




21 MAI 2024

RAPPORT DE PROJET

JEU DU BOBAIL

Pailler Hippolyte et Aliona Lejuste



SOMMAIRE

INTRODUCTION	2
AFFICHAGE	3
DEPLACEMENTS	3
<i>Gestion du « scanf »</i>	3
<i>Déplacement du bobail</i>	3
<i>Deplacement du pion</i>	4
VICTOIRE	4
<i>Ramener le bobail</i>	4
<i>Encercler le bobail</i>	4
MAIN	4
<i>Makefile</i>	4
<i>Main</i>	4
IMPLEMENTATION DES IA	5
<i>IA aléatoire</i>	5
<i>IA optimisée</i>	5
<i>IA MCTS</i>	6
CONCLUSION	8

INTRODUCTION

Dans le cadre de notre projet, nous avons entrepris l'implémentation du jeu du bobail en langage C. Le bobail est un ancien jeu traditionnel africain dans lequel on essaye de ramener le pion central, appelé bobail, dans son camp. Les pions ne peuvent pas se sauter mais servent d'obstacles.

Ce rapport présente une analyse détaillée des fonctionnalités que nous avons pu implémenter, des défis rencontrés tout au long du processus de développement, ainsi que des méthodes et des stratégies déployées pour les surmonter. Nous aborderons également des aspects clés tels que l'organisation du code, la documentation, la gestion du travail en équipe et de la gestion de versions avec Git.

En premier lieu, nous discuterons des fonctionnalités que nous avons réussi à implémenter, en mettant en lumière les aspects les plus pertinents de notre travail. Ensuite, nous explorerons les difficultés rencontrées tout au long du processus de développement, en identifiant les principaux défis techniques et conceptuels et en décrivant les approches que nous avons adoptées pour les résoudre. Nous examinerons également en détail l'organisation de notre code, en mettant en évidence la structure générale du projet et en expliquant la logique sous-jacente à chaque composant.

Par la suite, nous aborderons l'aspect de l'intelligence artificielle (IA) dans notre projet. Initialement, nous avons développé une IA de type aléatoire pour le jeu, puis nous avons utilisé cette IA comme base pour l'implémentation de l'algorithme Monte Carlo Tree Search (MCTS). Nous détaillerons les différentes étapes de cette implémentation et les résultats obtenus.

Nous espérons que ce rapport offrira un aperçu complet de notre travail, mettant en lumière nos réalisations, nos défis et nos stratégies pour les surmonter, ainsi que les leçons que nous avons tirées de cette expérience enrichissante.

AFFICHAGE

Pour l’affichage de notre jeu nous avons souhaité créer une interface simple pour l’utilisateur. Nous avons donc opté pour un système proche de celui de la bataille navale où les pions sont repérés par leurs coordonnées (ligne, colonne). Ainsi pour la gestion code de l’affichage et la gestion de la grille nous avons opté pour un tableau de 5 lignes et 5 colonnes.

Dans le *header* « *case.h* » nous avons défini les couleurs de pions : {Blanc, Bleu, Rouge, Vert}.

Le blanc sert à représenter les cases vides. Le bleu les pions de l’adversaire/ordinateur. Le vert le bobail et le rouge nos pions.

A l’aide de ces types nous avons donc créé notre tableau « *CouleursPions*** » dans lequel nous pouvons naviguer selon une logique ligne colonne.

La première fonction “*afficher*” était des plus simplistes et reposait sur un affichage de B X O pour représenter les différents états. Pour rendre l’affichage plus attrayant dans le terminal nous avons utilisé du langage Unicode pour tracer un vrai tableau avec des ronds de différentes couleurs.

Et ainsi notre affichage était prêt.

DEPLACEMENTS

Les joueurs jouent à tour de rôle. A chaque tour, le joueur déplace d’abord le bobail puis un de ses pions. Le bobail se déplace dans tous les sens d’une seule case à la fois.

Les pions se déplacent également dans toutes les directions mais ils doivent aller au bout de leur déplacement (jusqu’au bord du plateau ou jusqu’à rencontrer un autre pion ou le bobail.)

Nous avons créé deux fonctions spécifiques pour gérer les déplacements du bobail et des pions.

GESTION DU « *SCANF* »

Le premier problème avec « *scanf* » est survenu dans les fonctions déplacements.

