Algorithmique de la manipulation en robotique

Alexandre Thiault

ECOLE POLYTECHNIQUE
Promotion X2018
THIAULT Alexandre

RAPPORT DE STAGE DE RECHERCHE

Algorithmique de la manipulation en robotique

RAPPORT NON CONFIDENTIEL

Option : **Département d'informatique**
Enseignant référent : **Luca Castelli**
Tuteur de stage : **Florent Lamiraux**

Mars – Juillet 2021
LAAS-CNRS
7, avenue du Colonel Roche BP 54200 31031
Toulouse cedex 4, France

Déclaration d'intégrité relative au plagiat

Je soussigné THIAULT Alexandre certifie sur l'honneur :

1. Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
2. Que je suis l'auteur de ce rapport
3. Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Je déclare que ce travail ne peut être suspecté de plagiat

16/07/2021

Abstract

Dans le contexte du mouvement de robots, la planification de manipulation est le problème consistant à trouver l'enchaînement de mouvements exact qu'un robot devrait faire afin de réaliser des tâches, ou autrement dit la transition entre une situation initiale et une situation finale dans l'environnement du robot. Au sein de l'équipe Gepetto du LAAS-CNRS à Toulouse, je me suis intéressé à ce problème appliqué à des robots humanoïdes manipulant des objets à l'aide de pinces. Le problème a déjà fait l'objet de recherche et différents algorithmes existent pour y répondre. La complexité du problème, venant de la haute dimensionnalité de l'espace des solutions potentielles, oblige à recourir à l'aléatoire, si bien que la majorité des algorithmes actuels ont pour principe commun l'exploration d'un arbre aléatoire. Après une introduction aux définitions et structures nécessaires à sa compréhension, on expliquera le principe d'un nouvel algorithme de planification de manipulation, qui utilise l'aléatoire de manière innovante. Ensuite, ce rapport détaillera l'implémentation que j'ai produite de cet algorithme au sein du logiciel HPP, développé par le LAAS. Enfin, on évoquera les pistes de développements futurs basés sur mes travaux lors de ce stage.

In the context of robot movement, manipulation planning is the problem of finding the exact sequence of movements that a robot should follow in order to achieve tasks, i.e. the transition from an initial state to a goal state in the environment of the robot. Within the Gepetto team at LAAS-CNRS in Toulouse, I focused on this problem applied to humanoid robots manipulating objects with the help of grippers. The problem has already been the object of research and different algorithms exist to address it. The complexity of the problem, coming from the high dimensionality of the potential solutions space, forces us to turn to randomness, so that most of the current algorithms have random tree exploration as a common principle. After an introduction to the definitions and structures necessary for its understanding, we will explain the principle of a new manipulation planning algorithm, which uses randomness in an innovative way. Then, this report will detail the implementation that I produced of this algorithm within the HPP software, developed by the LAAS. Finally, we will discuss the insights for future developments based on my work during this internship.

Je tiens à remercier Joseph Mirabel et Olivier Stasse pour leur accueil lors de ma première semaine en présentiel, et tout particulièrement Florent Lamiraux, qui durant tout le reste du stage en télétravail, a été mon unique interlocuteur pour toutes les difficultés rencontrées.

Table des matières

Contexte du travail

La planification de tâches de manipulation est un des sujets à traiter lorsqu'il s'agit de faire effectuer automatiquement des tâches à un robot. Le but est de trouver une séquence de mouvements intermédiaires permettant de relier une position initiale à une position finale du robot et de son univers, comprenant potentiellement des objets à déplacer ou manipuler. Pour cela le robot peut se déplacer et bouger librement selon tous les degrés de liberté qui lui sont propres, saisir des objets d'une façon appropriée, les déplacer, les déposer au sol, le tout en évitant bien-sûr les collisions. D'autres types de mouvements peuvent être considérés, comme lancer un objet, ou pousser un objet en le faisant glisser sur le sol. Lors de ce stage j'ai été intégré à l'équipe Gepetto du LAAS-CNRS, spécialisée dans les robots anthropomorphes. J'ai donc travaillé sur des modèles de robots humanoïdes possédant des pinces capables de manipulation dite préhensible, c'est-à-dire faite seulement pour saisir des objets qui ne sauraient être lâchés en dehors d'un support stable.

Dans un cas simple, le plan de manipulation peut consister à échanger les positions de deux objets posés au sol en deux points A et B. Toute solution demanderait ainsi de placer l'un des deux objets à un emplacement tiers, le temps de déplacer l'autre objet. Dans un contexte plus pratique, tel que la construction de pièces d'avions Airbus, il peut s'agir de faire trouver à un robot muni d'une perceuse quelle serait une bonne façon de se déplacer en direction d'une pièce à percer, et d'y percer des trous aux endroits demandés.

La construction de pièces d'avions n'est qu'un exemple d'application sur lequel travaillent des collègues qui se penchaient sur d'autres aspects de la robotique, notamment la visualisation. Comparé à d'autres, mon travail s'éloignait un peu plus d'objectifs pratiques prédéterminés. Ma mission avait pour objectif d'être capable de résoudre des problèmes théoriques plus complexes ou de résoudre plus efficacement des problèmes que l'on sait déjà résoudre de manière sous optimale. En effet, la planification de manipulation recourt rapidement à des techniques d'exploration aléatoire des mouvements possibles dès lors que les quelques techniques déterministes ont échoué.

Durant ce stage, je me suis penché spécifiquement sur un nouvel algorithme de planification de manipulation. Avant d'entrer plus dans les détails de mon travail, il nous faut introduire l'état de l'art.

