Fabien Lagriffoul | Benjamin Andres

Combining task and motion planning: A culprit detection problem

Suggested citation referring to the original publication:
The International Journal of Robotics Research 35(8) (2015)
DOI https://doi.org/10.1177/0278364915619022
ISSN (print) 1741-3176
ISSN (online) 0278-3649

Article

The International Journal of Robotics Research
2015, Vol. 35(8) 890–927

DOI: 10.1177/0278364915619022
ijr.sagepub.com

Combining task and motion planning: A culprit detection problem

Fabien Lagriffoul[1] and Benjamin Andres[2]

Abstract

Solving problems combining task and motion planning requires searching across a symbolic search space and a geometric search space. Because of the semantic gap between symbolic and geometric representations, symbolic sequences of actions are not guaranteed to be geometrically feasible. This compels us to search in the combined search space, in which frequent backtracks between symbolic and geometric levels make the search inefficient. We address this problem by guiding symbolic search with rich information extracted from the geometric level through culprit detection mechanisms.

Keywords

Combined Task and Motion Planning, Manipulation Planning

1. Introduction

Popular robotic platforms such as ASIMO have demonstrated impressive skills for various types of tasks. These platforms embody the most recent achievements from the fields of computer vision, motion planning, automatic control, and actuation, which provide them with the capacity to achieve a great deal of complex actions. However, these impressive results rely for a large part on human intervention for scripting the sequences of actions executed by the robot. Setting aside the inherent issues of uncertainty in perception and execution, we focus on the planning techniques that could be used for replacing human scripting by a fully automated process. Automated planning techniques exist for computing symbolic plans containing hundreds of actions, likewise efficient motion planning techniques exist that could compute a motion path for each such action. Unfortunately, combining both planning techniques together is not straightforward. The main problem is that symbolic planning works on idealized representations of the real world, hence symbolic plans are not always geometrically feasible at the outset. Consequently, finding a geometrically feasible plan requires combining search both across symbolic and geometric levels. This problem is referred to as Combined Task and Motion Planning (CTAMP).

Searching in the combined search space is intractable in most cases, because the cross product of both search spaces is too large. Decoupling both search spaces is not workable either, because in the case of geometrically intricate problems, the dependencies between geometric actions (which are not captured by the symbolic level) lead to intensive backtracking between symbolic and geometric levels. The key idea for achieving intelligent search across both search spaces is to leverage information from the geometric level in order to guide search at the symbolic level (or vice versa). This idea has been used by many authors, but not fully exploited (see Section 2). In most cases, the information fed back to the task planner relates to a motion path that was unfeasible, or to an object that was occluding another object. We argue that such simple feedback cannot efficiently guide the task planner, because it provides a *local* explanation of failure, i.e. an explanation that is only valid for the particular sequence of actions that produced it.

If the task planner is fed back with local explanations for geometric failures, it may repeatedly end up with plans leading to similar failures. Consider for instance the problem illustrated in Figure 1. The task is to create a pile of blocks *a*-*b*-*c*-*d*, at any location. A geometric failure is detected when the motion planner is called for the last action $place(d, c)$. If the task planner is only notified that this action is unfeasible, it will backtrack to a previous decision point in order to reach the goal through a different sequence of actions. But without the explicit knowledge that the cause of failure is rooted in the choice of p_1 as the

[1]AASS Cognitive Robotic Systems Lab, Örebro University, Sweden
[2]Knowledge Processing and Information Systems, University of Potsdam, Germany

Corresponding author:
Fabien Lagriffoul, AASS Cognitive Robotic Systems Lab, Örebro University, S-70182 Örebro, Sweden.
Email: fabien.lagriffoul@oru.se

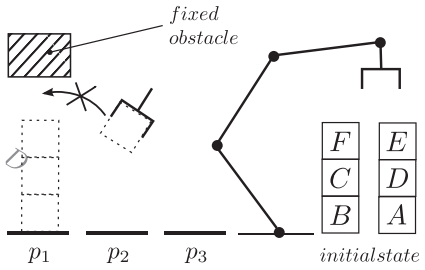


Fig. 1. Stacking the last block is not possible because of collisions between the gripper and a fixed obstacle.

location for block *a*, there is no reason for directly backtracking to that particular decision point. Hence, it may try out a large number of symbolic plans before finding one which avoids this pitfall. This could be avoided if the actual cause of failure was precisely identified. Our approach consists of focusing the computational effort on finding *minimal explanations* for geometric failures, in order to precisely guide the task planner towards a feasible plan. This idea is similar in principle to well-known search techniques used in artificial intelligence (AI) such as dependency-directed backtracking (Stallman and Sussman, 1977) or conflict-driven back-jumping (Dechter and Frost, 2002).

In this paper, we describe the core component of our approach, a geometric reasoner capable of computing minimal explanations for failures occurring in the process of geometrically instantiating a symbolic sequence of actions. These explanations are then used as logical constraints by a task planner based on answer set programming (ASP) (Lifschitz, 2008). Computing minimal explanations essentially boils down to a *culprit detection problem*, which is a difficult problem in general, since it reduces to the set covering problem (Bylander et al., 1991). We propose two techniques to address it. The first one is a polynomial-time algorithm for culprit detection in a constraint network representing a relaxed version of the geometric part of the CTAMP problem. The second one consists of constructing a graph of the geometric dependencies between the actions of unfeasible symbolic plans, in order to extract subsequences of actions which are separately evaluated as potential culprit subsequences. Beyond these two techniques, the main contribution of this paper is to propose a novel view on the problem of combining task and motion planning, by pointing out a culprit detection problem at the interface between the symbolic and geometric search spaces.

The rest of this paper is organized as follows. After reviewing some related work in Section 2, we describe the general principles of our approach in Section 3, which motivate the choices made for the architecture of our system, presented in Section 4. A brief introduction to planning with Answer Set Programming is given in Section 5. Then, the symbolic and geometric domains used for our experiments are described in Sections 6 and 7. The core of the article describes the culprit detection mechanisms in Sections 8, 9 and 10. Finally, we present the results of the experimental evaluation of the proposed approach in Section 11 and end up with some concluding remarks.

2. Related work

Different approaches to CTAMP have been devised, with different schemes for integrating symbolic and geometric reasoning. We review this work in the light of the topic of this paper, i.e. how the information at one level is used in order to guide the search at the other level. A number of relevant related problems in motion planning and constraint programming literature are also reviewed.

In some approaches, the geometric level steers the search and gets guidance from the symbolic level. In SamplSGD (Plaku and Hager, 2010) for instance, the system mainly works on a motion planning problem, while a heuristic task planner (FF, Hoffmann and Nebel (2001)) is repeatedly called in order to compute a utility value based on the length of the symbolic plan that achieves the goal. ASyMov (Cambon et al., 2009) uses a similar principle, but takes into account both the symbolic distance to the goal and the number of failures of the path planner (based on probabilistic roadmaps (PRMs), Kavraki et al. (1996)) to determine the heuristic values of the search nodes. These nodes represent hybrid symbolic/geometric states, and a plan is found using A* search. In these type of approaches, symbolic and geometric reasoning are tightly intertwined, i.e. each visited geometric state triggers a call to the task planner. This may be an issue for large problems, in which decoupling search spaces is necessary for reaching a solution. Our approach addresses this difficulty by *alternating* pure symbolic search and pure geometric search.

In a more common type of approach, the task planner is steering the search, while a geometric reasoner is called to geometrically evaluate the preconditions and compute the geometric effects of actions. These approaches include semantic attachments (Dornhege et al., 2009; Guitton and Farges, 2009; Karlsson et al., 2012). HPN (Kaelbling and Lozano-Pérez, 2011) differs by using a late commitment approach, more suited for interleaving execution and plan refinement. In all these approaches, the feedback from the geometric level to the symbolic level consists in mere "success" or "failure", the latter resulting in a dead-end for the task planner. This opens the door for repeatedly encountering similar geometric failures, as explained in the introductory example. By contrast, our approach prevents this by analyzing the very cause of geometric failures, and provides a meaningful feedback to the task planner so that the same failure cannot occur again.

