Constraint Propagation on Interval Bounds for Dealing with Geometric Backtracking

Fabien Lagriffoul, Dimitar Dimitrov, Alessandro Saffiotti and Lars Karlsson
AASS Cognitive Robotic Systems Lab
Örebro University, S-70182 Örebro, Sweden
<name>.<surname>@oru.se

***Abstract*— The combination of task and motion planning presents us with a new problem that we call *geometric backtracking*. This problem arises from the fact that a single symbolic state or action may be geometrically instantiated in infinitely many ways. When a symbolic action cannot be geometrically validated, we may need to backtrack in the space of geometric configurations, which greatly increases the complexity of the whole planning process. In this paper, we address this problem using *intervals* to represent geometric configurations, and constraint propagation techniques to shrink these intervals according to the geometric constraints of the problem. After propagation, either (i) the intervals are shrunk, thus reducing the search space in which geometric backtracking may occur, or (ii) the constraints are inconsistent, indicating the non-feasibility of the sequence of actions without further effort. We illustrate our approach on scenarios in which a two-arm robot manipulates a set of objects, and report experiments that show how the search space is reduced.**

I. Introduction and Motivation

Both task and motion planning have been studied for decades [1], [2], and efficient algorithms have been developed. However, combining them together is a challenge because motion planning, which is computationally expensive, has to be *interleaved* with task planning (which is itself a hard problem). We illustrate our approach on manipulation tasks by the DLR[1] humanoid robot, Justin [3] (Fig. 1). The tasks considered are simple, for instance sorting objects or stacking cups[2]. In this kind of problem, task planning is not complicated because there are few causal relations between actions. Motion planning is not difficult either, because the workspace of the robot is not very cluttered, and we use predefined grasps. Hence, we avoid doing grasp planning. Despite these favourable conditions, some problems turn out to be intractable because of *geometric backtracking*.

During task planning, geometric configurations, which are associated to symbolic states, are maintained. When the preconditions of a symbolic action are validated, the geometric configurations associated to the current state are used in order to assess the *geometric* applicability of the action. Next, we describe in detail the geometric backtracking problem through an example, and show how it impairs the planning process. We consider a stacking task (Fig. 1). The task consists in stacking four cups at a given location (the square area on the table). Symbolically, the domain is simple: four objects, one location, and four possible actions (*grasp* and *place*, with left or right arm). Looking at Fig. 1, one can see that the right arm of the robot has almost reached full extension. Stacking the first three cups is possible, but placing the last cup on top of the pile is not possible because the kinematic constraints of the robot do not allow it.

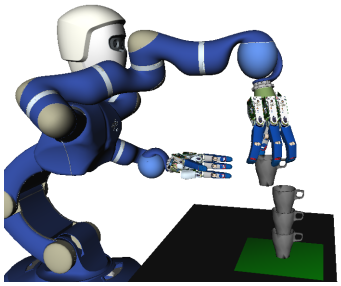

Fig. 1. Simulation of the two-arm system Justin (courtesy DLR): stacking the last cup is not possible due to kinematic constraints.

The last action is not feasible because the cup at the bottom of the pile was placed at a wrong position. If the first cup had been placed closer to the robot, the task could have been completed. Hence, the symbolic plan is actually feasible, but the geometric instance chosen for the first action does not allow the planner to complete the sequence. If the planner aborted the search at this point, it would be *incomplete*, because a solution exists but is not reached. In order to remain complete (up to some spatial resolution), the planner must try alternative geometric instances until a solution is found, or reject this last action after exhaustive search. We call this process *geometric backtracking*.

Geometric backtracking is problematic for two reasons. First, the number of geometric configurations is infinite, and remains very large even with a gross discretization. Fig. 2 describes a scheme for discretizing the space of geometric configurations:

- the *grasp* action can be done using one of the 16 precomputed grasp positions for each type of grasp;
- the *stack* action can be performed in 16 different ways (16 possible orientations for the cup) ;
- *place* action can be achieved in 256 ways (16 locations $\times$16 orientations).

Hence, for the "4 cups stacking" example, exhaustive

[1]Deutsche Zentrum für Luft-und Raumfahrt
[2]See videos with the real robot at http://www.aass.oru.se/∼fll/videos/

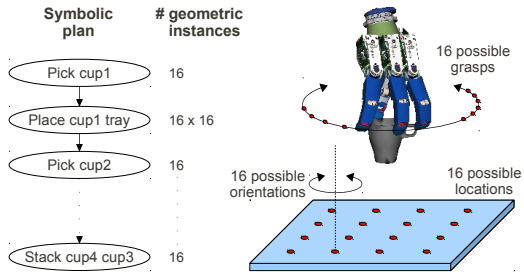

Fig. 2. An example of discretization of the space of geometric configurations.

search is infeasible (6.9×10^{10} possilities). But even on a short sequence composed of two actions, such as *pick* and *place*, the problem arises. If a specific final orientation for the object is required, then the success of the *place* action highly depends on the grasp chosen for the *pick* action. Without a proper strategy, the planner may have to try several grasps before finding one which allows it to place the cup with the desired orientation. And for each candidate grasp tried, we need to try all the 256 possible *place* actions to be sure that the sequence cannot be executed with this grasp.

This leads us to the second reason why geometric backtracking is problematic. Checking for feasibility requires calling the motion planner to ensure that the path is collision-free, which takes time (on average 100 ms for our platform and scenario). Even tough the feasibility test is sometimes much faster (when no inverse kinematic (IK) solution exists), a simple Pick and Place sequence of actions may still require a dozen seconds to be solved, and more time to be proved infeasible. This is not acceptable at the task planning level, when plenty of these sequences of actions need to be assessed.

