Algorithmique appliquée
Small-size league RoboCup

Auteurs :
ANEAS Alan
CATALDI Alexis
KERROUM Abderrahmane

Master - Année 2
Parcours Génie Logiciel

21 décembre 2020

Professeur en amphithéâtre :
GAVOILLE Cyril

Chargé de TD :
HOFER Ludovic

Sommaire

1 Définition informelle du problème

Soit un match de RoboCup, nous jouerons les défenseurs et les adversaires seront les attaquants. Supposons que chaque adversaire soit capable de tirer dans le but, notre objectif est de trouver où placer nos défenseurs afin d'arrêter le plus de tirs menant au but possible, en minimisant le nombre de défenseurs ou leur distance à parcourir (extension défenseurs initiaux).

2 Définition formelle du problème

2.1 Langage

— terrain :
  1. Le terrain est l'aire dans laquelle les joueurs et les buts pourront être positionnés.
  2. noté F, avec F $\subset \mathbb{R}^2$

— pos_{step} : métrique des points où placer les robots
  1. Pour simplifier la modélisation, nous définissons une distance entre les positions possibles pour placer les défenseurs.
  2. notée $pos_{step} \in \mathbb{R}$

— grille :
  1. Pour simplifier la modélisation, les défenseurs ne pourront être positionnés que sur une grille de valeurs discrètes dont les coordonnées seront espacées de multiples de pos_{step}.
  2. notée G, dont les croisements (x,y) sont définis par {(x,y) $\in$ F, x mod pos_{step} = 0, y mod pos_{step} = 0} G $\subset$ F

— grille (version extension) : positions défenseurs possibles
  1. Une extension demande à éviter les collisions entre les défenseurs, afin d'y répondre plus simplement, supposons une sub-grille issue de la première dont les croisements sont espacés afin que deux robots posés sur ces points ne puissent pas être en collision
  2. notée G' = {g $\in$ G, $\forall_{o \in O}$, ¬collision(g, o)} (la propriété collision() est expliquée dans la section Relations du document)

— attaquants / adversaires :
  1. Les attaquants sont les joueurs adverses, chacun sera capable de tirer dans les buts.
  2. Chaque attaquant est modélisé par un point correspondant à sa position (x,y) sur le terrain F.

— défenseurs :
  1. Les défenseurs seront les joueurs de notre équipe, leur objectif est de se positionner sur la trajectoire des tirs adverses menant aux buts.
  2. Chaque défenseur est modélisé par un point correspondant à sa position (x,y) sur la grille G'

— joueurs : les joueurs sont les attaquants et les défenseurs.

— θ_{step} : angles des tirs considérés
  1. Pour simplifier la modélisation, supposons que les attaquants ne puissent tirer selon un nombre limité de directions (représentables par des demi-droites partant du point de l'attaquant et suivant la direction de l'angle sur un cercle trigonométrique) espacées d'un angle d'une taille que nous nommerons θ_{step}.
  2. $\theta_{step} \in \mathbb{R}$

— point : (x,y) $\in$ F

— R_{radius} : rayon du cercle correspondant à la surface couverte par un défenseur (*hitbox*)
  1. Une extension demande d'éviter les collisions, pour cela, il faut prendre en compte une zone autour de chaque joueur dans laquelle ne devrait se trouver aucune autre zone de surface prise par un autre joueur, afin d'éviter que les joueurs n'entrent en collision.
  2. $R_{radius} \in \mathbb{R}$

2.2 Relations

— collision :

1. La collision consiste en savoir si deux robots seraient trop proches pour se placer sur des points donnés
2. collision $F^2 \rightarrow$ {VRAI, FAUX}

$$collision(p1, p2) = \begin{cases} VRAI : ||p1 - p2||_2 \leq 2 * R_{radius}, \\ FAUX : sinon \end{cases}$$

— tir cadré :
  — Un tir est cadré quand sa trajectoire intersecte le segment de but, du côté ouvert du but. Sa valeur sera le point d'intersection entre le but et le tir s'il y en a un, None sinon.
— tir intercepté (Intersection entre le tir et le cercle (position d + R_{radius}) représentant un joueur)
— tir courbé (extension, calculer aussi la trajectoire courbe depuis un attaquant)

2.3 *Input*

2.3.1 Problème initial

— position des adversaires : O (ensemble de points x,y)
— position des cages de but (cible) : T (modélisé par deux points et une direction)
— taille robots : $R_{radius} \in \mathbb{R}^+$
— terrain : F $\subset \mathbb{R}^2$ (un rectangle défini par deux points)
— pos_{step} (distance entre les points de la grille G)
— θ_{step} (angle entre deux tirs possibles depuis la même position)

2.3.2 Extension : Plusieurs buts

Cette extension a pour effet de modifier l'entrée en modifiant l'élément T, celui-ci n'est pas seulement un segment avec l'orientation des cages de but mais un ensemble de segments et leur orientation, cela demandera de placer les défenseurs dans les trajectoires menant à des buts dans toutes les cages possibles.