Etat de l'art

La planification de mouvement a été largement étudiée ces deux dernières décennies, y compris dans des cadres plus généraux ou différents de ceux qui intéressaient mon équipe. Le problème général de validation continue de non-collision le long d'un chemin entre deux configurations est très bien résolu et a fait l'objet de nombreuses publications, notamment par le LAAS lui-même [1]. Il est connu depuis longtemps que la planification de mouvements de systèmes à haute dimension tels que les robots humanoïdes est complexe et nécessite le recours à des échantillonnages aléatoires de configurations. De nombreux algorithmes d'exploration basée sur des échantillonnages aléatoires existent, du plus simple et général, le *RRT* [2], à des plus compliqués mais plus efficaces comme le *RMR** [3], et même des algorithmes spécialisés pour le cas de la planification de manipulation [4]. Les cas de manipulation non préhensile d'objets qui pourraient être lancés [5] ou poussés [6] ont été également décrits. Enfin, la manipulation préhensile est depuis longtemps le sujet d'étude d'une équipe dédiée du LAAS, qui a produit en particulier le logiciel *Humanoid Path Planner* (HPP) pour le résoudre [7].

Définitions

Configurations

La position exacte d'un robot humanoïde (ou plusieurs) et des objets avec lesquels il peut interagir à un instant donné est appelée **configuration**. Une configuration est décrite par un ensemble de paramètres décrivant les différents degrés de liberté des articulations du (des) robot(s) ainsi que des objets avec lesquels une interaction est possible. Ces paramètres sont soit des coordonnées de translation dans R^3, des quaternions décrivant une rotation dans *SO(2)*, ou une rotation 3D décrite par un élément de *SO(3)*. **L'espace des configurations** est alors le produit cartésien des espaces associés à chaque degré de liberté. Algorithmiquement, les configurations sont représentées par un vecteur de nombres réels, concaténation des vecteurs d'éléments des espaces représentant chaque degré de liberté. Cela facilite l'application de transformations comme multiplication par une matrice ad hoc.

Une configuration peut être **valide** si elle respecte certaines conditions. Ces conditions peuvent être l'absence de collisions, ou le fait que les transformations décrivant des articulations restent dans des bornes données, par exemple un genou qui ne doit pas tourner du mauvais sens. Ou encore le fait qu'un objet ne se trouve pas en l'air mais, soit dans une main du robot, soit collé au sol. En effet on demande au robot de ne lâcher un objet qu'une fois placé au sol. En toute généralité on parlera de **gripper** plutôt que de main, l'organe attrapant pouvant être plus original.

L'évolution continue des configurations du robot en fonction du temps suit un **chemin**. Un chemin sera valide si on peut vérifier de manière continue que toutes les configurations intermédiaires de ce chemin du début à la fin sont valides.

Contraintes

Pour assurer la cohérence temporelle d'un chemin, et éviter des aberrations telles que la téléportation d'objets, certaines configurations ou portions de chemin peuvent avoir à vérifier des ensembles d'équations, appelés **contraintes**. On leur donne en général un nom intelligible.

Les contraintes s'expriment comme une équation de type $f(q) = b$ où q est le vecteur de configuration, b est appelé le **second membre**, et f est une fonction différentiable, pas forcément linéaire. Par exemple, pour une balle de rayon r repérée par un vecteur de configuration de longueur 7, soit un vecteur 3D de translation (t_1, t_2, t_3) et un quaternion de rotation (q_0, q_1, q_2, q_3), le placement peut s'écrire tout simplement $t_3=r$, ou $(t_3, q_0, q_1, q_2, q_3) = (r, 1, 0, 0, 0)$ si l'on veut imposer une orientation au sol.

On distinguera des contraintes d'état, dites **non paramétrables**, qui doivent être respectées à un instant t et qui possèdent un second membre nul, et des contraintes de chemin, dites **complémentaires** ou **paramétrables**, qui ne fournissent pas un second membre à atteindre mais demandent de conserver le même second membre le long du chemin.

Des exemples de contraintes non paramétrables sont :

- *placement* : pour un objet, il s'agit d'être à une position et dans une orientation donnée. Cette contrainte devrait être vérifiée pour tout objet dès lors qu'il n'est pas dans le gripper d'un robot.
- *grasp* : pour un couple objet-gripper, cela requiert que l'objet se trouve exactement dans le gripper du robot s'il est supposé le tenir actuellement. Grasp est en quelque sorte l'opposé de placement.
- *pregrasp* : avant de saisir un objet au sol (ou après l'avoir posé), on peut demander au robot d'aligner son gripper verticalement à une petite distance donnée au-dessus de l'objet pour faciliter sa préhension.
- *preplacement* : avant de poser un objet (resp. après l'avoir saisi), on peut demander au robot d'aligner son gripper, et donc l'objet tenu, verticalement à une petite distance donnée au-dessus du sol. Cela peut servir lorsqu'on doit poser ou sortir l'objet d'une boîte pour éviter de heurter les murs de la boîte avec le gripper.

Voici quelques exemples de contraintes paramétrables :

- *placement/complement* : pour un objet, il s'agit de rester à une même position et orientation. Peu importe lesquelles.
- *grasp/complement* : pour un objet, il s'agit de suivre en temps réel les mouvements du gripper qui le tient actuellement.
- *vertical* : pour un objet en déplacement, donc qui suit déjà une contrainte grasp/complement, il s'agit de spécifier que la trajectoire de l'objet doit en plus être verticale par rapport au sol.

Etats

En synthétisant une configuration par seulement le caractère tenu ou non de chaque objet, et si tenu, par quel gripper, on groupe les configurations en **états**. Certains états sont irréalisables. Par exemple, on ne peut pas demander à un gripper de tenir plusieurs objets à la fois. On appelle notamment **free** l'état le plus simple dans lequel aucun gripper ne tient d'objet. Dans le cas d'un robot à 2 grippers et 2 objets, il y a donc 7 types d'états : le free, 4 états où un objet est tenu, 2 états où les deux objets sont tenus. Un état est donc défini par un ensemble de contraintes non-paramétrables.