En effet, via cette fonction nous demandions la nouvelle position du pion ou du bobail sous le modèle d’une bataille navale. Le format demandé était donc 2 entiers entre 0 et 4 inclus. Vérifier que le nombre entier était compris entre 0 et 4 est plutôt simple mais si le joueur rentrait autre chose qu’un entier le programme rentrait dans une boucle infinie pour finir sur un « *Segmentation fault* ».

Pour résoudre ce problème, nous avons dû comprendre le comportement de « *scanf* ». Lorsque deux entiers sont attendus, « *scanf* » retourne 2. Nous avons donc ajouté une vérification avec une condition if et utilisé une boucle « *while* » pour vider la ligne générée par l’erreur. Dès qu’une entrée incorrecte est détectée, le programme appelle récursivement la fonction pour redemander la position correcte.

DEPLACEMENT DU BOBAIL

Pour le déplacement du bobail, nous avons pris en compte plusieurs cas interdits tels que les limites du plateau, la présence d’un pion à la position demandée par le joueur, et la légalité du coup. Nous avons ajouté de nombreuses conditions « *if* » pour vérifier ces cas. Pour garantir que toutes les erreurs étaient

prises en compte, nous avons testé le jeu plusieurs fois contre notre IA aléatoire, identifiant et corrigeant ainsi les erreurs non prévues initialement.

DEPLACEMENT DU PION

Le déplacement des pions présentait des défis similaires à ceux du bobail, avec des complexités supplémentaires pour vérifier la légalité des coups. Contrairement au bobail, les pions n'ont pas de distance fixe de déplacement ; il fallait donc adapter le code à chaque situation. Nous avons vérifié la présence d'un obstacle immédiatement après la position demandée et nous avons vérifié que la ligne de déplacement était libre (pas de pion sur la trajectoire). Comme pour le bobail, nous avons utilisé des conditions « *if* » pour gérer chaque cas spécifique.

VICTOIRE

Dans ce jeu il existe deux façons de gagner. Le joueur peut soit ramener le bobail sur une case où se trouvaient ses propres pions au début soit encercler le bobail afin que l'adversaire ne puisse pas le déplacer au début de son tour.

RAMENER LE BOBAIL

La difficulté première à prendre en compte est que le joueur 1 peut forcer le joueur 2 à mettre le bobail dans les positions initiales adverses. Pour cette raison, il n'était pas suffisant de simplement utiliser un booléen pour déterminer la victoire sans distinguer le joueur actif et le joueur gagnant. Nous avons donc changé notre fonction « *bool* » en « *CouleurPion*** » pour stocker le joueur gagnant.

ENCERCLER LE BOBAIL

Pour déterminer si le bobail est encerclé il suffit de regarder s'il y a la présence d'une case blanche autour de lui. Il est important de ne pas considérer la limite du plateau comme cases vides. Nous avons néanmoins évité ce problème en n'allouant le type blanc qu'aux cases situées à l'intérieur du plateau. La véritable difficulté a été de trouver le bon moment pour appeler cette fonction dans le *main*. La victoire est attribuée au joueur 1 lorsque le joueur 2 ne peut plus bouger le bobail, donc il est crucial d'appeler cette vérification à la fin du tour du joueur 1, et non au début du tour du joueur 2.

MAIN

MAKEFILE

Le seul problème rencontré avec le *makefile* concernait la ligne « *-Werror -Wall* » qui pouvait empêcher la compilation lors de tests. À part cela, nous n'avons rencontré aucun autre problème avec le « *makefile* ».

MAIN

Pour pouvoir utiliser un *main* avec plusieurs fonctions, nous avons créé un *makefile* et des fichiers d'en-tête (*headers*). La difficulté principale du code résidait dans l'appel des différentes fonctions dans le bon ordre, la gestion des erreurs liées à « *scanf* » et à l'allocation de mémoire avec « *malloc* ». Malgré ces défis, nous n'avons pas été freinés par ces problèmes.

Pour différencier les différents cas de jeu (Humain vs Humain et Machine vs Humain), nous avons utilisé la structure de contrôle « *switch* » et les déclarations « *case* ».

IMPLEMENTATION DES IA

L'implémentation des IA s'est faite en 3 étapes. Nous avons construit 3 niveaux d'intelligence artificielle.

