Rapport psc

groupe info

April 21, 2019

1 Introduction

sur vrep

2 Présentation du programme central

2.1 Vue d'ensemble

Comme précisé dans le rapport intermédiaire, le programme principale a été écrit en python, son rôle est de gérer l'ensemble des entrées pour produire les actions que doit réaliser le robot. Ces entrées sont : l'ensemble des distances envoyés par les capteurs présents sur le robot, les positions des éléments envoyés par la caméra puis le retour de l'Arduino après une tentative de réalisation d'une action donnée (figure 1).

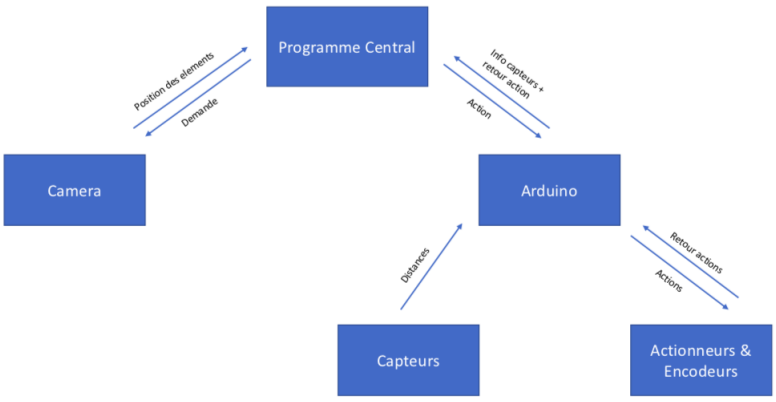

Figure 1: Schéma représentatif du fonctionnement du robot

2.2 Implémentation

Pour réaliser le schéma précèdent, le code a été conçu comme suit (figure 2):

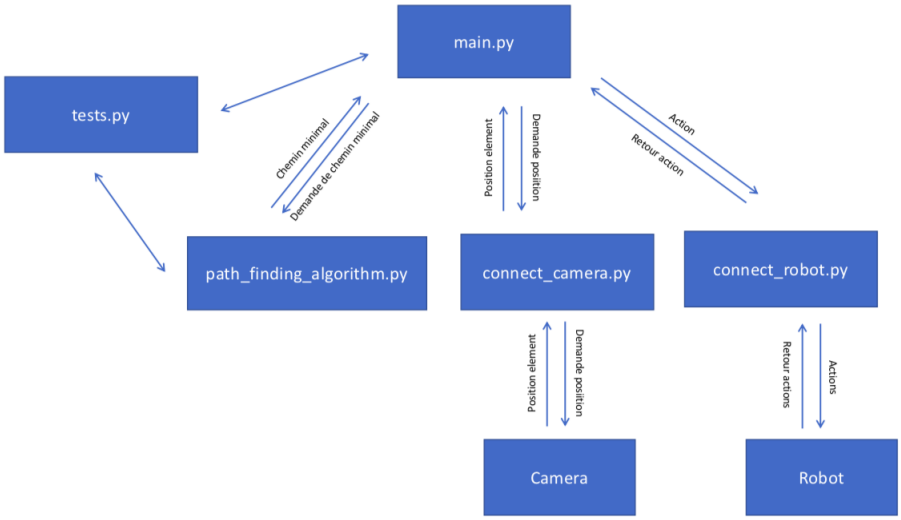

Figure 2: Représentation schématique du programme

- **main.py:** fichier principal qui traite les informations d'entrée et trouve à l'aide d'un algorithme de calcul de chemin (codé indépendamment dans le fichier path_finding_algortihm.py) la prochaine étape à faire par l'Arduino, puis en se basant sur le retour de celle-ci ainsi que des retours de la caméra, ce programme fait une mise à jour de ce qui est connu sur la table de jeu pour pouvoir générer à nouveau la prochaine étape à suivre.

- **connect_robot.py:** fichier contenant les fonctions qui permettent l'envoie des commandes à l'Arduino et la réception d'un retour.

- **connect_camera.py:** contient les fonctions nécessaires à la communication avec la caméra.

- **path_finding_algorithm.py:** contient l'algorithme utilisé pour le calcul des chemins lors du mouvement du robot.

- **tests.py:** utilisé pour tester l'ensemble des fonctions codés.

Conformément à ce qu'on avait pris comme convention au début, le retour de l'Arduino est de la forme "$a_1a_2a_3....a_{21}$" où $a_1 = 0$ si l'action demandée à l'Arduino n'as pas pu être accomplie (robot qui rencontre un obstacle dans son chemin par exemple) et $a_1 = 1$ dans le cas contraire, puis a_2a_3 , a_4a_5, .. $a_{2i}a_{2i+1}$ .. , $a_{12}a_{13}$ sont des nombres à deux chiffres représentant la distance en centimètres du capteur numéro i pour $a_{2i}a_{2i+1}$ à ce qu'il voit devant lui, finalement $a_{14}..a_{17}$ et $a_{18}..a_{21}$ représentent respectivement la distance rée le parcourue par le robot et l'angle réel de sa rotation au début de l'action en millimètres.

En ce qui concerne le retour de la caméra, celui ci sera de la forme "$a_1a_2....a_{243}$" (flux de taille $log_2(234) = 7.87bits$) où chaque 6 chiffres représentent respectivement les cordonnées (x, y) en décimètres d'un élément de la table (il y en a 40 : 37 atome et 3 robots) et les 3 derniers chiffres l'orientation du robot.