Le long d'un chemin, les configurations d'un robot évoluent dans un état, et parfois changent d'état.

Evoluer dans un état signifie qu'il existe un chemin reliant deux configurations pendant lequel toutes les configurations intermédiaires sont dans le même état. Deux configurations du même état ne sont pas forcément reliables par un chemin restant dans le même état. Par exemple pour qu'un objet au sol se déplace, on ne peut pas rester dans l'état *free*, il faudra passer par un état où l'objet est tenu, le temps de le déplacer.

Changer d'état nécessite de passer par une configuration à l'intersection de deux états. Par exemple, pour saisir un objet, on veut passer d'un état free à un état où le gripper tient l'objet. A la frontière se trouve une configuration où le gripper tient l'objet qui est encore posé au sol : cette configuration est à l'intersection des deux états. Deux états ne possèdent pas forcément une intersection et peuvent nécessiter de passer par plusieurs états intermédiaires. Cela permet de définir un **graphe d'états**, dans lequel on relie des états qui ont une intersection, c'est-à-dire qui autorisent un passage direct de l'un à l'autre. De plus, on ajoute dans un graphe d'état des arêtes de boucle autour de chaque état pour représenter la possibilité de demeurer dans un même état.

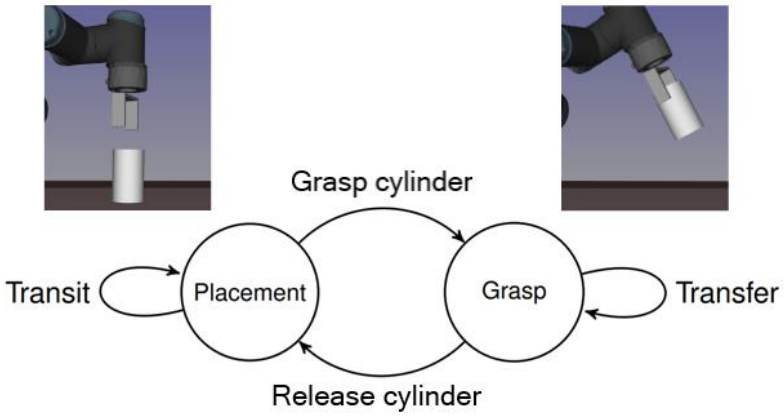

Figure 1: Graphe d'état simpliste associé à un univers constitué d'un objet et d'un robot à un gripper capable de saisir l'objet. L'objet est soit tenu soit posé au sol, et on peut effectivement changer d'état dans les deux sens.

Les arêtes de boucle ne sont pas uniquement visuelles. On peut les interpréter en leur faisant porter des contraintes propres, comme les états. En faisant l'amalgame entre un chemin de configurations et un chemin dans le graphe suivant ses arêtes, on dira que l'arête *Transit* de la figure 1 porte la contrainte *placement/complement*, non contenue dans l'état *Placement* défini uniquement par la contrainte *placement*. Ainsi l'arête *Transit* illustre le fait qu'un chemin qui ne quitte pas l'état *Placement* est contraint par *placement/complement*, qui empêche un objet posé de se déplacer tout seul.

L'ensemble des configurations **accessibles** depuis une configuration initiale tout **en suivant une arête** de boucle donnée peut ainsi être visualisé comme une sous-variété au sein de l'espace des configurations, en tant qu'espace des solutions des équations des contraintes contenues dans l'arête. Par abus de langage on peut aussi parler de configurations accessibles en restant dans un même état pour signifier qu'on suit l'arête qui boucle sur cet état (par exemple *Transit* et *Transfer* dans la figure 1).

Feuilles

Si deux configurations du même état n'ont pas tous leurs seconds membres égaux selon chaque contrainte paramétrable contenue dans l'arête de boucle associée à l'état, elles ne sont pas reliables dans le même état. En effet, par définition, changer un second membre violerait sa contrainte et donc induirait un changement d'état. En conservant les mêmes contraintes mais en changeant leur second membre, on obtient une nouvelle sous-variété « parallèle » à la première. On parlera de **feuilles** pour illustrer ces sous-variétés. Chacune constitue une proportion de l'espace des configurations. Cette proportion peut être nulle mais l'union des feuilles donne

obligatoirement un état entier. Les seconds membres ne sont pas le seul problème à l'accessibilité mutuelle de deux configurations : un obstacle physique et les collisions qui en découlent peuvent par exemple séparer une sous variété de configurations possibles en plusieurs.

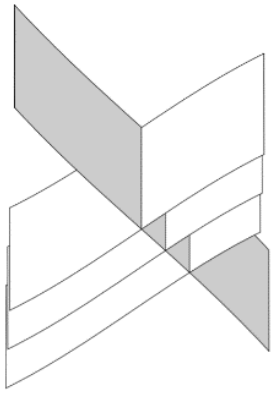

Figure 2: Exemple de visualisation possible pour la notion de feuilles : en blanc, trois feuilles définies par une même contrainte mais trois seconds membres différents, en gris, une feuille définie par une autre contrainte, possédant une intersection avec les trois autres.

Le problème de planification de tâches de manipulation consiste donc à explorer un espace de configurations qui s'arrange en union d'états, états qui sont eux-mêmes une union disjointe de feuilles définies par des contraintes et leurs seconds membres. La suite d'état à suivre pour trouver une solution est a priori inconnue et n'est pas unique. Au sein d'une feuille même, une résolution « en ligne droite » n'est pas toujours possible. Des détours peuvent être nécessaires pour contourner un obstacle afin d'éviter des collisions.

