Joint task sequencing and motion planning for a mobile manipulator-robot

Hannes van Overloop

Joint task sequencing and motion planning for a mobile manipulator-robot

Master's thesis - Master Parisien de Recherche Opérationnelle

Hannes Van Overloop
Christian Artigues, Cyrille Briand, Florent Lamiraux

September 2023

Keywords : Motion Planning, Robotic Task Sequencing, Combinatorial optimisation.

Contents

French summary

Mots clés : Planification de mouvements, Séquencement de tâches, Robotique, Optimisation combinatoire.

Dans le contexte de l'avènement des robots autonomes dans l'industrie, ce stage construit, implémente et étudie une approche heuristique du double problème de séquencement des tâches et de planification des mouvements d'un robot mobile effectuant des tâches sur une pièce 3D de forme libre. Le cas d'étude est l'inspection d'un ensemble de points d'intérêt, chaque inspection étant assimilée à une tâche.

Du fait du grand nombre de degrés de liberté présent sur un tel robot, le nombre de configurations permettant d'effectuer une tâche est infini. Pour pallier ce problème, nous discrétisons le problème et le modélisons comme un Problème du Voyageur de Commerce Généralisé (GTSP). Nous tentons alors, au moyen d'une aggrégation, de limiter la taille de l'instance de GTSP résultante afin de maintenir le temps de résolution à une durée acceptable. Différentes méthodes sont ensuite implémentées pour pallier les problèmes engendrés par une telle aggrégation. Ceci nous permet finalement de réduire drastiquement la taille du problème tout en maintenant des solutions de bonne qualité.

D'autres perspectives sont également présentées mais non implémentées, faute de temps. Finalement, une prise de recul sur les travaux produits et les résultats obtenus est effectuée, afin d'orienter les recherches futures vers des horizons prometteurs.

1 Introduction

As automation is increasingly adopted in the industry, humanoid-like robots are being designed in research laboratories in order to perform more specific tasks, which cannot be tackled with classical and fixed industrial robots [14]. For instance, to perform small tasks such as deburring, on a three-dimensional free-form surface, one would like a robot able to move through the workshop while performing precise tasks. Such robots are thus composed of a *base*, moving on the workshop floor, and an *arm*, used to effectively perform the task.

In this context, **we want to sequence the tasks, while, at the same time, plan the motion of the robot**. All of this should be done in reasonable computation times while providing near-optimal solutions. This problem is known as the *Mobile-Manipulator Robotic Task-Sequencing Problem* [1] (Mobile-Manipulator RTSP). Our goal is to build a **very generic and versatile approach**, *i.e.* one that is easily applicable to different robots and/or parts. Ideally, it should be easily usable within softwares for robotic path-planning.

In the case of deburring or drilling, tasks are holes in the part worked on, but other tasks such as object picking or placing, or product inspection can be considered too.

The remainder of the document is organised as follows. In Section 2, we provide a more in depth definition of the problem before summarizing the state of the art in Section 3. We then model the problem in Section 4, detail our discretisation approach in Section 5, and our resolution method in Section 6. In Section 7, we relate encountered issues and their solutions. Eventually, in Section 8, we take a critical step back to analyse our work and hint future directions.

2 Problem definition

Let us start by defining specific concepts (tasks, quaternions, configurations) before defining the full problem.

2.1 Tasks

Tasks are part of $SE(3)$, the *Special Euclidian group of rigid body displacements*. This algebraic group allows to describe movements of rigid bodies and thus contains the well-known group of rotations in three dimensions, denoted $SO(3)$, as a subgroup.

A task t is thus encoded as a 3D position and direction, and is stored as a translation O_t and a rotation q_t in the coordinate system of the environment. This rotation is encoded as a quaternion [6]. A task is thus stored as a vector of 7 coordinates.

The set of tasks the robot has to execute is the *task-space* or T-space, and is denoted T.

2.2 Quaternions

The quaternion number system extends the complex numbers. A quaternion is written as $q = (w, x, y, z) = w + ix + jy + kz = (s, \mathbf{v})$.

When comparing to complex numbers, w, the scalar part of q, can be seen as the real part, whereas (x, y, z), the vector part of q, can be seen as the imaginary part. As an extension of the complex numbers, a quaternion has three indeterminates : i, j, k. They satisfy $i^2 = j^2 = k^2 = -1$ as well as $ij = -ji = k$, $jk = -kj = i$ and $ki = -ik = j$. This implies that $ijk = -1$.

The last notation must be understood as follows : $s = \cos\frac{\theta}{2}$ and $\mathbf{v} = \mathbf{u}.\sin\frac{\theta}{2}$, where $\mathbf{u}$ is a unit vector defining the axis around which a rotation of θ occurs. As such, quaternions can be seen as rotations in a three-dimensional space. W.R. Hamilton, who first described quaternions, defined them as the quotient of two vectors. They can thus be understood as the rotation required to go from one vector to the other.

The conjugate of q is $\overline{q} = (w, -x, -y, -z)$ and a quaternion is unitary if, seen as a vector of $\mathbb{R}^4$, it satisfies $||q||_2 = 1$.

It is possible to define a product operator $\otimes$ for two quaternions q_1, q_2. The result is equivalent to applying rotations q_1 and q_2 consecutively. Although it is associative, it is not commutative. It is defined as follows :

$$\begin{aligned} q_1 \otimes q_2 = \quad & w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 \\ & + (w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2)i \\ & + (w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2)j \\ & + (w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2)k \end{aligned}$$

Noting that the scalar part of the product is equal to $q_1 \cdot \overline{q_2}$, it is easy to retrieve θ as follows : $\theta = 2 \arccos(q_1 \cdot \overline{q_2})$. Nevertheless, for unit quaternions, the dot-product is close to one and the anticosine operator is very sensitive to errors for small angles.

From $q = (s, \mathbf{v})$, it is obvious that $||\mathbf{v}_{q_1 \otimes q_2}||_2 = \sin \frac{\theta}{2}$, where $\mathbf{v}_{q_1 \otimes q_2}$ is the L^2 norm of the vector part of the product $q_1 \otimes q_2$. This allows us to compute the angle θ as follows :

$$\theta = 2 \arcsin ||\mathbf{v}_{q_1 \otimes q_2}||_2 \tag{1}$$

Alternatively, a quaternion q can be represented as a rotation matrix $R(q)$ expressed as :

$$R(q) = \begin{pmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(wy + xz) \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(xz - wy) & 2(wx + yz) & 1 - 2(x^2 + y^2) \end{pmatrix}$$

From the above expression, it is obvious that $R(q) = R(-q)$, and hence q and $-q$ encode the same rotation.

2.3 Configurations

A position c of the robot, called a *configuration*, is given by the vector of its coordinates as follows : the position and orientation of the base in the coordinate system of the environment, and the rotation angle of each joint in its relative coordinate system for the arm. It can thus be seen as a base configuration c^{base} and an arm configuration c^{arm}.

c is said to be part of the C-space, *i.e.* the *configuration-space*, denoted $\mathcal{C}$.

2.4 Complete problem

The problem is fully defined by a model of the robot, the 3D model of the piece to work on, and the set T of tasks to execute.

The goal is to minimize the total execution time of the tasks, similarly to minimizing the makespan in a scheduling problem. To do so, the order of execution of the tasks has to be decided, as well as how each task is executed, through inverse kinematics.

Figure 1 shows, on the left hand side, the robot we will be using for our experiments. On the right hand side, the robot is shown in a Graphical User Interface (GUI), together with an engine pylon, which is the part to work on.

Since all tasks have the same nature (drilling and deburring, for instance, will not be mixed), the movement of the robot will only be considered up until the tool is in front of the task. Indeed, the execution time from the position in front of the task is considered as a constant. This hypothesis drastically simplifies the motion planning by reducing the chance of collisions, because it keeps the robot further away from the part.

To put it in a nutshell, we have to find feasible configurations, *i.e.* collision-free configurations. These configurations should allow us to reach and thus perform all the tasks. The last requirement

is that there should exist collision-free paths between the chosen configurations. Of course, this selection should minimize the total time needed to execute all tasks.

