1ère année informatique

Projet de programmation C : Carcassonne

Rapport final du projet

Team :
Bahrami Tara
Maghraoui Sahar
Nis Cedric
Dermigny Basile

Encadrant :
M. CASSAGNE Adrien

18 mai 2018

Table des matières

1 Introduction

Le but de ce projet est la modélisation du jeu de société "Caracassonne" en le codant en langage C .

1.1 Carcassone

Il s'agit d'un jeu de construction de plateau tour par tour en se basant principalement sur le placement des tuiles. Une tuile peut être composée de villes, champs, abbayes ou routes. Ainsi, le plateau de jeu est initialement composé d'une seule tuile puis se construit au fur et à mesure de la partie. En effet, chaque joueur pioche une tuile et doit la placer dans le plateau du jeu en respectant les tuiles déjà placées.

Lorsqu'un joueur place une tuile, il peut également choisir de placer un pion sur la partie ville champs ou route. Lorsqu'une route ou ville est complétée, les propriétaires de cette dernière peuvent compter leurs points et récupérer leurs pions.

Le jeu se termine lorsque toutes les tuiles sont posées et le joueur avec le plus de points est le gagnant.

1.2 Organisation

L'écriture du code a été faite via l'éditeur de texte Emacs et la compilation grâce à gcc. Le partage du code a été rendu possible par l'outil subversion : svn. Le rapport à été réalisé en Latex grâce à l'outil en ligne sharelatex afin de permettre son écriture collaborative.

1.3 Les étapes

Ce jeu est modélisé grâce à plein de structures qu'on a définit. On commencera par aborder nos choix d'implémentation, en précisant la complexité et la corrections des fonctions qui interviennent. Enfin, nous discuterons les fonctions 'tests'.

2 Les Bases

2.1 Le jeu

Afin de pouvoir modéliser correctement le jeu, il faut au préalable fixer les règles du jeu :

- Il y a 24 tuiles différentes, pour un total de 72.
- Il y a 4 structures différentes (champ, ville, route, abbaye).
- Il y a 7 pions par joueur.
- Chaque joueur pioche et joue une tuile.
- Si un joueur ne peut pas poser la tuile, il passe son tour et la tuile est défaussée.
- Chaque joueur peut choisir de placer un de ses pions sur la tuile.
- Les pions ne sont récupérés que quand la structure (route, ville, champ et abbaye) est finie.

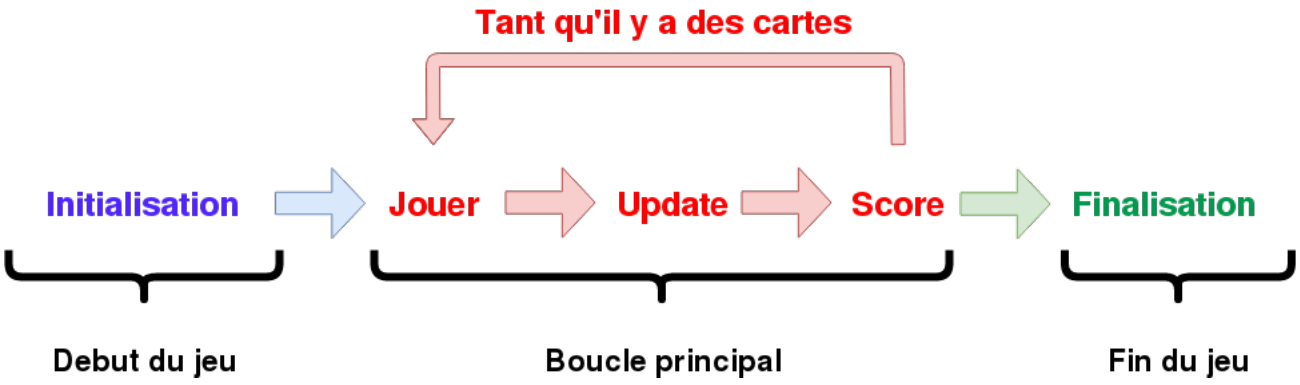
Début de la partie

Au début, on dispose de la carte de départ dans la table. Ensuite, il faut mélanger les tuiles. Chaque joueur reçoit 7 partisans (pions) correspondant à la couleur de son choix.
La phase d'initialisation prépare le plateau afin qu'il soit fonctionnel :

- Création du plateau
- Configuration des joueurs
- Création et mélange du deck

La boucle de jeu

On peut schématiser un tour de jeu comme suit :



Les joueurs jouent à tour de rôle tant qu'il y a des tuiles à piocher. A chaque tour, ils doivent réaliser les actions suivantes en respecter l'ordre indiqué également :

— Un joueur pioche une tuile
— Il joue cette tuile
— Il place son pion
— Il gagne des points

Si grâce à cette tuile, il réussit à achever des chemins, des villes ou des abbayes, tous les joueurs doivent les évaluer et compter les points correspondants. Et enfin, lorsqu'il n'y a plus de tuiles à piocher, on compte les points restant pour départager les joueurs.

2.2 Les impératifs

La modélisation de ce jeu requiert différents points. Tout d'abord, il faut trouver une représentation propice des différents composantes : comment choisir les structures des tuiles et du plateau de jeu ? Comment modéliser les villes, champs, routes et abbayes ? Ensuite, il faut conceptualiser les algorithmes gérant la dynamique du jeu pour qu'ils collent au fonctionnement attendu. De plus, il faut répartir ces fonctions dans plusieurs fichiers sources, et donc trouver un découpage adéquat. Enfin, pour gérer la compilation séparée de ces fichiers et ne pas recompiler inutilement des fichiers non modifiés, il faut établir un Makefile gérant les compilations.

2.3 Choix d'implémentation

La particularité de "Carcassonne" est que le plateau de jeu n'est pas fixe. On part d'un plateau vide qui devient plus grand à chaque tour de jeu . De plus, les tuiles du jeu contiennent pleins d'informations, d'où les différentes structures.

