# Adaptation du RRT à un robot (1,1)

Description :

Plutôt que de verifier si une configuration tirée est atteignable, on va fabriquer une configuration atteignable, en tirant une commande et en l'appliquant à un noeud random de l'arbre. Puis on testera si le point (x,y) atteint n'entre pas en collision avec un obstacle...

Notons que nous ferons un effort pour tirer des commandes « réalistes » et applicables (en particulier : continuement dérivables) autant que possible.

Notes sur la structure de données utilisées : Structure d'arbres créés à partir d'une classe Nœud

```
class Node:
    def __init__(self, q=None):
        if q is None:
            self.q = [0, 0, 0, 0]
        else:
            self.q = q
        self.neighbors = []
        self.commands = []  # For each neighbor, the command to reach it
```

Problématique du *merge* :

Un nœud contient ses voisins et les commandes à appliquer au robot (1,1) pour les atteindre. Nous avions commis l'erreur initiale de définir plutôt un nœud par ses successeurs ou « fils », ce qui posait un problème lors du *merge* puis du A* (les arbres n'étant pas construits dans le même sens). Si on espère contourner le problème en appliquant le A* 2 fois entre q1 et qmerge, puis q2 et qmerge, on perd la possibilié d'envisager plusieurs *merge*.

Cette erreur est importante car elle découle naturellement de l'appelation -un peu fausse- d'arbre. Si T1 et T2 grandissent bien comme des arbres, leur regroupement futur sera un graphe, et ne peut le *merge* ne peut donc se faire correctement que si on considère les arbes comme des graphes dès le départ.

Problèmes rencontrés et paramètres importants :

La convergence de cet algorithme s'avère complexe, étant donnée la modification théorique appliquée (cf. Description). Il est également important de se familiariser avec les paramètres de réglage, qui pourront avoir un effet déterminant sur la réussite de la méthode.

Première approche du problème de convergence : les difficultées propres au robot (1,1) (commande d'un Béta point).

- ➔ le robot tend à tourner en rond -> agglutination des trajectoires vers le point de départ, absence d'exploration de la map

Notre première approche était de tirer une commande aléatoire (V et Béta_point entre 0 et un maximum) puis un noeud aléatoire dans l'arbre où l'appliquer. L'observation est claire: le robot n'explore pas la map, il reste globalement coincé près du point de départ.

- ➔ Le premier problème est qu'il va tourner en rond énormement, trop pour s'éloigner réellement. La méthode de détermination de béta_point, celle de v et le temps d'application de la commande (step_size) jouent un rôle crucial sur ce phénomène.

- ➔ Le second problème vient du non respect de la méthode RRT à deux niveaux. Premièrement : tirer un point aléatoirement sur la map et s'en approcher, ce qui assure d'aller explorer dans toutes les directions (ce que tirer une commande aléatoirement n'assure pas, encore moins dans le cas du robot (1,1) qui tend à tourner en rond). Deuxièmement le principe de sélection du plus proche voisin d'où commencer un mouvement d'exploration vers le point aléatoirement tiré (le remplacer par un tirage aléatoire ne permet pas de concentrer l'extension du graphe sur ses nœuds périphériques, ce qui est pourtant crucial).

- ➔ Le troisième problème est plus un phénomène qui accentue les problèmes ci-dessus. Plus les chemins s'agglutinent « inutilement », plus ils vont avoir tendance à le faire. Cela est vrai en particulier en regard de la dernière remarque ci-dessus : on a de moins en moins de chances d'explorer le graphe lorsque le point d'application de la commande ne se trouve pas en périphérie du graphe, et de moins en moins de chances de tirer de tels points de départs que le nombre de noeuds dans l'arbre augmente.

Tous ces problèmes, appuyés par ce dernier cercle vicieux, empêche complétement l'étalement de l'arbre sur la map. La complexité n'étant pas linéaire (plutôt quadratique en le nombre de noeuds), on se rend compte du besoin d'optimiser le comportement d'exploration sans rien laisser au hasard, sous peine de voir les arbres prendre de plus en plus de temps pour s'étaler. On peut résumer cette observation ainsi de manière dramatique et théâtrale :

# ##

> *« L'algorithme RRT pour le robot (1,1) convergera rapidement, ou il ne convergera pas du tout ! »*
>
> *-Nous, Mars 2018*

# Global variables:

*step_size = 3.0*

Détermine le temps d'application d'une commande tirée.