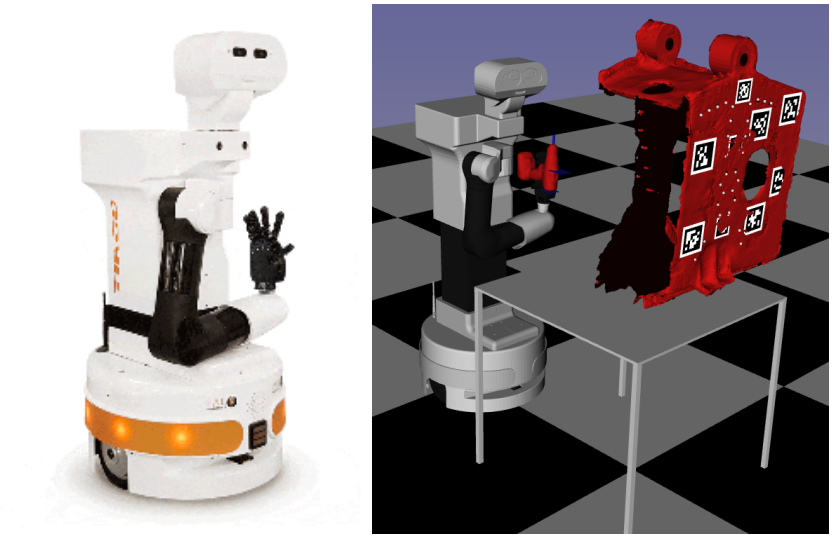

Figure 1: An example of a robot[1]and a piece to work on (in a simulation environment). The task is to insert a tool into holes in order to perform deburring.

2.5 Remarks

A first remark is the fact that it is costly to move the base because it is slow and subject to high uncertainties. Moreover, due to safety reasons, the arm and the base cannot move simultaneously. Therefore, one of the underlying goals is to perform as much tasks as possible from a given base configuration.

In short, the number of different base configurations should be as small as possible and, for a given base configuration, a sequence has to be decided, fixing the order of the tasks that can be reached from it.

A second remark is the fact that exact computation of costs, *i.e.* movement times, is extremely costly since it relies on robotics algorithms computing collision-free paths through space [3]. These algorithms also greatly rely on randomness and hence are not able to provide robust solutions. It is therefore of great importance to minimize calls to these functions by approximating exact movement times by as precise as possible metrics.

Last but not least, since space is continuous, the C-space is continuous. Furthermore, due to the number of degrees of freedom, multiple configurations can be considered to perform a given task (actually infinitely many, due to continuity). This is illustrated in Figure 2. Section 4 details how we cope with this issue.

3 Related work

This internship is a continuation of exploratory work performed in [17]. Summarized below are different heuristics that were tested to address the problem.

To cope with continuity, the C-space is discretized as follows. Given the set T of tasks, $|T|$ configurations were generated and it was then checked which tasks could be performed from each

[1] `https://pal-robotics.com/robots/tiago/`

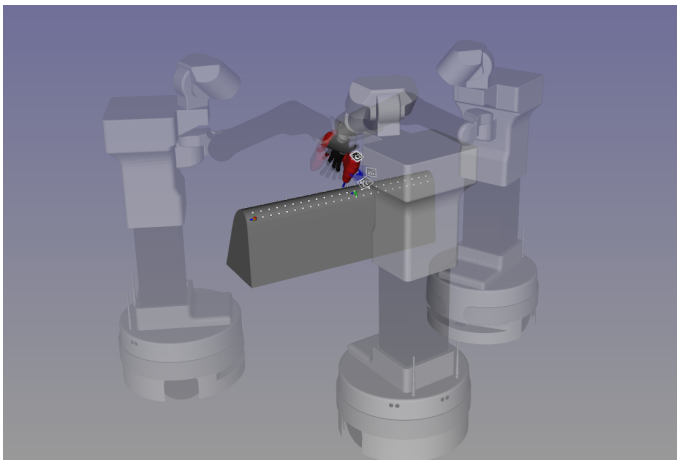

Figure 2: Three configurations reaching the same task.

resulting base configuration, resulting in $\mathcal{O}(|T|^2)$ configurations. This provides an instance of the *Symmetric Generalized Traveling Salesman Problem* (E-GTSP).

Differently from in the TSP, in the GTSP, vertices of a graph are gathered in clusters and the goal is to find a path of minimum cost going over at least one vertex per cluster. In our case, a cluster is composed of configurations completing a given task, as a consequence clusters are disjoint, and costs consist of the time to go from one configuration to another, hence they are symmetric since robot movements are assumed to be reversible. The graph is therefore undirected. Furthermore, the path has to go over exactly one vertex per cluster, hence E-GTSP (where E stands for equality).

A GTSP solver would then solve the resulting instance of the problem. Since checking feasibility of a task from a given base configuration requires inverse kinematics, approximating feasibility by an error metric was considered promising. A threshold would then tell if a task is feasible from a given base configuration. This was then combined with heuristics aimed at reducing the number of considered base configurations and thus the size of the GTSP instance.

The first heuristic solves a *Set Covering Problem* (SCP) selecting base configurations covering all tasks. Since the error metric is used for feasibility of tasks, set covering is applied multiple times to ensure all tasks can indeed be performed and tasks feasible from within multiple selected configurations are assigned to the closest one according to a Euclidean-like C-space distance measure.

The second heuristic applies the *K-medoids* clustering algorithm (a variant of *K-means*). The main issue is then that picking parameters for the algorithm to perform well, such as K, is greatly problem-dependent, which we do not want. Nevertheless, by reducing the number of considered base configurations, these methods reduce the number of calls to the inverse kinematics algorithm as well as the size of the GTSP, resulting in much smaller computation times without deteriorating the solution too much.

Notably, the GTSPs were solved using the algorithm proposed in [18], used as a *black-box*, which did not allow warm start after the update of certain costs, for instance.

A very generic framework to model robotics applications of simultaneous task sequencing and motion planning is presented in [22]. Here, the GTSP encountered when discretizing the C-space is translated into an ATSP [16], allowing the use of Google OR-Tools routing algorithms to solve the problem. Any other, possibly better performing, TSP solver could also be considered.

Another active research area concerns the extension of the rapidly-exploring random trees (RRT) algorithm for path-planning. The latter is only able to return a feasible path, but was extended into RRT* to take into account a cost to minimize. It is proven to asymptotically find

the path of minimum cost [19]. Managing a list of tasks to execute is also done by the extensions developed in [7], [8].

Because of the slow convergence rate of RRT*, these algorithms struggle to solve problems with a lot of tasks, but they have the advantage to manage a wide variety of cost functions and thus apply to problems very different, at first sight, from the one at stake here, *e.g.*, the simulation and study of proteins.

Let us go back to a more combinatorial optimisation-like approach now.

One can start by selecting a sequence of the tasks, thanks to a TSP solving algorithm and a *T-space metric* that has to be defined appropriately with respect to the robot's geometry.

Then, given this sequence and configurations for each task, obtained by inverse kinematics, Dijkstra's algorithm computes a shortest path over the configurations based on an appropriately determined *C-space metric*.

Eventually, the collision-free path can be computed using a motion-planning algorithm. This was originally done for fixed base industrial robots [20].

Let's first note that the C-space metric used here, *a priori*, looks more appropriate than the L^2-like distance used by [17]. Put simply, this method decomposes the problem into two smaller subproblems, namely a TSP and a shortest path.

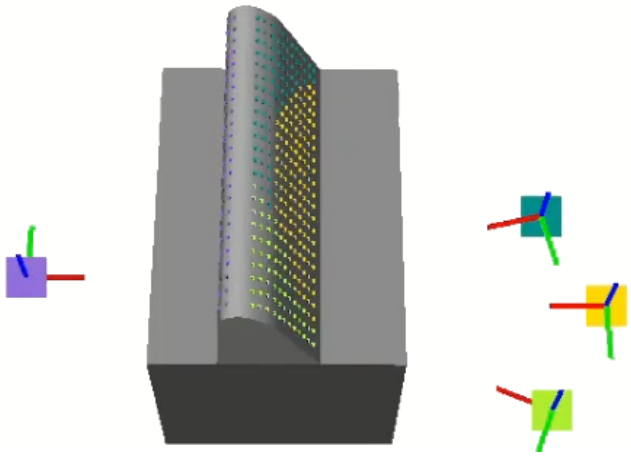

Figure 3: Solution obtained by [15] seen in a GUI.

The above method was extended to mobile manipulators [15]. For a mobile manipulator, the method proposed in [20] is applied to every chosen base configuration. Selecting base configurations is done by applying a SCP of minimal cardinality to a bipartite graph linking the tasks and the base configurations from which they can be reached. Those base configurations are easily obtained by discretizing the workshop floor, and reachability is verified by geometric properties. These properties rely on heavy hypotheses and computations based on the geometric characteristics of the considered robot.

The result of the SCP is a partition of the tasks and a base configuration for every part of the partition. The method for fixed base manipulators is then applied to every part partition and its associated base configuration.

Although the approach produced promising results, shown in Figure 3, it is difficult to apply it to another robot, and the hypotheses made restrict the tasks that are considered reachable. In the illustrated use case, tasks are aggregated into four clusters and the robot is thus given four base configurations to execute all the tasks.

