WEntreeEntrée WEntreesEntrées WSaisirSaisir WInitialisationInitialisation WTraitementTraitement WSortieSortie WSortiesSorties SiSinonSiSinonifthenelse ifelseendif TqWhiledoEndWhile

École nationale supérieure d'électronique,
informatique, télécommunications,
mathématiques et mécanique de Bordeaux

Rapport de projet S6

Le jeu Carcassonne

Réalisateurs :
Cheikh Lilia
Sandri Jordan
Chlih Zakaria
Naddami Anas
Promotion 2017-2018

Encadrant : Herbreteau Frédéric

18 mai 2018

Table des matières

1 Préambule

Dans le cadre du projet du semestre 6 et dans le but d'approfondir nos connaissances en programmation en language C, il nous a été proposé de modéliser un jeu de cartes appelé *Carcassonne*, coder les stratégies necéssaires pour le déroulement du jeu, et définir les structures permettant de représenter le mieux possible les joueurs, le plateau de jeu, les cartes et les coups des joueurs. L'objectif du projet consiste à implémenter un ensemble de fonctions permettant de faire jouer un nombre quelconque de joueurs à une partie de Carcassonne.

1.1 Vocabulaire

Afin de bien comprendre le fonctionnement du jeu, il est primordial de définir un certain nombre de vocabulaire propre à ce dernier.

- *Deck :* Le deck représente l'ensemble des cartes du jeu qui sont au nombre de 72 cartes. Il existe 24 cartes diffèrentes avec une carte de départ de dos plus foncé nommé CARD_ROAD_STRAIGHT_

- *Tile :* Il s'agit d'une carte tirée du deck (tuile en français) jouée par l'un des joueurs dans le but de gagner des points.

- *Nodes :* Les nodes sont les 13 points placés sur chaque carte permettant de faire des connexions entre les diffèrentes cartes pour pouvoir construire des paysages.

- *Meeples :* Les meeples sont les pions utilisés par les joueurs pour marquer leur territoire.

2 Introduction

2.1 Mise en situation

Carcassonne est un jeu de société. Le thème est la construction d'un paysage médiéval par la pose de tuiles, incluant des villes fortifiées telles que Carcassonne. Ainsi, Carcassonne est un jeu où l'on construit le plateau de jeu au cours de la partie. Des points sont attribués en fonction de la taille des combinaisons créées — villes, champs, routes, abbayes.

Le jeu commence avec une seule tuile, les autres étant dans une pioche. Chaque joueur à tour de rôle pioche une tuile et doit la placer en respectant les tuiles déjà en place : les villes et les routes ne peuvent être coupées.

Après avoir placé sa tuile, et uniquement à ce moment-là, le joueur peut s'il le souhaite placer un pion, désigné par le terme *meeple* en anglais, sur une des parties de cette tuile (morceaux de villes ou de champs, tronçons de chemins, abbayes). La ville, le champ, l'abbaye ou le chemin formé par les éléments contigus deviennent alors la propriété exclusive de ce joueur, et personne, pas même le propriétaire, ne pourra y placer d'autre pion en l'agrandissant par une nouvelle tuile contiguë. Cependant, une nouvelle tuile peut réunir des parties disjointes sur lesquelles il y a déjà des pions. C'est alors le joueur qui a le plus de pions sur le domaine en question qui devient le propriétaire de l'ensemble (si les joueurs sont à égalité, le terrain appartient autant à chacun).

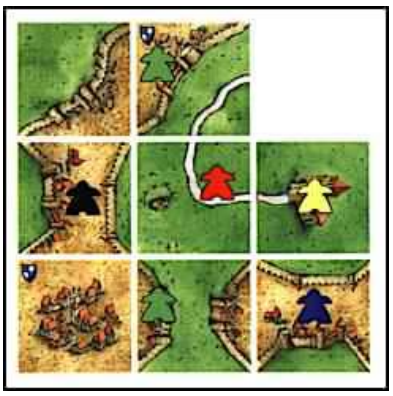

FIGURE 1 – Les différents paysages (villes,chemin,abbaye)

Quand une route, une ville ou une abbaye est complétée, son ou ses propriétaires comptent leurs points (chaque tuile a une certaine valeur), et récupèrent leurs pions. Les pions placés sur les champs y restent jusqu'à la fin du jeu.

Le jeu est terminé quand toutes les tuiles ont été placées. On compte alors des points pour les abbayes, les champs, les routes et les villes non complétées.

