Bilan Première Partie de stage

Bernard Parfaite

June 2022

Introduction

Dans ce bilan nous allons parler d'abord des pré-requis qu'il faut ainsi que certaines fonctionnalités afin de pouvoir rapidement prendre en main la bibliothèque pionocchio et ensuite faire un petit bilan sur les différents algorithme de motion planning que nous avons eu à faire en utilisant la librairie pinocchio.

1 Pré-requis

Tout d'abord faut bien s'assurer d'avoir bien assimiler les notions de robotiques apprises, (avoir une vague idée sur le modèle géométrique direct, indirect , le modèle cinématique, dynamique,...) et pouvoir codé en python ou en C++.

1.1 Charger le robot

La première des choses à faire lorsque l'on veut utiliser pinocchio c'est de charger le modèle du robot avec lequel on veut travailler. Des exemples de robot avec leurs fichiers de descriptions sont déjà disponibles et il vous suffit juste d'importer la méthode "load" du package "*example_robot_data*" .Ce package viens avec l'installation de pinocchio. La méthode load prend en paramètre le nom du robot que vous voulez charger (Tip : Pour trouver ces noms de robots qui sont pris en paramètre vous pouvez juste mettre n'importe quel caractère en paramètre de la fonction load et un message d'erreur sera produit avec la liste des noms des robots que vous pouvez charger). Cette méthode renvoie un objet de type "RobotWarper" et c'est cette objets qui vous permettra d'accéder à toutes les informations du robot. Dans le cas où le robot avec lequel vous voulez travailler ne se trouve pas dans cette

liste faite en sorte de posséder le dossier de description du robot. Utilisez la méthode RobotWrapper.BuildFromURDF de la classe RobotWrapper que vous allez importer du package pinocchio.robot-wrapper. Cette méthode prend en paramètre le chemin absolue du fichier .urdf qui se trouve dans votre dossier de description ainsi que le chemin du répertoire où se trouve le dossier de description. Elle renvoie aussi un objet de type RobotWrapperer.

1.2 Visualiser le robot

Après avoir charger le robot, l'objet renvoyé a un attribut model et un attribut data. L'attribut model vous permet d'avoir des informations sur les articulations du robot(leurs noms ainsi que l'articulation (parent) à laquelle elles sont rattachées). Nous reviendront plus tard sur la nature de l'argument "data" . Vous avez la possibilité de visualiser le model 3d du robot charger grâce à "gepetto-gui" Après l'avoir démarrer sur un terminal, la méthode initViewer de l'objet RobotWrapper vous permet d'afficher le model 3d du robot sur la fenêtre de gepetto-gui(robot.initViewer(load = True). A présent vous pouvez afficher n'importe quel configuration avec "robot.display(q)" q étant votre configuration. Si vous avez du mal à choisir une configuration vous pouvez juste chercher la dimension de votre espace de configuration qui est la valeur de "robot.model.nq" et crée un vecteur de même dimension ou juste utilisé une méthode de pinocchio qui prend en paramètre le modéle du robot et renvoie un configuration aléatoire (pinocchio.randomConfiguration(robot.model)).

Vous pouvez ajouter des objets dans le visualiseur "gepetto-gui" en utilsant des méthodes de l'objet robot.viewer.gui (*robot* est le robot charger , viewer est un attribut de robot et gui celui de viewer). Pour par exemple ajouter une sphère (comme cible ou obstacle peut être) la méthode à utiliser est addSphere qui prend en paramètre le nom de l'objet, son rayon , sa couleur exprimer en rgb et une dernière valeur pour la transparence (robot.viewer.gui.addSphere('world/target',0.1,[1.,0.,0.,1.]).

Après avoir crée l'objet vous pouvez le placer dans l'environnement du robot en précisant le nom que vous lui avez donné ainsi que les coordonnées de là où vous le voulez tout cela en utilisant la fonction applyConfiguration ($robot.viewer.gui.applyConfiguration('world/target', [0.3, 0.5, 0.4, 0., 0., 0., 1]$). Et le vecteur permettant d'ajouter les coordonnées de l'objet doit être un quaternion. Puis utlisez la méthode refresh() pour rafraîchir la fenêtre du visualiseur et ainsi afficher l'objet crée (robot.viewer.gui.refresh())