Gathering the best of both worlds, we believe there is still space for a better performing approach. First, clustering the tasks in order to generate base configurations on those clusters sounds

more promising to us than discretising space and consider all obtained positions because it should allow more emphasis on areas regarded as more important or promising. Next, solving the GTSP in two stages, as done in [20], potentially hides good solutions. Eventually, using more appropriate distance measures will lead to more meaningful costs and thus better solutions, before the exact collision-free path is computed.

Our approach can thus be synthesized as follows : tasks are clustered in order to generate a reduced amount of configurations while still ensuring all tasks are reached. This results in a smaller GTSP instance, which we solve to get the sequence of tasks as well as the configurations executing every one of them. Eventually we generate the collision-free path associated with the solution.

4 Problem modelling

This section aims at explaining why and how the problem is solved through a discrete model, whereas some variables are obviously continuous.

4.1 MINLP

Informally, the optimisation model of the problem to be solved roughly looks as follows :

$$\begin{array}{rl} \text{Min} & \text{execution time} \\ \text{s.t.} & \text{all tasks are reached} \\ & \text{all configurations are feasible} \\ & \text{there are no collisions} \end{array} \tag{2}$$

This problem is a *Mixed Integer Non-Linear Problem* (MINLP). Indeed integer, or even boolean, variables would tell whether a task is reached, and continuous variables come from the coordinates encoding the robot's configurations. Reachability constraints are far from being linear as they are projections on manifolds of the C-space. Neither are the feasibility constraints.

Since a single configuration consists of about 20 continuous coordinates, $\mathcal{O}(20|T|)$, which is a lower bound, there are a lot of continuous variables. Adding the $\mathcal{O}(|T|)$ integer variables, obviously the problem is huge and untractable. Therefore, the generally adopted approach tends to discretize the C-space, so that the continuous variables can be removed, which allows us to get back to combinatorial optimisation techniques.

4.2 GTSP

As explained in Section 3, discretizing the C-space provides us with a GTSP model of the problem. The latter is composed of a finite set of configurations, each one reaching one task.

This allows us to build a graph $G = (V, E, C, c)$, forming the GTSP instance I :

- V contains a vertex per configuration.
- C is the set of clusters, and a cluster is composed of vertices reaching the same task. The clusters form a partition of the vertices.
- E is composed of all edges between vertices of different clusters.
- The cost c_e of an edge e is the time needed by the robot to get from one configuration to the other.

This definition gives an E-GTSP instance with disjoint clusters and symmetric costs. The graph is therefore undirected. It is obvious that symmetric E-GTSP is $NP - hard$ since it generalizes to the symmetric TSP when clusters are of size 1. E-GTSP finds applications in various domains spanning flowshop scheduling [10], airplane control [4] or drone deliveries [21]. In the latter, problem specific constraints can introduce other variants of the problem [9].

As an illustration, Figure 4 shows the case with two tasks A, B, and two configurations per task. The dotted lines represent the clusters and thus enclose configurations (*i.e.* vertices) on the same task (*i.e.* part of the same cluster). Costs are not specified but are encoded in the the edges.

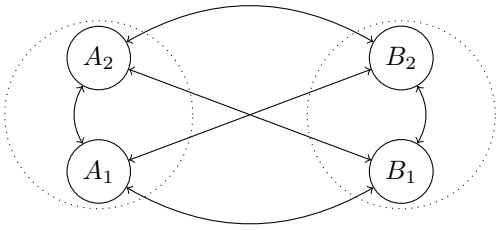

Figure 4: Graph encoding the GTSP instance for two tasks and two configurations per task.

The question remaining now is how to discretize the C-space. Indeed, this requires to pick a finite number of configurations, ideally including the ones providing an optimal solution of the MINLP problem. Section 5 provides more details.

5 Discretisation

This section details how configurations are selected, in a as smart as possible way, in order to avoid to lose the ones that can be part of some global optimal solutions.

5.1 Naive approach

To ensure reachability of all tasks, one way to discretize the C-space is to generate, for every task $t \in T$, n_{config} configurations reaching it. We call this the *naive method*. This results in B base configurations [17]. By then checking which tasks are reached from every resulting base configuration, $\mathcal{O}(B|T|)$ configurations are obtained, and as many vertices in the GTSP instance.

Our goal is to reduce the amount of configurations considered, while being able to cover every task.

This means reducing B and/or $|T|$. The ability to cover all tasks implies it is not possible, for instance, to consider a random subset of T and apply the naive method to that subset. Doing so would introduce a significant risk of not being able to perform a task. Reducing n_{config} to reduce B, on the other hand, would seriously reduce the variety of the base configurations and thus the quality of the final solution.

The size of T has thus to be *reduced*, while ensuring all tasks are reached. To do so, we partition the tasks according to their origin and orientation (Section 5.3). The centroids of the clusters form fictitious tasks supposed to be representative of the real ones. This nevertheless requires to define a distance measure between tasks, called *T-space metric* (Section 5.2).

5.2 T-space metric

The T-space metric, referred to as d_{tasks}, is intended to allow the clustering algorithm to quantify *how far apart* two tasks are from each other. It should take into account the distance between the origins of two tasks, as well as the angular difference between their orientations.

This metric is thus a heuristic distance measure on $SE(3)$. Furthermore, it should be defined in a way allowing it to be easily tuned according to the considered robot, in order for our approach to remain versatile with respect to the handled instances. This makes the metric robot-dependent.

Recall a task t is encoded as (O_t, q_t) where O_t is its origin and q_t a quaternion encoding its rotation. We could use [11] for the definition of the metric, as it explores different distance measures

for quaternions.

Since both the origin and the orientation have to be taken into account, we define the T-space metric between two tasks u, v as follows :

$$d_{tasks}(u,v) = \sqrt{d_{origin}(O_u, O_v)^2 + k_{tasks}.d_{direction}(q_u, q_v)^2} \tag{3}$$

The factor k_{tasks} allows to adapt the importance of a difference in direction for the considered robot. The distance measure is thus tunable according to the robot, and is close to a weighted L^2 distance.

For the distance between the origins of the tasks, the most reasonable choice is to take the Euclidian distance in $\mathbb{R}^3$. Hence :

$$d_{origin}(O_u, O_v) = ||O_v - O_u||_2 \tag{4}$$

When computing the distance between two directions, it is the angle between these directions that is interesting. A first definition could thus use equation (1). Hence :

$$^1d_{direction}(q_u, q_v) = 2 \arcsin ||\mathbf{v}_{q_u \otimes q_v}||_2 \tag{5}$$

The above definition differs from what is presented in [11], where the distance between two quaternions q_1, q_2 is defined as follows :

$$^2d_{direction}(q_u, q_v) = \arccos\left(|q_u \cdot q_v|\right) \tag{6}$$

In equation (6), the anticosine of the absolute value of the dot-product of the two quaternions is taken as the distance between them. Although the conjugate is not considered here, this definition has to be put in perspective with what is presented in Section 2.2 : $\cos\frac{\theta}{2} = q_u \cdot \overline{q_v}$. It is thus a computation of the angle between the two quaternions.

The dot-product is taken in absolute value in order to eliminate the symmetry between q and $-q$, which represent the same rotations.

Since, $\theta = 2\arccos(q_u \cdot \overline{q_v})$, one issue with equation (6) is the fact that it struggles to cope with quaternions having very different directions. Indeed, a dot-product close to -1 should represent very different directions, but since it is taken in absolute value, it is close to 1, which represents quite similar directions. This issue is discussed in Section 7.2. Furthermore, as mentioned earlier, the anticosine operator is very sensitive to errors for values close to 1, *i.e.* small angles.

Both equations (5) and (6) have a limited scope of angles they handle. We therefore opt for the two-argument antitangent operator, denoted *arctan2*, and define the distance between two quaternions as follows :

$$d_{direction}(q_u, q_v) = 2.\text{arctan2}(||\mathbf{v}_{q_u \otimes q_v}||_2 \ , \ q_u \cdot \overline{q_v}) \tag{7}$$

This definition is able to manage angles superior to $\frac{\pi}{2}$ and returns an angle in $[0, \pi[$, corresponding to the absolute value of the angle between the two tasks u, v. Hence, the distance measure is positive and symmetric, has value 0 for a pair (u, u), but does not satisfy the triangle inequality. Although the triangle inequality is a desirable property, it is not an issue not to verify it, since the distance measure is used as a heuristic to estimate *how far apart* tasks are for the robot.

5.3 ISODATA

To cluster the tasks, an algorithm with parameters independent of the points to be clusterized is needed. In our case, the points are the tasks. At the same time, we would like to be able to tune the algorithm according to the robot.

ISODATA (Algorithm 1) meets the above-defined expectations. Its parameters are robot-dependent. More precisely, they depend on the T-space metric, which in turn is robot-dependent.

