Azzi Yoann N'GBALA YAO
Angèle RAMAUGE

TD5

Luttes du Mido-âge

Table des matières

Dans le cadre de notre deuxième année de licence Mathématiques Informatique, nous avons eu la possibilité de réaliser un projet en langage C dans le but de mettre en pratique ce que nous avons appris durant le premier semestre.

Ce jeu de combat nommé Lutte du Mido-âge permet aux joueurs de réaliser les combats de deux des maisons les plus importantes de Westeros[1] : les ***Lannister*** et les ***Stark***.
Dans ce document, nous allons d'abord expliquer nos choix dans la réalisation du jeu, ensuite nous ferons un point sur les fonctionnalités apportées et enfin nous évoquerons les difficultés rencontrées

[1] Westeros : Continent fictif de la série Game of Thrones

Dans ce document nous utiliserons d'autres noms pour désigner les structures que nous avons créées.

Voici un tableau de concordances de ces termes.

Types	Noms	Mot utilisé
Structure	World	Monde
Structure	Character	Personnage
Structure	Node	Noeud
Structure	Neighbours	Voisins
Structure	Player	Joueur
Personnage	Castle	Chateau
Personnage	Warrior	Guerrier
Personnage	Lord	Seigneur
Tableau de pointeur de Case	Board	Plateau
Structure Liste de voisins	ListN	Liste

I. Analyse du programme

Le programme du jeu contient au total 9 fichiers qui sont :

- Perso.h
- Perso.c
- Case.h
- Case.c
- Monde.h
- Monde.c
- Partie.c
- Partie.h
- main.c

Perso.h

Dans le premier fichier Perso.h, nous avons défini une de nos structures principales : la structure Personnage. Comme champs de cette structure, nous avons d'abord le type du personnage et sa couleur, qui sont tous deux de type caractère, ensuite deux tableaux de deux entiers contenant les coordonnées de la position et la destination. Notre idée originale était de créer un pointeur vers une case directement mais cela avait pour conséquence que la structure Personnage et la structure Case définie plus loin dépendent l'une de l'autre. Nous avons donc préféré donner la position du Personnage sous la forme d'un tableau. Chaque personnage possède aussi deux pointeurs vers deux autres personnages, celui qui le précède et celui qui vient après lui, cette caractéristique a été rajoutée du fait de la liste chaînée des agents qu'on devait créer. Le Personnage de type château possède deux attributs en plus qui ne sont pas pris en compte chez les autres personnages. Ces caractéristiques sont la production : un caractère donnant le type du personnage en train d'être produit, et le temps de production donnant le nombre de tour à attendre avant la création du personnage demandé. Nous avons choisi de ne pas mettre d'autres champs pour des personnages placés sur une même case.En effet, réutiliser les listes de voisins comme pour les châteaux a facilité en quelque points notre passage au niveau 4:les fonctions de cette structure étant déjà créé cela a évité la réécriture de de nouvelles fonctions. Nous définissons ensuite les prototypes des fonctions associées directement à la structure Personnage qui seront présentes dans le fichier Perso.c

Perso.c

Les fonctions contenues dans Perso.c ont des algorithmes très simples et linéaires. Ils sont de complexité Θ (1).

Regardons d'abord la fonction ***createCharacter(),*** cette fonction retourne un pointeur sur un Personnage. Le personnage retourné a les coordonnées de sa position qui sont égales à celles de sa destination pour marquer son immobilité. De plus, sa couleur et son type sont en argument de la fonction.
On initialise la production du Personnage à '\0','\0' étant le caractère nul il ne désigne rien de particulier.Le temps de production du personnage est aussi initialisé à -1 car un temps de production positif ou égal à 0 est réservé seulement pour les châteaux qui sont en train de produire. On initialise aussi les pointeurs sur les personnages précédant et suivant à NULL, le personnage n'étant lié à aucun château au départ. La fonction retourne le personnage ainsi initialisé.

Parlons maintenant de la fonction ***DisplayCharacter(),*** cette fonction prenant en argument un pointeur vers un Personnage, permet d'afficher un personnage par son type et par sa couleur.Elle ne retourne rien.
La fonction ***productionCost()*** prenant en argument un Personnage et renvoyant son coût de production à partir de son type. Cette fonction a été créée dans le but faciliter l'utilisation de la fonction ***battle()*** dont on parlera un peu plus, plus bas.

