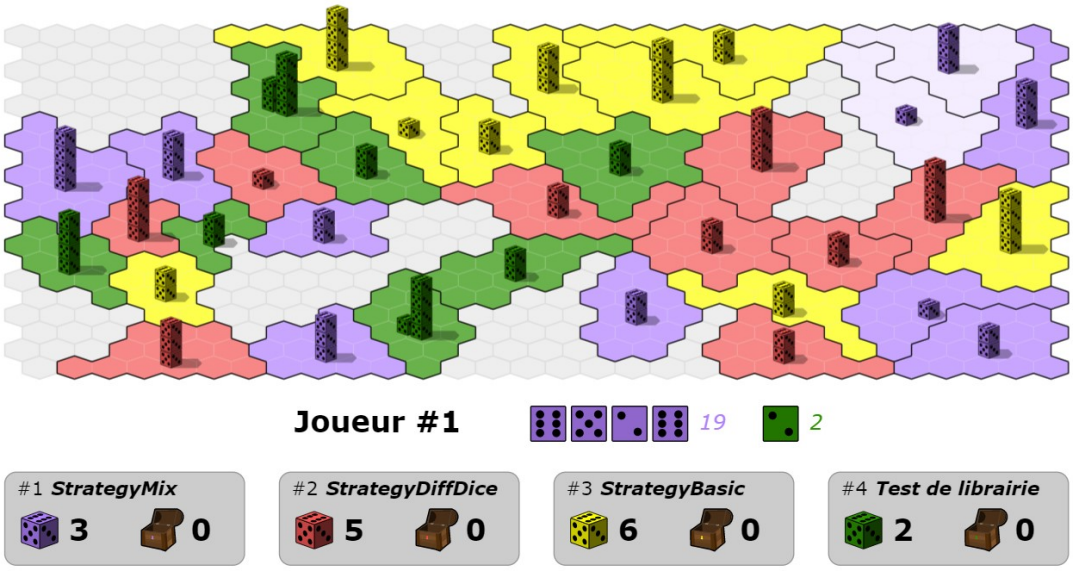


Mini-Projet-C++

INFO4-2023

Dice Wars

Chenrui ZHU , Xiangyong LI , Xianxiang ZHANG , Mamisoa RANDRIANARIMANANA

1 Introduction

1.1 Contexte

Ce projet a été demandé par M. Picarougne dans le cadre du sujet C++. Le but est de développer un système autour du jeu Dice Wars qui se décompose en 2 parties :

- Génération de carte de jeu aléatoire
- Création d'une ou plusieurs stratégies pour assurer le bon déroulement du jeu et gagner le jeu

Le but principale est de jouer contre différentes stratégies dans une carte.

1.2 Présentation de Dice Wars

Dice Wars est un jeu de stratégie au tour par tour, plus ou moins similaire à Risk. Il repose principalement sur une combinaison de stratégies, mais aussi sur la chance des lancers de dés. Il est important de noter que plus la stratégie d'un joueur est raffinée, moins elle dépend de la chance.

1.3 Règle

Pour attaquer, un joueur peut sélectionner son territoire, puis choisir un territoire adverse bordant la zone sélectionnée. Celui qui lance le plus de dés gagne. L'attaquant doit lancer un nombre plus élevé que le défenseur. Un joueur peut attaquer plusieurs fois par tour. A la fin du tour, le joueur reçoit un nombre de dés égal au plus grand territoire connecté qu'il contrôle. Les dés sont ajoutés au hasard. Ensuite, c'est au tour du joueur suivant.

1.4 Caractéristiques

- 3 tailles de cartes
- Différentes stratégies d'IA
- Jusqu'à 7 joueurs d"IA

2 Création de la map

2.1 Création de la carte

Pour implémenter une carte, il est nécessaire d'interpréter la position de la grille hexagonale dans la matrice. Cette interprétation variera en fonction du décalage des lignes paires ou impaires de la cellule.

