MCTS Projet : mini-jeu StarCraft

Lothair Kizardjian

March 2020

Contents

1 IMPORTANT : emplacement du code **3**

2 Introduction **3**

3 Starcraft 2 API **4**

4 Simulateur de combat **4**

5 Les actions du MCTS **5**

6 Un premier MCTS **6**

7 Approche UCT **6**

8 Conclusions **8**

9 Requirements et Tuto d'installation du projet **8**

1 IMPORTANT : emplacement du code

Le code que j'ai écrit se trouve dans "custom_classes/bot_examples.h" à partir de la ligne 261, dans "custom_classes/bot_examples.cc" à partir de la ligne 3283 et dans "examples/DefeatRoachesBot.cpp"

2 Introduction

Le projet consiste à adapter une méthode MCTS à un jeu de notre choix. Connaissant starcraft depuis très longtemps et ayant été stupéfait de ce qu'a pu accomplir DeepMind avec Alpha-Star [1] j'ai voulu tenter ma chance et réaliser un bot utilisant un MCTS pour jouer à un mini-jeu sur Starcraft. En effet essayer de faire un MCTS jouant au jeu complet demanderai beaucoup trop de temps et serait beaucoup trop complexe à réaliser. La difficulté majeure qui se pose est qu'il s'agit d'un jeu en temps réel, il faut donc prendre une décision très rapidement, le MCTS n'a donc que peu de temps pour tourner. Le but de ce mini-jeu "Defeat Roaches" consiste à tuer tous les ennemis à savoir une armée composée de 4 unités Zerg (une des races jouables) : les roachs (figure 1), avec une armée composée de 9 unités Terran (une autre race jouable) : les marines (figure 1).

Figure 1: left : marine and right : roach

Le jeu se déroule sur une carte où toute la vision est disponible (figure 2), on sait où se trouve les deux armées. Un système de score est aussi présent : on gagne des points lorsqu'une unité adverse est tuée et on en perd lorsqu'une des notre est tuée. Un exemple du mini jeu se trouve à l'adresse suivante : `https://www.youtube.com/watch?v=5LJd_5Y6g_o`. Une fois les 4 unités adverses tuées un autre niveau est lancé où il faut encore tuer 4 roachs et certaines unités alliées sont soignées voire de nouvelles unités sont rajoutées pour compenser celles perdues durant le précédent niveau mais je n'ai pas très bien compris comment récupérer et soigner les unités alliées. Pour mon projet j'ai décidé de ne faire que sur un seul niveau donc ces problématiques de restauration de vie et de rajout d'unité pour un niveau suivant ne m'intéressent pas.

Figure 2: Defeat Roaches mini game

Je vais présenter par la suite comment j'ai pris en main les outils mis à disposition par Blizzard [2] pour pouvoir coder un bot pour le jeu ainsi que la bibliothèque développée par un étudiant en thèse [3] [4] qui permettait de simuler des combats plus ou moins précisément, ce qui m'a grandement facilité la tâche.

3 Starcraft 2 API

Comme j'ai pu le dire précédemment Blizzard a mis à disposition toute une api [2] permettant de coder facilement un bot et lancer une partie sur le "vrai" jeu avec le bot codé comme participant contre l'ordinateur. Cela demande donc d'avoir le jeu installé sur l'ordinateur et sous windows car le projet est construit via VisualStudio 2017 (les versions plus récentes vont provoquer des erreurs de build).

```cpp
#include <sc2api/sc2_api.h>

#include <iostream>

using namespace sc2;

class Bot : public Agent {
public:
    virtual void OnGameStart() final {
        std::cout << "Hello, World!" << std::endl;
    }

    virtual void OnStep() final {
        std::cout << Observation()->GetGameLoop() << std::endl;
    }
};
```

Figure 3: Simple bot example

On peut voir sur cette figure un exemple d'un bot très simple qui va afficher sur le terminal "Hello, World!" lorsqu'il sera lancé puis à chaque step affichera diverses information sur les observations de l'état du jeu sur le terminal. Ici aucun contrôle des unités n'est fait, ce bot ne joue pas au jeu. C'est grâce à la surcharge de la fonction 'OnStep()' que le bot peut gérer les unités. Cette fonction est appelée très souvent; de ce que j'ai pu tester c'est au moins tous les centièmes de secondes mais c'est peut être moins, il faut donc un PC assez performant ainsi qu'un MCTS très optimisé pour pouvoir retourner une action à faire dans cet intervalle de temps.