2.2 Les règles du jeu

Le jeu *Carcassonne* est géré par un ensemble de règles qu'on pourrait résumer de la manière suivante :

- Au début du jeu, on dispose d'une pile de 72 cartes bien mélangées entre elles (24 types de cartes, chaque type apparaîssant un nombre bien determiné de fois dans la pile). Ces cartes présentent des paysages (chemins, villes, pré, abbaye). La première carte a être posé sur le plateau est CARD_ROAD_STRAIGHT_CITY, le plateau se construit alors autour de cette carte.
- Chaque joueur pioche une carte à tour de rôle, la pose sur le plateau de jeu de telle manière qu'elle soit accolée à au moins une autre carte figurant sur le plateau et qu'elle forme avec cette carte là une tranche d'un paysage. Étant donné les règles du jeu, il est presque impossible que le joueur ne puisse pas placer une carte. Mais si le cas se présente il doit mettre cette carte paysage de côté et piocher une nouvelle carte.

- Une fois qu'il a pioché une carte , le joueur peut positionner un unique *partisan* sur une portion de la carte qu'il vient de déposer sur le plateau. Chaque joueur dispose de 8 *pions* de même couleur. L'emplacement choisi par le joueur pour son *partisan* défini la nature de celui-ci . Par exemple, un *chevalier* devrait être placé dans le quartier d'une ville , un *voleur* sur un chemin, un paysan dans un pré et un *moine* dans une abbaye. Cependant, il existe certaines contraintes qui doivent être prises en compte lors du choix de l'emplacement d'un *pion*. En effet, le joueur ne peut placer son partisan sur la partie de son choix que si il n'y a pas encore d'autres partisans dans les villes, sur les chemins ou dans les prés qu'il a complétés avec sa carte de paysage et ceci indépendamment de la distance qui le sépare de l'autre partisan. Par exemple, s'il decide de placer un chevalier, il doit s'assurer que dans ce paysage là, il n'existe aucun chevalier. Dans le cas contraire, il doit choisir un nouvel emplacement pour son *pion*.

FIGURE 2 – Modélisations des pions sous formes de bonhommes colorés

– Une fois qu'un paysage est complet sur le plateau de jeu, il faut évaluer les points qui en découlent et les affecter au total de points du/des joueurs en question. Les points attribués dependent de la nature du paysage construit, du type de pions placés ainsi que du nombre de cartes des joueurs participant à la création de ce paysage.
Le calcul des points se fait suivant les règles suivantes :
RÈGLE 1 : Une ville est dite achevée si le paysage formé est un espace fermé et si les extrémités de ce paysage sont des remparts. Des points seront attribuées au joueur dont le chevalier figure sur le paysage. Il reçoit 2 points par carte utilisée pour former la ville aussi bien que 2 points par symbole. Une exception s'applique dans le cas où la ville est formée par deux cartes seulement, en effet, le joueur en question ne reçoit que 2 points.

FIGURE 3 – Construction de villes par deux ou plusieurs cartes

Ici le joueur en question reçoit 8 points dans le cas de la figure ?? (gauche) alors qu'il n'en reçoit que deux dans le deuxieme (droite).
Il peut toute fois arriver qu'il y ait plusieurs joueurs dans la même ville dans ce cas le joueur qui possède le plus de chevalier dans cette ville remporte les points. En cas d'égalité les joueurs concernés reçoivent la totalité des points. Dans le cas où il y a plusieurs chevaliers sur une même ville , c'est le joueur ayant le plus grand nombre de chevaliers qui emporte la totalité des points. En cas d'égalité, chacun des joueurs concernés reçoit la somme des points.

RÈGLE 2 : Un chemin est considéré comme achevé lorsqu'il est continu et cloturé sur ses 2 extrémités par une ville, un carrefour ou une abbaye. Le joueur, dont le voleur se trouve sur le chemin, compte alors les cartes qui lui ont permis de le créer et il reçoit le nombre de points correspondant.

Ici, le joueur remporte 3 points dans le cas de la Figure **??** la route est fermée est cloturé par un carrefour. Dans le cas où il y a plusieurs voleurs sur un même chemin, c'est le joueur ayant le plus grand nombre de voleurs qui emporte la totalité des points. En cas d'égalité, chacun des joueurs concernés reçoit la somme des points.