The approach of Srivastava et al. (2014) allows a richer feedback by means of logical predicates. They present a general interface which takes care of the geometric details, and assume optimistic default values for geometric preconditions. If a geometric failure is detected, the symbolic state is updated accordingly and re-planning is triggered. This approach relies on the assumption that the actual cause of

geometric failures lies in individual actions, and that the plan can be repaired from the current state. Again, this stands in contrast with our approach, which is based on the observation that locally dealing with geometric failures may cause them to re-occur over and over again.

Garrett et al. (2014) tightly connect geometric and symbolic levels via a conditional reachability graph used for computing the heuristic of the task planner. The heuristic implicitly informs the symbolic level about occluding objects that need to be moved and in which order they are to be moved. This approach is somehow opposite to Asymov (using FF as a heuristic guiding a PRM planner) since it uses a PRM planner to compute a heuristic for FF. It is not possible to pre-compute the conditional reachability graph for all possible situations, therefore these computations are performed on-demand while the heuristic is computed. The problem is then similar to Asymov: Each visited symbolic state triggers geometric computations, thus both search spaces are tightly intertwined. This may be problematic for large problems that require decoupling of the search spaces.

Lozano-Pérez and Kaelbling (2014) frame the CTAMP problem as a discrete Constraint Satisfaction Problem (CSP) (Rossi et al., 2006) for quickly assessing if a given symbolic plan is geometrically feasible or not. A solution to the CSP provides grasps and placements that do not interfere with each other. The strength of their approach is to account for path existence constraints in the CSP formulation, by pre-computing a map representing the free-space. This approach focuses on the geometric aspects of CTAMP, assuming a symbolic plan given by an external task planner, but it does not provide a mechanism for integrating geometric constraints in the symbolic search space, as we propose in this paper.

In previous work (Bidot et al., 2015; Lagriffoul et al., 2014), we combined a HTN (Hierarchical Task Network) planner (Nau et al., 2004) for task planning with a bidirectional RRT path planner (LaValle, 2006) for motion planning, and used a linear constraint network for pruning out kinematically inconsistent choices for grasps and placements. The limitation of this approach is that HTN (and more generally state-space planning) does not allow us to exploit geometric constraints in a meaningful way at the symbolic level. The reason for this is that geometric constraints are fed back to the task planner through the *preconditions* of symbolic operators. This restricts the expressiveness of constraints that can be fed back to the symbolic level, since they inherently relate to single actions. In the presented work, we address this limitation by replacing the HTN planner by a logic programming approach, which is allows us to leverage meaningful geometric constraints in the task planning process (see Section 3.2).

A third type of approach consists of stating the symbolic planning problem in terms of logic programming (Gelfond and Lifschitz, 1998; Kautz and Selman, 1992; Lifschitz, 2002). The main difference with the previous type of approaches (based on state-space planners) is the way the symbolic space is traversed. With logic-based planning, the task planner operates in a search space comparable to the space of plans. Such a search space enables pruning out families of plans regardless of the exact chronology of their actions, unlike state-space planners which can only prune out sub-trees rooted in the state currently visited. This feature is exploited in the approach presented in this paper: The geometric failures detected in a small number of unfeasible plans are used for pruning out entire families of plans containing the same flaws, although their sequences of actions may be very different.

In this vein, Choi and Amir (2009) use a sampling-based motion graph to build an action theory, from which a plan is computed. Only feasible actions are represented in the graph, hence failures do not directly guide the search. However, the reachability of objects is associated to *modes*, which implicitly represent the fact that some combinations of actions prevent some objects from being reached. In Luna et al. (2014), a Satisfiability Modulo Theories (SMT) solver is used for plan synthesis. Like in the approach of Choi and Amir (2009), the failures are not explicitly fed back, but the feasibility of geometric paths with respect to objects placements is connected with the logical level, through a manipulation graph (computed offline) encoded in the formula. Erdem et al. (2011) use the action language $C+$ to encode the planning problem into a logic program. The failures detected at the geometric level are fed back in the form of logical constraints and a new plan is computed. The feedback is limited to collisions or infeasibility of motion paths. A similar approach is taken by Aker et al. (2012), but using ASP programs. A similar scheme is used in our work, but the major difference in our approach (besides using culprit detection mechanisms) is the level of granularity used for symbolically representing the world (see Section 6.6 for a comparison between both approaches).

Toussaint (2015) addresses sequential manipulation planning problems of building stable piles of objects. His approach stands out from the previously mentioned ones in the sense that the symbolic level is not guided by geometric failures, but rather by a heuristic value calculated by optimizing different costs, computed for different levels of refinement of the symbolic action sequence. At the lowest level of refinement, the cost is given by optimizing the stability of the resulting pile, and in further refinement, kinematic constraints are taken into account for optimizing motions. As mention by the author, this approach is valid for problems where collisions are not of major concern (a flying robotic arm is used), because collisions are considered only at a later stage. However, this may be inefficient for more "classical" CTAMP scenarios, i.e. where collisions are the main cause of reconsidering symbolic decisions.

Related problems

A number of techniques have been developed beyond basic motion planning in order to cope with robotic problems.

The limitations of motion planning arise when obstacles need to be moved, or when task constraints impose discrete steps on motion paths. A general approach to the Manipulation Planning problem was proposed by Simeon (2004), based on the composition of several Probabilistic Roadmaps (PRMs). Stilman and Kuffner (2008) and Stilman et al. (2007) address the difficult problem of robot Navigation Among Movable Obstacles (NAMO), with a backward search algorithm that recursively moves occluding objects out of the space which the robot has to traverse. Multi-modal Motion Planning addresses high dimensional motion planning problems by planning discrete mode switches in which lower dimensional subspaces are sampled using domain-dependent strategies. This approach has been successfully applied by Hauser and Latombe (2010) to climbing robots, or push-planning by a humanoid robot (Hauser et al., 2007). These works however, do not take causal reasoning into consideration.

Recent work by Hauser (2014) on the Minimum Constraint Removal (MCR) problem is relevant to the present work. It is proven that deciding the minimum number of obstacles to remove for making a path feasible is NP-Hard. A greedy algorithm is presented, which can compute *parsimonious explanations for path planning failures*. This falls in line with our approach, which aims at computing minimal explanation for geometric failures, but is currently lacking methods for detecting path planning failures.

Several search techniques developed in other areas are also relevant to this work. Although they address different types of problems, they share with the present work the use of culprit detection mechanisms for pruning the search space. Stallman and Sussman (1977) introduced the Dependency-Directed Backtracking scheme to reduce the complexity of electronic circuit analysis. The possible operating regions of electronic devices are represented by discrete states, and their physics are described by algebraic relations. As a physical contradiction is detected, a dependency-structure is used to compute a relevant explanation and prevent similar choices occurring again. Similar techniques are used in Boolean Satisfiability (SAT) solvers. The conflicts occurring during search are analyzed by specialized procedures (Silva and Sakallah, 1996), and a clause expressing the negation of the cause of conflict is re-injected in the clause database for pruning the search space. Backjumping techniques (Dechter and Frost, 2002) analyze the dead-ends reached during search to identify inconsistent partial solutions, which allows the algorithm to backtrack several levels up in the decision tree, skipping irrelevant variables. Similarly in this paper, specialized procedures perform culprit detection at the geometric level, which are then exploited by the intelligent backtracking mechanisms of the ASP solver.