4 Simulateur de combat

Ce qui m'a aussi beaucoup aidé dans ce projet est le simulateur de combat [4] codé par Aron Granberg pour sa thèse sur un sujet similaire [3]. Ce simulateur permet de simuler une bataille entre deux armées avec des résultats très proches de ce qu'il se serait passé dans le jeu. On peut par exemple voir sur la figure 4 le résultat d'une bataille avec comme gagnant l'amrée appartenant au joueur numéro 2.

```
Owner Unit       Health
1     Marine       0/45
1     Medivac    150/150
1     Marine       0/45
2     Roach       58/145
2     Roach      145/145
2     Zergling     0/35

Winner player is 2
```

Figure 4: Simulation example

Cependant ce simulateur ne prend pas en compte un certain nombre de chose qui se trouvent être importantes, pour moi en tout cas. En effet j'ai choisi ce mini-jeu car il est relativement simple comparé au jeu de base qui est bien plus complexe, ici il suffit uniquement de contrôler une armée et d'en vaincre une autre. Le challenge se trouve donc dans le micro-contrôlement des unités à savoir contrôler les unités distinctement, ne pas donner le même ordre à chacune mais donner un ordre plus pertinent en fonction de l'état de l'unité. Pour pouvoir lancer une simulation avec cette Bibliothèque il faut avant tout créer les unités de chaque armée, cependant comme l'auteur de cette bibliothèque a décidé de ne pas utiliser les positions des unités afin de simplifier les simulations il n'est pas possible de simuler les déplacements des unités via cette simulation. Je pourrai bien entendu modifier le code mais cela m'aurai pris trop de temps j'ai donc décidé de l'utiliser tel quel. J'ai cependant bien ajouté un attribut 'Position' à cette classe car j'en ai tout de même besoin pour les actions du MCTS. Ce simulateur sert avant tout d'heuristique pour estimer la valeur d'un état lorsqu'il n'y a plus de temps pour dérouler le MCTS. Il permet aussi de calculer les dégâts que chaque unité peut infliger aux unités adverses (en effet chaque unité possède des caractéristiques qui lui sont propres, par exemple les roachs sont de type 'blindés' ils prennent moins de dégâts de la part des unités de type 'léger' comme les marines), ce qui me permet de coder les différentes actions du MCTS. Finalement il me sert à détecter la fin de la partie grâce à des fonctions qui m'indiquent si les armées peuvent encore mutuellement s'attaquer. Si elles ne peuvent plus s'attaquer mutuellement (soit une des deux armées est anéantie soit il reste des unités dans chaque groupe qui ne peuvent pas s'attaquer) alors le joueur avec le meilleur 'outcome' remporte la bataille (celui qui a subit le moins de pertes).

5 Les actions du MCTS

En terme de complexité et de nombre d'actions possibles Starcraft se démarque (en effet cet espace d'actions possibles est continu), une action peut concerner une unité, un groupe d'unités, un bâtiment, un groupe de bâtiments ... le nombre d'actions augmente exponentiellement à mesure que la partie avance et que le nombre d'unité augmente. Bien que ce mini-jeu ne soit pas aussi complexe qu'une partie normale il est possible de donner une action à chaque unité sous notre contrôle ou bien à chaque combinaison d'unités qu'il est possible de former avec 9 marines. Les actions peuvent être : de déplacer l'unité ou les unités sélectionnées sur la carte et, bien qu'elle soit assez petite, le nombre de positions possibles pour chaque unité est lui aussi exponentiel; d'attaquer une unité qu'il faut donc d'abord sélectionner (le nombre de combinaisons attaquant/attaqué est lui aussi très grand). J'ai donc décidé de choisir quelques actions représentatives qu'un joueur pourrait faire naturellement : attaquer l'ennemi qui a le moins de points de vie (chaque unité alliée attaque le même ennemi), attaquer l'ennemi le plus proche (chaque unité alliée peut attaquer un ennemi différent car l'ennemi le plus proche pour une unité n'est peut être pas le même que pour une autre unité) et finalement faire reculer l'unité alliée qui a le moins de points de vie afin qu'elle ne soit plus à portée d'attaque des ennemis. C'est surtout cette dernière action que je voudrai voir le bot utiliser au cours de la bataille pour micro-gérer les unités dont les points de vie tombent assez bas.

