POLYTECH NANTES

École polytechnique de l'université de Nantes

Mini-Projet C++

Rapport du projet

`Dicewars`

Corentin Banier

Ibrahim El Umari

Omar Jeridi

Bilal Zaraket

2022 - 2023

1 Modèles d'objets

1.1 Générateur de carte

La génération de la carte est inspirée du diagramme de Voronoï.
« En mathématiques, un diagramme de Voronoï est un pavage (découpage) du plan en cellules (régions adjacentes) à partir d'un ensemble discret de points appelés germes. Chaque cellule renferme un seul germe, et forme l'ensemble des points du plan plus proches de ce germe que d'aucun autre. La cellule représente en quelque sorte la zone d'influence du germe. » (https://fr.wikipedia.org/wiki/Diagramme_de_Vorono%C3%AF)

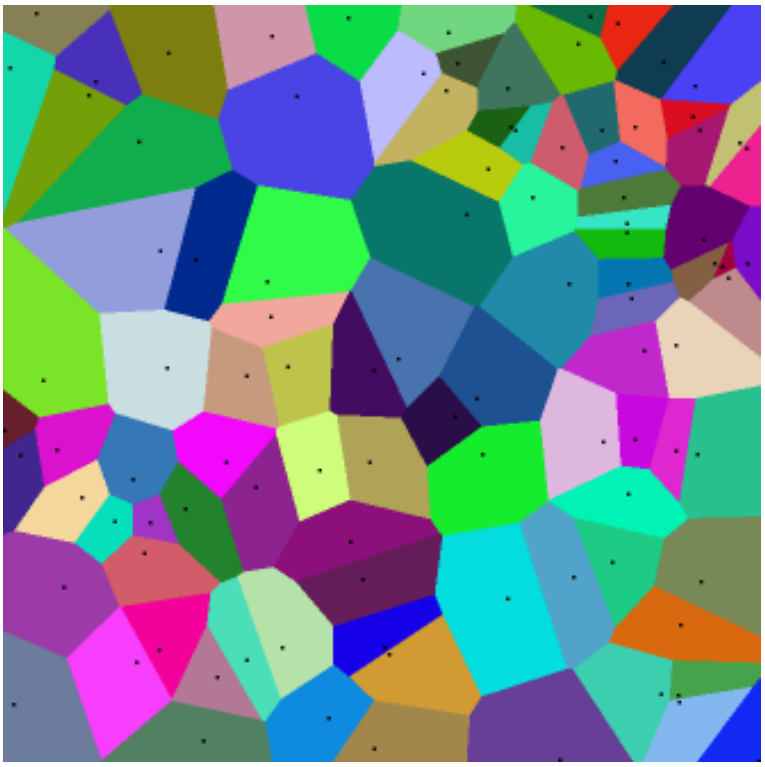

FIGURE 1 – Diagramme de Voronoï

1.2 Stratégie

La stratégie que nous avons implémenté consiste à calculer la plus grande région du joueur, puis de lister toutes les cibles possibles. Nous faisons cela, car nous cherchons à maximiser le nombre de dés que l'on peut recevoir, et par conséquent, nous partons de notre plus grande région. Nous trions cette liste à l'aide des facteurs suivants :

1. La différence de dés, nous favorisons les cibles faciles.
2. Si la source d'attaque se situe dans la plus grande région alors on augmente la priorité.

Pour résumer, la stratégie attaque les cibles les plus vulnérables et cherche à maximiser la plus grande région.

2 Spécificités dans l'implémentation

2.1 Générateur de cartes

2.1.1 Placement des germes

Nous commençons par placer les germes « Seeds » dans la structure `Map` *aléatoirement, en choisissant le nombre de* `seeds` *comme étant le « NbR » puisque le nombre de germes représente aussi le nombre de régions obtenues à la fin.*

```
    // Choose a random set of seeds for the Voronoi diagram
    // by choosing always random seed we garuntee that its always a random map
    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<std::mt19937::result_type> distX(0, mapWidth - 1);
    std::uniform_int_distribution<std::mt19937::result_type> distY(0, mapHeight - 1);
    const int numSeeds = c;  // number of seed also reflects number of regions
    std::vector<Territory> seeds;
    for (int i = 0; i < numSeeds; i++) {
      seeds.push_back(Territory(distX(rng), distY(rng), i));
    }
```