3. General principles

Our approach relies on two key components. First, there is a geometric reasoner capable of analyzing the cause of geometric failures. This is achieved by two culprit detection mechanisms which we introduce in this section. Secondly, since the cause of failure is not a mere "success/failure" answer, we need a common language between the geometric reasoner and the task planner, so that the cause of failure can directly be used by the symbolic search process. In most approaches, the common language is defined by the *preconditions* of symbolic operators, which are true/false depending on the success/failure of the geometric reasoner. Here, since we use a logic programming approach for task planning, the common language is more expressive since it can be any logical expression supported by the task planner.

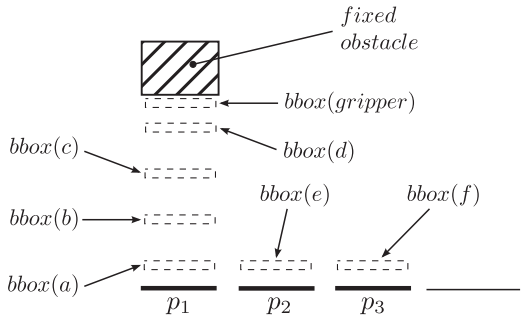


Fig. 2. A set of bounding boxes (bbox) representing all the possible poses that the center of each object can occupy is computed. In the illustrated example, block e is placed on p_2 and block f is placed on p_3. Some bounding boxes are also computed for the intermediate poses of objects, but they are not represented in this figure.

3.1. Finding minimal explanations for geometric failures

Consider again the blocks-world problem illustrated in Figure 1. If the task planner initially decides to build the pile at location p_1, the action $place(d, c)$ always fails geometrically, because the gripper always collides with the fixed obstacle. The cause of failure is not the action $place(d, c)$ per se, because this action would be feasible if the pile was built at location p_2 or p_3. Rather, it is the result of the choice of p_1 as a location for a, combined with the geometric effects of actions $place(b, a)$, $place(c, b)$, $place(d, c)$, and the position of the fixed obstacle relative to p_1. Note also that, during the last action ($place(d, c)$), blocks e and f have been moved to some temporary locations, but neither the choice of these locations nor the order in which blocks are moved are relevant for explaining the failure. Wherever blocks e and f are placed, and whatever order is chosen for actions, the same problem will eventually occur. Therefore, a *minimal explanation* of the failure should only depend on blocks a, b, c, d and p_1, otherwise the task planner may return an infinite number of unfeasible plans by permuting the temporary locations of blocks e and f, by permuting the order of actions, or by increasing the number of actions. *Isolating the minimal number of factors explaining the failure is the culprit detection problem that we propose to solve.*

Definition *Culprit Detection Problem*

The input of a culprit detection problem is defined by a set of hypotheses and a set of observations to be explained. The output is an explanation, i.e. a parsimonious set of hypotheses which explains all the observations (Bylander et al., 1991).

Next, two methods are sketched out for addressing this culprit detection problem. In the first method, the hypotheses are a set of linear constraints, and the observation is the inconsistency of the constraint network. In the second method, the hypotheses are symbolic actions, and the observation is when a sequence of actions is not geometrically feasible.

The first method consists of computing a set of bounding boxes, which encompass all the poses that each object can possibly occupy at each time step (see Figure 2). The sizes and positions of the bounding boxes are computed using the spatial relations between objects (e.g. $on(a, p_1)$, $on(b, a)$) taken from the symbolic plan, plus some numeric information from the geometric domain, e.g. the pose of p_1, the dimensions of block a, etc. The bounding boxes are represented by a network of linear constraints. The constraint network is used to detect geometric failures caused by violation of *kinematic constraints*, and bounding boxes are used to sample objects/robots poses for detecting geometric failures caused by *collisions*. For instance, the bounding box of the gripper ($bbox(gripper)$ in Figure 2) is used to sample a discrete subset of the poses that the gripper can possibly occupy, and perform a collision check for each of them. Since all the samples cause a collision, the sequence of actions is unfeasible. Then, using the constraint network and culprit detection mechanisms, it is possible to prove that the pose of $bbox(gripper)$ only depends on the poses of p_1, blocks a, b, c and d, and create an explanation of the failure which is neither depending on blocks e and f, nor depending on the order of actions. The culprit detection mechanisms for achieving this are presented in Section 8. The drawback of this method is that the bounding boxes cover volumes which are often larger than what the manipulator can actually reach. Hence, some sample positions that are actually not feasible can be found to be collision-free, which causes some failures not to be detected.

The second method copes with this problem by discretizing the poses of robots and objects using the same process that is used for finding a geometric plan (see Section 7.3), the difference being that motion planning is not performed, i.e. only the initial and final configurations of actions are considered (more about this point in Section 4.3), and only subsets of actions from the task plan are considered. The goal is to find a minimum subset of actions causing the geometric failure. Imagine for instance a sequence of symbolic actions $\langle A_1, \ldots, A_n \rangle$. Let us assume that a geometric failure occurred for action A_7. It may be that the problem is intricate and, regardless of the geometric instances chosen for the symbolic actions, there is no solution to the problem.

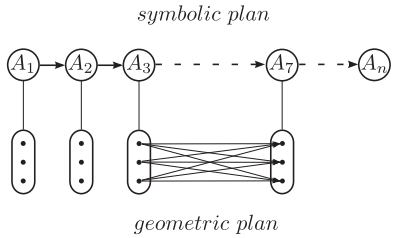


Fig. 3. Schematic illustration of the test of the subsequence $\langle A_3, A_7 \rangle$. The black dots represent geometric instances of symbolic actions resulting from the discretization process (see Section 7.3). All the combinations within these discretized geometric instances of A_3 and A_7 are tested.

But most often, geometric failures are caused by one or two actions only. For instance, if a large object is placed in a box (A_3), it is impossible to place another object in that box later on (A_7), and this problem is *independent* from the actions performed in between ($\langle A_4, A_5, A_6 \rangle$). Proving this is a culprit detection problem. In order to detect the culprit action(s), several subsequences of actions are tested in isolation, e.g. $\langle A_1, A_7 \rangle$, $\langle A_2, A_7 \rangle$, etc. All the possible combinations within the sets of discretized geometric instances of each action in the subsequence are tried out (see Figure 3), and if all of them fail, the subsequence is reported as unfeasible. Section 9 describes how this is done in practice, in particular how to select proper subsequences of actions, since trying all of them is intractable. Next, we discuss how to represent the causes of geometric failures, and how to use them within the task planning process.

3.2. Reasoning about failures in the planning process

Continuing on the blocks-world example (Figure 1), let us assume that the actual cause of failure is detected by the geometric reasoner, and returned to the task planner. As mentioned above, the cause of failure should not include the positions of blocks e and f, neither should it refer to the order of actions. One could represent it as a conjunction of logical statements, for instance

$$on(a, p_1) \wedge on(b, a) \wedge on(c, b) \wedge on(d, c) \qquad (1)$$

This information is valuable only if it can quickly guide the task planner to backtrack to the action $place(a, p_1)$ and build the pile at a different location. If the planning problem is modeled in propositional logic, this process is facilitated, because the planning problem is represented as a set of clauses $\mathcal{P}$ (Kautz et al., 1996), and expression (1) can be added as a logical constraint to the problem. Informally

$$\mathcal{P} \wedge \neg(\, on(a, p_1) \wedge on(b, a) \wedge on(c, b) \wedge on(d, c)\,)$$

which is equivalent to

$$\mathcal{P} \wedge \neg(\, on(a, p_1) \wedge goal)$$

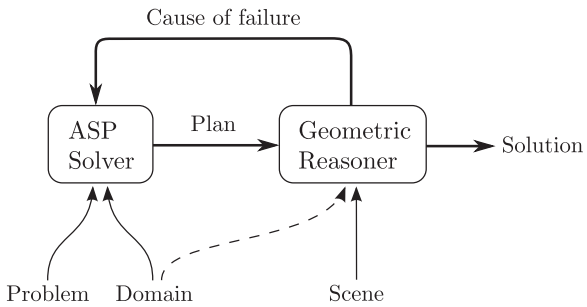


Fig. 4. The two main components of our system; the ASP solver for task planning, and the geometric reasoner to geometrically instantiate the symbolic plan or to analyze the causes of failure.