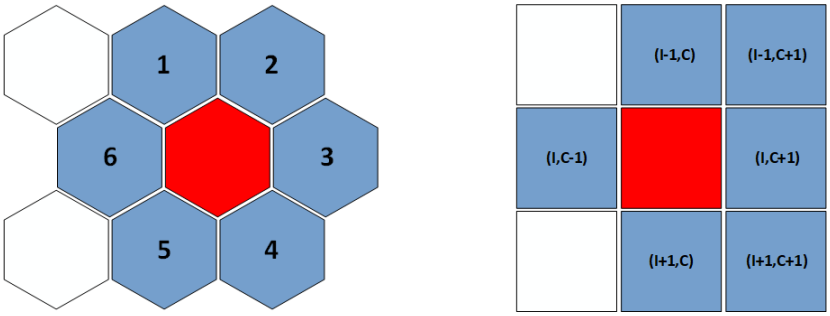


Figure 1: Pattern de voisinage - ligne impaire

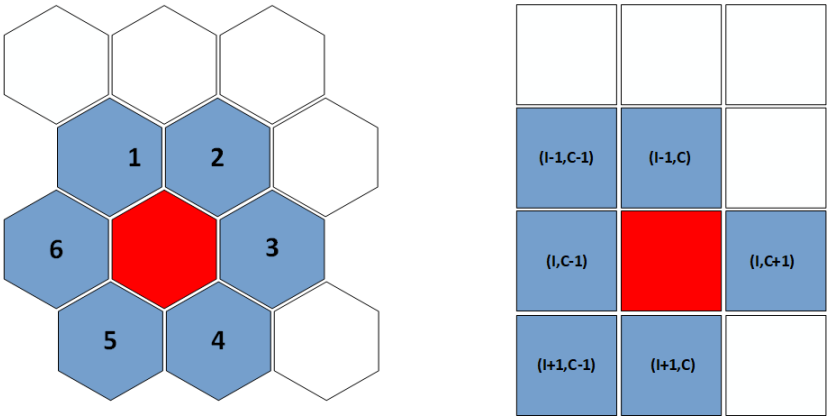


Figure 2: Pattern de voisinage - ligne paire

L'initialisation de la carte commence par placer aléatoirement un nombre de cellules égal au nombre de territoires souhaité. A ce stade, chaque cellule représentera chaque futur territoire. Nous développons chaque cellule par cellule, puis région par région.

Constructeur

Premier attribut r : les régions représentant la carte du jeu

nbR : nombre de régions pleines

nbRV = nombre de zones vides

nbL = nombre de lignes

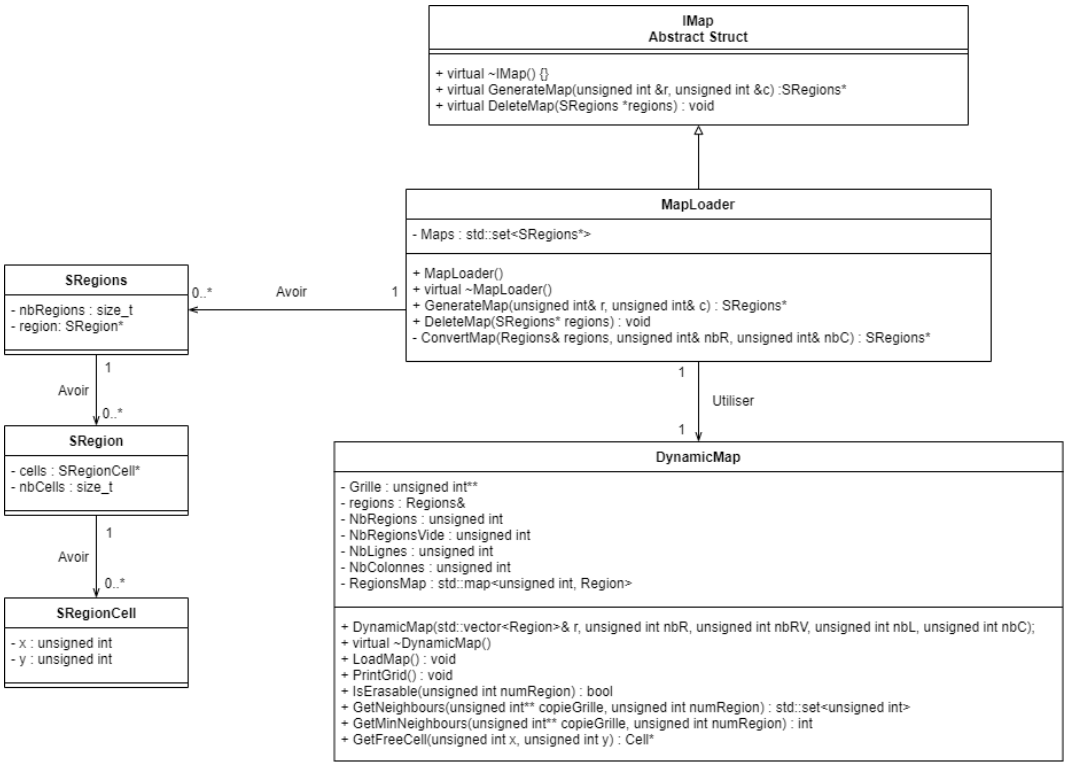
nbC = nombre de colonnes de la grille de jeu 2d.

