

Algorithmique appliquée

Modélisation du problème RoboCup

Pierre LORSON
Carlos VILLAVICENCIO
21 décembre 2018

1 DESCRIPTION DU PROBLÈME ET OBJECTIF

Dans la compétition RoboCup, plusieurs robots à roues se déplacent rapidement et disputent un match de football robotique. Le but est de traiter le problème de la défense. Nous devons placer le moins de robots de défense possible de manière à éviter les tirs des robots attaquants de façon optimale. Il faut couvrir tous les angles de tir possibles, avec un pas de discrétisation angulaire pour les angles du tir et un pas de discrétisation angulaire pour les positions des défenseurs.

2 VARIABLES DE DÉCISION

- R : Ensemble de robots
- A : Ensemble d'attaquants
- D : Ensemble de défenseurs
- px, py : Position des attaquants
- h, w : Taille du terrain
- ra : Rayon des robots
- tb : Taille du but
- θ_{step} : Pas de tir

3 DÉFINITION DU PROBLÈME

- $\forall a \in A, \forall k \in \mathbb{N}, \exists d \in D, interception(a, k * \theta step) \vee nonCadre(a, k * \theta step)$
- $edge((a(dx), a(dy), \theta), (d(px), d(py))) \iff interception(a, \theta, d)$

La définition mathématique du problème peut être liée aux problèmes de graphes $G = (S, A)$, en particulier au problème d'ensemble dominant. Nous choisissons donc la modélisation à l'aide de graphes, ce problème étant parfaitement adapté à la situation. Le problème d'ensemble dominant est de déterminer un ensemble de k sommets S tel que tout sommet u n'appartenant pas à S a au moins un voisin dans S . Dans ce cas particulier, nous savons que k ne peut pas excéder 6, il n'y a en effet que 6 robots dans une équipe. Ce type du problème est NP-complet. On doit chercher l'ensemble dominant avec la plus petite quantité des sommets.

REMARQUES IMPORTANTES

- Il n'est pas nécessaire d'avoir des arêtes entre les défenseurs.
- Il n'est pas nécessaire de prendre en compte les tirs non cadrés.

Il faut pour cela vérifier que le défenseur est placé selon un pas de tir T entre l'attaquant et le but. L'idée est de modéliser la situation par un graphe. On peut penser qu'un graphe dominant de défenseur est suffisant, mais il faut prendre en compte le pas de tir.

Un algorithme pourrait être :

- 1) Positionnement de base des joueurs.
- 2) Vérification de la présence d'un défenseur suivant le pas de tir d'un attaquant entre lui et le but.
- 3) Si la condition est remplie, on ne bouge pas, sinon, on utilise le poids des arêtes, représentant la distance entre le point le plus proche sur la trajectoire et tous les défenseurs disponibles, pour bouger un défenseur.