FIGURE 4 – Route achevée

RÈGLE 3 : Le pré est formé de plusieurs cartes de paysage mises les unes à cotés des autres pour obtenir un dessin continu. Pendant la partie, on ne peut pas donner une valeur aux prés ou aux partie de prés, mais on peut y placer des paysans. On ne reçoit les points correspondant qu'à la fin de la partie. Les paysans restent par conséquent, sur les cartes de paysage pendant toute la partie et les joueurs ne peuvent pas les récupérer. Les prés sont délimités par des chemins, des villes et le bout des cartes sur la table.

FIGURE 5 – Pré achevée

Dans le cas de la Figure **??** le joueur ne remportera pas de point puisque que pour marquer des points avec un pion paysan il faut que le champ de ce pion touche une ville fermée dans ce cas c'est 4 points par ville finie. Le champ est fini mais le joueur récupère pas son pion La même règle s'applique en cas d'égalité celui qui a le plus de paysan dans le champ remporte les points, si même nombre de pions les 2 joueurs remportent la totalité des points

RÈGLE 4 : Une abbaye est dite achévée si le paysage obtenu présente un batiment entouré par huit cartes. Dans ce cas là c'est le joueur dont le moine est placé dans l'abbaye qui reçoit un point pour chaque carte entourant l'abbaye et un point pour le pion, donc au total 9 points.

RÈGLE 5 : Si un joueur a terminé une ville, un chemin ou une abbaye en plaçant sa carte, il doit tout de suite compter les points correspondants.

RÈGLE 6 : Vu que chaque joueur dispose d'un nombre réduit de *pions*, le jeu permet, à chaque fois qu'il y a une évaluation d'une ville, d'un chemin ou d'une abbaye, de récupérer les partisans placés, sur les cartes de paysages correspondantes. Au tour suivant, les joueurs peuvent les réutiliser sur de nouvelles cartes de paysage qu'ils décident de placer.

– Après la pioche de la dernière carte du jeu et le placement d'un pion ainsi que la mise à jour des points des joueurs si necessaire vient la dernière étape du jeu, qui est l'évaluation finale. Il s'agit ici d'attribuer des points aux joueurs, selon l'emplacement de leurs pions, pour les villes, abbayes et chemins incomplets.

FIGURE 6 – Paysage incomplet

Dans la figure **??** ci-dessus, le joueur qui a déposé le voleur aura 3 points pour le chemin inachevé alors que le joueur ayant déposé le moine dans l'abbaye aura 5 points pour l'abbaye inachevée.

2.3 Objectif du projet

L'objectif de ce projet est d'écrire un code en C fidèle aux règles du jeu Carcassonne. D'autres objectifs sont de produire un code propre, mais également des fonctions avec une complexité, nous permettant de faire tourner le jeu en un temps d'exécution correct. Pour cela notre projet tournera autour de la question : Comment rendre le code optimisé et fidèle aux règles du jeu ?

3 Implémention

3.1 Intéraction client/serveur

Le jeu Carcassonne comporte beaucoup de règles que les joueurs se doivent de respecter pour cela nous mettrons en place un serveur qui jouera le rôle d'arbitre pour chaque coup joué et mettra à jour le plateau. Chaque joueur disposera de son propre plateau qui sera mis à jour en récupérant les coups valides des autres joueurs lorsque le joueur en question jouera. Seul le serveur dispose d'un plateau qui évolue à la fin de chaque coup joué par un joueur, et seul ce plateau compte pour l'arbitrage. Cependant si les joueurs jouent des coups valides ils auront le même plateau.

Un joueur qui ne respecte pas les règles sera lourdement sanctionné, le serveur l'éjectera du jeu sans sommation, et ne comptabilisera pas son coup sur le plateau, mais on l'ajoutera aux coups

joués en tant que coup invalide pour garder sa trace. Même si le joueur éjecté est gagnant vu son grand nombre de points, c'est le deuxième joueur qui possède le plus de point qui sera nommé vainqueur de la partie. On pourrait permettre au joueur de re-jouer mais le joueur pourrait s'amuser à ne pas faire avancer la partie en faisant que des mauvais coups. De plus cela facilite les modifications des plateaux de jeux des joueurs.

