1ère année informatique

Projet de programmation C

Rapport 2nd jalon du projet

Team :
Bahrami Tara
Maghraoui Sahar
Nis Cedric
Dermigny Basile

Encadrant :
M. CASSAGNE Adrien

13 avril 2018

Table des matières

Introduction

Le but de ce jalon est d'écrire un ensemble de structures de données et d'algorithmes permettant de représenter les cartes en jeu, et de calculer les points lors du comptage final.

1 Les cartes

Dans Carcassonne les cartes représentent les tuiles du jeu. Sur ces dernières, on retrouve toutes les informations nécessaire au jeux :

— les différents types de milieu : les villes, les chemins, les prés et les abbayes
— la position des pions
— la présence d'un bouclier (+1 point sur la valeur de la carte)

1.1 Les structures de données

Pour commencer, nous avons réfléchi à la manière d'implémenter les cartes du jeu. Nous avons donc choisi de créer une structure *tuile* contenant les paramètres suivant :

— Un tableau d'entier (4×3) qui contient le type du milieu en fonction de sa position $NORTH$, $WEST$, $EAST$ et $SOUTH$.
— Une énumération de direction décrivant l'orientation de la tuile.
— Un entier représentant le type de milieu au centre de la tuile, ce qui est important pour les carrefours et les villes.
— Un booléen mentionnant si la ville contient un bouclier.
— Un tableau contenant deux champs (x, y) mentionnant la position de la tuile sur le plateau.
— Un tableau de booléen (4×3) qui dit s'il y a un pion sur l'une des cases de la tuile.
— Un *id*, associé à l'image en question, qui sera utilisé plus tard pour l'affichage de la tuile

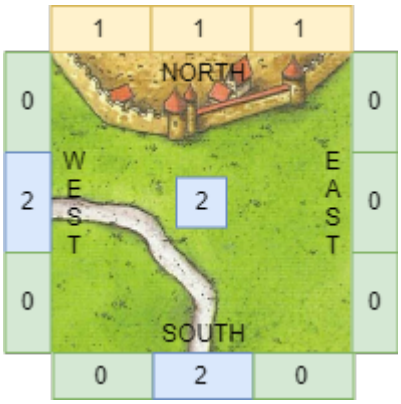
Les tuile sont initialisé comme ceci :

```
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_img = NONE
};
```

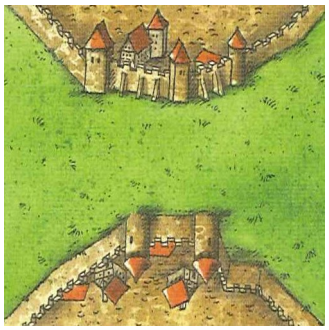
Un Exemple de tuile :

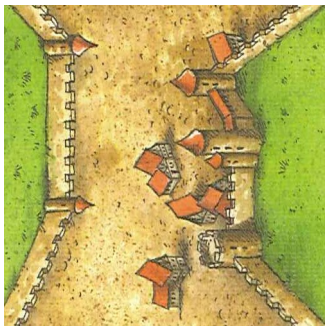
```
struct tuille {
  int type[4][3] = {{ 1, 1, 1}, {0, 2, 0}, {0, 0, 0}, {0, 2, 0}}
  enum direction dir = NORTH
  int milieu = 2
  int shield = FALSE
  int position[2] = {NONE,NONE} #initialisation
  int pion[4][3] = {{ 0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
  int id_img = 7
};
```

Ce qui correspond à :



Il est très important d'avoir une case au milieu afin de savoir si les bords de la tuiles passent par le milieu ou si il s'agit de deux zones distinctes. Par exemple, sans le milieu ont ne peut pas faire la différence entre ces deux tuiles :





1.2 Les structures des différents types de milieux

Dans un deuxième temps, nous avons créé une structure pour les chemins, les champs et les villes afin de faciliter le comptage des points. En effet, si on stocke dans des structures l'avancement des différents paysages, il est possible de connaître leur statu (fini ou non), et ce, sans avoir à faire de parcours. On traite le cas des abbayes en fin de parties.

