BORDEAUX INP Enseirb-Matmeca

PR105 Rapport Projet Carcassonne

Bouyer Kerrian, Genetet Maud,
Dufetrelle Arthur et Larragueta César

Sujet : Implémentation du jeu Carcasonne en C

S6 - Année universitaire 2023-2024

Table des figures

1 Commandes pour lancer le programme 5
2 Exemple d'éxecution 5
3 Représentation d'une tuile 6
4 Représentation de plusieurs tuiles et de leur graphe correspondant 7
5 Fonctionnement du tableau `connected_tiles` 8
6 Différence entre le sujet et notre implémentation 10
7 Pseudo-code de la boucle de jeu 11

Table des matières

1 Introduction **4**
1.1 Modalités . . . 4
1.2 Présentation du projet . . . 4
1.3 Collaboration . . . 4
1.4 Rendu final . . . 5

2 Environnement du jeu **6**
2.1 Stockage de nos tuiles . . . 6
2.2 Création de notre plateau de jeu . . . 7
2.3 Choix des coups possible . . . 7
2.4 Calcul du Score . . . 9
2.5 Serveur de Jeu . . . 10
2.6 Création des joueurs . . . 11
2.7 Stratégie de jeu . . . 12
2.8 Organisation de nos sources . . . 12

3 Expérience acquise **13**
3.1 Utilisation de graphes . . . 13
3.2 Création de bibliothèques partagées . . . 13
3.3 Environnement de travail . . . 13
3.3.1 Valgrind . . . 13
3.3.2 Makefile . . . 14

Conclusion et possibilités d'amélioration futures **15**

1 Introduction

Ce rapport présente la trame de réflexion sur le projet Carcassonne.

1.1 Modalités

Ce projet fait partie du cours de Programmation impérative 2 et développement logiciel de la formation d'ingénieur en Informatique à l'ENSEIRB-MATMECA, encadré par David RENAULT et Georges EYROLLES.
Réalisé en groupe de mars à mai 2024. L'objectif du projet était de produire un code propre et fonctionnel. En parallèle, la rédaction de ce rapport en Latex a été entreprise pour documenter notre démarche et nos réalisations, en vue d'une soutenance de 20 minutes devant nos professeurs à la fin du projet.

1.2 Présentation du projet

Ce projet reprend le jeu de société Carcassonne. Dans ce jeu, l'objectif des joueurs est de marquer des points en construisant des édifices (routes, châteaux, monastères, etc.), en optimisant leurs possibilités en fonction des cartes reçues, en terminant leurs propres constructions voire même en s'appropriant celles de leurs voisins. Pendant le jeu, les deux joueurs alternent pour placer des tuiles sur le plateau, veillant à ce qu'elles se connectent aux tuiles existantes par des bords de même couleur. Ils peuvent également placer un personnage (**meeple**) sur une tuile nouvellement placée, en respectant les règles de placement qui limitent à un seul personnage par tuile et ne permettent pas de placer plus de personnages que ceux disponibles. Si un joueur ne peut pas placer correctement sa tuile, il perd.
Le but général du projet est de développer un système permettant à deux joueurs de participer à une partie de Carcassonne. Ce système se compose de deux éléments principaux :

- **Clients** : Chaque client représente un joueur qui gère son propre plateau de jeu et prend des décisions de manière autonome. Les clients doivent être capables de jouer sans intervention humaine.
- **Serveur de jeu** : Le serveur organise et gère la partie en orchestrant les tours des joueurs. Il envoie les actions des adversaires à chaque client, enregistre les coups joués et informe les clients de la fin de la partie. Le serveur gère également la vérification de l'intégrité des coups joués par les joueurs pour éviter qu'ils trichent.

L'interopérabilité entre les clients et le serveur est assurée par une interface commune non-modifiable, définie dans les fichiers player.h, move.h, et deck.h. Cela garantit que tous les clients peuvent communiquer efficacement avec le serveur et entre eux, indépendamment de leur implémentation spécifique.

1.3 Collaboration

Dans ce projet, nous avons donc dû travailler à quatre. Chacun a pu apporter son expertise et son point de vue, ce qui a véritablement contribué à l'avancement du projet.

