'IA ! Et dans le cas d'Alpha-Bêta, cela risque de poser un problème.

Actuellement, lorsque l'humain joue, nous itérons sur tous les noeuds du premier niveau de notre arbre et nous recherchons le coup qui vient d'être fait par le joueur.

Or, dorénavant, si le joueur choisit de faire un rock, le coup n'est plus parmi les noeuds de l'arbre.

Le problème est que n'ayant pas de noeud correspondant, notre programme de recherche va tout simplement avoir une erreur car il va essayer d'accéder à la suite de l'arbre alors qu'il l'a déjà parcourus en entier.

La solution n'est vraiment pas compliqué. Si lors de notre parcours de l'arbre nous arrivons à la fin sans rien trouver, nous libérons la mémoire utilisé par l'arbre et ne gardons que la sentinelle. De ce fait, nous allons recréer un nouvelle arbre lors de notre prochaine utilisation d'Alpha-Bêta.

4.7 Online

4.7.1 Le serveur

L'objectif avec ce service online est de jouer avec un ami sur un même réseau. Au début nous avons imaginé un système de hub où il aurait été possible de mettre en relation plusieurs joueurs avec la possibilité de choisir avec qui jouer. Seulement cela nous a vite semblé trop complexe.

Nous avons donc choisi de faire un serveur simple qui ne pourrait host qu'une personne qui pourrait rejoindre le serveur à partir de son adresse IP locale.

Nous avons donc mis en place les connaissances acquises lors de nos cours et de nos travaux pratiques. Dans un premier temps, le serveur était très basique, il nous a d'abord fallu tester les différents moyens de communications possibles.

4.7.2 Le client

A l'origine nous voulions avoir un serveur auquel nous nous connecterions via un terminal de manière basique. Seulement cela impliquait que la personne faisant office de serveur devrait faire tous les calculs et devrait envoyer l'équivalent de l'affichage du plateau de jeu à chaque coup. Cela nous a semblé assez pénible.

Nous avons donc opté pour une autre solution, celle de faire un client.

Nous avons donc réalisé un client basique qui essaye de se connecter au serveur via l'adresse et le port passés en paramètre.

4.7.3 La communication c'est la clef!

Nous avons testé plusieurs façons de communiquer entre le serveur et le client.

Dans un premier temps, nous avons créé un jeu de fonctions qui permettrait l'envoi et la réception de données. Nous avons la fonction `send_move()` qui permettait d'envoyer les informations relatives à un déplacement qui venait d'être effectué. La fonction allant de paire avec cette dernière est `read_apply_move()` qui permet de lire l'information et d'appliquer le mouvement relatif.

Nous avons également la fonction `send_save()` qui permet d'envoyer une sauvegarde et la fonction `send_disconnection()` qui envoie l'information qu'un joueur s'est déconnecté.

Après quelques recherches, nous avons trouvé deux fonctions nous simplifiant la vie ainsi qu'un moyen bien plus facile de transférer des informations.

Nous utilisons donc les fonctions `send()` et `recv()` pour ce qui est de la partie directement en jeu.

Pour la partie chargement et envoi de la sauvegarde, nous utilisons des fonctions que nous avons implémentées. `send_save()` permet l'envoi du contenu du fichier sauvegarder, action qui est faite du serveur vers le client. La fonction `load_from_str()` permet d'initialiser un plateau de jeu à partir de caractères.

4.7.4 Le jeu : options propre à la version en ligne

La couleur ainsi que la partie chargée sont définies par l'hôte de la partie.

Durant un tour il y a deux possibilités, contrairement à la version local où les joueurs prennent le clavier à tour de rôle.

La première possibilité : c'est à votre tour de jouer. Vous avez donc plusieurs possibilités : jouer un coup, sauvegarder la partie, demander l'aide de l'IA et quitter la partie.

La seconde possibilité est que ce n'est pas votre tour de jouer, vous aller avoir un rôle passif durant ce tour. Vous devrez attendre que votre adversaire effectue une action : déplacer une pièce ou quitter le jeu. En attendant une réponse, aucune action de votre part n'est possible.

4.8 Le réseau de neurones

4.8.1 Réflexion et projet sur le réseau de neurones

Durant la première partie de projet, l'un de nos nombreux objectifs était de réaliser des recherches sur la manière d'implémenter un réseau de neurones pour notre jeu d'échecs. Nous nous étions basés sur deux algorithmes d'apprentissage : l'apprentissage par renforcement et l'apprentissage génétique.