*merge_threshold = 4*

Distance à partir de laquelle on tente de merger deux points proches de chacun des arbres.

*Ne = 50* # nbr de pas d'integration pour l'application de la commande

*Nc = 10* # nbr de pas pour la detection de collision lors du merging

*Vmax = 10.0*

Si elle est trop faible, le robot n'avancera pas assez, ce qui aura tendance à enliser l'algorithme et peut peut-être poser certains problèmes (voire le problème du « voisin privilégié » défini plus loin). Si elle est trop forte, beaucoup de trajectoires risquent d'être rejetées car elles traversent un obstacle, et le robot a plus de chances de tourner en rond à cause de sa roue orientable.

*BETA_POINTmax = 3.0* (borne max de bêta_point dans le cas d'un tirage aléatoire de celui-ci)

###

Recherche de solution :

Notons tout d'abord que nous avons réimplémenté le concept de plus proche voisin. Un point *qrand* est tiré aléatoirement dans l'espace quadridimensionnel *(x, y, theta, beta)*. Un choix important est la métrique utilisée pour déterminer le plus proche voisin :

- Si on donne du poids aux 4 coordonnées, va-t-on réussir à explorer efficacement l'espace (x,y) ?
- Si on donne trop de poids à (x,y), ne risque t-on pas de tirer toujours le même plus proche voisin d'une région donnée ? En effet, la manœuvre d'approche d'un qrand tiré dans cette région peut être telle qu'elle génère des points qui visent à s'approcher de qrand en terme de manœuvre (orientation de thêta par exemple), mais ne l'approche pas en terme de distance cartésienne en (x,y). De sorte, le premier voisin tiré (qu'on peut appeler « voisin privilégié ») restera toujours plus proche de qrand (au sens défini pour notre choix de métrique) que le point nouvellement généré dans l'arbre.

Illustration : pour les points qrand tirés dans cette région, le nœud Vp est un voisin privilégié si les commandes tirées à partir de lui pour s'approcher de cette région visent à aligner le robot vers qrand, mais ne lui laisse pas l'occasion de s'avancer vers ce point.


Ce problème n'en est néanmoins pas forcément un. On peut imaginer qu'il a peu de chances de se produire, et qu'il peut se régler de lui-même au fur et à mesure que l'arbre progresse par ailleurs. Enfin, il n'apparait que si la commande choisie a un risque de générer des manœuvres qui ne jouent que sur bêta et thêta sans jamais approcher x et y de qrand.

---

*« La clé du succès est de réussir à rendre les points qrand tirés ''attracteurs'' pour le robot. »*
*-Nous, Mars 2018*

---

4 approches d'une solution :

- Approche la plus naïve : régler les paramètres influençant le parcours d'une trajectoire, en se concentrant notamment sur l'exploration de l'espace (x,y) (objectif d'exploration : s'éloigner le plus possible SUR LA MAP réelle). Une exploration détaillée de cette solution est faite en dessous via l'***analyse des paramètres***. On peut obtenir des résultats A PRIORI satisfaisants en terme d'exploration de l'espace (x,y) en ajustant les paramètres, une fois la vraie méthode RRT implémentée (avec tirage de qrand et qnear).
  Toutefois le parcours réel du robot entre les différents points sera assez chaotique, la méthode restera loin d'être optimale et aura en particulier du mal à s'appliquer à de grandes maps, ou des maps avec des obstacles non triviaux. Enfin, cette approche s'avère très indéterministe en terme de résultats et donc de temps de calcul nécessaire pour arriver à un résultat.

- Approche seconde, méthode du « plus proche voisin retour » : on ne tire plus une seule commande, mais un panel de commandes qu'on appliquent virtuellement, et on conserve celle qui s'approche le plus du qrand. Le choix de la métrique pour définir « s'approche le plus » aura encore de l'importance.

- La méthode « sugar tracking » : qui consiste à donner une vitesse à un point virtuel (la « sucrerie ») de direction qnear -> qrand, et à générer la loi de commande du robot comme étant un suivi de ce point. C'est la méthode théorique la meilleure à laqulle on peut penser *a priori*. Elle demande d'arriver à calculer les équations de suivi du « sugar ».

