Amazons

Rapport de projet de programmation impérative

présenté par

CARMEN GIACCOTTO
FINN ROTERS

pour
l'Ecole Nationale Supérieure d'Électronique, Informatique, Télécommunications,
Mathématique et Mécanique de Bordeaux

Département d'informatique
12.5.2023

Table des matières

INTRODUCTION

1.1 Le jeu des amazones

Le jeu des amazones est un jeu de territoire dans lequel deux joueurs cherchent à préserver leur capacité de mouvement tout en réduisant celle de leur adversaire. Il se joue traditionnellement sur un plateau d'échecs, sur lequel chaque adversaire dispose d'un certain nombre d'amazones. Pour jouer un tour, un joueur doit d'abord déplacer l'une de ses pièces, puis tirer une flèche sur une case vide qui devient par la suite inaccessible.

Figure 1.1: Image tirée du site BoardGameGeek

Bien que d'invention relativement récente (1988), ce jeu fait l'objet de nombreuses publications, démontrant sa richesse. En 2005, R. A. Hearn a montré dans un article plutôt lisible que le fait de décider le gagnant d'une partie d'amazones était PSPACE-complet, et donc faisait partie des problèmes les plus difficiles à résoudre dans un espace polynomial. Par ailleurs, M. Buro a aussi montré que le fait de déterminer le vainqueur d'une fin de partie était un problème NP-complet. Ces caractéristiques rendent ce jeu intéressant parce qu'elles assurent qu'il ne peut se résoudre de manière triviale. A côté de ces résultats académiques, il existe plusieurs travaux proposant des heuristiques de jeu, comme une fonction d'évaluation des positions, ainsi que des comparaisons d'algorithmes pour les fins de parties. Il est donc envisageable de produire des heuristiques variées pour approximer des stratégies optimales. C'est à ce type d'objectif que ce projet propose de s'attaquer.

1.2 Établissement d'objectifs et planification des validations

Les objectifs qui ont été fixés par notre groupe lors de la phase initiale du projet sont les suivants :

- Développer les clients en s'assurant d'implémenter correctement les fonctions requises et de respecter les spécifications pour la validité des coups.
- Implémenter un serveur de jeu qui organisera les jeux entre clients.
- Développer des algorithmes stratégiques et efficaces pour les clients : il faudra donc explorer différentes stratégies et heuristiques possibles pour évaluer les positions et guider les choix des clients au cours de la partie.

Concernant la stratégie de vérification, nous avons défini les points suivants :

- Vérifier les coups des joueurs : le serveur se chargera de vérifier l'exactitude des coups envoyés par les clients. Des vérifications doivent être effectuées pour s'assurer que les coups respectent les règles du jeu et sont valides. Tout mouvement invalide sera signalé.
- Test du système : effectuez des tests pour vous assurer que l'ensemble du système fonctionne correctement. Nous veillons à ce que toutes les structures et variables que nous créons prennent les valeurs appropriées pour éviter des erreurs indésirables. Nous utilisons la commande printf pour savoir si un test est en cours.

1.3 Problématiques

Le problème principal a été de comprendre et de créer une abstraction du jeu en utilisant les structures de données et les fonctions prédéfinies de la version de base. Ce fut un défi particulièrement difficile pour notre groupe de développeurs. La version de base exigeait de respecter tous les prototypes des fonctions, ce qui nous a posé plusieurs problèmes que nous avons finalement résolus en modifiant différentes parties du code.

De plus, nous avons initialement rencontré des difficultés importantes pour implémenter correctement le Makefile. La création des bibliothèques dynamiques et la programmation correcte du serveur ont été des tâches complexes. Nous avons également eu des difficultés dans la création et l'utilisation des différents graphiques dans nos fonctions. Dans l'ensemble, notre groupe a dû consacrer beaucoup de temps à comprendre ces aspects. Malheureusement, cela a laissé peu de temps pour peaufiner le code.

1.4 Architecture du project

Le projet est constitué d'un ensemble de fichiers écrits en langage de programmation C. Pour compiler le projet, nous utilisons l'outil Makefile, qui crée un exécutable pour chaque fichier, puis compile tous les fichiers en un seul exécutable (serveur). Afin d'améliorer l'indépendance entre les fichiers, nous avons créé un en-tête pour chaque fichier contenant uniquement les includes nécessaires à la compilation.

La figure 1.2 montre la dépendance du fichier d'en-tête du serveur et des clients.