Pour rappel, le robot doit réaliser les taches suivantes: **déplacer les atomes**, **lancer une expérience**, **peser des atomes** ou **libérer un atome**. L'ensemble de ces actions peut être réalisé en utilisant cinq actions élémentaires, à savoir, **avancer**, **tourner à droite**, **tourner à gauche**, **attraper un élément**, **poser un élément**. C'est les cinq actions implémentés par l'algorithme.

La table de jeu est de dimensions 2 mètres x 3 mètres, le choix qui a été fait est de la représenter par une matrice de dimensions $[2\times ratio, 3\times ratio]$ initialisée à des zéros où le $ratio$ est une variable globale à choisir par l'utilisateur qui donne des résolutions différentes (et de là des précisions différentes du mouvement du robot), elle est à choisir selon la puissance de la machine exécutant le programme et les contraintes des cahiers de charge (nous l'avons initialisé à 100 ce qui veut dire une précision de 1cm dans les déplacements du robot) (figure 3)

Les différents cercles dans la figure (figure 3) représentent les zones interdites au centre du robot (figure 4), ainsi deux zones peuvent se superposer si les deux éléments correspondants sont proches à moins du diamètre du robot près. L'ajout d'un élément à la table se fait en augmentant les coefficients de la matrice dans la zone correspondante de 1, cette idée facilitera la suppression d'un élément en baissant simplement les coefficients de cette zone de 1 alors que si des zones superposés ont été représentés comme une seule zone de "uns" dans la matrice, la suppression de l'une d'elle supprimera une partie de l'autre (problème rencontré au début).

Le centre du robot doit alors se déplacer dans les zones nulles de la matrice pour réaliser les taches voulues.

Les éléments de la table sont représenté par les trois classes d'objets: *robot*, *atom* et *target* (qui est un élément fictif représentant un point de la table). Ces trois classes héritent d'une classe commune *element* vu leurs propriétés communes.

En ce qui concerne l'orientation du robot, celle-ci est repérée à l'aide de l'angle $\theta \in [-\pi, \pi]$ que fait le devant du robot avec l'axe des abscisses et qui est positif dans la direction des ordonnées positifs. (voir figure 5)

Voici la version la plus récente du programme: https://github.com/Coupe-de-France-de-robotiques/algorithmique/tree/master

N'hésitez pas à changer les paramètres globales en haut du fichier main.py

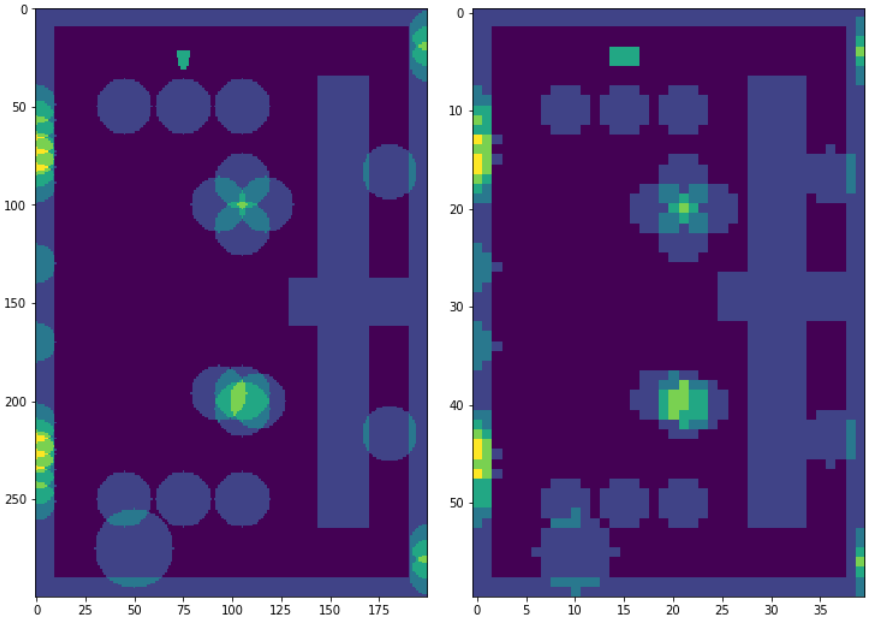

Figure 3: Table de jeu dans les deux cas ratio = 100 et ratio = 20

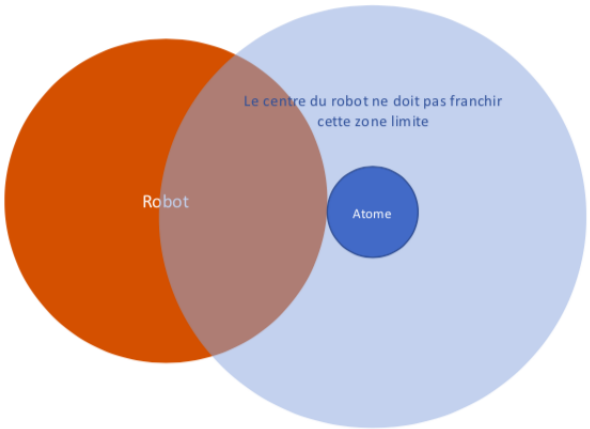

Figure 4: Zone interdite au centre du robot

pour voir les différents effets de ceux-ci et d'utiliser le fichier tests.py pour faire des tests (pour lancer la boucle principale, décommentez la fonction mn.action())