Je vais donner un peu plus de détail sur chaque action. Je n'ai pas fait d'action 'déplacer une unité X vers une position Y', cela aurait créé des arbres beaucoup trop grands et trop complexes mais surtout, dans les jeux de stratégie en temps réel, on va rarement déplacer une unité à la fois mais plutôt toute l'armée d'un seul coup. C'est pourquoi j'ai inclus cette fonctionnalité dans la fonction AttackLowestEnnemy. Pour chaque unité, si la distance euclidienne de la position de l'unité à la position de l'armée ennemie est supérieur à sa portée d'attaque alors elle n'attaque pas mais au lieu de ça elle avance en direction de l'armée ennemie. Cette position qui est plus proche de l'armée ennemie est calculée en prenant la position actuelle de l'unité et en ajoutant le vecteur unité de la distance euclidienne entre l'unité alliée et l'armée ennemie (cf figure 5)

```
for (const auto& unit : terran_units) {
    // récupération de la position de l'unité alliée
    sc2::Point2D unit_pos = unit->pos;
    // calcul du vecteur entre l'unité alliée et l'armée ennemie
    sc2::Point2D direction_from_ally_to_enemy = enemy_army_pos - unit_pos;
    // calcul de la norme de ce vecteur
    float norm = sqrt(pow(direction_from_ally_to_enemy.x, 2) + pow(direction_from_ally_to_enemy.y, 2));
    // si l'unitée alliée n'est pas à portée d'attaque alors
    // on calcule le point vers lequel elle doit avancer
    if (norm > 5.5) {
        sc2::Point2D target_point(
            unit_pos.x + ((2 / norm) * direction_from_ally_to_enemy.x),
            unit_pos.y + ((2 / norm) * direction_from_ally_to_enemy.y));
        action->UnitCommand(unit, ABILITY_ID::SMART, target_point);
    }
    else { // sinon elle peut attaquer sa cible
        action->UnitCommand(unit, ABILITY_ID::ATTACK, target_);
    }
}
```

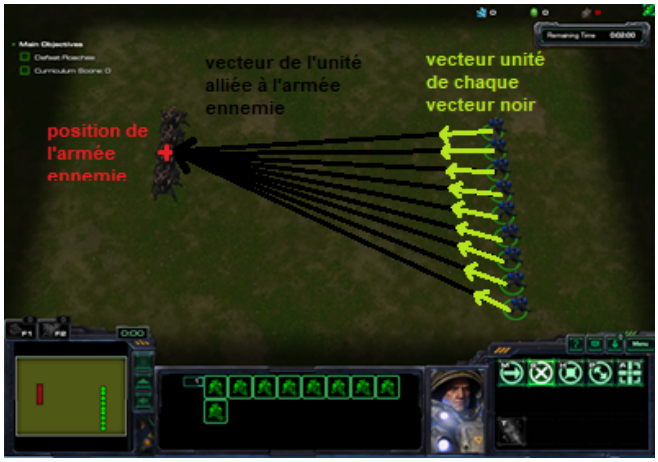

Figure 5: AttackLowestEnnemy action explaination

Concernant l'action MoveLowestAllyBack (qui fait donc reculer l'unité alliée qui a le moins de points de vie) j'utilise le même principe. D'abord je récupère cette unité puis je calcule le même vecteur mais au lieu de calculer la nouvelle position avec le vecteur unité dans la même direction je multiplie par -1 pour avoir la direction opposée. Il n'y a cependant pas que cela qui se passe lorsque cette action est effectuée. En effet comme c'est un jeu en temps réel, les autres unités qui ne sont pas déplacées ne font pas rien : si elles sont à portée d'attaque d'une unité ennemie (qui peut être différente pour chaque unité alliée) alors elles l'attaquent; si elles ne sont à portée d'attaque d'aucun ennemi cependant, alors elles ne font rien. La dernière action AttackClosestEnnemy fait exactement la dernière chose que j'ai expliquée pour l'action précédente mais sans avoir à déplacer une unité alliée avant. Dans certains scénarios il se peut qu'il n'y ai pas besoin de bouger une unité avant, au risque de perdre de la puissance de feu.