In our view, geometric backtracking is one of the main difficulties in combining task and motion planning. In this paper, we propose an approach to tackle this problem, by introducing an intermediate layer between task planning and motion planning: symbolic actions are not directly instantiated into geometric configurations. Instead, a set of constraints is extracted from the symbolic actions and from the geometric model of the robot. These constraints are used to prune out geometric configurations that can never be a part of a solution. In this way, geometric backtracking is not avoided, but significantly reduced. Moreover, when the set of constraints is inconsistent, we know that the sequence of actions is infeasible without backtracking at all.

II. Related work

To our knowledge, aSyMov [4] was the first planner to combine task and motion planning. Motion planning is done by composition of probabilistic roadmaps (PRMs) [5], while task planning is based on an A*-like search algorithm. It uses a hybrid state representation: a classical symbolic state, together with its geometric counterpart. During search, the algorithm alternates between finding a plan using the current roadmaps, or adding nodes to the roadmaps in order to refine its geometric knowledge of the world. Each state has a list of candidate geometric configurations, some of which are validated when they are known to be reachable from the previous state. The validation procedure tries to back-trace through valid configurations until the initial configuration is reached. If this is not possible, the algorithm may have (in the worst case) to check for all collision-free paths between all the candidate configurations of each successive state. The search then becomes exponential in the number of robots and objects. aSyMov performs well when the problem is constrained at the task level, but is less efficient on pure geometric problems (cf. the forklifts and boxes experiments in [4]).

In [6], an extension of the planning domain description language (PDDL) is proposed. The operators are augmented with a condition-checker and an effect-applicator, which causes calls to external specialized geometric reasoners during task planning. The grasps are predefined. They use numerical fluents to represent transformation matrices, robot configurations, and poses of objects. It is not clear though how predicates defining continuous placements on surfaces are dealt with, e.g., *On cup table*, but it seems likely that predefined locations are used. Backtracking occurs at the task level within a set of predefined discrete locations, and the question of geometric backtracking does not arise.

In [7], a hierarchical task network (HTN) approach is proposed in which the complexity of hybrid planning is indirectly tackled by decreasing the search horizon. The subtasks obtained from the task decomposition are executed as soon as primitive actions are reached. This allows to re-plan "in the now": a useful feature in dynamic or uncertain environments. On the other hand, this prevents from projecting the geometric consequences of actions far into the future, which implies that geometrically hard problems would require physical backtracking.

Combination of task and motion planning is addressed in other works, e.g., [8], [9], [10], [11], [12], but geometric backtracking is not explicitly identified and addressed. In [12], a sampling technique is proposed in order to detect infeasible motions earlier in PRM motion planning, which relates to geometric backtracking. This cannot be applied to our domain though, because we consider both arms as separate robots. Hence, the configuration of obstacles (mainly the other arm) is changing at each motion planning query, which is inappropriate for PRM techniques.

III. Generating the constraints

Our approach consists of introducing an intermediate step between task planning and motion planning, in which a set of constraints is generated. These constraints are *automatically* generated from the sequence of symbolic actions currently explored by the task planner, and from the geometric properties of the robot. These constraints express what can or cannot be geometrically achieved. Intuitively, such a set of constraint can capture that *"if object o_2 is picked from its current position with the right arm and placed*

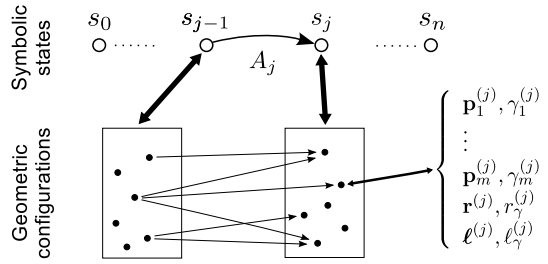

Fig. 3. Representation of symbolic states and stages of operation. A symbolic state can be instantiated into many geometric configurations.

on top of object o_3, the maximum achievable clockwise rotation is 65 degrees". Such constraints can drastically reduce search in the space of geometric configurations, because they eliminate many configurations resulting from the discretization process that cannot be part of a feasible sequence. For clarity in this paper, we do not describe the whole planning algorithm, but only how a single sequence of actions is handled. The rest of the section introduces some notation to represent actions and states, and how constraints are generated.

A. Representing actions and states

For conciseness of notation, in some cases, we will use $\boldsymbol{x} = (x_1, \ldots, x_n)$ to denote the elements of a column vector $\boldsymbol{x}$. All coordinates are expressed in the world frame.

We consider m rigid objects. The i^{th} object is denoted by o_i, $i \in \{1, \ldots, m\}$, and its pose is represented using $(\boldsymbol{p}_i, \gamma_i)_{\mathbf{k}}$, where $\boldsymbol{p}_i = (x_i, y_i, z_i) \in \mathbb{R}^3$ represents its position, and $\gamma_i \in \mathbb{R}$ the angle rotation about a unit axis $\boldsymbol{k} \in \mathbb{R}^3$. It is assumed here that all objects have the same axis of rotation $\boldsymbol{k}$, so $\boldsymbol{k}$ will be omitted in the notation.

We consider a sequence of n symbolic actions (e.g., *place o_i table*), in which the j^{th} action is denoted by A_j, $j \in \{1, \ldots, n\}$ (see Fig. 3). The position of o_i at symbolic state s_j (i.e., after action A_j has been completed) will be denoted by $\boldsymbol{p}_i^{(j)}$, and its orientation, by $\gamma_i^{(j)}$:

$$\left(\boldsymbol{p}_i^{(j-1)}, \gamma_i^{(j-1)}\right) \xrightarrow{A_j} \left(\boldsymbol{p}_i^{(j)}, \gamma_i^{(j)}\right).$$

Hence, a sequence of n actions $\langle A_1, \ldots, A_n \rangle$ results in a sequence of poses for all m objects

$$\langle A_1, \ldots, A_n \rangle \rightarrow \left\{ \begin{array}{c} \langle \boldsymbol{p}_1^{(0)}, \gamma_1^{(0)}, \ldots, \boldsymbol{p}_1^{(n)}, \gamma_1^{(n)} \rangle \\ \vdots \\ \langle \boldsymbol{p}_m^{(0)}, \gamma_m^{(0)}, \ldots, \boldsymbol{p}_m^{(n)}, \gamma_m^{(n)} \rangle \end{array} \right\}.$$

Note that $(\boldsymbol{p}_i^{(j-1)}, \gamma_i^{(j-1)}) = (\boldsymbol{p}_i^{(j)}, \gamma_i^{(j)})$ when object o_i is not manipulated.

