ENSEIRB
MATMECA
BORDEAUX

Projet programmation *C* : Carcassonne

Nicolas Bouzat, Jérémie Foucault, Ismail Moumine,
Guillaume Simon, Leo Villalba

Encadrants : Emmanuel Jeannot, Corentin Travers
Première année, filière informatique.
Bordeaux, le 6 mai 2013

Table des matières

Introduction

Le projet Carcassonne consiste à réaliser, en langage C, le jeu de société du même nom. Dans ce jeu, plusieurs joueurs essaient de construire des édifices (routes ou villes) à l'aide de cartes représentant des parties des dits édifices. Chaque joueur peut gagner des points en posant l'un de ses pions sur un édifice en cours de construction ; le gain sera calculé en fonction de la taille de l'édifice final, si tant est que celui-ci soit complété. La contrainte principale à la pose de ces cartes est le respect des tuiles déjà placées : routes et villes ne peuvent être abruptement coupées par des champs, par exemple.

L'objectif est de proposer une modélisation algorithmique des règles du jeu à l'aide de graphes, puis de les implémenter en langage C. L'idée est de posséder une architecture permettant de représenter visuellement le plateau de jeu, de déterminer les coups possibles à chaque tour, de calculer le score de chaque joueur ce qui permet à deux joueurs de jouer l'un contre l'autre, voire à un joueur de jouer contre une intelligence artificielle.

Dans ce rapport, nous allons dans un premier temps définir les problèmes à résoudre pour la modélisation du jeu. Dans un second temps, nous expliciterons en détail l'architecture du programme ; puis dans un troisième temps, détaillerons la réalisation et l'implémentation des algorithmes utilisés pour modéliser le jeu. Enfin, nous aborderons la réalisation de l'interface graphique, à l'aide de la bibliothèque *SDL*.

1 Définition du problème

1.1 Présentation du jeu

Carcassonne est un jeu de placement tactique dans lequel il faut construire un paysage médiéval composé de champs, routes et villes aux remparts fortifiés tels ceux de la ville éponyme. Le jeu peut se jouer jusqu'à 4 joueurs ; néanmoins, seuls deux joueurs peuvent jouer, l'un contre l'autre, dans la version que nous proposons aujourd'hui.

Le jeu commence avec une seule carte (ou, comme on les nommera dans le reste de ce rapport, une *tuile*) posée sur le plateau, les autres étant cachées dans la pioche. Chacun son tour, les joueurs piochent une tuile et tentent de la placer, en respectant les tuiles déjà placées : les villes et les routes ne peuvent être coupées. Chaque joueur peut placer l'un de ses 10 pions, ou partisans, sur un morceau de terrain de la tuile qu'il vient de poser. Il gagnera des points lorsque le-dit terrain, qu'il soit ville ou route, sera fermé, c'est-à-dire entièrement représenté sur le plateau de jeu. Plus le terrain est grand, plus le nombre de points sera élevé. À la fin de la partie, le vainqueur sera le joueur ayant gagné le plus de points.

FIGURE 1 – *Position Permise*

FIGURE 2 – *Position non Permise*

1.2 Déroulement du jeu

Chaque joueur démarre avec 10 partisans, après avoir placé sa tuile, et uniquement à ce moment-là, le joueur peut s'il le souhaite placer un pion sur une des parties de cette tuile (morceaux de villes ou de champs, tronçons de chemins, abbayes). La ville, le champ ou le chemin formé par les éléments contigus devient 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 y a le plus de pions qui devient le propriétaire de l'ensemble (si les joueurs sont à égalité, le terrain appartient autant à chacun).

Quand une route ou une ville est complétée, ses propriétaires comptent leurs points, 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 champs, et pour les routes et les villes non complétées.

2 Architecture du programme

L'exécution du programme *Carcassonne* est détaillée figure 4. Elle est régie par l'affichage et notamment par la bibliothèque *SDL*. Le fonctionnement est le suivant : la librairie multimédia *SDL* est tout d'abord initialisée ; elle ne charge dans un premier temps que les images du menu, qu'elle affiche immédiatement. Ceci est codé dans le fichier main.c. Une fois que le joueur accepte de commencer une partie, le programme initialise toute la mémoire utilisé pour la gestion du plateau de jeu et des tuiles. Cette mémoire sera libérée à la fin de l'execution par appel des fonctions dédiées. Le programme charge notamment les caractéristiques de chaque tuile depuis un fichier texte tuiles.txt, à l'aide du parseur dont le fonctionnement est détaillé dans la section 3.6.1.

Le jeu rentre ensuite dans une boucle "infinie", définie dans jeu.c, au cours de laquelle *SDL* attend que le joueur presse une touche, pour déplacer le curseur ou pour valider la position de celui-ci. Selon l'action, on fait alors appel aux différentes fonctions qui gèrent les graphes du plateau de jeu, dans les fichiers plateau.c, graphes.c et regles.c. Le résultat de ces fonctions modifie alors l'affichage, qui est rafraîchi par un appel aux fonctions du fichier affichage.c.