Since the goal must be true, a simple inference mechanism entails that $on(a, p_1)$ is false.

Most state-of-the-art planners are based on heuristic search and do not support this type of global inference mechanism. Therefore, we use the logic programming paradigm for the task planning component of our system. It allows us to efficiently guide task planning with logical constraints formulated from the explanations of geometric failures, and the inference mechanisms of logic programming. In the present work, we opted for ASP, which provides an expressive language and effective solvers. Note that the proposed approach does not specifically rely on ASP: The only requirement for the task planner is to support global inference mechanisms. Other logic-programming languages, or satisfiability-based planners could be used as well.

4. System overview

4.1. Overall architecture

The overall architecture of our system is simple (see Figure 4). The ASP solver takes as input a domain definition file and a problem definition file, both written in AnsProlog, the logic programming language of ASP. The domain describes the actions, when they can apply, and which logical effects they have. It also contains a set of rules which describe what does not change (frame axioms), and what is indirectly changed by the actions (indirect effects). More details about these rules are given in the next section and in Section 6. A problem definition file contains a symbolic description of the initial state and the goal state. The geometric reasoner takes as input the geometric description of the initial scene, i.e. the initial poses of robots, objects, and obstacles. The scene also includes the 3D representations of each robot, object, and obstacle. The geometric reasoner also gets some information from the symbolic domain: which objects are movable, and the kinematic structure of compound robots, i.e. which base is connected to which manipulator. The working process is a simple loop where (i) the ASP solver finds a symbolic solution plan, (ii) the plan is analyzed by the geometric reasoner, and (iii) the geometric reasoner feeds back the (potential) cause of failure to the ASP solver in the form of a logical constraint, or returns a geometric instance of the symbolic plan otherwise. With this logical constraint added to the problem, the ASP solver generates a new plan that is free from the detected failures, and the cycle repeats until a feasible plan is found.

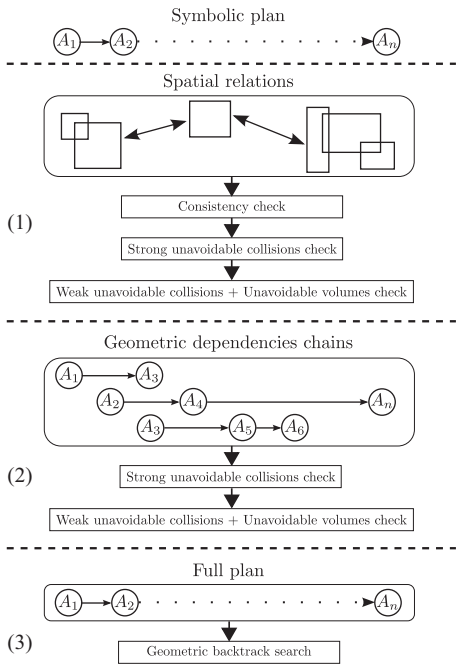


Fig. 5. The geometric reasoner analyzes the cause of failure through three layers representing the plan at different levels of abstraction: (1) spatial relations, (2) geometric dependencies chains, and (3) whole plan with motion paths.

For the goals of this paper, the core component of our system is the geometric reasoner (Figure 5). The geometric reasoner takes as input a sequence of symbolic actions. First, this sequence of actions is searched for geometric failures in layer (1) spatial relations and layer (2) geometric dependencies chains. If no failure is found, it attempts to geometrically instantiate the symbolic plan in layer (3) by searching for a motion path for each action. If a failure is detected in any of these layers, the geometric reasoner returns a logical expression describing the cause of failure. The layers are hierarchically organized from a high level of abstraction down to the motion planning level. The lower in this hierarchy, the more computationally expensive it is to detect a failure. Hence, when a failure is detected, a logical constraint is returned and the remaining more primitive checks are not performed. A more detailed description of the different layers is given in Section 4.2.

4.2. The geometric reasoner

Finding a *culprit subset* of elements is a difficult problem in general, because it requires checking all the subsets in the power set of these elements, which requires up to 2^N checks, N being the number of actions in a symbolic plan in our case. This quickly leads us to an intractable

number of subsets of actions to be checked. In CTAMP, this combinatorial problem is made worse by the fact that "checking" one subset of actions implies that various geometric computations are performed, including searching for feasible grasps and placements, and motion planning. The first layer of the geometric reasoner "Spatial relations" (1) copes with this complexity by working on an abstraction of the space of grasps and placements, by building a set of bounding boxes which encompass all the possible poses that objects/robots can occupy after completion of each action. Although this representation is not precise, it allows us to detect some geometric inconsistencies in polynomial time (see Section 8). These bounding boxes are also used to perform various collision checks which are explained in Section 10. No motion planning is performed in this layer. The logical constraints returned by this layer are expressed in terms of spatial relations between objects (see Section 6.4).

Since the first layer does not take into account the kinematics of robots, it may let some geometric failures go by undetected. The "geometric dependencies chains" in layer (2) copes with this problem. This layer analyzes the *geometric dependencies* between the actions of a symbolic plan. For example, if two mobile robots located at some distance from each other pick up two different objects, there are no geometric dependencies between both actions, but if robot A reaches robot B and hands over an object, then geometric dependencies between these actions exist. Layer (2) constructs a graph of the geometric dependencies between the actions of a plan, and uses this graph to select some subsequences of actions to be extracted from that plan. These subsequences are then geometrically evaluated separately from the other actions in the plan. Several types of collision checks are performed during this evaluation (see Section 10). The details of this process are presented in Section 9. Motion planning is not performed in this layer, i.e. only the feasibility of the initial and final configurations of the paths is checked. The logical constraints returned by this layer represent unfeasible subsequences as partially-ordered subsequences of actions.

Finally, if no geometric failure is detected by the previous layer, layer (3) evaluates the whole sequence of actions through geometric backtrack search, that is, depth-first search in the search space of possible grasps and placements until a motion path for each action is found (see Section 9.1, and our previous work, Bidot et al. (2015)). For this last step, a cutoff time is set. If no solution is found within the time limit, the logical constraint returned by the geometric reasoner is the subsequence of actions that it managed to instantiate within the time limit. For the ASP solver, this means that it must no longer return any plans that begin with this subsequence of actions.

4.3. Assumptions and completeness issues

In Section 3.1, the second method for culprit detection analyzes subsequences of actions by only considering initial and final configurations of motions. The reason why we exclusively consider these two configurations is motivated by the fact that we do not consider heavily cluttered environments. Therefore when kinematically feasible initial and final configurations have been found, finding a motion path is possible in most cases. Furthermore, the manipulators are more subject to kinematic constraints at grasp and release positions, because the pose of the gripper is constrained by the pose of the object to be grasped / target pose to place the object in, which is not the case during the transfer of the object. Note however that all the paths are computed in any case, i.e. our system does not produce motions which may cause collisions. But if an action is invalidated because of a path planning failure, no meaningful explanation is fed back to the symbolic level, and the same failure may be encountered again in a different sequence of actions. This owes to the fact that identifying the culprit colliding object(s) in a path planning failure is a difficult problem (Hauser, 2014). This issue is discussed further in the conclusion.

The proposed approach is not complete in different respects. Although the motions performed for each action are computed by a resolution-complete path planner, the start and goal configurations of these paths are a priori discretized, therefore many start and goal configurations are excluded from the search space. Another source of incompleteness lies in the fact that the geometric problem is broken down by the task planner into a sequence of subproblems, each of which is solved within a subspace of the configuration space. For instance, if the symbolic plan contains an action commanding the right arm of a humanoid robot, the subspace is the configuration space of the right manipulator, while the left arm acts as an obstacle. Potential solutions are lost in this way, compared with if the problem had been stated in the combined search space of both arms. For the same reason, only the objects represented in the symbolic domain can be acted upon. Therefore, if they are not symbolically represented, occluding objects cannot be moved away, nor can a flat surface be used as a temporary location.