2.1.2 Assignation des cellules aux régions

Nous appliquons la simulation du diagramme de Voronoï afin d'assigner chaque cellule aux germes les plus proches.

```
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapHeight; y++) {
            double minDistance = std::numeric_limits<double>::max();
            int closestSeed = -1;
            for (int i = 0; i < numSeeds; i++) {
                double d = distance(map[x][y], seeds[i]);
                if (d < minDistance) {
                    minDistance = d;
                    closestSeed = seeds[i].region;
                }
            }
            map[x][y].region = closestSeed;
        }
    }
```

2.1.3 « Perçage » de la carte

Les trois fonctions suivantes sont nécessaires afin d'ajouter du vide sur la carte : `removeRegion`, `findBorders` *et* `areNeighbors`

```cpp
void removeRegion(std::vector<std::vector<Territory>>& map, int region);

std::vector<Territory> findBorders(const std::vector<std::vector<Territory>>& map);

bool areNeighbors(const std::pair<unsigned int, unsigned int>& t1, const std::pair<unsigned int, unsigned int>& t2);
```

Ces fonctions nous permettent de supprimer des régions tout en gardant la `Map` *convexe. Puis, nous supprimons au hasard des régions (* $\frac{nb\_regions}{5}$ *). Nous vérifions qu'aucune région ne soit isolée.*

2.1.4 Résultat

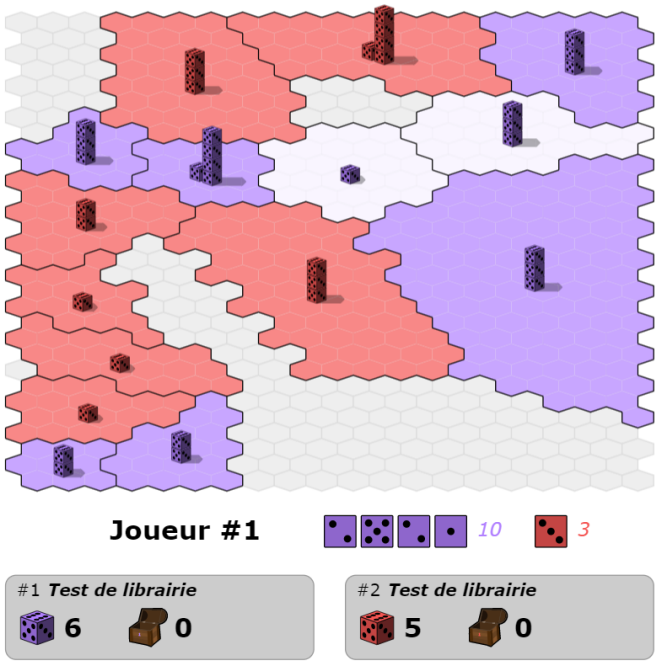

FIGURE 2 – Exemple d'un carte

2.2 Stratégie

Afin de regarder si un coup est autorisé, nous utilisons la fonction `isLegalMove`*.*

1. *La cellule de départ* `from` *doit avoir strictement plus de un dé.*
2. *La cellule de départ* `from` *a au moins le même nombre de dés que dans la cellule attaquée* `to`*.*

```cpp
bool isLegalMove(SMap& map,int Id,int from, int to);
```

Structure contenant les cibles possibles, cette queue est triée par ordre de priorité, ainsi décrit dans la section 1.2.

```
std::priority_queue<Attack,std::vector<Attack>,LessThanByDices> targets;
```

La fonction ***getPossibleTargets*** *permet de remplir* ***priority_queue*** *en prenant en compte la priorité.*

```
void getPossibleTargets(SMap& Map,int Id, std::set<int> biggestRegion, std::priority_queue<Attack, std::vector<Attack>, LessThanByDices>& targets);
```

Le tableau `regions` est très important dans les trois fonctions suivantes :

1. *Divise les cellules appartenant au joueur d'une région (groupes de cellules connectées les unes aux autres et appartenant au même joueur) Pour éviter de tester une zone plusieurs fois, nous faisons appel à la fonction récursive « explore_each_region_alone » pour atteindre toutes les cellules de chaque région.*

```
        void AggressiveAttackStrategy::Divide_into_regions();