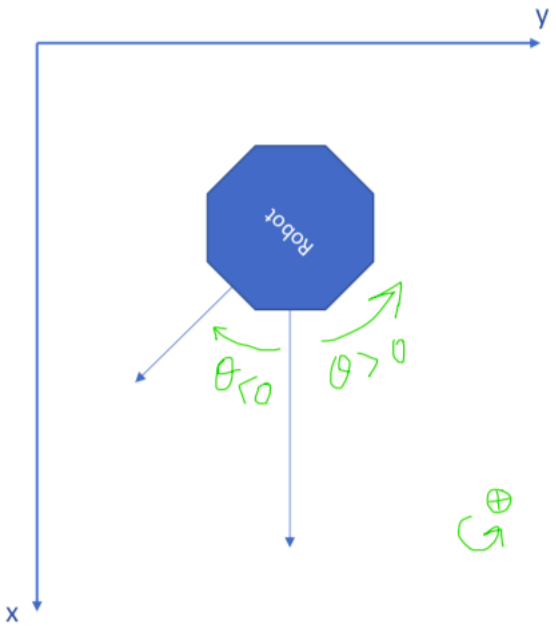

Figure 5: Angle de repérage du robot

2.3 Algorithme détaillé

Dans le fichier main.py, la table de jeu est initialisée avec la disposition initiale des éléments, puis un timer est lancé pour arrêter le programme après 100 secondes qui est le temps limite du match fixé par le règlement de la coupe. On rentre ensuite dans une boucle while qui fait les actions suivantes:

2.3.1 mettre à jour la table:

Cette fonction commence par demander l'état de la caméra, si celle ci marche, tous les positions des éléments de la table sont mis à jour.
Dans le cas contraire, une solution d'urgence est nécessaire, en effet, puisque le robot ne voit plus la dispositions de tous les éléments de la table celui ci ne peut que se contenter de ce qu'il savait déjà de la table et de ce qu'il voit directement devant lui à travers les capteurs ceci lui permet de mettre à jour sa connaissance sur la table du jeu tout au long de son chemin vers la cible.

Dans ce cas, la fonction commence par mettre à jour la position et l'orientation de notre robot en se basant sur le retour de l'Arduino après avoir tenté d'accomplir la tache précédente, puis elle essaie de deviner les éléments que voit le robot

à partir des données des capteurs et finit par mettre à jour les positions des éléments devinés.

Deviner un élément se fait de la manière suivante: On commence par repérer tous les pairs de capteurs successifs qui permettent de deviner le centre d'un élément de diamètre d [le diamètre de l'atome ici], sachant que le robot contient six capteurs distribués uniformément sur le devant du robot et à 27 l'un de l'autre (la disposition optimale de n capteur devant le robot sera de les mettre uniformément espacées dans les 135 du devant figure 6), ceci permet de deviner le centre de l'atome sachant la distance aux deux capteurs et le diamètre de l'atome (voir figure 7).

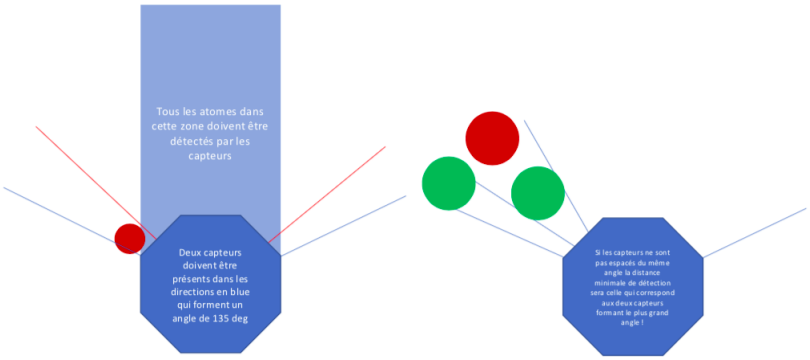

Figure 6: Positionnement optimal des capteurs pour éviter les zones mortes

Vu que les cordonnées des points A et B sont connues dans la figure 7 les distances aux capteurs doivent permettre d'avoir $d \leq 2r$, le calcul permet de trouver qu'avec six capteurs on peut trouver le centre d'un atome proche à 7 - 8 cm à peu près des deux capteurs (voir figure 8), ainsi on vérifie qu'on est bien dans la zone grise de la figure 8 avant de faire le calcul pour éviter les erreurs.

Il reste le cas où l'atome est sur les deux extrémités (c'est à dire qu'il est vu que par le capteur 0 tout seul ou le capteur 5 tout seul), dans ce cas on place l'atome en supposant que son centre est droit devant le capteur (c'est la solution la plus prudente).