Another issue with completeness concerns the detection of failures. Proving a continuous-space problem unfeasible is not possible with sampling-based techniques. However, our simplified approach for multi-step motion planning facilitates this process: since the resulting configurations of actions are discretized, and since they act as obligatory pathways for a global solution, failures can be easily detected by considering these configurations in priority. Failures owing to kinematic violations and failures owing to collisions present us with two different cases. In the case of kinematics violations, our approach is *conservative*, i.e. the bounding boxes always overestimate the actual capacities of manipulators, or the size of regions in which objects can be. Consequently, violations of kinematic constraints can be safely fed back to the ASP solver without loss of solutions. This is not true for the failures owing to collisions, because collision checks are performed on a finite set of samples, therefore feasible configurations may not be discovered. In

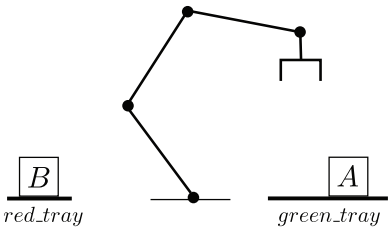


Fig. 6. Initial situation of the minimal example of placing block b onto the red_tray.

```
1 #program base.
2 block(a). block(b).
3 location(green_tray). location(red_tray).
4 actuator(arm).

6 init(on(a,green_tray)).
7 init(on(b,red_tray)).
8 goal(on(a,red_tray)).
```

Fig. 7. ASP instance for the minimal example.

this case, the constraint returned to the ASP solver prunes out potential solutions.

5. Planning with ASP

A formal definition of the ASP language is given in Appendix A. For illustration purposes, a small ASP example will be presented in this section. The problem in the example scenario is for a robotic arm to move a block `a` from a `green_tray` to a `red_tray`. The scenario also includes an additional block `b` which may obstruct a trivial solution.

As common in ASP, we divide the encoding of the example into two parts, a fact format for representing problem instances and a generic encoding for solving pick&place problems. Figure 7 presents the problem instance of our example. The facts in Line 2 to 4 define our environment, consisting of two blocks, two locations and an actuator (the robotic arm). Line 6 and 7 presents the initial conditions, with block `a` placed on the `green_tray` and block `b` on the `red_tray`. Finally, Line 8 defines the goal condition of our example with block a on the `red_tray`.

A general ASP encoding for solving this problem is shown in Figure 8. Note that this code is for illustration purposes, and is a simplified version of the actual code (Appendix C presents samples of the actual encoding). The encoding consists of three parts: `base`, `incremental` and `end`. The `base` part represents the initial situation of the scenario. In the `incremental` part possible actions and their effect on the environment at a specific point in time are defined. Finally, the goal conditions are described in the `end` part. As long as the `incremental` part is insufficient to satisfy the goal condition in the `end` part, an additional time step (action) is appended to the `incremental` part. This is handled by an outside controller as well as the identification of the new final action. While expanding the encoding with an `incremental` part all occurrences `t` are substituted by an integer, representing the time step to be added.

In the encoding shown in Figure 8, the `base` part extends from Line 1 to 6, the `incremental` part form Line 8 to 28 and the *end* part from Line 30 to 31. The rules in Line 3 and 4 formulate the potential actions the robotic arm is able to execute, with `pick_up(Block)` stating that `Block` is to be picked up and `place(Location)` that a currently grasped object is to be placed on `Location`.

The first rule of the `incremental` part is a choice rule (Line 10), stating that on every time step the task plan may include one action from the set of potential actions specified above for each actuator. The `do(Actuator,Action,t)` predicate represents that an `Actuator` performs an `Action` at time step `t`. The integrity rule in Line 11 ensures that any chosen action for each actuator must be possible for it to execute in this incremental step. Possible actions are defined by the rules in Line 13 to 20. The first rule states that it is possible for an actuator to `pick_up` any block, given that the block was placed on a location and the actuator was not grasping anything in the previous step. The second rule states that it is possible for an actuator to place a block at any location if the actuator was grasping the block in the previous step.

Lines 22 to 28 model the logical consequences of chosen actions implementing the frame axiom in ASP. If a `pick_up` action is chosen for an actuator, it holds for the current step that the object is grasped by the actuator (Line 22), while the condition that the object is on a location stops (Line 23). The rules are equivalent for the `place` action, but the block is now on the placed location (Line 25) and stops to be grasped by the corresponding actuator (Line 26). Line 28 declares that any fluent held in the previous step also holds in the current step unless it was stopped.

The `end` part of the encoding starts with the external literal `horizon(t)` which in Line 30 identifies the last incremental step of the solution, i.e. the last action of the action plan. Being external, the value of the literal is determined by the controller, not the solver. The controller sets `horizon(t)` to true if `t` is the last incremental step and to false if not. The integrity rule in Line 31 excludes all answer sets in which the goal of the example is not fulfilled in the last incremental step.

Since it is not possible to move block `a` to the `red_tray` in only one action, the ASP solver fails to find a solution with only the `base` and `incremental(1)` part. Thus, the controller adds the `incremental(2)` part to the encoding and a solution can be found:

```
do(arm,pick_up(a),1)
do(arm,place(red_tray),2)
```

Assuming the `red_tray` is not large enough to hold both blocks, the geometric solver rejects the plan and feeds back

```
#program base.

action(pick_up(Block)) :- block(Block).
action(place(Location)) :- location(Location).

holds(Fluent,0) :- init(Fluent).

#program incremental(t).

{do(Actuator,Action,t) : potential(Action)}1 :- actuator(Actuator).
:- do(Actuator,Action,t), not possible(Actuator,Action,t).

possible(Actuator,pick_up(Block),t) :-
  action(pick_up(Block)),
  actuator(Actuator),
  holds(on(Block,Location),t-1),
  not holds(grasped(_,Actuator),t-1).
possible(Actuator,place(Location),t) :-
  action(place(Location)),
  holds(grasped(_,Actuator),t-1).

holds(grasped(Block,Actuator),t) :- do(Actuator,pick_up(Block),t).
stop(on(Block,Location),t) :- holds(on(Block,Location),t-1), do(_,pick_up(Block),t).

holds(on(Block,Location),t) :- do(Actuator,place(Location),t), holds(grasped(Block,Actuator),t-1).
stop(grasped(Block,Actuator),t) :- holds(grasped(Block,Actuator),t-1), do(Actuator,place(Location),t).

holds(Fluent,t) :- holds(Fluent,t-1), not stop(Fluent,t).

#external horizon(t).
:- goal(Goal), horizon(t), not holds(Goal,t).
```

Fig. 8. ASP encoding for the minimal example.

an integrity constraint describing the cause of error to the ASP solver, i.e. that `a` and `b` may not be on `red_tray` at the same time step. Since there are now no valid solutions with only two actions, the encoding is extended by two additional incremental steps (3 and 4). The next solution is found with four actions by first placing the block `b` on the `green_tray` and then the `a` block on the `red_tray`:

```
do(arm,pick_up(b),1)
do(arm,place(green_tray),2)
do(arm,pick_up(a),3)
do(arm,place(red_tray),4)
```

Note that the plan length is only increased if the ASP solver proved that there are no valid plans for the current length.

6. Symbolic domain

In order to make the presentation of our techniques more concrete, we will use examples based on a concrete domain, in which we use three simulated robots with different capabilities. The first robot is Justin, the DLR humanoid robot (Ott et al., 2006), with two arms with 7 degrees of freedom (DoF) each, and two dexterous hands. The second robot is Fabot, a mobile manipulator with a 3 DoF arm that can translate along the vertical bar attached to its base, which allows it to grasp objects on the floor, or to reach high locations (see Figure 9). For the mobile part, Fabot's base is holonomic. The third robot is r2d2, a mobile robot with holonomic base and a flat area on top, which can be used as a mobile tray. Justin is constrained to be fixed, in order to enforce the cooperation between the robots.