Une tuile

Une tuile est modélisée par une structure *tuille* qui contient les champs suivant :

— **type** : Un tableau d'entiers (4×3) qui contient le type du milieu en fonction de sa position $NORTH$, $WEST$, $SOUTH$ et $EAST$.
— **dir** : Une énumération de direction décrivant l'orientation de la tuile.
— **milieu** : Un entier représentant le type du milieu au centre de la tuile (important pour les carrefours et les villes).
— **shield** : Un booléen mentionnant si la ville contient un bouclier.
— **position** : Un tableau contenant deux champs (x, y) représentant position de la tuile sur le plateau.

— **pion** : Un tableau (4×3) qui indique s'il y a un pion sur l'une des cases de la tuile et qui en est le propriétaire.
— **id** : Un *id*, propre au type de la tuile.

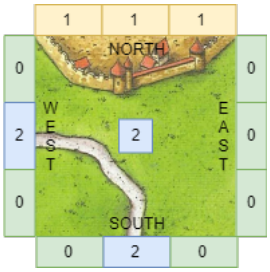
Les tuiles sont initialisées comme ceci : une tuile fantôme "NULL" .

```
struct tuille {
  int type[4][3] = {{ NONE, NONE, NONE}, {NONE, NONE, NONE},
  {NONE, NONE, NONE}, {NONE, NONE, NONE}};
  enum direction dir = NORTH;
  int milieu = NONE ;
  int shield = NONE;
  int position[2] = {NONE,NONE} ;
  int pion[4][3] = {{ NONE, NONE, NONE}, {NONE, NONE, NONE},
  {NONE, NONE, NONE}, {NONE, NONE, NONE}};
  int id = NONE;
};
```

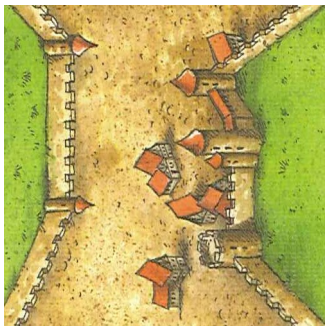
Voilà un exemple de tuile :

```
struct tuille {
  int type[4][3] = {{ 1, 1, 1}, {0, 2, 0}, {0, 0, 0}, {0, 2, 0}};
  enum direction dir = NORTH;
  int milieu = 2 ;
  int shield = FALSE;
  int position[2] = {NONE,NONE}; #initialisation
  int pion[4][3] = {{ 0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
  int id_img = 7;
};
```

Elle correspond à la tuile :



La tuile suivante montre l'importance du choix du champ **milieu** dans la structure **tuille** : savoir si les bords de la tuile passent par le milieu ou s'il s'agit de deux zones distinctes. Par exemple, sans ce champ, on ne peut pas faire la différence entre ces deux tuiles :



Afin de faciliter le comptage des points et le positionnement des pions, un chemin, un champ, une abbaye et une ville sont chacune modélises par une structure. En effet, si on stocke dans des structures l'avancement de ces différents édifices, il est possible de connaître leur statut (fini ou non), et ce, sans faire des parcours. A condition qu'elles soient mises à jour à chaque tour de jeu.

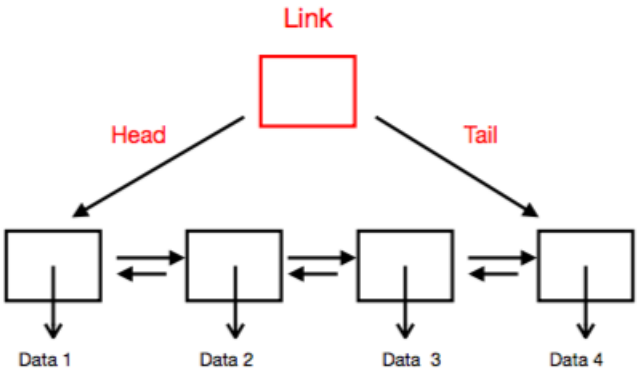
Pour faciliter leur utilisation, des structures supplémentaires ont été définies :

link : une structure de liste doublement chaînée (avant et arrière) avec un pointeur en tête et en queue, avec une donnée quelconque.

```
struct link {
  struct lelement * head;
  struct lelement * tail;
  int taille;
};
```

lelement : une structure d'élément contenant une donnée quelconque et deux pointeurs vers l'élément suivant et précèdent .

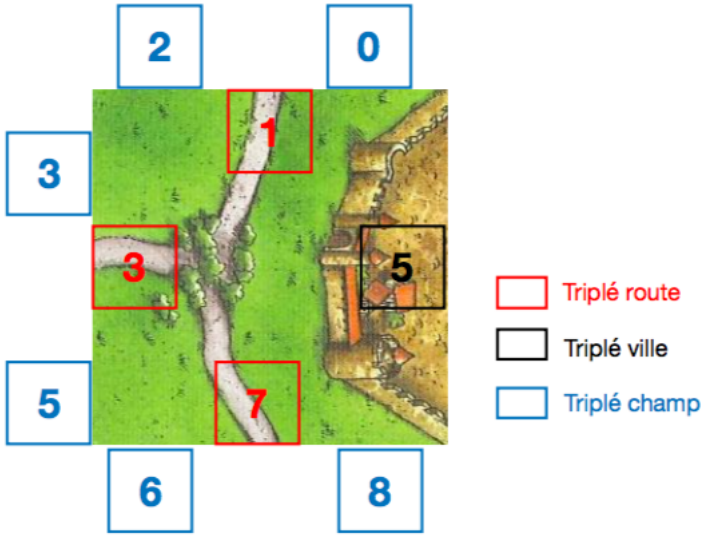
```
struct lelement {
  void * data;
  struct lelement * next;
  struct lelement * previous;
};
```



triplet : afin d'éviter les conflits entre les édifices lors de la recherche sur le plateau de jeu.

```
struct triplet{
  int x;
  int y;
  int z;
};
```