3.2 La boucle globale du jeu

Pour aborder le jeu et modéliser les intéractions entre les joueurs et le serveur, on a opté pour un algorithme général qui suit la démarche du pseudocode suivant :

each player $player \leftarrow initialise()$;; $deck \leftarrow fill\_deck()$;
each player (! is_empty(deck)) and (remaining_players(players) > 1) $card \leftarrow draw\_card()$;
playable(card) (! valid_move(move_player)) $eject\_player(player)$; enqueue(move_player, moves) ;
$player \leftarrow compute\_next\_player()$;
each player $player \leftarrow finalise()$; *Main loop of the game*

Les différentes fonctions figurant dans la boucle principale nous orientent vers la considération de différents types abstraits de données (TAD) pour modéliser chaque partie du jeu. Ces TAD seront exposés dans la section suivante.

4 Définitions des types abstraits de données nécessaires pour la modélisation du jeu

À l'aide de la boucle de jeu, on va déterminer petit à petit les fonctions dont on a besoin d'implémenter pour coder le jeu et également les TAD qu'ils leurs sont associés.

4.1 Les fonctions du TAD deck

Au cours du jeu, les joueurs doivent tirer à tour de rôle une carte parmi un deck de 72 cartes, c'est la première instruction dans la boucle While. On a besoin de fonction permettant d'intéragir avec le deck. Pour cela nous mettrons en place les fonctions de base suivante :

- *struct deck *init_ deck()* : fonction qui crée un deck vide, le remplit avec les cartes du jeu et le mélange.

- *enum card_ id draw_ card(struct deck *deck)* : fonction qui renvoie la carte sur le dessus du deck (la nouvelle carte à piocher).

- *int deck_ is_ empty(struct deck *deck)* : fonction qui renvoie TRUE si le deck est vide, FALSE sinon.

- *void deck_free(struct deck *deck)* : fonction qui libère la mémoire alloué par le deck.

4.2 Déduction d'une structure deck

On a besoin d'un tableau d'énumération pour stocker les cartes du jeu, ainsi que d'un repère pour savoir quelle carte est la suivante, on implémentera la structure suivante :

La structure deck se composera d'un tableau d'énumération *card_ id* contenant les 72 cartes du jeu *next_ card* indiquant la position de la carte suivante à tirer.

On a opté pour une structure semblable à la structure d'une pile. En effet, le champ contenant le tableau des cartes est notre pile qui va contenir toutes les cartes, et le champ next_card est un entier qui va mémoriser le sommet de la pile, indiquant l'emplacement de la première carte à piocher à chaque tour de jeu. Au début du jeu, cet entier est initialisé à 0, puis incrémenté à chaque fois qu'on pioche une carte.

4.3 Les fonctions du TAD moves

On a besoin de mémoriser les coups joués par les joueurs pour pouvoir valider les coups suivants mais également pour pouvoir compter les points par la suite. Une autre fonctionnalité de ce TAD est de transmettre l'information entre le serveur et le client. Pour cela nous aurons besoin des fonctions suivantes :

- *struct moves *init_ moves() :* fonction qui crée une structure moves et initialise les champs de la structure.

- *struct move *last_ n_ moves(struct moves *moves, int number_ of_ moves) :* fonction qui renvoie les coups du dernier tour joué.

- *int add_ move(struct moves *moves, struct move *move) :* fonction qui ajoute un coup joué à la liste des coups précédents.

- *void free_ moves(struct moves *moves :* fonction qui libère l'espace allouée à la structure moves.

4.4 Déduction d'une structure moves

Lorsqu'un coup est joué il faut retenir plusieurs informations. Quelle est la carte posée ? Sa position sur le plateau, l'emplacement d'un possible pion, etc. De plus cette structure nous sert de passerelle entre le client et le serveur on y rajoute alors un champs pour vérifier la validité du coup. Il y aura donc 2 structures pour les coups joués. La structure *move* qui va contenir les informations importantes à retenir lorsqu'un coup est joué et la structure *moves* qui est un tableau de structure move pour mémoriser les coups. On définira les structures suivantes :

La structure moves comportera un tableau contenant les struct move des joueurs, ainsi qu'un champ *count_ move* qui enregistre le nombre de struct move comprisent dans le tableau.

Ainsi que la structure *move* suivante :

La structure move contient un booléen qui indique TRUE si le coup est validé par le serveur FALSE sinon. Un champ *id_ player* qui renseigne l'id d'un joueur. Une énumération *card_ id*

qui est l'id de la carte jouée. Une struct position qui est la position de la carte sur le plateau. Une enum direction qui indique le nord de la carte et une enum place qui donne la position du pion posé sur la carte et *NO_MEEPLE* s'il n'y en a pas.