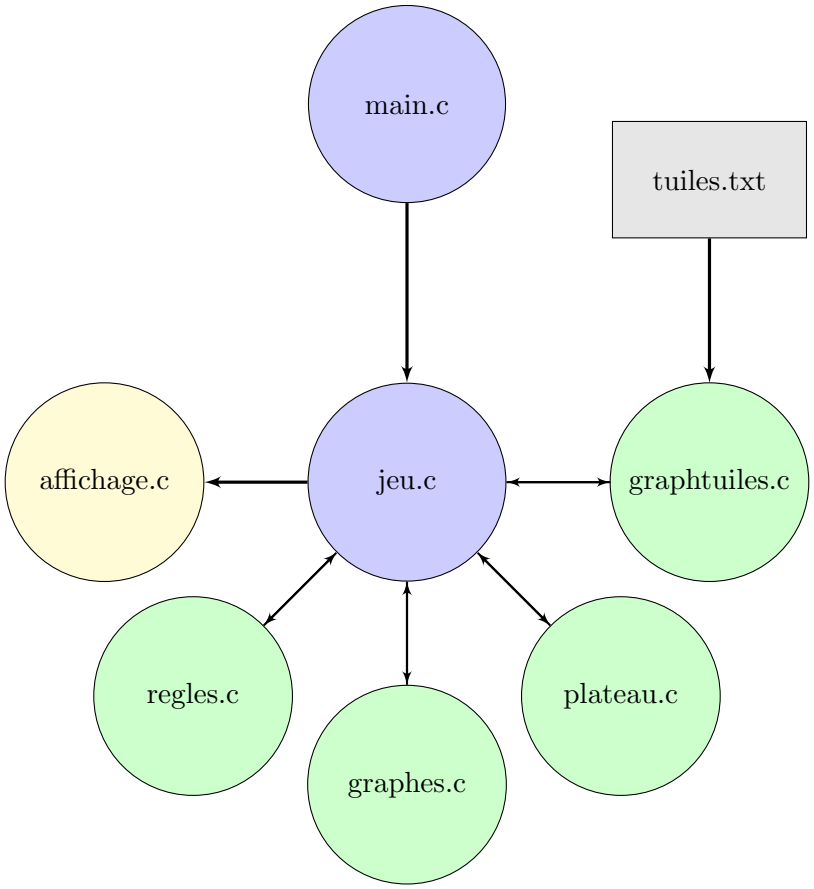


FIGURE 3 – Architecture des fichiers du programme

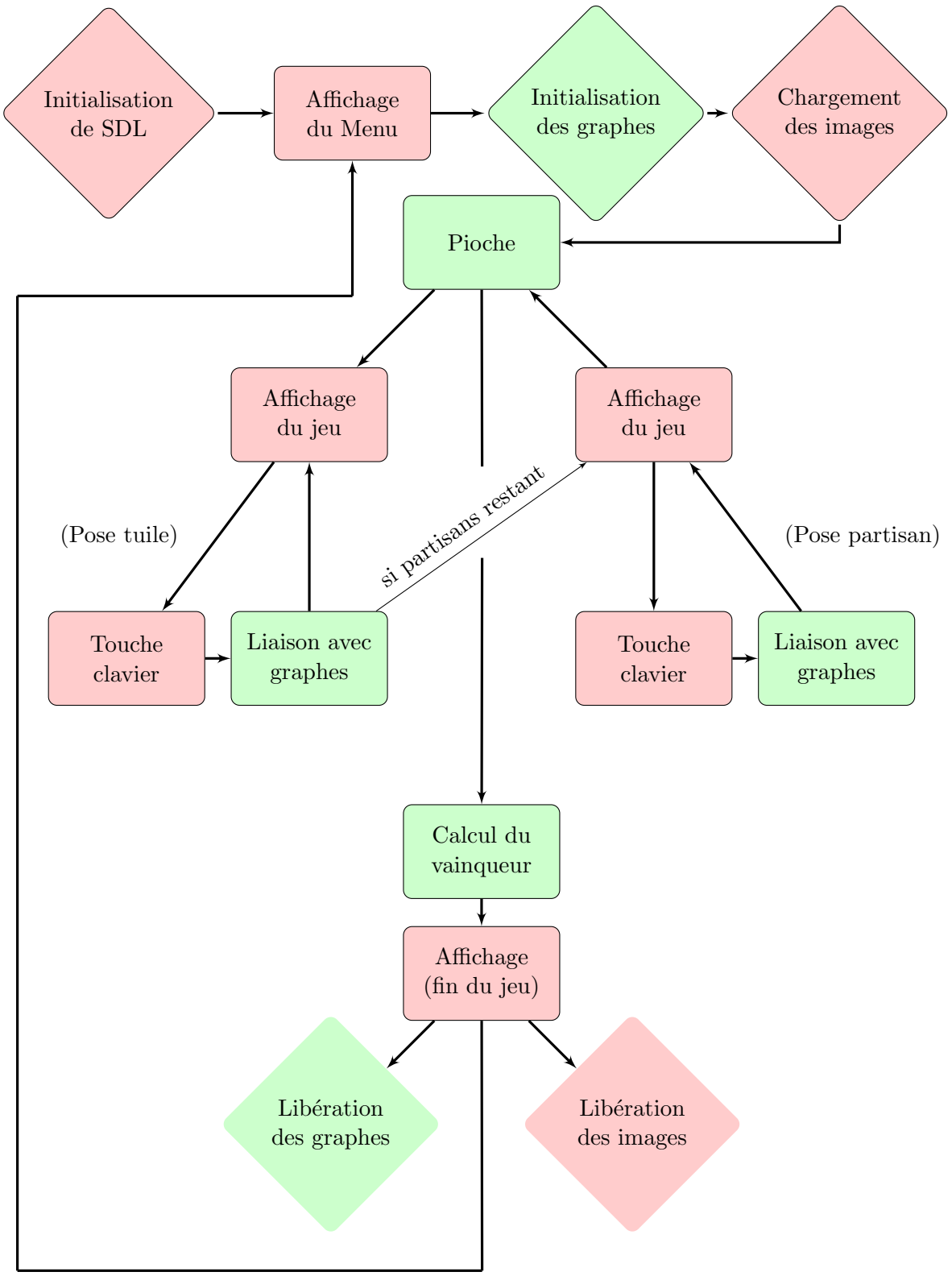


FIGURE 4 – Architecture des fonctions du programe

3 Réalisation algorithmique et implémentation des fonctions logique

3.1 Réalisation des tuiles

3.1.1 Réalisation algorithmique

La modélisation formelle du jeu commence par l'élément principal constituant le jeu ; les tuiles. On compte parmi toute les tuiles 4 types d'éléments :

- les champs
- les villes
- les routes
- les abbayes ou monastères

Chacune des 24 tuiles du jeu est une combinaison unique d'un ou plusieurs de ces éléments. La première étape à donc été de choisir une modélisation adaptée pour les représenter ; le modèle des graphes a été choisit. Ainsi chaque tuile est représentée par un graphe. Il s'agit d'un graphe à au plus 8 sommets, la répartition de ces sommets est donnée figure 5 et est liée à la forme carré de la carte.