Pour leur réalisation, nous avons eu besoin de faire des listes de différents type de données. C'est pourquoi nous avons aussi implémenté des listes doublement chaînées qui prennent des void * comme data.

structure road

Nous utilisons notre *structure road* pour stocker l'état d'une route, qui est mis à jour à chaque ajout de tuile sur le plateau.

La structure est composée de :
— la liste des tuiles qui composent la routes
— un tableau qui indique le nombre de voleur qu'a chaque joueur sur la route
— un entier qui indique l'avancement de la route (0 pas d'extrémité, 1 une extrémité, 2 la route est finit)
— la taille de la liste

Nous avons donc plus à parcourir le graphe pour avoir ces information, qui se font maintenant en temps constant :
— les tuiles qui la composent
— l'état de la route
— le nombre de point (= la taille)
— le(s) joueur(s) majoritaire(s)

Nous détaillerons la mis à jours des route dans la partie Fonction (2.4).
Exemple d'une struct road :

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

Ce qui correspond à :

structure field

Nous utilisons notre *structure field* pour stocker l'état d'un champ, qui est mis à jour à chaque ajout de tuile sur le plateau. Cette structure n'est pas utilisée en elle-même. Elle est présente pour compléter la structure ville (section suivante) afin de grandement simplifier le contage des points.
La structure est composée de :

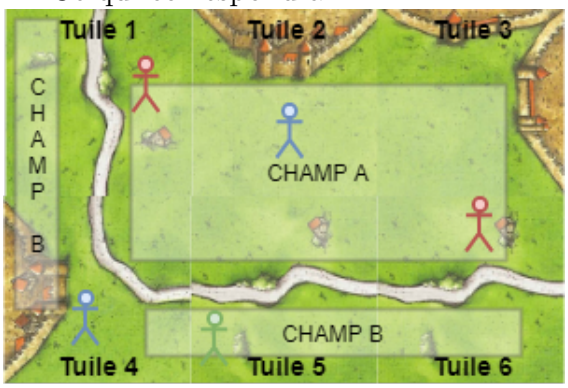
- la liste des tuiles qui composent le champ
- un tableau qui indique le nombre de paysans qu'a chaque joueur sur le champ
- la taille de la liste

Nous détaillerons la mise à jour des champs dans la partie Fonction (2.4).
Exemples d'une *struct field* :

```
struct field A {
  struct link * lst -> T1 <-> T2 <-> T3 <-> T4 <-> T5 <-> T6
  int paysans[PLAYER_MAX] = {2, 1, 0}
  int len = 6
};

struct field B{
  struct link * lst -> T1 <-> T4 <-> T5 <-> T6
  int paysans[PLAYER_MAX] ={0, 1, 1}
  int len =4
};
```

Ce qui correspond à :



structure city

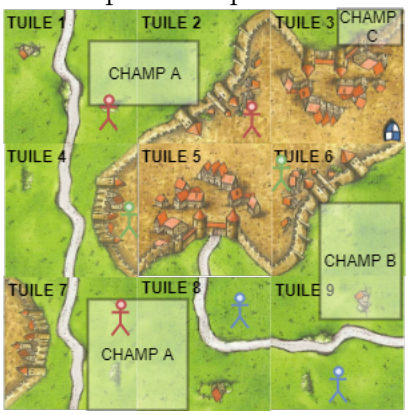
Nous utilisons notre *structure city* pour stocker l'état d'une ville, qui est mis à jour à chaque ajout de tuile sur le plateau. Cette structure contient aussi la liste des champs qui l'entoure afin de simplifier le calcul des points lié aux champs en fin de partie.