Symbolic actions on objects are meant to be applied using the robotic system Justin [3]. At the j^{th} symbolic state, the pose of the tool center point (TCP) of the right manipulator of Justin will be denoted by $(\boldsymbol{r}^{(j)}, r_\gamma^{(j)})_{\mathbf{k}}$, where $\boldsymbol{r}^{(j)} = (r_x^{(j)}, r_y^{(j)}, r_z^{(j)}), \in \mathbb{R}^3$ denotes Cartesian position, and $r_\gamma^{(j)} \in \mathbb{R}$ is an angle about the axis of rotation $\boldsymbol{k}$. We assume that, at each symbolic state, the axis of rotation of the end-effector is parallel to the axis of rotation of the object to be grasped or placed[3]. In a similar way, we define the pose of the TCP of the left manipulator as $(\boldsymbol{\ell}^{(j)}, \ell_\gamma^{(j)})_{\mathbf{k}}$.

For example, consider the following sequence of *grasp* (G) and *place* (P) actions:

states	s_1	s_2	s_3	s_4	s_5	s_6
actions	A_1	A_2	A_3	A_4	A_5	A_6
object o_1	$G_1^{(1)}$	$P_1^{(2)}$	·	·	·	·
object o_2	·	·	$G_2^{(3)}$	·	$P_2^{(5)}$	·
object o_3	·	·	·	$G_3^{(4)}$	·	$P_3^{(6)}$
right hand	✓	✓	✓	·	✓	·
left hand	·	·	·	✓	·	✓

As a result of the first symbolic action $A_1 = G_1^{(1)}$, o_1 is grasped by the right hand, resulting in the first symbolic state. Note that $(\boldsymbol{p}_1^{(1)}, \gamma_1^{(1)}) = (\boldsymbol{p}_1^{(0)}, \gamma_1^{(0)})$ as the object is not yet moved. The second action $A_2 = P_1^{(2)}$ places o_1 at $(\boldsymbol{p}_1^{(2)}, \gamma_1^{(2)}) \neq (\boldsymbol{p}_1^{(1)}, \gamma_1^{(1)})$ since the object has been moved. During the remaining stages o_1 is not acted upon, and hence its pose remains unchanged. At the 4^{th} symbolic state, Justin has grasped o_2 and o_3 in its right and left hand, respectively, and placed them during actions A_5 and A_6, at positions $\boldsymbol{p}_2^{(5)}$ and $\boldsymbol{p}_3^{(6)}$, with orientation $\gamma_2^{(5)}$ and $\gamma_3^{(6)}$, respectively.

With each action and state we associate constraints. The following constraints are given assuming that A_j is performed using the right manipulator. In a similar way we can define constraints when the left manipulator is used.

B. Placement constraints $\mathcal{C}_P$

These constraints are extracted from the symbolic states. For instance, if a symbolic state contains the predicate *On Cup Tray*, one can formulate a set of inequality constraints on the x and y coordinates of the cup, expressing the fact that the cup belongs to the rectangle defined by the tray. More generally, we can express such constraints after a *place* action at a state s_j for arbitrary regions, bounding them with a polyhedron $\mathcal{P}_i^{(j)}$. The constraint for an object o_i located in such region can be written as

$$\boldsymbol{M}_i^{(j)} \boldsymbol{p}_i^{(j)} \leq \boldsymbol{b}_i^{(j)},$$

where $\boldsymbol{M}_i^{(j)} \in \mathbb{R}^{n_{ij} \times 4}$, $\boldsymbol{b}_i^{(j)} \in \mathbb{R}^{n_{ij} \times 1}$ define n_{ij} linear inequality constraints specifying the region where the object can be. Similar constraints can be extracted from other predicates (i.e., *In, Left, Right, Under, StackedOn,...*). Placement constraints also include constraints on desired positions and orientations of objects: the initial and final poses are expressed using such constraints.

C. Transfer constraints $\mathcal{C}_T$

These constraints reflect the fact that when an object is manipulated, it undergoes the same translation and rotation as the TCP of the robot. This occurs during a *place* action.

[3]Hence, we limit our analysis to grasps such as top-grasps, or side-grasps

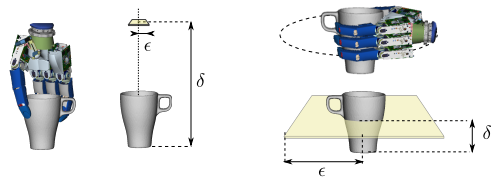

Fig. 4. The grasp constraints for a top-grasp (left) and a general-grasp (right). In any case, the position of the TCP relative to the object can be bounded by a polyhedron.

For an object o_i undergoing a *place* action resulting in state s_j we can formulate the following constraints for a top-grasp:

$$\boldsymbol{p}_i^{(q_{ij})} - \boldsymbol{p}_i^{(j)} = \boldsymbol{r}^{(q_{ij})} - \boldsymbol{r}^{(j)}$$
$$\gamma_i^{(q_{ij})} - \gamma_i^{(j)} = r_\gamma^{(q_{ij})} - r_\gamma^{(j)},$$

where q_{ij} denotes the state index during which o_i was grasped prior to its release during state s_j (for instance in the table above, $q_{12} = 1$, $q_{25} = 3$, $q_{36} = 4$). Note that the constraint on orientations introduces an additional difficulty due to the periodicity of angular values. Hence, we have to consider different cases depending on how a *place* action is performed (clockwise or counterclockwise), but this is out of the scope of this paper and does not change our approach to the problem.