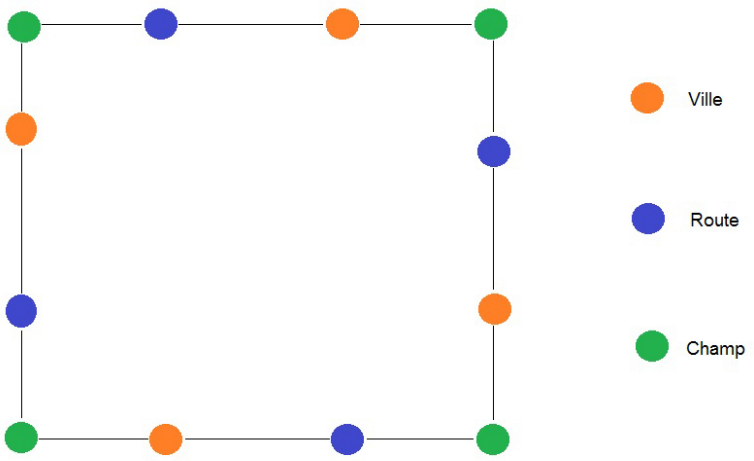


FIGURE 5 – Sommets du graphe d'une tuile

De plus elle est basée sur les faits suivant :

Fait 1 : il ne peut y avoir une route et une ville sur un même coté de tuile.

Fait 2 : il ne peut y avoir de route sans champ.

Fait 3 : il ne peut y avoir que des champs dans une tuile, au minimum une abbaye sera présente.

Fait 4 : la forme triangulaire des villes implique pour notre représentation qu'un champ et une ville puisse être activé sur la même arête d'une tuile.

Partant de ces constats, le choix de n'opter que pour 4 sommets aurait pu être fait, ce qui aurait impliqué que l'absence d'information aurait signifié la présence d'un champ. Or comme nous le verrons dans la partie 3.4.1 des informations sur les champs nous ont semblé nécessaire pour la gestion des partisans. En résumé chaque tuile compte au plus 8 sommets qui représentent un des types d'élément constituant une tuile.

Après avoir défini se que représente un sommet, nous pouvons définir le rôle d'une arête. Les arêtes ne sont pas orientées et peuvent seulement relier deux sommet de même type (route - route, ville - ville, champ - champ). Ces arêtes définissent les limites, les contours, de chaque éléments de la tuile. Ainsi par exemple deux sommets route reliés définirons une route, quand un sommet isolé représentera le début ou la fin d'une route.

Il demeure le cas particulier des abbayes, elles sont représentées par un sommet unique sans aucune arête.

3.1.2 Implémentation en *C*

Comme nous l'avons vu précédemment, les tuiles sont représentées par de petit graphe. Pour implémenter ces graphes, un tableau de sommets et une matrice de booléen est utilisé. Ils sont définis dans la structure `graphe` où chacun des types de sommets est séparé ainsi on a trois tableaux de sommets : `champs`, `villes`, `routes` qui contiennent des pointeurs vers des structures `sommets` qui elles même possèdent plusieurs champs :

- l'entier `actif` qui donne le statut du sommet, car toute les tuiles n'ont pas forcement tout leur sommets activés.
- l'entier `pion` qui indique la présence du partisans d'un des joueurs sur ce sommet.
- l'entier `parcourt_graphe` qui permet la gestion du parcourt du graphe global détaillé section 3.2.1.
- l'entier `indice_sommet` qui permet de repérer la position du sommet dans le graphe global.
- l'entier `fin_element` qui indique si le sommet permet de fermer une ville ou une route.
- l'entier `type` qui permet de typer le sommet en indiquant s'il s'agit d'un sommet ville, route ou champ.

Ce graphe est dans un structure `tuile` qui contient en plus, des champs permettant la gestion de l'affichage ainsi qu'un booléen indiquant une carte spécifique avec un emblème et donc rapportant potentiellement plus de points.

3.2 Concrétisation du plateau de jeu

3.2.1 Représentation algorithmique du plateau de jeu

D'un point de vue algorithmique le plateau de jeu est représenté par un graphe que l'on nommera *graphe global*. Ce graphe est la réunion des sous-graphes constituant chaque

tuiles. Il suffit d'ajouter des arêtes entre les sommets de deux tuiles compatibles et adjacent en respectant les faits suivant :

Fait 1 : l'arête ville [1] de deux tuiles différentes est prioritaire sur les arêtes champs qui sont alors interdites.

Fait 2 : le nombre d'arêtes entre deux tuiles est compris entre 1 et 3.

Fait 3 : Entre deux tuiles, une arête route sera accompagnée de deux arêtes champ.

Ainsi un plateau de jeu avec trois tuiles posée aura pour graphe celui de la figure 7. La construction de se graphe se fait au fur et à mesure du jeu.

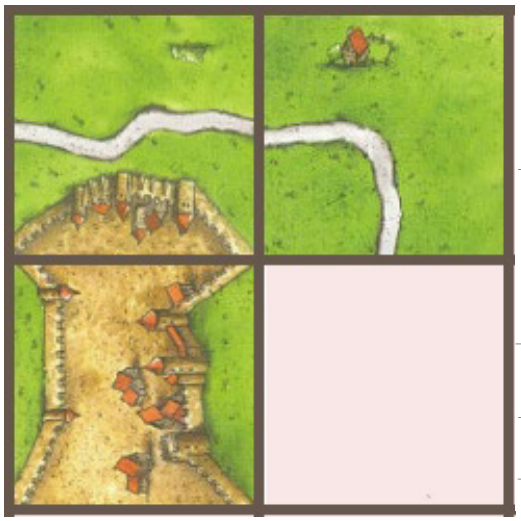


FIGURE 6 – Exemple de 3 tuiles

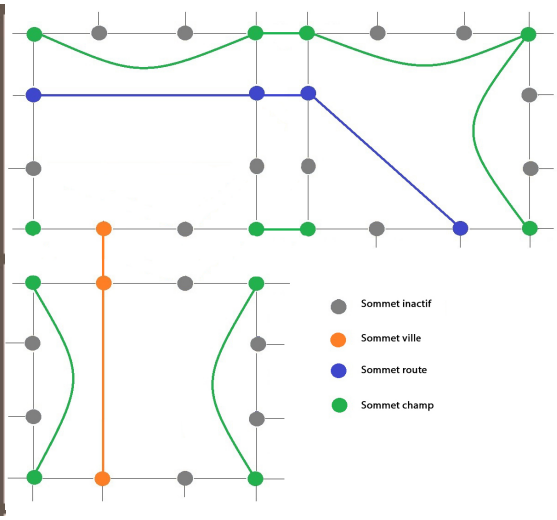


FIGURE 7 – Graphe correspondant au 3 tuiles

D'un point de vu complexité, si l'on considère que le graphe utilise une structure de type tableau de liste de successeur, l'ajout d'un sommet se fait en temps constant, l'ajout d'une arête se fait dans le pire des cas en $\Theta(m)$ où $m = |E|$.