- Discriminant de raccordement : n'ajouter un point sur l'arbre que si il est réellement nouveau, afin d'éviter l'enlisement du problème 3 (mais qu'entend-on par « nouveau » encore une fois ? Généralement : « loin des points déjà dans l'arbre ».. Mais selon quelle métrique ?).

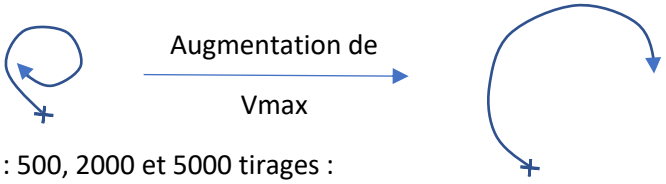
<u>Autre problématique : ce programme peut s'avérer très calculatoire !</u> Il est bon de penser à son optimisation non pas seulement en terme de code, mais aussi sur la méthode d'application des différentes étapes. Par exemple : quand commencer à tester le *merge* (test très calculatoire!) ? Une bonne idée est de commencer à partir d'une taille critique des arbres, mais alors on risque de louper des éventuels *merge* avant.

Si l'algorithme s'avère très performant en terme de ratio « nombre de nœuds dans l'arbre / surface* couverte», ou encore mieux « nombre de tirages / surface couverte », les tentatives de *merges* peuvent être tentées très tôt sans entraver trop la performance (c'est une opération linéaire en n1 ou n2, nombre de nœuds dans chacun des arbres).

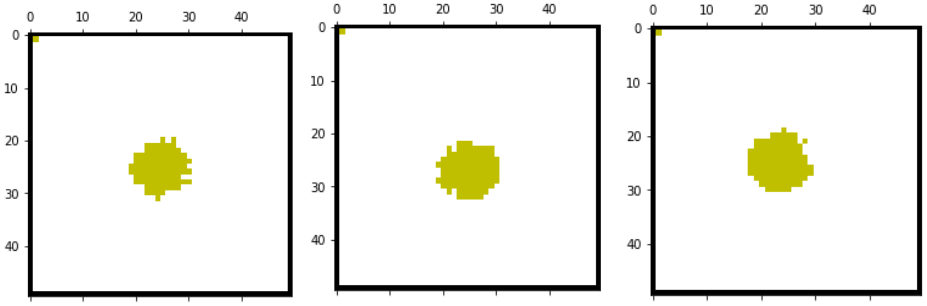
**sens dépendant de la métrique, encore une fois…*

Analyses de l'influence des paramètres : réalisée avec la commande aléatoire en bêta_point et V aléatoires. Bêta_pointmax et Vmax sont ici normalisés par step_size pour éviter l'influence de ce paramètre. Les valeurs des paramètres données ci-après sont en fait le produit de leur valeur par step_size pour garder un ordre d'idée parlant.

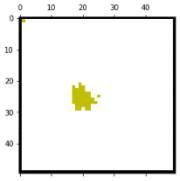
Vmax : Choisir Vmax trop petit va entraîner un phénomène de concentration de l'arbre comme décrit dans la partie des problèmes, car le robot tend à tourner en cercles. Choisir un Vmax trop grand « déroule » les trajectoires mais, d'un autre côté, va empêcher l'évitement d'obstacles.




Vmax=2.0 et bêta_max=5.0 : 500, 2000 et 5000 tirages :



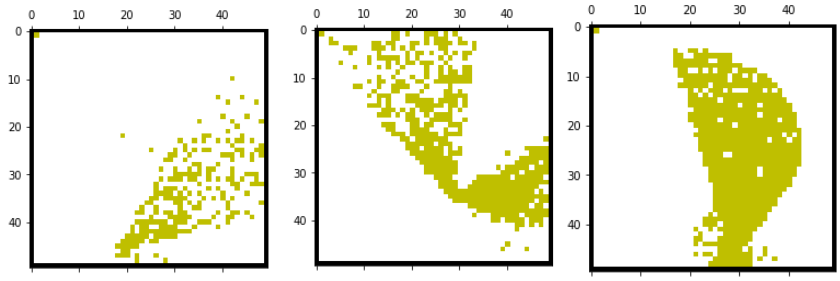
Et avec un bêta_max faible pour éviter de tourner en rond :



Le problème reste sensiblement le même, à cela prêt que le paquet tend à ressembler à une traînée de points car le robot en va avoir des difficultées pour changer de direction.