4.5 Les fonctions du TAD players

Pour le déroulement du jeu, on a besoin de charger les bibliothèques des joueurs, vérifier s'ils sont en vie, éjectés du jeu ou non et calculer les points accumulés lors de chaque tour. Les principales fonctions dont on a besoin sont les suivantes :

- *struct players *init_players()* : fonction qui alloue de la mémoire pour la structure *player* et initialise les differents champs de la structure.
- *unsigned int compute_next_player(struct players *players, struct player *player)* : fonction qui renvoie l'identité du prochain joueur en vie qui va jouer.
- *void eject_player(struct players *players, struct player *p)* fonction qui exclut un joueur du jeu une fois qu'il a joué un coup invalide.
- *int remaining_players(struct players *players)* fonction qui retourne le nombre de joueur encore en vie, les joueurs exclus comptent pour morts.
- *void free_players(struct players *p)* : fonction qui libère la mémoire allouée aux joueurs.

4.6 Déduction d'une structure players

Pour faciliter la communication entre le serveur et le client, il est pratique d'avoir des pointeurs de fonctions pour cela on met en place la structure suivante :

La structure *player_base* contient un pointeur de fonction vers la fonction *initialise* du client, ainsi qu'un pointeur de fonction vers la fonction play du joueur, qui lui permet de renvoyer son coup, un pointeur de fonction qui renvoie le nom du joueur, un pointeur de fonction vers la fonction *finalise* du client et handle un pointeur qui récupère ce que retourne la fonction *dlopen* qui charge les bibliothèques.

Cette structure nous sera trés utile vu qu'elle stocke les adresses des fonctions à charger pour un client. Ainsi, pour charger la bibliothèque relative à un client, on a codé une fonction auxilière *load_functions(void *player, struct player_base *base)* qui accède à la bibliothèque d'un joueur et initialise les champs de la structure *player_base*. Cette fonction nous sera utile pour le codage de *init_players()*.

Certaines informations relatives aux joueurs en plus de la structure player_base doivent être mémorisées, pour cela on met en place une structure *player* comme suit :

La structure player contient un pointeur vers la structure player base du joueur. Un booléen qui indique si le joueur est en vie l'id du joueur. Le nombre de pions qui restent au joueur, et le nombre de points qu'il génère au cours de la partie.

Le premier champs de la structure est un pointeur vers *struct player_ base* . Ceci nous permet de stocker toutes les adresses des fonctions des clients participant aux jeu.

Afin de repérer les joueurs encore en vie, on a ajouté le champ *int player_ alive* à la structure *player* : le joueur correspondant est vivant si ce champ vaut TRUE et exclu du jeu (suite à un coup invalide) ou mort si la valeur du champ est FALSE . Les champs *number_ of_ meeples* et *points* nous servirons pour le comptage des scores et la détermination du joueur gagnant.

À partir de cette structure et de la même manière que pour la structure move il sera plus aisé de stocker les joueurs dans un tableau de joueurs.

La structure players contient un tableau de structure player et un entier *number_ of_ players* qui indique le nombre de joueurs initials en début de partie.

5 Définition des types abstraits de données permettant de valider les coups joués

5.1 Les fonctions du TAD tile

Jusqu'à présent, l'ensemble des structures réalisées permettent de partager des informations sur un coup joué entre le serveur et le client mais aucune vérification n'est effectuée. Les joueurs peuvent placer des cartes sur le plateau comme bon leur semble et il en est de même pour les pions. Il est alors nécessaire de rajouter des fonctions qui vont nous permettre de vérifier les coups joués. Pour cela nous aurons besoin de définir les fonctions qui intéragissent avec les tuiles, nous définirons les fonctions suivantes :

- *struct tile* init_ tile() :* fonction qui alloue de la mémoire et initialise une tuile
- *void free_ tile(struct tile *tile) :* fonction qui libère la mémoire allouée à une tuile.
- *boolean is_ empty_ tile(struct tile *tile)* : fonction qui retourne TRUE si la tuile n'est pas encore initialisé.
- *match_ side(struct tile* tile1, struct tile *tile 2) :* fonction qui vérifie si 2 cartes adjacentes peuvent être connectées (vérification d'une seule face)

5.2 Déduction d'une structure tile

Quelles sont les informations qu'une tuile doit retenir ? Sa position, le type de décor que possède chaque côté de la carte. Nous définissons une énumération permettant de donner un nom au type de décor différent.