D. Grasp constraints $\mathcal{C}_G$

These constraints are formulated each time an object is grasped. They represent the possible relative positions of the TCP with respect to the object when the object is grasped or released. In our scenario, we use only top-grasps, but such constraints can be formulated for any type of grasp.

For a top-grasp, the TCP is situated exactly above the object (see Fig. 4), which can be formulated as follows for an object o_i, after the *grasp* action A_j:

$$\boldsymbol{r}^{(j)} = \boldsymbol{p}_i^{(j)} + \delta_k \boldsymbol{k},$$

where δ_k is a constant offset corresponding to the distance between the TCP and the object when the top-grasp is performed. For a side-grasp, the TCP is not aligned with the object along $\boldsymbol{k}$, but located on a circle around the object. In the general case, we can formulate constraints as linear inequality constraints limiting the position of the TCP relative to the object (see Fig. 4).

E. Manipulator constraints $\mathcal{C}_M$

These constraints are the core of our approach. They are very important for manipulation tasks because they express the relationship between the position of the TCP $\boldsymbol{r}^{(j)}$ in the workspace and its possible range of rotation. This relationship is non-linear and complex to compute. We approximate it using linear constraints.

In order to find a good linear approximation of these constraints, we compute two maps off-line, using a similar procedure to [13]. The workspace of the robot is discretized into a 3-dimensional grid, and for each cell, the existence of an IK solution is tested for a large set of possible rotations of the TCP around $\boldsymbol{k}$ (This procedure is done for both arms, and each type of grasp). From this data, we can build two maps γ_{min} and γ_{max}, which respectively associate the position $\boldsymbol{r}$ of the TCP to a lower and upper bound on r_γ (see Fig. 5).

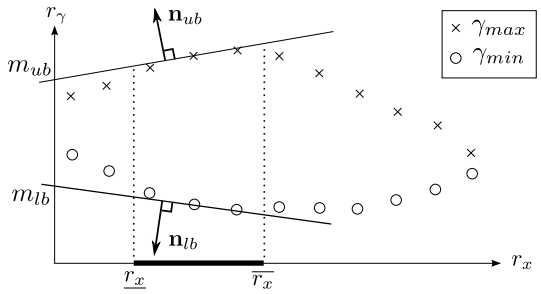

Fig. 5. Schematic 2-d view of the 4-dimensional linear outer approximations of γ_{min} and γ_{max} by the functions h_{min} and h_{max}

$$\gamma_{min} : (r_x, r_y, r_z) \mapsto r_{\gamma min}$$
$$\gamma_{max} : (r_x, r_y, r_z) \mapsto r_{\gamma max}.$$

$r_{\gamma min}$ and $r_{\gamma max}$ are computed such that if r_γ accepts an IK solution, then r_γ belongs to the interval $[r_{\gamma min}, r_{\gamma max}]$. In order to extract linear constraints from these maps, we define two functions:

$$h_{max}(\underline{\boldsymbol{r}}^{(j)}, \overline{\boldsymbol{r}}^{(j)}) \rightarrow (\boldsymbol{n}_{ub}^{(j)}, m_{ub}^{(j)})$$
$$h_{min}(\underline{\boldsymbol{r}}^{(j)}, \overline{\boldsymbol{r}}^{(j)}) \rightarrow (\boldsymbol{n}_{lb}^{(j)}, m_{lb}^{(j)}),$$

where $\underline{\boldsymbol{r}}^{(j)}$ and $\overline{\boldsymbol{r}}^{(j)}$ are respectively a lower and upper bound for the variables $r_x^{(j)}, r_y^{(j)}$ and $r_z^{(j)}$, i.e., a region of space for which we want to approximate these constraints. In the next section, we explain how these bounds are computed, but essentially, they come from the propagation of other constraints. These bounds ($\underline{r_x}$ and $\overline{r_x}$ on Fig. 5) are used to select a subset of points in γ_{max} and γ_{min}, from which a linear regression is used in order to identify the unit normals $(\boldsymbol{n}_{ub}^{(j)}, \boldsymbol{n}_{lb}^{(j)})$ and offsets $(m_{ub}^{(j)}, m_{lb}^{(j)})$ of two bounding hyperplanes (see Fig. 5). Using a subset of points allows us to get a tighter linear approximation. Then, these parameters are used to formulate the manipulator constraints, which give the range of possible rotation of the TCP during an action A_j:

$$\begin{bmatrix} \boldsymbol{n}_{lb}^{(j)\mathrm{T}} & m_{lb} \end{bmatrix} \begin{bmatrix} \boldsymbol{r}^{(j)} \\ 1 \end{bmatrix} \leq r_\gamma^{(j)} \leq \begin{bmatrix} \boldsymbol{n}_{ub}^{(j)\mathrm{T}} & m_{ub} \end{bmatrix} \begin{bmatrix} \boldsymbol{r}^{(j)} \\ 1 \end{bmatrix}.$$

This constraint is useful in two ways:

- For a given region of space, it provides bounds on the possible top-grasps that can be achieved.
- For a given set of top-grasps, it provides a region of space where these grasps can be achieved.