Lors de l'ajout d'une tuile au graphe global, il faut donc au pire ajouter autant d'arêtes qu'il y a de sommets. Dans notre cas le nombre de sommets d'une tuile vaut 12 mais nous généraliserons ce nombre de sommet d'une tuile à la quantité n_{sommet}. La complexité de cet ajout est donc $\Theta(n_{sommet} \times m)$.

3.2.2 Implémentation de la solution algorithmique

L'implémentation du plateau de jeu est réalisée de deux manières différentes mais complémentaires dans la gestion et l'application général des règles du jeu.

Tout d'abord le plateau est représenté par une matrice allouée dynamiquement et de

1. Une arête route, ville, champ, est une arête reliant deux sommets de type respectivement route, ville, champ

taille 145×145, ce qui représente le nombre maximal de tuiles que l'on pourrait aligner dans une seule direction en partant du centre. Afin d'améliorer la complexité en espace il aurait été judicieux d'utiliser une matrice à taille dynamique qui aurait été redimensionnée à chaque ajout de tuiles. Toute la mémoire allouée est évidement libérée à la fin de la partie par appel de fonctions qui y sont dédiées. La matrice que nous utilisons contient des pointeurs vers les tuiles qui occupent la case. Dans le cas où aucune tuile n'est placée à cet endroit, la case contient un pointeur vers *NULL*. Cette matrice est particulièrement adaptée au cas de tuiles carrées et à une surface de jeu plane. Elle serait à reconsidérer dans des conditions de jeu différentes mais permet ici de gérer facilement la pose des tuiles comme il est détaillé dans la section 3.3

Néanmoins la gestion des partisans et le comptage des points doit également être réalisé. Dans cette optique le plateau est également représenté par un graphe. Ce graphe utilise un tableau de successeurs représentés par une liste chainée de structure *sommet_successeurs* contenant deux champs :

- Un pointeur vers la structure `sommets` du sommet en cours
- Un entier indiquant le type de l'arête entre le sommet source et le sommet en cours. Il peut s'agir d'une arrête *interne* à la tuile ou *externe* entre deux tuiles.
- Un pointeur vers le successeur suivant.

Ce graphe est mis à jour à chaque tour une fois qu'une tuile à été validée puis posée. Les sommets et arêtes internes à la tuile sont alors ajoutés au graphe global par réallocation de la mémoire du tableau de sommets et du tableau de successeurs. Enfin les arêtes entre cette nouvelle tuile et le graphe existant sont ajoutés. Lors de cette mise à jour du graphe les pointeurs des sommets contenus dans la tuile sont ajoutés au graphe global. Ce choix est voulu est permet ainsi de conserver une correspondance entre le plateau sous forme de matrice et celui sous forme de graphe. De plus, même si l'impact est limité, la complexité en espace est dans ce cas plus faible que la solution consistant à doubler chaque sommet.

3.3 Pose des tuiles

3.3.1 Algorithme de pose

À tours de rôle le joueur pioche une tuile qu'il doit poser sur le plateau de jeu en respectant les règles qui ont été définies dans la section 1.1. Partant de se principe il est alors nécessaire de vérifier qu'une tuile posée par un joueur respecte bien les règles avant de valider la pose. Pour se faire nous utilisons l'implémentation de notre plateaux de jeux par une matrice de graphe. Ainsi il suffit de vérifier pour chacune des tuiles adjacentes aux cotés de celle qui va être posée que les sommets voisins sont compatibles comme le montre la figure 8. Pour vérifier la compatibilité de deux sommets, nous avons considéré les sommets villes comme étant les plus important et donc que la seule activation de deux sommets villes sur les deux tuiles suffit à valider le coté. Viens ensuite les routes qui doivent respecter les même conditions et enfin par défaut il y aura toujours un champ. En résumé ceci revient à vérifier que deux sommets adjacents ne sont pas d'un type différent.

Cette vérification s'effectue en temps linéaire en le nombre de sommets. C'est à dire dans le pire des cas, ici, 12 sommets. On peut donc considérer que pour notre application cette vérification se fait en temps constant.

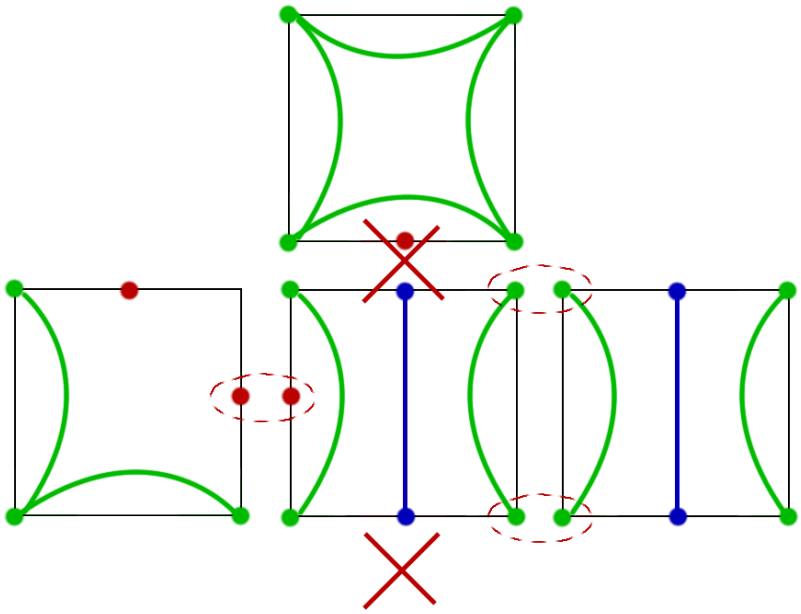


FIGURE 8 – La tuile du centre est compatible avec celles de gauche et droite, mais ne l'est pas avec celle du haut. Elle ne peut pas être posée.