Bêta_max : Vmax = 15.0

Si bêta_max est trop faible, le robot ne peut pas vraiment faire demi-tour. Il a tendance à aller tout droit. On observe une mauvaise couverture de la map :

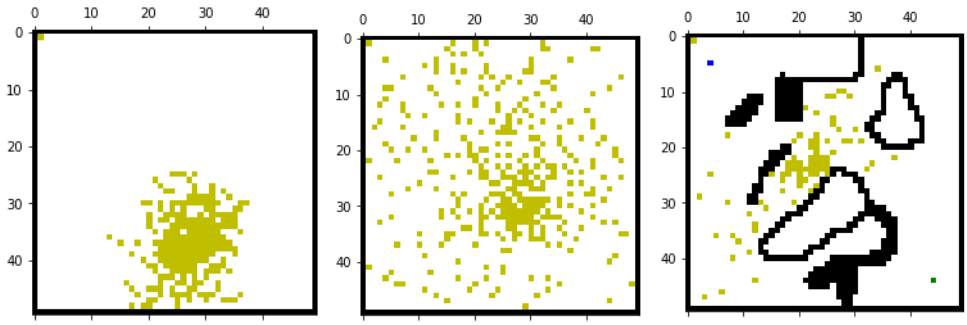


*3 essais à 500, 2000 et 5000 tirages*

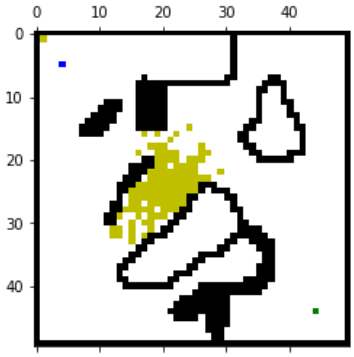
Si bêta_max est trop grand, on observera une plus grande chance de tourner en rond, donc des tirages assez serrés. Cela ne sera observable sur la carte que si Vmax est faible où modérée. La véritée du problème est plus profonde et sera abordée après : d'une part rattrapper le « tournage en rond » par une grande vitesse V n'est qu'une illusion de réussite, tant les risques de collision avec un obstacle

deviennet grands (en effet, la map fini par être explorée car le tournage en rond du robot l'amène tout de même assez loin si il se cumule avec une vitesse linéaire grande, mais alors la distance parcouru est un bout de spirale qui est assuré de rencontrer un obstacle dans une map complexe). D'autre part, que tirer réellement, pour une application concrète, d'un algorithme qui nous proposera un chemin où les parcours d'un nœud à l'autre ont toutes les chances d'être d'effroyables zig-zags ?

Avec beta_point max = 13, en 500 tirages ; avec Vmax= 5.0, puis avec Vmax= 15.0 ; puis avec ajout d'obstacle :

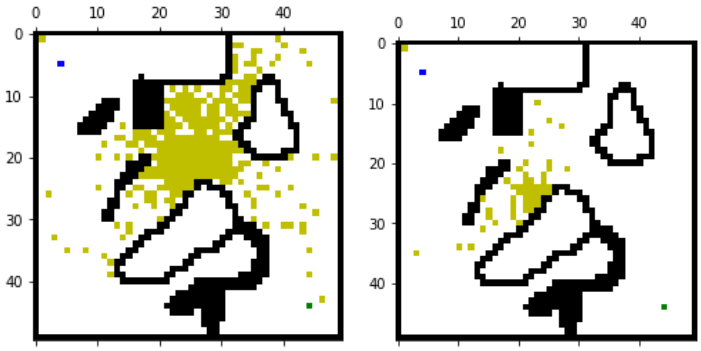



Tandis que les mêmes obstacles, avec une Vmax=5.0 et un beta_point max=2.0, beaucoup plus de nœuds sont ajoutés :




Parcourir la map avec des commandes aléatoires demande donc des valeurs de vitesses maximales élevées pour assurer l'exploration, mais cela implique que la plupart des tirages vont finir dans un obstacle. Cela pourrait être une stratégie viable en terme d'efficacité calculatoire (cf. problème 3) : on compte sur les tirages aléatoires de type « beta_point assez grand et v assez faible » là où le robot doit tourner, et du type inverse quand il faut aller en ligne droite, et les mauvais « mélanges » finissant dans un obstacle ont au moins l'avantage de ne pas encombrer l'arbre, mais en réalité cela expose à une inefficacité, des résultats très aléatoires et même au risque de ne pas réussir du tout à trouver un chemin. Ce laissez-faire est néfaste et relativement contraire au concept même du RRT.