L'énumération landscape contiendra ROAD, PLAIN, CITY, CROSSROAD et ABBEY.
Puis la structure tile contiendra un tableau d'énumération landscape des 13 points primordiaux de la carte voir figure **??**, une structure position avec les coordonées x et y de la carte sur le plateau de jeu, l'orientation du nord de la carte, et son nom.

L'orientation de la carte et son nom nous servent particulièrement pour l'affichage.

FIGURE 7 – Représentation des 13 points représentant une carte

5.3 Les fonctions du TAD set_graph

Une fois que nous avons défini ces tuiles nous allons les stocker dans un tableau. Cela va nous permettre d'avoir accès à ces tuiles pendant la partie et de vérifier que les coups joués par les joueurs sont valides. Nous aurons besoin de ces fonctions :

- *struct graph *init_graph() :* fonction qui crée un nouveau graphe et l'initialise.
- *int add_tail_tile(struct graph *graph, struct move *move) :* fonction qui ajoute à la fin du tableau une tuile.
- *int *is_playable(struct graph *graph, enum card_id card) :* fonction qui vérifie si la carte tirée dans la pioche est jouable à ce stade du jeu
- *struct graph *valid_move_card(struct graph *graph, struct move *move) :* fonction qui vérifie si la carte placée par le joueur est valide.
- *struct graph *free_graph() :* fonction qui libère le graphe.

5.4 Déduction d'une structure set_graph

la structure set_graph en découle d'elle même, elle se rapproche de la structure *moves* présentée plutôt.

Cette structure contiendra les champs suivants : Un tableau qui contient les tuiles du jeu au fur et à mesure qu'elles sont jouées et le nombre de tuiles présentent dans ce tableau.

5.5 Les fonctions du TAD node

Pour vérifier la validité du placement des pions par le joueur, la structure tile n'est plus suffisante. Il ne suffit plus de savoir si les côtés des cartes concordent mais si une zone est libre ou occupée par un pion adverse.

- *struct node *init_node() :* qui retourne une structure node alloué et initialisée.

– *int node_ connection(struct node *node1, struct node *node2) :* qui lie 2 nodes entre elles.

– *int node_ disconnection(struct node *node1, struct node *node2) :* qui délie 2 nodes entre elles.

– *int is_ meeple_ in_ area(struct node *node) :* qui parcours le graphe dont la node fournie en fait partie et renvoie si sur le chemin il y avait un pion ou non.

5.6 Déduction d'une structure node

La structure node a besoin de tous les renseignements que doit savoir une node de la carte pour pouvoir parcourir un graphe c'est à dire qu'elles sont les nodes suivantes liées à cette node, quelle est leur landscape, s'il y a un pion et a qui appartient t-il ?

La structure node se compose comme suit : un enum landscape qui indique la catégorie de la node, un booléan qui indique si la node contient un pion et un champ *id_ player* qui indique à qui appartient le pion si le booléen précedent est vrai, un tableau de pointeur vers des struct node voisines pour permettre le parcours de graphe et un entier indiquant le nombres de nodes voisines, pour faciliter le parcours de graphe.

6 Liaisons entre les TAD

Les TAD se divisent en 2 catégories les TAD qui intéragissent entre le client et le serveur et permettent également d'entretenir le plateau, et les TAD qui permettent seulement d'entretenir le plateau. La première catégorie des TAD sont des TAD qui ne sont pas imbriqués les uns dans les autres, tandis que dans la deuxième catégorie on a une inclusion des TAD par ordre d'abstraction. Tout d'abord nous avons vu que dans la structure set_graph on avait un tableau qui contenait les tuiles du jeu au fur et à mesure qu'elles sont posées, et ces tuiles comportes 13 nodes sur les points essentiels de la carte. On a alors une inclusion de la structure node dans la structure tile et de la structure tile dans la structure set_graph.

7 Modifier les structures : les fonctions importantes

Certaines fonctions permettent d'intéragir entre le client et le serveur et d'autres fonctions permettent de vérifier les coups joués par le client. Un vrai réseau se crée entre les tuiles au fur et à mesure de l'avancée de la partie. Par exemple pour vérifier qu'un coup est valide, on appelle la fonction *valid_ move* qui se décompose en 2 appels. Un appel à la fonction *valid_ move_ card* et un appel à la fonction *valid_ move_ meeple*.