3.3.2 Implémentation de la vérification

D'un point de vu purement algorithmique la solution détaillée précédemment est relativement simple. Cependant son implémentation s'avère plus ardue dans la mesure où de nombreux cas sont à gérer. En effet, il faut détecter la position des cartes voisines pour s'assurer de ne pas faire de tests sur le pointeur *NULL*. Nous parcourons donc les quatre cases, de la matrice adjacentes, à la position de la tuiles souhaité et enregistrons les tuiles présentes. De plus afin d'améliorer la robustesse de notre programme, les cas limites sont testés, i.e, les 4 bords de notre matrice. Ensuite pour chacune des positions détectée, il suffit de vérifier que les sommets, activés, des deux tuiles correspondent. Enfin, afin de tester séparément les sommets route des sommets ville, le choix de réaliser deux fonctions a été retenu. Cependant après réalisation de l'ensemble du projet, la distinction de ces deux fonctions ne semble pas si important et une fonction générique avec un caractère indiquant le type de sommet à tester aurait probablement été plus judicieux.

3.4 Pose des partisans

3.4.1 Modèle de vérification de la pose des partisans

Les partisans constituent un rôle majeur dans ce jeu. En effet ils sont à l'origine du comptage des points et permettent donc de désigner le vainqueur d'une partie. Il est donc nécessaire de gérer la pose de ces partisans suivant les règles du jeu. Nous avons déjà rappelé que chaque joueur possède un nombre limité de partisans, 10, qu'ils doivent poser dans l'une des zone de la tuile qu'ils viennent de placer sur le plateau.

Dans notre jeu, la gestion des partisans et le contrôle de la pose est réalisé grâce à notre graphe représentant le plateau de jeu. Il suffit alors de parcourir tous les sommets connexes

à un sommet de départ, qui correspond au sommet où l'on souhaite placer le partisans, afin de déterminer si un autre partisans occupe déjà la zone et auquel cas refuser la pose. Nous utilisons le parcourt en profondeur de l'algorithme 1. De plus afin d'améliorer la généricité de nos fonctions, ce parcourt générique exécutera sur chaque sommet une fonction donnée en paramètre. Ici la fonction à exécuter est *est-pion* et doit retourner *FAUX*.

Concernant la complexité, cet algorithme effectue un parcourt du graphe qui se fait en temps linéaire en le nombre de sommet et applique pour pour chacun de ces sommets une fonction qui se fait en temps constant. La complexité de cet algorithme est donc dans le pire des cas en $\Theta(m)$ où $m = |E_{graphe}|$, cependant en pratique tous les sommets ne sont pas connexe et ne sont donc pas tous parcourus.

```
   Entrées : G :graphe, som :sommet, fct_exec :fonctions à exécuter,
             resultat_fct :entier
   Sorties : Booléen
 1 som.couleur ← gris;
 2 si ¬a_succ(som) alors
 3 |  retourner Succes
 4 si fct_exec(som) ≠ resultat_fct alors
 5 |  retourner Echec
 6 pour chaque successeur succ de som faire
 7 |  si est_blanc(succ) alors
 8 |  |  si fct_exec(som) ≠ resultat_fct ou
 9 |  |  parcourt_graphe(G, succ, fct_exec, resultat_fct) = Echec alors
10 |  |  |  retourner Echec
11 som.couleur ← noir;
12 retourner Succes
```

Algorithme 1: Algorithme de parcourt en profondeur d'un graphe

3.4.2 Réalisation des fonctions

La première étape est la réalisation de la fonction de parcourt du graphe, qui sera réutilisée dans la section 3.5 pour le comptage des points. Il est donc nécessaire que celle-ci soit générique. Pour se faire deux tableaux ont été utilisés :

- un tableau de pointeur de fonctions, qui seront exécutées par ordre de rangement sur chacun des sommets lors du parcourt
- un tableau d'entiers correspondant au retour de fonction attendu pour chacune des fonctions du tableau précédent.

L'implémentation de la fonction se fait suivant l'algorithme 1 et le repère de sommets, déjà, ou, en cours de visite est réalisé grâce au champ `parcourt_graphe` de la structure *sommet*. Ce marquage des sommets parcourus implique donc une initialisation du champ de la structure pour tout les sommets avant d'effectuer un nouveau parcourt. De plus trouver l'ensemble des successeurs d'un sommet se fait par parcourt de sa liste chaînée de structures `sommet_successeurs`.

La fonction *est-pion* prend un sommet en paramètre et vérifie que ni le partisan du

joueur bleu ni celui du joueur rouge n'y est posé. Cette fonction incrémente en réalité la variable `nb_pions` dont le pointeur est donné en paramètre et qui correspond au nombre de pions rencontrés lors du parcourt. Cette variable est incrémenté de la façon suivante :

+1 pour un partisan du joueur rouge
+10 pour un partisan du joueur bleu

Afin de décider si la pose d'un partisan est possible il suffit après parcourt de vérifier que la valeur de `nb_pions` est nulle. On activera alors le champ `pion` du sommet avec la valeur du partisan correspondant au joueur et définie par les constantes `PARTISAN_BLEU` et `PARTISAN_ROUGE`.