Les rollouts se font donc en exécutant une des actions précédentes au hasard jusqu'à ce qu'aucune des deux armées ne puisse attaquer l'autre ce qui veut dire que l'une des deux ne possède plus d'unités.

6 Un premier MCTS

J'ai avant tout voulu voir si j'étais capable de renvoyer des actions à faire assez rapidement pour l'action 'OnStep()' du bot qui demande des actions à faire très souvent. J'ai donc fait un MCTS full random très simple sans table de hachage. Sans lancer le bot et le jeu, les rollouts se font très vite (quelques milli-secondes), cependant, lorsque je lance le bot et un MCTS avec 10 rollouts (ce qui est peu) pour chaque action, à chaque appel de l'action 'OnStep()', celle-ci met 30 secondes à se terminer (cf. figure 6). Je me suis donc heurté à un problème technique : mon pc n'est soit pas assez puissant soit la gestion de mon MCTS et de la représentation d'une partie n'est pas assez optimisée. J'ai donc malheureusement du laisser tomber l'idée de faire ce projet sur le jeu en temps réel, mais il me restait la possibilité de le faire en tour par tour.

```
Launched SC2 (D:\StarCraft II\Versions\Base78285\SC2_x64.exe), PID: 5192
Waiting for connection
Connected to 127.0.0.1:8167
Waiting for the JoinGame response.
WaitJoinGame finished successfully.
Running mcts for player 1 : 10 rollout per action (3 actions)
Action : AttackLowestEnemy -- win_rate : 0.9
Action : MoveLowestAllyBack -- win_rate : 1
Action : AttackClosestEnemy -- win_rate : 1
OnStep() action over after : 30949.4 millisec.
```

Figure 6: 30s to run the whole 'OnStep' action

Il y a cependant des aspects qui diffèrent du jeu original. Je ne peux pas simuler les collisions entre unités donc il se peut que des unités se trouvent sur la même position, ce n'est pas très grave ici car il n'y a pas beaucoup d'unités mais en temps normal cela peut poser problème car toutes les unités pourraient être à portée d'attaque alors qu'en réalité ce n'est pas le cas. Autre chose qui advient à cause de la mécanique tour par tour est que l'armée qui reste immobile et qui attend que l'autre armée arrive sera énormément avantagée car elle a juste à attendre que l'autre armée arrive à porté d'attaque et c'est donc l'armée immobile qui attaque la première, ce qui est un avantage considérable. C'est le comportement que j'ai pu observer quelque fois avec ce MCTS full random, l'armée de 9 marines attendait que l'autre armée se déplace et pareil pour l'armée de 4 roachs. Avec ce MCTS full random, l'armée de 9 marines avait quasiment 100% de taux de victoire parce qu'à chaque tour le win rate calculé par action cumulait celui des états précédents, (au début c'est environs du 50/50) mais au fur et à mesure que la partie avance l'armée ennemie (celle des 4 roachs) ne fait que calculer des win rate de 0 pour chaque action et ne faisait donc que la même action à chaque fois ce qui lui coûtait en effet la victoire bien qu'il soit tout à fait possible pour elle de gagner. C'est pourquoi j'ai arrêté là le développement de ce MCTS et suis passé à l'approche UCT.