La fonction *valid_ move_ card* permet de vérifier qu'une carte est posée au bonne endroit. Tout d'abord à l'aide la fonction *match_ card* on vérifie que les cartes adjacentes à la nouvelle carte posée sont du même landscape si c'est le cas les connexions entre les cartes sont réalisées comme sur la figure **??** :

C'est la fonction *add_ tail_ tile* qui s'occope d'ajouter la carte dans le tableau des cartes, elle appelle ensuite *add_ card_ connexion* qui fait la connexion entre les cartes et plus précisement les nodes. La fonction *valid_ move_ card* est en Θ(nombre de voisin) c'est à dire en $\Theta(1)$. L'ajout d'une nouvelle carte dans le tableau est également en $\Theta(1)$ du fait que l'on tient à jour le

FIGURE 8 – Représentation des liaisons entre les 13 points représentant une carte

nombre de tuiles présentent dans le tableau. Les connexions entre les cartes sont également en $\Theta(1)$ puisqu'on relie les nodes voisines par des pointeurs et ses nodes sont mémorisées dans un tableau dans la structure node.

La fonction *valid_move_meeple* permet de vérifier qu'un pion poser sur une carte à le droit d'être poser en fonction des autres pions sur le plateau, mais aussi en fonction du tableau *allowed_positions* qui donne les positions permissent d'un pion sur chaque carte. Enfin, elle vérifie si le joueur à assez de pion pour jouer et met à jour le nombre de pions du joueur en sa possésion. La vérification de la validité de ce pion dépend des autres pions sur le plateau. À l'aide des liaisons faites entre les cartes, on peut parcourir un graphe en fonction d'une node données et vérifier dans chaque node s'il elle contient un meeple si c'est le cas le coup n'était pas valide.
Le parcours du graphe est un parcours en profondeur, comme le montre l'algorithme **??**. À l'aide d'une pile on empile les voisins de la node de départ puis on dépile la dernière node ajoutée et on réalise le traitement souhaité, ici regarder s'il y a un pion posé sur cette node. On continue l'algorithme jusqu'à ce que la pile se vide. La compléxité est en $\Theta(|V|)$ où $|V|$ représente le nombre de nodes.

$stack \leftarrow init\_pile()$;
$visited \leftarrow init\_pile()$;
$add\_pile(stack, n\_check)$;
$add\_pile(visite, n\_check)$;
(! is_empty(pile)) $n\_check \leftarrow dequeue\_pile()$;
is_meeple(n_check) $free\_pile(stack)$;
$free\_pile(visited)$;
$return\ TRUE$;
each neighbour nodes ! (find_pile(visited, neighbour(n_check))) $add\_pile(stack, neighbour(n\_check))$;
$add\_pile(stack, neighbour(n\_check))$;
$free\_pile(stack)$;
$free\_pile(visited)$;
$return\ FALSE$;
each player $player \leftarrow finalise()$; *is_meeple_in_area(struct node *n)*

8 Implémenter une stratégie de jeu pour les joueurs

Carcassonne est un jeu de pose de tuiles où l'objectif est de marquer le plus de points. Chaque joueur est répresenté par une bilbiothèque différente et peut donc avoir sa propre stratégie.

8.1 Les stratégies implémentées par nos joueurs

La stratégie implémentée dans notre version est un joueur qui jouerait d'une manière bien particulière, plus précisement un joueur qui poserait sa carte dès qu'il trouve une position valide. Pour cela, on appelle la fonction *search_valid_position_card*, qui réalise un parcours du tableau contenant les tuiles posées sur le plateau et vérifie à l'aide de *is_connectable* si une carte peut se placer à côté d'autres cartes peu importe son orientation.

tile $\leftarrow$ *init_tile*(*card*, *direction*, *position*);
all tiles in graphall neighbours of this tile *tile* $\leftarrow$ *neighbour_pos*(*tile*) ;
is_empty_tile(neighbour(tile)) and is_connectable(graph, tile) *update_pos*(*move*);
$free(tile)$;
returnmove; $free(move)$;;
returnmove; *struct move search_valid_position_card (struct graph *g, struct move *m)*

Cet algorithme est en $\Theta(nombre\_de\_tuiles^2 * nombre\_de\_voisins)$. La recherche du voisin pour mettre à jour tile pourrait se faire en temps constant si chaque carte posée sur le plateau pointait vers son voisin direct. Il suffirait d'ajouter la carte au graphe et de faire les liaisons $\Theta(1)$ puis d'acceder aux voisins $\Theta(1)$. L'algorithme serait alors en Θ(nombre_de_tuiles * nombre_de_voisins).