Finalement, nous avons la fonction ***freeCharacter()*** qui libère l'espace alloué à un Personnage.

Case.h

Dans le fichier case.h, nous avons inclus le fichier Perso.h car nous utilisons la structure Personnage dans la définition de la structure Case. Elle est constituée d'un pointeur sur une Liste de Personnages qui occupent la case.

Nous avons aussi défini une structure Noeud ayant pour champs deux entiers représentant les coordonnées d'une case. Cette structure a été créée pour la fonction déplacement.

Case.c

Les fonctions associées à la structure Case sont définies dans les fichiers Monde.c et Monde.h

La structure Noeud quant à elle a pour fonctions spécifiques :

La fonction ***createNode()*** prend en argument deux entiers et renvoie un pointeur sur un nœud en initialisant ces valeurs avec les coordonnées rentrées en argument. Elle servira dans la fonction ***move()*** à pouvoir créer des nœuds des différentes cases voisines de la case qu'occupe un Personnage. Puis ***correctIndex(),*** cette fonction vérifie si les coordonnées : deux entiers ne dépassent pas les dimensions du tableau du jeu dans notre cas ici 8x8.
distanceCalculation() qui prend en argument 4 entiers, coordonnées de deux Nœuds et permet de calculer la distance euclidienne entre ces Nœuds. Toutes ces fonctions ont une complexité de Θ (1).

Enfin, la fonction ***freeNodeArray()*** qui permet de libérer l'espace alloué à des nœuds contenus dans un tableau plus précisément le tableau utilisé dans la fonction ***move().*** Le temps d'exécution de cette fonction dépend du nombre de nœuds créés dans les fonctions précédentes. On a donc une complexité de Θ(a) où a est le nombre de Nœuds créés dans le tableau de Nœuds.

Monde.h

Dans le premier fichier, qui est Monde.h, nous donnons en premier la structure Voisins qui est composée d'un pointeur sur un personnage et de deux pointeurs vers les Voisins précédant et suivant. Cette structure sert seulement de maillon pour une Liste. La structure Liste est une liste basique de maillons doublement chaînés. Nous avons créé cette structure pour manier facilement les listes de Voisins que ce soit de château ou au sein d'une même Case.

Nous avons ici défini la structure Monde qui est essentielle au jeu. Le premier champ de cette structure est un tableau de Cases qui est le support du jeu. Les deux champs suivants sont des pointeurs sur les châteaux initiaux de chacune des couleurs.
Puis, nous avons défini les champs qui représentent les trésors des couleurs. Cela permet d'avoir accès au trésor facilement dans n'importe laquelle des fonctions qui a une structure World en arguments (c'est le cas de la plupart des fonctions). Les deux champs suivants sont le nombre de victimes de chaque couleur. Ces champs permettent de tenir compte des victoires ou capitulations (en cas de suicide de l'adversaire) des différentes couleurs. Nous avons ensuite placé dans cette structure deux pointeurs vers des Liste de château voisins.

Vient ensuite la définition d'une structure Joueur qui permet de conserver durant le jeu, les données des deux joueurs : les scores qui seront incrémentés au fur et à mesure en fonction de l'activité du joueur dans le jeu.
Nous définissons ensuite les fonctions ayant besoin du Monde en argument.

Monde.c

La première fonction définie dans ***Monde.c*** est ***createWorld()*** qui renvoie un pointeur vers une structure World. Cette fonction est de complexité Θ (1) car la fonction ***createListeN()*** et ***createCases()*** le sont aussi. Cette fonction n'est pas différente de ce que l'on a fait en cours et créé le Monde et l'initialise avec des 0 ou des NULL.

La fonction ***initWorld()*** crée les Personnages déjà placés au début de la partie et initialise les trésors. Cette fonction a une complexité de O(m) où m est le nombre de Personnages de cette couleur présents sur le plateau car elle fait appelle à la fonction ***addCharacter()*** qui est de cette complexité et que le reste de la fonction est Θ(1). On a séparé cette initialisation de la création du Monde car le Monde n'est pas initialisé de la même façon si on commence une nouvelle partie ou si on reprend une partie sauvegardée.