Durant les séances de projet en salle de TD, des réunions étaient organisées pour permettre d'évaluer l'avancement du projet et de décider de la répartition des tâches. De plus, le travail en groupe (de 2 ou 4 personnes) était favorisé pour aider à la réflexion sur les

structures de donnée à implémenter. C'est le cas, par exemple, du type de donnée pour la représentation du monde (`world_t`) qui devait permettre le stockage des tuiles jouées, ainsi que d'en avoir une représentation graphique pour calculer les scores.

Ensuite, étant donné que le projet était développé sur un dépôt distant, un gestionnaire de versions comme **Git** a été utilisé. Pour éviter les conflits lors des modifications du code, une bonne organisation était indispensable, il a donc fallu communiquer de manière fluide et efficace. Cependant, cela n'a pas empêché que quelques conflits apparaissent. Mais après une vérification des versions en local, les conflits ont pu être résolus et les branches fusionnées.

1.4 Rendu final

Aujourd'hui, le projet permet la création d'un exécutable ainsi que la création de deux librairies dynamiques. L'exécutable nommé `server` permet le chargement de deux librairies `clients`. Pour ce faire, il suffit d'exécuter les commandes figure 1.

shell bash

```
Première commande: make install
Seconde commande: ./install/server [-m NO_MEEPLE | INFINITE_MEEPLE |
    FINITE_MEEPLE ] ./player1.so ./player2.so
```

FIGURE 1 – Commandes pour lancer le programme

```
seed: 1715894840
Nom du joueur 1: Bilal
Nom du joueur 2: Darkiki
Tour 1
Coups valide!
Tour 2
Coups valide!
Tour 3
Coups valide!
Tour 4
Coups valide!
Le gagnant est le joueur Bilal
Score Darkiki: 0
Score Bilal: 48
```

FIGURE 2 – Exemple d'éxecution

La figure 2 représente l'exécution d'une partie de jeu avec notre serveur. Notre serveur affiche la **seed** du générateur de nombres aléatoires dans un premier temps pour permettre de déboguer plus facilement. Il récupère les noms des joueurs chargés par les fonctions **get_player_name** des bibliothèques dynamiques de nos joueurs. À chaque tour, le coup est validé par le serveur et affiché sur la sortie, indiquant le gagnant de la partie ainsi que le score des deux joueurs.
Nous avons donc pu réaliser dans notre projet notre plateau de jeu en utilisant des graphes

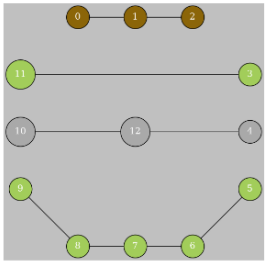
et des tableaux de tuiles. Cette modélisation nous a permis de jouer des parties automatiquement en sélectionnant les meilleurs coups possibles pour chaque joueur.

2 Environnement du jeu

Dans cette section, l'implémentation du jeu sera présentée en détail, en abordant plusieurs aspects clés. Le stockage des tuiles sera expliqué, avec les méthodes et les structures de données utilisées pour organiser et conserver les tuiles jouées. Ensuite, les algorithmes permettant de déterminer les coups possibles pour les joueurs seront décrits. Le calcul du score sera détaillé, en expliquant l'algorithme utilisé pour attribuer des points en fonction des actions (`moves`) des joueurs. L'architecture et les fonctionnalités du serveur de jeu seront également présentées. Les fonctions des joueurs seront examinées, ainsi que les stratégies employées. Les défis rencontrés à chaque étape seront mis en lumière, ainsi que les solutions apportées pour les surmonter.

2.1 Stockage de nos tuiles

Dans ce projet, une tuile est considérée comme un graphe à `MAX_CONNECTIONS` sommets. Ici, `MAX_CONNECTIONS` est égale à 13, 3 sommets par direction cardinal et un sommet au centre (figure 3). L'implémentation imposée par le sujet pour les tuiles est une structure contenant deux tableaux de taille `MAX_CONNECTIONS`, chaque élément du tableau correspond à un sommet de la tuile, le premier tableau représente les connections entre les différents sommets et le deuxième tableau la couleur associée à ces sommets.




```
{
  // The connected components of the tile
  // Here : {0,1,2}, {3,11}, {4,10,12}, {5,6,7,8,9}
  .t={0,0,0,1,2,3,3,3,3,3,2,1,2},
  // The colors of the vertices
  .c={BROWN,BROWN,BROWN,GREEN,GRAY,GREEN,
      GREEN,GREEN,GREEN,GREEN,GRAY,GREEN,GRAY}
}
```