- Création d'une matrice 2D de nbL lignes et nbC colonnes pour représenter la grille de jeu.Les cellules sont initialisées à 0 lorsque le jeu est initialisé (aucune cellule n'appartient encore à

une région).Charge la carte dans l'attribut "régions" à partir des différents critères initialisés dans le constructeur.Utilisez les fonctions rand et srand pour générer de manière aléatoire la première position de cellule de différentes régions et étendre la plage de régions en fonction des cellules adjacentes.

- Pour chaque région, on cherche un voisin de la dernière cellule ajoutée à la région, on itère jusqu'à trouver une cellule ayant un voisin et ajout de la cellule pour la région, libération mémoire de la cellule.
- Pour une région, il est jugé si la région est effaçable ou non. Autrement dit, si une fois cette région supprimée, toutes les régions restantes forment toujours un composant connexe.
- Pour rechercher une cellule adjacente à une cellule de la plage, nous appliquons GetFreeCell aux cellules de la plage jusqu'à ce que la méthode renvoie une cellule libre adjacente à une cellule de la plage. La cellule libre se voit attribuer la valeur k, où k est la kème région. Les méthodes Cell* GetFreeCell(x,y) prennent des coordonnées comme paramètres et renvoient une cellule voisine libre (6 voisins) ou un pointeur nul. Une cellule est une structure d'abscisse x et d'ordonnée y.

Diagramme de classe et les méthodes de la génération de la carte



La méthode IsErasable(numRegion) peut aider à résoudre certains problèmes de génération de régions non valides. Cette méthode permet de récupérer le nombre minimum de régions accessibles depuis chacun de ses voisins à l'aide de GetMinNeighbours(numRegion). Si la valeur minimale est égale au nombre actuel de régions - 1, nous renvoyons vrai, c'est-à-dire que la région peut être supprimée.

GetMinNeighbours() : pour chaque voisin de la région numRegion, nous empilons la région actuelle et récupérons (en utilisant GetNeighbours()) et empilons ses voisins, nous les pop tant que la pile n'est pas vide, et pour chaque région, nous obtenons ses voisins , nous les empilons. L'idée est similaire à la récursivité, mais itère en utilisant une pile pour obtenir les voisins de chaque région traversée.

GetNeighbours(numRegion) : renvoie un ensemble de numéros de régions voisines pour une région. Pour chaque cellule d'une région, on regarde chaque bloc de 6 voisins : si un bloc n'est pas vide et pas la région courante, on l'ajoute