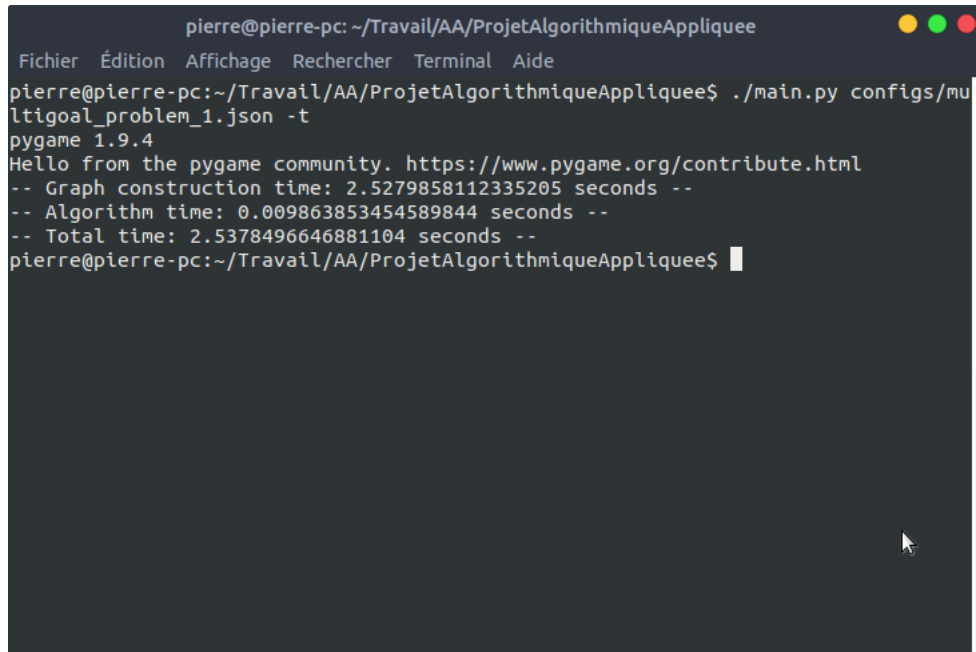
4 TRAVAIL EFFECTUÉ

4.1 UTILISATION ET MODES DU PROGRAMME

Avant de rentrer en détail dans l'implémentation, il convient de préciser le fonctionnement du programme et les protocoles de test. Le programme nécessite l'utilisation de trois paramètres, le premier étant le fichier JSON contenant les coordonnées de position des attaquants, le deuxième le mode choisi et le troisième l'algorithme de résolution choisi. Si les paramètres ne sont pas spécifiques, les paramètres par défaut sont choisis.

Nous avons donc deux modes dans notre application. Le premier, dit "time", se choisit avec le paramètre "-t". Il permet d'exécuter la résolution du problème sans visualisation et affiche ensuite la durée de la résolution via trois valeurs : la première renseigne le temps de création du graphe, la deuxième le temps de résolution du problème. et enfin, le total s'affiche. Une capture d'écran l'illustre ci-dessous.

Le deuxième mode s'exécute avec l'option "-g" : la visualisation se lance et permet de voir facilement la résolution du problème, de façon graphique. C'est l'option par défaut. On choisit ensuite en dernier argument l'algorithme voulu lors de la résolution, en écrivant "brute" dans le cas de l'algorithme force brute ou "greedy" si l'on veut l'algorithme greedy. Si rien n'est spécifié, greedy est choisi par défaut.

A screenshot of a terminal window with a dark background. The title bar at the top reads "pierre@pierre-pc: ~/Travail/AA/ProjetAlgorithmiqueAppliquee" and includes three window control buttons (yellow, green, red). The menu bar shows "Fichier", "Édition", "Affichage", "Rechercher", "Terminal", and "Aide". The terminal content shows the command `./main.py configs/multigoal_problem_1.json -t` being executed. The output includes the pygame version (1.9.4), a URL for the pygame community, and timing information: "Graph construction time: 2.5279858112335205 seconds", "Algorithm time: 0.009863853454589844 seconds", and "Total time: 2.5378496646881104 seconds". The prompt `pierre@pierre-pc:~/Travail/AA/ProjetAlgorithmiqueAppliquee$` is visible at the bottom.

```
pierre@pierre-pc: ~/Travail/AA/ProjetAlgorithmiqueAppliquee
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
pierre@pierre-pc:~/Travail/AA/ProjetAlgorithmiqueAppliquee$ ./main.py configs/multigoal_problem_1.json -t
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
-- Graph construction time: 2.5279858112335205 seconds --
-- Algorithm time: 0.009863853454589844 seconds --
-- Total time: 2.5378496646881104 seconds --
pierre@pierre-pc:~/Travail/AA/ProjetAlgorithmiqueAppliquee$
```

4.2 PARSER JSON ET VISUALISEUR

Avec autorisation, nous avons repris le programme Python déjà existant de M. Hofer, permettant de visualiser un problème et sa solution encodés sous forme de deux fichiers JSON. Il nous a simplement fallu ajouter l'écriture de fichier JSON automatisée contenant la solution au problème donné. Pour cela, on se sert simplement de la bibliothèque json officielle de Python 3.