La structure est composée de :
— la liste des tuiles qui composent la ville
— la liste des champs qui sont adjacent à la ville
— un tableau qui indique le nombre de chevalier qu'a chaque joueur dans la ville
— un booléen qui indique si la ville est finie
— le nombre de bouclier
— la taille de la ville

Nous détaillerons la mis à jours des villes dans la partie Fonction (2.4).
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 :



structure Boardgame

Pour finir, nous avons décidé de centraliser toutes les informations dans une *structure plateau* :

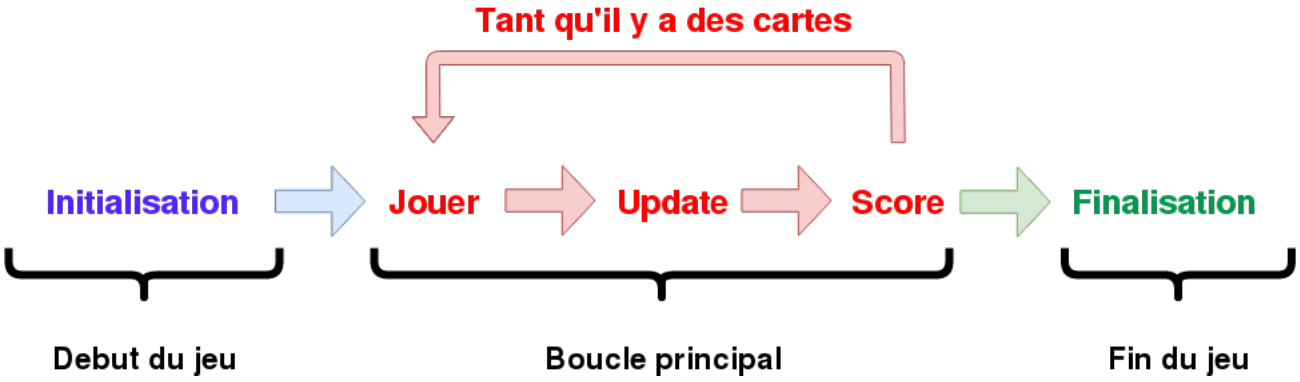
- la liste des joueurs
- le plateau de jeu
- la liste des champs
- la liste des routes
- la liste des villes
- la liste des abbayes
- le deck de cartes
- le curseur du deck

```
struct boardgame {
  struct player joueur[PLAYER_MAX];
  struct tuille board[2*CARD_MAX+1][2*CARD_MAX+1];
  struct link * field;
  struct link * roads;
  struct link * cities;
  struct tuille deck[CARD_MAX];
  struct link * abbayes;
  int len;
};
```

Nous avons décidé de représenter le plateau de jeux sous la forme d'un tableau de taille $CARD\_MAX \times CARD\_MAX + 1$ pour pouvoir avoir accès en temps constant au case adjacente au détriment de la complexité mémoire.

2 Les fonctions

Dans cette section, nous allons voir les fonctions qui utilisent ces structures. En effet, pour garantir l'efficacité de ces fonctions, on doit les mettre à jour à chaque fois que l'on rajoute une tuile, une partie se déroule donc comme si :



2.1 Début de partie

Initialisation

Cette phase se déroule au début du jeux avant le premier tour de jeu et après avoir chargé les joueurs. Elle consiste principalement à vérifier les joueurs et à initialiser le plateau avec des tuiles NONE. Ainsi que la création et le mélange du deck de tuile qui servira pour la partie.

Algorithm 1 Initialisation du deck

```
Require: deck vide
Require: nombre de carte max
  while deck est incomplet do
    rajouter nouvelle carte au deck
  end while
  for i de 0 à taille(deck) do
    r = random(i,taille(deck))
    permuter(deck[i],deck[r])
  end for
```

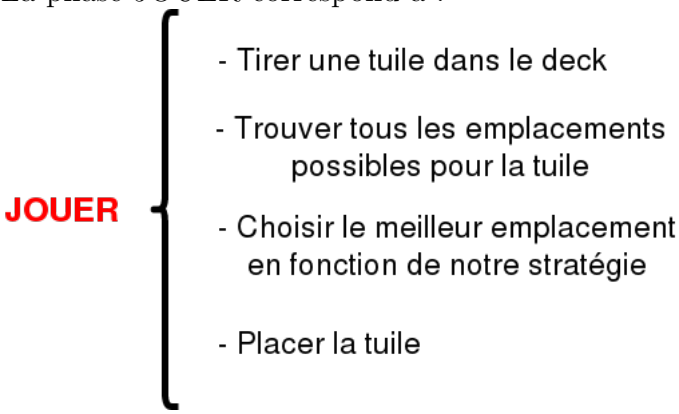
la complexité est $\mathcal{O}(Nombredecarte)$