2.2 UML pour génération de la carte

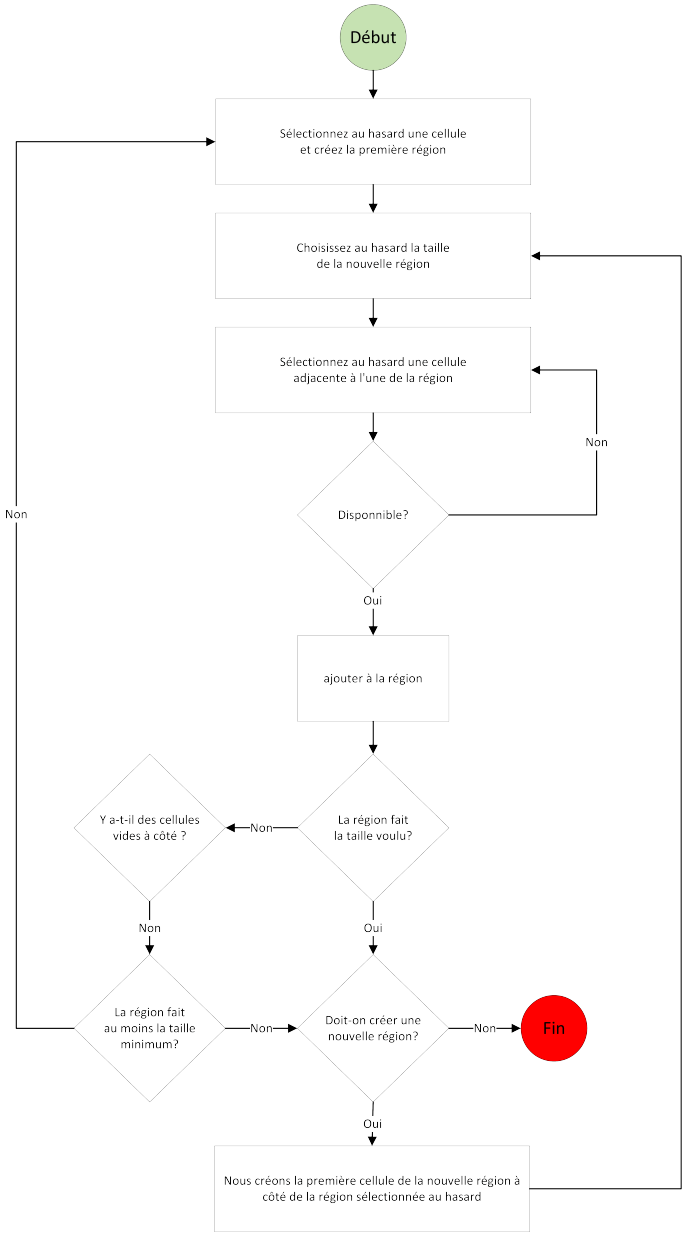


Figure 3: UML pour Génération de map

3 Stratégies

Il ne faut jamais attaquer une cellule si l'on a un nombre de dés inférieur à celui de la cellule que l'on attaque, et en cas d'égalité, défendre la cellule que l'on a déjà gagnée. Une attaque serait préférable quand on est presque sûr de gagner est essentiel, mais d'un autre côté, même si notre nombre de dés est supérieur au nombre de cellules attaquées, il vaut mieux attendre que de risquer de perdre le duel.

Nous avons utilisé deux types de stratégie en fonction de ce que nous avons compris, et ils ont fini par se fondre dans une troisième stratégie qui est la stratégie finale que nous avons choisie.

3.1 Stratégie basique

Il s'agit d'une stratégie de base, mais suffisamment efficace pour nous aider à jeter les bases du développement d'autres stratégies. Lorsque l'adversaire du quartier a moins de dés que le joueur du tour en cours, la stratégie consiste à lancer une attaque. À l'inverse, lorsque la position est désavantageuse, cette stratégie rend le joueur du tour en cours constamment sur la défensive et dans une certaine mesure vulnérable.

Lancez une attaque lorsqu'il y a un avantage dans le nombre de dés et cherchez la cellule avec la plus grande différence de dés. Lorsque le nombre de dés est égal, il faut souvent considérer le nombre de tours non exploités, le nombre de territoires de l'adversaire et le risque d'attaquer.

1. Dès que plus de deux tours sont inopérants, la stratégie prend le risque de lancer une attaque.
2. Si le nombre de tours non exploités est inférieur ou égal à 2, si l'adversaire dispose d'un petit nombre de territoires, l'attaque est lancée directement
3. Sinon, si une attaque réussie entraîne l'acquisition d'autres territoires adjacents de votre côté, l'attaque est lancée
4. Sinon, si le nombre de dés attribués pour une attaque réussie est tel que le nombre de dés dans votre propre territoire peut être rempli dans une plus grande mesure, alors l'attaque est lancée

Comme on peut le voir, cette stratégie a un fort désir d'attaquer et est adaptée pour perturber le rythme de l'adversaire. Bien qu'il y ait une réflexion sur la disposition des cellules après l'attaque, elle ne semble pas encore assez détaillée et est plutôt risquée.