2.3.3 Extension : Gardien

Cette extension a pour effet de considérer un joueur comme étant un gardien, lui octroyant une zone dans laquelle il est le seul défenseur à pouvoir se trouver, que nous appellerons G_a (goalkeeper_area), $G_a \subset$ F. Les défenseurs non-gardiens n'ont pas le droit d'aller dans cette zone.

2.3.4 Extension : Position initiale des défenseurs

Cette extension a pour effet d'ajouter en entrée l'ensemble des positions des défenseurs, D, celle-ci est représentée par un ensemble de couples (x,y) de G'. L'objectif est de demander aux robots les plus proches de se déplacer aux points à défendre, afin de minimiser les trajets et que les défenseurs se trouvent au bon endroit lors du tir adverse pour intercepter la balle.

2.3.5 Extension : Distance minimale entre joueurs

Cette extension a pour effet d'ajouter à la surface au sol occupée par un robot une distance supplémentaire afin d'éviter les collisions, et éventuellement les chutes. Pour ce faire, elle ajoute en entrée une nouvelle valeur M (minDist), M $\in \mathbb{R}$ et nécessite la modification de la relation collision afin qu'elle prenne en compte cette distance minimale entre deux robots.

2.3.6 Extension : Vitesse maximale

Cette extension a pour effet de considérer la possibilité pour un défenseur de récupérer la balle si elle est tirée par un attaquant avant qu'elle n'atteigne les cages de but. Pour ce faire, elle ajoute en entrée deux nouvelles valeurs B_s (ball_max_speed) et R_s (robot_max_speed), $B_s \in \mathbb{R}$, $r_s \in \mathbb{R}$ et nécessite la modification de la relation de tir intercepté pour considérer un tir interceptable.

2.4 *Output*

— Positions défenseurs alliés : D = $(d_1, d_2, d_3 \ldots)$ (ensemble de points) (ordre important car extension position initiale des défenseurs : pas de téléportation) $d_i \in G'$

2.5 Contraintes sur l'*output*

— $|D| \leq nb_{defenseurs}$, avec |D| étant le plus petit possible tout en interceptant tous les tirs cadrés
— $\forall(i,j) \in [1,|D|]^2$, i ≠ j, ¬collision$(d_i, d_j)$
— Les positions des défenseurs ne devront pas sortir du terrain

3 Modélisation du problème

3.1 Modèle de graphe non-orienté

3.1.1 Nœuds

Dans ce modèle, ils y aurait deux types de nœuds, représentant :
— les tirs cadrés possibles de l'équipe ennemie (différentiables grâce à des triplets de valeur (x, y, angle θ)).
— les positions possibles des défenseurs (différentiables par un couple de valeur (x, y)) interceptant au moins 1 tir cadré.

3.1.2 Arêtes

Les arêtes relieraient les tirs cadrés aux positions de défenseurs se trouvant dans la trajectoire du tir cadré. Ces arêtes seraient codées par la présence d'un tir dans une liste de tirs arrêtés par défenseurs et inversement la présence d'un défenseur dans une liste d'intercepteurs d'un tir.

4 Pistes de résolution du problème

4.1 Solution

Une solution serait de générer un graphe depuis les données en entrées puis de lancer un algorithme sur ce graphe afin de renvoyer un ensemble de points répondant aux contraintes du problème initial et des extensions sélectionnées. Cet algorithme sélectionnerait les points défenseurs nécessaires afin d'arrêter les tirs ennemis. L'ensemble de points obtenus, mis en relation avec les tirs ennemis pourrait être interprété comme un graphe de solution.