FIGURE 3 – Représentation d'une tuile

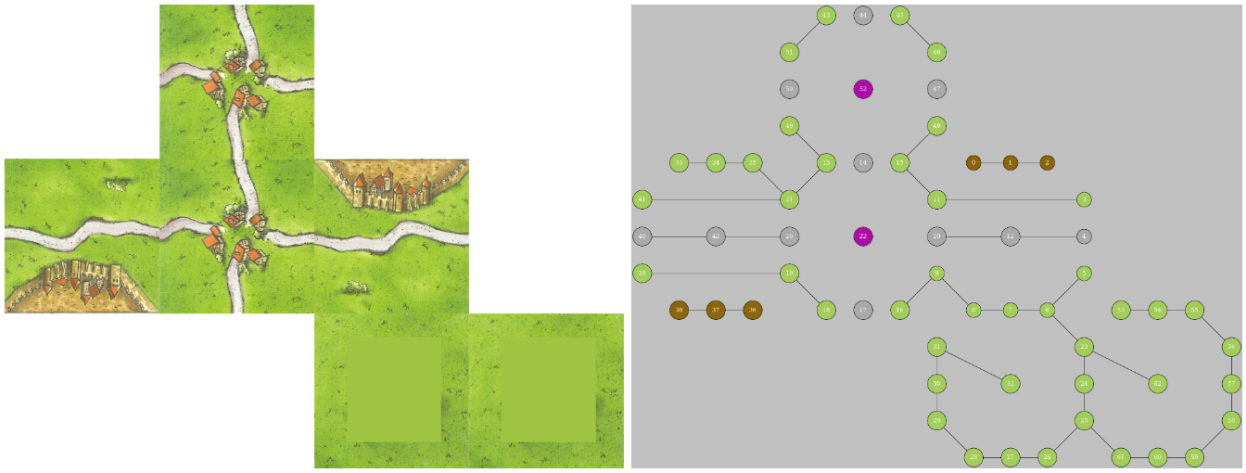


FIGURE 4 – Représentation de plusieurs tuiles et de leur graphe correspondant

Lors d'une partie, le `server` crée et génère une pioche de tuiles. Pour ce faire, le `server` utilise le module `deck`. Lors de la création de la configuration de la partie, la fonction `make_deck` est appelé. Celle-ci permet de générer une pile de tuiles. Cette pile est ensuite stockée dans une variable du `server` et à chaque tour de boucle de la partie, le `server` récupère la dernière tuile pour la transférer au joueur jouant le prochain coup.

2.2 Création de notre plateau de jeu

Pour pouvoir garder les informations sur les tuiles jouées ainsi que leur placement et leur connexité, il a fallu implémenter un structure de donnée nommée `world_t`. Cette structure est composée de trois variables : `tiles`, `graph_board` et `num_placed_tiles`. La première variable `tiles` est un tableau d'éléments de type `placed_tile_t`. Ce type de données est constitué d'une tuile (`tile_t`), de deux coordonnées entières `x` et `y`, ainsi que d'un entier `id`. Ainsi le tableau `tiles` de longueur `MAX_TILES` permet de sauvegarder les informations sur les tuiles posées et donc sur le jeu. La deuxième variable `graph_board` sert à stocker l'information du graphe de la partie. L'utilité du graphe sera détaillée dans la sous-section 2.4. La troisième variable `num_placed_tiles` permet de stocker l'information sur le nombre de tuiles déjà posées dans la partie.

2.3 Choix des coups possible

Que ce soit le `server` ou le `client`, il est important de savoir si un coup est jouable ou non ; s'il respecte les règles ou non. A cet effet, la fonction `is_invalid` permet de détecter si un coup est invalide en regardant si le mouvement (`move`) donné est correct et ne présente aucune triche.

Pour ce faire, la fonction procède de la manière suivante : premièrement, elle regarde si avec les coordonnées du mouvement, la tuile à placer est adjacente à une tuile déjà placée sur le plateau (`tileIsConnected`), si c'est pas le cas alors le mouvement est invalide. Dans le même temps, s'il la position donnée dans le mouvement correspond à une position déjà occupée par une autre tuile, alors le coup est directement compté comme invalide. Si la position permet à la tuile d'être voisine d'une autre tuile, alors la fonction vérifie la couleur des sommets connexes puisque les sommets adjacents ne peuvent être que de la même couleur

(`can_connect`).