La fonction ***displayWorld()*** est une fonction permettant d'afficher le plateau de jeu. Elle est de complexité Θ (1) car chaque boucle n'est itérée qu'un nombre fini de fois et n'appelle que des fonctions de complexité Θ (1). N'ayant pas eu le temps d'apprendre à faire une interface une interface graphique cette fonction a été ajustée pour afficher des tirets pour permettre de visualiser le plateau.

Nous avons décidé de séparer la fonction de création du Personnage ***createCharacter()*** et celle de ***addCharacter()*** qui ajoute le Personnage sur le plateau car dans le cas de la mort d'un château tous les manants qu'il a produit se retrouvent ajoutés à la liste des agents d'un château adverse sans être supprimés. Cette fonction a une complexité de O(t) où t est le nombre de Personnage de cette même couleur déjà existants (château ou non). Cette complexité est dû au fait qu'on parcourt successivement au plus t voisins

jusqu'à trouver le château auquel on relie notre nouveau personnage et on ajoute le personnage créé à la fin de la liste des agents de ce château. Ainsi dans le pire des cas, on parcourt tous les châteaux jusqu'au dernier. Nous avons eu des difficultés à lier correctement le nouveau Personnage au château correspondant. Nous avons réussi à corriger ces erreurs en prenant en argument le château auquel on doit lier le nouveau personnage.

La fonction ***compareArray()*** est une fonction simple de complexité Θ(k) où k est la longueur des tableaux à comparer. Dans le programme on l'utilise principalement pour comparer les positions ou destinations de différents Personnages. Nous avons créé cette fonction pour éviter des répétitions.

La fonction ***decreaseTreasure()*** est de complexité Θ(1) et sert à vérifier si le trésor est suffisant pour créer un Personnage ou le transformer. Si c'est le cas, il diminue le trésor.

La fonction ***transfoC()*** sert uniquement à transformer un seigneur en château. Cette fonction est de complexité O(n) où n est le nombre de château de la couleur du seigneur à transformer et m le nombre total de Personnage de cette couleur. On a une telle complexité car on parcourt au plus les n Voisins puis on appelle une fois la fonction ***addToList()*** qui est de complexité Θ(1).

On a ensuite quatre fonctions similaires ***returnW(), returnL(), returnM() et returnC()*** qui renvoient chacune le premier Personnage d'un type précis rencontré. Elles ont donc une complexité de O(n) où n est le nombre de Personnage dans la case. Elles ne sont utilisées que dans les combats.

On a ensuite ***toBeBeaten()*** qui sert lors d'un combat à décider quel est le Personnage se battant en premier. Elle sert à respecter l'ordre d'engagement dans un combat des agents présents sur une même case. Pour cela, elle utilise les fonctions ***returnW(), returnL(), returnM() et returnC()*** successivement. On a donc une complexité de O(n4).

La fonction ***battle()*** est de complexité O(tl+n)où t est le nombre de Personnage d'une couleur, l le nombre de Personnages liés à un château et n le nombre de château de cette couleur. On a une telle complexité car on utilise la fonction ***toBeBeaten()*** qui renvoie le Personnage qui s'engage dans le combat avec le nouvel arrivant. De plus ***battle()*** utilise les fonctions ***deleteNeighbour()***, ***characterSuicide()*** et ***CastleSuicide()*** si le personnage tué est un château. La seule difficultée rencontrée en faisant cette fonction est que nous nous sommes rendu compte que très tardivement que nous devions proposer un mode où le joueur peut choisir l'issu du combat. A cause de cela nous avons dû modifier de nombreuses fonctions pour pouvoir passer en arguments de cette fonction une variable donnant le mode de combat choisi par le joueur.

Nous avons créé la fonction ***characterCastle()*** qui renvoie la château auquel est lié un Personnage pour que dans la fonction ***castleSuicide(),*** on puisse correctement lier un manant à un château adverse si son château est tué lors d'un combat. Cette fonction à une complexité de O(t) où t est le nombre de personnages (châteaux compris) d'une couleur car dans le pire des cas le personnage qui se bat contre le château est le dernier des personnages du dernier château.