2.2 Boucle principal

Cette phase constitue le corps du jeu

Jouer

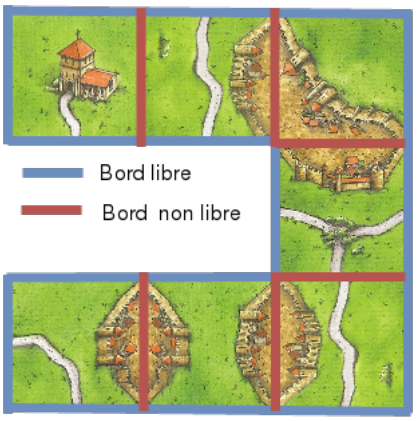
La phase JOUER correspond à :



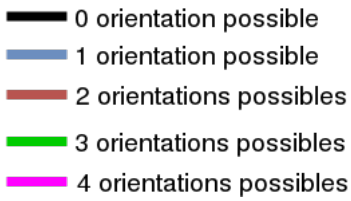
Les fonctions "tirer une carte" et placer se font directement grâce à la *structure boardgame*.

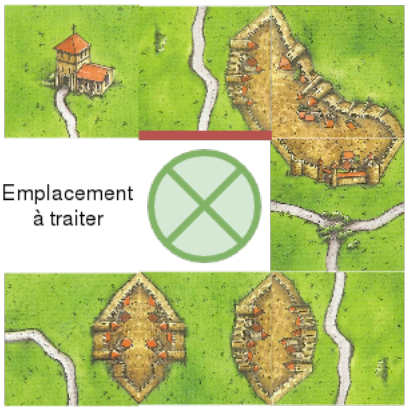
Pour trouver un emplacement possible il faut :

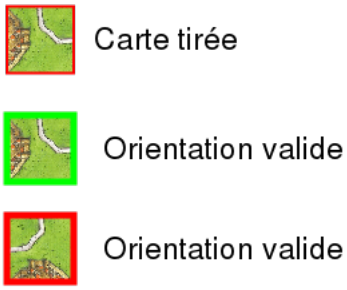
— trouver toutes les tuiles qui ont des emplacement libres
— vérifier que le coté coïncide avec notre tuile pioché
— vérifier que les autres coté s'insère avec leur voisin
— et cela pour les 4 orientations possibles de la tuile piochée



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 lequel il sera effectué des test pour savoir si la tuile est posable ou pas.



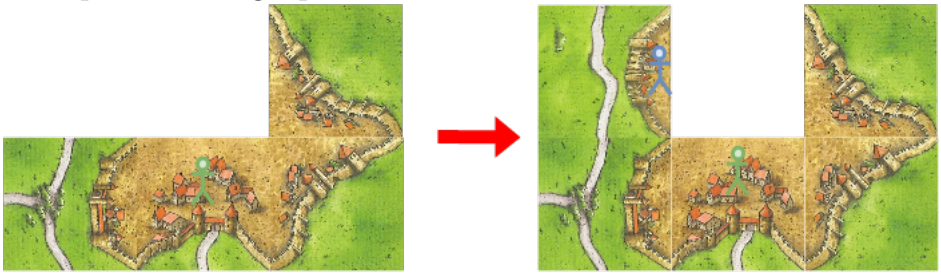




Dans cette situation, on se demande si l'on peut placer la tuile à partir du rebord en rouge (deux orientations sont donc à fortiori possible). En étudiant la topographie de la carte, l'algorithme va remarquer qu'une seul des deux orientations possibles est valide. En effet, l'orientation typé "rouge"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