Finally, we define the vector of all the variables of the problem:

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_N),$$

to which we associate a domain

$$\mathcal{D} = \langle [\underline{v_1}, \overline{v_1}], [\underline{v_2}, \overline{v_2}], \ldots, [\underline{v_N}, \overline{v_N}] \rangle,$$

where each variable v_i has associated bounds $[\underline{v_i}, \overline{v_i}]$. The set of all constraints of the problem

$$\mathcal{C} = \{\mathcal{C}_P^{(j)}, \mathcal{C}_T^{(j)}, \mathcal{C}_G^{(j)}, \mathcal{C}_M^{(j)}\},\ j \in \{1, \ldots, n\}$$

can be expressed as

$$\boldsymbol{P}\boldsymbol{v} \leq \boldsymbol{d} \tag{1}$$

$$\boldsymbol{Q}\boldsymbol{v} = \boldsymbol{e}, \tag{2}$$

where (1) and (2) represent respectively the inequality and equality constraints of our problem. In addition, we define a *geometric configuration* (see Fig. 3) as a set of values representing the poses of all objects, and the poses of both TCPs:

$$c = \{\boldsymbol{p_1}, \ldots, \boldsymbol{p_m}, \boldsymbol{r}, \boldsymbol{\ell}\}. \tag{3}$$

IV. Using Constraints to Compute Intervals

The geometric constraints of the problem are formulated with a set of linear inequalities (1) and equalities (2). The manipulator constraints $\mathcal{C}_M$ have been formulated in terms of lower and upper bounds on the actual capabilities of the manipulator. Consequently, the set of constraints is *conservative*, i.e., if a solution exists, it must belong to the feasible set defined by the constraints (conversely, if the constraints result in an empty feasible set, the problem has no solution). Note, however, that

- we still have to search the feasible set for a sequence of configurations which solves the problem;
- we still have to do motion planning to find collision-free paths connecting grasp and release positions.

For these reasons, instead of searching for a single solution, we use the constraints to tighten the bounds of a set of intervals which contain all the solutions to the problem.

A. Narrowing intervals

Algorithm 1: FilterDomain

Function **FilterDomain**$(\mathcal{D}, \mathcal{C})$

input : $\mathcal{D}$: a domain
C: a set of linear constraints

1 ϵ = minimal domain reduction
2 $\mathcal{D}' = \mathcal{D}$
3 **repeat**
4 $\quad \mathcal{D} = \mathcal{D}'$
5 $\quad UpdateManipulatorConstraints(C, \mathcal{D})$
6 $\quad$ **for** $i \leftarrow 1$ **to** N **do**
7 $\quad\quad$ minimize v_i, subject to $\boldsymbol{P}\boldsymbol{v} \leq \boldsymbol{d}, \boldsymbol{Q}\boldsymbol{v} = \boldsymbol{e}$
8 $\quad\quad \underline{v_i}' \leftarrow \max(\underline{v_i}, v_i^\star)$
9 $\quad\quad$ maximize v_i, subject to $\boldsymbol{P}\boldsymbol{v} \leq \boldsymbol{d}, \boldsymbol{Q}\boldsymbol{v} = \boldsymbol{e}$
10 $\quad\quad \overline{v_i}' \leftarrow \min(\overline{v_i}, v_i^\star)$
11 **until** $Dist(\mathcal{D}, \mathcal{D}') \leq \epsilon$ **or** $\mathcal{D}' = \emptyset$
12 **return** $\mathcal{D}'$

The bounds of the intervals are computed using Algorithm 1, a *global* filtering algorithm (adapted from [14]) which converges rapidly and detects inconsistency at the first iteration. This algorithm solves several linear programs (LP) in order to find the minimum (resp. maximum) value $v_i^\star$ of each variable v_i. $v_i^\star$ is then used to update the lower (resp. upper) bound of v_i (lines 8 and 10). The values are updated in a temporary copy of the domain $\mathcal{D}' = \langle[\underline{v_1}', \overline{v_1}'], [\underline{v_2}', \overline{v_2}'], \ldots, [\underline{v_N}', \overline{v_N}']\rangle$, which is used in order to measure how much the intervals have shrunk after each iteration. This is done by the $Dist$ function (line 11), which returns the average of the differences between upper and lower bounds in $\mathcal{D}$ and $\mathcal{D}'$. The process is repeated until the domains do not change more than a predefined ϵ value. The result is a domain in which the intervals are narrowed with respect to the constraints, or $\emptyset$ if an inconsistency is detected during the resolution of a LP.

We have modified the original algorithm by adding the function $UpdateManipulatorConstraints(C, \mathcal{D})$ in the main loop (line 5). Indeed, after each iteration, the intervals may shrink. If the intervals representing the TCP positions are reduced, it is meaningful to refine the manipulator constraints using the functions h_{min} and h_{max} in order to get a tighter linear approximation of the real problem.

Refining the manipulator constraints while filtering the domains is a very efficient process. Let us illustrate this with a numerical example for a pick action. Initially, the problem consists of four variables representing the position of the object "cup" located at $(0.6, 0.25, 0.1)$ with orientation 0.

$$\boldsymbol{v} = (x_{cup}^{(0)},\ y_{cup}^{(0)},\ z_{cup}^{(0)},\ \gamma_{cup}^{(0)})$$
$$\mathcal{D} = \langle[0.6, 0.6],\ [0.25, 0.25],\ [0.1, 0.1],\ [0, 0]\rangle.$$

The lower bounds are equal to the upper bounds because the values of the variables are determined. The pick action leads us to the creation of 4 new variables for the TCP, which are initially assigned arbitrarily large intervals:

$$\boldsymbol{v} = (x_{cup}^{(0)},\ y_{cup}^{(0)},\ z_{cup}^{(0)},\ \gamma_{cup}^{(0)},\ r_x^{(1)},\ r_y^{(1)},\ r_z^{(1)},\ r_\gamma^{(1)})$$
$$\mathcal{D} = \langle[0.6, 0.6],\ [0.25, 0.25],\ [0.10, 0.10],\ [0, 0],$$
$$[-10, 10],\ [-10, 10],\ [-10, 10],\ [-\pi, \pi]\rangle.$$