L'exemple de tuile suivante montre bien que sans cette information supplémentaire, on ne peut pas faire de distinction entre les différentes routes :



Une route

Une route est modélisée par une structure **route** mise à jour à chaque fois qu'une tuile est ajoutée sur le plateau de jeu. Elle contient les champs suivants :

- **lst_tutuile** : la liste des tuiles qui la composent.
- **lst_triplet** : la liste des triplets qui la composent.
- **voleurs** : les pions que chaque joueurs possède sur la route.
- **avancement** : l'état de la route (0 pas d'extrémité, 1 une extrémité, 2 la route est finie).
- **len** : taille de la liste des tuiles.

```
struct road {
  struct link * lst_tutuille;
  struct link * lst_triplet;
  struct link * lst_champ;
  int voleurs[PLAYER_MAX];
  int avencement;
  int len;
};
```

Exemple d'une *struct road* :

```
struct road {
  struct link * lst_tuile = lnk -> T1 <-> T2 <-> T3 <-> T4
  # la liste des tuiles
  struct link * lst_triplet = lnk -> {0,0,7} <-> {1,0,5} <-> {1,0,7} etc.
  #la liste des triplets
  int voleurs[PLAYER_MAX] = {1, 1, 0}
  # seul le joueur 3 n'a pas de voleur sur la route
  int avencement = 1
  # la route a déjà une extrémité : la tuile1
  int len = 4
  # ce qui fait 4 points pour les joueurs 1 et 2
};
```

Elle correspond à la route :

Un champ

Un champ est représenté par une structure **field** qui est mise à jour après chaque ajout de tuile sur le plateau de jeu. Cette structure n'est pas utilisée en elle-même. Elle est présente pour compléter la structure ville et simplifier le comptage des points. Elle contient les champs :

- **lst_tutuille** : la liste des tuiles qui la composent.
- **lst_triplet** : la liste des triplés qui la composent.
- **paysans** : un tableau qui indique le nombre de paysans que chaque joueur possède sur le champ.
- **len** : la taille de la liste.
- **intown** : un booléen qui indique si le champ appartient à une ville.

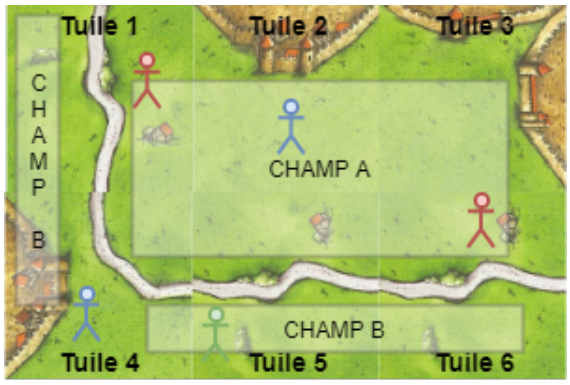
```
struct field {
  struct link * lst_tutuille;
  struct link * lst_triplet;
  int paysans[PLAYER_MAX];
  int len;
  int inTown;
};
```

Exemples de *struct field* :

```
struct field A {
  struct link * lst_tutuille -> T1 <-> T2 <-> T3 <-> T4 <-> T5 <-> T6
  struct link * lst_triplet -> {0,0,8} <-> {0,0,9} <-> {0,0,10} etc
  int paysans[PLAYER_MAX] = {2, 1, 0}
  int len = 6
  int inTown = 1
};

struct field B{
  struct link * lst_tutuille -> T1 <-> T4 <-> T5 <-> T6
  struct link * lst_triplet -> {0,0,2} <-> {0,0,3} <-> {0,0,4} etc
  int paysans[PLAYER_MAX] ={0, 1, 1}
  int len = 4
  int inTown = 1
};
```

Ce qui correspond à :



Une ville

Une ville est représentée par une *structure city* qui est mise à jour après chaque ajout de tuile sur le plateau. Cette structure contient aussi la liste des champs qui l'entourent afin de simplifier le calcul des points liés aux champs en fin de partie. Elle contient les champs :

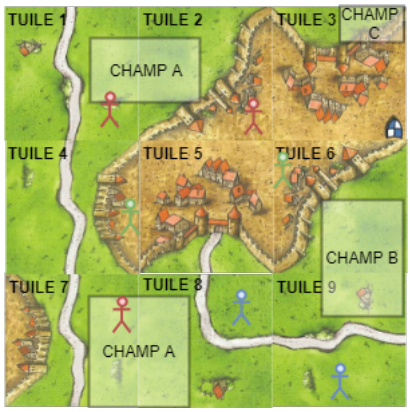
— **lst_tutuille** : la liste des tuiles qui composent la ville
— **lst_champ** : la liste des champs adjacents à la ville
— **lst_triplet** : la liste des triplés qui composent la ville
— **chevaliers** : un tableau qui indique le nombre de chevaliers que chaque joueur possède dans la ville
— **border** : le nombre de bord non fini de la ville (0 = ville finie)
— **len** : la taille de la ville
— **shield** : le nombre de boucliers

```
struct city {
  struct link * lst_tutuille;
  struct link * lst_champ;
  struct link * lst_triplet;
  int chevaliers[PLAYER_MAX];
  int border;
  int len;
  int shield;
};
```

Exemple d'une *struct city* :

```
struct city {
  struct link * lst_tutuille-> T1 <-> T2 <-> T3 <-> T4 <-> T5 <-> T6
  <-> T7 <-> T8 <-> T9
  struct link * lst_champ -> CA <-> CB <-> CC
  int paysans[PLAYER_MAX] = {1, 0, 2}
  int fini = FALSE
  int shields = 1
  int len = 5
};
```

Ce qui correspond à :



Une abbaye

Une abbaye est représentée par une structure *abbaye*. Elle contient les champs :
— **cloister** :la tuile qui représente l'abbaye
— **voisin** :le nombre de tuiles voisines
— **possesseur** :le joueur qui possède l'abbaye

```
struct abbaye {
  struct tuille cloister;
  int voisin;
  int possesseur;
};
```