7 Approche UCT

L'avantage de cette approche est que les différents états du jeu sont sauvegardés et donc il est possible de s'adapter à chaque situation s'il y a un assez grand nombre de simulations à partir de cet état. Le plus important est donc de faire une bonne fonction de hashing pour représenter un état, et que ce hash soit unique pour chaque état. Dans le code de la libraire qu'Aron Granberg a codée pour sa thèse [4] j'ai pu trouver une fonction de hashing pour un état que j'ai reprise puis modifiée pour l'adapter à mon problème (cf. figure 7).

```
// returns the hash of the game state
uint64_t get_hash(std::vector<CombatUnit> units) {
    uint64_t hash = 0;
    for (auto unit : units) {
        hash = hash ^ (uint64_t)unit.owner;
        hash = hash ^ (uint64_t)unit.health;
        hash = hash ^ (uint64_t)unit.pos.x;
        hash = hash ^ (uint64_t)unit.pos.y;
    }
    return hash;
}
```

Figure 7: Hashing function

Il y a cependant quelque chose que je n'ai pas fait (car je n'ai pas compris pourquoi cela a été fait) qu'Aron Granberg a fait : c'est, avant de faire le XOR, multiplier le hash par 31. En tout cas, pour un vecteur d'unités donnés (ce vecteur contient chaque unité vivante de l'état actuel de la partie) le hash obtenu grâce à cette fonction est toujours le même. Lors des cours de cette année, ce que nous avions fait était d'attribuer un nombre aléatoire pour chaque état possible du jeu. Je ne pouvais pas faire cela car je me serai retrouvé à stocker beaucoup trop d'états : il y a 9 marines qui possèdent des points de vie entre 0 et 45, 4 roachs qui possèdent des points de vie entre 0 et 145 et chaque unité a une

position (x,y) avec x et y entiers compris entre 0 et 15. Donc le nombre d'états pour lesquels je devrais générer un nombre aléatoire est de (9x45) x (4x145) x (15x15) ce qui donne à peu près 53 millions d'états si les calculs sont bons, je ne peux pas me permettre de stocker une map aussi grande bien que l'accès y soit instantané. C'est pourquoi j'ai décidé d'utiliser cette fonction qui génère un hash depuis un état directement.

J'ai ensuite codé l'algorithme UCT et j'ai lancé une partie mais je me suis vite rendu compte d'un problème: certains états totalement différents génèrent le même hash (cf. figure 8 rectangles rouges). De plus, comme mentionné précédemment, je ne gère pas les collisions entre unités dans cette représentation du jeu, il est donc possible que plusieurs unités se retrouvent sur la même position ce qui va poser des problèmes pour le hash d'un état donné et donc c'est une cause potentielle du problème mentionné. D'autre part, certaines actions depuis un état ne sont jamais testées, aucun rollout n'est effectué pour ces actions (cf. figure 8 rectangles bleus) alors que si le hash d'un état n'a jamais été vu alors pour chaque action depuis cet état au moins un rollout devrait être effectué (c'est ce que j'ai codé); mais je ne comprend pas pourquoi certaines fois ce n'est pas le cas. Le résultat est que certains états ne sont jamais visités et il n'y a donc aucune statistique sur ces derniers.

