Projet Programmation en Python
El Chapito's Game

LAURA MISRACHI
laura.misrachi@gmail.com

Ce rapport consiste en un résumé succinct du projet Python développé dans le cadre du cours de Programmation en Python du Master d'Ingénierie Mathématique (option MPE) de l'UPMC. Le but du projet est de développer une stratégie gagnante pour deux joueurs adversaires, un gardien et El Chapito. Par soucis de concision, les règles du jeu, présentes dans l'énoncée, ne sont pas rappelée ici. Lors de la réalisation de ce projet, j'ai choisi d'optimiser les déplacements des deux joueurs adversaires et non d'un seul. Entre autre, cela m'a permis de les faire jouer l'un contre l'autre et d'améliorer leurs stratégies gagnantes respectives sur base des actions de leurs adversaires.

Contents

I. Stockage des données

Afin de mettre en place notre jeu, il est nécessaire de stocker les données des différents graphes, qui seront nos supports de jeu. Cela a pu être fait grâce à la définition d'une classe ***Graph***, qui est codée et amplement expliquée dans le fichier ***Vertex.py***. Les attributs de la classe ***Graph*** sont les suivants:

- *vertices* : dictionnaire contenant les noeuds et leurs coordonnées de R2. Il est de la forme:

$$\{noeud : [coord_x; coord_y]\}$$

- *node_adjacent* : dictionnaire contenant les noeuds et leur liste de voisins, d'adjacence. Il est de la forme:

$$\{noeud : [liste\_de\_voisins]\}$$

- *visited_by_guardian* : dictionnaire contenant les noeuds déjà visités par le gardien et indiquant par quelles portes de sortie ils ont été visités. Il est de la forme:

$$\{noeud : porte\_de\_sortie\}$$

- *weights* : dictionnaire contenant les arêtes et indiquant leurs poids. Ces poids sont utilisés pour développer la stratégie de déplacement optimale d'El Chapito et seront abordés avec plus de précision dans la suite de ce rapport. Cet attribut est de la forme:

$$\{(noeud1, noeud2) : poids\}$$

- *nb_of_outdoors_adj* : dictionnaire indiquant, pour chaque noeud, le nombre de porte de sorties qui lui sont adjacentes. Cet attribut prend la forme suivante:

$$\{noeud : nb\_porte\_sorties\_adjacentes\}$$

- *nb_closed_edges*: entier indiquant le nombre d'arête déjà fermées par le gardien dans le jeu.

Par ailleurs différentes méthodes ont été définies au sein de cette classe:

- ***add_edge*** : afin d'ajouter une arête au graphe, notamment à la lecture du fichier.
- ***delete_edge*** : afin de supprimer une arête du graphe, notamment lorsqu'une porte est fermée par le gardien.
- ***node_get_adj*** : afin d'obtenir la liste de noeuds adjacents à un certain noeud.
- ***node_visited_guardian*** : afin d'indiquer si un noeud a été visité par le gardien et depuis quelle porte de sortie.
- ***increment_closed_edges*** : afin d'incrémenter le nombre de portes fermées par le gardien.
- ***update_weight*** : afin de mettre à jour les poids d'une arête (noeud1, noeud2).

II. Algorithme du plus court chemin : Dijkstra

Comme on peut déjà s'en douter, la notion de plus court chemin jouera un rôle primordial dans ce jeu et pour cette raison, nous avons codé l'algorithme de ***Dijkstra*** au sein du fichier *main_file.py*. Deux versions, l'une avec poids unitaire "Dijkstra" et une autre version "Dijkstra2" avec poids fixés par l'attribut *weights* ont été implémentées toutes deux.

III. Stratégie gagnante du Gardien

Dans cette section, nous allons résumer succinctement la stratégie gagnante du gardien mise en place.