Maintenant qu'on sait qu'il y a un atome à un point connu de la table (le centre calculé précédemment), il reste à choisir lequel des atomes doit on placer à cet endroit là, la solution choisie pour ce problème est d'y poser des atomes inutiles (dummy atoms). Remarquez que ce choix fait que si notre robot rencontre le robot adverse, celui ci le considère comme si c'était des atomes placés cote à cote devant le robot ce qui n'est pas un problème pour l'exécution de la suite des actions. Si il reste plus de dummy atoms, on reviens à la première et ainsi de suite. Si un atome inutile s'est fait bougé par le robot adverse ceci ne sera pas du tout un problème puisque ces atomes ne vont pas être

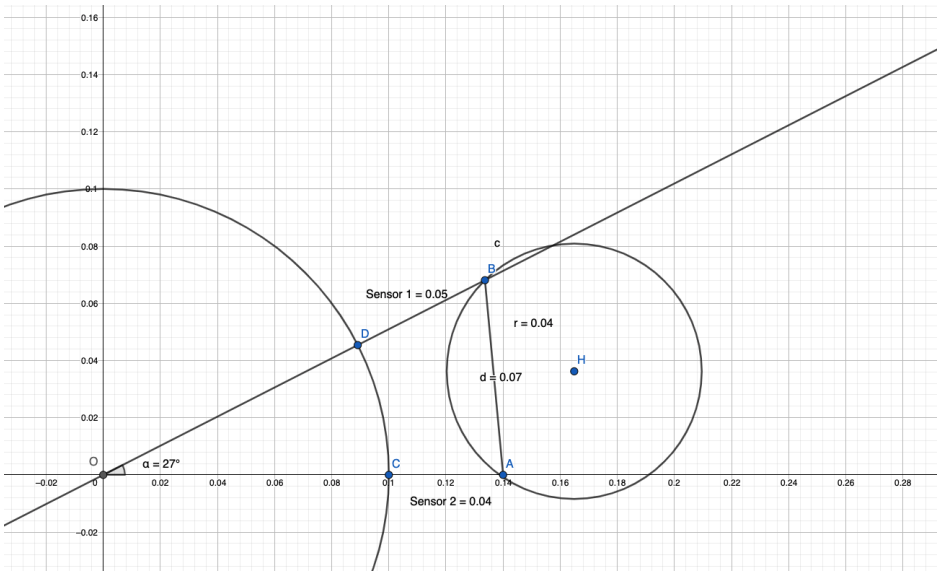

Figure 7: Sachant les distances DB et CA et le diamètre de l'atome, le centre de l'atome peut être calculé

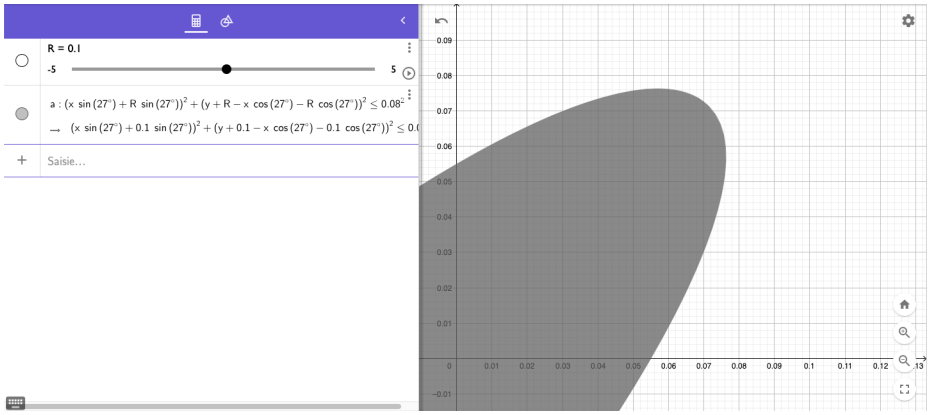

Figure 8: En gris les différentes pairs de distances (m) DB et CA qui vérifient la condition $d \leq 2r$ et permettent de calculer le centre de l'atome

utilisés dans des missions et par conséquence leurs positions exactes ne sont pas très importantes. Par contre s'il s'agit d'un atome principale ceci causera un problème et on aura plus la possibilité d'accomplir la mission correspondante ce qui est tout à fait normal car les capteurs seuls ne permettent pas de savoir ce qui se passe ailleurs d'où l'intérêt d'avoir le système de la caméra qui marche bien. En pratique ceci n'est pas un grand problème, en effet, le match ne dure

pas très longtemps et dans la plupart des cas les robots s'occupent des atomes proches d'eux et n'aurons pas beaucoup d'influence sur le reste des atomes.

2.3.2 envoyer l'action suivante:

Étant donnée la nouvelle configuration de la table, cette fonction commence par tester si la mission en cours (stockée dans une variable globale currentMission) est encore faisable, si c'est pas le cas elle passe à la prochaine mission après avoir ordonné l'Arduino de poser l'atome si le robot porte une.

Après avoir trouver la mission à accomplir (i.e il existe un chemin vers la cible), on extrait du chemin trouvé sa première partie qui est une ligne droite. On aura ainsi l'angle initial que doit faire le robot pour se mettre sur cette droite et la longueur de ce premier segment du chemin, puis on demande au robot de tourner puis avancer pour faire cette première partie. L'idée est que le robot fait à chaque fois une rotation puis une translation continue jusqu'au prochain coin dans le chemin, cette idée est justifiée par le faire que le robot s'arrêtera en tout cas pour tourner ainsi le mieux c'est de lui demander de faire toute la partie droite avant le prochain virage.