- La première : IA en aléatoire
 - Les coups pions et bobail sont choisis au hasard par la fonction « *rand()* » ;
 - IA de base dont les codes ont été utiles pour construire les deux autres.
- La deuxième : *IA_up*
 - On simule des parties à partir de certaines configurations de départ ;
 - Calcul du taux de victoire pour le choix de la configuration à renvoyer.
- La troisième : MCTS
 - On utilise la méthode de Monte Carlo Tree Search pour trouver les configurations gagnantes.

IA ALEATOIRE

Pour faire fonctionner cette IA il nous fallait deux fonctions : une pour le bobail et la deuxième pour les pions. Les approches étaient cependant assez similaires.

Pour cette version de l'IA on procède par un jeu aléatoire.

Pour les pions :

- Etape 1 : Choix d'un pion en aléatoire entre 1 et 5
- Etape 2 : Vérifier que ce pion peut bouger
- Etape 3 : Choisir un des déplacements en aléatoire
- Etape 4 : Réaliser le mouvement

Les différents choix reposaient sur la fonction « *rand()* ». L'initialisation se faisait au début de partie dans le *main*. Dans la première partie on choisit un pion aléatoire et on recueille ses coordonnées. Puis on vérifie qu'il y a au moins une case vide autour de ce pion, signe qu'il peut bouger. On choisit une de ces cases vides encore au hasard puis on fait progresser le pion jusqu'au premier obstacle en précisant bien dans les conditions le respect des dimensions du tableau et de la nécessité d'une case blanche pour avancer. En cas d'échec, si jamais le pion choisi ne peut pas se déplacer, on relance la fonction.

Pour le bobail :

- Etape 1 : Repérer le bobail
- Etape 2 : Choisir un mouvement au hasard
- Etape 3 : Vérifier sa faisabilité

IA OPTIMISEE

Pour notre deuxième IA, nous avons procédé par un système de maximum de victoires. Le code créé n'est malheureusement pas abouti à cause d'un souci de mémoire qui entraîne un « *Segmentation fault* » à un moment ou un autre dans le jeu. Le code fourni est théoriquement pour le bobail mais la structure même était répliquable quasiment à l'identique pour les pions.

L'idée est qu'à partir de la configuration de jeu initiale on crée une liste de tableau d'une dizaine de cases ou chaque case contient la configuration de jeu après le déplacement aléatoire du bobail. A partir de ces configurations on simule une centaine de parties et on note le taux de victoire. La configuration ayant le taux le plus haut est alors choisie comme le coup à jouer.

Cette IA a nécessité la création d'une fonction « *copy_grille* » car il s'agit de conserver la grille initiale et celles enfants afin de faire les simulations et surtout pour renvoyer un nouvel état où les pions ne se sont pas téléportés à l'autre bout de la grille sans raison.

La fonction « *simu_jeu* » est assez simple et reprend l'idée d'une bataille entre deux IA aléatoires. Pour le calcul du gain on a simplement défini la victoire de l'IA à 1, sa défaite à -1 et le match nul à 0. Ce cas du match nul est une sorte de filet si jamais la victoire prend trop de temps à arriver.

On procède donc ainsi pour les 10 enfants et on prend comme configuration gagnante celle ayant le score le plus élevé.

Le problème rencontré est que l'architecture de cette IA est prête à l'emploi dans le code rendu, cependant la fonction « *simu_jeu* » entraîne des « *Segmentation fault* » que nous n'avons pas réussi à résoudre malgré des tentatives de débogage intensives.

IA MCTS

Pour notre dernière IA nous voulions nous appuyer sur la méthode du Monte Carlo Tree Search. L'algorithme MCTS se décompose en quatre phases principales : la sélection, l'expansion, la simulation et la rétropropagation. Ces phases sont répétées jusqu'à ce qu'un critère d'arrêt soit atteint (par exemple, un nombre fixe d'itérations ou une limite de temps).

- 1. Sélection

Dans cette phase, l'algorithme parcourt l'arbre de recherche à partir de la racine en sélectionnant les nœuds selon une stratégie spécifique, nous avons utilisé l'indicateur UCB1. Celui-ci permet d'équilibrer l'exploration de nœuds prometteurs et l'exploitation de nœuds bien explorés.