A pick action also generates grasp constraints $\mathcal{C}_G$ and manipulator constraints $\mathcal{C}_M$. We use $\boldsymbol{k} = (0, 0, 1)$ and $\delta_k = 0.34$:

$$r_x^{(1)} = x_{cup}^{(0)}$$
$$r_y^{(1)} = y_{cup}^{(0)} \qquad -0.70 \ \leq\ r_\gamma\ \leq\ 2.36$$
$$r_z^{(1)} = z_{cup}^{(0)} + 0.34 \qquad \text{(using } h_{min} \text{ and } h_{max}\text{)}.$$

After applying the function $FilterDomain$, $\mathcal{D}$ becomes:

$$\mathcal{D} = \langle[0.6, 0.6],\ [0.25, 0.25],\ [0.10, 0.10],\ [0, 0],$$
$$[0.6, 0.6],\ [0.25, 0.25],\ [0.44, 0.44],\ [-0.70, 2.36]\rangle.$$

The grasp constraints have propagated the values of the position of the cup to the position of the TCP. In the second iteration, the bounds on the orientation of the TCP $r_\gamma^{(1)}$ have been updated with the linear approximations of the maps, but since the domain of the TCP is now a single point, these bounds represent the possible rotation of the TCP at this

point. Hence, in order to pick the cup, the orientation of the top-grasp must be chosen between -0.70 and 2.36 radians.

This constraint propagation process is interesting for longer sequences of actions, because it allows us to propagate the consequences of early choices until the final actions. It could for instance solve the stacking problem described in the introduction, and even give an approximation of a region on the table which is appropriate for placing the first cup.

B. Narrowing intervals during search

In order to find a solution, we use a basic depth-first-search algorithm, endowed with a pruning step (see algorithm 2: $SearchAndFilter$ (SAF)). Geometric instances of configurations are not chosen arbitrarily, but such that the variables representing them (see definition (3)) belong to their respective intervals. This is the first level of pruning. But after an action has been chosen (e.g., to place the cup at position $(0.7, -0.25, 0.1)$ with $\gamma = \pi/2$), the variables representing this choice are assigned fixed values, so the corresponding intervals can be reduced to single points (i.e., for a variable v_i, $\underline{v_i} = \overline{v_i}$). Then, we can filter the domain *again* in order to propagate this choice to other variables through the constraints. The other intervals will be shrunk accordingly, which will reduce even more the possibilities for further actions. This process is repeated each time an action is chosen, so that intervals are shrunk as the search progresses.

Algorithm 2: SearchAndFilter

```
Function SearchAndFilter(c_1, Seq, D)
input : c_1: a geometric configuration
        Seq: a sequence of symbolic actions
        D: a domain
1  if Seq = ⟨⟩ then return c_1
2  Action = Seq.head
3  Rest = Seq.tail
4  foreach A_i ∈ geometricInstanceOf (Action) do
5      c_2 = getSuccesorConf(c_1, A_i)
6      if c_2 ∈ D then
7          D' = assignValues(D, c_2)
8          D' = filterDomain(D')
9          if D' ≠ ∅ then
10             feasible = pathPlanning(c_1, c_2)
11             if feasible then
12                 s = SearchAndFilter(c_2, Rest, D')
13                 if s ≠ false then
14                     return ⟨c_2, s⟩
15 return false
```

Algorithm 2 is initially called with the initial geometric configuration, the sequence of symbolic actions, and the initial domain filtered according to the constraints of the problem. An action A_i is chosen among the possible geometric instances of $Action$ (e.g., 16 for Pick, 256 for Place). c_2 is the result of applying A_i to c_1. If this configuration belongs to the domain, we apply the strategy described above, that assigns the values to the domain and filters it again (lines 7-8). If no inconsistency appears, the motion planning algorithm is called to check if a collision-free path exists to reach c_2. If a path exists, the function is recursively called on c_2 with the remaining actions and the shrunk domain $\mathcal{D}'$, otherwise the next action A_i is tried. If all the actions fail, the function returns $false$ to the calling function via the return statement line 15. If a final configuration is reached (line 1), the solution is incrementally built (line 14) and returned to the main calling function. The result is a list of geometric configurations and paths which are used to execute the final plan (paths are smoothed after a plan is found, see end of section V.A).

C. Detecting inconsistency and pruning

One of the main problems of geometric backtracking is when no geometric instantiation of the action sequence exists. This happens often during task planning, because no geometric information is used. For instance, the task planner may try a sequence in which the right arm of the robot grasps an object situated on the left side. In the worst case, for such a sequence, all the space of configurations has to be searched in order to discover that it is infeasible, which may be computationally expensive. The only solution to avoid this is to impose a time limit on the backtracking process. Unfortunately by doing this, completeness is lost for cases when the problem *is* feasible.

On the other hand in our approach, inconsistency can be detected *before* entering the backtracking procedure, while we filter the initial domain according to the constraints of the problem. This is more efficient since no search is required. Inconsistency can also be exploited *during* search in order to prune out a whole branch of the search tree. This happens when the problem is initially consistent, and at some point in the search, an action is chosen that makes the problem inconsistent. This will be detected during filtering (line 8-9 in the Algorithm 2). Then, we do not need to search further with this action sequence, and can try another action.

V. Experimental results

A. Experimental setup

Geometric backtracking might occur while evaluating the feasibility of a sequence of symbolic actions. What a task planner does is essentially to evaluate many of these sequences. Hence, we evaluated our approach by evaluating single sequences of actions. We compare our algorithm SAF to a standard depth-first-search (DFS) procedure, i.e., SAF without the filtering process (lines 6 to 9), and without using the argument $\mathcal{D}'$ at line 12. We compare our algorithm against DFS, because DFS is equivalent to the strategies used in similar work (see Section II), i.e., a non-informed backtracking search.