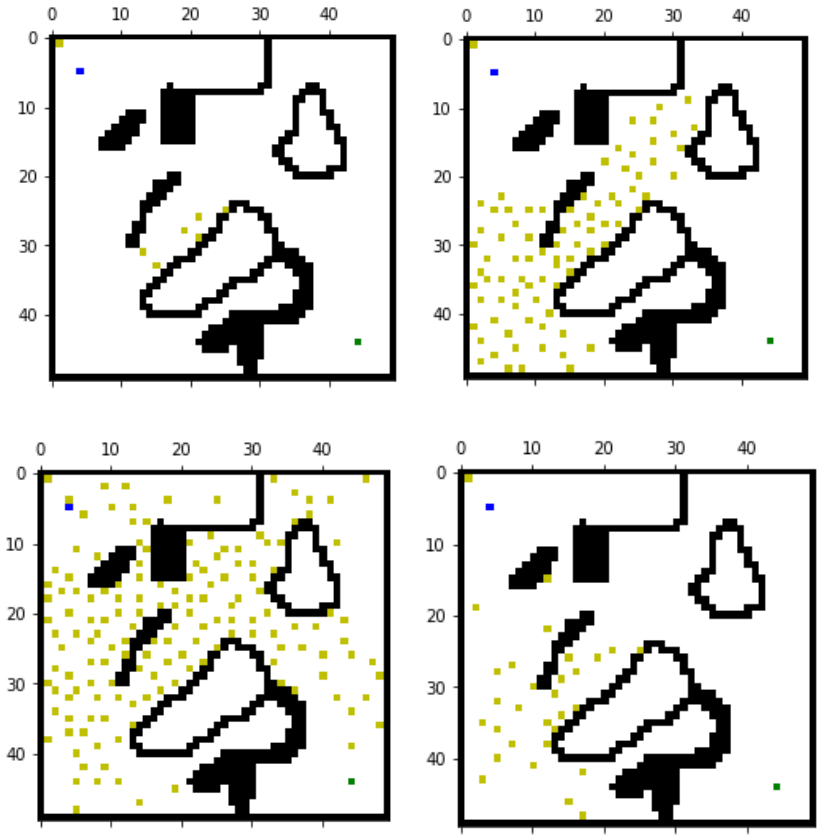
Si par chance on arrive à *merge*, le nombre de points sera probablement réduit (mauvais taux de couverture de la map) ce qui assure que le chemin n'est pas optimal (à cela s'ajoute la remarque précédente sur la difficultée d'utiliser réellement ces parcours très en zig-zags et comportant potentiellement de nombreuses boucles inutiles). Vouloir augmenter les tirages pour compenser amenerait à une incertitude totale sur le temps de calcul, très dépendant du nombre de points de l'arbre.