La fonctionnalité de suicide est la fonctionnalité qui a été la plus difficile à créer. En effet à l'origine il s'agissait d'une unique fonction ne renvoyant rien. Cependant cela a posé des erreurs dans le déroulé du jeu car dans un Tour le programme ne pouvait pas passer au Personnage suivant. Nous avons alors pensé à retourner le personnage qui doit jouer après. Cela a fonctionné mais il y avait des redondances dans le code. Nous avons alors diviser la fonction suicide en deux sous-fonctions ***characterSuicide()*** et ***caslteSuicide()***. Ainsi la fonction

characterSuicide() est de complexité O(t+n) où n est le nombre de voisin dans la Case du Personnage et t le nombre de Personnages de cette couleur présents sur le plateau. En effet cette fonction parcourt au plus les t Personnages de la couleur pour trouver celui que l'on doit tuer puis toujours dans le pire des cas le suicide n'est pas lié à la perte d'un combat et on utilise ***deleteNeighbour()*** qui a une complexité de Θ(n). La fonction ***castleSuicide()*** est de complexité O(n+l(t+m)) où n est le nombre de châteaux d'une couleur, l le nombre de Personnages liés au château qui meurt, t le nombre de Personnages de la couleur du château et m le nombre de Personnages maximum étant présent sur une Case de la couleur du château.

Nous avons cette complexité car la fonction parcours au plus tous les châteaux puis après avoir trouvé le château à tuer, supprime un à un les l Personnages liés avec la fonction ***characterSuicide().*** Cette fonction nous a causé beaucoup de difficultés pour éviter toute fuite de mémoire. Nous avons réussi à enlever toute fuite de mémoire en reprenant minutieusement chaque allocation et libération de mémoire de cette fonction et des autres.

Pour créer la fonction ***move()***, il y a eu beaucoup de tentatives et d'idées abandonnées. Finalement, nous sommes arrivées à version simple qui s'inspire de l'algorithme a*.
En effet, la fonction prend en paramètre le personnage à déplacer, fait un tableau de nœud avec les cases voisines du personnage : chaque nœud contient les coordonnées d'une case voisine à la case du personnage à déplacer. Ensuite à l'aide d'un algorithme de tri par sélection, les nœuds sont triés en fonction de leur distance euclidienne à la destination du personnage par ordre croissant. Le premier élément du tableau est ainsi la case où doit aller le personnage.
Si cette case est occupée par un personnage d'une autre couleur, cela donne lieu à un combat entre les personnages ou si elle est occupée par un personnage de la même couleur sinon le personnage se déplace vers cette case. La complexité de cette fonction est en Θ (1) (Θ(taille tableau voisin) vu que la taille du tableau est constante i.e. 8 et les autres instructions se font en Θ(1)

Nous allons maintenant voir les fonctions associées aux structures Liste et Voisins.
Les fonctions ***createNeighbour()***, ***createListN()*** et ***addToList()*** sont simples et de complexité Θ(1). Ces fonctions ne diffèrent pas de ce qui a été vu en cours de TDs et TPs.

La fonction ***deleteNeighbour()*** est de complexité O(n) où n est le nombre de Voisins dans la Liste des châteaux voisins d'une couleur. Au début nous n'avions pas fait de disjonction de cas pour les Voisins en tête de Liste et les autres. Cela ne donnait aucunes erreurs au début mais rapidement on a remarqué que les Voisins se supprimaient mal. Nous avons alors différencié le cas où on supprime le premier Voisin, le dernier Voisin où un autre Voisin quelconque.

La fonction ***freeListN()*** utilise la fonction ***deleteNeighbour()*** dans une boucle parcourant les n Voisins de la Liste. On a donc une complexité de Θ(n).

Nous avons remarqué cependant que quand on essayait de libérer les Listes de voisins d'une Case à la fin de la partie il y avait des fuites de mémoires. En effet nous nous sommes rendues compte qu'en libérant les Listes des Cases avec ***freeListN()*** on ne libérait pas les Personnages mais seulement les Voisins. Ainsi nous avons créé une nouvelle fonction ***freeListNCase()*** qui libère une Liste et donc ses Voisins avec les personnages à l'intérieur de la liste aussi. Cette fonction parcourt tous les Voisins d'une Liste et on libère leur Personnage avant de les libérer. Ainsi on a une complexité de Θ(n^2).