The algorithm is given a set of points P that it will gather in k_{clus} clusters $S = (S_i)_{i\in[k_{clus}]}$ of centroids $Z = (z_i)_{i\in[k_{clus}]}$, of size at least n_{min}, and according to the distance d_{tasks}. It starts with $k_{clus} = k_{init}$ clusters and iteratively splits one when its variance exceeds σ_{max} or combines two when the distance between the centroids is less than L_{min}. At most P_{max} combinations are performed at each iteration and the algorithm stops after I_{max} iterations.

Thus, different values are computed:

- $\Delta_i = \frac{1}{|S_i|}\sum_{x\in S_i} d_{tasks}(x, z_i)$: the average distance to the centroid within cluster i.
- $\Delta = \frac{1}{n}\sum_{i\in[k_{clus}]} |S_i|\Delta_i$: the weighted average distance between centroids.
- $\sigma_{il} = \sqrt{\frac{1}{|S_i|}\sum_{x\in S_i}(x_l - z_{il})^2}$: the variance according to coordinate l within cluster i.
- $d_{ij} = d_{tasks}(z_i, z_j)$: the distance between two centroids z_i and z_j.

In our case, the set of points is the set of tasks, *i.e.* $P = T$. The latter are encoded by affine vectors of 7 coordinates as explained in Section 2.1.

To ensure the algorithm performs well, σ_{max}, d_{tasks} and L_{min} must be adapted to the geometry of the robot. d_{tasks} was defined in Section 5.2, and we fix $n_{min} = 1$ to allow isolated tasks to form their own cluster.

A general description is also provided in [13].

Algorithm 2 details how a cluster S, of centroid z and variance σ, is split. It requires a parameter α caracterising the distance to z, as well as the distance measure d_{tasks} between two points. It returns two clusters S_1, S_2 of centroids $z_1 \neq z_2$, such that $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.

Since the clustering algorithm is only given the set of points, it does not take into account that the tasks, which form the set of points, should be located on the surface of the part. This issue is discussed in Section 7.1.

5.4 Configuration generation

Gathering all the above knowledge, we can precisely define how to discretise the C-space in a, let us hope so, clever way.

After having clustered the tasks of T, we obtain a set of fictitious tasks T^{fict} expected to be representative of T. For every $t \in T^{fict}$, n_{config} configurations reaching it are generated. This results in B^{fict} base configurations. Since $T^{fict} \leq T$, the following inequality holds : $B^{fict} \leq B$.

We then check which tasks $t \in T$ can be reached from the resulting base configurations. This results in $\mathcal{O}(B^{fict}|T|)$ configurations and as many vertices in our GTSP instance.

Although this sounds powerful, it does not ensure all tasks can be reached from the generated base configurations. Because of this, the configuration generation procedure is a bit more complicated and goes as follows :

1. Clustering on T. Tasks that could not be reached are put in T'.
2. Clustering on T'. Tasks that could not be reached are put in T'.
3. Naive approach on T'.

The clustering is applied twice in order to keep the number of configurations as small as possible but since it cannot be applied infinitely many times, the naive approach is applied after the second run, hoping T' got small by then.

Having generated the configurations, *i.e.* the vertices of the GTSP instance, we can now further build that instance.

Algorithm 1: ISODATA

INPUT : P, k_{init}, n_{min}, I_{max}, σ_{max}, L_{min}, P_{max}, d_{tasks}
OUTPUT : A clustering of the tasks

// Initialisation
iteration $\leftarrow 1$
$k_{clus} \leftarrow k_{init}$
Random selection of k centroids $Z = (z_i)_{i \in [k_{clus}]}$ in P.
// Allocation
iteration $\leftarrow$ iteration $+ 1$
Allocation of points of P to the closest centroid according to d_{tasks} to form clusters $(S_i)_{i \in [k]}$.
Removing clusters i verifying $|S_i| < n_{min}$.
Updating centroids.
Updating k_{clus}.
If deletion happened in step 6 : go to step 4.
Computation of mean distances $(\Delta_i)_{i \in [k_{clus}]}$ and Δ.
if iteration $= I_{max}$ **then**
    $L_{min} = 0$ and go to step 21.
if $\left[2k_{clus} > k_{init}\right]$ AND $\left[\text{even iteration OR } k_{clus} \geq 2k_{init}\right]$ **then**
    Go to step 21.
// Splitting
Computation of the variances $(\sigma_i)_{i \in [k_{clus}]}$ and $\sigma_i^{max} = ||\sigma_i||_\infty$
for $i \in [k_{clus}]$ **do**
    if $\left[\sigma_i^{max} > \sigma_{max}\right]$ AND $\left[\lfloor \Delta_i > \Delta \text{ and } n_i > 2(n_{min} + 1) \rfloor \text{ OR } k_{clus} \leq \frac{k_{init}}{2}\right]$ **then**
        Split S_i in two new clusters close to z_i (*cf.* Algo 2).
        $k_{clus} \leftarrow k_{clus} + 1$
If a cluster was split, go to step 4.
// Combining
Computation of $(d_{ij})_{i,j \in [k_{clus}]}$.
Order $(i,j)_{[k_{clus}] \times [k_{clus}]}$ by ascending d_{ij}.
Take the P_{max} first pairs verifying $d_{ij} < L_{min}$. Combine the associated S_i, S_j. Update k_{clus}.
If iteration $\leq I_{max}$, go to step 4.
return $(S_i)_{i \in [k_{clus}]}$

5.5 Performances

Before moving on to the final computations necessary to generate the GTSP instance, let us take some time to study the performances of the discretisation method.

Supposing that the number of clusters is proportional to the number of tasks, both the naive approach and our clustering approach produce a number of configurations of the order of magnitude of $\mathcal{O}(|T|^2)$. Nevertheless, compared to the naive method, the clustering method produces drastically less base configurations. When the naive method produces exactly $|T|$ base configurations, the clustering allows to divide this quantity by roughly 10. Unfortunately, the Landau notation hides these gains.

Since the problem is scarcely studied, very few instances are available in the litterature. We work with a Tiago robot (Figure 1) and had two parts at our disposal. One was given to us by Airbus and is an engine pylon (in red on the right hand side image of Figure 1), and the second one is an airfoil used as use case in [15] (left hand side of Figure 5).

The **engine pylon** instance contains 18 tasks, all located on the same surface which means

Algorithm 2: Splitting a cluster

INPUT : $S,\ z,\ \sigma,\ \alpha,\ d_{tasks}$
OUTPUT : Divides a cluster into two

// Initialisation
$pos \leftarrow \text{argmax}_l\ \sigma_l$
// Creation
Creation of S_1, S_2 of centroids $z_1 = z_2 = z$.
$z_1^{pos} \leftarrow z^{pos} - \alpha z^{pos}$
$z_2^{pos} \leftarrow z^{pos} + \alpha z^{pos}$
// Allocation
for $x \in S$ **do**
    $k \leftarrow \text{argmin}_{i \in [2]}\ d_{tasks}(x, z_i)$
    $S_k \leftarrow x$
// Caracterisation
Update of the centroids z_1, z_2 and the variances σ_1, σ_2.
return S_1, S_2

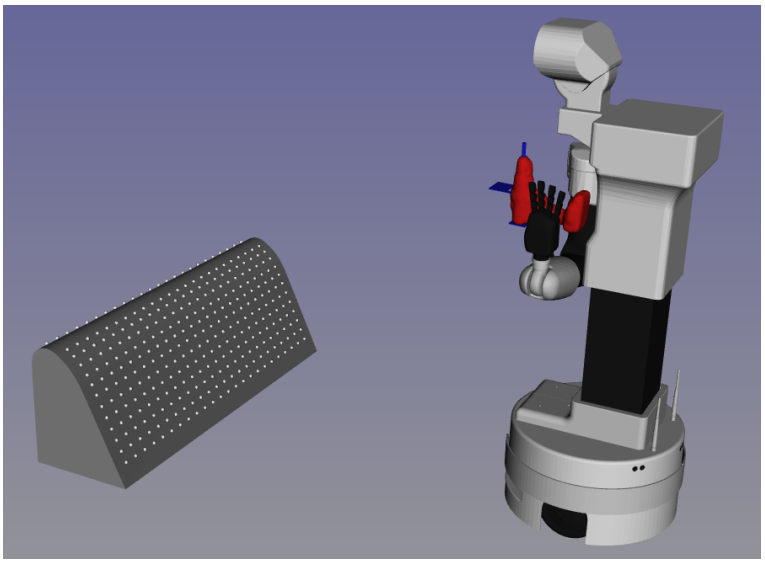

Figure 5: Tiago robot and the airfoil (in the GUI)

they all have the same orientation. Since the piece is not that big, and considering the geometric properties of the robot, the clustering algorithm can be expected to put all tasks in one single cluster. Furthermore, a configuration on a given task almost always allows the robot to reach all 17 other tasks.