Le plateau de jeu

Enfin le plateau de jeu est modélisé par une structure *structure boardgame* qui contient les champs :
— **board** :le tableau de jeu
— **fields** :la liste des champs
— **roads** :la liste des routes
— **cities** : la liste des villes
— **deck** :le deck des cartes (tuiles)
— **abbayes** :la liste des abbayes
— **points** : les points de chaque joueur

```
struct boardgame {
  struct tuille board[2*CARD_MAX+1][2*CARD_MAX+1];
  struct link * fields;
  struct link * roads;
  struct link * cities;
  struct tuille deck[CARD_MAX];
  struct link * abbayes;
  int points[PLAYER_MAX];
};
```

Le plateau de jeu est représenté sous la forme d'un tableau de taille $2CARD\_MAX \times 2CARD\_MAX + 1$ pour pouvoir avoir accès en temps constant aux cases adjacentes au détriment de la complexité mémoire.

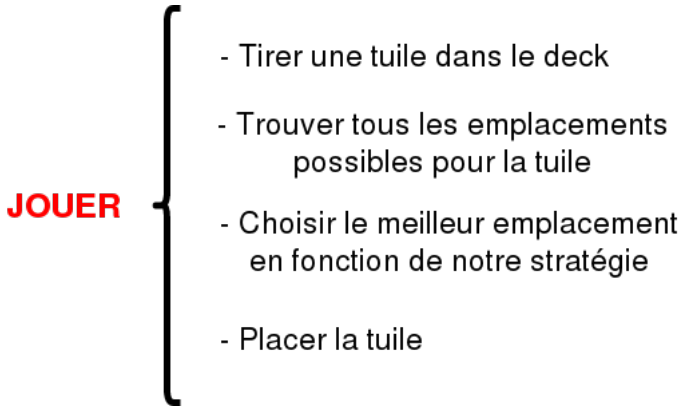
3 Déroulement du Jeu

3.1 Initialisation

Cette phase se déroule au début du jeu avant le premier tour de jeu et après le chargement des joueurs. Elle consiste principalement en l'initialisation de plateau de jeu avec des tuiles "fantômes" :NONE, l'ajout de la tuile de départ et l'initialisation des édifices. Ainsi que la création et le mélange du deck des tuiles qui serviront pour la partie.

3.2 Dynamique du jeu

La phase JOUER correspond à :

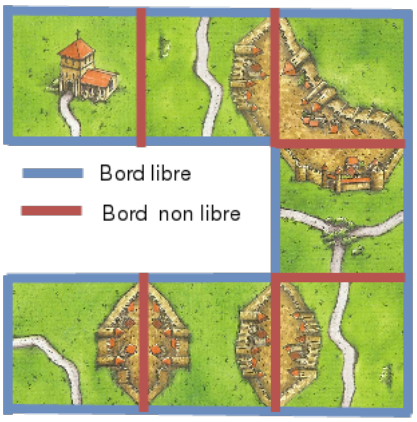


Les fonctions "tirer une carte" et "la placer dans le boardgame" se font directement grâce à la *structure boardgame*.

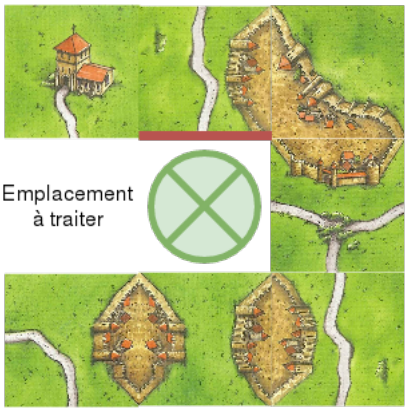
Algorithm 1 Trouver un Emplacement

Require: P : Plateau de jeux
Require: T_p : tuile pioché
  $Lst_{possible} \leftarrow Lst_{vide}()$
  for $T_i \in P$ **do**
    if $\exists Emplacement \in Voisinage(T_i)$ **then**
      $Ajout(Lst_{possible}, Emplacement)$
    end if
  end for

la complexité de cette algorithme est $\mathcal{O}(n^2)$ (n=Nombre_Cartes)
Pour trouver un emplacement possible il faut :
— trouver toutes les tuiles qui ont des emplacements libres
— vérifier que le coté coïncide avec notre tuile piochée
— vérifier que les autres cotés s'insèrent avec leur voisin
— et cela pour les 4 orientations possibles de la tuile piochée.



Dans les cas où les bords sont non libres, l'algorithme va simplement continuer à se propager dans le plateau de jeu. Et ce, jusqu'à trouver un bord libre pour procéder après aux tests pour savoir si la tuile peut être posée ou non.



Dans cette situation, on se demande si l'on peut placer la tuile à partir du rebord en rouge (deux orientations sont donc possibles). En étudiant la topographie de la carte, l'algorithme va remarquer qu'une seule des deux orientations possibles est valide. En effet, l'orientation typée "rouge" est invalide. Elle impose qu'un milieu de ville soit directement en contact avec un milieu de champ. Seule la verte sera donc retenue parmi les choix possibles.