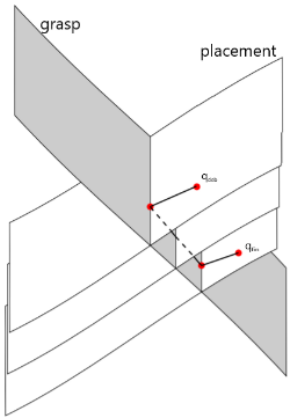

Figure 3: Représentation sur feuilles d'un chemin possible reliant deux configurations dans le même état (placement) mais nécessitant de passer par l'intermédiaire d'un autre état (grasp) pour changer de feuille. Le problème de déplacement d'un objet au sol vers un second emplacement peut donner lieu à ce genre de solution.

Motivations pour un nouvel algorithme de planification de tâches

Lors de mon stage, ma mission est d'enrichir le logiciel HPP. Le logiciel a déjà atteint un stade avancé et le LAAS cherche en ce moment à l'optimiser.

Plus précisément, en réduisant à ce qui m'aura été utile, HPP est déjà capable de laisser un utilisateur définir un problème, définir des contraintes, définir des états, spécifier les transitions d'états autorisées, choisir des configurations initiales et finales, et appliquer un des algorithmes existants pour trouver des chemins entre deux configurations. Ces algorithmes utilisent les différents modules de HPP, par exemple échantillonner une configuration, vérifier ou appliquer des contraintes sur un chemin, tester les collisions ponctuellement pour une configuration et de manière continue pour un chemin. Toutes ces fonctionnalités bas niveau sont déjà au point et il s'agit de trouver de meilleurs algorithmes. On cherche une balance entre les choix aléatoires, lorsqu'il y a trop de possibilités, et les choix déterministes, lorsqu'il y a un avantage clair à guider la résolution.

Tous les algorithmes de planification de manipulation actuellement disponibles dans HPP sont basés sur un arbre d'exploration aléatoire de l'espace des configurations, mais aucun ne guide cette exploration par un plan précis quant aux états par lesquels passer. L'exploration aléatoire basée sur une variante d'un *RRT* fait explorer des configurations proches de configurations déjà explorées, et les changements d'états y sont décidés de manière aléatoire. Chaque étape consiste à ajouter une nouvelle branche à l'arbre. Pour cela, on part d'une configuration q_0 dans l'arbre et on cherche une nouvelle configuration accessible par un chemin direct depuis q_0. Si on décide de changer d'état vers un état voisin, on projette une configuration aléatoire q_{rand} vers une configuration q_{proj} dans cet état voisin, et on tente de créer une configuration q_{inter} à l'intersection des deux états qui servirait de point de passage entre q_0 et q_{proj}. Mais ceci n'est pas toujours réalisable.

Une idée nouvelle, pour accélérer la recherche de chemin solution et limiter les explorations inutiles de branches infructueuses, est de décomposer le problème en définissant à l'avance une suite d'états intermédiaires par lesquels le chemin solution doit passer. Mais comme on ne connaît pas a priori de telle suite solution, on testera de nombreuses candidates.

Le premier avantage de cette idée suppose qu'on arrive à donner un sens aux arêtes de transition du graphe d'état, de la même façon qu'on a su attacher des contraintes aux arêtes de boucle. Si cela est fait, alors une suite imposée d'états décomposera naturellement un éventuel chemin solution en portions dont les contraintes sont connues, en tant que contraintes de l'arête du graphe d'état reliant deux états adjacents. Une analyse de ces contraintes permettra d'éliminer plus ou moins tôt des candidats. Ce sera autant de temps gagné par rapport à un algorithme

classique qui ne connait que les *timeout* ou dépassements de seuils arbitraires similaires pour annoncer un échec.

Le second avantage est qu'on divise ainsi la résolution du problème en deux sous-problèmes plus simples, ce qui est très souvent une bonne chose en algorithmique :

1. chercher une suite de configurations suivant les étapes de la suite d'états
2. à partir de la suite de configurations ainsi générée, relier les paires de configurations consécutives par un chemin continu.

Calculer une suite de configurations ne demande aucunement de calculer des interpolations continues entre deux configurations, et requiert une quantité minimale de tests de collisions par opposition à une validation continue de chemin, plus coûteuse en temps de calcul. L'intérêt de relier ensuite les paires de configurations consécutives de cette suite est que deux configurations consécutives sont sur une même feuille et on peut donc espérer les relier sans changer d'état, réduisant considérablement la dimension de l'espace de configurations à explorer pour les algorithmes d'exploration aléatoire appelés pour relier ces paires de configurations.

D'où l'intérêt de rechercher un algorithme suivant ce plan général où le choix de la suite d'état précède toute résolution de configuration ou chemin. Pour cela on propose un algorithme qui itère par ordre de longueur sur toutes les suites d'états possibles pour aller de l'état initial à l'état final, puis qui cherche pour chaque suite générée à résoudre un chemin qui suive cette suite. Cet ordre maximisera les chances de trouver en premier une solution plus courte. Ceci contraste avec les algorithmes classiques qui agrandissent l'arbre d'exploration pas à pas à partir de la configuration initiale en usant de différentes heuristiques pour faire croître les chances que des branches de l'arbre atteignent la configuration finale.