Seulement nous nous sommes rendu compte au fur et à mesure de nos recherches qu'implémenter un réseau de neurones pour un jeu d'échecs est extrêmement complexe. En effet, il nous faudrait alors un nombre variable d'input et donc de neurones dans chaque layer. Or tout le principe d'un réseau de neurones est de calculer des poids et des biais pour chacun des neurones, ce qui est impossible si leur nombre varie ! Nous avons beau retourner le problème dans tous les sens, nous n'arrivions pas à trouver quelque chose qui nous semblait réalisable.

C'est alors que l'un de nous a eu une idée. Et si nous faisons un réseau de neurones qui prendrait en entrée des paramètres tels que les coefficients de nos fonctions d'évaluation et les valeurs des pièces ? L'objectif serait alors différent. Nous ne chercherions plus un réseau de neurones capable d'apprendre à jouer, mais un réseau de neurones capable de nous fournir la meilleure fonction d'évaluation possible.

L'idée serait alors d'utiliser nos fonctions d'évaluations comme pour notre algorithme MinMax, mais sans parcourir en profondeur les différents coups. Le réseau de neurones nous donnerait le meilleur coup à jouer dans l'état actuel du jeu.

Avec le schéma de notre réseau de neurones en tête, nous nous sommes penchés sur la manière dont nous allons l'entraîner. Après quelques réflexions, nous avons consenti qu'un entraînement génétique serait plus intéressant qu'un entraînement par renforcement pour notre réseau. Et il est vrai que l'idée de faire un tournoi d'IA et de les voir se battre entre elles était quand même plus drôle !

Donc voilà notre objectif fixé : Créer un réseau de neurones génétique qui devra trouver les paramètres optimaux pour nos fonctions d'évaluations.

4.8.2 Structure du réseau de neurones

Dans un premier temps, nous voulions commencé par développer un réseau de neurones qui apprendrait une fonction peu complexe comparé à un jeu d'échec. L'objectif était donc de faire apprendre la fonction XOR à notre réseau de neurones pas un apprentissage génétique.

Pour réaliser notre réseau de neurones, nous avons implémenté plusieurs structures imbriquées les unes dans les autres. Ces structures représentent les différents éléments des réseaux de neurones. Il y a deux structures supplémentaires qui sont spécifiques à l'entraînement génétique.

Voici la structure qui représente les neurones :

```
typedef struct neurone
{
    float *weights;
    float bias;
    float value;
    size_t size;
}neurone;
```

Les neurones prennent un tableau des poids de la couche de neurones précédente ainsi que la taille ce tableau. Cette structure a aussi comme attribut le biais ainsi que la valeur de ce neurone.

Voici la structure des couches de neurones :

```
typedef struct layer
{
    neurone *neurones;
    size_t size;
}layer;
```

Cette structure prend un tableau de neurones et la taille de ce tableau.

Voici la structure du réseau de neurones :

```
typedef struct network
{
    layer *layers;
    size_t *sizes;
    size_t nb_layer;
}network;
```

La structure est composée d'un tableau de couches de neurones, du nombre de couches et d'un tableau des tailles de chaque couches.

Voici la structure qui va permettre de propager en avant et stocker le score de ce réseau de neurones :

```
typedef struct bot
{
    network *net;
    float score;
}bot;
```

Dans cette structure, il y a le réseau de neurones à utiliser et son score

Ci-dessous vous trouverez la structure qui représente une génération de réseau de neurones :

```
typedef struct generation
{
    bot *bots;
    size_t size;
}generation;
```

Cette structure contient un tableau de bots qui seront triés suivant leurs scores ainsi que la taille de ce tableau.

4.8.3 Principe général

L'idée étant de créer une population de réseaux qui vont exécuter leur tâche, et de les catégoriser en fonction de leur compétences par la suite.

En fonction du score du réseau, un traitement différent va lui être appliqué.

Le score est calculé suivant que le résultat obtenu soit le bon et suivant que les valeurs de sortie soient proches des résultats espérés. Après de nombreux tests qui récompensaient différemment la réussite, nous avons réussi à trouver une fonction de score qui convenait.

Notre but étant d'entraîner toute la population pour qu'elle devienne bonne à ce qu'elle fait, nous allons la faire muter. S'il est mauvais dans ce qu'il essaie de faire, nous le jetons à la poubelle et le remplaçons par un modèle neuf potentiellement plus performant.

Si un réseau n'est pas irrécupérable on va lui conseiller de s'inspirer des meilleurs, et même les meilleurs n'ont pas forcément toutes les qualités possibles, on a toujours à apprendre de plus petits que soi. Tout le monde va donc tenter d'apprendre des autres.