Il est à noter que pour récupérer un atome le point cible pour calculer le chemin optimal doit être un des points au bord de la zone interdite au centre du robot (figure 4) et que le robot doit après son arriver à ce point ajuster son angle d'orientation pour pouvoir rattraper l'atome. Pour des raison de simplicité le point cible considéré est le plus proche point accessible sur le bord de la zone parmi les quatre points extrêmes suivant l'axe des abscisses et l'axe des ordonnées. (figure 9)

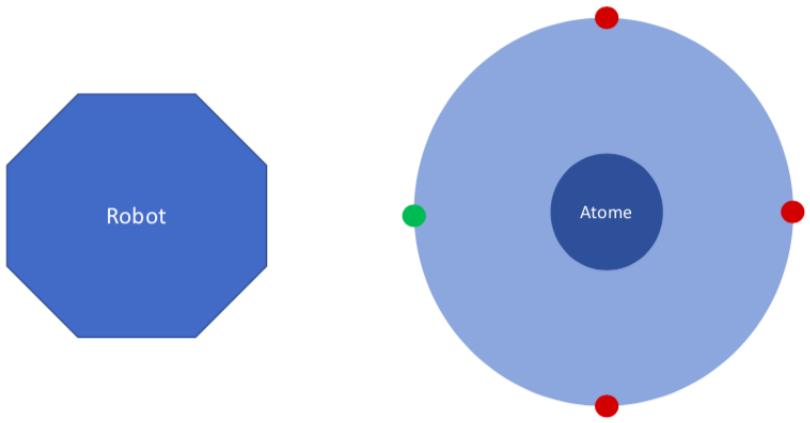

Figure 9: Le point cible qui sera considéré dans ce cas est le point en vert, c'est le point le plus proche au robot

Dans le cas où le robot arrive devant un atome, il l'a rattrape puis passe à la prochaine étape de la mission. Si le robot arrive à sa cible finale, il pose l'atome et augmente le score.

2.4 Algorithme du plus court chemin

L'algorithme de calcul du chemin optimal est implémenté dans le fichier path_finding_algorithm.py, vous trouverez dedans (au moment de l'écriture de ces lignes) deux versions d'algorithmes codé dans les fonctions **astarV1(array, start, goal)** et **astarV2(array, start, goal)**, *array* est une matrice où les zéros sont les cellules accessibles et les cellules contenant tout autre nombre représentent des obstacles, *start* et *goal* sont les deux tuples de cordonnées du point de départ et d'arrivée.

astarV1() implémente une version naïve de l'algorithme classique A* où les voisins de chaque point sont les huit voisins des cotés et des coins (les voisins de (0,0) par exemple sont [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]), cet algorithme de type greedy visite les points en commençant par ceux qu'il pense être les points les plus proches à la destination (dans notre cas l'algorithme considère la distance euclidienne du point en cours à la cible), c'est un des algorithmes les plus classiques de calcul de chemins optimaux vu sa simplicité et sa faible consommation en mémoire. Voici un exemple de calcul de chemin entre deux points de la table en utilisant astarV1() (figure 10)

Le problème dans cet implémentation naïve (et la motivation d'en trouver une amélioration) est le faite que celui-ci calcul des chemin certes optimales mais contenant beaucoup de virages (figure 10), sachant qu'à chaque itération de la boucle le robot reçoit un ordre de type *tourner* puis *avancer* et que faire tourner le robot consomme du temps, l'algorithme précèdent est loin d'être optimale si le chemin est évalué en terme de durée de parcours, l'idée est alors de trouver une amélioration de A* où les chemins contiennent le moins de virages possibles sans perdre de l'optimalité spatiale de l'algorithme, étant donné que le souci ici est de réaliser les taches le plus rapidement possible.

L'idée implémentée dans le astarV2() est la suivante: Étant donné deux points A et B de la table, au lieu permettre à l'algorithme de parcourir toute la table pour trouver un chemin, on commence par des résolution plus faibles et on augmente la résolution jusqu'à ce qu'on trouve le chemin, par exemple, dans la figure 11, on a commencé par chercher le chemin entre A et B avec une résolution $res_x = |A_x - B_x|$ en x et $res_y = |A_y - B_y|$ en y où A_x, A_y et B_x, B_y représentent les cordonnées du point A et B respectivement dans la matrice, ce qui veut dire que seuls les points de la table qui s'écrivent comme $(A_x + n \times res_x, A_y + m \times res_y)$ vont être parcourus par l'algorithme (dans le cas où l'un de ces deux résolutions et nul on lui donne 1), si un tel chemin existe celui-ci contiendra très peu de virages et sera ainsi plus rapide à parcourir, dans le cas contraire on augmente la résolution en choisissant de façon décroissante res_x dans l'ensemble des diviseurs de $|A_x - B_x|$ et res_y dans l'ensemble des diviseurs de $|A_y - B_y|$ (pour s'assurer que la grille $\{A_x + n \times res_x, A_y + m \times res_y\}_{n \in Z, m \in Z} \bigcap \{(x, y)\}_{(x,y) \in table}$ contient le point cible B) (voir figure 12)