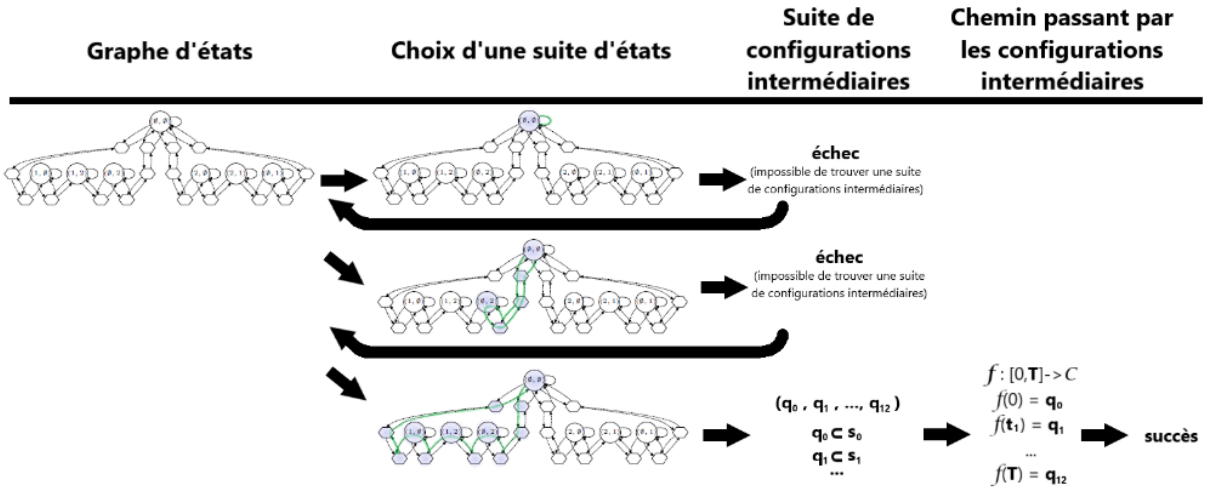

Figure 4: Illustration des grandes étapes évoquées plus haut de l'algorithme proposé.

Dans le cadre de ce nouvel algorithme nous devons donc associer des contraintes à toutes les arêtes. Pour cela nous avons besoin de considérer un raffinement du graphe d'état qui discrimine toutes les étapes de changements de contraintes. Ces étapes incluent les intersections d'états adjacents comme point de passage. En effet, en ces intersections, les contraintes des deux états doivent être respectées en même temps, et c'est de part et d'autre que le changement d'ensemble de contrainte à respecter s'opère. Ainsi le déplacement d'un objet pourrait passer par des états *free* → *[free ∩ arm1_grasps_obj1]* → *arm1_grasps_obj1* → *[free ∩ arm1_grasps_obj1]* → *free*. Seuls les 1er, 3e et 5e sont de vrais états au sens précédemment évoqué. Les autres sont appelés **états de points de cheminement (EPC)**.

Pour des raisons pratiques, et dans le but de minimiser le taux d'échecs dus aux collisions, on intercale entre ces états et EPC d'autres EPC où sont intégrées des contraintes de *pregrasp* et *preplacement*, quand cela a du sens. Cela permet également de donner plus de contrôle à l'utilisateur sur le chemin qu'il veut obtenir : dans le problème de la figure 5 on peut vouloir forcer la verticalité de la trajectoire d'objets à l'approche du gripper ou du sol en ajoutant la contrainte *vertical* aux arêtes entrant et sortant de l'EPC « *grasp-placement* ».

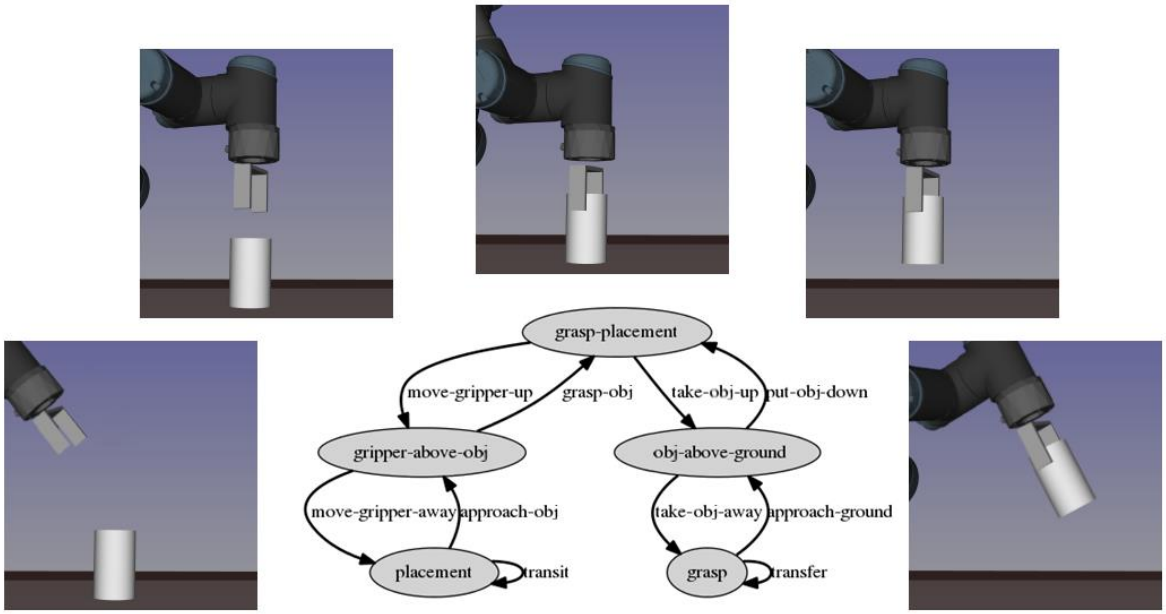

Figure 5: Le graphe complet correspondant au problème simpliste d'un objet et d'un robot à un gripper. En plus des états placement et grasp, on ajoute l'EPC d'intersection « grasp-placement » où les contraintes des deux états sont à respecter en même temps, ainsi que le pregrasp « gripper-above-obj » et le preplacement « obj-above-ground » pour limiter les collisions entre gripper et objet à l'approche du grasp-placement.