Plus concrètement, sur une génération de taille quelconque, on choisit arbitrairement d'éliminer les 100 derniers réseaux et d'essayer de les recréer aléatoirement.

Les 100 suivants sont remplacés par les meilleurs réseaux et sont aléatoirement modifiés en fonction des résultats des meilleurs réseaux (eux même finalement). Tout le reste des réseaux sont aléatoirement modifiés en fonction des 20 meilleurs réseaux, et sont ensuite mutés.

Ceci représente la mutation d'une population entière, cette mutation a lieu à chaque fois que nous voulons entraîner cette même population. L'entraînement correspond à l'obtention d'un score pour chaque réseau, au tri des réseaux par qualité du score, ensuite de la mutation de chaque réseau. Pour obtenir un réseau suffisamment performant il faudra entraîner la population peut-être un millier de fois voire plus.

4.8.4 Le fonctionnement

Un réseau de neurones est composé de différentes couches, on notera les couches intermédiaires, ainsi qu'une couche d'entrée et une couche de sortie. Chaque couche est composée d'un nombre potentiellement différent de neurones.

Chaque neurone est également composé d'un biais d'une valeur, et de poids. Ces variables vont servir au calcul des valeurs de sorties du réseau.

En partant de la couche d'entrée nous allons parcourir le réseau de couche en couche en direction de la couche de sortie. Chaque passage d'une couche à une autre passe par un calcul à l'aide d'une fonction d'activation.

Le calcul est le suivant, nous attribuerons à chaque neurone de la couche suivante une nouvelle valeur égale à la somme des poids et du biais (du neurone) passés à la fonction d'activation. Et ce jusqu'à la couche de sortie.

4.8.5 Les outils nécessaires

Nous avons aussi développé des outils pour faciliter l'usage du réseau de neurones. Nous avons créé des fonctions qui servent à enregistrer et charger les réseaux de neurones et une fonction pour afficher les résultats d'un réseau de neurones.

Les fonctions de sauvegarde et de chargement écrivent et lisent un fichier suivant un modèle particulier. Pour chaque neurones, on écrit ou lit la taille de la liste des poids contenus dans la structure neurones et la liste entre crochets. Ensuite pour chaque couches, on écrit ou lit la taille de la couche, puis on utilise la fonction précédente pour chacun des neurones de la couche. Enfin pour le réseau de neurones, on écrit ou lit le nombre de couches puis on utilise la fonction précédente.

La fonction d'affichage nous permet de visualiser facilement les résultats des réseaux de neurones. Cette fonction met les les entrées possibles de la fonction XOR dans les neurones d'entrées, propage vers l'avant, puis récupère le résultat dans les neurones de sorties. On affiche ces résultats avec la valeur des neurones de sorties. On compare le résultat obtenu et le résultat espéré, si ces valeurs sont égales on affiche un OK en vert, sinon un KO en rouge.

4.8.6 Optimisation

Avant de s'attaquer à l'adaptation du réseau de neurones au jeu d'échecs, nous avons voulu rendre l'entraînement de notre réseau beaucoup plus optimisé, car nous savons que jouer une partie est très long, et que nous voulons tirer un maximum de chaque partie.

Pour se faire nous avons repensé la manière de récompenser un bon réseau par rapport à un mauvais, en fonction des résultats attendus. Nous avons trouvé une fonction d'activation plus adaptée au problème, après plusieurs essais nous avons choisi la fonction Leaky ReLU :

$$\text{Leaky ReLU} : f(x) = \begin{cases} \text{si } x \leq 0, 0.01x \\ \text{si } x > 0, x \end{cases}$$

Également nous avons revu l'algorithme de mutation d'une génération, nous prenons plus en compte les réseaux qui pourraient avoir une stratégie intéressante mais qui sont mal classés.

Après ces quelques optimisations nous avons réussi à diviser le nombre de réseaux par génération par 10 et le nombre d'entraînement par 10 pour avoir de meilleurs résultats. Après avoir obtenu des résultats sur la porte XOR plus satisfaisants, nous avons adapté le réseau au jeu d'échecs.

4.8.7 L'adaptation au problème

Comme il a été décidé préalablement, qu'il serait trop complexe de faire jouer le réseau de neurones aux échecs. Nous avons préféré faire fonctionner notre algorithme Alpha-Bêta, et évaluer si ses coups sont bons ou non par apprentissage génétique.

C'est pourquoi pour réaliser notre intelligence artificielle, nous nous basons sur le principe de l'algorithme Alpha-Bêta. Notre réseau de neurones sera une fonction d'évaluation qui apprendra à mieux jouer au fur et à mesure.