*Deux essais successifs avec des paramètres identiques, et 2000 tirages.*

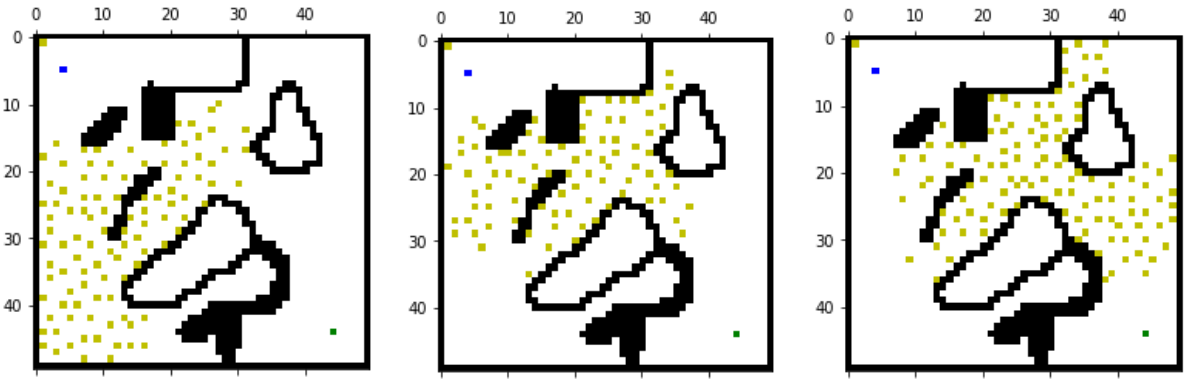
Méthode du « discriminant » : on réalise de nombreux tirages avec des vitesses raisonnables, mais on ne garde que les tirages qui placent le nouveau nœud assez loin des autres nœuds, afin de ne pas surcharger inutilement l'arbre.

Rq : Avec des grandes vitesses (Vmax=15.0, beta_point=10.0) le problème énnoncé ci-dessus apparait à nouveau très clairement. Il faut donc tenter avec un très grand nombre de tirages et des vitesses modérées :




*Deux séries de 2 tests successifs avec des paramètres égaux, pour 5000 tirages (au dessus) et 20000 tirages (en dessous)*

Vmax = 8.0 ; beta_point = 3.0 et 20000 tirages :




Inconvénient : on a encore une couverture médiocre (normal, puisqu'on est obligé de mettre un discriminant assez élevé pour éviter la surcharge de l'arbre vu le nombre de tirages), un grand aléa sur le résultat même avec énormement de tirages, et on n'est toujours pas à l'abris bien sûr, des fantaisies qu'on risque de découvrir au moment de l'utilisation réelle de ces chemins.

La méthode du « plus proche voisin retour » n'est pas présentée en terme de résultats, car elle utilise sensiblement la même idée qu'ici. Il s'agit d'augmenter brutalement le nombre de tentatives, mais de se réserver une sélection de celles qui vont réellement générer un point dans l'arbre.

Considérons :

nbr nœuds / surface couverte = nbr tirages / surface couverte x nbr nœuds / nbr tirages

Qu'on écrit a = b * c

Le ratio en rouge est relativement inchangé car dû à la nature de la génération de commandes comme on l'a vu. Les méthodes « plus proche voisin retour » et « discriminant » baissent le ratio en bleu, afin de baisser globalement celui en vert. La stratégie sous jacente étant en réalité la suivante :

surface_couverte = nbr_tirages /(nbr tirages / surface couverte) = nbr_tirages/b

temps_calculs = Cte * (nbr_noeuds)² = Cte * (nbr nœuds / nbr tirages)² * nbr_tirages²

(= Cte * (nbr nœuds / surface )² * surface*nbr_tirages/b)

Donc : **surface/temps** = 1 / (Cte* (nbr tirages / surface couverte) *(nbr nœuds / nbr tirages)²*nbr_tirages)

= 1/(b*Cte) x 1 / (c² * nbr_tirages) = **$Cte_2$/(c² * nbr_tirages)**

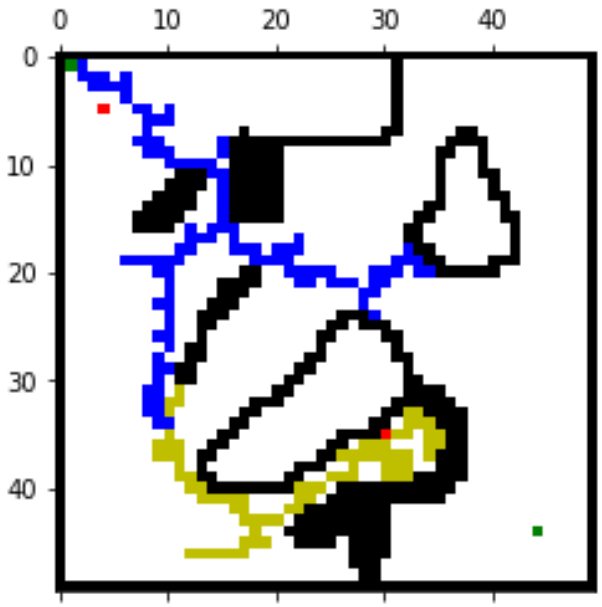
(= b/(Cte * (nbr nœuds / surface )² *nbr_tirages) = **$Cte_3$/(a² * nbr_tirages)** )