3.2 Stratégie de la différence de nombre de dés

L'idée est d'effectuer un calcul de score en se concentrant sur le nombre de voisins ennemis à la cellule qui attaque ainsi que leur nombre de dés et le nombre de voisins ennemis à la cellule qu'on attaque et leur nombre de dés, au moyen d'une fonction séparée. Une fois que le score de l'attaque à effectuer est ¿ 0, il nous indique quelle est la meilleure attaque. Le contraire met fin au round.

Si l'attaque est réussie:

- Pour la cellule attaquée, si elle a des cellules voisines qui ne font pas partie de la cellule attaquante, le score dépend de la différence entre son nombre de dés et (nombre de dés des cellules voisines / nombre de cellules voisines). Inversement, le score ne dépend que de son propre nombre de dés. Ce score détermine la situation autour de la cellule attaquée.

- Pour la cellule attaquante, le score dépend de la différence entre après et avant l'attaque. Avant l'attaque : la différence entre (nombre de dés dans la cellule voisine / nombre de cellules voisines) et son nombre de dés; Après l'attaque : (nombre de dés dans les autres cellules voisines / nombre d'autres cellules voisines) - 1

- La note finale est l'addition des deux parties

Si l'attaque échoue:

- Seul le score de la cellule attaquante est pris en compte.

D'autre part, la probabilité de succès est approximativement prédite en utilisant (nombre de dés dans la cellule attaquante - nombre de dés dans la cellule attaquée + 1) / nombre de dés dans la cellule attaquante. Bien sûr, nous devons reconnaître que cette probabilité n'est pas une vraie probabilité.

Enfin, le score de réussite * probabilité de réussite + le score d'échec * probabilité d'échec donne le score final, et s'il est supérieur à 0, l'opération est simplement considérée comme possible.

Nous pouvons voir que cette stratégie fait que le côté qui est désavantagé ou même réduit à quelques cellules a tendance à être sur la défensive. Cependant, le choix d'attaquer ou non peut ne pas être aussi simple que dans la première stratégie.

3.3 Stratégie hybride

Cette stratégie est principalement un algorithme hybride tenté dans un but d'expansion. C'est-à-dire que la stratégie basique est utilisée lorsqu'il y a un avantage dans le nombre de dés, et la stratégie de la différence est utilisée lorsqu'il y a un nombre égal de dés. L'avantage de cette méthode est qu'elle maximise l'efficacité de l'attaque et donne également à la cellule une théorie mathématique pour soutenir son choix si elle se trouve dans une impasse.

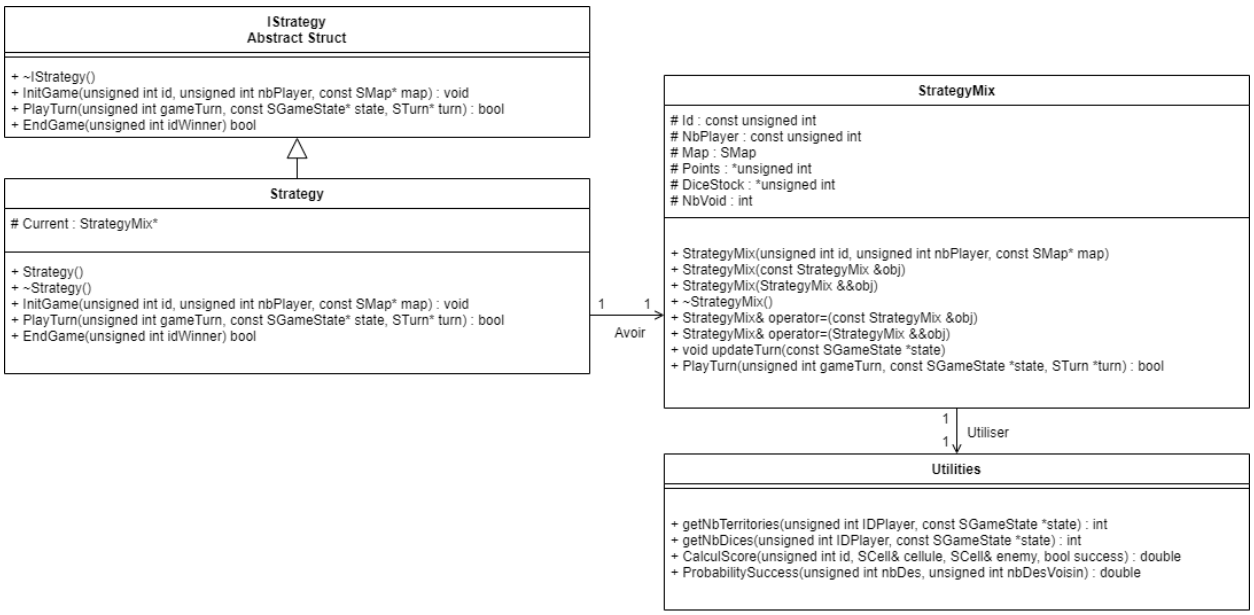
Dans le cas d'un nombre égal de dés, la stratégie privilégiera le jugement du nombre de tours sans opération.