Le réseau de neurones prendra en entrée le plateau de jeu avec un chiffre suivant l'état de la case. On a deux couches cachées de 32 neurones. En sortie nous retournons un score qui évalue si la situation est bonne ou non. Cependant pour l'attribution des scores nous n'avons pas de bases de données pour entraîner notre réseau.

L'entraînement est cependant assez long puisque nous devons faire jouer une partie pour attribuer un score au réseau. Or pour entraîner notre réseau de neurones par algorithme génétique il faut beaucoup d'entraînements pour avoir un réseau de neurones performant. C'est pour cela que nous avons fait du multi-threading pour accélérer notre entraînement.

Toutefois, l'entraînement est toujours assez long puisqu'il va falloir jouer un grand nombre de partie pour avoir des résultats concluants.

4.9 Dérroulement du jeu

L'une des choses qui se veut essentielle dans notre jeu. En effet puisque nous nous sommes détachés d'une interface graphique classique, il nous est nécessaire d'avoir un contact avec le joueur confortable. Pour cela nous avons voulu diminuer un maximum le nombre d'entrées clavier. Nous avons alors fait le choix de poser les questions en proposant toujours deux choix et pour choisir l'individu n'aura qu'à taper 0 ou 1. Il peut également choisir de quitter l'application à tout moment en entrant "q".

Il y a plusieurs paramètres à récupérer pour choisir le type de partie.

- En ligne ou hors ligne ? Cela va changer totalement la suite des demandes car si le joueur veut faire une partie en ligne il faudra le connecter, lui trouver son joueur, etc.
- Contre un joueur local ou une intelligence artificielle ? Le but de ce choix est assez évident. Par joueur local nous entendons deux joueurs côte à côte avec un seul et même clavier.
- Contre une intelligence artificielle algorithmique ou contre un réseau de neurone ? Par algorithmique nous sous-entendons alpha-beta. Puisque nous avons créer deux types d'intelligences artificielles ; nous préférons laisser le choix au joueur entre celles-ci.
- Nouvelle partie ou charger une sauvegarde ? Si le joueur désire charger une partie nous lui affichons toutes les sauvegardes disponibles dans le sous-dossier "save/". Puis il devra écrire en toute lettre le nom du fichier. Pour parler un petit peu de la technique, nous avons utilisé les connaissances des tp de programmation pour afficher les sauvegardes. Nous avons créer un nouveau processus à l'aide de la commande `fork()`, puis l'avons écrasé avec le processus de "ls" en utilisant `execvp()`.
- Noir ou blanc ? Cette question ne sera posé en ligne que pour l'hébergeur de la partie car évidemment les deux joueurs ne peuvent pas choisir la même partie.

```
[epita@archlinux test_chess]$ ./game.out
Do you want to play online (0) or offline (1) ? (q to quit) :
1
Do you want to play against a local friend (0) or an IA (1)? (q to quit) :
1
Do you want to play against a neural network (0) or an algorithmique IA (1) ? (q to quit) :
1
Do you want create a new game (0) or load a save (1) ? (q to quit) :
0
Do you want to play with blacks (0) or whites (1) ? (q to quit) :
0
```

4.10 Travaux inachevés

Comme dit précédemment concernant le réseau de neurones. Il est très long de lui apprendre à devenir ne serait-ce qu'un peu meilleur aux échecs, car le système de sélection naturelle repose sur le fait que la chance donne des attributs génétiques intéressant pour telle ou telle tâche. Dans le cas d'un exercice compliqué comme une partie d'échecs, cela peut prendre énormément de temps d'avoir des résultats concluants.

C'est pourquoi un objectif non atteint, aurait été de prendre le temps d'éduquer notre réseau. Pour ce faire, il aurait fallu entraîner notre réseau sur plusieurs machines en réseau et les faire jouer les unes contre les autres, le tout toujours en multi-threading. De manière à faire énormément de parties et de multiplier nos chances d'avoir de bons réseaux (Pourquoi ne pas réquisitionner les salles machines de Villejuif).

Vous l'avez sûrement constaté, mais notre partie ne ligne n'est pas en paire à paire. Cela s'explique que par le fait que lors de nos recherches, nous nous sommes rendu compte que la technologie paire à paire n'était pas la plus intéressante pour notre projet.

Il n'est pas nécessaire que les deux joueurs soient à la fois serveur et client, cela pourrait même créer des conflits, tel que le chargement de la partie par exemple. Un système client-serveur est bien plus intéressant dans notre cas, nous ne sommes pas obligé d'avoir un flux de donnée continue. Il nous suffit simplement de transmettre les informations du joueur qui est en train de jouer à celui qui est en train d'attendre son tour.