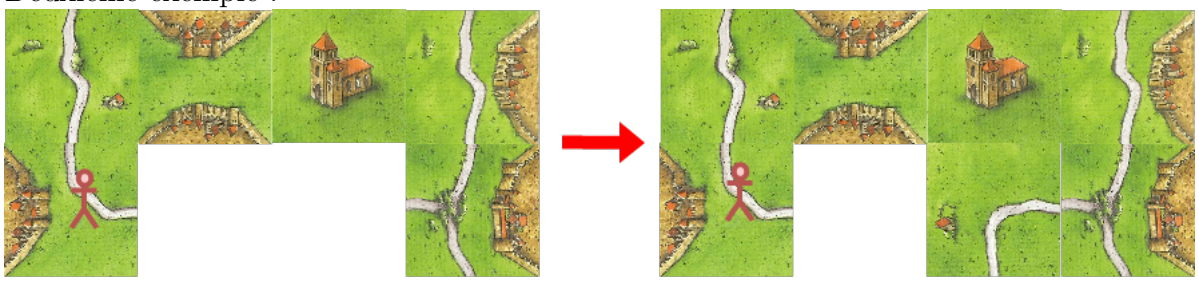
Enfin la sélection de la position finale de la tuile demande de mettre en place une stratégie. Celle-ci devra prendre en compte les coups joués précédemment, la position et l'orientation des tuiles et surtout le positionnement des pions.

Exemple de stratégie possible :



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 à marquer autant de point que le joueur vert avec toutes ses tuiles.

Deuxième exemple :



Une autre stratégie simple est d'empêcher un joueur de finir sa construction. Ici le joueur aura beaucoup de mal a finir sa route à moins d'obtenir la seul tuile qui va à l'emplacement suivant.

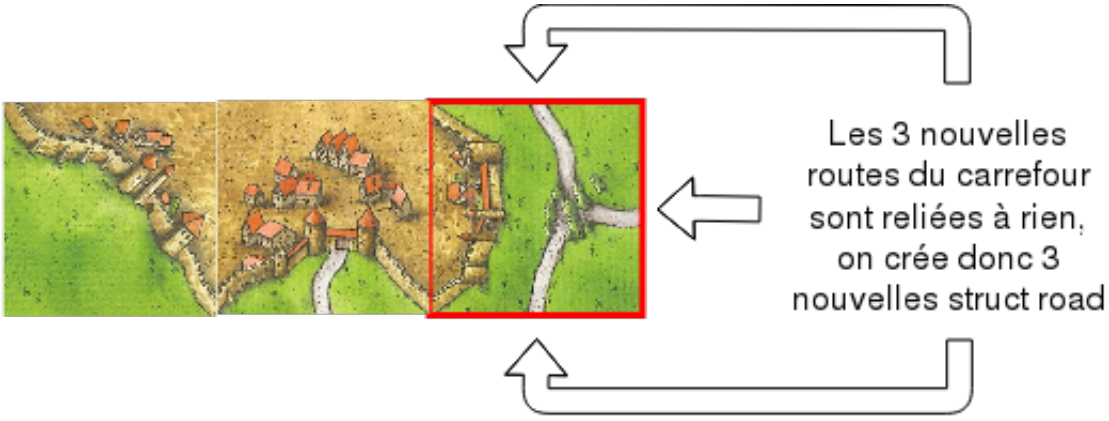
Update

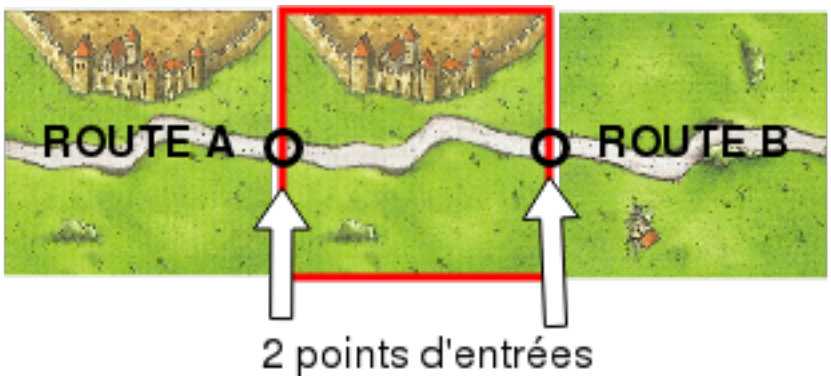
Cette phase est la plus importante, il s'agit de mettre à jour nos structures de données après l'ajout d'une tuile. En effet, nos structures contiennent beaucoup d'information ce qui nous permet de réduire notre complexité en temps et de simplifier le calcule des points. Elle se découpe comme suit :