Augmenter le nombre de tirages améliore la surface couverte mais réduit le ratio surface/temps si on ne peut pas toucher au ratio bleu (ou vert). Le but de ces stratégies est d'inverser la tendance, c'est-à-dire que le ratio surface/temps s'équilibre quand on augmente le nombre de tentatives ce qui stabilise le temps de calcul (solution théorique au problème 3). En réalité, le surcoût de calcul pour diminuer le ratio bleu en appliquant ces méthodes est aussi à prendre en compte, même s'il est vrai que ces stratégies s'avèrent plutôt bonnes.

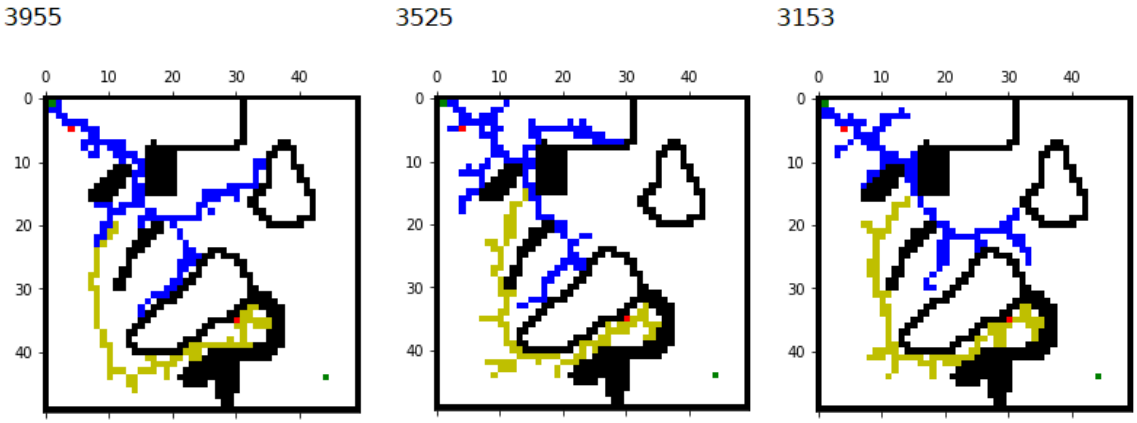
Méthode sugar-tracking : résultats

Les résultats obtenus ici sont bien entendu excellents en comparaison de ceux obtenus auparavant. Cela nous permet notamment de garder des paramètres raisonnables :

Step_size = 0.5, Vmax = 1.0, bêta_pointmax = 2.0. Avec 4000 points :

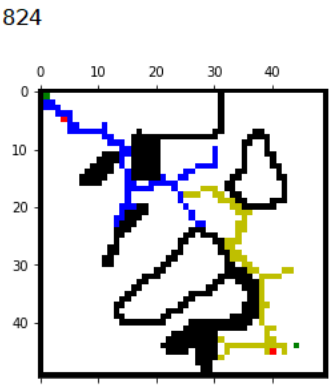



Vu la complexité de la position des points sur cet exemple, le merge se fait environ en 3000 à 4000 tentatives.




*L'algorithme s'arrête ici dès son premier merge. Les arbres ont environ 900 nœuds cumulément, ce qui est un très bon score nœuds créés/nbr tentatives, surtout étant donné la position du point 2 !*

Cet exemple plus basique (à droite) se termine en seulement 824 tentatives, pour un graphe final avec près de 500 nœuds !




On répète également que les chemins tracés dans l'arbre sont tous dans $\mathscr{C}$ [1] et qu'avec cette méthode, ils correspondent à des chemins très réalistes (pas de détours, de tour sur soit-même, etc…) et optimaux pour le robot (pas d'à-coups, de changement brusques de direction, de manœuvres inutiles…).

Liste des changements pour mener à l'algorithme final :

- Une fonction calcule le vecteur vitesse du *sugar* entre qnear et qrand à chaque nouveau tirage.
- Une commande de suivi du *sugar* est appliquée en robot.
- La distance cartésienne (métrique) a été élargie à l'ensemble de l'état, et non plus seulement (x, y).