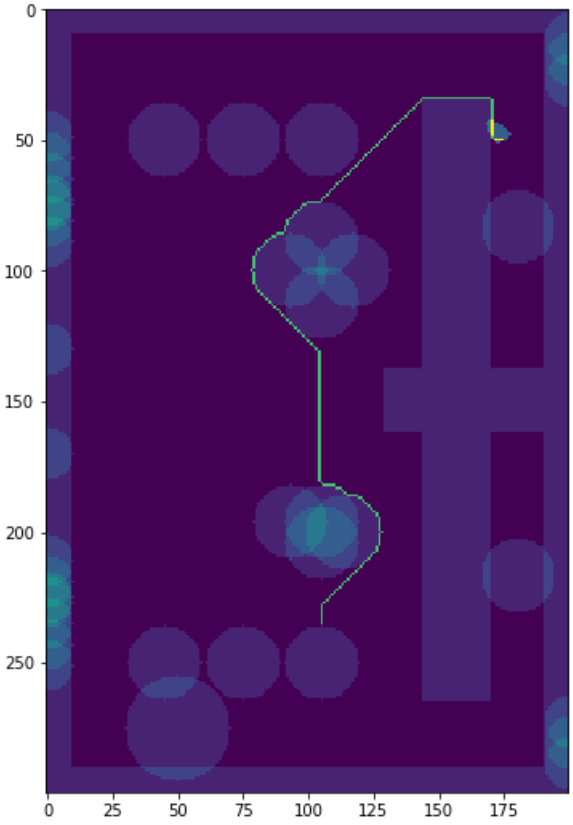

Figure 10: Calcul de chemin optimal en utilisant l'algorithme A* naïf

Pour mettre en place cette idée il suffit de modifier l'ensemble des voisins d'un élément dans l'algorithme naif, en effet, les voisins d'un élément (x, y) sous la résolution res_x, res_y seront cette fois ci $\{(a, b) \in \{(x + res_x, y), (x - res_x, y), (x, y + res_y), (x, y - res_y)\}$ tels que $table[i][j] = 0$ pour $i \in (a, x)$ et $j \in (b, y)\}$, c'est à dire les éléments voisins (avec la résolution considérée) qui sont accessibles en ligne droite depuis le point de départ (x,y). Si le chemin n'a pas été trouvé, on augmente la résolution en passant à de plus petit diviseurs et donc des voisins plus proches. Ceci veut dire que l'algorithme converge et on tombera dans le pire des cas sur la résolution (1,1) qui est rien que l'algorithme naïf astarV1().

Les nouveaux chemin sont maintenant plus rapide à calculer (le chemin peut être trouvé sans avoir besoin de passer à une grade résolution) et ont moins de virages (figure 13)

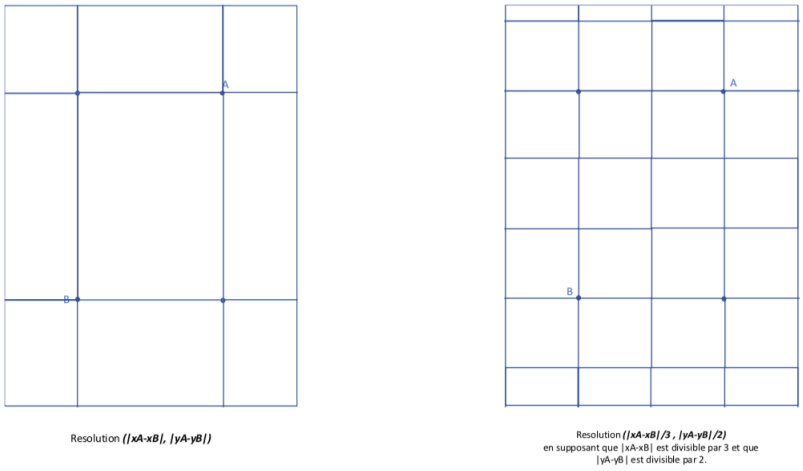

Figure 11: Implémentation de l'idée des résolutions

2.5 Tests

2.5.1 Construction de la table

Comme mentionné précédemment, la table du jeu est initialisée à une matrice de zéros, ensuite, les obstacles statiques de la table sont ajoutés et enfin on ajoute toutes les zones correspondantes à la disposition initiale des atomes. Les tests 1, 2, 3 et 4 permettent de tester cela. (figure 14)

2.5.2 calcul de chemin

Le test 5 et 6 permettent de calculer et dessiner le chemin optimal entre le robot et un atome de la table (pour passer de *astarV2()* à *astarV1()* changer le retour de la fonction *algorithme()* dans le fichier *path_finding_algorithm.py*) (figure 15)

2.5.3 fonctions utiles

La fonction *getThetaFromSourceToTarget()* du *test7()* permet de calculer l'angle entre une source et une cible avec la convention de la figure 5.

Dans le *test8()* et *test9()*, *updateOurRobotPosition()* et *updatePosition()* permettent de mettre à jour la position de notre robot et d'un élément quelconque de la table respectivement.

Ensuite le *test10()*, *test11()* testent les fonctions *getExpXY()* et *theRobotIsLookingAt()* qui calculent respectivement le centre d'un élément à deux distances données des capteurs et prédisent l'élément à partir d'une réponse de l'Arduino