Enfin, la sélection de la position finale de la tuile demande de mettre en place une stratégie. Celle-ci devra prendre en compte les coups joués précédemment, la position et l'orientation des tuiles et surtout le positionnement des pions. Ce qui est simplifié par nos structures de données car elles permettent d' accéder directement aux édifices.On peut différencier 3 types de stratégies :

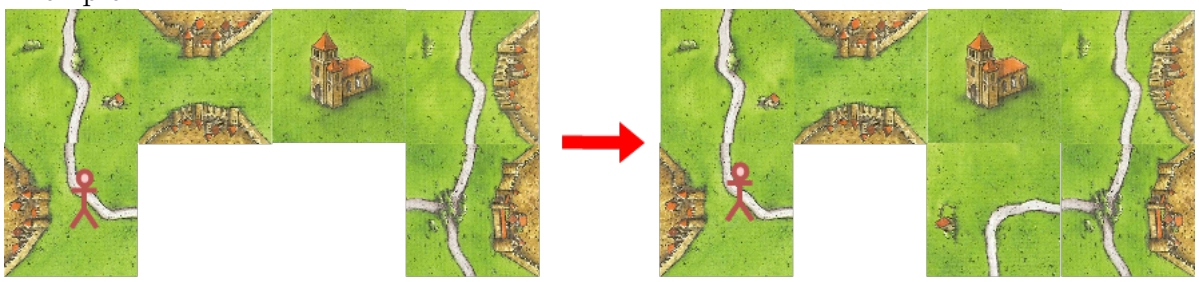
Renforcement

Il s'agit de la stratégie de base qui consiste à continuer à achever les champs, villes, abbayes et routes et d'essayer de maximiser les points qu'elles rapportent (maximiser leur taille). Cette stratégie ignore donc les autres joueurs pour se concentrer sur ses propres édifices.

Bloqueur

C'est la stratégie inverse, elle consiste à se concentrer principalement sur les édifices et à rajouter des tuiles qui bloquent/compliquent leur développement.

Exemple :

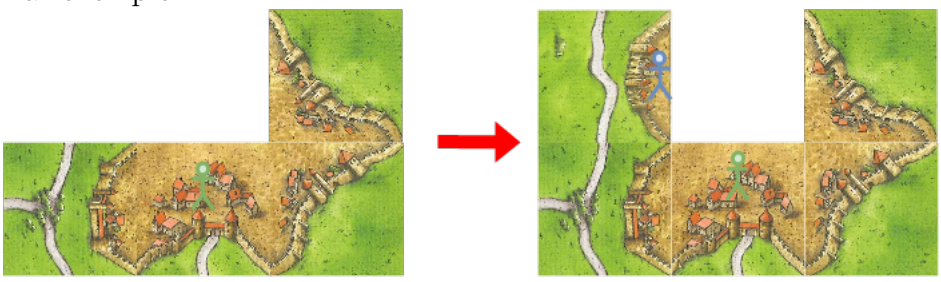


Ici, le joueur aura beaucoup de mal à finir sa route à moins d'obtenir la seule tuile qui va à l'emplacement suivant.

Parasite

Cette stratégie prend le contre pied de la précédente. Elle se concentre aussi sur les autres joueurs mais essaye plutôt de s'incruster dans les édifices des autres joueurs afin de gagner le même nombre de points que eux.

Par exemple :



Le joueur vert est en train de construire une grande ville qui va lui rapporter énormément de points. Cependant, le joueur bleu, en posant sa tuile juste à coté, se donne la possibilité de se raccrocher à la ville de son voisin. Ainsi avec une tuile le joueur bleu a marqué autant de points que le joueur vert avec toutes ses tuiles.

On finit par insérer la tuile sur le plateau, avec le pion placé au milieu choisi, ce qui se fait facilement grâce à nos structures : soit le milieu n'appartient à aucune structure et on peut le mettre, soit il appartient à une structure et on ne peut pas placer de pion.

3.3 La mise à jour

Cette phase est la plus importante, il faut mettre à jour nos structures de données après l'ajout d'une tuile. En effet, nos structures contiennent beaucoup d'informations ce qui nous permet de réduire notre complexité en temps et de simplifier le calcul des points. Elle se fait suivant les étapes :

La mise à jour des routes

Algorithm 2 Update Road

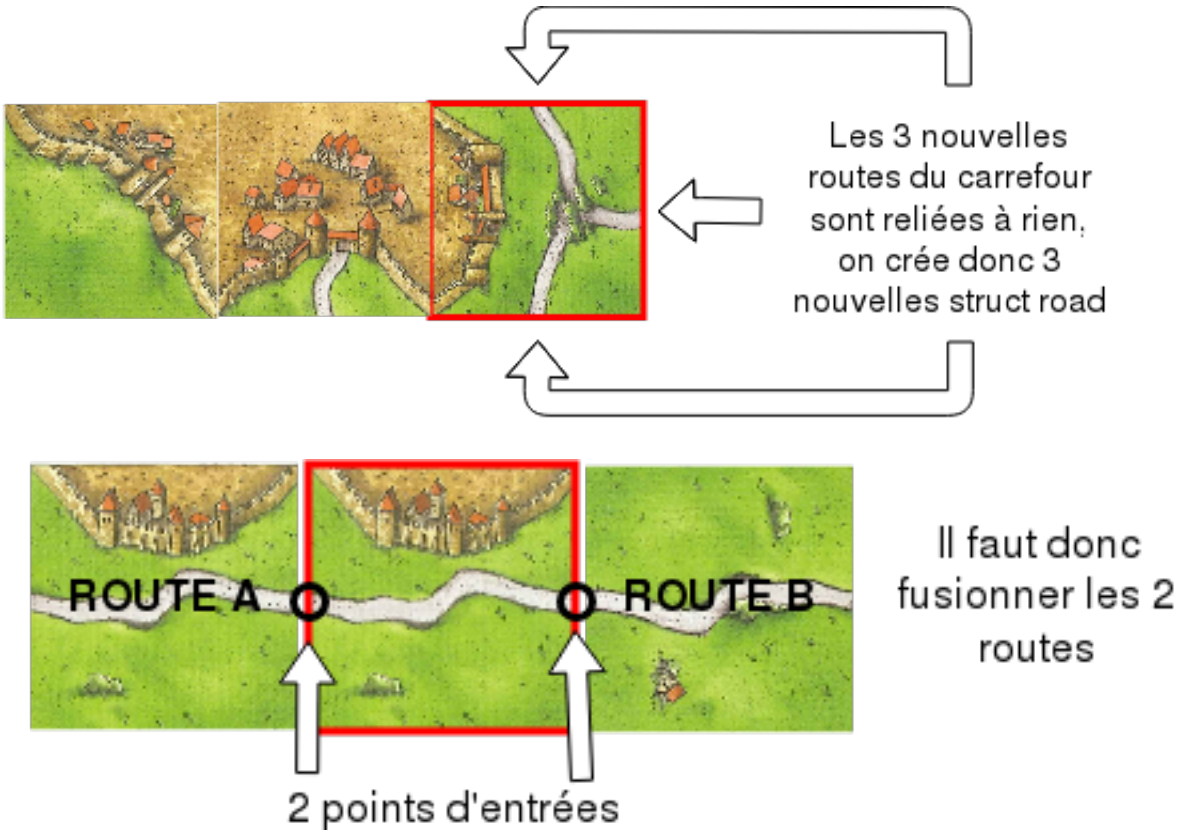
Require: B : boardgame
Require: T_p : tuile pioché
$Lst_{route} \leftarrow Lst_{vide}()$
$Lst_{route} \leftarrow Extraire\_Route\_In\_Tuile(T_p)$
for all $R_i \in Lst_{route}$ **do**
$FusionRoute(B, R_i)$
end for