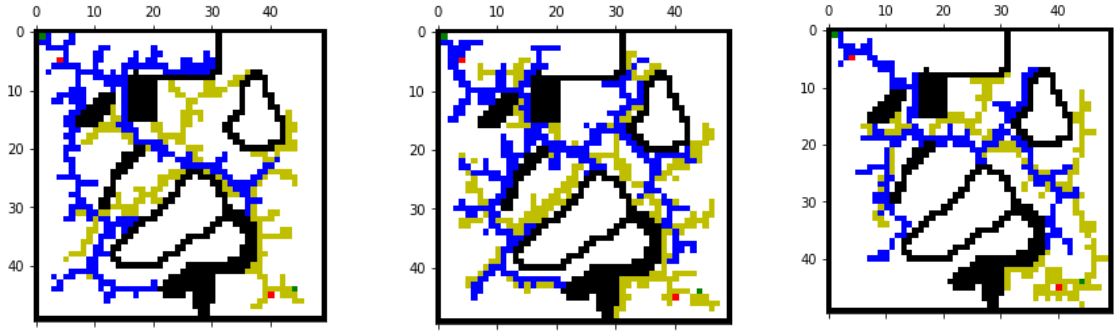
<u>Comparaison des temps de calcul **avec** et **sans** une méthode d'amélioration des ratios :</u>

On veut savoir si une méthode d'amélioration des ratios est encore utile. Certes l'algorithme est bien plus performant, mais le nombre de nœuds reste quand même une légère gêne en terme de temps de calcul.

On va reprendre les points de départ au même endroit que sur la toute dernière figure, mais cette fois on va laisser l'arbre aller au bout de ses tentatives quitte à faire plusieurs merge. On va alors afficher le temps de calcul avec et sans méthode d'amélioration des ratios (ici on choisi la méthode du discriminant).

Avec 4000 tentatives et un merge_threshold de 0.5 :

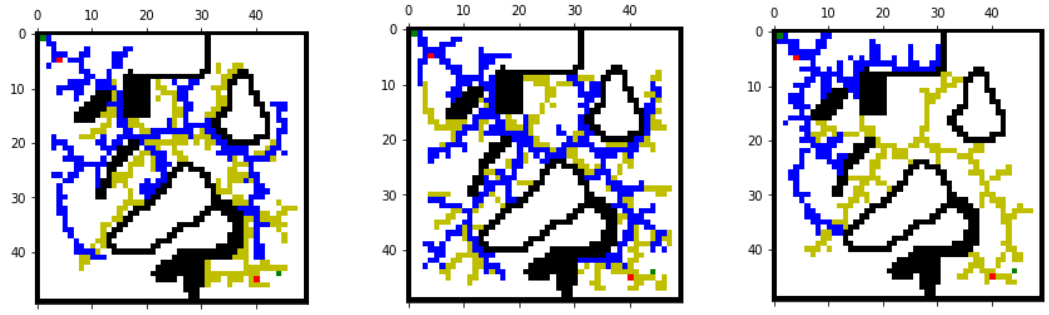
```
Nombre de merging = 459        Nombre de merging = 1183      Nombre de merging = 403
Temps d'execution = 5.515349   Temps d'execution = 7.554845  Temps d'execution = 3.805297
```



La moyenne, sur 20 tests de 2000 tentatives chacun : Tps exection moyen = 1.48 secondes

<u>Même expérience avec la méthode du discriminant :</u> discriminant = 0.15

```
Nombre de merging = 304        Nombre de merging = 694       Nombre de merging = 35
Temps d'execution = 5.178273   Temps d'execution = 6.964396  Temps d'execution = 3.471565
Nbr refus :  897               Nbr refus :  1033             Nbr refus :  880
```



La moyenne, sur 20 tests de 2000 tentatives chacun : Tps exection moyen = 1.12 secondes

Soit un gain de temps de 24%. Ce chiffre encourage plutôt à garder quand-même un script d'amélioration des ratios, mais on remarque tout de même que notre méthode « sugar-tracking » est assez fiable pour pouvoir s'en passer éventuellement.