1.3 Savoir se repérer

Savoir dans quelle repère on évolue lorsque nous notre robot fait un mouvement est très important. A noter que chaque articulation possède un repère et il primordial de pouvoir connaître la position de chaque parties du robot par rapport à un référentiel (le référentiel le plus facile à utilisé est le repère de la partie du robot qui reste immobile par rapport à toute les autres parties). Nous pouvons calculer l'emplacement des articulations grâce à la fonction forwardKinematics du package pinocchio qui prend en paramètre robot.model, robot.data et la configuration "q" que l'on veut (pinocchio.forwardKinematics(robot.model,robot.data,q). Ces emplacements calculés sont ensuite directement stockés dans l'attribut data que l'on avait cité plus haut. La liste de ces emplacements est robot.data.oMi[] et dans cette liste se trouve l'emplacement de chaque articulation sous la forme d'un élément de SE3 contenant la matrice de rotation ainsi que le vecteur de position .

Si vous voulez particulièrement utilisez l'emplacement d'une des articulations, commencez par trouver son indice. Pour cela vous pouvez afficher le contenu du modèle (robot.model), vous verrez la liste des articulations et en déduire leur indice suivant leur ordre. Alors l'emplacement de l'articulation est M = robot.data.oMi[indice].

Vous pouvez travailler aussi avec le repère de chaque articulation. Pour cela vous devez mettre à jour leur emplacement en utilisant la méthode pinocchio.updateFramePlacement(robot.model,robot.data) qui prend en paramètre que le modèle et les données. Même si la méthode forwardKinematics vous à permis de calculer l'emplacement des articulations elle ne met pas à jours ceux de leur repère . Pour vous éviter de faire cette mise à jour vous pouvez dés le départ utiliser la méthode pinocchio.frameForwardKinematics au lieu de forwardKinematics. Ils prennent les mêmes paramètres. Cette fois-ci les emplacements des repères se trouvent dans la liste robot.data.oMf[]. Pour trouver l'emplacement d'un repère en particulier vous devez d'abords trouver son indice dans la liste des repères qui se trouve dans robot.model.frames. Pour accéder aux noms de chaque repère vous pouvez parcourir la liste robot.model.frames[i].name. Vous pouvez trouver l'indice voulu si vous connaissez le nom de l'articulation avec la fonction robot.model.getFrameId('...') en lui donnant le nom en paramètre. Cette fonction vous renverra l'indice i du repère. Alors l'emplacement du repère est M = robot.data.oMf[i]. Si donc vous possédez déjà l'indice du repère vous pouvez aussi avoir son emplacement en faisant robot.model.frames[index].placement.

Pour n'importe quel emplacement obtenu, vous pouvez avoir la partie translation juste en faisant appel à l'attribut "translation" (M.translation) ce

qui vous renvoie un vecteur position. De même avec la rotation , M.rotation vous donne la matrice de rotation.

1.4 Calcul de différentiel, de dérivée et de dynamique

Si par exemple vous possédez une vitesse v_q avec une configuration de départ q_0 et que vous voulez trouver la configuration qui en résulte q vous pouvez le faire en utilisant la fonction $intergrate$ de pinocchio en lui donnant en paramètre le modèle($robot.model$) ,la configuration(q_0) et la vitesse (v_q)

(q =pinocchio.intergrate($robot.model$,q_0,v_q)

Si cette fois ci vous possédez une configuration de départ q_0 et un configuration d'arrivéeq vous pouvez trouvez la vitesse pour aller de l'un vers l'autre en utilisant la fonction $difference$ de pinocchio qui prend en paramètre le modèle $robot.model$, la configuration de départ q_0, celle d'arrivée q et renvoie la vitesse . ($v_q = pinocchio.difference(robot.model, q_0, q)$)

Dans le modèle cinématique vous aurez peut être à calculer la jacobienne d'un repère. Dans ce cas calculez d'abords l'emplacement des articulations en utilisant la fonction citée plus haut $forwardKinematics$, puis calculez les jacobiennes des articulations en utilisant : pinocchio.$computeJointJacobian$ qui prend en paramètre le modèle, les données et la configuration (les même arguments que $forwardKinematics$). Puis mettez à jours les emplacements des repères de chaque articulation avec la fonction que l'on a déjà vue et qui est $updateFramePlacement$. Et maintenant pour avoir la jacobienne de votre repère utilisez la fonction $pinocchio.getFrameJacobian('...')$. Cette fonction prend en paramètre le modèle, les données, l'index du repère que l'on a évoqué plus haut , puis le repère de référence par rapport auquel vous voulez calculer votre jacobienne. Si le repère de référence est le repère local vous pouvez l'obtenir en faisant pinocchio.ReferenceFrame.LOCAL et si par contre c'est le repère monde , le référentiel est obtenu par pinocchio.ReferenceFrame.WORLD .