Les algorithmes que nous avons implantés sont détaillés lors de la section 6 Algorithmes implémentés

4.2 Modèle de graphe non-orienté

Dans le modèle graphe non-orienté, l'on peut définir le fait d'avoir obtenu un graphe de solution si le graphe contient les éléments suivants :
— Le nombre de nœuds défenseur est inférieur ou égal au nombre de défenseurs sur le terrain.
— L'ensemble des nœuds défenseurs dominent les nœuds tir.
— Aucune arête de collision entre les nœuds.

Une fois le graphe solution trouvé, il suffit ensuite de récupérer les couples (x, y) de chaque nœud défenseur et de les transmettre au défenseur correspondant.

5 Compatibilité de la modélisation choisie avec les extensions

5.1 Modèle graphe non-orienté

5.1.1 Extension : Plusieurs buts

Afin d'implémenter cette extension, il suffit de prendre en compte la présence de plusieurs buts lors de la génération de l'algorithme, c'est-à-dire que lorsque que l'on ajoute un nœud tir cadré

l'on définisse un tir cadré comme étant un tir marquant dans un des buts et non pas dans le seul but.

5.1.2 Extension : Gardien

Une implémentation possible pour cette extension serait d'avoir un nouveau type de nœud pour le gardien, ce nouveau type de nœud correspondant à chacune des positions possibles pour le gardien. De plus lors de la récupération et l'envoi des informations au robot, il faudrait que le robot gardien soit différentiable des autres. Aussi, les défenseurs non-gardiens, n'ayant pas le droit d'entrer dans la zone du gardien, les nœuds positions défenseurs seront donc restreints aux positions sur le terrain ne faisant pas partie de la zone du gardien.

Nous avons simplifié ce problème en considérant qu'il ne faut qu'un défenseur dans la zone de gardien de but dans l'ensemble de positions sélectionnées en sortie pour le placement des défenseurs. Critique : cette modélisation n'empêcherait pas qu'un défenseur non-gardien entre dans la zone de gardien pendant son déplacement, ou, si le gardien sort de sa zone, que le défenseur non-gardien finisse son déplacement dans la zone.

5.1.3 Extension : Position initiale des défenseurs

Une implémentation possible pour cette extension d'associer à chaque nœud de défenseur, ses distances avec les positions initiales, puis faire un calcul pour savoir, selon les nœuds nécessaires pour arrêter tous les tirs, à quel nœud associer chaque position initiale de défenseur.

Critique : notre programme lance les algorithmes que nous avons implémentés pour le programme de base puis choisit la solution qui requiert le moins de déplacement parmi les deux, il serait plus pertinent d'implémenter un nouvel algorithme spécialement pour ce problème, même si les résultats des autres algorithmes sont plutôt corrects.

5.1.4 Extension : Distance minimale entre joueurs

Une implémentation possible pour cette extension consisterait à modifier la relation de collision afin qu'elle prenne en compte la distance minimale entre deux joueurs de manière à relier deux positions comme étant trop proches.

Collision(p1,p2) = $\{1 : ||p1 - p2||_2 \leq R_{radius} + \text{minDist}, 0 : \text{sinon}\}$

Avec cette modification, nous pouvons juste faire en sorte que les solutions retenues ne contiennent pas d'arête de collision entre deux nœuds du graphe solution.

5.1.5 Extension : Vitesse maximale

Une implémentation possible pour cette extension correspondrait à modifier le calcul du tir intercepté pour qu'il prenne en compte un disque de rayon grossissant autour des défenseurs et un segment s'allongeant par tir, au fil du temps. Si l'extrémité du segment n'étant pas sous l'attaquant rencontre le disque avant de toucher les cages de but, alors le tir est intercepté.

Critique : en conditions réelles, il faudrait considérer le temps que le défenseur calcule la direction prise par le tir, et considérer l'accélération du robot (il ne pourra pas être à sa vitesse maximale dès le départ, et accélérer trop vite peut mener à des dérapages).

6 Algorithmes implémentés

Nous avons implémenté un algorithme approximatif qui va tout d'abord générer la liste des positions possibles sur le terrain ; une fois toutes les positions créées, il va commencer à faire le tour des positions en sélectionnant d'en garder ou d'en supprimer jusqu'à ce qu'il trouve une solution valide ou ait fini de parcourir toutes les solutions.

Nous avons actuellement implémenté trois types de recherche de solution, la première se base sur le nombre de tirs interceptés par une position, les tirs sont triés par nombre décroissant de tirs arrêtés et, lors du parcours, on garde les positions contenant au moins un tir non-défendu par une position déjà défendue.