La dernière fonction de ce fichier ***freeWorld()*** *libère* les Cases une à une avec ***freeCases()*** puis nous libérons les Listes de châteaux Voisin avec ***freeListNCase().*** Ainsi nous avons une complexité de Θ(n^2).
La fonction ***createCase()*** ne prenant aucun argument et renvoyant un pointeur sur une Case, crée un pointeur vers une Case en lui allouant une place en mémoire. Le pointeur vers une Liste de Personnages occupants la Case est initialisé avec ***createListN()*** qui est de complexité Θ(1). Ainsi cette fonction est aussi de complexité Θ (1).

On a ensuite la fonction ***displayCharacterN()*** qui affiche une liste de Personnages étant dans la même Case. Cette fonction utilise ***displayCharacter()*** qui est de complexité Θ(1) donc ***displayCharacterN()*** est de complexité Θ(1). A cause du manque de place pour représenter les Personnages sur le plateau, seuls la tête et la queue de la liste de personnages voisins de chaque case sont affichées.
La fonction ***displayCase()***, prenant en argument un pointeur sur une Case permet d'afficher celle-ci : elle affiche des espaces si la case est inoccupée à cause d'une erreur quelconque ou appelle la fonction displayCharacterN() sinon. On a donc une complexité de Θ (1).

Enfin, nous avons la fonction ***freeCase()*** qui libère la case après avoir libéré la Liste de Personnages qu'elle contenait avec la fonction ***freeListN()*** qui est de complexité Θ(n) où n est le nombre de Voisins présents dans la Liste. Ainsi ***freeCase()*** a une complexité de Θ(n).

Partie.h

On s'intéresse ensuite au fichier Partie.h. Ce fichier contient les prototypes des différentes fonctions utilisées lors d'une partie de lutte Mido-âge. Elles utilisent les différentes structures définies dans les autres fichiers headers. Ces fonctions sont alors définies dans le fichier Partie.c.

Partie.c

Dans ce fichier on a d'abord ***characterProduction()*** qui crée le Personnage en production d'un château. Cette fonction utilise ***createCharacter()*** et ***addCharacter()***. Elle a donc une complexité de Θ(t) où t est le nombre Personnages d'une couleur. Si cette fonction a posé des problèmes avant le niveau 4 à cause des Cases occupées ou non ce n'est plus le cas après le niveau 4 où le Personnage est créé sur la même Case que le château.

La fonction ***produce()*** change les champs d'un château si il y a assez de trésor. Cette fonction est de complexité Θ (1).

On a ensuite quatre fonctions qui définissent quelles actions peuvent faire les différents agents.
actionC() a la même complexité que ***caslteSuicide()*** dans le pire des cas et est Θ(1) si le Personnage passe son tour.
Les trois autres fonctions : ***actionL(), actionW()*** et ***actionM()*** ont la même complexité que ***battle()*** au pire des cas et Θ(1) dans le meilleur des cas.

Ensuite il y a la fonction ***turn()*** qui parcours les t Personnages d'une couleur et qui pour chacun affiche le Monde avec ***displayWorld()*** puis appelle la fonction d'action du type du Personnage. On a donc une complexité de O(t^2).

La fonction ***scores()*** parcours tous les Personnages du plateau et augmente les scores des joueurs en fonction de la couleur et du type du Personnage. Cette fonction a une complexité de Θ(t) où t est le nombre total de Personnages du Monde.

save() est une fonction qui sauvegarde tous les Personnages du Monde 1 à 1. Ainsi elle a une complexité de Θ(t) où t est le nombre total de Personnages du Monde.

leave() est une fonction courte qui est de complexité Θ(1) si on ne sauvegarde pas et Θ(t) dans le cas contraire car on utilise ***save()***.

match() est la fonction utilisée quand on commence une nouvelle partie depuis le début. Cette fonction dépend des joueurs qui jouent au jeu, il n'y a donc pas de limite d'opérations faites par cette fonction. En effet si les joueurs passent leur tour indéfiniment le jeu continuerait indéfiniment. Il en est de même pour la fonction ***currentMatch()*** qui permet de reprendre une partie sauvegardée.