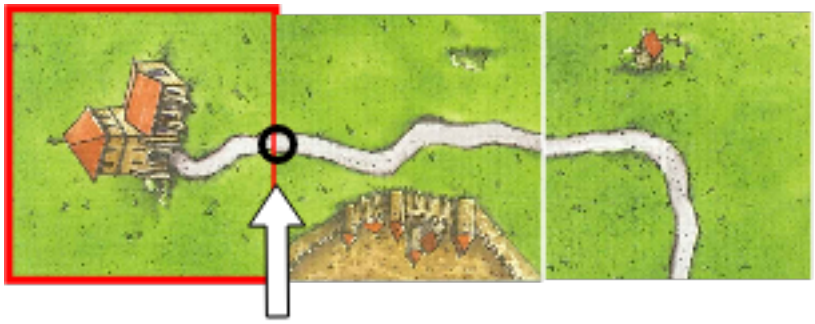
la complexité est $\mathcal{O}(taille_{max}(R_i) * Nombre\_Route\_In(B))$
Après avoir rajouté une tuile, on vérifie chaque route qui la compose :

— Soit elle n'est reliée à rien, et donc on créé une nouvelle *struct road* que l'on rajoute à la liste des routes dans la *struct boardgame*.
— Soit la route continue une route existante ; il faut donc mettre à jour sa *struct road* :
  — Si notre route a deux points d'entrées, c'est un segment, elle ne fait donc que prolonger la route, il suffit donc de la rajouter à la liste des tuiles de la *struct road* et fusionner les deux *struct road* si elle a fait la liaison entre deux routes.

— Sinon, si notre route possède 1, 3 ou 4 entrées, c'est une extrémité de route, il faut donc en plus de la rajouter à la liste des tuiles de la *struct road*, modifier son paramètre d'avancement.
— Il faut aussi traiter le cas rare des routes circulaires.

Exemple de mise en application :





1 points d'entrée

Il faut donc rajouter la nouvelle tuile et marquer le fait que la route à une extrémité maintenant.

La mise à jour du score lié aux routes se fait en même temps que celle des routes, si on vient de finir une route lors de l'update on crédite les joueurs majoritaires et on leurs rend leurs pions qui correspondent à : $points = 2 * len(Route)$

La mise à jour des villes

Après avoir rajouté une tuile, on vérifie chaque ville qui la compose :

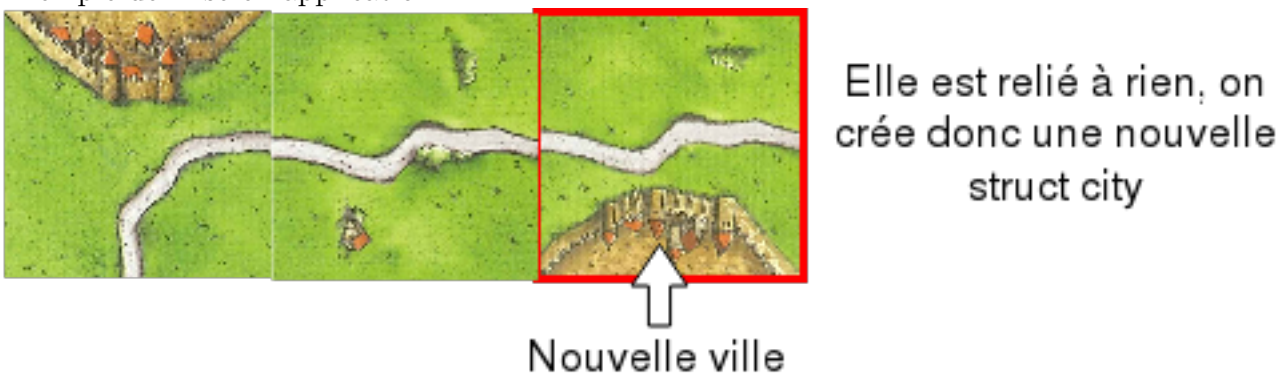
- Soit elle n'est reliée à rien, donc on créé une nouvelle *struct city* que l'on rajoute à la liste des villes dans la *struct boardgame*.
- Soit la ville continue une ou plusieurs villes existantes : il faut donc mettre à jour sa *struct ville* :
  - On regarde donc chaque ville adjacente à notre tuile et on fusionne toutes les *struct city* qui se recoupent en une nouvelle.
  - il faut aussi fusionner les champs associés aux villes.