4.3 GRAPHE

Pour représenter notre problème, nous utilisons la théorie des graphes. La première partie de notre travail consiste à générer un graphe basé sur les informations obtenues à partir du problème stocké au format JSON. Une fois que nous avons lu le fichier du problème, nous avons besoin des données suivantes : les adversaires (robots attaquants), le rayon des robots, le `theta_step` et le `pos_step`.

Un graphe simple (et pas nécessairement connecté) est créé à l'aide de listes d'adjacence. Les définitions des fonctions sont visibles dans la classe `Graph` du code développé. Pour chaque nœud, nous avons une liste d'adjacence qui indique les arêtes (connexions) avec un autre nœud. Nos nœuds sont définis dans la classe `Vertex`, où nous trouvons également une propriété « `type` » qui indique le type de robot (Robot attaquant ou Robot en défense). Ils tiennent également compte des propriétés du rayon et de la direction de tir des robots.

Notre procédure de construction de graphique comprend 3 étapes :

- Génération des nœuds représentant les adversaires (Robots attaquants)
- Génération des nœuds représentant la défense (Robots de défense)
- Génération des arêtes

4.3.1 GÉNÉRATION DE NŒUDS DES ADVERSAIRES

Avec les données extraites des fichiers contenant le problème et à l'aide de la fonction `createAttackers()`, `theta_step` est analysé pour vérifier les plans possibles pouvant être obtenus par le robot attaquant. Pour chaque tir qui entre dans le but, un nœud est généré pour un robot attaquant.

4.3.2 GÉNÉRATION DE NŒUDS DE DÉFENSE

L'approche utilisée pour nos robots de défense utilise les attaquants de la propriété `pos_step`. Dans notre zone de jeu, chaque `pos_step` est généré par un nœud pour un robot défenseur utilisant la fonction `createDefense()`.

4.3.3 GÉNÉRATION DES ARÊTES

La fonction `createEdges()` vérifie si les robots de la défense peuvent arrêter les tirs des robots attaquants. Si les tirs peuvent être arrêtés, une arête est créée entre le robot attaquant et le défenseur. On vérifie également s'il existe une collision entre le nœud attaquant et le nœud défenseur à l'aide du rayon et de la position. En cas de collision entre le nœud attaquant et le nœud de défense, aucune arête est créée entre les nœuds, car les deux robots ne peuvent pas être superposés.

4.4 RÉOLUTION DU PROBLÈME

En brève introduction à cette partie, signalons que toutes les solutions présentées ci-dessous sont compatibles pour chaque variation de pas et de largeur de robot. De même, les robots ne peuvent pas se rentrer dedans.

Nos algorithmes cherchent des moyens de couvrir tous les angles de tirs valides en utilisant le moins de robots possible. En bref, nous travaillons à la recherche d'un ensemble dominant minimal. Des méthodologies récursives telles que l'algorithme de force brute et des méthodes heuristiques telles qu'un algorithme de type greedy ont été utilisées. Il est important de noter que quand un tir ne peut pas être intercepté, les deux algorithmes ne donneront pas de solution (les défenseurs ne seront pas affichés dans les graphiques).

4.4.1 RÉOLUTION EN BRUTE-FORCE

Premièrement, un algorithme de force brute a été implémenté. Comme son nom l'indique, cet algorithme va simplement tester chaque solution pour les défenseurs jusqu'à tomber sur une solution minimale valide. Comme on peut l'imaginer, un tel algorithme n'est pas optimal en temps, mais il a le mérite d'être simple à implémenter et à imaginer. On peut trouver son implémentation dans la classe "BruteForce". Cette dernière est composée d'une liste récupérée de la classe Graph à l'aide des getters de cette dernière, à savoir attackers. L'exécution en elle-même se passe dans la fonction execute(), où l'on déroule l'algorithme comme expliqué dans le cours. On a une liste de solutions possibles et chaque fois qu'une nouvelle solution est générée, on vérifie si elle est dans la liste. Si elle l'est, la recherche dans cette section est omise car il est bien entendu inutile de chercher des solutions que nous avons déjà dans la liste. On cherche ensuite une solution minimale via la fonction searchMinimalSolution(self).