Notre calcul des scores tenant compte des morts tout au long du jeu, cela pose un problème avec les sauvegardes qui elles ne sauvegardent aucun score.

Pour résoudre ce problème on a fait une fonction ***recalculateScores()*** qui quand on reprend une partie calcule une compensation en fonction des Personnages présents sur le plateau. Ainsi cette fonction parcourt tous les Personnages d'une couleur. On a une complexité de Θ(t) où t est le nombre de Personnages de cette couleur. A la fin de chaque partie on calcul les scores et on enregistre les meilleurs dans un fichier. Pour cela on appelle la fonction ***saveBestScores()***. Cette fonction prend ligne par ligne les données stockées dans le fichier bestScores.got et les met dans un tableau puis après avoir ajouté les scores des joueurs, trie le nouveau tableau et remet les scores dans les fichiers. Le nombre de données étant toujours fixé cette fonction est Θ (1).

Main.c :

Dans l'unique fonction de main.c aussi appelée main, il est donné la possibilité de recommencer une partie déjà enregistrée, pour cela il faut mettre en argument le nom du fichier. S'il n'y a pas de fichier en argument, le compilateur considère que l'utilisateur veut commencer une nouvelle partie.

II. Fonctionnement du jeu

Le but du jeu est de détruire tous les châteaux adverses. Pour cela, on fait jouer successivement les Personnages de notre couleur. Au début de chaque partie, le choix est donné aux joueurs de choisir pour quelle couleur ils veulent jouer, puis ils entrent leurs noms. Les actions de chaque Personnage sont déterminées par des commandes écrites par le joueur. Avant chaque action le jeu liste les possibilités pour le Personnage. Si la commande écrite par l'utilisateur ne correspond pas exactement à celle proposée au-dessus, le jeu considère que le joueur passe le tour du Personnage. Un manant immobilisé, le reste jusqu'à la fin de la partie et crée une pièce de trésor par tour. Une fois qu'un personnage est en déplacement ou qu'il n'est pas arrivé à destination, on ne peut changer sa destination à moins de le tuer.

III. Difficultés rencontrées et résolutions

Ce projet est le premier projet de programmation que nous n'ayons jamais fait. Cela en fait une première bonne expérience malgré les nombreuses difficultés rencontrées. De plus, le fait de coder en C demande que nous pensions à chaque cas de figure dans chacune des fonctions de notre programme. Toutes ces exceptions et détails nous ont poussé à réfléchir le plus que nous pouvions.
Les majeures difficultés que nous avons rencontrées sont la fonctionnalité de suicide, et le passage au niveau quatre qui nous a fait changer les ossatures de presque toutes nos fonctions révélant ainsi de nombreuses imperfections.
On peut citer par exemple la fonction ***suicideCharacter()***.

Une autre difficulté a été d'écrire un projet aussi long avec autant de fonctions et ainsi de devoir structurer notre code et notre réflexion pour être le plus efficace et le moins redondant possible.
Notre programme devait ne pas causer de fuites de mémoire, au départ notre programme ne libérait pas toute la mémoire allouée, puis il faisait plus de libération d'espace que d'allocation. Nous avons heureusement pu résoudre ces soucis grâce à une condition de plus dans la fonction ***castleSuicide()*** lors du changement d'allégeance des manants.

Au-delà des problèmes de réflexion et/ou de programmation c'est l'organisation du travail à distance qui a été le plus compliqué. Nous n'avons pas réussi à trouver un environnement de travail qui convienne pour travailler communément et s'accorde à notre rythme de travail.
En effet nous avons essayé de travailler avec GitHub ou en même temps sur le même document avec Atom télétype. Cependant nous n'avons continué ni avec le premier qui demandait de télécharger chaque modification pour que l'autre puisse la voir, ni avec le second qui ne nous convenait pas à cause de nos rythmes de travail différents. Nous avons fini par envoyer par Teams chaque fichier que nous modifions. Cette solution n'est pas la plus efficace car soit on ne peut pas coder en même temps soit on doit reporter chaque modification sur le dossier puis le renvoyer ce qui peut être très long mais c'est la meilleure solution que nous avons trouvée.