We use a simulation environment provided by DLR for the robotic platform Justin [3]. Simulation is more suited for this kind of experiments, but similar tasks have been

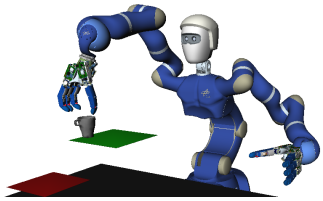

Fig. 6. One arm constrained regrasping in Experiment 1

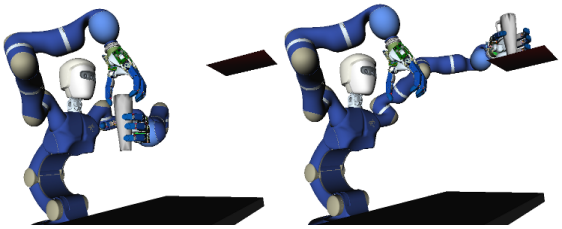

Fig. 7. Hand over in Experiment 2

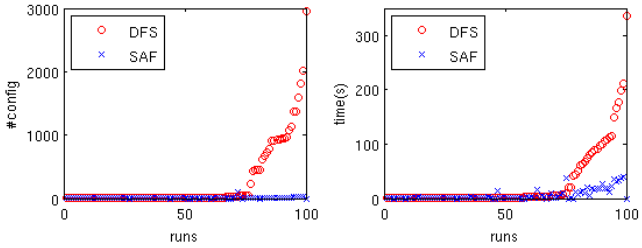

Fig. 8. Results for Experiment 1: #config on the left, time on the right.

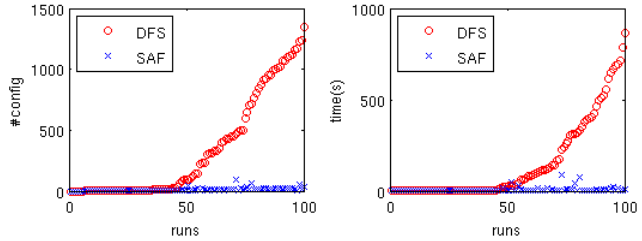

Fig. 9. Results for Experiment 2: A similar trend is observed.

successfully executed on the real robot (see [15]). Justin is a humanoid robot with two arms with 7 DoF each, and two dexterous hands. The robot is situated in front of a table, on which are placed 30 $cm \times 30$ cm trays/shelves, and some cups that can be manipulated. The space is discretized with a resolution of 5 cm for the trays and 15 cm for the table, and orientations with an angular value of $\pi/8$. (which means 36 possible positions on trays, 32 on the table, and 16 possible orientations). We evaluated our approach on two different sequences of actions:

In **Experiment 1** (see Fig. 6), we used one object, a sequence of four actions, and only the right arm:

- *Pick right top cup1*
- *Place right cup1 tray1*
- *Pick right top cup1*
- *Place cup1 tray2,*

where *tray1* can be randomly situated from 10 cm to 40 cm above the surface of the table. We also imposed a constraint on the final orientation of the cup ($\gamma_1^{(4)} = \pi$).

In **Experiment 2**, both arms and top/side grasps where used:

- *Pick right top cup2*
- *Place right cup2 regrasp-region*
- *Pick left side cup2*
- *Place left cup2 shelf1,*

where the cup is initially randomly located on the right side of the table, and *shelf1* on the left side, at a high position, with random variation. A constraint was imposed on the final orientation of the cup $\gamma_1^{(5)}$, which was also randomly chosen.

For all experiments, we have measured the number of geometric configurations explored (#config), and the search time (time). Both algorithms were run on the same problems, and 100 runs were conducted. Linear programs were solved with Gurobi[16], and motion planning with standard rapidly exploring random tree (RRT). A raw trajectory is computed to assess reachability during search. The computation of the final smooth trajectory, which is used for actual execution of the plan, is not shown in the results (it takes 3 to 10 s for each action). The algorithms are implemented in java, and run on a MacBook Pro (Intel Core i7 dual-core 2.66 GHz).

B. Results

The results for Experiment 1 and 2 are shown on Fig. 8 and Fig. 9 respectively. The horizontal axis represent the runs, sorted by increasing number of configurations explored (resp. time) by the DFS algorithm. Hence, the horizontal axis represents the complexity of the problem measured "ex post facto" by DFS. The flat part of the curve represents "simple" cases, for which both algorithms explore a small number of configurations. These problems are solved in 1.5 s by SAF, and 0.75 s by DFS, that is 0.75 s overhead for computing intervals (with not particularly optimized implementation). Computing intervals clearly pays off for more "difficult" cases, i.e., when geometric backtracking is required. On average, the overhead is largely compensated by the gain observed in "difficult" cases.

In Experiment 1, geometric backtracking is necessary when *tray1* is high or ill-placed. The possibilities for re-grasping from there are then limited (see Fig. 6), which may cause the last *place* action to fail. Hence, the choice in the intermediate position and orientation of the cup is important to complete the sequence. Similarly in Experiment 2, a position for the cup has to be found where both arms can reach it, and the high position of *shelf1* limits the possible orientations achievable by a side-grasp. Hence, each action has to be carefully chosen in order to achieve the desired orientation while respecting kinematic constraints.

In 75% of the cases for experiment 1 (resp. 50% for experiment 2), the task does not require backtracking, and both algorithms perform well. But in the remaining cases, the time spent by DFS explodes because it arbitrarily selects the configurations, which is often a wrong choice in "difficult

cases". This entails backtracking, and increases exponentially the number of configurations explored. On the other hand, SAF takes advantage of the constraints to choose a suitable intermediate position, which reduces backtracking. This is clear on the longest runs where the number of configurations visited explodes for DFS. In terms of time, the trend is similar, since the RRT planner is called for each configuration explored. In Experiment 1 however, the time for SAF increases slightly more than the number of configurations explored. This is because in Experiment 1, difficult cases are also complicated in terms of cluttering of the scene, because *tray1* acts as an obstacle. Hence, the RRT planner takes longer time to compute each path.

