Mini-Projet C++

Introduction

Ce projet est un projet de développement qui consiste à construire la carte et la stratégie d'un jeu : Dicewars.

Il faut savoir que nous avons commencé le développement à partir d'un projet déjà existant. Nous avions une Map statique et une stratégie simple.

Nous avons tout d'abord réfléchie à la génération aléatoire de Map. Ensuite, après avoir développé des maps différentes de manière aléatoire, on a développé la stratégie.

Génération de Map :

On instancie un objet array 2D remplit de 0 qu'on nomme map. Dans cette partie on utilise une structure de données `class RegionV`. Cette class est un vecteur contenant à la fois la cellule et les cellules voisines. RegionV est une region et un utilise un vecteur de RegionV que l'on nomme Region**s**V pour contenir l'ensemble des régions de la map.

Ensuite, on utilise une classe nous permettant de stocker toutes les coordonnées de la qu'on veut générer.

On remplit map puis on la troue.

Remplissage de la map :

Afin de remplir la map, on utilise une pair d'entier du type «pair<unsigned int, unsigned int>» qu'on nomme Coord à l'aide de :

using Coord = pair<unsigned int, unsigned int>

Cette pair d'entier est utile pour rechercher à travers la fonction RechercherMap, l'élément qu'on a pas remplit dans la map.

Ensuite on rentre dans une boucle while. La condition étant « tant que la première valeurs de coord est différente de 1000 ». C'est une manière de vérifier tant que le tableau n'est pas plein. On créer des régions et on actualise les régions voisines. Ensuite, on remplit les régions de cellules.

Trouer la map :

Après la phase de remplissage d'une map, on a la phase de trouage. On supprimme les régions trop petites tout en s'assurant que la map reste, d'un point de vue mathématique, un graphe fortement connexe. Cela implique qu'on ait qu'une seule map où toutes les régions sont reliées entre elles.

Stratégie :

Comment a-t-on trouvé notre stratégie :

Concernant la stratégie, le temps de réflexion fût plus long. Afin de mieux comprendre le jeu et les différentes façons de gagner, nous avons chacun de notre côté effectué un grand nombre de parties contre l'ordinateur (que ce soit contre 1 ou 7 joueurs) sur le jeu en ligne https://www.gamedesign.jp/games/dicewars/ .

Nous nous sommes rapidement rendus compte de l'importance d'avoir un territoire composé de cellules adjacentes afin d'optimiser le nombre de dés obtenus à chaque fin de partie (fonction du nombre de cellules adjacentes). Mais aussi de l'importance des coins afin de sécuriser son territoire (le nombre de voisins d'une cellule a donc une importance capitale qu'on retrouvera par la suite dans notre stratégie finale).

Et surtout une règle de base mais surement la plus importante : qu'il ne fallait jamais attaquer une cellule si notre nombre de dès était inférieur à la cellule qu'on attaque et qu'en cas d'égalité c'est la cellule qui défend qui remporte la

confrontation. Il est donc primordial d'attaquer lorsqu'on est quasiment sûr de gagner, et l'on s'est donc rendu compte que parfois, même si notre nombre dés est supérieur à ceux de la cellule attaquée, il valait mieux attendre que prendre le risque de perdre le duel et de se retrouver à la merci des cellules ennemies.

Quelle stratégie :

Ayant tout cela en tête, nous nous sommes demandés comment faire pour mettre en place une stratégie combinant tous ces aspects du jeu et s'adaptant à la situation. C'est là que nous est venu l'idée d'un score calculé à chaque tour et qui nous indiquerait qu'elle est la meilleure attaque à effectuer, et ce tant qu'une attaque à effectuer a un score > 0. Lorsque le score n'est pas > 0, le tour s'arrête alors. Grâce à cette stratégie, lorsqu'aucun score d'attaque n'est > 0, on passe directement notre tour, car il vaut parfois mieux d'attendre un ou plusieurs tours, plutôt que de se jeter dans la bataille et perdre inutilement des dès.

Pour être pertinent, notre score devait reprendre les éléments vus dans la première partie, à savoir : le nombre de dés de la cellule qui attaque, le nombre de dés de la cellule qu'on attaque, le nombre de cellules alliées adjacentes à la cellule qui attaque, si la cellule attaquée nous permettait d'avoir un coin, 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. Par manque de temps et par complexité, nous avons enlevés les paramètres le nombre de cellules alliées adjacentes à la cellule qui attaque et si la cellule attaquée nous permettait d'avoir un coin pour garder uniquement les autres.

La question est maintenant : comment avec ses paramètres pouvons-nous calculer un score qui reflèterait le plus possible la meilleure attaque à faire ? Pour cela nous avons décomposer le score final en plusieurs parties.