La vérification des couleurs se fait à l'aide de la fonction `can_connect_2_tiles` qui récupère le mouvement à tester ainsi que le tableau `connected_tiles` retourner par la fonction `tileIsConnected`. Ce tableau est utilisé de la manière décrite dans la figure 5. Si l'id de la tuile est égale à 0 alors cela signifie qu'il n'y a pas de tuile connexe à la position du mouvement, dans la direction donnée. Sinon la tuile correspondante à l'id est récupérée est les couleurs des sommets à connecter sont comparés.

Id tile :	`tile1.id`	`tile2.id`	`tile3.id`	`tile4.id`
Connection orientation :	North	East	South	West

FIGURE 5 – Fonctionnement du tableau `connected_tiles`

2.4 Calcul du Score

Cette section s'intéresse à comment réussir à évaluer le score d'une partie. Pour rappel, dans le jeu Carcassonne, nous considérons uniquement le score lié au fait que les joueurs terminent une structure.
Terminer une structure dans le cadre de notre projet signifie qu'une composante connexe de sommets de même couleur ne peut plus se voir ajouter d'autres sommets à la composante connexe. Cela est dû au placement des tuiles qui peuvent amener à terminer des structures.

La fonction d'évaluation du score de jeu selon un plateau de jeu doit donc vérifier à chaque action du joueur si le joueur en question a terminé une structure. Si c'est le cas, elle doit déterminer le type de structure terminée et le nombre de sommets de la composante connexe. Ces informations seront utilisées pour associer un score en suivant les règles simples suivantes :

- 4 point par sommet de route
- 8 points par sommet de château
- 1 point par sommet de champ, arrondi à l'entier supérieur.

Dans un premier temps, il faut donc qu'a chaque action d'un joueur, le serveur puisse détecter si une composante connexe de notre graphe se ferme.

Pour cela, le pseudo code suivant montre la logique implémentée pour détecter qu'une structure est fermée.

calcul score

```
fonction calcul_score(world, move):
    Tab_id[] <- get_id_current_tile_move()
    pour tout id de Tab_id :
        connected_id <- get_connected_id(world->graph, id)
        pour tout id de connected_id :
            si (not id_is_closed(id, world)) :
                struct_is_closed <- False
                break
        si (struct_is_closed):
            score += get_score(connected_id)
    retourner score
```

La logique est donc, si une composante connexe est fermée si tous les sommets de la composante sont fermées en considérant qu'un sommet du **graph** est considérer comme fermé en suivant cette logique

id is closed

```
fonction id_is_closed(id, world):
    si (id_is_center_of_tile(id))
        retourner vrai;
    local_id <- get_local_id(id, world)
    direction <- get_direction_from_local_id(id)
    si (id_has_neighbour_in_direction(local_id, direction)) :
        retourner vrai;
    retourner faux
```

Cette logique d'implémentation fonctionne et permet de calculer des scores logiques avec notre implémentation de graphe, cependant deux choses sont critiquables.

Premièrement, la complexité de notre fonction est critiquable. En effet, la fonction de calcul de score est de complexité quadratique en fonction du nombre de sommets de la tuile et de la composante connexe. Dans des cas où la partie contiendrait plus de cartes, le graphe serait donc beaucoup plus grand et la complexité de nos fonctions pourrait devenir un problème, surtout qu'elles sont également utilisées dans la stratégie du joueur.
Afin d'améliorer la complexité de la fonction, une solution serait de maintenir dans le monde un tableau de booléens contenant les informations de si un id est fermé ou non. La valeur du booléen serait mise à jour à chaque ajout de tuile sur le plateau pour garantir le bon fonctionnement de nos fonctions de calcul de score.
Cela permettrait d'accéder en temps constant à l'information de si un sommet est fermé. Cependant, cette approche n'a pas été retenue dans le projet par manque de temps.