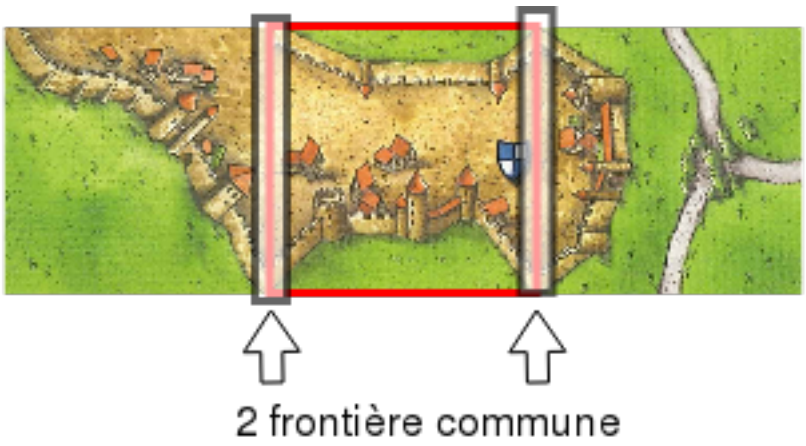
Algorithm 3 Update Town

Require: B : boardgame
Require: T_p : tuile piochée
$Lst_{Ville} \leftarrow Lst_{vide}()$
$Lst_{Champ_{in\_Ville}} \leftarrow Lst_{vide}()$
$Lst_{Ville} \leftarrow Extraire\_Ville\_In\_Tuile(T_p)$
$Lst_{Champ_in\_Ville} \leftarrow Extraire\_Champ\_In\_Tuile(T_p)$
for all $T_i \in Lst_{Ville}$ **do**
$FusionVille(B, T_i)$
end for
for all $F_i \in Lst_{Champ_in\_Ville}$ **do**
$FusionChamp(B, T_i)$
end for

la complexité est $\mathcal{O}(taille_{max}(F_i) * Nombre\_Champ\_In(B))$.

Exemple de mise en application :





Les 2 villes A et B sont maintenant relié par la nouvelle tuile, ils faut donc fusionner city A et city B.

La mise à jour du score liée au villes se fait en même temps, si on vient de finir une ville lors de lla mise à jour, on crédite les joueurs majoritaires et on leurs rend leurs pions qui correspondent à : $points = 2 * (len(Ville) + nombre(bouclier))$

La mise à jour des champs

Après avoir rajouté une tuile, on vérifie chaque champ qui la compose :

- Soit il n'est relié à rien et donc on crée une nouvelle *struct field* que l'on rajoute à la liste des champs dans la *struct boardgame*.
- Soit le champ continue un ou plusieurs champs existants : il faut donc mettre à jour les *struct fields* :
  - On regarde donc chaque champ adjacent à notre tuile et on fusionne toutes les *struct fiels* qui se recoupent en une nouvelle.

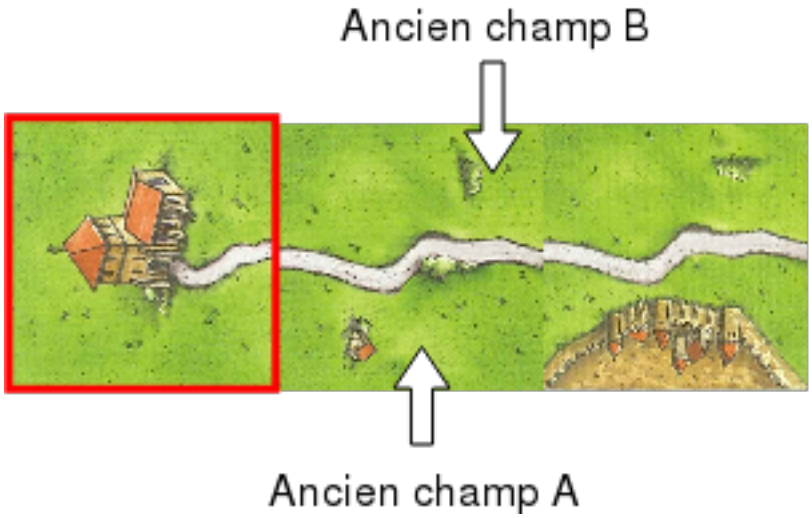
Algorithm 4 Update Champ

Require: B : boardgame
Require: T_p : tuile piochée
  $Lst_{Champ} \leftarrow Lst_{vide}()$
  $Lst_{Champ} \leftarrow Extraire\_Champ\_In\_Tuile(T_p)$
  for all $F_i \in Lst_{Champ}$ **do**
    $FusionChamp(B, T_i)$
  end for

la complexité est $\mathcal{O}(taille_{max}(F_i) * Nombre\_Champ\_In(B))$

Exemple de mise en application :

Le nouveau champ est relié à rien, on crée donc une nouvelle struct field



La nouvelle tuile fusionne les champs A et B

La mise à jour du score lié au champ ne se fait qu'à la fin, ce qui implique que les pions mis dans un champs ne son pas récupérables.

La mise à jour des abbayes

La mise à jour de cette structure est plus simple, il suffit d'incrémenter le nombre de voisins des abbayes autour de la nouvelle tuile et de rajouter les nouvelles abbayes à la liste des abbayes dans le boardgame. la complexité est $\mathcal{O}(Nombre\_Abbaye\_In(B))$

La mise à jour du score lié au abbaye se fait en même temps, si on vient de finir une

abbaye lors de l'update, on crédite le joueur propriétaire et on lui rend son pion. Ce qui ce correspond à : $points = 9$

3.4 Finalisation

Une fois que toutes les tuiles ont été placées, le jeu est terminé. À cet instant, il faut compter les points fournis par les champs ainsi que les points fournis par les structures non terminées.

La fonction $finalisation$ exécute successivement les fonctions suivantes :

— Donner les points aux joueurs possédant une ville non finie.

— Donner les points aux joueurs possédant une route non finie.