Chapitre 5

Conclusion

5.1 Expérience individuelle

5.1.1 Alan

Lors de la réalisation de ce projet, nous avons tous travaillé sur des choses assez différentes. Au départ, après que l'on ait mis en place la structure du projet, je me suis attaqué avec mes collègues à la réalisation du jeu d'échec. Dans un premier temps je me suis surtout intéressé à l'affichage de notre jeu. Puis, j'ai rejoint mes collaborateurs sur l'implémentation du jeu en lui même. Enfin, j'ai réalisé la structure principale du site internet demandé.

Après la première soutenance, avec Théo, nous avons implémenté les fonctions qui spécifient les règles du jeu manquantes. Après plusieurs problèmes nous avons réussi à traiter de nombreux cas possibles.

Ensuite, nous nous sommes attaqués au réseau de neurones avec Loïc. Nous avons d'abord réalisé les différentes structures, puis nous avons fait les fonctions nécessaires qui utilisent les structures. Sur cette partie nous n'avons pas eu de problèmes mais c'est lorsque nous avons voulu faire fonctionner le réseau de neurones que ce fut plus compliqué. Après plusieurs essais et après avoir corrigé des erreurs d'allocations de mémoires avec l'aide de Théo, nous avons réussi à avoir un réseau de neurones qui fonctionnait avec un entraînement par générations. Ensuite avec Théo nous avons cherché à améliorer notre entraînement de génération et nous avons réussi à avoir des résultats plutôt concluant.

Enfin, lors de cette troisième période nous nous sommes surtout concentré sur le réseau de neurone avec Loïc. Nous avons d'abord essayé d'optimiser un maximum l'entraînement du réseau de neurone par algorithme génétique pour la fonction XOR

dans le but de mieux comprendre ce qui rend cette entraînement efficace. Nous avons réussi à bien optimiser notre entraînement puisque nous avons réussi à avoir de meilleurs résultat avec moins de générations et moins d'entraînements. Ensuite nous avons adapté notre réseau de neurones au problème que nous nous sommes fixé. Sur cette partie nous n'avons pas eu de problèmes particuliers même si cela nous a pris pas mal de temps.

J'ai trouvé ce projet très intéressant à réaliser en groupe. Il est cependant dommage que nous n'ayons pas pu assez entraîner un réseau de neurones pour qu'il ait un niveau correct. J'ai beaucoup apprécié travailler avec ce groupe qui est resté très sérieux et travailleur.

5.1.2 Loïc

Voici l'aventure qui touche à sa fin, la route a été longue, concernant mon expérience au sein de ce groupe, elle fut la plus homogène de toute. Pour ma part j'ai participé comme tout le monde déjà à l'élaboration structurelle du projet (architecture, makefile), ainsi qu'à son organisation (github, trello), ensuite à l'implémentation du moteur du jeu d'échecs.

Très vite après la "finalisation" du moteur de jeu, nous nous sommes dispersés, avec Alan nous avons pris la route du réseau de neurones, une route semée d'embûches. J'ai trouvé la programmation du réseau de neurones peu intéressante, autant la théorie l'est, autant sa programmation est barbante sans être compliquée, pourtant il faut être concentré en permanence car ce n'est pas une structure aisée à debug, il faut connaître la théorie sur le bout des doigts pour déceler d'où peut provenir un éventuel problème.

Après réussite de l'implémentation du réseau de neurones pour la porte XOR en tant que preuve de concept, il a fallu optimiser le processus pour pouvoir l'adapter à notre jeu d'échecs, une fois l'optimisation validée, on a pu transposer notre concept à notre projet, et c'est là que l'on constate les limites de l'évolution de notre projet dans le cadre de l'école. C'est un peu frustrant de se rendre compte que l'algorithme génétique (que nous avons choisi parmi tant d'autres) ne peut pas fonctionner à notre échelle (un seul ordinateur), mais c'est en se frottant à des problèmes que nous trouverons des solutions dans le futur.

5.1.3 Félix

Le chemin traverser en compagnie de notre groupe a encore été un grand plaisir. Nous avons tous donné de l'énergie pour avancer et notre organisation s'est améliorée jusqu'à la fin.