On one hand, the naive method almost always produces $18 \times 18 = 324$ configurations. On the other hand, the clustering puts all tasks in a single cluster, as expected. Taking $n_{config} = 2$, this reduces the number of base configurations down to 2, compared with 18 with the naive method. In the end, the clustering method almost always produces $18 \times 2 = 36$ configurations.

The use of the clustering method thus resulted in a reduction of the number of generated configurations by a factor of $^{324}/_{36} = 9$ for the engine pylon. Since the distance matrix of the GTSP instance is of size the square of the number of configurations, this reduces the size of that instance by a factor of $9^2 = 81$, which is a non-negligible gain.

The **airfoil** instance contains 336 tasks and is thus consequently bigger than the one of the engine pylon. The naive approach would produce around 10^5 configurations. It is more challenging

not only because of the increased size, but also because of the fact that the tasks have different orientations since they are not all located on the same side of the part. This will challenge the quality of the clustering algorithm.

As can be seen in Figure 3, applying the method of [15] (detailed in Section 3) to the airfoil and a Denso VS-087 robot with 6 degrees of freedom (DOF), results in four clusters and thus four base configurations. Three allow to reach the tasks on the angled side of the part whereas a fourth one makes sure all tasks on the vertical side are reached. Considering the abilities of the Denso robot and the fact that one base configuration sufficed to reach all the tasks on one side, we expect a best-case scenario where all tasks can be reached from within as little as two different base configurations. For $n_{config} = 2$, this would give around 10^3 configurations.

In our case, the Tiago robot's arm has a slightly smaller range than the Denso. Nevertheless, when applying the clustering algorithm with correct parameters, we manage to generate as little as 2.10^3 to reach all tasks (with $n_{config} = 2$). This is a reduction by a factor of ${}^{10^5}/_{2.10^3} = 50$ compared to the naive method. This gain reduces the size of the GTSP instance by a factor of $50^2 = 2500$.

6 Resolution

To complete the instance, the costs of its arcs need to be computed. Recall that exact cost computation is extremely time consuming because it relies on collision-free path planning algorithms. We therefore approximate this cost through a *C-space metric* defined in Section 6.1.

Once the costs computed, Section 6.2 explains how the resulting GTSP instance is solved.

6.1 C-space metric

Recall a configuration is given by the configuration of the base c^{base} and the configuration of the arm c^{arm}. This section defines the metric d_{config} measuring the distance between two configurations c_1 and c_2.

This metric should be robot-dependent to be as precise and realistic as possible. It will therefore have to take into account both some planar distance, related to the more costly movement of the base, as well as a distance related to the movement of the arm. For the latter, [20] tests diverse metrics.

We define the distance measure as follows. The difference in order of magnitude of the time needed to move the base compared to moving the arm is taken into account through the constant k_{config}.

$$d_{config}(c_1, c_2) = k_{config}.d_{base}(c_1, c_2) + d_{arm}(c_1, c_2) \tag{8}$$

We define the distance for the base as the L^1 distance between its planar coordinates, also called *manhattan distance*, multiplied by a factor depending on the variation of its direction. The importance of a variation of the direction is encoded in k_{base}.

$$d_{base}(c_1, c_2) = \left|\left|c_2^{base} - c_1^{base}\right|\right|_1 \left(1 + k_{base}|\theta_2^{base} - \theta_1^{base}|\right) \tag{9}$$

For the distance between arm configurations, multiple definitions are possible. We consider the two following ones :

$$d'_{arm}(c_1, c_2) = \max_i \left|\frac{c_2^i - c_1^i}{v_i^{max}}\right| \tag{10}$$

$$d''_{arm}(c_1, c_2) = \sqrt{\sum_i \left(\frac{c_2^i - c_1^i}{v_i^{max}}\right)^2} \tag{11}$$

The first one, (10), is the maximal motion time of the different joints when omitting the acceleration phase. The second one, (11), is an L^2 norm of the angular differences, weighted by the joint speeds.

Eventually, in equation (8), when two configurations differ in base configuration, the distance between them is composed of the distance between the base configurations and two arm configuration distances corresponding to the folding and unfolding of the arm before and after the base is moved. If only the arm has to move: $d_{config} = d_{arm}$.

This metric allows us to compute the costs of the edges of the graph G encoding our GTSP instance. Section 6.2 explains how the latter is solved.

6.2 Solving the GTSP

Following [22], we attempt to solve the problem using Google OR-Tools, an open-source software suite for optimisation that incorporates heuristics to solve the TSP. This requires to transform the GTSP instance into an ATSP one [16].

Given a GTSP instance of our problem $I = G = (V, E, c)$, an equivalent ATSP instance $J = G' = (V, A, c')$ can be built as follows :

1. Orientation of the edges by doubling them, as our problem is originally symmetric.
2. Within each cluster c^k, arbitrarily number the vertices: $c^k_1, ...c^k_{|c^k|}$.
3. Within each cluster, add arcs of cost 0 to form a cycle over the vertices according to their numbering, as our problem originally contains no inner-cluster arcs.
4. Every arc (c^k_i, c^l_j) between two clusters $c^k \neq c^l$ is replaced by an arc (c^k_{i-1}, c^l_j) of same cost (with the convention that if $i = 1$, then $i - 1 = |c^k|$).

G and G' have the same set of vertices V, the set of arcs in G' satisfies $|A| = 2|E| + |E| - k$, and the cost function c is trivially extended to A with steps 3 and 4.

Figure 6 shows the TSP instance obtained after applying the transformation to the GTSP instance from the example depicted in Figure 4.

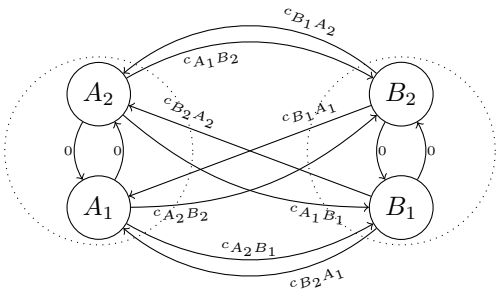

Figure 6: TSP instance associated with the GTSP from the previous example.

From an optimal solution of the TSP instance J, an optimal solution, of same cost, of the GTSP problem can easily be retrieved.

Indeed, when the TSP solution first visits a vertex c^k_i of a cluster k, it visits all other vertices of that cluster by going over the cycle of cost 0 until vertex c^k_{i-1}. Then, it leaves the cluster by going over edge (c^k_{i-1}, c^l_j), which is a clone of the edge (c^k_i, c^l_j) used by the optimal GTSP solution after visiting vertex c^k_i.

6.3 Performances

As mentioned earlier, the GTSP problem obtained for the engine pylon instance contains approximately 36 vertices. They correspond to two base configurations, both allowing the robot to reach all tasks. This allows to easily find a solution, of which we will describe the quality now.

The solution is shown in Figure 7.

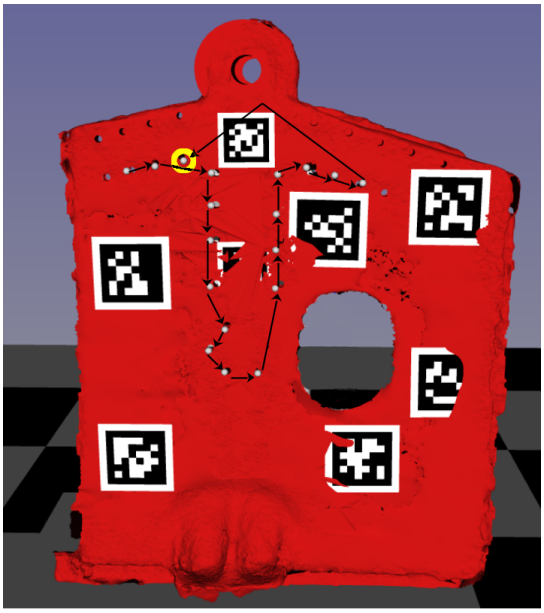

Figure 7: Solution obtained with OR-Tools for the engine pylon.

Let us first say that the resolution was able to select configurations all corresponding to the same base configuration. Since, changing the position of the base is very costly, if one base configuration is sufficient, it is of great value to select it, and no other ones.

The tour depicted in Figure 7 thus shows the order in which the tasks are performed. At first glance, the tour does not look too bad. The algorithm performs reasonably well. Nevertheless, when taking a closer look, one can see that at least one task is missing, which forces the robot to come back to it at the end of the tour. This task is circled in yellow on Figure 7.