Avec pinocchio vous pouvez aussi travailler dans le modèle dynamique. Les algorithmes : RNEA (Recursive Newton Euler Algorithm) , ABA (Articulated Body Algorithm), et le CRBA (Composite Rigid Body Algorithm) sont déjà implémentés dans pinocchio. Pour calculer la dynamique inverse, utiliser la fonction $pinocchio.rnea(robot.model, robot.data, q$,v$_q$,$a_q)$

qui prend en paramètre le modèle, les données, la configuration q , la vitesse v_q et l'accélération a_q. Pour calculer par contre la dynamique a_q, vous pouvez utilisez la fonction pinocchio.aba(robot.model,robot.data,q,v_q,τ_q).

Elle prend en paramètre le modèle, les données, la configuration q , la vitesse v_q et la dynamique inverse τ_q.

Pour calculer la matrice de masse M il faut utiliser la fonction pinocchio.crba(robot.model, robot.data,q) .

Elle prend en paramètre le modèle, les données et la configuration. La fonction $pinocchio.computeAllTerms(robot.model, robot.data, q, \mathrm{v}_q)$

permet de calculer le terme $b(q, v_q) = c(q, v_q) + g(q)$ et le stocke dans robot.data.nle .Elle calcul aussi en même temps la matrice de masse M et le stocke dans robot.data.M

1.5 Collisions

Vous avez la possibilité d'ajouter dans l'environnement du robot des obstacles que vous pouvez vous même crée. Dans la classe GeometryObject de pinocchio vous pouvez par exemple crée un obstacle en forme de capsule avec la fonction CreateCapsule qui prend en paramètre le rayon et la longueur de la capsule. Cela vous renvoie un objet dont vous avez à préciser la couleur (obstacle.meshColor = [1.0 ,0.2, 0.3 ,1.0]) qui est un vecteur de la forme rgb transparence,le nom (obstacle.name = 'obstacle'), l'objet parent (obstacle.parentJoint = 0 avec 0 indice de l'univers), l'emplacement de l'objet (obstacle.placement = matrice SE3). Ensuite l'ajouter dans le modèle de collision du robot ainsi que dans son modèle visuel :

$robot.collision\_model.addGeometryObject(obstacle)$

$robot.visual\_model.addGeometryObject(obstacle)$. Pour trouver le nombre de modèle géométrique dans le modèle de collision on a: $robot.collision\_model.ngeoms$
Pour supprimer toutes les paires de collision vous faite :

$robot.collision\_model.removeAllCollisionPairs()$. Pour créer une paire de collision vous avez : pinocchio.CollisionPair(..,..). Et pour ajouter une paire de collision : $robot.collision\_model.addCollisionPair()$ et mettre en paramètre la paire de collision préalablement créée. Il ne faut pas oublié à la fin de générer les données correspondantes :

$robot.collision\_data = pinocchio.GeometryData(robot.collision\_model)$

$robot.visual\_data = pin.GeometryData(robot.visual\_model)$. Pour vérifier si le robot est en collision avec les obstacles ajoutés dans son environnement vous devez d'abord mettre à jour l'emplacement de l'ensemble des objets géométriques de l'environnement en faisant :

$pinocchio.updateGeometryPlacements(robot.model, robot.data, robot.collision\_model,$
robot.collision$_data, q)$, donc il prend en paramètre le modèle , les données , le modèle de collision, les données de collision et la configuration. Et ensuite la fonction

$pinocchio.computeCollisions(robot.collision\_model, robot.collision\_data, False)$
renvoie $True$ ou $False$ s'il y a collision ou pas .Si la valeur du dernier

paramètre est $True$, alors la fonction arrête la boucle sur les paires de collisions lorsque la première collision est détectée Il est possible de trouver la distance qui sépare le robot de l'environnement en faisant :

$pinocchio.computeDistances(robot.collision\_model, robot.collision\_data)$

qui retourne un index. Cette fonction enregistre les distances calculées dans les données de collisions précisément dans $robot.collision\_data.distanceResults$. Dans cette liste a l'indice renvoyé par la précédente fonction vous avez la liste des distances calculées entre le robot et les obstacles.Vous pouvez récupérer la distance minimal en faisant : $robot.collision\_data.distanceResults[index].min\_distance$

2 Bilan des algorithmes de motion planning

2.1 Description de l'algorithme n°1

Pour cherchez la cible ,notre premier algorithme part d'une configuration sans collision aléatoire ou choisi. Et de cette configuration q il va chercher à s'approcher de la cible en faisant des petits pas aléatoires dq en vérifiant que q+dq est sans collision et que q+dq nous rapproche de la cible. Et le même schéma est suivi avec 130 itérations.