Au cours de ce semestre nous avons d'abord passer du temps à nous documenter, lire des forums et papiers scientifiques, puis nous avons commencer à coder. Sans jeu rien n'étant possible, nous avons avancé à 4, ensemble. Puis j'ai travaillé avec Théo presque l'ensemble du semestre. D'abord sur l'algorithme MinMax, transformé ensuite en Alpha-Beta, puis sur le réseau en ligne, et enfin sur l'interface utilisateur du lancement du jeu. J'ai également été responsable, seul cette fois, de la gestion des sauvegardes et chargements des plateaux de jeu.

Nous avons été surpris avec Théo par la difficulté de corriger les erreurs dans les structures récursives du C et il nous a parfois fallu prendre plusieurs heures pour trouver la petite condition qui cafouillait. Mais nous y sommes arrivé et tous les bugs ont cédé devant notre persévérance.

Pour la gestion du réseau ; encore une fois la difficulté n'était pas dans le code directement mais dans comment organiser celui-ci. Une fois que la structure était claire dans nos tête elle le fut tout autant dans notre code.

J'ai passé un excellent moment avec mon groupe cette année et je suis sûr que les compétences acquises ici me servirons un jour. Si ce projet était à refaire, je le ferais de la même façon.

5.1.4 Théo

Durant ce projet, j'ai un peu touché à tout. Dans un premier temps, alors que nous nous étions encore en train de débattre et d'ajuster la structure Game, j'ai entrepris de faire le parsing de l'input. Par la suite je me suis attelé à aider mes camarades sur certaines fonctions. Après j'ai commencé avec Félix à travailler sur l'algorithme Min-max, c'est ce qui nous a pris le plus de temps pour la première soutenance, et le plus d'énergie à mon avis. Vers la fin de cette première partie, j'ai également travaillé sur certaines règles du jeu d'échec.

Durant la deuxième partie, j'ai été présent sur tous les fronts. J'ai d'abord commencé, avec Alan, à finir les règles du jeu. Et comme vous l'avez vu avec king_suicide, ça n'a pas forcément été de tout repos !

Ensuite j'ai travaillé avec Félix sur l'algorithme MinMax et, une fois terminé, nous avons enchaîné avec Alpha-Bêta.

Je n'étais pas censé travailler sur le réseau de neurones durant ce projet, mais le groupe avait des difficultés avec le réseau de neurones, donc je suis allé les aider. Avec l'aide d'Alan, j'ai commencé par modifier la manière dont on initialisait les générations ainsi que la manière dont on libérait l'espace mémoire, avec cela j'ai réussi à régler tous les pertes de mémoire. J'ai ensuite réfléchi avec Alan sur la manière dont on entraînait notre génération, et nous avons fait deux trois ajustements pour que le réseau fonctionne.

Durant la dernière partie du projet, je me suis chargé, avec Félix, de la partie en ligne du projet. Nous avons décidé d'en faire une en paire à paire, mais après quelques recherches nous nous sommes rendu compte que ce n'était pas très pertinent pour notre projet. Nous avons donc opté pour un client / serveur comme nous avons pu le voir en cours.

Il a également fallu recréer un déroulement du jeu et gérer le transfert de données. Cette partie c'est déroulé sans trop d'accrocs et nous sommes plutôt satisfait du résultat.

Globalement, nous avons réussi à tenir nos objectifs. Le jeu est fonctionnel, nous avons nos 3 intelligences artificielles (plus ou moins intelligente) et nous avons la possibilité de jouer en ligne.

Le groupe a gardé une bonne dynamique durant toute la réalisation du projet. Cela nous a permis de progresser régulièrement sans être pressé par le temps.

Ce projet a été très plaisant et intéressant, j'ai appris beaucoup de choses et cela m'a permis de renforcer mes bases en programmation et en C.

5.2 Conclusion

Pour conclure, ce projet s'est déroulé sans trop d'encombre. Nous avons réussi à réaliser tous les objectifs que nous nous étions fixé, seul le réseau de neurones mériterait un peu plus de temps. Ce projet a été très intéressant et nous avons eu une bonne motivation ce qui nous a permis de bien avancer sur ce projet. La partie sur les différentes intelligences artificielles furent les plus ardues, mais ce sont également celles qui piquaient le plus notre curiosité!

Avec du recul, nous nous rendons compte que certaines parties auraient pu aller plus vite et certaines optimisations auraient pu être réalisées. Mais tout de même, nous avons été battu par notre propre IA! C'est sûrement le fait qui va principalement rester dans nos mémoires.