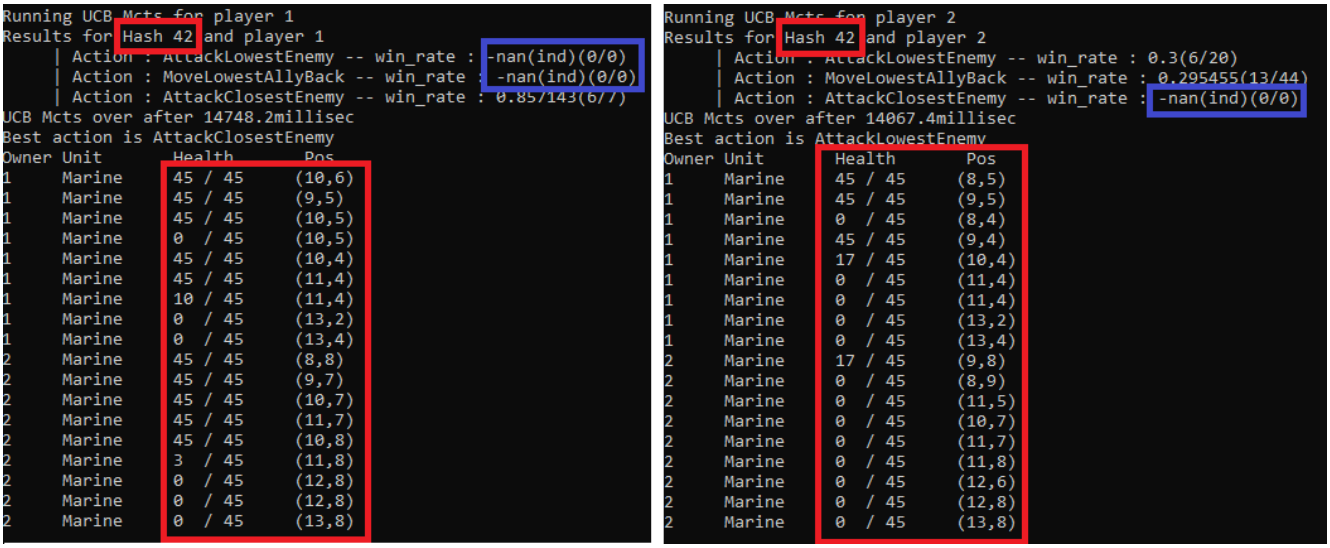

Figure 8: Same hash for different states

Sur la figure 8 c'est une partie où deux armées de chacune 9 marines s'affronte car j'ai trouvé plus intéressant cette situation que celle proposée au départ par le mini-jeu où l'on a 9 marines contre 4 roachs. Malgré les soucis au niveau de la fonction de hash et de l'algorithme UCT les parties sont quand même menées à leur terme et j'ai constaté que l'armée 1 (celle qui fait la première action) gagne 70% du temps contre l'armée 2 qui est aussi contrôlée par un autre UCT. J'ai pu aussi retrouver les comportements qui m'intéressaient à savoir, dans une situation où une unité possède peu de points de vie, la faire reculer pour qu'elle ne soit plus à portée d'attaque de ses ennemies et permette donc de sauver cette unité pour peut être la voir revenir plus tard dans la bataille. Cependant j'ai aussi pu observer que ces unités qui avaient été éloignées de la bataille ne revenaient quasiment jamais car leur seul moyen de revenir vers l'armée ennemie était via la fonction "AttackLowestEnnemy" qui faisait bouger l'armée alliée entière vers l'ennemi qui a le moins de points de vies; cependant si cet ennemi qui avait le moins de points de vie était lui aussi éloigné de la bataille c'est une très mauvaise idée de faire bouger l'armée alliée entière vers cet ennemi; c'est pourquoi j'ai pensé à coder une autre action qui ramènerait ces unités éloignées mais je n'ai pas eu le temps de le faire. Finalement, toujours concernant l'éloignement des alliés qui n'ont plus beaucoup de points de vie il y avait quelques scénarios où la fonction "MoveLowestAllyBack" n'était pas suffisante : par exemple une unité alliée à qui il reste 9/45 points de vie a déjà été reculée, mais une autre unité alliée qui est toujours à portée de l'ennemi est à 10/45 points de vie, il faudrait peut être la reculer pour la sauver; cette action "MoveLowestAllyBack" ne permet pas de reculer l'unité ayant 10/45 points de vie car ce n'est pas celle qui a le moins de vie. Il faudrait donc que je code une autre action qui permettrait d'éloigner les unités qui sont le plus en "danger" à la place.

8 Conclusions

Pour conclure, ce projet m'a permis de me rendre compte que développer une méthode MCTS pour un jeu en temps réel est bien moins facile que pour un jeu en tour par tour, cela demande une optimisation rigoureuse et un matériel performant pour obtenir des réponses du MCTS très rapidement. De plus la complexité du jeu demande à coder une fonction de hashing très précise afin de pouvoir utiliser à son plein potentiel l'algorithme UCT. Concernant ce dernier je pense ne pas avoir très bien implémenté l'algorithme ce qui m'a vallu ces petits problèmes au niveau des visites de certains états. J'ai pensé à adapter l'algorithme décrit dans le papier "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go" [5] mais par manque de temps je n'ai pas pu le faire. J'aurai aussi souhaité essayer de paralléliser le mcts de manière asynchrone pour essayer d'améliorer sa rapidité. D'autre part j'aurai voulu essayer d'implémenter une méthode RAVE ou GRAVE mais là aussi manque de temps. Finalement une méthode Nested MCTS n'était pas envisageable pour le jeu en temps réel car elle est connue pour prendre beaucoup de temps mais potentiellement faisable pour ma version en tour par tour. Mon objectif final aurait été une méthode NRPA car je pense que c'est celle qui aurait le mieux fonctionné.