For a small instance like the one of the engine pylon, OR-Tools is thus able to find a near optimal solution. The latter required $10s$ of running time, which is reasonable. Nevertheless, the heuristic nature of OR-Tools leads to a few issues when running bigger instances. This is discussed in Section 7.3.

7 Issues

While presenting our approach, we already pointed out a few possible issues. This section details theses issues and explains how they were fixed.

7.1 Projection of the fictitious tasks

Since the clustering algorithm is only given the set of points, it does not take into account that the tasks forming the set of points should be located on the surface of the part. This can result in fictitious tasks being located above the part, or, more annoyingly, inside the part, as illustrated in Figure 8.

Indeed, since the goal of the fictitious tasks is to generate configurations on them, if the latter are inside the part, any configuration reaching it will collide with the surface of the part and will not be considered valid.

Below, two methods aimed at fixing this issue are presented, one of which we implemented.

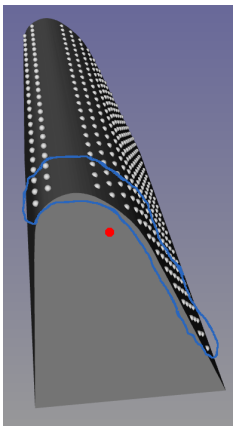

Figure 8: Example of a fictitious task inside the part, here the airfoil.
White : origins of the real tasks.
Blue : limit of the cluster.
Red : origin of the centroid (inside the part).

In a quite straightforward way, the fictitious tasks can be projected on the surface of the part. Since the latter is encoded as a mesh, the projection is done by looking for the node of the mesh that is closest to the origin of the fictitious task, with respect to the L^2 norm in $\mathbb{R}^3$. That node is then considered as the origin O_t of the fictitious task t.

The projection is a relatively heavy task, since it requires the computation of the distance to every node of the mesh. Therefore it is done only once : after the clustering procedure. This possibly deteriorates the solution returned by ISODATA in terms of representation quality of the real tasks, but a compromise has to be found.

Another, more complicated, idea, called *double projection*, was elaborated but not implemented. It is illustrated in Figure 9 and is summarized below. First, denote E, the ellipsoid hull of the part, and $\mathcal{S}$ the surface of the workshop floor. Define the closed curve ϵ as : $\epsilon = E \cap \mathcal{S}$. Define l as the maximum distance between the robot and a task it is able to reach from its current position. The method then goes as follows :

1. Project the origin of every real task $t \in T$ on the ellipsoid hull of the part, with respect to the Euclidian distance on $\mathbb{R}^3$. The projections are directed such that they are normal to the surface of E, and are denoted p_E^t.
   This breaks down to computing, for every $t \in T$, the following quantity :
   $$p_E^t \in \underset{p \in \mathcal{S}_E}{argmin} ||t - p||_{\mathbb{R}^3}$$

2. Further project these projections on the closed curve ϵ, also with respect to the Euclidian distance on $\mathbb{R}^3$. The orientation is unchanged. These second projections are denoted p_ϵ^t.
   This breaks down to computing, for every $t \in T$, the following quantity :
   $$p_\epsilon^t \in \underset{p \in \epsilon}{argmin} ||p_E^t - p||_{\mathbb{R}^3}$$
   Note that $p = (x, y, z) \in \epsilon$ implies $z = 0$ and (x, y) is on the closed curve ϵ.

3. Regarding l as the length of a segment of ϵ, apply a 1D clustering on the $(p_\epsilon^t)_{t \in T}$ with $\sigma_{max} = l$. The used distance measure should define the distance between two points as the distance to travel on ϵ to go from the first point to the other one.

These steps allow to generate fictitious tasks supposed representative of the real tasks. It then remains to generate base configurations for the robot. This can be visualised as follows : place

the robot at the position of the fictitious task, move it in the direction normal to ϵ over a distance depending of the robot (far enough to reach as many real tasks as possible, but not too far either or the robot will not reach any real task at all). Direct the base along this same direction, such that the robot is facing the part. This is illustrated in Figure 10.

This ensures that configurations are not in collision with the part.

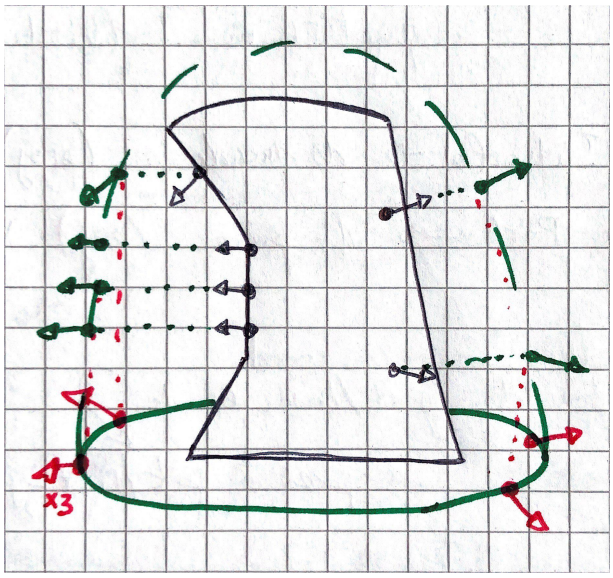

Figure 9: Visual representation of the double projection method.
Black : real tasks $t \in T$, **Green** : projections p_E^t on the ellipsoid hull E, **Red** : projections p_ϵ^t on the closed curve ϵ.

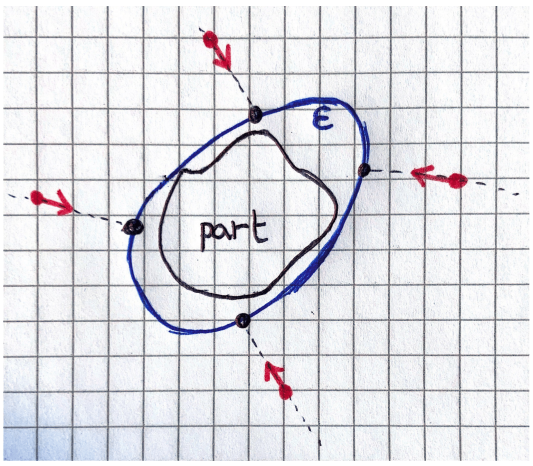

Figure 10: Visual representation of the base configuration generation after the double projection.
Black : origins of the fictitious tasks, **Red** : resulting base configurations.

Due to the complexity of computing the ellipsoid hull of a mesh, as well as the good performances and simplicity of the first method, the double projection method was not implemented. Nevertheless, we expect it to perform well on parts having a volume approaching convexity. The performances could be much more disappointing if that hypothesis were not satisfied.

7.2 Orientation of fictitious tasks

The metrics defined on the T-space in Section 5.2, are useful for an algorithm such as ISODATA that requires a lot of distance computations, because they are easy to compute.

To compute the centroids, ISODATA averages every one of the seven coordinates of the vector encoding a task. Since these centroids represent fictitious tasks, the four coordinates of the vector that correspond to the quaternion, *i.e.* the direction, are also simply averaged. This does not, unfortunately, provide something like an *average direction*, as one would expect.

The following subsections present how an average direction can be computed.

7.2.1 Quaternion average

To solve this problem, a quaternion *barycenter* can be computed through the Frobenius-norm and the rotation matrix [5] associated to the quaternion.

This method can be implemented efficiently as it breaks down to computing the eigenvector associated to the maximum eigenvalue of some 4x4 matrix derived from the sum of the rotation matrices. Taking the eigenvector of unit norm gives us a unit quaternion q_t for the barycenter, which is the direction of fictitious task t.

However great sounding, this method showed poor empirical performances with fictitious tasks still having orientations far from the expected *average*. Indeed, it is not rare to have all real tasks directed to the same region (not the same point, as that would mean they have the exact same direction), but still obtain an average direction going the opposite way. This is possibly due to the nature of the matrix of which the eigenvector is computed : it is not clear what this matrix is supposed to represent.

This method is thus not satisfying. A more basic approach eventually solved the problem of the orientation of the fictitious tasks.

7.2.2 Normal to the surface

Rather than looking for some kind of average, the most robust, and perhaps easiest, approach was to simply direct the fictitious task such that it is normal to the surface of the part. This ensures the ability to generate configurations on the task, and is in line with the fact that, in the studied cases, all real tasks are also normal to the surface. Furthermore, regarding the nature of practical use cases (drilling, deburring, quality checking, etc ...) tasks being normal to the surface is a sound hypothesis.