4.4.2 RÉOLUTION AVEC UN ALGORITHME DE TYPE GREEDY

Nous avons utilisé un algorithme de type greedy pour résoudre notre problème le plus rapidement possible. Le temps de résolution est de l'ordre de moins de 6 millisecondes pour un problème tel que le basic_problem_1.json, où on a 3 attaquants, 3 défenseurs et une cage de but. L'implémentation est faite via la classe "Greedy", qui est composée de 3 listes : deux listes récupérées de la classe Graph à l'aide des getters de cette dernière, à savoir attackersList et defenseList, puis une liste "result" où l'on stocke les sommets retenus par l'algorithme : ce sont nos résultats finaux.

L'exécution en elle-même se passe dans la fonction execute(), où l'on déroule l'algorithme comme expliqué dans le cours.

4.5 OPTIMISATION GÉNÉRALES

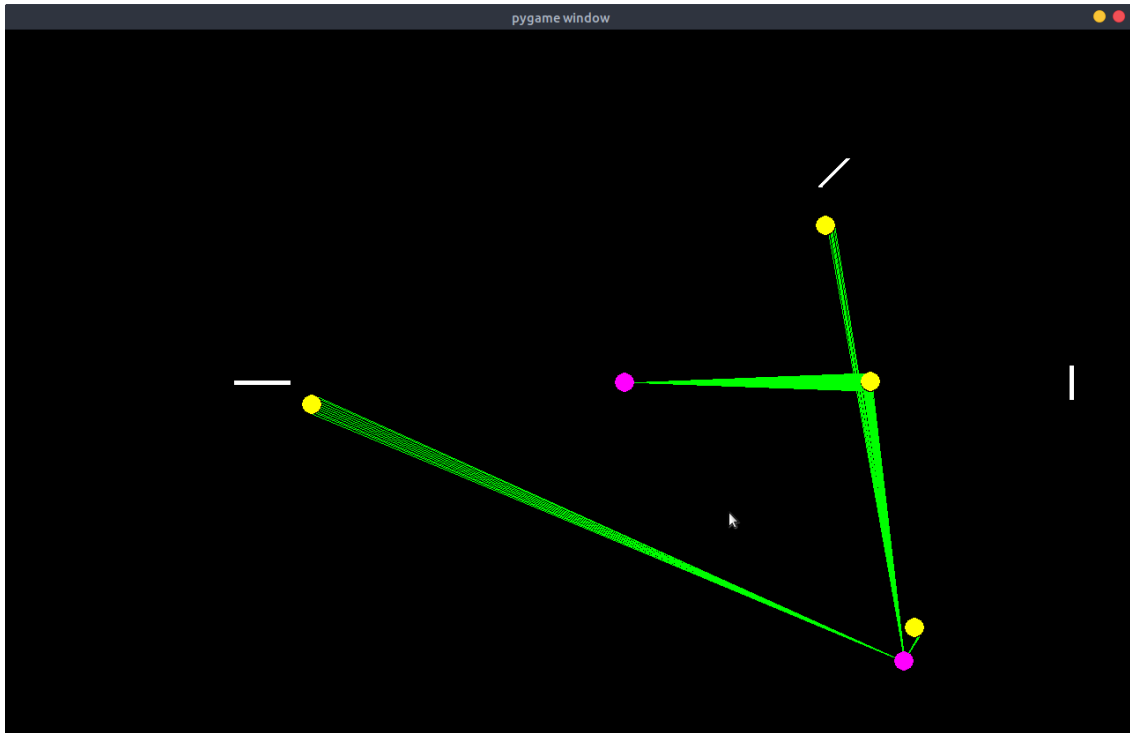
Certaines optimisations peuvent être appliquées à nos deux algorithmes. Par exemple, nous avons choisi d'arrêter la recherche si on s'aperçoit qu'un attaquant ne va pas avoir d'arêtes avec un défenseur, et que le problème devient insolvable.