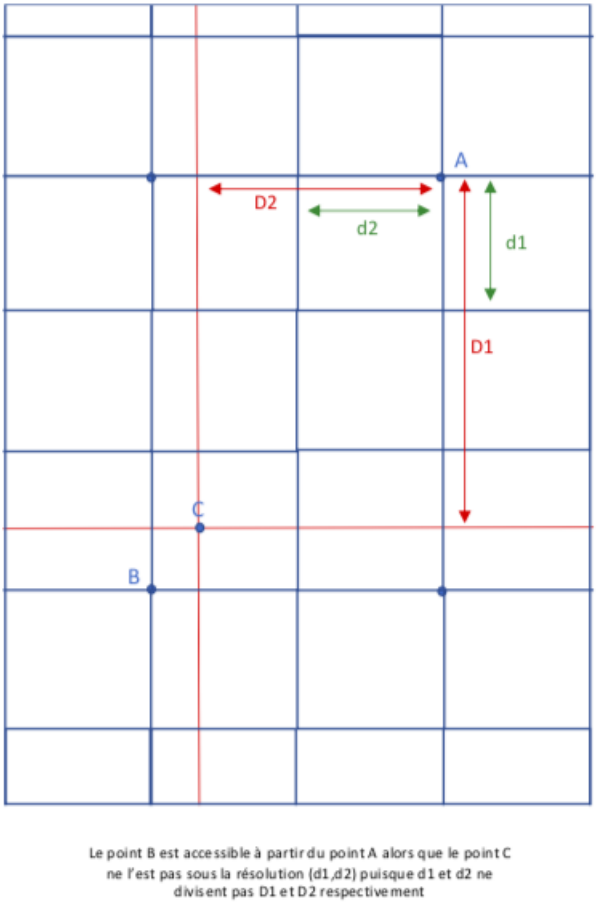

Figure 12: Points accessibles sous une résolution donnée

Finalement, *updateTable()* dans le *test12()* est une des deux fonctions principales du programme (avec la fonction *sendNextAction()*) qui permet de mettre à jour la table à partir d'une réponse de l'Arduino (et de la caméra au cas où celle ci marche bien)

Pour tester l'ensemble de l'algorithme *mn.action()* fait appel à la boucle principale qui lance le programme, vous verrez le robot qui est entrain de calculer son chemin, envoyer les consignes à *connect_robot.py*, qui lui rendra une réponse

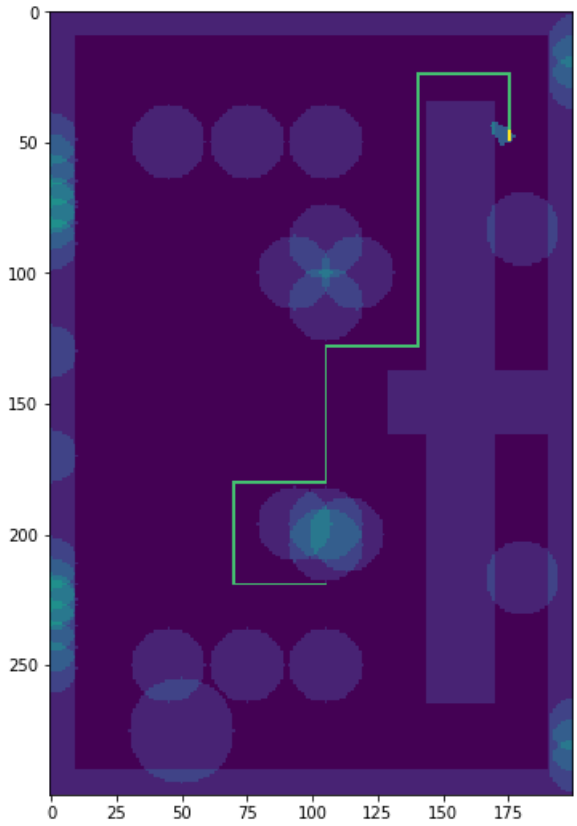

Figure 13: Calcul de chemin optimal en utilisant l'algorithme A* optimisé

puis une mise à jour de la table est fait où vous verrez que le robot a fait une rotation et le premier segment du chemin, le cycle recommence tant que la mission est faisable, dans le cas contraire le robot passe à la mission suivante. (16)

Finalement pour simuler un obstacle devant le robot, on doit jouer sur la réponse de l'Arduino, celle ci est codée dans le fichier *connect_robot.py* de façon à ce qu'elle envoie que le mission était bien accomplie et que le robot a bien fait la distance ou l'angle qui lui a été demandé. Pour simuler un obstacle, vous pouvez dé-commenter la partie correspondante (voir l'explication en haut du fichier *connect_robot.py*) et vous verrez que le robot s'arrête devant l'obstacle et calculera un nouveau chemin comme prévu (figure 17).