La seconde méthode de recherche va favoriser les positions autour des attaquants, en ne gardant que les positions qui arrêtent tous les tirs d'un attaquant, car souvent, la meilleure façon de couvrir les tirs d'un attaquant est de se placer juste devant celui-ci, en revanche cette méthode

devient moins utile avec une distance minimale entre les joueurs plus grande et des tirs très espacés pour un seul joueur par exemple en tirant sur plusieurs buts éloignés, dans ce cas, il pourrait être envisageable d'implémenter un algorithme supplémentaire où l'on sélectionne les positions défendant tous les tirs menant à un but, mais cette méthode ne fonctionnerait vraiment que si les cages de buts sont petites et que l'on considère de multiples zones de gardiens pour un gardien au total (par la suite, nous supposerons que les extensions ne se combinent pas), en revanche elle pourrait être utile combinée à l'extension de vitesse maximale, mais est finalement redondante avec le premier algorithme.

Enfin, nous avons implémenté un algorithme de force brute qui regarde tous les n-uplets de positions possibles de défenseurs, de 1 à 8 (ou seulement pour le nombre de défenseurs indiqués, si extension de défenseurs initiaux), et s'arrête après avoir trouvé une solution optimale (minimisant la distance parcourue si extension défenseurs initiaux, demandant le moins de défenseurs sinon).

Une fois les solutions générées, elle passe ensuite par une fonction vérifiant leur validité, si une solution passe la fonction de validité alors elle est choisie puis écrite dans le fichier 'computed_solution.json' (il sera éventuellement possible dans une version future de donner un nom de fichier en entrée pour désigner le fichier de réponse).

Si les deux méthodes fonctionnent alors, si des défenseurs initiaux sont précisés, la méthode choisie est celle demandant le moins de distance totale à parcourir pour que les défenseurs atteignent leur point d'arrivée si des défenseurs initiaux lui sont spécifiés ; sinon, c'est la solution demandant le moins de défenseurs.

Si aucune méthode ne donne de solution satisfaisante, alors les deux algorithmes sont relancés sans la contrainte des extensions. Ces algorithmes ne constituent pas une méthode exacte, il est possible de ne pas trouver de résultat arrêtant tous les tirs, car après avoir sélectionné une position, nous ne revenons pas en arrière pour essayer de la permuter avec plus de positions arrêtant plus de tirs et ne causant pas de collision (on ne revient en arrière que pour supprimer les positions inutiles dont tous les tirs interceptés sont interceptés par d'autres positions retenues, afin de ne pas demander trop de performances pour un algorithme non-exact, sans être sûr que le *backtracking* le rende vraiment parfait), mais d'après les tests effectués ces algorithmes sont tout de même plutôt efficaces ; en revanche l'algorithme de force brute devrait être un algorithme exact dans lequel nous testons toutes les possibilités de positions et ensembles de positions possibles, il n'est lançable que depuis une option car il est probable que tester toutes ces combinaisons et permutations soit exponentiel en temps.

7 Extensions : fonctionnalités développées et envisagées

7.1 Fonctionnalités développées

7.1.1 buildGraph.py

Les fonctionnalités développées sont la recherche d'une solution valide sur les problèmes basiques, mais aussi sur les exemples correspondant aux différentes extensions comme le cas du gardien de but, le cas d'une distance minimale entre joueurs, le cas des positions initiales, le cas de multiples zones de but et le cas d'un arrêt en temps réel selon la vitesse de la balle et des robots. Une fois que le programme trouve une solution valide, il va ensuite l'écrire dans un nouveau fichier, nommé 'sol_to_<nom_du_problème>.json' si l'option de nommage ne lui est pas donnée, qui peut ensuite être passé comme argument au fichier main.py.

7.1.2 Arguments pour utiliser les fonctionnalités de buildGraph.py

Par défaut, le logiciel lancera sur le fichier de problème .json donné en deuxième argument (celui suivant ./buildGraph.py) les deux premiers algorithmes mentionnés (nommés "keepMostDefending" et "closeToOpponent") avec le plus d'extensions possibles (toutes celles spécifiés dans le problème .json), comparera les deux résultats, si aucun ne donne de résultat satisfaisant les contraintes relancera les algorithmes sans les extentions, puis sélectionnera le résultat répondant le mieux au problème (minimisant la distance à parcourir ou le nombre de défenseurs) pour l'écrire dans le fichier 'sol_to_<nom_du_problème>.json'.