6.1. Representing robots

Robot parts are referred to as *components*, represented by variables, and some predicates are used to define properties or relationships between them. For instance, Fabot is defined as follows:

```
component(fabot_base)
component(fabot_arm)
architecture_child(fabot_base,fabot_arm)
base(fabot_base)
able(fabot_base,moving)
able(fabot_arm,manipulating)
skilled(fabot_arm)
```

The predicate `architecture_child` indicates that the two mentioned components belong to the same robot, and the predicate `base` identifies its base. The specific abilities of the components are represented with the `able` predicate, which determines which actions each component is supported for. The `skilled` predicate specifies that Fabot can pick piles of objects (because the design of its manipulator prevents the gripper from tilting). This can be easily modeled by adding a constraint in the domain (see Appendix C, 3) without the need for defining a different `pick` action for each robot. This scheme allows us to model more complex robots such as Justin (see Appendix C, 5).

Objects and locations are also represented by variables and predicates. For instance:

```
location(table)
object(cup)
object(block_a)
```

Types can be assigned to objects, for domain-specific use:

```
block(block_a)
```

A robot such as r2d2 can also be used as a location, i.e. objects can be placed on it:

```
location(r2d2)
```

6.2. Geometric predicates

Geometric predicates form a language which allows the ASP solver to symbolically reason about the physical world. These predicates accept a time parameter (see parameter `t` in Figure 8) which is omitted here for brevity.

1. `moved(X)` represents the fact that `X` moves or is moved. `X` can be a component, object, or location.
2. `connected(Parent,Child)` implies that if `Parent` is moved, then `Child` moves as well (but not necessarily the converse). It applies to a wide range of situations: robot composition, object grasp, or object support. Examples: `connected(fabot_base,fabot_arm)`, `connected(fabot_arm,cup)`, `connected(tray, cup)`.
3. `on_location(Object,X)` represents the relation resulting from the transitivity of the `connected` relation. `X` can be a location or a component. Example: `block_a` is on the `tray` and `block_b` is on `block_a`, then `on_location(block_b,tray)`.
4. `oriented(Object,Orientation)` represents the gross orientation of `Object`, i.e. its alignment/anti-alignment with one of the reference axes (see Section 7.1). `Orientation` can be `x1`, `x2`, `y1`, `y2`, `z1` or `z2`, e.g. `z1` represents upright and `z2` upside-down.
5. `reachable(X,Component)` represents the fact that `Component` is located at a sufficient distance from `X` for attempting a pick, place or stack action. `X` can be an object or a location.
6. `manipulated(Component,Object)` represents the fact that `Object` is actively acted upon by `Component`, directly or indirectly. Examples of manipulated cup: Fabot grasps a cup, Fabot moves its base while holding a cup, r2d2 moves with a cup on top of it.

The value of these predicates changes over time by the direct effect of actions, but also indirectly through side-effects and ramification. For instance, when the base of a robot moves, the locations/objects reachable by the manipulator are not `reachable` any longer. Similarly, if the robot is holding an object, this object is also `moved`. By using ASP as a modeling language, we are able to express ramifications and indirect effects in a native way. Appendix C (1,2) presents rule samples that illustrate how this can be handled.

6.3. Actions

The symbolic domain consists of six actions: The manipulators are able to perform `pick`, `place` and `stack` actions, while the bases can do `reach` and `dock` actions. All components can perform the `move` action. A general description of actions is given here, the reader may consult Appendix C for a complete AnsProlog implementation of the `pick` action as an example.

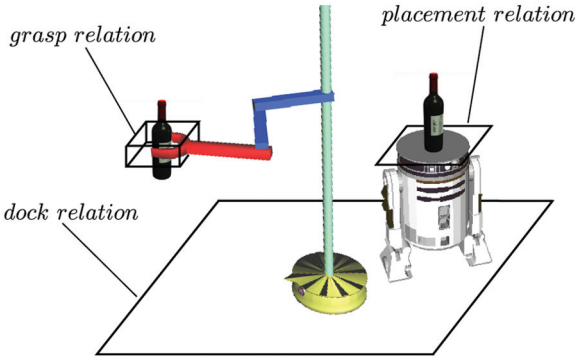


Fig. 9. The three types of spatial relations.

1. `pick(Object,GraspType)` represents the action of picking `Object` using a given grasp type (`top`, `side`, or `bottom`).
2. `place(Orientation,GraspType,Location)` represents the action of placing the held object on `Location` using `GraspType` in a given orientation (`z1` or `z2`). `Location` must be of type `location`.
3. `stack(Orientation,GraspType,Location)` is similar to `place`, but the target location must be of type `object`. Geometrically, sample poses for stacking are limited to one point centered on top of the supporting object, with different orientations.
4. `reach(X,Manipulator)` moves the base to which `Manipulator` is connected so that `X` becomes reachable by it. `X` can be a location or an object.
5. `dock(Base,Manipulator)` is the converse of `reach`: it moves `Base` so that `Manipulator` can reach it (used by the r2d2 robot).
6. `move(Component)` simply moves `Component` away from its current pose[1]. This action is used if a component needs to be `moved`.

6.4. Spatial relations

Just as symbolic actions are the symbolic counterparts of geometric actions, spatial relations are the symbolic counterparts of spatial constraints (see Section 8). They are used in order for the ASP solver to reason upon the logical constraints fed back from the "Spatial relations layer". The general form of a spatial relation predicate is:

```
relation(X, Y, type, p1, …, pn, t)
```

where `X` and `Y` represent the two objects/robots/locations on which the relation applies. We define three types of spatial relations: *grasp, placement,* and *dock*, which are illustrated in Figure 9.

A *grasp relation* exists between an object and a Tool Center Point (TCP) whenever an object is picked, placed, or

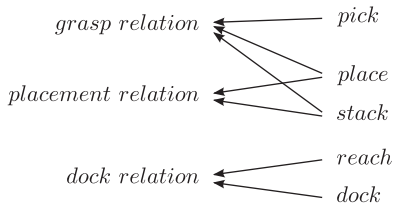


Fig. 10. Entailment of spatial relations.

stacked. It represents the fact that, at some point during the action, the TCP of the robot is necessarily within a volume centered at the object. A grasp relation does not hold any longer if:

- the object is not grasped and the TCP is moved or;
- the object is not grasped and the object is moved.

A *placement relation* exists between a robot/object/location o_1 and an object o_2 whenever o_2 is placed or stacked. It represents the fact that, at the end of the action, o_2 is necessarily located in a region centered around o_1. A placement relation does not hold any longer if:

- o_2 is not connected with o_1 and o_1 is moved or;
- o_2 is not connected with o_1 and o_2 is moved.

A *dock relation* exists between location/robot r_1 and a robot r_2 whenever r_1 docks to r_2 or when r_2 reaches a location/robot r_1. It represents the fact that, at the end of the action, r_2 is necessarily located in a region centered at the location/robot r_1. A dock relation does not hold any longer if:

- r_1 is moved or;
- r_2 is moved.

The implementation of these rules is provided in Appendix C (4). Note that spatial relations, just as with spatial constraints, do not represent an actual state of the world. Their role is rather to express a necessary relation resulting from an action. For instance, the existence of a grasp relation does not mean that the object can be grasped, however, if the grasp relation cannot hold, then a fortiori the object cannot be grasped (nor placed, nor stacked). Therefore, when a spatial constraint is proven not to hold (by the geometric reasoner), the corresponding spatial relation can be use by the ASP solver for pruning the actions by which it is entailed.