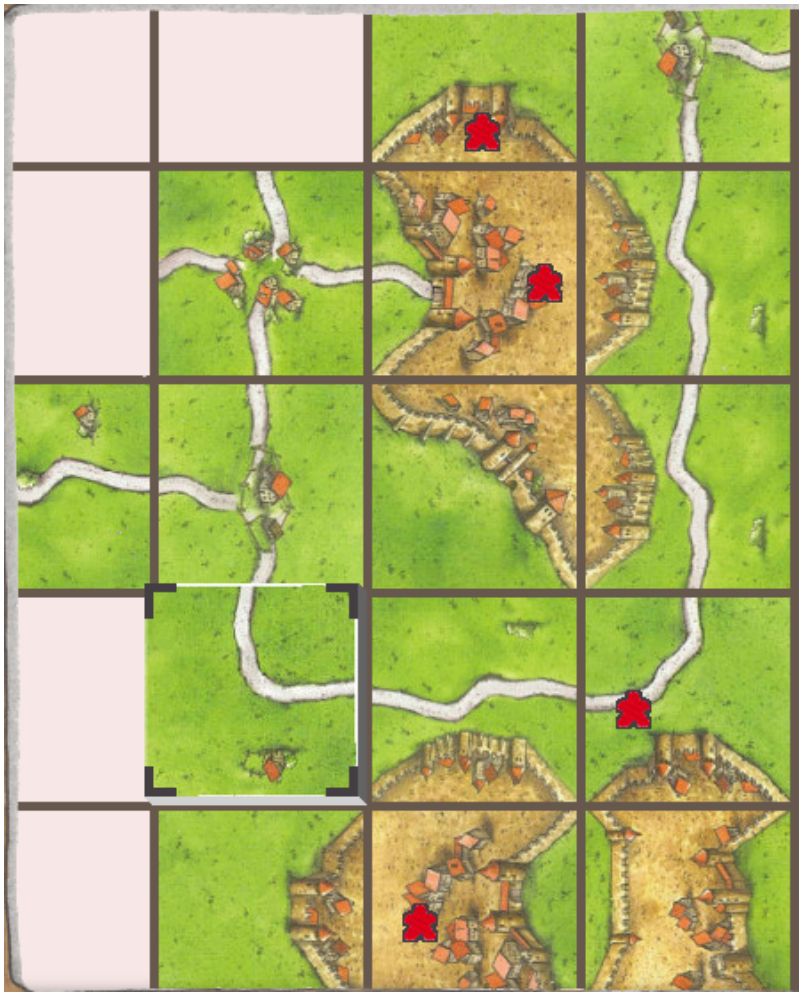


FIGURE 9 – Plateau valide avec partisans

3.5 Calcul des points

Le comptage des points se fait en deux étapes : au fur et à mesure du jeu, lorsqu'une construction de type ville ou route est terminée le nombre de points qui y est associée est

calculé. Il est fonction du nombre de tuile utilisées pour la réalisation et les points sont attribuées au joueur ayant le plus de partisans sur cette construction. Dans le cas d'un nombre de partisans égal la moitié des points est attribuée à chacun des deux joueurs. Dans un second temps à la fin du jeu, les constructions de type champ sont évaluées. Des points sont attribués au joueur possédant un ou plusieurs partisans sur un champ mitoyen à une ou plusieurs villes terminées.

3.5.1 Algorithme

Dans un premier temps intéressons nous au cas des constructions ville et route. Pour attribuer des points à un joueur, il est nécessaire d'effectuer un parcourt du graphe et de valider plusieurs prérequis. Nous utilisons ainsi l'algorithme de parcourt 1 et en exécutant sur chaque sommet plusieurs fonctions. Le principe de cette fonction est donné par l'algorithme 2.

Cet algorithme est exécuté lorsque qu'une tuile est posée sur le plateau, seulement si celle-ci est validée, sur chacun des sommets ville et route de la tuile. L'algorithme 2 effectue deux parcours exécutant pour chaque sommet des fonctions en temps constant. Sa complexité est donc encore une fois dans le pire des cas $\Theta(m)$ où $m = |E_{graphe}|$, mais est inférieur en moyenne.

```
   Entrées : tuile :graphe, G :graphe, joueurs :structure
1  pour chaque sommets (ville ou route) som de tuile faire
2      si a_successeur(som) alors
3          si parcourt_graphe(G, som, est_complet_route_ville(), Vrai) alors
4              si parcourt_graphe(G, som, {est_pion(), marquer_complet(),
5              supprimer_pion()}, {Vrai, Vrai, Vrai}) alors
6                  calcul_score(G, som, joueurs);
```

Algorithme 2: Algorithme détectant une structure achevée et calculant le score associé

Ensuite, à la fin de la partie il est possible d'évaluer les constructions champ suivant l'algorithme 3. Il s'agit d'un parcourt sur les sommets champ. Le principe repose sur l'exécution de l'algorithme 2, en effet ce dernier marque les constructions terminées ainsi il est ici aisé de repérer un ville achevée. De plus afin de ne pas considérer deux fois une même ville lors du parcourt d'un même champ, les sommets villes achevées sont marqués comme parcourus par ce champ.

Entrées : G :**graphe**, joueurs :**structure**

```
1 pour chaque sommets champs som de G faire
2     si est_blanc(som) alors
3         si parcourt_graphe(G, som, {est_pion(), est_ville_complete_adjacente()},
4         {Vrai, Vrai}) alors
5             nb_ville ← nb_ville + 1;
6 calcul_score_champ(G, nb_ville, joueurs);
```

Algorithme 3: Algorithme calculant le score correspondant aux champs

3.5.2 Implémentation *C*

L'implémentation de l'algorithme 2 se fait en réutilisant les idées données dans la section 3.4.2. Le calcul des points se fait par appel de la fonction `calcul_score_pions` cette fonction détermine le nombre de partisan de chaque joueur à partir du nombre de partisans total. Cependant il faut noter que la solution choisit pourrait poser problème dans le cas ou le nombre de partisan détectés lors d'un même parcourt est supérieur à 9. Toutefois dans la mesure ou le nombre de partisans est limité à 10, il semble peu probable, mais pas improbable, que ce cas se produise. Ensuite connaissant le nombre de partisan de chaque joueurs, le nombre de tuiles et le type de structure achevée, il est possible de calculer le nombre de points et de choisir à qui les attribuer.

L'étape suivante de ce projet aurait été l'implémentation de la fonction d'évaluation des champ mais faute de temps cela n'a pu être réalisé. Toutefois elle aurait été implémentée en suivant l'algorithme 3. De plus la recherche d'une ville adjacente à un sommet champ aurait été réalisé en utilisant le fait suivant :

Fait : dans notre graphe global, si un champ est adjacent à une ville alors le sommet de cette ville a pour indice dans notre graphe `sommet_indice_champ + 1`. Ainsi tous les éléments nécessaire à la réalisation de cet algorithme sont réunis.

3.6 Autres fonctions et outils

3.6.1 Parsing des tuiles

Afin de charger dynamiquement et en dehors du code toutes les définitions des tuiles, un parseur à été implémenté. Les tableaux de sommets de chaque tuile sont ici définis par des séquences commençant et finissant par * sur une seule et même ligne. Ces séquences commencent par le numéro de la tuile puis sont composées de sous séquences délimitées par une lettre (c=champ, r=route, v=ville, m=monastère, e=emblème) et composées d'une suite de quatre chiffres binaires où 1 représente un sommet activé et 0 l'inverse.