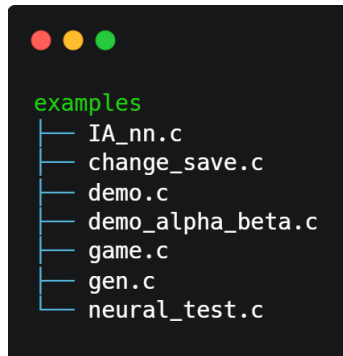
Vous pouvez télécharger notre projet via notre site web :

<https://the-albatross-s-king.github.io/chess-web/>

Chapitre 6

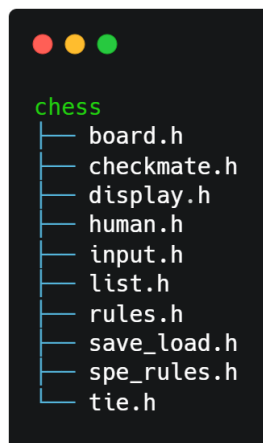
Annexe

Voici le contenu du dossier exemples :



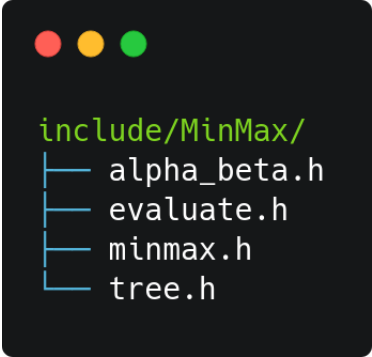
```
examples
├── IA_nn.c
├── change_save.c
├── demo.c
├── demo_alpha_beta.c
├── game.c
├── gen.c
└── neural_test.c
```

Voici les headers de la partie jeu d'échecs :



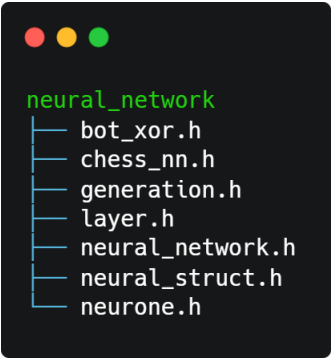
```
chess
├── board.h
├── checkmate.h
├── display.h
├── human.h
├── input.h
├── list.h
├── rules.h
├── save_load.h
├── spe_rules.h
└── tie.h
```

Voici les headers de la partie Min-Max :



```
include/MinMax/  
├── alpha_beta.h  
├── evaluate.h  
├── minmax.h  
└── tree.h
```

Voici les headers de la partie réseau de neurones :



```
neural_network  
├── bot_xor.h  
├── chess_nn.h  
├── generation.h  
├── layer.h  
├── neural_network.h  
├── neural_struct.h  
└── neurone.h
```

Voici les headers de la partie online :



```
online  
├── client.h  
├── online_game.h  
└── single_server.h
```

Voici les fichiers sources de la partie jeu d'échecs :

```
chess
├── board.c
├── checkmate.c
├── display.c
├── human.c
├── input.c
├── list.c
├── rules.c
├── save_load.c
├── spe_rules.c
└── tie.c
```

Voici les fichiers sources de la partie Min-Max :

```
src/MinMax/
├── alpha_beta.c
├── evaluate.c
├── minmax.c
└── tree.c
```

Voici les fichiers sources de la partie réseau de neurones :

```
neural_network
├── bot_xor.c
├── chess_nn.c
├── generation.c
├── layer.c
├── neural_network.c
└── neurone.c
```

Voici les fichiers sources de la partie online :



Voici l'évolution de notre repos git :

Les commits sont listés du plus récent au plus ancien.

```
* - (origin/neural_network, neural_network) <Alan GUERET>
* - <Alan GUERET>
* - <Alan GUERET>
* - <Alan GUERET>
* - <Alan GUERET>
* - <Alan GUERET>
* - <alan gueret>
* - (HEAD -> master, origin/master, origin/HEAD) <alan gueret>
* - <alan gueret>
* - <Loic Segundo>
| \
| * - <alan gueret>
* | - <Loic Segundo>
* | - <Loic Segundo>
| \
| * - <Loic Segundo>
* | - <Loic Segundo>
* | - <Loic Segundo>
* | - <Loic Segundo>
| | * - (origin/online) <Théo Daronat>
| | * - <Felix LENA>
| | * - <Théo Daronat>
| | * - <Théo Daronat>
| | * - <Théo Daronat>
| | * - <Théo Daronat>
| | * - <Felix LENA>
| | * - <Théo Daronat>
| | * - <Théo Daronat>
| | * - (origin/soutenance2) <Théo Daronat>
| | * - <Théo Daronat>
| | * - <Théo Daronat>
| | \
| | /
| | /
| * - <Alan GUERET>
| * - <alan gueret>
| * - <Alan GUERET>
| /
| * - <Théo Daronat>
| /
| * - <alan gueret>
| \
| * - <Théo Daronat>
| \
```

```

| \
| * - (origin/MinMax2) <Felix LENA>
| * - <Théo Daronat>
| * | - <Théo Daronat>
| \ \
| \ \
| //
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| * - <Théo Daronat>
| \ \
| * | - <Théo Daronat>
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <Théo Daronat>
| * | - <alan gueret>
| * | - <Théo Daronat>
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <alan gueret>
| \ \ \ \
| * | | | - <Loic Segundo>
| * | | | - <alan gueret>
| // // //
| * | | - <alan gueret>
| * | | - <alan gueret>
| * | | - <alan gueret>
| * | | - <Loic Segundo>
| * | | - <Loic Segundo>
| * | | - <alan gueret>
| * | | - <Loic Segundo>
| \ \
| * | - <Loic Segundo>
| |