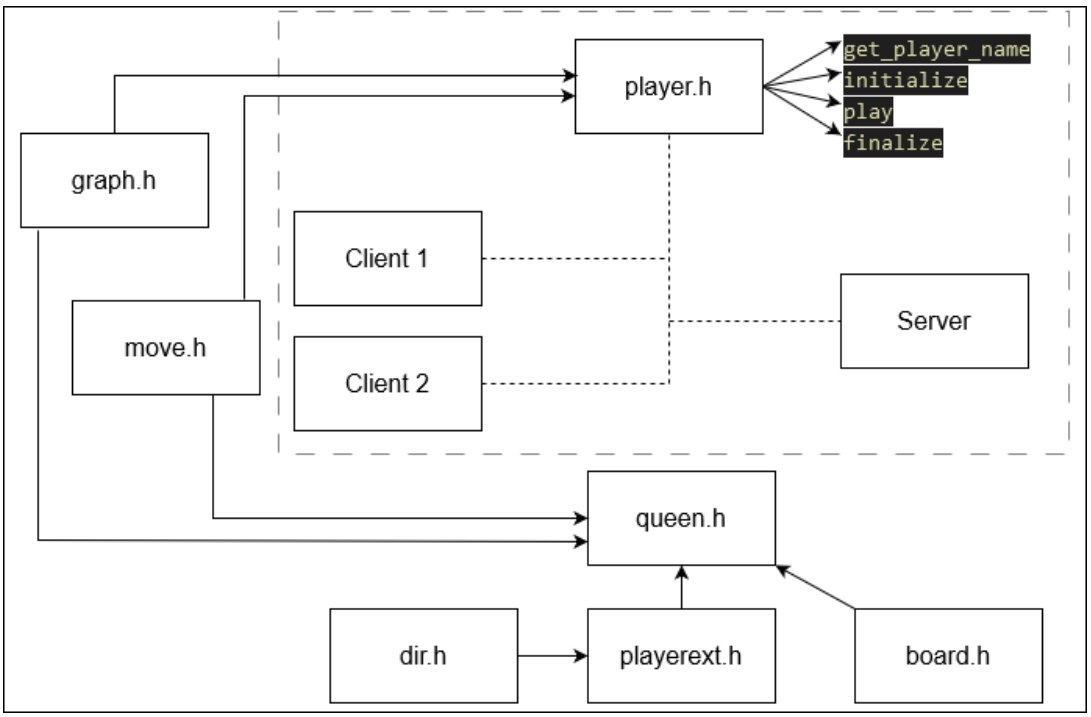


Figure 1.2: Graphe des dépendances

GRAPHE DE JEU

2.1 Génération de graphe

Les plateaux de jeu utilisées pour jouer au jeu des Amazones sont programmées à l'aide de graphes. Ces graphes peuvent avoir différentes formes et tailles variables. Dans notre cas, vous pouvez choisir parmi 4 formes différentes.

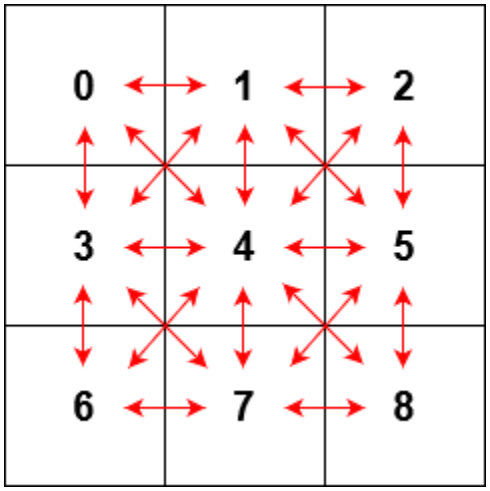
La fonction "generate_graph" est responsable de la génération d'un graphe en fonction des paramètres spécifiés : "m", qui représente la taille du graphe (le nombre de sommets par côté), et "t", qui spécifies le type de graphe à générer. Le paramètre "t" peut prendre des valeurs telles que c, d, t et 8, correspondant à différents types de graphes.

Notre graphe est créé en utilisant une matrice creuse, qui représente les arêtes du graphe. En fait, la fonction crée une structure "graph_t", qui contient un champ indiquant le nombre de vertices du graphe et un autre représentant une matrice creuse. Pour créer les arêtes, une boucle est exécutée pour parcourir chaque vertex présent. À l'intérieur de la boucle, la fonction "index_is_good" est utilisée pour vérifier si un indice de vertex donné est valide, afin de déterminer quels vertices doivent être connectés entre eux en fonction du type de graphe spécifié.