Une deuxième critique de cette fonction est qu'elle ne suit pas la même logique de calcul de score que le sujet imposait. En effet, d'après le sujet, quand une tuile est ajoutée au monde, certains sommets se superposent et donc certains sont enlevés du graphe de connexion. Cependant, nous avons choisi une approche différente à l'ajout d'une nouvelle tuile. Dans notre implémentation, à chaque fois qu'une nouvelle tuile est ajoutée au monde de jeu, aucun sommet n'est enlevé, mais un arc est créé entre les sommets qui devraient, d'après le sujet, se superposer.
Cette différence d'implémentation est représentée par la figure 6 :

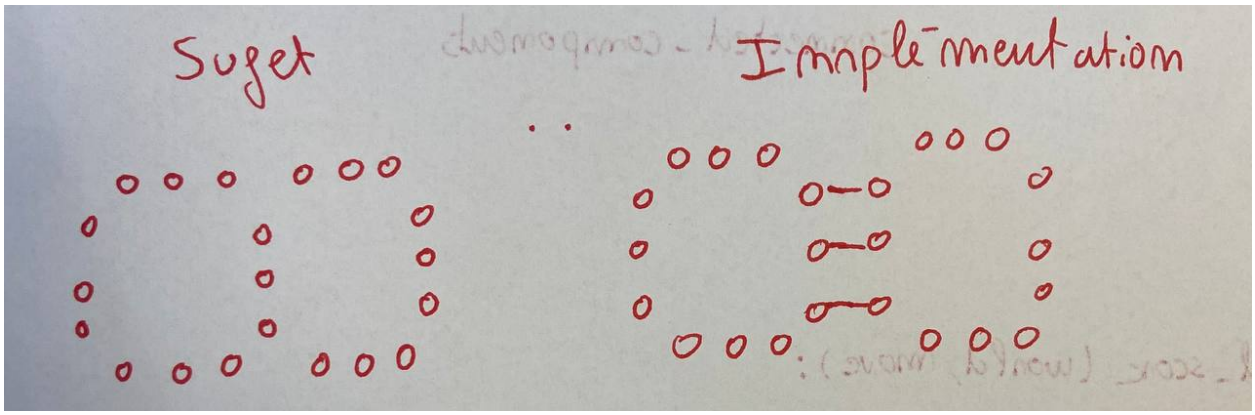


FIGURE 6 – Différence entre le sujet et notre implémentation

Cette différence d'implémentation se traduit concrètement par le fait que le score renvoyé par la fonction ne soit pas égal à celui que le sujet trouverait. Ce problème peut cependant facilement être géré en implémentant une fonction prenant une liste d'ID et renvoyant une liste d'ID de sommets considérés comme différents d'après la première implémentation.

2.5 Serveur de Jeu

L'exécutable `server` permet de générer des parties aléatoires de Carcarsonne entre les deux `clients` précisés sur la ligne de commande (figure 1). Le `server` est une suite d'instructions permettant d'initialiser toutes les variables nécessaires à l'exécution d'une partie, ainsi que la réalisation de celle-ci. Pour cela, l'algorithme du `server` est semblable à celui décrit en figure 7. Dans un premier temps, le `server` récupère les chemins vers les librairies des `clients`. Une fois les chemins récupérés, le `server` charge les librairies à l'aide de `dlopen`. Les `clients` étant implémentés de manière à être interopérable, bibliothèque partagée et

```
joueurs ← chargement_des_joueurs(client1, client2);
partie ← initialisation_partie(mode, nombre_de_tuile);
while Vrai do
    nouveau_joueur ← joueur_suivant(joueurs);
    tuile_a_placer ← prochaine_tuile(partie.pioche);
    mouvement_a_jouer ← nouveau_joueur.jouer(tuile_a_placer);
    if mouvement_a_jouer est invalide then
        return joueur_suivant(joueurs);
    mise_a_jour(partie);
    if longueur(partie.pioche) == 0 then
        return vainqueur(joueurs)
```

FIGURE 7 – Pseudo-code de la boucle de jeu

changeable de manière dynamique, le `server` peut donc charger les `clients` et utiliser les fonctionnalités de chacun.

Le `server` gère aussi l'initialisation de la partie. Pour ce faire, le serveur récupère le mode de jeu entrer en option (figure 1) et génère la configuration de la partie avec la fonction `make_gameconfig`. Ensuite, le `server` positionne la première tuile avant dans lancer la boucle de jeu dans laquelle il récupère le mouvement décidé par la fonction `play` du joueur considéré. Chaque mouvement doit être vérifié par le `server`, c'est pourquoi la fonction `is_invalid`, présentée en section 2.3, est appelée avec le mouvement du joueur, si le coup n'est pas validé le joueur est déclaré perdant.