Les matrices de booléens sont ici définies par des séquences commençant et finissant par % sur une seule et même ligne. Ces dernières commencent par le numéro de la tuile puis sont composées de sous-séquences délimitées par une lettre (c=champ, r=route, v=ville, m=monastère) et composées d'une suite de 3 entiers le premier étant le numéro de ligne, compris entre 0 et 2, le second le numéro de colonne, compris entre 1 et 3, et le dernier la

valeur du booléen (0 ou 1).

Les espaces ne sont pas pris en compte. Par défaut tout les sommets sont désactivés si une séquence n'est pas renseignée, elle sera initialisée à 0000. Chaque ligne de commentaire doit commencer par # et se terminer par un saut de ligne. Enfin tout autre caractère que ceux mentionnées précédemment conduiront à l'échec du parsing.

Durant le parsing un exemplaire de chaque tuiles est alloué, et rangé dans le tableau `tuiles`.

3.6.2 Gestion de la pioche

La gestion de la pioche est réalisée à l'aide de la structure `pioche`, allouée et libérée dynamiquement, qui contient les champs suivant :

- `tuile_pioche` qui correspond au pointeur vers la tuile piochée.
- `nb_carte` qui est un tableau contenant le nombre de fois qu'une tuile a été piochée.
- `nb_carte_max` qui est un tableau contenant le nombre maximal de fois qu'une tuile peut être piochée.

Une fonction se charge de mettre à jour la structure à chaque tours. Celle-ci, tire de manière aléatoire une tuile dans le tableau des tuiles parsées. Elle vérifie que cette tuile peut encore être tirée grâce au tableau *nb_carte_max* et auquel cas met a jour le pointeur de *tuile_pioche* et incrémente son compteur. Sinon une autre tuile est tirée et ainsi de suite. Lors de ce tirage si la carte est disponible avant de mettre à jour le champ *tuile_pioche*, la tuile est dupliquée et c'est sa copie qui se retrouve dans la structure *pioche*.

3.6.3 Rotation des tuiles

Afin de pouvoir faire tourner une tuile de 45 ° vers la gauche ou la droite, on s'est proposé d'implémenter une fonction qui applique une rotation circulaire aux sommet du graphe de la tuiles. En pratique il faut modifier les indices des tableaux de sommet et des matrices de booléens. Cette fonction modifie également le numéro de l'image associé à la tuile pour la partie graphique. Enfin précisons que cette transformation se fait par effet de bord et en temps linéaire en le nombre de sommets, on peut donc considérer ici que la rotation se fait en temps constant.

4 Réalisation de l'interface graphique

Nous avons décidé d'utiliser la bibliothèque multimédia *SDL* pour réaliser la partie graphique du jeu. Cette bibliothèque est en effet facile d'utilisation, assez puissante pour les besoins du projet et possède une importante documentation accessible sur le net.

4.1 Principe d'utilisation de *SDL*

La bibliothèque *SDL* fonctionne de la façon suivante : toute image qui doit être affichée est dans un premier temps chargée dans une variable de type SDL_Surface. On définit ensuite des coordonnées auxquelles placer cette image, dans une variable de type SDL_Rect. Puis on appelle la fonction SDL_BlitSurface, qui lie image, coordonnées et surface sur laquelle afficher l'image. Ce processus est répété pour chaque image à afficher, en sachant que l'ordre d'appel de ces fonctions SDL_BlitSurface est pris en compte ; en plaçant deux images aux mêmes coordonnées, par exemple, la seconde image plaçée (dans l'ordre du code) sera superposée à la première. Une fois toutes les images placées, on appelle alors la fonction SDL_Flip, qui affiche chacune de ces images à son emplacement dans la fenêtre *SDL*. Le processus est répété à chaque appel de la fonction affichage.

4.2 Chargement des images

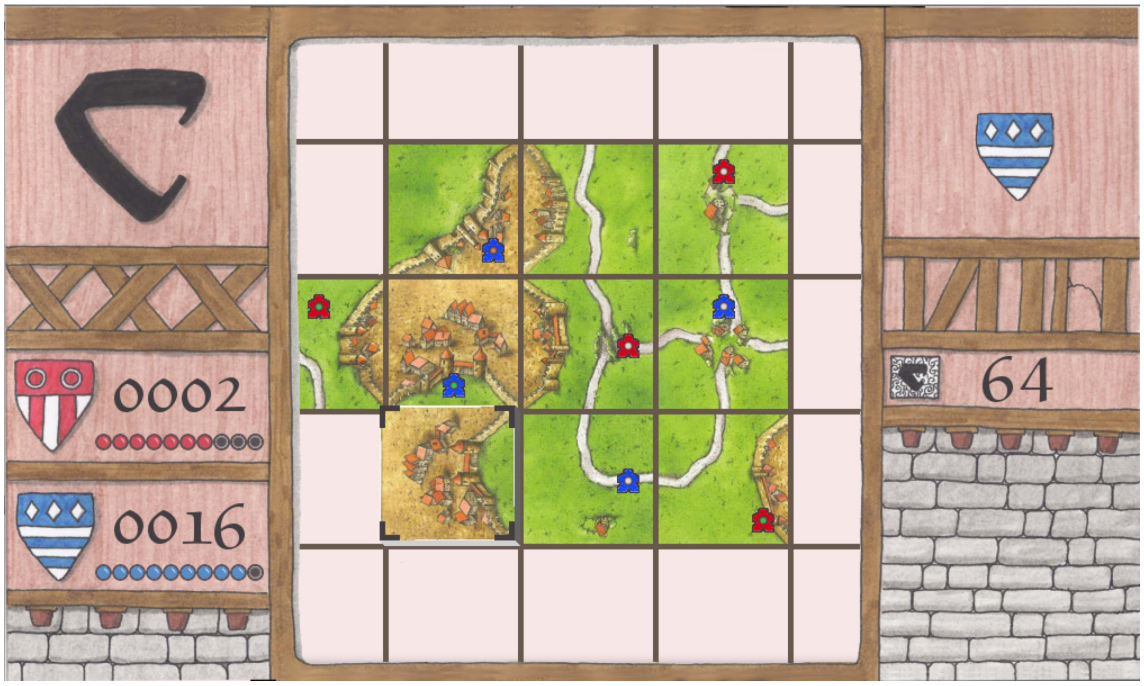
Comme on vient de le voir, il est nécessaire de charger toute image à afficher dans une variable SDL_Surface. Mais ces variables ne sont pas nécessaires au reste du programme ; elles sont de fait locales à la fonction d'affichage. Afin d'éviter de devoir charger toutes les images à chaque appel de la fonction, toutes les variables SDL_Surfaces ont été définies comme étant *statiques*. Ainsi, au premier appel de "affichage", les images sont toutes chargées (à l'aide d'une condition sur un entier statique initialisé à 01 que l'on passe à 10à la fin du chargement) une seule est unique fois au cours de l'exécution du programme.