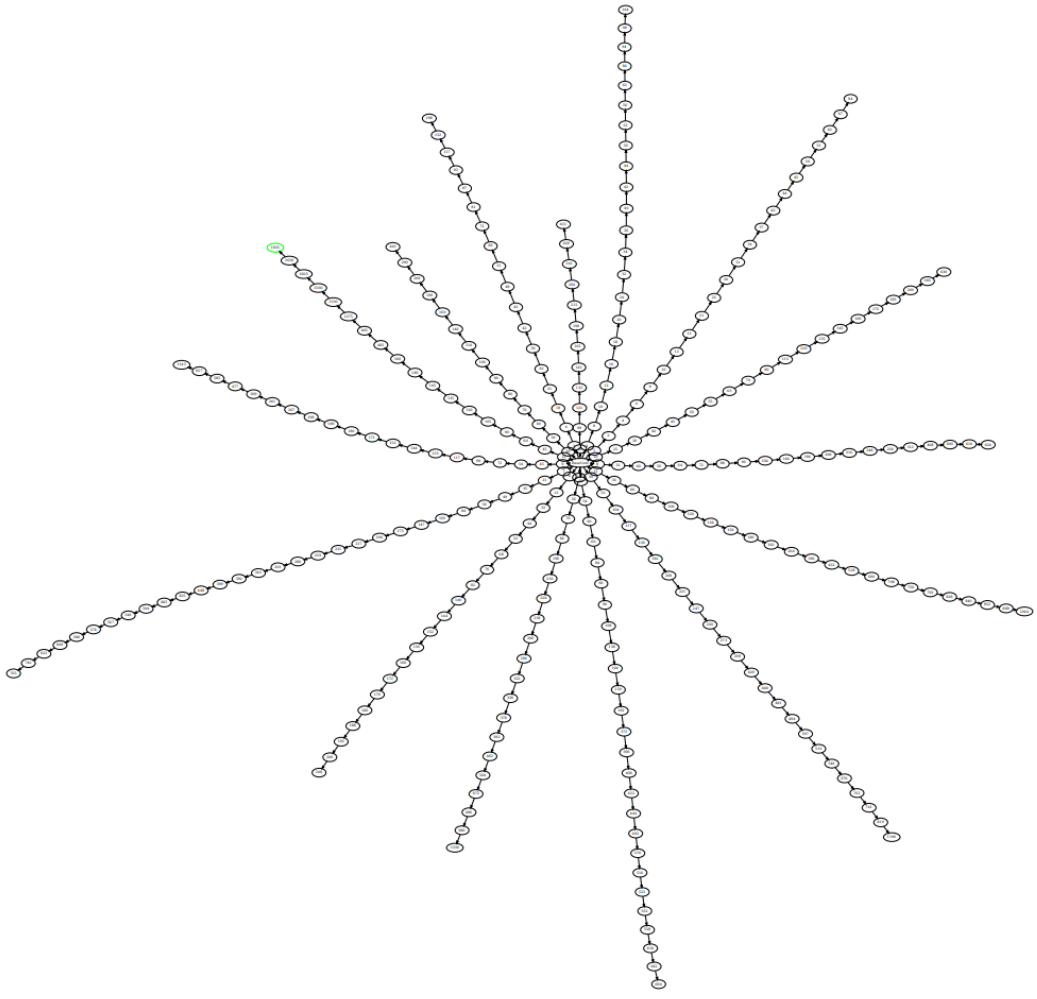

Figure 1: visualisation Graphique de l'algorithme

2.2 Description de l'algorithme n°2

L'algorithme n°2 est une variante du rrt . Il choisit seulement les configurations aléatoires qui rapproche l'organe terminal du robot de la cible. Ce qui le diffère encore du rrt c'est qu'il va d'une configuration à une autre que si le chemin entier est sans collision , il ne s'arrête pas tout juste avant la collision comme le rrt. En quelque sorte cette algorithme à plus de contrainte que le rrt.