La méthode utilisée pour la pose des pions est semblable sauf que cette fois ci on parcours le tableau *allowed_positions* qui nous donne les positions valides pour la pose d'un pion sur chaque carte et lorsqu'il repère une position correcte, il pose le pion.

8.2 Les stratégies implémentables pour une version améliorée

- Méthode 1 : Il serait possible alors d'implémenter une strategie réellement aléatoire. On pourrait par exemple mémoriser dans un tableau toutes les positions valides et choisir l'une de ces positions aléatoirement pour poser la carte et pareillement pour le pion.

- Méthode 2 : Une autre méthode serait quand un joueur qui fait un peu plus attention aux points, c'est à dire qu'il cherche à marquer des points, préfererait une position plutôt qu'une autre. Par exemple toujours en récupérant le tableau des positions valides pour une carte, on peut vérifier pour chaque carte si elle achève une route ou une ville ou poser un pion sur une ville ou une route qui ne comporte pas encore de pions, en utilisant le parcours de graphe décrit précédement. Pour la pose des pions, le joueur pourrait préférablement poser son pion sur une abbaye ou une ville lorsqu'il joue une nouvelle carte.

- Méthode 3 : Une dernière méthode serait un joueur qui essaye de bloquer les coups des autres joueurs par exemple il pourrait agrandir une ville ou une route où un joueur adverse se trouve. Il pourrait également chercher à ne pas poser de carte à coté d'une abbaye adversaire. S'il ne peut pas bloquer un autre joueur adopterait la méthode 1 ou 2.

9 Compter les points des joueurs

Compter les points des joueurs sur Carcassonne est compliqué, il faut parfois les ajouter en cours de partie et parfois en fin de partie.

Pour compter les points des routes il s'agirait de faire un parcours des nodes dans les tuiles en rajoutant un champ indiquant si la node représente une route qui se termine, et de vérifier que dans le graph des routes il y ait bien 2 routes qui se terminent.

Pour compter les points dans une ville la méthode est encore une fois similaire un parcours de graphe à partir d'une node ville. Cependant, il faudra vérifier en plus que la ville n'est pas un trou en son centre et que les bords de la carte soient bien des remparts.

Pour compter les champs, on réalise encore une fois un parcours, mais cette fois-ci il faudrait avoir connaissance des villes qui sont en contact avec le champ, en rajoutant un champ dans la structure node.

Pour compter les abbayes on pourrait rajouter un champ dans la tuile d'une abbaye comptant le nombre de cartes voisines de celle-ci.

10 Conclusion

10.1 Résultats expérimentaux

À ce stade du code, le jeu est prêt à lancer une partie de Carcassonne sans la pose des pions. De nombreuses parties ont été lancées afin de vérifier que le déroulement de chaque partie est cohérent avec les règles énoncées plutôt. Une version avec les pions est disponible sur le dépôt sous le nom de *carcassonne_ meeple* mais n'est pas totalement fonctionnelle. La pose des pions reste encore fausse à certains moments.

10.2 Le code fourni atteint-il l'objectif fixé ?

L'objectif fixé était de fournir un code optimisé et fidèle aux règles du jeu. Les règles sont respectées dans un sens où effectivement l'algorithme ne prend pas en compte les pions, mais le code fourni ne contredit pas les règles établies pour la pose des cartes. Le code proposé a été réalisé dans un souci de clarté et de propreté. Autant de fonctions auxiliaires que possible ont été réalisées, afin de rendre chaque fonction claire, précise, et facile à comprendre. Les noms des variables et des fonctions ont été choisies avec soin, toujours dans le but de proposer le code le plus clair et compréhensible possible. Des commentaires ont été écrits aux endroits clés du code pour des raisons de clareté et de compréhension. De plus, de nombreuses séparations du code en différents fichiers (e.g. *.c, *.h) ont été réalisées, afin de rendre la manipulation du code plus claire, et sa maintenance plus facile. Le codage systématique des fonctions de test permet aussi d'affirmer que le code produit répond aux exigences, ainsi qu'aux règles du jeu, de manière fidèle. Ainsi, même si des optimisations au niveau algorithmique sont envisageables, le code proposé semble en un sens répondre à l'objetif fixé.