9 Requirements et Tuto d'installation du projet

Avant tout il faut posséder le jeu StarCraft 2 qui est payant et l'avoir sous Windows si vous désirez lancer le bot sur une vraie partie. Sinon il est uniquement possible de lancer le code avec ma version tour par tour qui afficher des informations sur le terminal.

C'est la première fois que j'utilise Visual Studio pour coder en C++ et que j'utilise la fonctionnalité de Build ainsi que CMake je n'étais donc pas très sûr de ce que j'ai fait. Je n'ai donc pas créé de nouveau fichier à part "DefeatRoachesBot.cpp" et j'ai tout codé dans "bot_examples.h" et "bot_examples.cc" qui sont deux classes qui regrouppent toutes les fonctions pour les bots codés par Blizzard dans son api [2]. Cependant je ne peux pas les push sur mon github car le projet se clone via un "git clone –recursive", le projet contient donc des submodules dont le sc2-api qui contient lui même ces deux fichiers cpp. Tout ce que je peux faire c'est créer un dossier avec les différentes classes que j'ai modifiées et qu'il faudra copier/coller à la place de celles existantes après avoir cloné le répository.

Pour créer le projet il faut donc d'abord installer Visual Studio 2017 (très important, aucune autre version sinon cela ne fonctionne pas), et installer CMake. Il suffit ensuite d'exécuter les commandes suivantes dans un terminal:

- git clone –recursive https://github.com/LothairKizardjian/sc2-libvoxelbot
- cd sc2-libvoxelbot
- mkdir build
- cd build
- cmake ../ -G "Visual Studio 15 2017 Win64"
- start libvoxelbot.sln

Après avoir exécuté la dernière commande Visual Studio devrait s'ouvrir. Il suffit ensuite de copier la classe cpp "bot_examples.h" qui se trouve à la racine du projet dans le dossier "custom_classes" dans le dossier "Header Files" et la classe "bot_examples.cc" qui se trouve dans le même dossier que le précédent, qu'il faut copier dans "Source Files" comme sur la figure 9. Pour ce faire il suffit de faire un clic-droit $->$ copier du fichier "bot_examples.h" par exemple directement dans le dossier puis d'aller sur visual studio et faire un clic-doit $->$ coller sur le dossier "Header Files".