Retrieving the normal to the surface is not straightforward since the latter is encoded as a mesh, and thus a set of vertices. Nevertheless, since the fictitious tasks are projected on the surface, two points are available : the centroid of the cluster, and its projection on the surface. The direction d (a vector of size 3) defined by these two points should be close to normal to the surface at the location of the projection and can thus be used as the direction of the fictitious task. Since directions of tasks are encoded as quaternions, the quaternion has to be derived from its rotation matrix, which itself is derived from the vector d.

Note that the direction defined by the centroid and its projection is not, in the general case, exactly normal to the surface. This, in turn, is due to the discretisation of the surface to encode it as a mesh.

Deriving the rotation matrix R from the vector d goes as follows.

Recall that a task t is given by its origin O_t and its direction q_t. Now, for any task in T, the coordinate system associated to it has O_t as origin, and the direction defined by q_t as X-axis. It is such a coordinate system that the rotation matrix is expected to define.

One first has to apply the Gram-Schmidt orthonormalisation process to orthonormalise the following set of three vectors : $\{d, (0,1,0), (0,0,1)\}$ into $\{e_1, e_2, e_3\}$. R is then defined as follows, where $\wedge$ is the cross product :

$$R = \begin{pmatrix} e_1^T & e_2^T & (e_1 \wedge e_2)^T \end{pmatrix}$$

Starting the Gram-Schmidt process with d ensures e_1 has the same orientation as d, and using the cross product for the third column of R ensures $det(R) = 1$, which ensures that R is a rotation matrix associated to a right-handed coordinate system. R is thus **the** rotation matrix associated to **the** right-handed coordinate system with X-axis corresponding to d. The wanted quaternion can then be retrieved from the matrix.

7.2.3 Practical behaviour

Let us first pin the importance of having $det(R) = 1$. If it were equal to -1, R would define a left-handed coordinate system. But, the computation of the quaternion associated to a rotation matrix makes the hypothesis of a right-handed coordinate system. A left-handed one would produce *some* quaternion, rather uncorrelated to the computations.

Combining the solutions of Sections 7.2 and 7.1, the obtained fictitious tasks are well positioned, *i.e.* with an origin on the surface of the part, and well oriented, *i.e.* normal to the surface, as is the case for every real task.

Although configurations can be generated on the fictitious tasks, the randomness of the robotics algorithms limits the performances of the base configuration generation. This is detailed below.

Let us first state what is wanted. A fictitious task $\bar{t}$ is representative of a set $T^{\bar{t}} \subseteq T$ of real tasks. The configurations generated to reach $\bar{t}$ should thus allow us to reach all the real tasks $t \in T^{\bar{t}}$. Most importantly here : the origin $O_{\bar{t}}$ of the fictitious task is the average of the origins of the real tasks $t \in T^{\bar{t}}$ (up to the projection). To reach all $t \in T^{\bar{t}}$, the configurations on $\bar{t}$ should thus be *in front of* $O_{\bar{t}}$. By this we mean that the base should be close to $O_{\bar{t}}$.

Now, because the robotics algorithms are not aware of our underlying goal when generating configurations on the fictitious tasks, these configurations are often not as wanted. Indeed, the algorithm first generates a random base configuration, and then checks (through inverse kinematics) if there is a full configuration able to reach the task from that position. When such a configuration is found, the algorithm stops and returns it. Because of the randomness of the base configuration, it is not rare to obtain a configuration on $\bar{t}$ with the base far from $O_{\bar{t}}$.

One way to cope with this issue is to force the configuration to have a certain arm position, *i.e.* a certain value for every joint, or at least every joint within a given range. Since finding a configuration is done by solving a non-linear problem, this can be done by putting the associated constraints into the objective function as a penalty. Another solution would be to build a hierarchical solver. The latter would first try to find a configuration as is done now, and in a second phase modify the solution to push the joints into the enforced ranges, while maintaining respect of the constraints of the first phase.

7.3 GTSP solver

Generating a reduced amount of configurations in order to decrease the size of the GTSP instance does not, unfortunately, solve all of our problems.

7.3.1 Issues

As mentioned in Section 6.3, since OR-Tools is a heuristic solver, there is no guarantee that it returns an optimal solution. Recall that, from an optimal solution of the TSP instance, an optimal solution, of same cost, of the GTSP instance it is derived from can easily be retrieved. Unfortunately, a feasible solution for the TSP instance, when it is not optimal, is not necessarily feasible for the GTSP problem it is derived from. Indeed, any tour is admissible for the TSP, whereas, for the GTSP, every cluster is expected to be visited, and thus entered, only once.

A tour entering and leaving a cluster more than once, does not provide a sequence of the tasks, as is expected from a GTSP solution. Furthermore, there is no unique sequence associated to it. Retrieving a GTSP solution from such a tour is therefore not possible, but OR-Tools could very

well return such a tour.

This is illustrated in Figures 11 and 12 with a small example consisting of two tasks, A and B, and two configurations per task. The graph associated to the GTSP instance I of this example is simply the complete undirected graph over four vertices (often denoted K_4). The resulting TSP instance J is visually represented in Figure 11. The violet path is a GTSP-admissible solution and the blue path is the TSP-solution associated to it. It is clear that the TSP solution visits all vertices of every cluster before leaving it. The resulting solution is thus to perform task A in configuration A_1 before performing task B in configuration B_2.

Now, in Figure 12, the represented tour is indeed TSP-admissible, since it is a tour. Nevertheless, after visiting A_1, the tour leaves cluster A for node B_1. It thus leaves cluster A before all of its vertices have been visited. The same problem occurs with cluster B. The latter is entered through vertex B_1, but, after visiting that vertex, the path goes to A_2 rather than B_2. From this tour, one could derive GTSP solution (A_1, B_1) as well (B_1, A_2) or (A_2, B_2), which have different costs and are thus not of same quality.

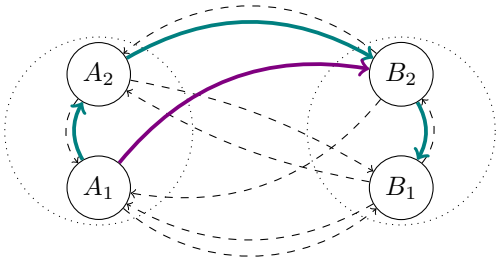

Figure 11: **Violet** : GTSP solution, **Blue** : associated TSP solution.

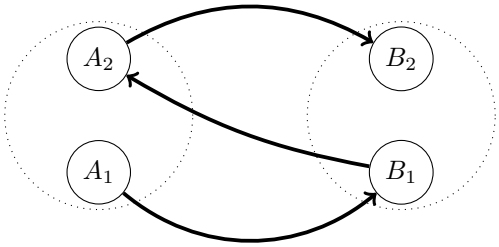

Figure 12: Admissible TSP solution that is not a valid GTSP solution.

7.3.2 Concorde

On top of the above problem, and for the sake of solution quality, it is natural to want an optimal solution of the GTSP, since the latter is already a discretisation of the original MINLP. This requires a more powerful solver such as Concorde [2]. It relies on state of the art theory of the symmetric TSP which often allows it to find an optimal solution of a problem, even though it is known to be $NP - hard$.

Concorde cannot solve the ATSP instance derived from the GTSP instance, because it is asymmetric. But, an ATSP instance can be transformed into a symmetric TSP instance [12]. Given a distance matrix D, encoding the asymmetric instance, the matrix D' encoding the associated

symmetric instance is built as follows :

$$D' = \begin{pmatrix} \infty & \overline{D}^T \\ \overline{D} & \infty \end{pmatrix} \qquad \text{with} \qquad \overline{D} = \begin{cases} D_{i,j} & \text{if} \quad i \neq j \\ -d & \text{if} \quad i = j \end{cases}$$

Where d is a sufficiently large positive constant.

This has the disadvantage of doubling the number of vertices, and thus requiring four times more space to store the associated distance matrix.

7.3.3 Implementation

When trying to implement Concorde into our solver, various problems were encountered. The main issue was the fact that Concorde was released during the 1990's and was not updated since 2003, whereas CPLEX continually evolved since then. As Concorde relies on CPLEX to compute lower bounds through the continuous relaxation of the problem, we ran into segmentation fault error messages during calls to CPLEX functions. As CPLEX is not open-source, modifying or correcting these functions was not possible.

Luckily, Concorde can also be used with the QSopt linear solver, which we eventually used. Unfortunately, this did not allow us to find the optimal solution of the TSP instance derived from the GTSP model of the problem for the airfoil instance, even after an entire day of computations. Those instances typically had 3000 nodes, as 1500 different configurations were considered to reach all the tasks. On top of that, the intermediate solutions found by the solver did not provide GTSP admissible solutions.

Using a GTSP specific solver thus seems the only way to try to solve the problem, as this would ensure that intermediate solutions are also admissible for our problem, the RTSP.