Mise en œuvre

Prérequis

On suppose que le graphe d'états est connu à l'avance : les états possibles (EPC compris), ainsi que les arêtes les reliant et donc les contraintes associées à chaque arête et chaque état. C'est à l'utilisateur de définir les mouvements possibles pour son robot et de définir les interactions que son robot peut avoir avec les objets de son environnement.

On peut donc naturellement parler par la suite d'une liste d'états, ordonnée arbitrairement, ainsi que de la liste de contraintes du graphe, union de l'ensemble des contraintes contenues dans chaque arête et état du graphe.

Génération des suites d'états par ordre de longueur

Cette partie étant loin d'être la plus coûteuse en ressources et en temps, plusieurs solutions peuvent être implémentées : c'est un exercice de parcours de graphe relativement classique. La version implémentée dans HPP a été optimisée et saurait difficilement être décrite en détail mais utilise le principe suivant basé sur la structure de *queue*. On part d'une *queue* contenant une seule suite d'états, la suite d'états de longueur 1 contenant seulement s_0 l'état de la configuration initiale. On parcourt la *queue* et pour chaque chemin s_0-s_1-s_2-…-s_n rencontré, on crée des nouveaux chemins de la forme s_0-s_1- …-s_n-s_* pour tous les états voisins s_* de s_n, et on ajoute ces nouveaux chemins au bout de la *queue*. Lorsque s_n est l'état de la configuration finale, on envoie le chemin obtenu à la prochaine étape de l'algorithme. Ce parcours de queue s'arrête dans deux cas :

- Lorsque la suite de l'algorithme réussit à trouver une solution valide à partir d'un chemin s_0-s_1-…-s_n rencontré.
- Quand n dépasse un certain seuil fixé à l'avance, dans ce cas on considère que l'algorithme a échoué à trouver une solution suffisamment « courte », au sens du nombre de changements d'états à effectuer.

Par la suite, pour alléger les écritures, on notera q_0 la configuration initiale, q_n la configuration finale, et pour k entre 1 et n-1, q_k désignera une configuration, a priori inconnue mais que l'on cherchera à calculer, point de passage de la solution dans l'état s_k. Trouver une telle suite $(q_i)_i$ est un objectif intermédiaire de cet algorithme.

Analyse préliminaire des contraintes

Après avoir reçu une suite d'états à suivre de la part de l'étape précédente de l'algorithme, on peut repérer les contraintes qui doivent être conservées depuis la configuration q_0, ou à partir d'une certaine étape jusqu'à la configuration q_n, ou juste le long d'un tronçon de la solution.

Ainsi, si une même contrainte c se retrouve dans les k premières arêtes, on possède déjà une équation pour contraindre la configuration q_k. Si une contrainte c n'est pas valable dès la première arête mais est valable sur les k dernières arêtes, on a une équation sur la configuration q_{n-k}. Si une contrainte est vérifiée de l'arête k_1 à l'arête k_2-1, on sait que la configuration q_{k1} suffit à contraindre les configurations intermédiaires jusqu'à q_{k2}. Enfin, si une contrainte c est dans toutes les arêtes, on obtient une équation reliant q_0 à q_n. Si cette dernière crée un conflit, on en déduit immédiatement que la suite d'état ne peut fournir de solution et on retourne à l'étape précédente de l'algorithme pour chercher le prochain candidat de suite d'états.

Toutes ces informations sont enregistrées dans un tableau T bidimensionnel qui, pour chaque contrainte de la liste de contraintes du graphe, et chaque point de passage indexé par k entre 1 et n-1, donne l'une des 4 possibilités suivantes :

- ABSENT pour une contrainte ne restreignant pas cette étape.
- EQUAL_TO_INIT pour une contrainte présente depuis la première arête.
- EQUAL_TO_GOAL pour une contrainte présente jusqu'à la fin.
- EQUAL_TO_PREVIOUS pour les autres cas.

On retient alors les contraintes pertinentes à la résolution de q_k pour chaque k, à savoir les contraintes de l'arête k repérées par un statut EQUAL_TO_INIT ou EQUAL_TO_PREVIOUS dans le tableau T, et les contraintes de l'arête k+1 repérées par un statut EQUAL_TO_GOAL.

Résolution d'une suite de configurations

Cette partie, le cœur de l'algorithme, consiste à obtenir pour chaque état intermédiaire s_k pour k allant de 1 à n-1, une configuration q_k telle que :

- elle soit dans s_k, c'est-à-dire que q_k doit respecter les contraintes de l'état s_k,
- les seconds membres de q_k pour chaque contrainte d'arête de l'arête liant s_{k-1} à s_k soient égaux aux seconds membres pour ces contraintes de q_{k-1}.

En effet, pour effectivement bénéficier de la possibilité de relier des paires (q_{k-1},q_k) par un chemin continu restant dans une même arête à la prochaine étape, il faut

s'assurer que q_{k-1} et q_k soient dans la même feuille de cette arête (cf Figure 3 : ici ce n'est pas le cas et il faut changer d'état).

Partant de q_0 déjà connu, on essaye de déterminer des q_i pour tout i de 1 à n-1, un par un récursivement, avec possibilité de *backtrack* si un échec est rencontré pour un certain k, afin d'éviter de tout recommencer alors que le début de la liste de q_i pourrait être conservé. On entretiendra un compteur d'échecs à chaque étape pour déterminer à quel moment reprendre à la même étape et à quels autres moments revenir en arrière et de combien.