- 2. Expansion

Lorsque l'algorithme atteint un nœud non terminal avec des actions non encore explorées, il développe l'arbre en ajoutant un ou plusieurs nouveaux nœuds. Cela permet d'examiner de nouvelles possibilités de jeu.

- 3. Simulation

À partir du nœud nouvellement ajouté, l'algorithme simule aléatoirement une série de coups jusqu'à atteindre une condition terminale (par exemple, la fin de la partie). La simulation aléatoire permet d'estimer la valeur de ce nœud.

- 4. Rétropropagation

Les résultats de la simulation sont ensuite propagés en remontant l'arbre, en ajustant les valeurs des nœuds visités pour refléter les résultats des simulations. Cette phase met à jour les informations sur les nœuds, permettant à l'algorithme de mieux évaluer les coups futurs.

Pour cela nous avons utilisé une structure de liste chaînée pour remonter plus facilement l'arbre. Chaque nœud comporte un état du plateau, le lien de son parent, ceux de ses enfants, le nombre de simulation et le nombre de victoire. Ces deux derniers paramètres sont utiles pour le calcul de l'UCB1 afin de déterminer l'ordre de parcours de l'arbre.

Dans les fonctions utiles que nous avons codées :

- *Crea_node* : permet d'initialiser un nouveau nœud, cette fonction est particulièrement utilisée dans l'étape d'expansion ;
- *Ucb* : pour un nœud donné, calcule la valeur de l'UCB1 associé ;
- *Select_noeud* : à partir d'un parent donné, permet de choisir un enfant selon leur valeur d'UCB1 calculé grâce à la fonction précédente ;
- *Simu_jeu* : simule une partie jusqu'à la victoire de l'IA pour une configuration donnée ;
- *Backpropagate* : permet de remonter les nouveaux scores des enfants vers leurs parents pour déterminer le nœud gagnant final ;
- *Mcts* : gère l'architecture générale en faisant appel aux fonctions précédentes.

Cependant comme précédemment nous sommes tombés sur des problèmes de mémoire qui nous ont empêchés de faire de notre programme une IA performante pour notre jeu.

En réalité nous avons même commencer par cette IA et devant les problèmes rencontrés nous avons décidé de la simplifier donnant naissance à *IA_optimisée* afin de mieux évaluer d'où provenait l'erreur.

CONCLUSION

Ce projet nous a offert une occasion de mettre en œuvre un jeu traditionnel africain, le bobail, en langage C, ainsi que de développer des compétences en intelligence artificielle et en gestion de projets logiciels. Nous avons réussi à implémenter plusieurs fonctionnalités du jeu et à concevoir trois niveaux d'IA, dont un utilisant l'algorithme Monte Carlo Tree Search (MCTS).

L'une des principales difficultés que nous avons rencontrées a été l'utilisation de GitHub pour la gestion de versions et la collaboration en équipe. Étant novices dans l'utilisation de cet outil, la gestion des branches, la résolution des conflits de fusion et la coordination des contributions de chaque membre ont parfois ralenti notre progression. Malgré ces obstacles, nous avons progressivement acquis une meilleure maîtrise de GitHub, ce qui nous a permis d'améliorer notre collaboration et de maintenir une version cohérente et à jour de notre code.

Un autre défi majeur a été l'implémentation de l'algorithme MCTS. Bien que nous ayons réussi à intégrer une IA de type aléatoire, la mise en œuvre de MCTS s'est révélée plus complexe que prévu. Les résultats obtenus avec MCTS n'ont pas toujours été satisfaisants, en partie à cause de la complexité de l'algorithme et des subtilités du jeu de bobail. L'ajustement des paramètres et l'optimisation de l'algorithme ont nécessité beaucoup de temps et d'efforts, et malgré nos efforts, les performances de l'IA utilisant MCTS n'ont pas atteint nos attentes initiales.

Malgré ces défis, ce projet nous a permis d'acquérir des compétences précieuses en programmation, en résolution de problèmes et en travail d'équipe. Nous avons appris à gérer un projet logiciel du début à la fin, en surmontant des obstacles techniques et organisationnels. Les leçons tirées de cette expérience enrichissante nous seront certainement utiles dans nos futurs projets.