Si une option est spécifiée, alors toutes les autres options sont désactivées, et ne seront activées que celles qui sont spécifiées :

— -h : affiche une message expliquant les différentes options
— -kd : lance l'algorithme "KeepMostDefending"
— -co : lance l'algorithme "CloseToOpponent"
— -bf : lance l'algorithme "Bruteforce"
— -re : Recommence sans les extensions dans le cas où aucune solution n'est trouvée avec
— -ga : prend on compte l'extension "GoalkeeperArea"
— -d : prend on compte l'extension "InitialDefenders"
— -ms : prend on compte l'extension "MaximalSpeed"
— -mi : prend on compte l'extension "MinDist"
— Il n'y a pas d'option pour activer l'option "MultiGoal" car elle sera toujours active.
— -na : sans -na le nom de la solution sera sol_to_<problem>.json, avec il prendra la nom écrit juste après -na.

Notez que python (ou python3), pygame, et numpy sont nécessaires pour faire tourner le programme.

7.1.3 randomProblem.py et runTests.sh

Afin de faire des tests sur l'efficacité des algorithmes, nous avons implémenté randomProblem.py qui génère un fichier de problème .json aléatoirement selon les arguments qui lui sont fournis, et runTests.sh qui lance des tests dont les entrées sont générées par randomProblem.py et affiche leur temps d'exécution et leurs résultats.

7.1.4 Arguments pour utiliser les fonctionnalités de randomProblem.py

Par défaut les informations du fichier généré seront identiques à celles de configs/basic_problem_1.json Ajouter un argument permet de modifier une valeur, après chaque argument il est possible d'ajouter une valeur après afin de donner la valeur exacte à mettre dans le .json, si elle n'est pas précisée, elle sera aléatoire.

— -ps : sans -ps par défaut pos_step = 0.01, sinon, pos_step prend une valeur aléatoire si pas de valeur précisée après -ps, sinon pos_step = la valeur précisée
— -fl : sans -fl, par défaut field_limits = [[-4.5,4.5],[-3,3]], sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -g : sans -g, par défaut goals = [[[[4.5, -0.5], [4.5,0.5]], [-1,0]]], sinon un seul aléatoire si pas de valeur précisée, sinon cette valeur (peut avoir plusieurs goals)
— -mg : sans -mg, par défaut goals = [[[[4.5, -0.5], [4.5,0.5]], [-1,0]]], sinon un ou plusieurs aléatoires si pas de valeur précisée, sinon valeur donnée
— -o : sans -o, par défaut opponents = [[0.5, 0.0], [-2.0, -2.0], [2, 1.0]], sinon un seul aléatoire si pas de valeur précisée, sinon cette valeur (peut avoir plusieurs goals)
— -mo : sans -mo, par défaut opponents = [[0.5, 0.0], [-2.0, -2.0], [2, 1.0]], sinon un ou plusieurs aléatoires si pas de valeur précisée, sinon valeur donnée
— -rr : sans -rr, par défaut robot_radius = 0.09, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -ts : sans -ts, par défaut theta_step = 0.031416, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -ga : sans -ga, par défaut goalkeeper_area ne sera pas présent, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -d : sans -d, par défaut defenders ne sera pas présent, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -bs : sans -bs, par défaut ball_max_speed ne sera pas présent, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -rs : sans -rs, par défaut robot_max_speed ne sera pas présent, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -mi : sans -mi, par défaut min_dist ne sera pas présent, sinon aléatoire si pas de valeur précisée, sinon cette valeur
— -na : sans -na le fichier se nommera random.json, sinon valeur à préciser après -na

7.2 Fonctionnalités envisagées

— Pour l'instant nous générons toutes les positions possibles des défenseurs sur la grille, il pourrait être plus pratique de ne générer que les positions étant sur la trajectoire des tirs. Cependant, cette méthode semble compliquée à implémenter, et la gestion que nous avons des positions à parcourir nous permet de ne perdre du temps que lors de la génération de la liste, du calcul des interceptions par position et du tri de la liste, mais pas lors de l'utilisation des deux algorithmes parcourant ces positions, en effet dès qu'un algorithme a trouvé une solution ou qu'il trouve une position ne couvrant aucun but (la liste étant triée, toutes les positions suivantes ne couvriront aucun but) il s'arrête là.