Pour résoudre q_k à partir de q_{k-1}, on ajoute les équations associées à chaque contrainte pertinente à un *solver* S_k. Les *solvers* sont une structure de HPP destinée à mettre en forme les systèmes d'équation et à projeter des configurations sur l'espace des solutions de ces équations, que je n'ai pas eue à coder. Les seconds membres de ces équations sont ceux de q_0, q_{k-1} ou q_n, selon le statut dans T de la contrainte.

Lors d'un premier essai pour résoudre q_k, on tente d'appliquer le *solver* S_k à q_{k-1} : on veut projeter q_{k-1} sur l'espace des solutions que q_k devrait respecter. Si cela ne marche pas, ou si la configuration obtenue n'est pas valide (au sens des collisions notamment), on recommence quelques fois mais en appliquant S_k à des configurations aléatoires. L'intérêt de commencer par projeter q_{k-1} plutôt qu'une configuration aléatoire est qu'on peut ainsi espérer que la solution trouvée q_k soit proche de q_{k-1}. Au bout d'un certain nombre d'échecs, on remonte un cran en arrière : on tente de résoudre q_{k-1} en fonction de q_{k-2} après avoir incrémenté d'1 le compteur d'échecs de résolution de q_{k-1} et remis à zéro celui de q_k.

Cette étape de l'algorithme termine à un de ces deux moments :

- Lorsque q_{n-1} est résolu. Dans ce premier cas c'est une réussite et on passe la suite de configurations ainsi résolue à l'étape suivante de l'algorithme.
- Lorsque le compteur d'échecs de résolution de q_1 dépasse une limite fixée. On retourne à la première étape calculer le prochain candidat de suite d'état.

Dans certains problèmes, on peut rencontrer un cas où, quel que soit le début de la résolution, toutes les tentatives de résoudre q_k échoueront au même k proche de n, entrainant un nombre exponentiel en k de calculs de résolution des configurations avant l'étape k jusqu'à épuiser tous les compteurs d'échecs. Pour se prémunir de trop longs calculs dans ce cas, on met à jour à chaque échec un compteur global d'échecs. Quand ce compteur global dépasse un certain seuil dépendant de n, on réinitialise l'intégralité de la liste et on incrémente directement le compteur d'échecs de q_1.

Avec cet autre critère d'échec et après de nombreux tests il m'a été possible de déterminer des seuils où fixer le maximum des compteurs d'échecs de façon à conduire à des faibles taux de rejets de faux négatifs (des candidats de suites d'états

pour lesquelles il aurait été possible de trouver une suite de configuration suivant la suite d'état mais où trop d'erreurs de projection ont fait échouer le candidat) tout en gardant un temps d'exécution satisfaisant : négligeable devant d'autres parties de l'algorithme.

Analyse préliminaire des collisions

Cette étape est facultative et constitue une optimisation permettant d'éliminer rapidement des candidats de suite d'états qui échoueraient forcément à l'étape de résolution, après des calculs plus lourds. Elle survient avant l'étape décrite précédemment de résolution d'une suite de configurations, mais la seconde justifiant la première, il était nécessaire de les introduire en ordre inverse.

L'analyse préliminaire de collisions cherche s'il existe un i entre 1 et n-1 où peu importe q_{i-1}, il y aura forcément une collision dans q_i. Si un tel événement se produit, on peut éliminer le mauvais candidat en temps linéaire en n, sans avoir résolu une liste de configurations intermédiaires $(q_i)_i$, donc sans s'être engagé dans le *backtracking*. Pour cela, on exploite le caractère prévisible des équations de contraintes aux statuts EQUAL_TO_INIT et EQUAL_TO_GOAL, par opposition aux contraintes au statut EQUAL_TO_PREVIOUS qui ne sont utilisables que si on résout la suite de $(q_i)_i$ en entier. Dans certains cas, ces équations prévisibles déterminent entièrement q_i ou une partie de la configuration. S'il y a déjà une collision parmi ce qui a pu être déterminé, le candidat est éliminé.

Par exemple les équations prévisibles peuvent suffire à déterminer les positions des objets tandis que la position du robot peut rester inconnue. Si deux objets dont la position est déterminée sont en collision, alors peu importe la position du robot, il y aura toujours une collision.

Pour pouvoir réaliser des tests de collision dans une configuration partielle, HPP dispose d'une fonctionnalité pour savoir quelles parties des robots et des objets sont contraintes par rapport aux autres. On projette donc une configuration aléatoire sur l'espace des solutions des équations prévisibles et on teste si une collision est détectée entre deux parties de l'univers (objet ou articulation d'un robot…) contraintes l'une par rapport à l'autre. Dans l'exemple précédent on ignorerait une collision entre le robot et un objet si le robot n'est pas contraint avec l'objet : le robot s'est trouvé en collision avec l'objet par hasard, pas à cause d'une contrainte, et on compte sur le fait que cette collision n'est pas inévitable.

En pratique on peut observer que l'ajout de cette étape de vérification peut être d'une aide cruciale pour diviser le temps d'exécution, lorsque de telles analyses démontrent en effet des collisions prévisibles. En revanche pour d'autres problèmes

où ce genre de démonstration n'aboutit jamais, faire ce test est une pure perte de temps. De plus il est nécessaire de refaire la projection plusieurs fois à chaque étape, jusqu'à ce que la projection n'échoue pas, et la projection sur un ensemble réduit d'équations semble en effet échouer assez souvent. Lorsqu'elle échoue un certain nombre de fois sans jamais avoir réussi, on passe simplement à l'étape suivante pour ne pas consacrer trop de temps à un test facultatif.