- En priorité, le gardien ferme une arête si El Chapito est sur le point d'accéder à une porte de sortie au prochain tour.
- On définit certains noeuds dits *critiques*, qui sont ceux qui possèdent au moins deux portes de sorties dans leurs adjacents directs. En deuxième priorité, le gardien va s'attarder à supprimer les noeuds les plus critiques parmi ces noeuds déjà critiques, que l'on va surnommer noeuds supercritique. Ces noeuds supercritiques sont déterminés par la fonction *guardian_critical_node_to_choose*. Il s'agit des noeuds critiques pour lesquels, partant de la position actuelle d'El Chapito, ce dernier peut les atteindre en forçant le gardien à fermer une porte à chaque itération. Concrètement, lorsqu'El Chapito est à une distance unitaire d'une telle séquence le menant à un noeud supercritique, il est assuré de gagner. Lorsque plusieurs noeud supercritiques existent, on choisi de supprimer l'arête le contenant qui se trouve le plus proche de la position actuelle d'El Chapito.
- Lorsque tous les noeuds supercritiques ont été passés en revue, on supprime plus simplement les arête liées à des noeuds critiques, toujours en privilégiant ceux qui se trouvent le plus proche d'El Chapito.
- Enfin, lorsque tous les noeuds critiques ont été passés en revue, on supprime simplement l'arête qui se trouve la plus proche de la position actuelle d'El Chapito, en utilisant Dijkstra.

Je tiens à faire remarquer que ma fonction *one_move_guardian* qui indique le déplacement optimale du gardien pour une configuration de jeu bien précise, n'est pas assez optimisée du point de vue de l'occupation de la mémoire. Cela ne pose pas de problème ici puisque nos graphes sont de tailles relativement faible, mais avec plus de temps, il faudrait bien entendu nettoyer cette partie du code davantage. Notamment, il faut faire passer toute la partie prioritaire sur la recherche des points critiques avant la recherche plus classique des portes proches d'El Chapito avec l'algorithme Dijkstra afin de de ne pas faire cette dernière étape inutilement à chaque appel de la fonction lorsqu'il existe des points critiques.

IV. Stratégie gagnante d'El Chapito

Dans cette section, nous allons présenter la stratégie gagnante d'El Chapito.

- Afin de contrer la stratégie du gardien, en priorité, El Chapito se déplace vers les noeuds critiques énoncés précèdemment, qui sont le plus proche de lui.
- Lorsque tous les noeuds critiques ont été supprimés par le gardien, El Chapito cherche simplement a se déplacer vers les noeuds les plus proches de lui en terme de poids, à l'aide de l'algorithme Dijkstra2. Il nous faut donc désormais préciser comment ont été mis en place les poids des différentes arêtes, lors de l'appel de la fonction *update_weight_node_adj_outdoors*.

Les arêtes classiques (c'est-à-dire celles qui contiennent des noeuds qui ne sont adjacents à aucune porte de sortie) ont un poids fixé à 5. Les arêtes qui contiennent des noeuds qui sont adjacents à deux portes de sorties au moins ont un poids fixés à 1. On souhaite effectivement les rendre très attractives pour El Chapito. Les arêtes qui contiennent des noeuds qui sont adjacents à une seule porte de sortie, qui de surcroît termine le graphe (pas d'autre déplacement possible) ont un poids fixé à 10. Effectivement, on veut à tout prix empêcher El Chapito de s'y rendre, puisque le gardien pourra systématiquement refermer la porte avant qu'El Chapito n'y parvienne. Et de surcroît, il reste plus intéressant pour El Chapito de se déplacer ailleurs qu'à ses endroits là de façon absolue. On affecte un poids de 2 aux arêtes qui contiennent des noeuds adjacents à des portes de sorties qui ne sont pas terminales (i.e. qui possèdent au moins deux voisins). Ces arrêtes sont intéressantes pour El Chapito, mais elles le sont moins que des noeuds critiques.

V. Tests unitaires

Tous les tests unitaires fournis ont été testés avec notre algorithme. Les résultats obtenus sont très similaires à ceux des tests, à savoir que le gagnant est systématiquement celui indiqué par le cas test et que le déroulé de la partie est très similaire. Cela peut être vérifier en lançant les différents cas tests à l'aide de notre fichier ***main_file.py***.