8 Critique des performances

8.1 Temps d'exécution

8.1.1 Remarques globales

Le temps d'exécution, bien qu'étant plutôt correct, est pour l'instant assez aléatoire, dépendant par exemple du succès de certains algorithmes et de la nécessité de recommencer la recherche de solution sans prendre en compte les extensions (ce qui pourra être modifié avec le système d'entrée d'options).

Le temps d'exécution du programme actuel pourrait être amélioré en réduisant la quantité de positions considérées pour les défenseurs en ne prenant pas en compte les positions ne défendant aucun tir par exemple, en n'utilisant qu'un seul système de recherche de solution plutôt que les deux, une fois le système d'option effectué, ainsi qu'en réduisant la quantité de tirs considérés avec le potentiel ajout des coordonnées du ballon, si un seul attaquant possède un ballon, le nombre de tirs possibles sera réduit en conséquence.

8.1.2 Algorithmes

Les algorithmes keepMostDefending et closeToOpponent donnent des temps d'exécutions plutôt similaires pour des données en entrées égales, en revanche l'algorithme bruteForce donne un temps bien trop grand : Si la réponse nécessite 1 ou 2 défenseurs, bruteForce prend à peu près autant de temps, voire moins que les autres algorithmes, soit 200ms avec les arguments suivants :

— pos_step = 0.5
— theta_step = 0.05
— robot_radius = 0.1
— 1 ou 2 adversaires
— goals posts = [[2, -0.5], [2, 0.5]]
— field_limits = [[-2, 2], [-2, 2]]

Dès 3 défenseurs nécessaires, bruteForce peut prendre des heures. Et si aucune réponse ne peut être trouvée, alors le temps est encore plus long que de tester jusqu'à 8 défenseurs.

8.1.3 Choix entre deux algorithmes

Lancer le programme sur basic_problem_1 avec keepMostDefending a pris 4126 millisecondes, avec closeToOpponent 4163ms, et les deux à la suite puis ensuite choisir le meilleur résultat a pris 4291 ms. Cela laisse penser que les algorithmes en eux-mêmes ne prennent pas très longtemps, c'est la préparation pré-algorithme (la génération des positions possibles par exemple) qui prend le plus de temps.

On peut noter que 4 secondes dans un vrai match peut être un peu long.

Dans la suite du document, les deux algorithmes keepMostDefending et closeToOpponent seront considérés comme plus ou moins équivalent, donc il ne sera pas nécessairement précisé que les tests ont été effectué séparément sur les deux et que les résultats sont proches. La plupart des calculs suivants seront faits sur l'exemple de basic_problem_1 sur lequel on ne changera qu'une variable.

8.1.4 pos_step

Plus pos_step est fine, plus l'algorithme prend du temps :

— lancer le programme avec une pos_step de 0.05 prenait environ 16.50 secondes

— pos_step = 0.10 prenait environ 4.50 secondes
— pos_step = 0.20 prenait environ 1.20 secondes
— pos_step = 0.30 prenait environ 0.70 secondes
— pos_step = 0.40 prenait environ 0.45 secondes

Cela peut-être expliquer par le fait que les positions possibles pour les défenseurs se font plus nombreuses avec l'espacement réduit.

8.1.5 field_limits

Augmenter la taille du terrain fait augmenter le temps de calcul :
— field_limits = [[-0.5,0.5],[-0.5,0.5]] prend 251ms
— field_limits = [[-0.5,0.5],[-5.0,5.0]] prend 959ms
— field_limits = [[-0.5,0.5],[-10.0,10.0]] prend 1722ms

De la même façon, plus le terrain est grand, plus il y a de positions possibles. Augmenter la largeur plutôt que la longueur ne change pas tant le temps de calcul :
— field_limits = [[-0.5,0.5],[-2.5,2.5]] prend 565ms
— field_limits = [[-2.5,2.5],[-0.5,0.5]] prend 610ms

L'aire du terrain restant la même, le nombre de positions possibles reste similaire.

8.1.6 goals

Augmenter la taille des buts augmente le temps de calcul :
— goals posts = [[4.5,-0.50],[4.5,0.50]] prend 4148ms
— goals posts = [[4.5,-1.50],[4.5,1.50]] prend 12472ms
— goals posts = [[4.5,-3.00],[4.5,3.00]] prend 23765ms

Cette fois-ci, il est plus probable que ce soient les algorithmes et non leur préparation qui prenne du temps, augmenter la taille des buts augmente le nombre de tirs cadrés qu'il faudra intercepter, donc il faudra chercher plus longtemps pour trouver les positions qui arrêteront tous les tirs.