Ensuite, une deuxième boucle est exécutée pour explorer toutes les directions possibles (dir_t) à partir d'un vertex. Pour chaque direction, la fonction "get_neighbor" est utilisée pour obtenir l'indice du voisin correspondant pour cette direction. La fonction "get_direction" renvoie la direction entre le vertex actuel et le voisin.

Une fois que l'indice et la direction du voisin sont obtenus, la fonction "gsl_spmatrix_uint_set" est utilisée pour définir la valeur dans la matrice creuse. Cette valeur représente l'arête entre le vertex actuel et son voisin, indiquant également la direction de l'arête. Enfin, la matrice est compressée au format Compressed Sparse Row (CSR) en utilisant la fonction "gsl_spmatrix_uint_compress". Cette étape optimise la mémoire utilisée par la matrice et simplifie les opérations ultérieures.

La figure 2.1 illustre la relation entre le plateau de jeu et la façon dont le graphe crée des liens entre les différentes cases.



(a) Exemple de plateau de jeu

	0	1	2	3	4	5	6	7	8
0	-	E	0	S	SE	0	0	0	0
1	W	-	E	SW	S	SE	0	0	0
2	0	W	-	0	SW	S	0	0	0
3	N	NE	0	-	E	0	S	SE	0
4	NW	N	NE	W	-	E	SW	S	SE
5	0	NW	N	0	W	-	0	SW	S
6	0	0	0	N	NE	0	-	E	0
7	0	0	0	NW	N	NE	W	-	E
8	0	0	0	0	NW	N	0	E	-

(b) Exemple de graph

Figure 2.1: Plateau de jeu et graphe comparés

2.2 Considérations sur complexité et optimisation

L'implémentation du graphe utilise une représentation des arêtes sous forme de matrice creuse, ce qui permet d'économiser de l'espace mémoire en n'allouant que les arêtes réellement présentes. Ce choix a été fait pour optimiser l'utilisation des ressources disponibles. Cependant, la complexité temporelle de la création du graphe est actuellement $O(m^2)$, car une double boucle est utilisée pour parcourir les vertices et leurs voisins. Cet aspect pourrait être amélioré en utilisant des algorithmes plus efficaces.

Malgré cela, l'implémentation a été choisie pour sa clarté et sa facilité de compréhension. Le code suit une logique intuitive et utilise des structures de données standard, telles que la matrice creuse fournie par la bibliothèque GSL, ce qui le rend facilement maintenable et adaptable. La modularité du code est assurée en séparant les fonctions d'assistance pour obtenir les voisins et les directions, ce qui permet des modifications ou des extensions spécifiques à ces fonctions sans affecter le reste du système.

Nous estimons que cette implémentation est le compromis parfait pour gérer des graphes de taille modérée. Enfin, le temps imparti a fait en sorte que l'optimisation n'était pas notre priorité, et nous avons préféré nous concentrer sur la compréhension, la clarté du code et avoir un graphe qui nous permettait de mener à bien notre projet.

IMPLÉMENTATION DE CLIENTS

3.1 Structure des joueurs

Pour la création d'un client, nous avons commencé par créer une structure de données appelée "player_t", qui représente un joueur dans un jeu.

Nous estimons que cette structure est efficace, car elle regroupe toutes les informations pertinentes pour un joueur en une seule entité. Cela simplifie la gestion du joueur dans le code, permettant un accès rapide et direct à ses propriétés.

De plus, l'utilisation d'un tableau de pointeurs pour stocker les positions des reines permet d'éviter la duplication des données et d'économiser de la mémoire. Au lieu de créer deux tableaux séparés pour les reines du joueur et de l'adversaire, on utilise un seul tableau bidimensionnel, qui offre un moyen efficace d'obtenir et de manipuler les positions des reines pour les deux joueurs.

3.2 Création d'un plateau séparé pour le joueur

Pour rendre toutes les informations sur les mouvements du jeu facilement accessibles aux joueurs, nous avons décidé de représenter un plateau en utilisant une structure appelée "board_t".

La structure "enum_pos_type" définit différents types de positions qui peuvent être présentes sur le plateau :

- EMPTY : une position vide.
- QUEEN : une position occupée par une reine du joueur.
- QUEEN OPP : une position occupée par une reine de l'adversaire.
- BLOCKED : une position bloquée ou inaccessible.