Spatial relations are entailed by actions as depicted in Figure 10. Spatial relations are used when the cause of failure is detected by the "Spatial relations layer" of the geometric reasoner. A logical constraint expressed with spatial relations is more powerful than a logical constraint expressed with actions, because (i) there is a "many-to-one" mapping between actions and relations, and (ii) because the predicates of spatial relation sometimes have less parameters than the predicates of actions. For instance, imagine that the geometric reasoner computes the following constraint:

```
:- relation(r2d2, bottle, placement, z1, t)
relation(r2d2, fabot, dock, t)
relation(justin, bottle, grasp, top, t)
```

This means that Justin cannot pick/place the bottle in upright position ($z1^2$) from/on r2d2 with a top-grasp, while r2d2 is docked to Fabot. This constraint is powerful because (i) it applies to both *pick* and *place* actions, (ii) it does not explicitly say how the placement relation is created (it could be any robot using any type of grasp), (iii) nor does it say how the dock relation is created (it could be r2d2 docking to Fabot or Fabot reaching r2d2). This type of constraint can rule out a large number of symbolic plans. Therefore, constraints expressed in terms of spatial relations achieve a stronger guidance of the task planner, compared to constraints expressed in terms of actions.

6.5. Generalized constraints with types

In many scenarios, it is needed to manipulate several objects that are instances of the same type. One can reasonably assume that during the perception of the scene, it is possible to compare the shape of objects and assign them to different classes. In the present work, we manually assigned the type "block" for all objects:

```
block(block_a). block(block_b). block(block_c)...
```

The idea is to make some logical constraints more general by using typed variables instead of object instances. Consider for instance the geometric failure depicted in Figure 1. This failure can be described using spatial relations:

```
:- relation(p1, block_a, placement, z1, t)
relation(block_a, block_b, placement, z1, t)
relation(block_b, block_c, placement, z1, t)
relation(block_c, block_d, placement, z1, t)
relation(gripper, block_d, grasp, top, t)
```

Since all blocks have the same geometry and can afford the same grasps, this constraint is actually valid for any combination of block instances. Hence, we can use the following generalized constraint instead:

```
:- relation(p1, X1, placement, z1, t)
relation(X1, X2, placement, z1, t)
relation(X2, X3, placement, z1, t)
relation(X3, X4, placement, z1, t)
relation(gripper, X4, grasp, top, t)
block(X1), block(X2), block(X3), block(X4)
```

Note that if the object instance has a feature that is not shared by all instances, this substitution is not allowed. For example, if an object instance is in its initial pose, it is unique with respect to reachability/graspability, and therefore cannot be substituted. This technique results in additional computational costs, because the ASP solver must ground the constraint with respect to all possible variable substitutions. Nevertheless, the planning performance is radically improved because generalized constraints have a stronger pruning effect.

6.6. Granularity of symbolic representations

An important issue for designing the symbolic domain is how detailed the symbolic representations should be, e.g. should the precise poses of objects be symbolically represented, should the modeling of grasping actions include the movement of the base, or the opening/closing of the gripper? A good point of comparison with our work is the work by Aker et al. (2012) which sometimes uses more detailed representations, and sometimes less detailed ones. For object/robots poses, they gridize the geometric space and each cell is represented in the symbolic domain with a row-column scheme, while the choice of grasp is entirely dealt with at the geometric level. Conversely in our approach, object/robots pose are dealt with by the geometric reasoner and the type of grasp is decided at the symbolic level. One may argue that this is solely an issue about delegating more or less computational effort to the task planner or to the geometric reasoner, and that the overall cost is the same. Next, we present some arguments for nuancing this statement.

An obvious limitation of using detailed symbolic representations is that the task planner is literally "drowned in details", and therefore cannot efficiently reason about the big picture, i.e. causal or temporal aspects of the problem. Another limitation of this approach is that it prevents us from using specialized (algebraic, constraint-based) methods for dealing more efficiently with the continuous aspects of CTAMP, which the task planner is not designed for.

On the other hand, our domain uses semi-detailed symbolic representations for some geometric aspects. For instance, we represent the gross orientation of objects and the type of grasp used for picking objects. This increases the complexity at the symbolic level, but it also simplifies the work of the geometric reasoner by excluding unfeasible actions from its search space (see Appendix C, 3 and 6). In specific cases, it can prove geometric tasks unfeasible only by means of causal reasoning, e.g. the robot Fabot cannot bring an object from upside-down to the upright position on its own.

7. Geometric domain

This section describes how object poses, actions, and states are represented at the geometric level. It also explains how the continuous configuration space is discretized.

7.1. Hybrid pose representation

In order to have a symbolic representation of the orientations of objects/TCPs, we use a hybrid discrete-continuous scheme to represent the pose of a body. We use bold lowercase letters to denote a column vector, e.g. $\boldsymbol{p}$, and bold capital letters to denote matrices, e.g. $\boldsymbol{T}$. All coordinates are expressed in the world frame. The pose of a body is obtained by applying a rotation, a translation and a template transformation to the body (see Figure 11). Hence,

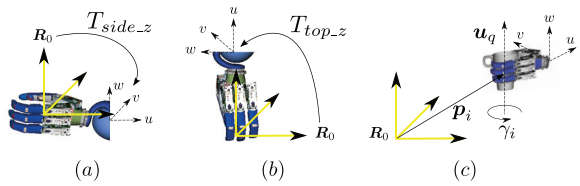


Fig. 11. Examples of template transformations $\boldsymbol{T}_{side\_z}$ (a) and $\boldsymbol{T}_{top\_z}$ (b) for the left gripper. Side grasp poses for instance, can be parametrized by $\boldsymbol{p}_i$, γ_i and $\boldsymbol{u}_q$, applying the translation $\boldsymbol{p}_i$, the rotation γ_i about $\boldsymbol{u}_q$ (z in this example), and the transformation $\boldsymbol{T}_{side\_z}$ to the gripper (c). (u, v, w) represents the body-fixed frame attached to the gripper.

the pose of a body o_i will be noted $(\boldsymbol{p}_i, \boldsymbol{T}_p, \boldsymbol{u}_q, \gamma_i)$, where $\boldsymbol{p}_i = (x_i, y_i, z_i) \in \mathbb{R}^3$ represents the translation of the i^{th} body. $\boldsymbol{T}_p$ represents a transformation of the body-fixed frame in the world frame, which we define as a *template transformation*. Template transformations represent natural positions of interest for objects and grippers, i.e. upright or upside-down for objects, and top, bottom, or side grasps for grippers. $\boldsymbol{u}_q \in \mathbb{R}^3$ is a unit vector which we define as *reference axis*, and $\gamma_i \in \mathbb{R}$ an angle of rotation about the axis $\boldsymbol{u}_q$. $\boldsymbol{T}_p$ belongs to a predefined set of transformations, and $\boldsymbol{u}_q$ is chosen among a predefined set of axes. Both are determined by the geometric reasoner depending on symbolic information. As an example, for sampling poses of an object to be placed in upright position on a table (see Figure 12), the geometric reasoner selects the upright template transformation of this object, and $\boldsymbol{u}_q = \boldsymbol{z}$ as template axis. The geometric reasoner computes the z parameter according to the pose of the table and the height of the object. Then, translations (x, y) and orientations (γ) are sampled and the transformation matrix $\boldsymbol{M}$ of the object is then given by

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{R}_{\boldsymbol{u}_q}(\gamma) & \boldsymbol{p} \\ \boldsymbol{O} & 1 \end{bmatrix} \boldsymbol{T}_{upright}$$

with

$$\boldsymbol{R}_{\boldsymbol{u}_q} = \boldsymbol{R}_z = \begin{bmatrix} cos(\gamma) & -sin(\gamma) & 0 \\ sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \boldsymbol{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The limitation of this representation is that all possible orientations cannot be represented, since finite sets of template transformations and reference axes are used. The advantage is that the orientations of the TCPs and objects can be represented at the symbolic level by ignoring the intrinsic orientation γ_i, using instead the gross orientation `x1, x2, y1, y2, z1` or `z2` (see Section 6.2). As explained in Section 6.6, symbolic reasoning about orientations presents some advantages. In the next sections, the pose of a body is simply noted as $(\boldsymbol{p}_i, \gamma_i)$ for clarity.