De la même façon, augmenter le nombre de buts augmente le temps de calcul :
— basic_problem_1 prend 4080ms
— basic_problem_1 avec un but supplémentaire (plus petit) prend 8174ms
— basic_problem_1 avec deux buts supplémentaires (de même taille, plus petits que le premier) prend 13969ms

Augmenter le nombre de buts augmente aussi le nombre de tirs cadrés à devoir intercepter.

8.1.7 opponents

De la même façon, augmenter le nombre d'adversaires augmente le temps de calcul :
— 1 opponent prend 785ms
— 3 opponents prend 4763ms
— 5 opponents prend 8888ms
— 7 opponents prend 13969ms

Augmenter le nombre d'adversaires augmente aussi le nombre de tirs cadrés à devoir intercepter. Aussi, plus les adversaires sont proches du but, plus ils auront d'angles de tirs cadrés.

8.1.8 robot_radius

Modifier la taille des robots ne change pas beaucoup le temps de calcul :
— robot_radius = 0.01 prend 4244ms
— robot_radius = 0.10 prend 4243ms
— robot_radius = 0.20 prend 4498ms

Il faudrait des valeurs vraiment très grandes pour influencer le temps de calcul :
— robot_radius = 1.0 prend 2514ms
— robot_radius = 2.0 prend 450ms
— robot_radius = 5.0 prend 213ms

Plus le rayon des robots est grand, moins il y a de positions possibles sans collision.

8.1.9 theta_step

Augmenter le pas de l'angle de tir réduit le temps de calcul :
— theta_step = 0.01 prend 13271ms
— theta_step = 0.02 prend 6911ms
— theta_step = 0.04 prend 3627ms
— theta_step = 0.07 prend 2007ms
— theta_step = 0.10 prend 1479ms
Augmenter le pas de l'angle de tir réduit le nombre de tirs cadrés à intercepter.

8.1.10 goalkeeper_area

La taille et la position de la zone de gardien de but n'a pas d'influence majeure sur le temps de calcul :
— goalkeeper_area = [[4.0,4.5],[-0.5,0.5]] prend 4272ms
— goalkeeper_area = [[4.0,4.5],[-3.0,3.0]] prend 4170ms
— goalkeeper_area = [[2.0,4.5],[-0.5,0.5]] prend 4230ms
Si le fait d'imposer une zone de gardien de but oblige à l'algorithme de continuer jusqu'à trouver des positions de défenseurs hors de la zone, le temps supplémentaire n'est pas significatif, même avec une très grande zone de but :
— goalkeeper_area = [[2.0,4.5],[-1.5,1.5]] prend 4133ms
— goalkeeper_area = [[0.0,4.5],[-2,2]] prend 4359ms
— goalkeeper_area = [[-3.0,4.5],[-2.9,2.9]] prend 4228ms

8.1.11 defenders

Contrairement à ce que l'on pourrait supposer, augmenter le nombre de défenseurs initiaux n'augmente pas tant le temps de calcul du programme :
— 1 defender prend 4371ms
— 4 defenders prend 4280ms
— 8 defenders prend 5146ms
Étant donné que l'algorithme s'arrête de tourner une fois qu'il a trouvé une solution, alors s'il faut 3 positions pour arrêter tous les tirs, alors toutes défenseurs non-assignés à une position resteront à leur place, donc cela prend le temps de trouver 3 positions, pas 8 défenseurs.

8.1.12 ball_max_speed et robot_max_speed

Les différentes valeurs de vitesse de balle et de robots ne font pas beaucoup varier le temps de calcul :
— ball_max_speed = 3 et robot_max_speed = 1 prend 5937ms
— ball_max_speed = 3 et robot_max_speed = 5 prend 7169ms
— ball_max_speed = 5 et robot_max_speed = 3 prend 6433ms
— ball_max_speed = 10 et robot_max_speed = 5 prend 6320ms
En revanche la présence de ces extensions fait augmenter de deux secondes le temps de calcul par rapport à basic_problem_1, c'est possible que cela soit dû aux calculs différents pour calculer les interceptions.