La structure "board_t" contient un pointeur vers un tableau de type "enum_pos_type", qui représente l'état des positions sur le plateau. Lors de la phase d'initialisation du joueur, le plateau est également initialisé en plaçant toutes les reines du joueur et de l'adversaire aux positions correspondantes dans le tableau "pos", en utilisant les informations présentes dans la structure "player_t".

Par exemple, si nous considérons un plateau de taille 3x3 et les valeurs initiales suivantes:

- p.num_vertices = 9
- p.num_queens = 2
- p.player_id = 0
- p.player_id opponent = 1
- p.queens[0] = 0, 4 (positions des reines du joueur)
- p.queens[1] = 2, 8 (positions des reines de l'adversaire)

Après l'exécution de la fonction "initialize_board", le tableau "pos" sera : pos = [QUEEN, EMPTY, QUEEN OPP, EMPTY, QUEEN, EMPTY, EMPTY, EMPTY, QUEEN OPP]

Le plateau représenté par le tableau "pos" aura l'apparence suivante :

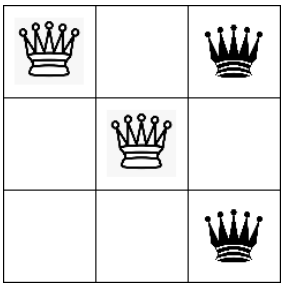


Figure 3.1: Exemple de plateau

De cette manière, chaque position peut être facilement accessible et mise à jour pendant le jeu. La structure permet de vérifier le type de position de manière rapide et directe, simplifiant ainsi la mise en œuvre des règles du jeu et des logiques de déplacement des reines.

3.3 Coup

L'algorithme qui choisit un mouvement à effectuer dans le jeu est réalisé via la fonction "play".

Tout d'abord, une vérification est effectuée sur les mouvements précédents, et si des informations sont disponibles, le plateau est mis à jour en conséquence.

Ensuite, nous avons développé l'algorithme pour effectuer le mouvement de la reine. Toutes les fonctions que nous avons décidé d'utiliser pour le mouvement et les interactions avec les reines ont été déclarées dans un en-tête appelé "amazon_queen.h". Ces fonctions sont essentielles pour l'algorithme de jeu et fournissent des opérations spécifiques sur la reine qui peuvent être utilisées dans différentes parties du code. L'utilisation d'un en-tête séparé contribue à améliorer l'organisation du code et sa modularité, et permet d'inclure uniquement les fonctions nécessaires dans différentes parties du code, réduisant ainsi les dépendances et simplifiant la maintenance du code.

L'algorithme choisi, qui sera expliqué en détail dans les paragraphes suivants, est un algorithme qui suit une logique aléatoire. Il offre une bonne combinaison de hasard et de précision dans le choix des mouvements.

3.3.1 Déplacement de la reine

Nous avons développé un algorithme pour déplacer une reine de manière aléatoire à l'intérieur de l'échiquier. Le choix de la reine se fait de manière aléatoire en sélectionnant un indice au hasard parmi les reines disponibles. Avant de procéder au déplacement, nous vérifions si la reine choisie aléatoirement est bloquée, c'est-à-dire si elle ne peut se déplacer dans aucune direction. Dans le cas où elle est bloquée, nous parcourons les autres reines pour déterminer s'il en existe d'autres disponibles. Une fois qu'une reine disponible est sélectionnée, nous procédons au déplacement. La reine est déplacée vers la première position disponible, dans une case adjacente à sa position actuelle. Pour déterminer les directions disponibles, nous utilisons le format de représentation CSR (Compressed Sparse Row), qui stocke les indices des directions dans un tableau. Nous comparons ces indices avec notre échiquier et, si la position correspondante est vide, nous déplaçons la reine. La figure 3.2 donne une idée de la façon dont la reine se déplace.

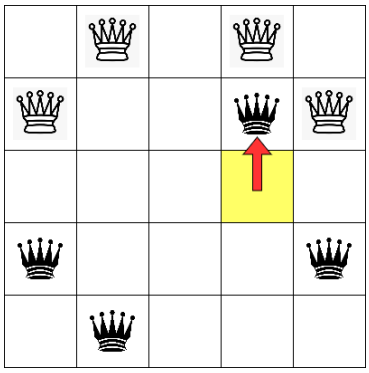


Figure 3.2: Mouvement de la reine

L'algorithme a été choisi pour sa simplicité et sa randomisation dans le choix de la reine à déplacer, ce qui peut rendre les coups plus imprévisibles et augmenter la diversité des stratégies de jeu. Il y a certainement des possibilités d'amélioration et d'implémentation de stratégies plus sophistiquées à l'avenir.