Par souci de clarté, les images des tuiles sont chargées dans un tableau de SDL_Surface, ce qui permettra de les appeler facilement par le numéro de leur case.

Enfin, par défaut, *SDL* ne peut charger que des images au format bitmap (.bmp) ; cependant, ce format ne prend notamment pas en compte la transparence, nécessaire entre autres pour superposer des images non carrées, tels des partisans par exemple. Nous avons donc utilisé la fonction *IMG_Load* de la librairie SDL_image.h, qui permet le chargement de fichiers de types plus divers, notamment le type Portable Network Graphics (ou PNG), qui prend en compte la transparence ; c'est sous cette extension que nous avons donc enregistré toutes les images du jeu.

4.3 Affichage du plateau de jeu

A chaque appel de la fonction d'affichage, le programme parcourt un tableau de graphes similaire à celui décrivant le plateau de jeu entier (*plateau_graphe*), nommé *plateau_visible*. Ce tableau de 4x4 cases contient les 12 tuiles que doivent être affichées à l'écran, d'après la position du curseur. La fonction lit le champ "image" de chaque tuile, qui correspond au numéro de la case du tableau de SDL_Surface décrit précédemment ; et appelle SDL_BlitSurface pour chacune de ces images aux coordonnées voulues.



4.4 Rotations de cartes

Trois solutions existaient pour traduire graphiquement la rotation de tuiles à l'écran. La première était d'utiliser la fonction *rotozoom* de la bibliothèque SDL_sfx.h, qui permet d'effectuer des rotations sur des SDL_Surfaces définies au préalable. Cependant, afin de faciliter le travail des autres membres de l'équipe, seules des librairie *SDL* installées par défaut sur les ordinateurs de l'école ont été utilisé ; et SDL_sfx.h n'en fait pas partie. Une seconde solution était d'utiliser la bibliothèque SDL_OpenGL.h, qui est disponible à l'*ENSEIRB* et qui permet d'effectuer des rotations. Cependant, *OpenGL* se manipule de façon différente de *SDL*, et impliquait de réécrire entièrement le code des fonctions d'affichage, ce qui nous était impossible au moment où le problème des rotations s'est posé. En outre, SDL_OpenGL.h est beaucoup moins documentée et est extrêmement puissante comparée aux besoins de notre programme ; il nous a donc semblé préférable d'abandonner cette option.

Nous avons donc choisi la troisième option, la moins élégante mais la plus simple à implémenter dans les délais du projet. Pour chaque tuile, nous chargeons 4 images, une pour chaque orientation de la tuile. L'ordre dans lequel cette opération est effectuée est le suivant : dans les 24 premières cases (numérotées à partir de 1) du tableau de SDL_Surfaces sont chargées les images des 24 tuiles. Dans les 24 secondes, sont chargées, dans le même ordre, 24 images de ces tuiles après une rotation de 90 $^\circ$; et ainsi de suite pour les 48 autres. Ainsi, connaissant le numéro de l'image d'une tuile que l'on veut tourner, il suffit d'ajouter ou d'enlever (selon le sens de la rotation) 24 modulo 96 à ce numéro pour obtenir celui de l'image de la tuile tournée.

4.5 Compteurs de scores

Plusieurs compteurs devaient être affichés à l'écran : deux compteurs de score, ainsi qu'un compteur du nombre de cartes restantes dans la pioche et un compteur du nombre de partisans restant à chaque joueur. Si l'on a choisi d'afficher ce dernier d'une façon différente, les trois autres fonctionnent sur le même principe.

Dans un premier temps, lors de la phase de charge des images, on charge 10 images dans un tableau de SDL_Surface. Dans chaque case est stockée une image du chiffre correspondant au numéro de la case (la case 1 contient ainsi un chiffre "1" dans la police voulue). Dans un second temps, une fonction remplit par effet de bord un tableau d'entiers avec 4 chiffres, un par case, chaque chiffre correspondant à l'un des chiffres qui compose le score. Dans un dernier temps, il suffit alors d'appeller une boucle *for* qui placera chaque image de chiffre voulue à l'endroit demandé. Le score sera ainsi mis à jour à chaque appel de la fonction affichage.c, c'est-à-dire en permanence.

Le compteur de partisans fonctionne sur un principe plus simple ; puisqu'un joueur ne peut posséder plus de 10 partisans, nous avons décidé de représenter ce compteur par 10 images, une pour chaque partisan ; une boucle sur le nombre de partisans restants permet donc d'afficher à l'écran le nombre de partisans de chaque joueur.

Conclusion

Nous avons donc réalisé, à l'issue de ce projet, un programme permettant de jouer au jeu *Carcassonne*. Ce fut l'occasion de réutiliser des outils tels que *Cmake* ou *Doxygen* qui permettent d'améliorer l'environnement de travail, la lecture et la compréhension du code, ainsi que *gdb* et *Valgrind*, qui nous ont permis de corriger facilement de nombreux bugs. Ce fut de plus l'occasion de mettre en application le concept des graphes, pour un cas concret, d'un point de vu algorithmique mais également par leur implémentation. Enfin, après l'utilisation de *ncurses* au premier semestre, nous avons pu découvrir une nouvelle bibliothèque graphique plus complexe.

Nous n'avons pas pu implémenter l'ensemble des tâches requises pour ce projet : l'implémentation des abbayes n'a pas pu être réalisée, et le comptage des points obtenus par les champs n'est pas complet. En outre, nous n'avons pas eu le temps d'implémenter une intelligence artificielle. Néanmoins, notre programme final permet à deux joueurs de jouer à une version de Carcassonne relativement fidèle au jeu original. Notre modélisation des règles du jeu est fonctionnelle, et nous espérons proposer une interface graphique agréable pour l'utilisateur.