- Une fois qu'il est supérieur à une certaine valeur, elle attaque sans hésiter

- S'il est supérieur à une valeur intermédiaire, si le territoire restant de l'adversaire est inférieur à 2, alors elle attaque. Sinon si l'attaque réussit et qu'il y a un propre territoire adjacent, alors elle attaque.

- Sinon, la stratégie de calcul du score est appelé

Diagramme de classe de la stratégie hybirde



4 Spécification

4.1 Utilities.h

Il contient toutes les fonctions spéciales nécessaires à la stratégie pour améliorer la lisibilité du code.

- getNbTerritories() - retourne le nombre de territoires du joueur actuel
- getNBDices() - retourne le nombre de tous les dés pour le joueur actuel
- CalculScore() - calcule un score d'attaque, basé sur le nombre de voisins de la cellule attaquante, son nombre de dés, et le nombre de voisins ennemis qu'elle a et leur nombre de dés
- ProbabilitySuccess() - la probabilité virtuelle qu'une cellule attaque avec succès, en fonction du nombre de dés dans la cellule qui attaque et du nombre de dés dans la cellule qui est attaqué.

4.2 DynamicMapGenerator.h

Il contient la classe DynamicMap avec un structure Cell pour générer une carte par nous-même.

- GetNeighbours() - pour une région, renvoie l'ensemble de ses régions voisines (en nombre).
- GetMinNeighbours() - pour une région, regarde pour chacune de ses régions voisines le nombre de régions accessibles depuis cette région voisine. Parmi ces nombres, la fonction retourne celui le plus petit pour confirmer que chaque région permet bien d'accéder à chacune des autres (connexe)

- GetFreeCell() - renvoie une cellule disponible
- IsErasable() - pour une région, renvoie un résultat vrai si, une fois la région supprimée, les régions restantes forment toujours un composant pertinent, sinon renvoie un résultat faux

5 Conclusion

L'optimisation possible de la stratégie peut être un utilisation du nombre maximum de territoires adjacents du joueur actuel. Ce nombre peut être obtenu par DFS, BFS ou A*, qui va permettre de un choix offensifs et défensifs intelligent.

Le projet est constitué de deux petits blocs ficelés ensemble et la version de base comprend un module de test très bien développé. Nous avons pu soit simplement modifier la stratégie, soit programmer le générateur de cartes en premier, ce qui a permis à plusieurs d'entre nous de mieux se répartir le travail et de réduire largement le problème d'interférence dans les projets collaboratifs.

Il devrait être plus important de réfléchir à la manière de résoudre le problème plutôt qu'à la manière de le coder. Les bases du cours sur la théorie des graphes et les algorithmes nous ont aidés à comprendre et à analyser les problèmes dans une certaine mesure. De plus, ce projet nous a permis d'appliquer les leçons de C++ que nous avions apprises pendant le semestre. Se réunir pour développer une stratégie pour un jeu était très amusant.