```

2. *La fonction récursive* ***explore_each_region_alone*** *prend en paramètre un terrain, et l'ajoute dans la région* ***region_number****. Puis, elle parcourt ses voisins afin de en trouver lesquels ont le même propriétaire et celles qui ne sont pas explorés.*

```
        void AggressiveAttackStrategy::explore_each_region_alone(int land, int *explored, int owner, int region_number);

```

3. *La fonction* ***get_largest_region*** *boucle sur toutes les cellules dont nous sommes propriétaire et détermine les autres régions qui ont le plus grand nombre de cellules connectées. La fonction retourne la plus grande région.*

```
        int AggressiveAttackStrategy::get_largest_region();

```

3 Analyse

3.1 Analyse statique

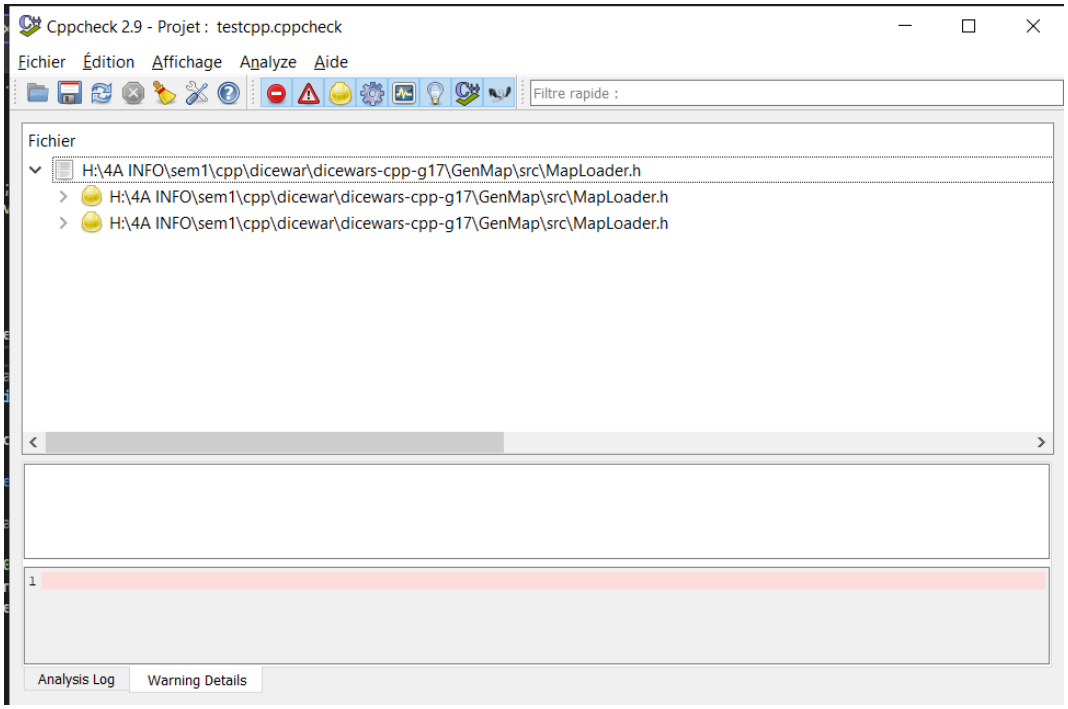

FIGURE 3 – Résultat de `Cppcheck`

3.2 Analyse runtime

Notre stratégie gagne toujours lorsqu'elle commence une partie.

4 Remarques

Nous avons également essayé de mettre en œuvre d'autres stratégies et d'ajouter d'autres fonctionnalités à notre stratégie, mais nous n'avons pas réussi à les implémenter. Les voici :

1. Mise en place d'un algorithme pour connecter la région la plus grande à la région la plus proche de celle-ci afin d'augmenter la récompense en fin de tour (déjà implémenté, mais pas placé sur la branche `master` à cause de bugs non résolus).

   Cette stratégie comporte deux approches :

   (a) La première approche est la plus rapide, où elle trouve juste le nombre minimum de terres nécessaires afin d'atteindre une autre région.

   (b) La deuxième approche passe par la route ayant le nombre moyen minimum de dés.

2. Un autre algorithme que nous avons implémenté, mais que nous avons ensuite supprimé, car il n'était pas assez efficace pour améliorer les performances de notre stratégie. Dans cette précédente stratégie, nous avions ajouté un facteur de faiblesse qui faisait que

l'algorithme terminait son tour uniquement si le nombre de terrains ayant un faible nombre de dés dépassait une certaine valeur seuil.