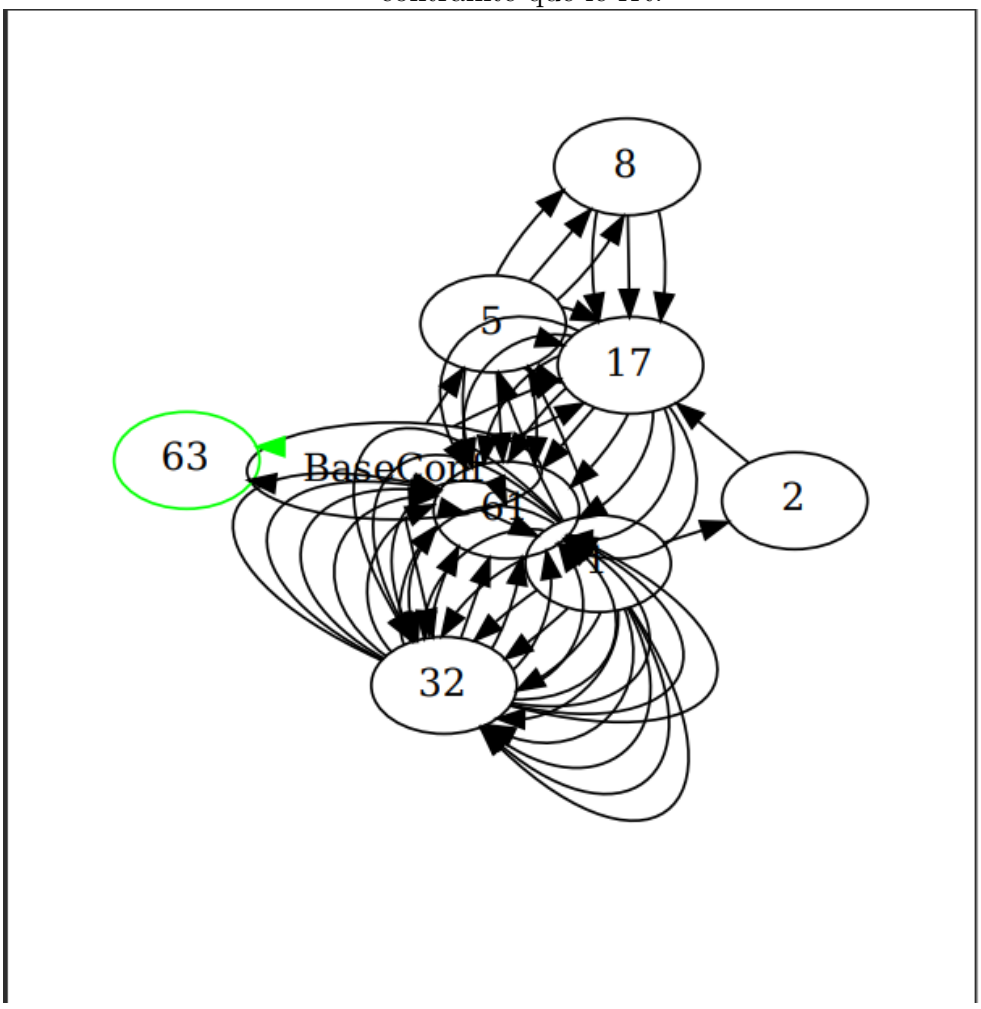

Figure 2: visualisation Graphique de l'algorithme

2.3 Description de l'algorithme n°3

L'algorithme n°3 est un *rrt* . Il part d'une configuration sans collision choisie ou aléatoire. De cette configuration, Pour arriver à la cible, il produit une configuration aléatoire sans collision puis il cherche parmi les configurations produites celle qui est le plus proche de la configuration nouvellement produite . Puis de la configuration la plus proche il essaie d'avancer jusqu'à elle avec des petits pas de dq = (q_2 - q_1) *0.05. Il s'arrête dés qu'il détecte une collision et ajoute dans la liste des configurations produite celle qui précède d'un pas dq la collision.Les configurations aléatoires produites sont biaisées, c'est à dire qu'une fois sur 3 il produit une configuration proche de la cible. Et tout ce processus boucle à l'infinie jusqu'à ce qu'il trouve la cible. (Pour les tests nous avons était obligé à certain moment d'arrêter l'exécution de l'algorithme quand il dépassais un certain temps)