Construction d'un chemin entre deux configurations sur une même feuille

Une fois la suite $(q_i)_i$ construite, le problème est réduit à n applications d'un algorithme de construction de chemin entre deux configurations d'une même feuille, où les contraintes à respecter sont connues et valable sur tout le chemin à construire. Cette tâche simple est déjà aisément résolue par les algorithmes de manipulation préexistants de HPP, basés sur de l'exploration aléatoire, il suffit de leur donner la liste de contraintes à respecter, et lorsqu'une solution existe, elle est trouvée rapidement, d'autant plus vite que les configurations à relier sont proches.

Toutefois le succès n'est pas garanti à cette étape. Des obstacles physiques peuvent toujours se trouver sur le chemin direct et demander de passer par un détour. Cela mène parfois à dépasser un seuil de *timeout*. Il est même possible qu'il n'existe pas de chemin ne quittant pas la feuille lorsqu'un obstacle physique est insurmontable sans changer d'état.

En cas d'échec sur une seule des n parties du chemin, la suite de configurations déterminée à la précédente étape doit être abandonnée mais pas forcément la liste d'états dans laquelle ces configurations ont été trouvées. L'aléatoire jouant toujours dans la détermination de la suite de configurations, l'algorithme revient seulement au début de l'étape de la suite de configurations. Il n'est alors pas nécessaire de refaire l'analyse préliminaire de collisions car elle ne prend en compte que des collisions déterministes, indépendantes de tout choix aléatoire.

En cas de succès, les n chemins trouvés sont concaténés et constituent la solution au problème de manipulation que renvoie l'algorithme.

Si après plusieurs échecs de cette dernière étape sur la même liste d'états, on ne trouve toujours pas de solution, on juge la liste d'états mauvais candidat et on revient à la première étape pour le prochain candidat. En effet il peut subsister des cas où peu importe les choix faits, une collision arrivera inévitablement entre deux configurations intermédiaires. L'étape d'analyse de collisions n'analysant que ces configurations intermédiaires, on ne pouvait pas le prévoir à l'avance avec cet algorithme.

Développements futurs

Pour chaque nouvel algorithme de planification de manipulation, il est intéressant de démontrer sa complétude, c'est-à-dire se demander si, en laissant un temps infini à l'algorithme sans le limiter par un *timeout* ou un nombre maximal d'itérations autorisées, la probabilité que l'algorithme trouve une solution, si elle existe, tende vers 1. La complétude a surtout un intérêt théorique. En pratique on sacrifiera souvent cette qualité si cela permet d'intégrer des optimisations qui accélèrent l'algorithme. L'objectif de la complétude a été évoqué à la fin de mon stage et il a été décidé de le garder pour une autre occasion étant donné le peu de temps qui me restait.

Les tests de mon code ont permis de découvrir des comportements inattendus dans le reste du logiciel HPP en utilisant des modules et fonctions dans des cas de figure jamais explorés auparavant. Certains bugs ont été résolus par la même occasion, d'autres restent ouverts mais sont désormais connus.

Il y a d'autres façons de gérer les échecs à la dernière étape reliant les paires de configurations intermédiaires consécutives. J'ai codé la façon qui semblait la plus naturelle, et j'en ai mesuré les performances sur les différents tests dont je disposais mais je n'ai pas eu le temps d'en coder et tester d'autres. On pourrait réutiliser partiellement les configurations intermédiaires en ne résolvant à nouveau que quelques solutions autour de la portion où la dernière étape a échoué. Ou encore pouvoir réutiliser des portions de chemins continus déjà résolus lorsque des tentatives futures réutilisent les mêmes extrémités du chemin.

Enfin, le dernier axe de progression de mon algorithme, que je n'ai pas eu le temps de proprement finir, devrait être poursuivi. Il s'agit de trouver des raccourcis dans le chemin trouvé en solution, et ainsi de trouver une meilleure solution. Pour l'instant il n'est possible que de trouver des raccourcis « locaux », qui sont intégralement contenus entre deux des configurations intermédiaires q_i et q_{i+1}. Être capable de trouver des raccourcis globaux qui évitent une des configurations intermédiaires pourrait être un objectif futur.

Bibliographie

[1] D. Bury, J. Izard, M. Gouttefarde, F. Lamiraux.
Continuous Collision Detection for a Robotic Arm Mounted on a Cable-Driven Parallel Robot. CoRR abs/1909.10857, 2019

[2] LaValle, Steven M. Rapidly-exploring random trees: A new tool for path planning. Technical Report. Computer Science Department, Iowa State University (TR 98–11), October 1998.

[3] K. Hauser and V. Ng-Thow-Hing, "Randomized multi-modal motion planning for a humanoid robot manipulation task," The International Journal of Robotics Research, vol. 30, no. 6, pp. 678–698, 2011.

[4] P. S. Schmitt, W. Neubauer, W. Feiten, K. M. Wurm, G. V. Wichert, and W. Burgard. Optimal, sampling-based manipulation planning. In 2017 IEEE International Conference on Robotics and Automation (ICRA), pages 3426–3432. IEEE, 2017. doi: 10.1109/ICRA.2017.7989390. URL http://ais.informatik.uni-freiburg.de/publications/papers/schmitt17icra.pdf.

[5] J. Z.Woodruff and K. M. Lynch. Planning and control for dynamic, nonprehensile, and hybrid manipulation tasks. In 2017 IEEE International Conference on Robotics and Automation (ICRA), pages 4066–4073. IEEE, 2017.

[6] O. Ben-Shahar and E. Rivlin. Practical pushing planning for rearrangement tasks. IEEE Transactions On Robotics And Automation, 14(4):549–565, August 1998.

[7] F. Lamiraux, J. Mirabel. Prehensile Manipulation Planning: Modelling, Algorithms and Implementation. Rapport LAAS n° 20298. 2020. hal-02995125