Une fois le coup validé, le `server` doit vérifier l'état de la partie. Si la pile de pioche est vide, le `server` compare les scores des deux joueurs afin de déterminer le vainqueur de la partie. Si la pioche n'est pas vide, le `server` s'assure que le prochain joueur peut jouer la tuile suivante. Dans le cas contraire, si la tuile n'est pas jouable, le joueur suivant est déclaré perdant.

2.6 Création des joueurs

Dans la structure client / serveur du projet, les joueurs sont caractérisés par 4 fonctions : **get_player_name()**, **initialize()**, **play()** et **finalize**. Ce sont les fonctions qui lui permettent de communiquer avec le serveur. Afin de rendre tous les joueurs de toutes les équipes travaillant sur ce projet interopérable entre eux, l'interface commune à tous les joueurs que constituent les 4 fonctions précédentes doit être compilée sous la forme d'une bibliothèque partagée (**fichier.so**) chargeable dynamiquement par le serveur.
L'interface de communication entre le client et le serveur ne permet pas la transmission de l'état du plateau de jeu. Une première difficulté réside donc dans le fait que le joueur a besoin de garder à jour une représentation du plateau de son coté afin de suivre la partie. Cette difficulté a été surmontée en utilisant les mêmes structures de données qu'utilise le serveur (*world*), et en l'actualisant à chaque fois que le serveur communique au joueur le dernier move qui a été joué.
En retour, le client renvoie au serveur le **move** qu'il a choisi de jouer suivant la stratégie que

l'on décrira dans la prochaine partie.

2.7 Stratégie de jeu

Une partie du projet visait à implémenter, dans un second temps, une fonction de jeu permettant au joueur de jouer de manière "optimale". L'idée retenue par notre équipe pour jouer de manière optimale est un algorithme "glouton" qui énumère l'ensemble des positions où la tuile courante peut être jouée. On ajoute à cet ensemble de positions l'ensemble des positions où la tuile peut être jouée en la retournant. Ces ensembles de positions associés à une tuile (tournée ou non) sont ensuite stockés dans un tableau de coups. Chaque coup est évalué selon une fonction prenant en argument un monde et un coup, et en calculant le score associé au fait de jouer ce coup. Cette fonction de calcul de score utilisée est la même que celle utilisée par le serveur pour calculer des scores en détectant les fermetures de structures.

Cette première idée de fonction heuristique nous permet de jouer et de gagner contre un joueur jouant aléatoirement. Cependant, cette approche peut être critiquée pour deux raisons. Premièrement, la fonction de calcul de score de notre serveur n'est pas la même que celle utilisée par le serveur contre lequel nos joueurs jouent.

Deuxièmement, cette approche peut être critiquable dans certains jeux de plateau, car choisir un coup maximisant le score à l'instant t+1 ne garantit pas que le score à la fin de la partie soit optimal. Cependant, dans ce jeu, cette première approximation permet d'avoir un joueur que l'on peut considérer comme plutôt bon car meilleur qu'un joueur aléatoire.

2.8 Organisation de nos sources

Le projet est organisé de manière à séparer les différentes parties du code et à faciliter le développement. En voici la structure :

- **deck.c, deck.h** : Les fichiers source et en-tête du module `deck` qui gère la création de la pioche de tuiles.
- **move.h** : Le fichier en-tête qui gère la création des structures du projet.
- **board.c, board.h** : Les fichiers source et en-tête du module `board` qui implémente la structure de données `world_t` pour gérer les informations sur les tuiles jouées.
- **player1/2.c, player.h** : Les fichiers source et en-tête du module `player` qui implémente les joueurs du jeu avec leurs fonctionnalités.
- **server.c, server.h** : Les fichiers source et en-tête du module `server` qui gère le serveur de jeu, le déroulement du jeu et les règles du jeu.
- **server/player_utils.c, server/player_utils.h** : Les fichiers source et en-tête du module `server/player_utils` qui contient des fonctions utilitaires utilisées dans le projet.

Cette organisation permet de travailler de manière modulaire et de faciliter la collaboration entre les membres de l'équipe. Chaque module est responsable d'une partie spécifique du jeu, ce qui rend le code plus lisible et maintenable. De plus, l'utilisation de bibliothèques partagées permet de changer facilement l'implémentation des joueurs sans avoir à recompiler tout le projet.