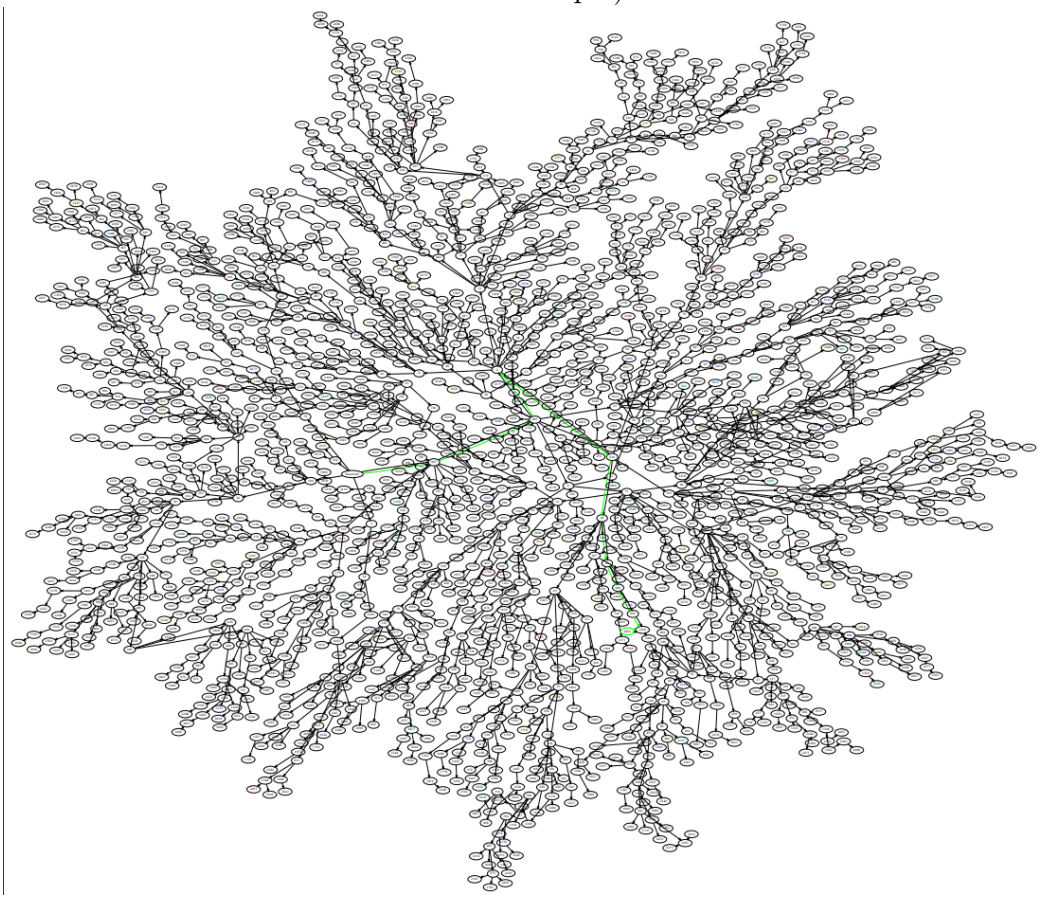

Figure 3: visualisation Graphique de l'algorithme

2.4 Description de l'algorithme n°4

L'algorithme n°4 est le même que l'algorithme 2 sauf que ici l'aléatoire n'est pas biaisé.