8 Conclusion

The use of the clustering algorithm ISODATA produced promising results. The versatility of the approach and the good results it showed make it a very flexible and practical method to approach RTSP instances without requiring strong hypotheses. It allows to consider generic applications where neither the robot, nor the part, are known in advance. Applying it allows to drastically reduce the size of the GTSP intance one has to solve, compared to using the naive method or an *a priori* discretisation of the workshop floor *i.e.* the set of base configurations.

The results obtained for the engine pylon, thanks to OR-Tools, show good performances of the heuristic distance measure between configurations. Nevertheless, to obtain a solution for the airfoil instance, or any other larger instance, one needs a more powerful solver, more adapted to the problem : a GTSP solver.

To extend what was achieved during this internship, an interesting direction to go into would indeed be the development of a GTSP solver. The latter could be a generic solver for the GTSP, or one adapted to the specifics of robotics. This would allow using specific heuristics, and a better integration of the robotics algorithms into the solving process.

As part of the building of such a solver, time should be put into picking the best way to model the problem. Such a model should be able to manage extra features, enabling it to model a wider range of problems. Indeed, the RTSP is very close to problems where one wants to cover a surface. This can be for painting, abrasion or simply hoovering. Another application is inspection of a surface with a camera-equipped drone. These different applications all have similarities as well as different singularities, but one would like them to fit into the same generic model.

Due to a lack of instances for the problem, it would be interesting to test the proposed approach on a wider range of problem instances, with parts of various forms, and a wider range of number of tasks. Such instances could be highly valuable for the robotics research community.

Besides promising results for the clustering, but no big results for the global approach, this internship has been very enriching and filled with important lessons.

First of all, it was a great immersion into academic research. The publication of a paper treating the exact same problem [15], with an approach close to the one we were implementing, showed me how one can differentiate its work from other publications and must keep motivation to attempt things. It also taught me to regularly take a step back from work in progress, in order not to lose the bigger picture. This allows to consider different solutions rather than locking oneself up in the one currently tried.

Due to the complexity of collision-free motion planning, and the very applied nature of the considered problem, it was necessary to dive into the specifics of robotics. This allowed me to broaden the scope of my scientific knowledge and fresh up some previously learned theorems. It also led to brainstorming sessions on problems not directly related to optimisation. This showed me that it is not always easy to integrate the specifics of another field into the tools and ideas of combinatorial optimisation. This is all the more true when one relies on already existing tools and is thus enforced a model, *e.g.* the quaternion for directions.

Another important lesson of this internship is the fact that the difficulty rarely lies where it is expected to be. Indeed, despite plenty of ideas, the implementation of those ideas was often the limiting factor. Of course, some ideas were not tractable, computationally speaking, but even things with pre-existing solutions were often not straightforward to implement. The best example is probably Concorde, but it would be dishonest to cite only that example. More details are given in the below paragraph.

This brings me to what I was able to learn from a more informatics, and thus practical, perspective.

During this internship, we wanted to take advantage of existing algorithms and programs as much as possible, believing there was a way to fit the appropriate ones together into a well performing solver for the RTSP. The accomplished work did not prove us wrong, but it highlighted the difficulty of such an approach. For a given subproblem, although there is a program solving it, it is not said that it is easy, or even possible, to fit the program or the solution into the rest of the work.

It is at my expense that I learned how difficult it can be to use an algorithm found online. The example of the implementation of ISODATA is surely relevant. Thinking it was a good idea, an *off-the-shelf* implementation was first used. although it ran trouble-free, it turned out to be subtly different from the expected algorithm. Furthermore, it was quite ill-written and did not take advantage of better performing and more robust data structures. To integrate it into the rest of our solving routine, we have thus had to completely change the data structures it relied on. Eventually, the program was even better performing, but the time put into correcting the code almost surely surpassed the time that would have been needed for an implementation *from scratch*.

Another issue, better illustrated by Concorde, is the sustainability of a code through time. First of all, code downloaded as a binary package is concealed and thus leaves no possibility for modifications to the users. Then, code downloaded from source allows to make modifications and could thus be adapted in order to fit into another software suite. Nevertheless, as is the case for Concorde, although the code is open-source, it is not open to global modifications or updates from end users, as is possible with code on GitHub. This results in code unchanged for 20 years, not compatible anymore with current versions of dependencies it has. Hence the importance to look for code that has been sustained over time, kept up-to-date when changes occurred in dependencies, and ideally has no dependencies at all.

Using an already existing software containing the robotics algorithms, and having as goal to integrate my work into that software, it was of great importance to work consistently regarding the structure of the software. Inconsistencies could lead to the entirety of my work being abandoned in the future, because it could not be sustained, understood or further developed. Sticking to an imposed strucutre was a good exercise and a good training towards working in and on bigger projects.

References

[1] S. Alatartsev, S. Stellmacher, and F. Ortmeier. Robotic Task Sequencing Problem: A Survey. *Journal of Intelligent & Robotic Systems*, 80(2):279–298, Nov. 2015.

[2] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde TSP solver, 2006.

[3] J. Canny. *The Complexity of Robot Motion Planning*. MIT Press, 1988.

[4] N. Ceccarelli, J. J. Enright, E. Frazzoli, S. J. Rasmussen, and C. J. Schumacher. Micro UAV Path Planning for Reconnaissance in Wind. In *2007 American Control Conference*, pages 5310–5315, July 2007.

[5] Y. Cheng, Markley, F. Landis, Crassidid, John L., and Yaakov, Oshman. Quaternion Averaging. 2007.

[6] C. A. Deavours. The Quaternion Calculus. *The American Mathematical Monthly*, 80(9):995–1008, Nov. 1973.

[7] D. Devaurs, T. Simeon, and J. Cortés. A multi-tree extension of the Transition-based RRT: Application to ordering-and-pathfinding problems in continuous cost spaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page 6 pages, Sept. 2014.

[8] D. Devaurs, T. Simeon, and J. Cortés. Optimal Path Planning in Complex Cost Spaces With Sampling-Based Algorithms. *IEEE Transactions on Automation Science and Engineering*, 13(2):pp.415, 2016.

[9] A. Di Placido, C. Archetti, C. Cerrone, and B. Golden. The generalized close enough traveling salesman problem. *European Journal of Operational Research*, 310(3):974–991, Nov. 2023.

[10] N. G. Hall, G. Laporte, E. Selvarajah, and C. Sriskandarajah. Scheduling and Lot Streaming in Flowshops with No-Wait in Process. *Journal of Scheduling*, 6(4):339–354, July 2003.

[11] D. Q. Huynh. Metrics for 3D Rotations: Comparison and Analysis. *Journal of Mathematical Imaging and Vision*, 35(2):155–164, Oct. 2009.

[12] R. Jonker and T. Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161–163, Nov. 1983.

[13] N. Memarsadeghi, D. M. Mount, N. S. Netanyahu, and J. Le Moigne. A FAST IMPLEMENTATION OF THE ISODATA CLUSTERING ALGORITHM. *International Journal of Computational Geometry & Applications*, 17(01):71–103, Feb. 2007.

[14] C. Möller, H. C. Schmidt, P. Koch, C. Böhlmann, S.-M. Kothe, J. Wollnack, and W. Hintze. Machining of large scaled CFRP-Parts with mobile CNC-based robotic system in aerospace industry. *Procedia Manufacturing*, 14:17–29, 2017.

[15] Q.-N. Nguyen, N. Adrian, and Q.-C. Pham. Task-Space Clustering for Mobile Manipulator Task Sequencing, May 2023.

[16] C. E. Noon and J. C. Bean. An Efficient Transformation Of The Generalized Traveling Salesman Problem. *INFOR: Information Systems and Operational Research*, 31(1):39–44, Feb. 1993.

[17] Réot, Antoine. Robotic task scheduling for industrial applications. Master's thesis, Ecole Nationale de l'Aviation Civile, Aug. 2021.

[18] S. L. Smith and F. Imeson. GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem. *Computers & Operations Research*, 87:1–19, Nov. 2017.

[19] K. Solovey, L. Janson, E. Schmerling, E. Frazzoli, and M. Pavone. Revisiting the Asymptotic Optimality of RRT. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2189–2195, May 2020.

[20] F. Suárez-Ruiz, T. S. Lembono, and Q.-C. Pham. RoboTSP - A Fast Solution to the Robotic Task Sequencing Problem, Oct. 2017.

[21] S. Yu, J. Puchinger, and S. Sun. Two-echelon urban deliveries using autonomous vehicles. *Transportation Research Part E: Logistics and Transportation Review*, 141:102018, Sept. 2020.

[22] L. Zahorán and A. Kovács. ProSeqqo: A generic solver for process planning and sequencing in industrial robotics. *Robotics and Computer-Integrated Manufacturing*, 78:102387, Dec. 2022.