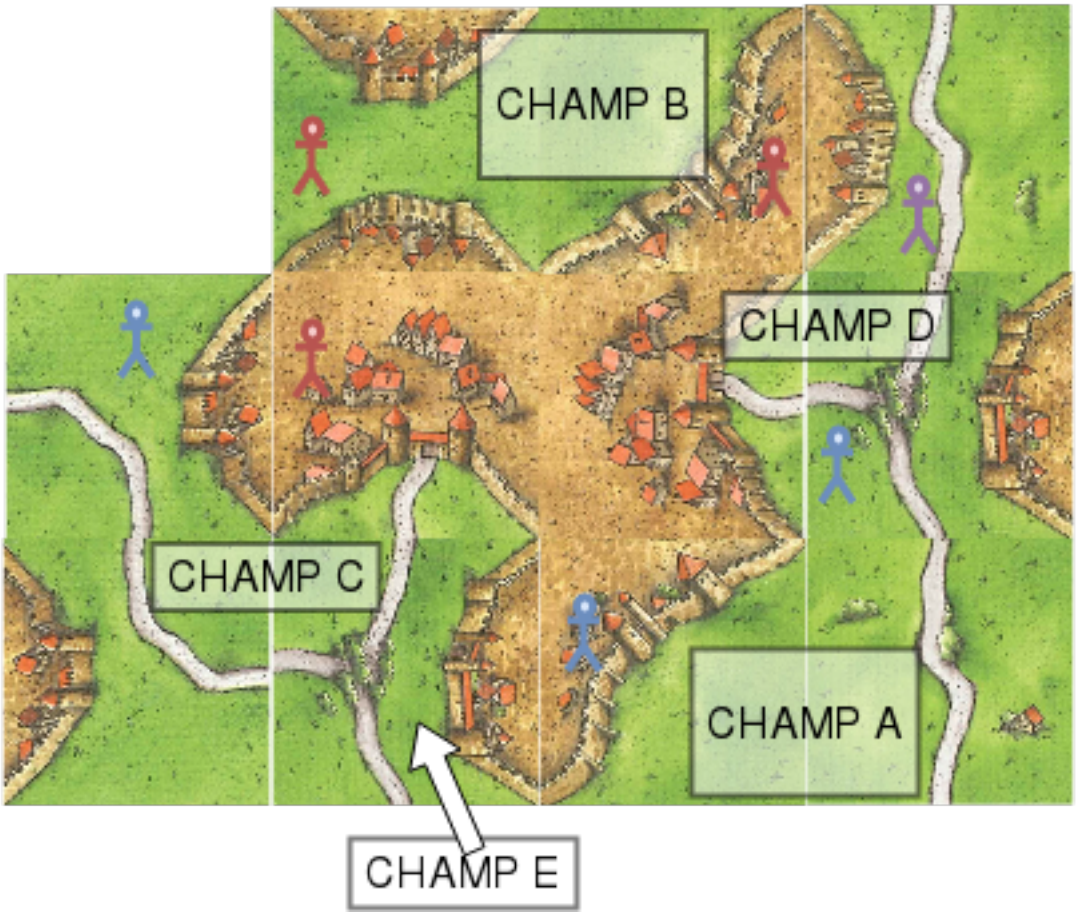
— Accorder les points liés aux champs.

— Déclarer le gagnant.

Les choses se compliquent avec les champs. En effet, il faut réfléchir en terme de ville. Pour chaque ville finie, le joueur possédant le plus de champs en contact direct avec cette ville gagne trois points. En cas d'égalité, les deux joueurs gagnent trois points (les n joueurs possédant le plus de champ pourplus de précision).

Mais grâce notre *structure city*, rien de plus simple. Les champs en contact avec une ville sont stockés dans la structure. Il suffit donc de voir qui sont les propriétaires de ces champs.

Pour cela, il est nécessaire de regarder chaque champ, de voir à qui il appartient, et de continuer en comptant l'importance des propriétaires (leur nombre de champs en contact avec la ville). Les mêmes règles d'égalités sont appliquées.



Dans cette situation, quatre champs sont possédés par des joueurs. Ces champs touchent une ville (relativement grosse). De ce fait, le comptage des points se fait de cette manière :

1. le joueur rouge possède le champ B
2. le joueur violet possède le champ D
3. le joueur bleu possède les champs A et C.

Le joueur bleu possède le plus de champs autour de la ville, il gagne donc trois points, ce qui n'est pas le cas de ses adversaires.

4 Limitations et Contraintes

La réalisation de ce projet a soulevé de nombreux problèmes, aussi bien algorithmiques, que d'implémentation.

Premièrement, la généralisation des listes chaînées nous a permet de rajouter et enlever facilement des éléments. Cependant, afin de pouvoir les utiliser avec des types de donnés quelconques, nous étions obligé de caster tous nos pointeurs, faire attention à ne pas faire de listes hétérogènes et augmenter le nombre de pointeurs (déja aloués) à liberer à la fin du programme.

Suite à des contraintes de temps, il nous a été impossible d'implémenter les stratégies détaillées plus haut. Ainsi les tuiles sont choisies aléatoirement parmi la liste des possibilités. De même les algorithmes qui choisissent les pions ont été implémentés, mais n'ont pas pu être intégré correctement à l'ensemble.

De plus, les règles de Carcassonne comporten de nombreuses exceptions et situations ambiguës, ce qui a énormément complexifié la généralisation du code pour les mide à jour. C'est pourquoi, même si on a réussi à séparer les tuiles en plusieurs grosses catégories, certaines disjonctions de cas doivent être traités de manière unique.

Ainsi bien que nous ayons réalisé l'ensemble des fonctions de mise à jour et qu'elles soient testées indépendamment du serveur (et presque intégralement pour les champs), nous avons du, afin d'assurer que l'exécutable soit stable, ne mettre dans le serveur et les clients que updateCloister, updateroad et updatetown.

Cependant, la plus importante limitation/problème de notre réalisation est la complexité en mémoire. En effet, bien que notre complexité en temps soit plutôt acceptable. Nos structures prennent énormément de place ce qui nous a causé un débordement de la pile sur le tas. Nous avons réglé ce problème en faisant une allocation dynamique de nos structures. Cependant une meilleure solution aurait été de faire un tableau de tuile et de le parcourir à chaque fois : on passerait d'une complexité en espace de $\mathcal{O}(n^2)$ à $\mathcal{O}(n)$.