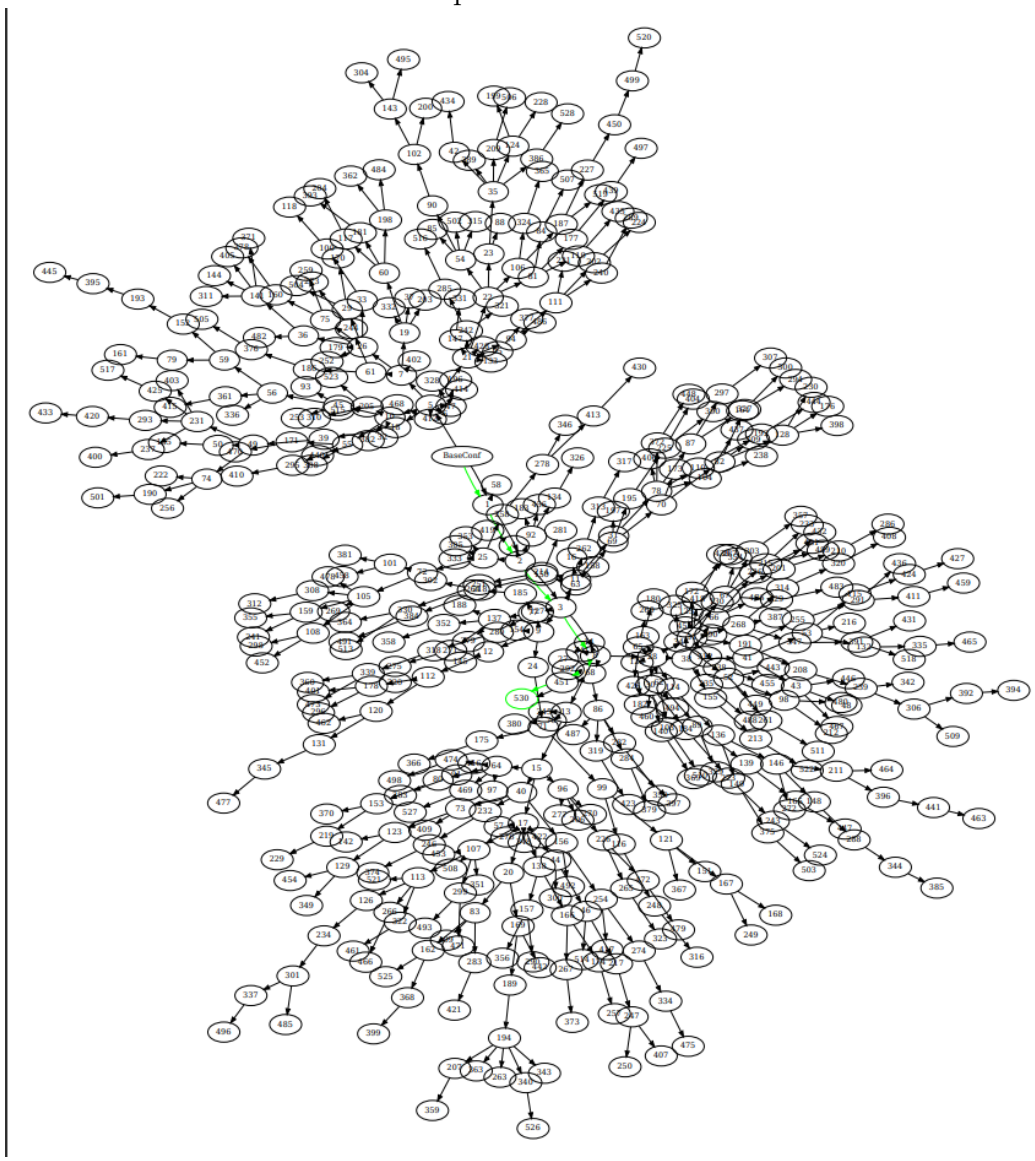

Figure 4: visualisation Graphique de l'algorithme

2.5 Résultats des tests

Pour mener une étude comparative entre les différents algorithmes, nous les avons testés avec 10 configurations différentes dans un environnement relativement simple(réduction de la taille des obstacles) puis dans un environnement complexe (augmentation de la taille des obstacles).

2.5.1 Environnement simple

Algorithme	Temps moyen de calcul	Nombre de configuration moyen	Fréquence de réussite
Algo n°1	0,146262074	16,55555556	0,9
Algo n°2	18,92638201	2,666666667	0,9
Algo n°3	1,850170851	4,5	0,2
Algo n°4	26,45436519	5,333333333	0,3

Figure 5: Résultats des tests

Ici dans ce tableau nous avons le temps moyen de calcul en seconde lorsque la cible est trouvée, le nombre moyen de configuration qui nous permet d'arriver à la cible et la fréquence à laquelle il trouve la cible sur les 10 essais.