```

```

|/|
* | - <alan gueret>
| \
| * | - <alan gueret>
| * | - <alan gueret>
| \ \
| \ | \
| \ | \
| * | - <Théo Daronat>
| * | - <alan gueret>
| \ \
| \ | \
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <alan gueret>
| * | - <alan gueret>
|/ /
|/ /
* | - <alan gueret>
* | - <alan gueret>
* | - <alan gueret>
* | - <alan gueret>
* | - <Théo Daronat>
* | - <alan gueret>
* | - <alan gueret>
* | - <alan gueret>
* | - <Théo Daronat>
* | - <alan gueret>
* | - <alan gueret>
* | - <alan gueret>
* | - <alan gueret>
* | - <alan gueret>
|/
* | - <alan gueret>
* | - <alan gueret>
* | - (origin/SOUTENANCE1, SOUTENANCE1) <Théo Daronat>
| \
| * | - <Alan GUERET>
| * | - <Théo Daronat>
|/
* | - <Théo Daronat>
* | - <Loïc Segundo>
* | - <Loïc Segundo>
* | - (origin/felix) <Felix LENA>

```



```

* - <Felix LENA>
* - <Felix LENA>
| \
* | - <Felix LENA>
* | - <Felix LENA>
* | - <Théo Daronat>
* | - <Felix LENA>
* | - <Felix LENA>
* | - <Felix LENA>
| \ \
* | | - <Felix LENA>
* | | - <Felix LENA>
* | | - <Felix LENA>
| \ \ \
* | \ \ \ - <Felix LENA>
| \ \ \ \
* | | | | - <Felix LENA>
| | | | * - (origin/gameboard, gameboard) <Alan GUERET>
| | | | /
| | | | * - <Théo Daronat>
| | | | * - <Théo Daronat>
| | | | /
| | | | * - <Alan GUERET>
| | | | * - <Alan GUERET>
| | | | * - <Loic Segundo>
| | | | * - <Loic Segundo>
| | | | * - <Loic Segundo>
| | | | * - (refs/stash) <Alan GUERET>
| | | | \
| | | | * - <Alan GUERET>
| | | | /
| | | | * - <Alan GUERET>
| | | | /
| | | | * - (origin/parsing_input) <Loic Segundo>
| | | | \
| | | | * - <Alan GUERET>
| | | | * - <Alan GUERET>
| | | | /
| | | | /
| | | | * - <Loic Segundo>
| | | | * - <Loic Segundo>
| | | | \
| | | | /
| | | | /
| | | | /

```

```

| | | / /
| | | / /
| * | | - <Loic Segundo>
| / / /
* | | - <Felix LENA>
* | | - <Loic Segundo>
* | | - <Felix LENA>
* | | - <Felix LENA>
* | | - <Felix LENA>
* | | - <Théo Daronat>
| * | | - <Théo Daronat>
| \ \ \
| * | | - <Loic Segundo>
| * | | - <Théo Daronat>
| / / /
| * | | - (origin/makefile) <unknown>
| * | | - <Loic Segundo>
| * | | - <Loic Segundo>
| * | | - <Loic Segundo>
| \ \ \
| / / /
| * | | - <Loic Segundo>
| * | | - <Loic Segundo>
| * | | - <unknown>
| /
| * | | - <Alan GUERET>
| /
| * | | - <Alan GUERET>
| * | | - <Loic Segundo>
| * | | - <Loic Segundo>
| * | | - <Felix LENA>
| * | | - <Loic Segundo>
| * | | - <Felix LENA>
| * | | - <Felix LENA>
| /
| * | | - <Alan GUERET>
| * | | - <alanretgue>

```