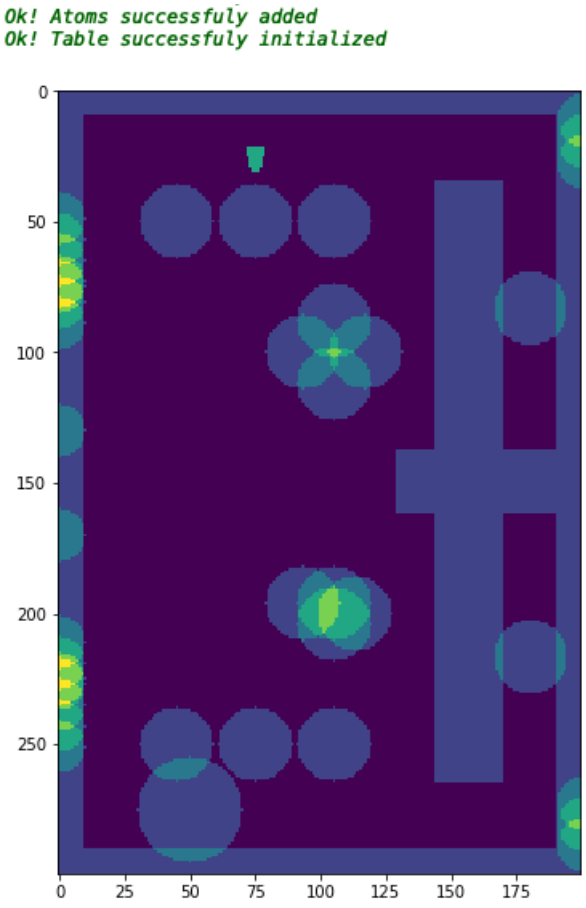

Figure 14: Construction et ajout des differents elements de la table

3 Point sur le travail fait par le group mécanique:

References

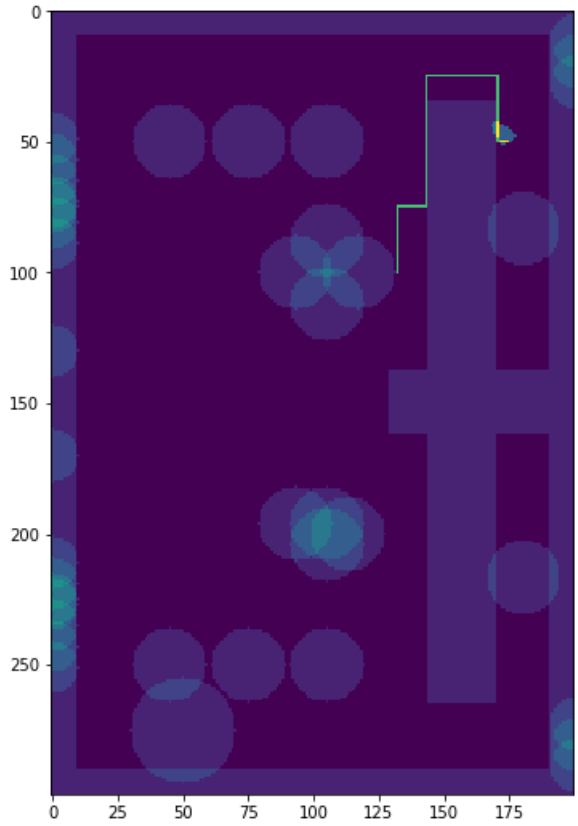

Figure 15: Calcul du chemin optimale à une atome de la table

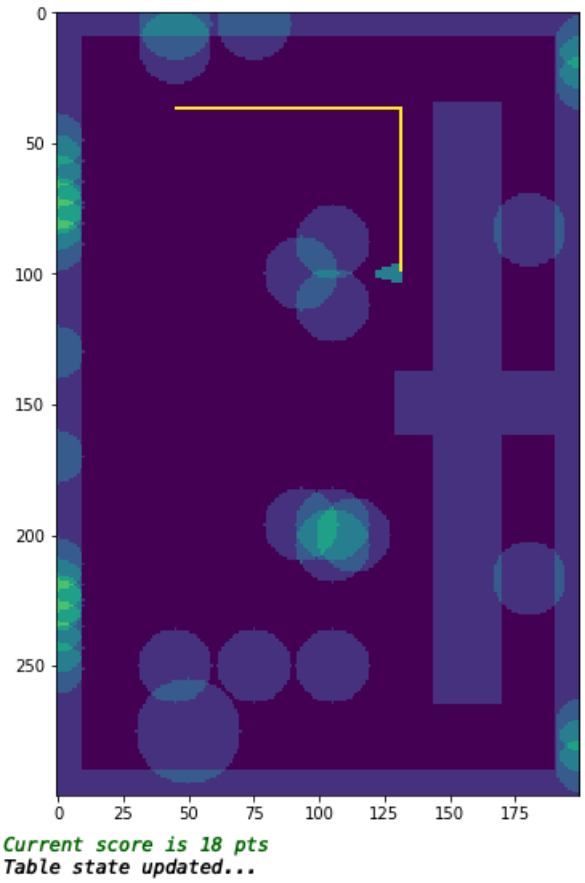

Figure 16: L'algorithme en action

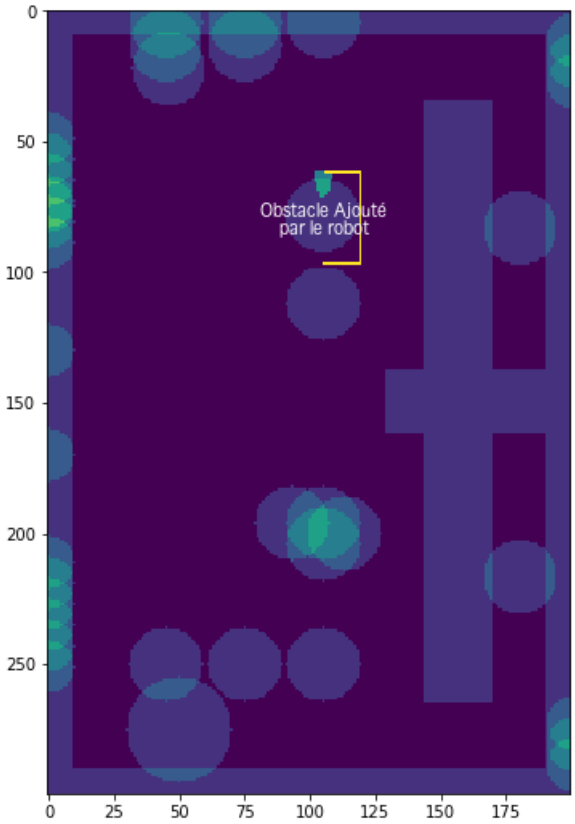

Figure 17: Ajout d'un obstacle par le robot et recalcule du nouveau chemin optimal