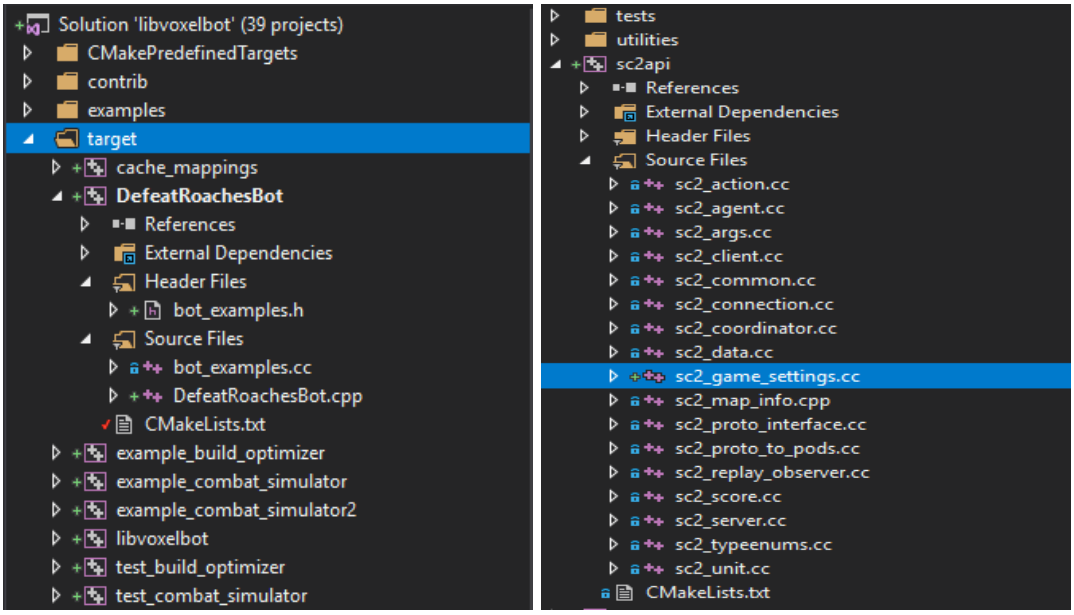

Figure 9: Copy/Paste bot_examples.h, bot_examples.cc (left) and sc2_game_settings.cc (right)

Il faut finalement remplacer un dernier fichier (cela ne concerne que les personnes ayant StarCraft 2 d'installé). Faites un clic-droit $->$ copier sur le fichier "sc2_game_settings.cc" puis dans visual studio clic-droit $->$ coller dans le sous-dossier "Source-Files" du dossier "sc2api" comme sur la figure 9. Il faudra d'abord supprimer le fichier "sc2_game_settings.cc" original qui s'y trouve. Une fois cela fait il ne reste plus qu'à télécharger la map "DefeatRoaches" à l'adresse suivante : `https://github.com/deepmind/pysc2` section "Get the maps" du Readme puis cliquez sur "mini games". Une fois le .zip téléchargé, ouvrez le dossier d'installation du jeu StarCraft 2, allez dans le dossier "Map" puis créez un dossier "Examples" et décompressez-y l'archive que vous venez de télécharger (il ne faut pas le sous-dossier "mini games", il faut que les maps soient directement dans le dossier "Examples").

Vous pouvez maintenant, sur Visual Studio, faire un clic-droit sur "DefeatRoachesBot" puis cliquer sur "Set as StartUp Project" (cf. figure 10)et ensuite appuyez sur "Ctrl + F5" ce qui buildera le projet sur Visual Studio et devrait ensuite lancer le terminal. Si vous souhaitez lancer le bot et le vrai jeu il faut, dans le fichier "DefeatRoachesBot.cpp" commenter la ligne "play_MCTS_game(10,"ucb",verbose);" et décommenter ce qu'il se trouve après.

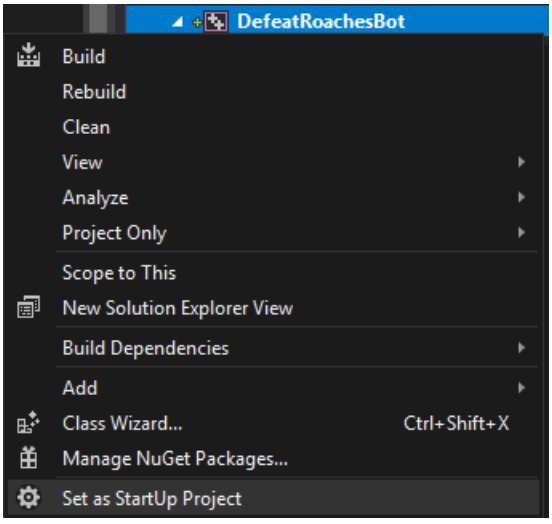

Figure 10: Set as startup project

References

[1] Vinyals, O., Babuschkin, I., Czarnecki, W.M. et al., *Grandmaster level in StarCraft II using multi-agent reinforcement learning.* Nature volume 575, pages 350–354, 2019.

[2] Blizzard, *Starcraft 2 API :* `https://github.com/Blizzard/s2client-api`

[3] ARON GRANBERG, *Monto Carlo Tree Search in Real Time Strategy Games with Applications to Starcraft 2.* STOCKHOLM, SWEDEN 2019

[4] ARON GRANBERG, *Bibliothèque "Libvoxelbot" contenant un simulateur de combat :* `https://github.com/HalfVoxel/sc2-libvoxelbot`

[5] Sylvain Gelly, David Silver, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go* `http://www.cs.utexas.edu/~pstone/Courses/394Rspring13/resources/mcrave.pdf`