De même, il est inutile (et en plus, incorrect si choisi comme solution) de calculer les positions où les défenseurs empiéteraient sur les attaquants. Cela permet un gain de temps non négligeable, spécialement sur l'algorithme de force brute qui va essayer chaque position possible pour déterminer une solution valide.

4.6 EXTENSIONS

4.6.1 MULTIGOAL

Les solutions implementées sont compatibles avec l'extension multigoal, où l'on souhaite modéliser un terrain dans lequel il y a plusieurs cages de but. On peut observer une perte de performances négligeable, malgré la montée en difficulté du problème. Une capture d'écran d'un exemple de résolution peut être trouvé ci-dessous.



4.7 INTERPRÉTATION DES RÉSULTATS ET CONCLUSION

L'ajout du mode "time" nous a permis de prendre du recul sur le temps d'exécution du programme et de bien comprendre quel était l'étape la plus coûteuse en temps. Pour les cas simples auxquels nous sommes confrontés, la construction du graphe semble la plus coûteuse, peu importe que l'on se place en multigoal ou en problème simple, mais cette intuition n'est pas suffisante.

Afin d'affiner notre interprétation, nous avons procédé à une expérience simple mais très évocatrice : nous avons pris comme base le problème "basic_problem_1" et avons tenté de le résoudre d'abord dans sa forme originale, puis avec un pas divisé par 10, passant donc de 0.2 à 0.02. Les résultats sont les suivants :

```
pierre@pierre-pc:~/Travail/AA/ProjetAlgorithmiqueAppliquee$ python3 main.py configs/basic_problem_1.json -t
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
-- Graph construction time: 0.6101734638214111 seconds --
-- Algorithm time: 0.0038521289825439453 seconds --
-- Total time: 0.6140255928039551 seconds --
pierre@pierre-pc:~/Travail/AA/ProjetAlgorithmiqueAppliquee$ python3 main.py configs/basic_problem_1.json -t
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
-- Graph construction time: 58.328404664993286 seconds --
-- Algorithm time: 25.532336235046387 seconds --
-- Total time: 83.86074090003967 seconds --
```

Premièrement, on peut observer que la construction du graphe se fait efficacement, en environ 0.62 secondes. L'algorithme greedy est quant à lui extrêmement rapide ici, avec une résolution de presque 4 millisecondes. Ces résultats sont satisfaisants, et nous montons donc en charge, en diminuant le pas par un facteur 10, ce qui fait totalement exploser le nombre de positions possibles. On passe à 58 secondes de création de graphe et à 25 secondes de résolution, ce qui donne un facteur de l'ordre de 95 pour le graphe et de... 6250 pour la résolution. Ces résultats semblent cohérents : la complexité de la tâche pour l'algorithme explose et au final, le temps total est multiplié par 136. On a donc vérifié que la création de graphe est la plus coûteuse en temps lors de cas très simples, mais au final dérisoire lors d'un cas plus complexe.

En conclusion, ce projet nous a montré qu'il est essentiel de bien comprendre et modéliser un problème avant d'envisager sa résolution : non seulement le gain en temps dans la totalité du projet est considérable puisqu'elle permet de bien cibler les tâches, implementations à faire et les algorithmes ou heuristiques à choisir, mais elle évite de faire des erreurs coûteuses : dans notre cas par exemple, il aurait été bête de considérer les tirs non cadrés, et ne pas penser à les négliger est une erreur facile à faire si l'on ne pose pas le problème correctement.