3.3.2 Choix aléatoire de la position à bloquer

Nous avons mis en œuvre un algorithme aléatoire pour déterminer la position où tirer la flèche avec la reine. Pour ce faire, nous avons créé la fonction "shoot_rand".

Dans la fonction "shoot rand", nous effectuons une boucle sur toutes les directions possibles. Nous utilisons la fonction "get_neighbor_queen" pour obtenir l'indice du voisin de la reine dans une direction spécifique. Si le voisin est valide (c'est-à-dire s'il existe et que la case correspondante sur le plateau est vide), nous l'ajoutons à un tableau de positions disponibles où il est possible de tirer.

Nous continuons à nous déplacer dans la même direction jusqu'à ce que nous trouvions une position bloquée ou atteignions le bord de le plateau. Ainsi, le tableau "available_shoot" contiendra toutes les positions où il est possible de tirer la flèche à partir de la reine sélectionnée.

Enfin, nous sélectionnons aléatoirement une position dans le tableau "available_shoot" et la renvoyons comme résultat.

La figure 3.3 donne une idée de la façon dont la reine tire une flèche.

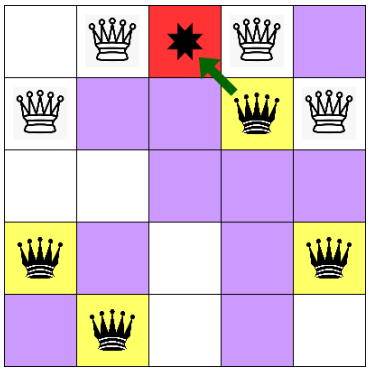


Figure 3.3: Lancement d'une flèche

Un point fort de cette approche est qu'elle permet à la reine d'explorer différentes directions et points de l'échiquier de manière aléatoire, offrant ainsi une plus grande diversité de mouvements et de stratégies potentielles. Cela offre un bon point de départ pour le déplacement de la flèche de la reine.

ÉLABORATION DU JEU

4.1 Définition des structures de données

Toutes les données importantes pour le jeu et les deux joueurs sont enregistrées dans le serveur. Il y a une structure qui stocke la taille du jeu, la forme du jeu, les reines et le graphique. Des copies du graphique et des reines sont créées pour chaque joueur. Dans une autre structure, les pointeurs vers les fonctions player_name, initialize, play et finalize sont stockés pour les joueurs. Ces deux structures sont accessibles à tout moment pendant le jeu.

4.2 Initialisation du jeu

Pour initialiser le jeu, on commence par prendre en argument les deux bibliothèques dynamiques ainsi que t (la forme du jeu) et m (la taille du jeu). Ensuite, pour charger les pointeurs des fonctions de jeu, on ouvre les deux bibliothèques dès le début et on les enregistre dans la structure mentionnée ci-dessus. En utilisant une fonction, les positions des dames sont définies en fonction de la taille du jeu. La figure 4.1 montre une formation de départ possible pour les Queens.

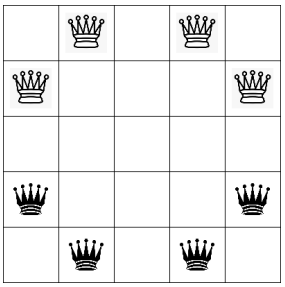


Figure 4.1: Initialisation des queens de taille m = 5

Les deux clients reçoivent toutes les informations nécessaires du jeu via la fonction initialize.

4.3 Début du jeu et mise à jour du tableau

Après l'initialisation du jeu, le premier joueur est choisi au hasard pour effectuer le premier coup. Le serveur vérifie la validité de ce coup. Si le coup n'est pas valide, le jeu se termine immédiatement et l'adversaire remporte la partie. Sinon, le terrain de jeu ainsi que les positions des dames sont mises à jour.

De plus, pour chaque tour, le terrain de jeu est affiché dans le terminal afin de voir dans quel état se trouve le jeu. L'illustration 4.2 montre un exemple de jeu avec la forme Donut et la taille 9.

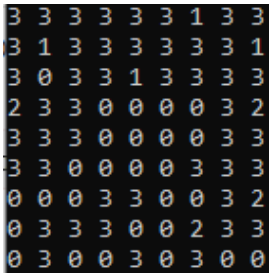


Figure 4.2: Capture d'écran du serveur

Les zéros au milieu sont injouables, car il s'agit de la forme donut. Les autres zéros sont des cases libres. Les 1 sont les dames du joueur 1, les 2 sont les dames du joueur 2 et les 3 sont des cases bloquées par des flèches.