For the proposed scenarios, in rare cases ($\approx$1%), the task is not feasible due to complicated interaction of the constraints (i.e., because of peculiar initial position and orientation of the cup, there exists no combination of grasps that can achieves the final orientation). In these cases, SAF quickly detects inconsistency in the constraints (100 ms), while DFS needs to search through the entire space of configurations (hours). Apart from such peculiar cases, Experiment 2 shows that a realistic task (placing an object on a shelf at a given orientation, with regrasping) involves a non negligible amount of geometric backtracking in 50% of the cases, which accounts for the interest of the proposed technique.

VI. Conclusion

The main contribution of this paper is twofold. First, we have identified *geometric backtracking* as one of the major sources of complexity when combining task and motion planning. While new approaches that combine task and motion planning are being increasingly proposed, to the best of our knowledge the problem of geometric backtracking has not been explicitly identified and addressed until now. The second contribution is a method for dealing with geometric backtracking. The core idea is to extract a set of linear constraints from the symbolic plan and the kinematics of the robot, and to apply linear programming techniques to compute intervals reducing the space of geometric configurations, which avoids unnecessary calls to the motion planner. The proposed technique is efficient for geometrically constrained tasks, and tasks in which action dependencies require backtracking far in the sequence. Another advantage of the proposed approach is to quickly detect unfeasible cases, which is an important feature when used in combination with a task planner.

In this paper, we used a single axis of rotation in order demonstrate our approach. Ongoing work shows that the number of axes can be increased without impairing the performance (more maps need to be computed, but this is done offline), which increases the range of possible manipulations. An open issue of our approach is how to model linear grasp constraints for objects with arbitrary shapes. More work needs to be done in this direction. Regarding scalability, the number of the LPs to solve linearly depends on the number of actions. We could cope with longer sequences of actions by splitting them into sub-sequences where actions are interdependent (i.e., the same object is manipulated).

In this work, we have used intervals to deal specifically with geometric constraints. Intervals are a compact representation which allows us to reason about space without being affected by the curse of dimensionality caused by the discretization process. Therefore, we believe that intervals are appropriate for bridging the gap between task and motion planning in general.

Acknowledgments

This work was partially supported by EU FP7 project "Generalizing Robot Manipulation Tasks" (GeRT, contract number 248273). We would like to thank in particular Florian Schmidt from the Robotics and Mechatronics Center of DLR, for his advice, and for the functionalities he developed in Justin's simulator, which made this work possible.

References

[1] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*, 2004.

[2] S. LaValle, *Planning Algorithms*, 2006.

[3] C. Ott, O. Eiberger, W. Friedl, B. Bäuml, U. Hillenbr, C. Borst, A. Albu-schäffer, B. Brunner, H. Hirschmüller, S. Kielhöfer, R. Konietschke, T. Wimböck, F. Zacharias, and G. Hirzinger, "A humanoid two-arm system for dexterous manipulation," in *2006 IEEE Int. Conf. on Humanoid Robots*, 2006, pp. 276–283.

[4] S. Cambon, R. Alami, and F. Gravot, "A hybrid approach to intricate motion, manipulation and task planning," *Int. J. Rob. Res.*, vol. 28, no. 1, pp. 104–126, 2009.

[5] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in *IEEE Int. Conf. on Robotics and Automation*, 1996, pp. 566–580.

[6] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, "Semantic attachments for domain-independent planning systems," in *Proc. of the 19th Int. Conf. on Automated Planning and Scheduling (ICAPS09)*, 2009, pp. 114–122.

[7] L. P. Kaelbling and T. Lozano-Perez, "Hierarchical planning in the now," in *Proc. of Workshop on Bridging the Gap between Task and Motion Planning (AAAI)*, 2010.

[8] E. Plaku and G. Hager, "Sampling-based motion planning with symbolic, geometric, and differential constraints," in *Proc. of ICRA10*, 2010.

[9] S. Alili, A. K. Pandey, E. A. Sisbot, and R. Alami, "Interleaving symbolic and geometric reasoning for a robotic assistant," in *ICAPS Workshop on Combining Action and Motion Planning*, 2010.

[10] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *Proc. of the 20th Int. Conf. on Automated Planning and Scheduling (ICAPS10)*, 2010, pp. 254–258.

[11] J. Guitton and J.-L. Farges, "Taking into account geometric constraints for task-oriented motion planning," in *Proc. Bridging the gap Between Task And Motion Planning, BTAMP'09 (ICAPS Workshop)*, 2009.

[12] K. Hauser and J.-C. Latombe, "Integrating task and prm motion planning: Dealing with many infeasible motion planning queries," in *Proc. Bridging the gap Between Task And Motion Planning, BTAMP'09 (ICAPS Workshop)*, 2009.

[13] F. Zacharias, Ch.Borst, and G. Hirzinger, "Capturing robot workspace structure: representing robot capabilities," in *Proc. of IROS'07, the IEEE Int. Conf. on Intelligent Robots and Systems*, 2007, pp. 3229–3236.

[14] Y. Lebbah, M. Rueher, and C. Michel, "A global filtering algorithm for handling systems of quadratic equations and inequations," in *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming*, ser. CP '02, 2002, pp. 109–123.

[15] L. Karlsson, J. Bidot, F. Lagriffoul, A. Saffiotti, U. Hillenbrand, and F. Schmidt, "Combining task and path planning for a humanoid two-arm robotic system," in *TAMPRA: Combining Task and Motion Planning for Real-World Applications (ICAPS workshop)*, 2012.

[16] "Gurobi optimizer." [Online]. Available: http://www.gurobi.com/