3 Expérience acquise

Dans cette section, seront abordées les compétences acquises et les enseignements tirés de ce projet.

3.1 Utilisation de graphes

Dans le cadre de ce projet Carcassonne, la bibliothèque *igraph* est utilisée pour manipuler les structures de graphes qui représentent les tuiles et le plateau de jeu. Ainsi, après avoir installé la bibliothèque et compilé en prenant soin de spécifier le chemin de cette dernière, ses fonctions peuvent être utilisées pour :

- Crée des tuiles : Une tuile est un graphe à 16 sommets.
- Manipuler le plateau de jeu : Le plateau est représenté en machine par un graphe qui croit au fur et à mesure qu'on y ajoute des tuiles.
- Calculer le score : le score d'un plateau se calcule accédant à des informations sur le graphe qui le représente telles le nombre de composantes connexes du graphe notamment. *igraph* fournit des fonctions nécessaires à la réalisation de telles opérations sur les graphes.

Par conséquent, les enseignements du cours d'algorithmique des graphes ont pu être mis en pratique sur des problématiques concrètes.

3.2 Création de bibliothèques partagées

Nous avons également eu recours à l'utilisation de bibliothèques partagées pour le développement de nos joueurs. Ces bibliothèques ont été d'une grande utilité, car elles nous ont permis de jouer avec un serveur implémenté par nos professeurs. Ce serveur orchestrait ensuite les parties en faisant jouer nos joueurs contre ceux des autres groupes de travail.

L'avantage majeur de cette approche réside dans la facilité de changement de code. En effet, en utilisant des bibliothèques partagées et en intégrant nos joueurs dans un environnement standardisé, nous avons pu modifier et améliorer nos stratégies de jeu de manière aisée et rapide, sans compromettre la compatibilité avec le serveur et les joueurs des autres équipes. Ce projet a donc été l'occasion de se familiariser avec le chargement dynamique de bibliothèques partagées.

3.3 Environnement de travail

Dans cette partie, nous allons parler de notre utilisation de **Valgrind** et du **MakeFile**.

3.3.1 Valgrind

Valgrind nous a été d'une grande aide pour détecter et corriger les fuites de mémoire dans notre projet. Grâce à cet outil, nous avons pu identifier les zones problématiques de notre code et les rectifier, ce qui a grandement amélioré la stabilité et la fiabilité de notre application.

Cependant, l'utilisation de **Valgrind** n'a pas été sans difficultés, notamment en ce qui concerne la bibliothèque **iGraph**. Nous avons souvent rencontré des problèmes générés par

iGraph que **Valgrind** signalait, mais qui étaient difficiles à résoudre. Ces défis ont parfois ralenti notre progression, car il n'était pas toujours évident de distinguer les erreurs commises du à **iGraph**. Malgré ces obstacles, **Valgrind** a joué un rôle crucial dans le maintien de la qualité de notre code.

3.3.2 Makefile

Le **Makefile** joue un rôle crucial dans la compilation du projet de Carcassonne, qui suit une architecture `client`/`serveur`. Il contient des règles spécifiques pour compiler différentes parties du projet de manière séparée. Il nous permet par exemple de compiler le fichier `player1.c` en une bibliothèque partagée `libplayer1.so` en spécifiant les dépendances nécessaires et les options de compilation appropriées. De plus, il offre la possibilité de compiler les tests unitaires de manière distincte pour assurer la qualité du code, facilitant ainsi le processus de développement et de maintenance du jeu des amazones.

Conclusion et possibilités d'améliorations futures

Bien que les résultats obtenus soient satisfaisants, il apparaît plusieurs voies d'amélioration pour le projet. Tout d'abord, il serait possible d'explorer d'autres stratégies pour les joueurs afin d'améliorer nos performances.
De plus, nous pourrions nous focaliser sur la partie graphique des graphes en implémentant des fonctions de visualisation. Bien que nous ayons commencé cette partie durant le projet, il serait bénéfique de continuer sur cette voie pour créer une version graphique complète de notre jeu.

En somme, ce projet a été une expérience enrichissante qui nous a permis d'appliquer certaines connaissances en programmation à un problème concret et intéressant.