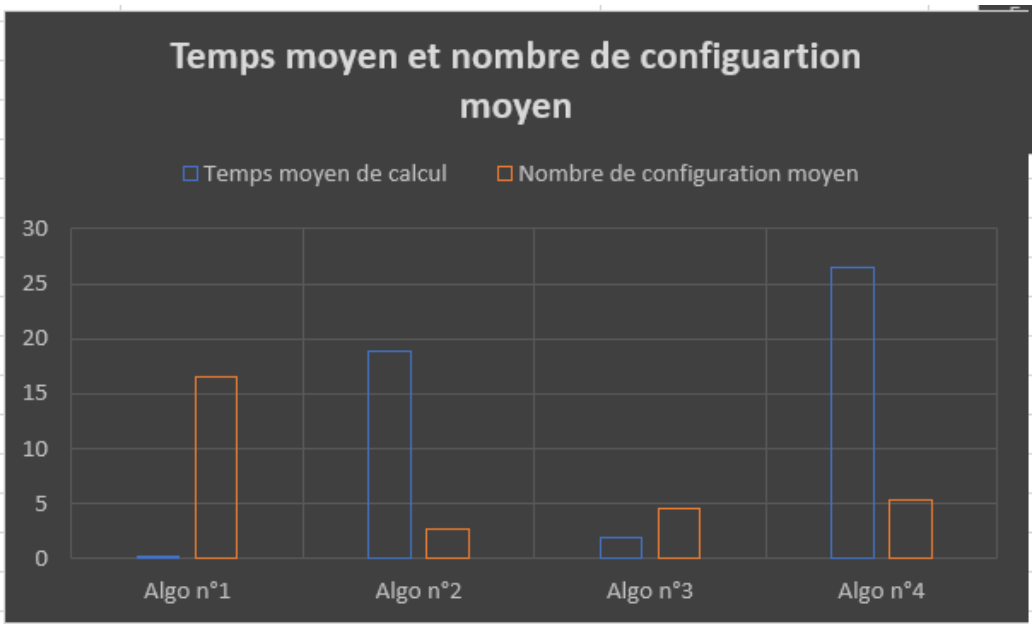

Figure 6: Temps de calcul moyen et nombre de configuration moyen trouvé des différents algorithmes

2.5.2 Environnement Complexe

Algorithme	Temps de calcul moyen	Nombre de configuration moye	fréquence de réussite
Algo n°1	0,023692346	21,8	1
Algo n°2	29,49733673	2,777777778	0,9
Algo n°3	18,89863992	7	0,3
Algo n°4	21,88089943	4,666666667	0,3

Figure 7: Résultats des tests

Ici aussi comme dans l'environnement simple nous avons le temps moyen de calcul en seconde lorsque la cible est trouvée, le nombre moyen de configuration qui nous permet d'arriver à la cible et la fréquence à laquelle il trouve la cible sur les 10 essais.

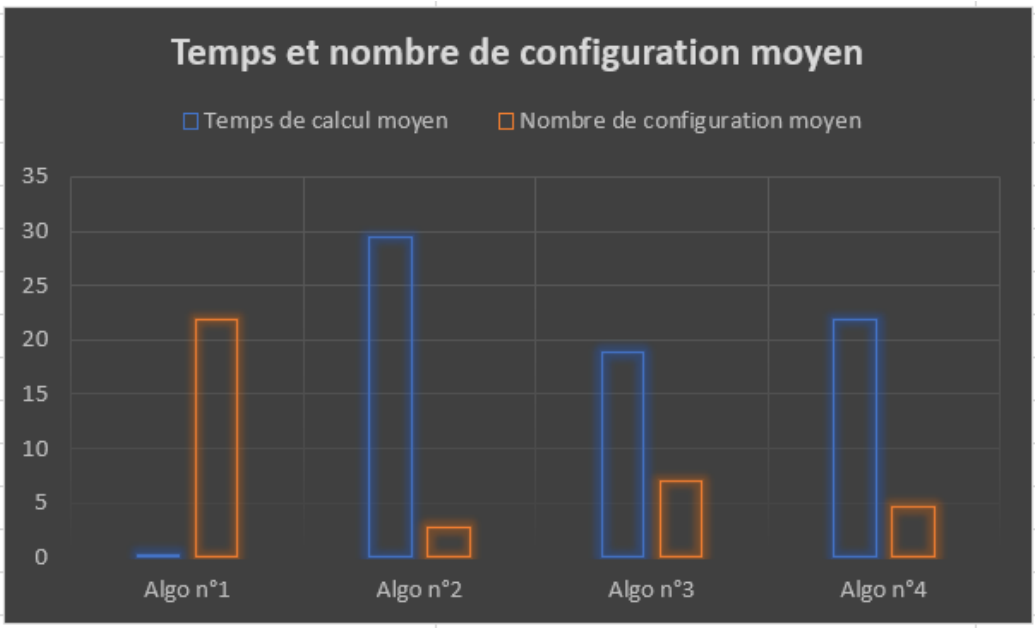

Figure 8: Temps de calcul moyen et nombre de configuration moyen trouvé des différents algorithmes