Mis à jour des routes

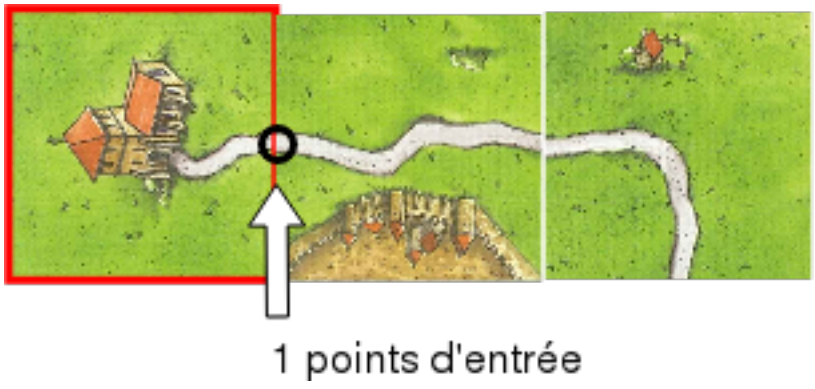
Après avoir rajouté une tuile, on vérifie chaque route qui la compose :
— soit elle n'est relié à rien, et donc on créé une nouvel *struct road* que l'on rajoute à la liste des routes dans la *struct boardgame*.
— soit la route continue une route existante ; il faut donc mettre à jour sa *struct road* :
  — si notre route a deux points d'entrer, c'est un segment, elle ne fait donc que prolonger la route, il suffit donc de la rajouter à la liste des tuile de la *struct road* et fusionner les deux *struct road* si elle a fait la liaison entre de 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*, modifié son paramètre avancement

Exemple de mis en application :





Il faut donc fusionner les 2 routes

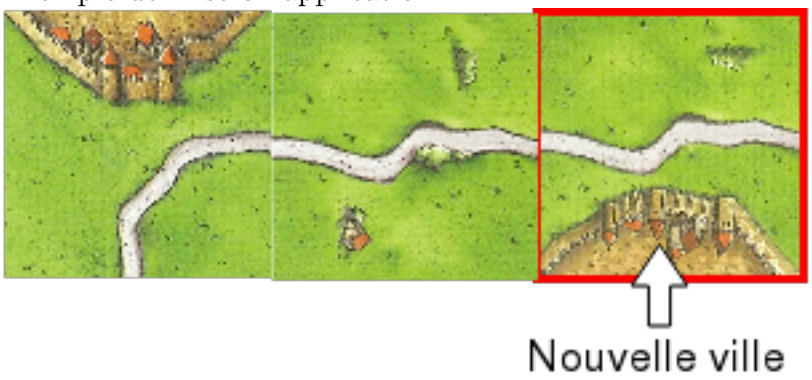


Il faut donc rajouter la nouvelle tuile et marquer le fait que la route à une extrémité maintenant.

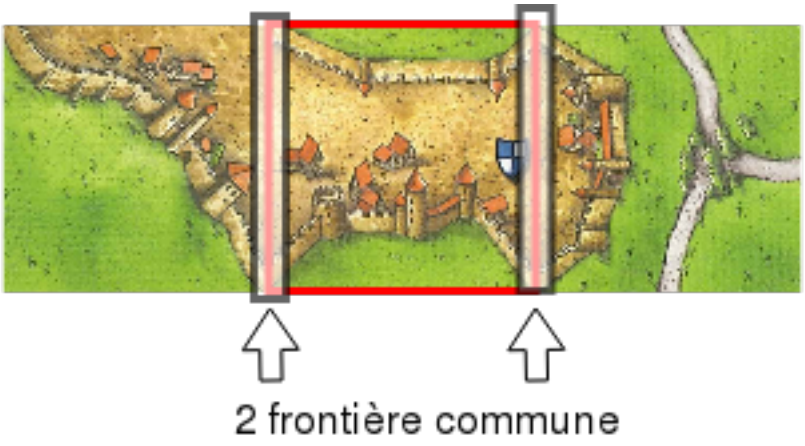
Mis à jour des villes

Après avoir rajouté une tuile, on vérifie chaque ville qui la compose :
— soit elle n'est reliée à rien, donc on créé une nouvel *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 fusion toutes les *struct city* qui se recoupe en une nouvelle
  — il faut aussi fusionner les champs associés aux villes