8.1.13 min_dist

Avec cette implémentation, la distance minimale entre deux robots ne fait pas beaucoup varier le temps de calcul :
— min_dist = 0.1 prend 4302ms
— min_dist = 0.5 prend 4051ms
— min_dist = 1.0 prend 3514ms
Si contrairement à robot_radius, le temps de calcul ne descend pas au fur et à mesure que la distance minimale grandit, c'est parce que dans notre implémentation, la grille de positions possibles est générée selon robot_radius, et la collision dans la distance minimale n'est calculée que lors de l'éventuel ajout d'une position défendue à retenir. En effet, générer la grille selon la distance minimale enlèverait beaucoup de positions possibles, alors nous avons fait le choix de ne pas les enlever afin d'obtenir de meilleurs résultats.

8.1.14 Déduction

Les facteurs influant le plus sur le temps sont le nombre de positions possibles pour un défenseur et le nombre de tirs cadrés à interceptés. On peut noter que réduire le nombre de positions possibles et angles de tirs cadrés considérés réduirait le temps de calcul mais réduirait aussi les solutions possibles pour arrêter tous les tirs.

8.2 Efficacité

Bruteforce est l'algorithme donnant des résultats exacts et optimaux, mais sa complexité est trop grande, voire aléatoire, étant donné qu'on ne sait pas forcément combien de joueurs seront nécessaires pour défendre tous les tirs. Ensuite l'algorithme keepMostDefending est très efficace mais rien ne garantit son optimalité, il peut donner des résultats non-optimaux ou ne pas trouver de réponse alors qu'il en existe. Finalement l'algorithme closeToOpponent n'est efficace que dans des cas plutôt particuliers et prend généralement légèrement plus de temps que keepMostDefending. Si l'on devait garder un seul algorithme ce serait keepMostDefending, mais closeToOpponent lui est complémentaire dans certaines situations, et utiliser les deux algorithmes puis choisir la meilleure solution ne consomme pas beaucoup plus de temps.

Note sur la validité d'une solution : nous supposerons que les extensions ne se combinent pas, et qu'une solution est valide pour une extension même si elle ne respecte pas les besoins d'une autre extension, par exemple :

- La distance la plus courte à calculer en cas de défenseurs initiaux, ne prendra pas en compte de devoir faire des détours pour éviter les collisions, distances minimales, ou zones de buts.
- Il suffit qu'il n'y ait qu'un seul défenseur dans une zone de but pour que la solution soit valide, on ne vérifie pas s'il y a des positions initiales que c'est bien le même défenseur qui soit dans la zone de but initialement et dans la solution proposée.
- Nous supposons qu'il n'y a pas plusieurs zones de buts, il semblerait que les fichiers qui nous ont été donnés ne le supportent pas.

9 Proposition de modification de la formalisation du problème afin de le rendre plus réaliste

Une possible modification du modèle consisterait à ajouter la position du ballon. Ceci est faisable en ajoutant simplement un couple (x,y) correspondant aux coordonnées du ballon sur le terrain, ou, de façon plus simplifiée, d'ajouter une variable booléenne hasBall pour chaque attaquant. Grâce à cet ajout, il serait possible de diminuer la quantité de tirs à prendre en compte lors de la résolution du problème, ce qui permettrait de réduire le temps de recherche, cependant cela peut aussi poser problème, car en ne prenant seulement en compte les tirs basés sur la position du ballon, il serait envisageable que tous les défenseurs se retrouvent d'un côté du terrain rendant l'autre côté indéfendable ce qui poserait problème dans le cas ou l'attaquant fasse une passe à un autre attaquant se situant sur l'autre côté du terrain, mais cette possibilité est improbable puisque peu de défenseurs seraient nécessaires pour défendre un seul attaquant, les autres seraient donc en mesure de se déplacer pour défendre un attaquant venant d'obtenir la balle par passe ; aussi, il pourrait être envisageable de considérer les attaquants sans ballon comme des cages de buts secondaires avec une importance croissante selon s'ils sont proches des cages de buts.

Dans ce cas, il serait intéressant de considérer les tirs de passe en plus de tirs pour marquer, ces tirs seraient secondaires par rapport aux tirs pour marquer, mais il faudrait en intercepter le plus possible proche de nos cages de but tant que tous les tirs cadrés sont interceptés en priorité.

Dans le cas où l'on voudrait rendre plus réaliste la modélisation afin d'utiliser ces algorithmes dans des cas réels, il serait pertinent de calculer différents types de tirs, tels que des tirs courbés.