Tout d'abord nous calculons le score de la cellule qui attaque, avant qu'elle effectue son attaque. Ce score est calculé en fonction du nombre de dés de la cellule et du nombre de voisins ennemis à cette cellule et de leur nombre de dés. La formule est la suivante :

score_avant = (nb_dés_voisins / nb_voisins_ennemies) − (cellule.infos.nbDices)

Ce score varie entre -7 et 6 et un score négatif est ici bon signe, puisqu'on le verra, ce score sera soustrait à un autre.

Notre deuxième score est donc le score de la cellule qui attaque, après qu'elle ait effectué son attaque. Ce score vaut 1 si la cellule n'a plus de voisins ennemis après son attaque (il reste alors 1 dé sur la cellule) et vaut la formule suivante lorsqu'elle possède au moins 1 voisin ennemi :

```
score_apres = (nb_des_voisins / nb_voisins_ennemies) - 1 (varie entre 0 et 7)
```

On fait alors la soustraction 2 de ces 2 scores pour obtenir :

```
score = (score_apres - score_avant) (varie entre -6 et 14)
```

Ce score est le score final de l'attaque, dans le cas où celle-ci échoue. En effet lorsqu'elle réussie il faut également prendre en compte le fait qu'on a gagné un territoire (une cellule). On additionne alors au score, un autre score qui sera lui calculé en fonction du nombre de dés de la cellule qu'on a attaqué et du nombre de voisins ennemis à cette cellule et de leur nombre de dés. La formule est la suivante :

```
score = (cellule.infos.nbDices - 1) - (nb_des_voisins / nb_voisins_ennemies)(-7 à 6)
```

Et lorsque la cellule qu'on a attaquée (et prise) n'a pas de voisins ennemis, son score est son nombre de dés.

```
score = cellule.infos.nbDices - 1 (varie entre 1 et 7)
```

Notre fonction de calcul de score `atqCalculScore()` prend donc en paramètre un booléen qui lorsqu'il vaudra « True » indiquera la réussite de l'attaque. Et donc qu'il faudra prendre en compte dans le calcul de score, le score de la cellule attaquée, pour obtenir un score variant entre -13 et 21. Lorsque le booléen vaudra « False », le calcul final s'arrêtera à :

```
score = (score_apres - score_avant)
```

De plus le fait de calculer un score dans le cas où l'attaque échoue, et un autre dans le cas où elle réussit, nous permet d'aller plus loin dans le calcul final de notre score. En effet le calcul final du score prendra compte de la probabilité de gagner l'attaque selon la formule suivante :

```
score = (score_atq_reussi * proba_reussite) + (score_atq_echoue * (1 - proba_reussite))
```

Ainsi, en plus de tous les éléments déjà cités, notre score rendra compte de la probabilité que notre attaque réussisse ou non.

Concernant le calcul de la probabilité, nous avons longuement réfléchis à celle-ci. Mais n'ayant pas le temps et la formule étant particulièrement complexe à trouver nous n'avons pas utilisé la formule de probabilité exacte en ce qui concerne les affrontements au DICEWARS ; nous avons utilisé notre propre formule qui est la suivante :

proba_reussite = `(nbDes - nbDesVoisin + 1) / nbDes`

Cette probabilité est comprise entre 0.125 et 1 car nous n'attaquons jamais si notre cellule a moins de dés que celle qu'on veut attaquer. Cette formule transcrit assez bien le fait que plus la différence de nombre de dés est élevée, plus la probabilité de gagner l'est également. Mais aussi le fait que la probabilité de gagner à 8 dés contre 6 est plus faible que celle de gagner à 3 dés contre 1. Elle n'est cependant pas parfaite mais elle nous suffit pour accentuer le calcul de notre score en faisant bénéficier les attaques avec une forte chance de réussir.

Ainsi, notre fonction `PlayTurn()` va calculer pour chaque attaque possible (càd une cellule qui a une cellule ennemie et plus de dés que celle-ci) un score et elle sélectionnera le meilleur score et donc le meilleur coup à jouer. Lorsqu'un coup est effectué, elle recommencera ce calcul et ainsi de suite jusqu'à n'obtenir aucun score > 0, dans quel cas le tour s'arrêtera.

Conclusion

Le projet est la concaténation de deux projets. Durant le projet, lorsqu'on a finit de programmer le générateur de map aléatoire, on pouvait considérer le jeu qu'on codait comme un assemblage de bloc. Cela met en évidence que ce projet pouvait commencer de différentes façons et se réaliser selon différentes apprôches.

Il était donc important de réfléchir à la manière dont on allait résoudre le problème plutôt qu'à la manière de le coder. Les cours de théorie des graphes et d'algorithmie en S6 ont été très important pour résoudre les problèmes qu'on a eut à résoudre. Néanmoins, avec bon esprit d'équipe et une

participation de tous les membres, on a réussit à mener à bien un projet qui nous a semblé pas si facile de prime abord.

Une connaissance avancée du C++ a été très important ainsi qu'une bonne capacité à résoudre des bugs. Le résultat à la fin est un jeu, avec un générateur de map aléatoire et une stratégie.