Exemple de mise en application :



Elle est relié à rien, on crée donc une nouvelle struct city

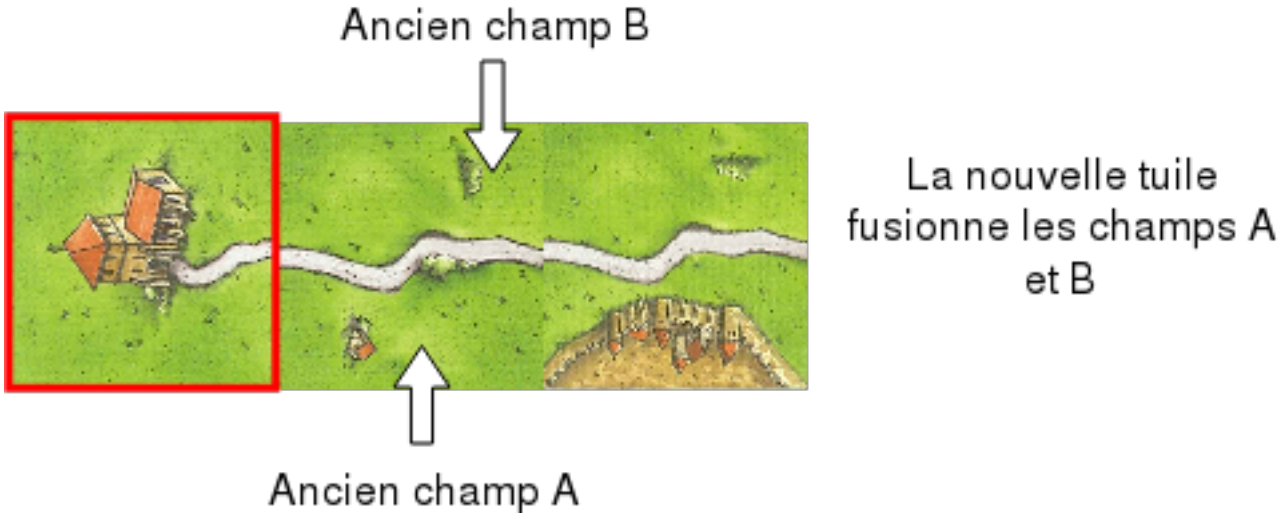


Les 2 villes A et B sont maintenant relié par la nouvelle tuile, ils faut donc fusionner city A et city B.

Mis 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 créé une nouvel *struct field* que l'on rajoute à la liste des champs dans la *struct boardgame*.
— soit le champs 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 fusion toutes les *struct fiels* qui se recoupent en une nouvelle.

Exemple de mise en application :



Pour chaque mise à jour d'une structure, la fusion se fait en temps constant, car on utilise des listes doublement chaîné, et les tableaux contenant la liste des pions et de taille fixe (= le nombre des joueurs).

Score

Cette phase consiste à mettre à jour le score de chaque joueur en même temps que les structures. Ainsi pendant la mise à jour des structures si un bâtiment est fini, on crédite les joueurs du nombre de point correspondant :

- si une route est finie (paramètre avancement = 2), chaque joueur en majorité gagne : la taille de la route (accès constant)
- si une ville est finie (paramètre fini = 1), chaque joueur majoritaire dans la ville remporte : 2 si la taille est égale à 2, ou $2 \times (taille + bouclier)$ accès constant
- les champs sont traités en fin de jeu, avec les abbayes, les bâtiments et routes incomplètes

2.3 Fin du jeu

Une fois que toutes les tuiles ont été placé, 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.