4.4 Déterminer la fin du jeu

Le jeu se termine lorsqu'aucune des dames d'un joueur ne peut plus se déplacer, car toutes les positions avoisinantes sont bloquées, comme illustré dans le figure 4.3.

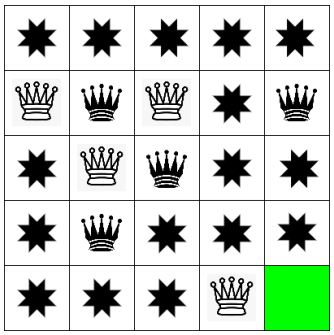


Figure 4.3: Initialisation des queens de taille m = 5

Pour s'assurer que les conditions sont remplies avant de jouer chaque coup, une boucle while est mise en place. Si les conditions sont remplies, une fonction est appelée pour libérer chaque zone de mémoire allouée et appeler les fonctions finalize des deux joueurs.

AMÉLIORATIONS POSSIBLES

La totalité de nos travaux dans le projet "Amazons" ne peuvent pas s'arrêter à ce niveau, plusieurs fonctionnalités parmi celles qu'on a pu faire, peuvent être largement améliorées et plusieurs nouvelles idées peuvent perfectionner notre travail. Dans cette partie, on parlera de quelques-unes, et on fera une analyse critique de notre travail tout au long le projet.

5.1 Algorithme gagnant pour faire un coup

Nous sommes conscients que l'un des aspects que nous aurions aimé améliorer dans notre jeu est l'algorithme pour effectuer les mouvements. Nous avons consacré beaucoup de temps à réfléchir à des stratégies gagnantes et à la façon de les implémenter, mais en raison de contraintes de temps et de personnel, nous avons rencontré certaines difficultés à les représenter de manière adéquate. Cependant, nous souhaitons tout de même expliquer brièvement ces stratégies et rendre hommage au temps que nous avons passé pour les élaborer.

Notre idée était d'avoir deux types de joueurs distincts : l'un agissant de manière aléatoire, que nous avons effectivement implémenté, et un autre doté d'une stratégie gagnante. La stratégie consiste à générer le premier mouvement de manière aléatoire. Ensuite, nous déplaçons la reine qui est dans la plus grande difficulté, c'est-à-dire celle qui a le plus d'obstacles autour d'elle. Cette reine bloquera une position adjacente à l'une des reines adverses qui a le plus de côtés bloqués. Ce mouvement est répété pendant une dizaine de tours, puis nous changeons de stratégie.

L'idée est de comprendre quelle reine se trouve dans une zone suffisamment libre de l'échiquier et de chercher à l'enfermer à l'intérieur de cette zone, de manière à la rendre impossible à bloquer pour l'adversaire. De cette façon, nous visons à limiter les possibilités de mouvement de l'adversaire et à créer une situation favorable pour nous.

5.2 Organisation du code et complexité algorithmique

Comme discutée précédemment dans certaines parties de ce rapport, la complexité générale n'a pas été notre priorité et aurait pu être améliorée. Cependant, le projet actuel a une structure claire et fonctionne de manière efficace. Si nous avions eu plus de temps, nous aurions pu également améliorer davantage l'organisation du code en utilisant des en-têtes et des fichiers séparés pour faciliter et rendre plus immédiat l'accès à certaines parties du code. Nous considérons certainement que prendre en compte la complexité est très important, car cela peut avoir un impact significatif sur la qualité et la maintenabilité du travail accompli.

CONCLUSION

Ce projet a été un défi passionnant pour nous. Il représente une étape significative dans notre parcours de croissance en programmation avec le langage C : il nous a permis de développer considérablement nos compétences, de réfléchir à nos limites et d'améliorer notre capacité d'organisation et de travail d'équipe.

Bien qu'il y ait encore de nombreux aspects qui peuvent être améliorés, nous sommes extrêmement fiers des résultats obtenus. Pendant l'implémentation, nous avons rencontré plusieurs difficultés et il n'a pas toujours été facile de trouver des solutions adéquates.

De plus, ce projet nous a fait comprendre l'importance de divers aspects de la programmation auxquels nous pourrions ne pas avoir accordé suffisamment d'attention par le passé, tels que la complexité du code, la création d'une architecture fonctionnelle et efficace, et l'utilisation de tests pour garantir le bon fonctionnement des fonctions.