Finalisation

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

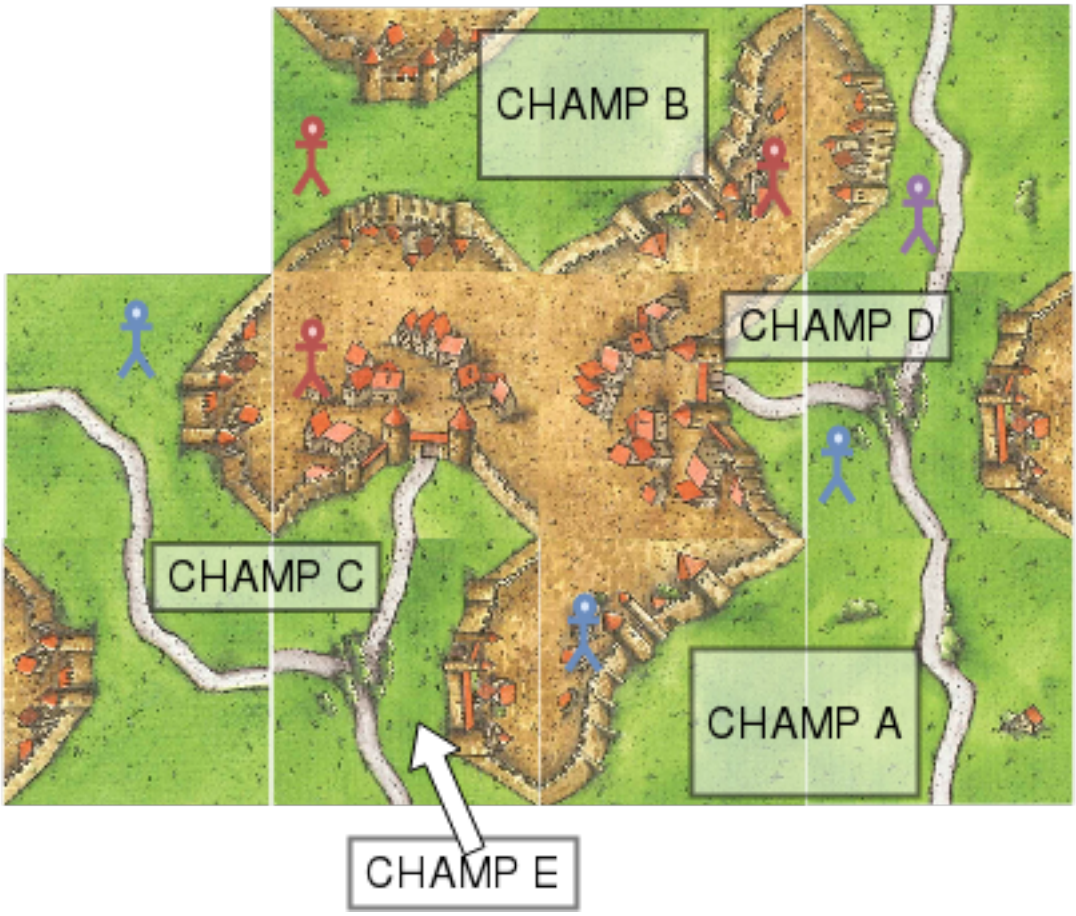
Les points accordés via les routes et les villes sont divisés par deux dans cette situation (1 point par tuile). Pour cela, on a qu'à utiliser la taille de la *struct road* .

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 champ en contacte directe avec cette ville gagne trois points. En cas d'égalité, les deux joueurs gagnent trois points (les n joueurs possédant le plus de champ pour être plus précis).

Mais grâce à la *structure city*, rien de plus simple. Les champs en contact avec une ville sont notés 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 contacte avec la ville). Les mêmes règles d'égalités sont appliquées.

Il ne reste plus qu'a traité les abbayes en comptant le nombre de tuiles autour.



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 champ autour de la ville, il gagne donc trois points, ce qui n'est pas le cas de ses adversaires.

2.4 Conclusion

Avec cette réalisation, nous avons mis l'accent sur l'optimisation en temps, en limitant au maximum les parcours de graphes. Au détriment, de la complexité spatiale.