7.2. Representing actions and states

Let $\langle A_1, \ldots, A_n \rangle$ be a sequence of symbolic actions. We denote by s_j the geometric state resulting from applying

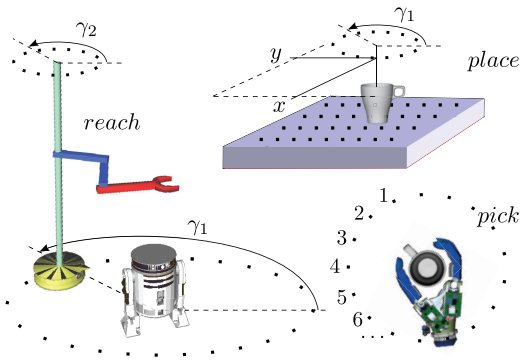


Fig. 12. The discretization schemes for different actions.

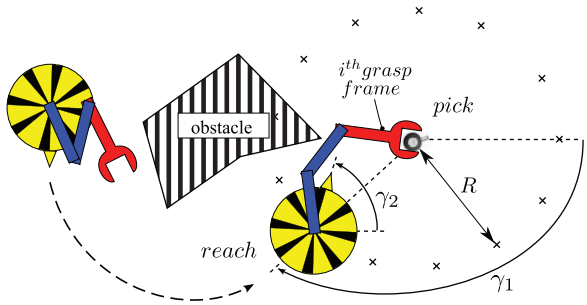


Fig. 13. Reach and pick example with Fabot and a cup, top view. Crosses represent possible locations for the mobile base for different values of γ_1. The orientation of the base is determined by γ_2.

the symbolic action A_j on the previous geometric state. We consider m rigid bodies. The i^{th} object is denoted by o_i, $i \in \{1, \ldots, m\}$. The position of object o_i in state s_j (i.e. after action A_j has been completed) is denoted by $\boldsymbol{p}_i^{(j)}$, and its orientation, by $\gamma_i^{(j)}$

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

Finally, we define a geometric state, or *configuration* as the set of values representing the poses of all objects, mobile bases, and TCPs

$$\begin{aligned} c = \{&\boldsymbol{p}_1, \gamma_1, \ldots, \boldsymbol{p}_m, \gamma_m, \\ &\boldsymbol{p}_{base1}, \gamma_{base1}, \boldsymbol{p}_{base2}, \gamma_{base2}, \ldots \\ &\boldsymbol{p}_{tcp1}, \gamma_{tcp1}, \boldsymbol{p}_{tcp2}, \gamma_{tcp2}, \ldots \\ &\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots \} \end{aligned}$$

where q_i represents the configuration chosen for the i^{th} robotic manipulator to place the gripper at $(\boldsymbol{p}_{tcpi}, \gamma_{tcpi})$. In addition to the geometric state, we also need to keep track of which objects are attached to which ones, in order to predict how the state will change when robots are actuated.

At the geometric level, a symbolic action A_j can be performed in various ways, e.g. a *pick* action can be performed with different orientations of the TCP, a *place* action can result in different positions/orientations of the object, and a *dock/reach* action can result in different positions/orientations for the mobile robot (see Figure 12). We denote *one* geometric instantiation of a symbolic action A_j by ${}^k a_j, k \in \{1, \ldots, r\}$, where r, the resolution, depends on the type of action and the resolution used for discretization. k is later referred to as the *action index*.

7.3. Domain dependent discretization

Discretization of grasps and placements is a limitation of this approach, but we emphasize the fact that discretization only concerns the *resulting configuration* of each action. In other words, the final motion plan consists of discretized configurations (one for each action) which are connected to each other by calling a bi-directional RRT algorithm (LaValle, 2006) working in the continuous domain.

When dealing with robot manipulation tasks, an important issue is to sample *transition configurations* at the intersection of different sub-spaces, e.g. the space of a mobile robot moving its base towards an object, and the space of its manipulator grasping the object. This problem is addressed in the multi-modal planning literature Hauser and Latombe (2010), the idea is to use intelligent strategies for sampling transition configurations between the different modes of the system. A similar but simpler approach is used here, the difference being that transitions between different modes are decided at the symbolic level. Next, we illustrate through an example the simple domain dependent strategies used for a "reach and pick" task, which involves two distinct modes.

In this example (see Figure 13), the mobile manipulator Fabot is to pick a cup which is out of reach. Unlike multi-modal planning, the problem is not defined by an initial and a goal configuration, but rather by an initial configuration, an initial symbolic state, and a symbolic plan (computed by the ASP solver). In the initial symbolic state, the cup is not `reachable` by Fabot (symbolically), while the action `pick` only applies to reachable objects. The action `reach` makes an object `reachable` by a robot. Hence, the ASP solver computes a symbolic plan consisting of a `reach` action followed by a `pick` action. These actions are discretized (see Figure 12 and Table 1) as follows;

1. The `reach` action is discretized into 40 poses parametrized by two angular values γ_1 and γ_2. The poses are distributed on a circle[3] centered around the reached object, with radius R depending on the type of robot performing the action (see Figure 13). R was empirically determined such that the gripper affords a wide range of approach directions, while the base remains far enough from the object to minimize the risk of collision with a potential supporting object.
2. The `pick` action is discretized into 16 grasp frames, which are pre-computed for all possible gripper-object pairs. These grasp frames are such that the gripper does not collide with a potential flat surface under the object. For objects with axial symmetry, the grasp frames are obtained by incremental rotations of a template grasp

Table 1. Parametrization and typical resolutions used for discretization. The "index" column refers to a list of pre-computed grasp frames.

Action	x	y	z	γ_1	γ_2	Index	Total
Pick	–	–	–	–	–	16	16
Stack	1	1	1	16	–	–	16
Place	7	7	1	16	–	–	784
Dock / Reach	–	–	–	8	5	–	40

frame. For other objects (e.g. a tray), the grasp frames are manually created for different graspable areas of the object. The grasp frame is then applied to the gripper, and inverse kinematic (IK) solutions are computed for the manipulator. The first collision-free solution is selected.

The problem boils down to finding a resulting configuration for the `reach` action, from which a feasible grasp can be performed. With the typical resolutions shown in Table 1, this may require 40×16, namely 640 configurations to check in the worst case. This is done by the *GeometricBacktracking()* algorithm (see Algorithm 1, Section 9.1), which proceeds in a depth-first search manner. In a nutshell, each action is computed "backwards", i.e. (i) the resulting configuration is computed first, (ii) collisions are checked, (iii) a path from the current configuration to the resulting configuration is computed. In case of success, the next action is processed, otherwise another resulting configuration is tried. If none of the configurations work, the algorithm backtracks to the previous action. Configurations are chosen according to van der Corput sequences (Kuipers and Niederreiter, 1974), which guarantee a uniform distribution of the samples.

This approach for multi-step motion planning is incomplete because of the naive discretization and because the modes are enforced by the symbolic level, e.g. a robot cannot fold/unfold its manipulator for getting through a narrow corridor if the implementation of the `reach` action does not allow for it. We do not use sophisticated sampling strategies for sampling grasp configurations, e.g. using loop-closure constraints (Cortés and Siméon, 2004), but simply select among the set of discretized grasps, those which accept an IK solution. However, although the focus of this work is not multi-modal planning, tasks requiring complex object manipulation could be solved, as shown in the experimental evaluation (Section 11). We also refer the reader to the GeRT (Generalizing Robot manipulation Tasks) project for more details about the application of these techniques on the real robotic platform Justin (Ott et al., 2006) and provide video links for concrete illustration[4].

In the next sections, we describe the techniques used for culprit detection in the different layers of the geometric reasoner (see Figure 5): spatial constraints in Section 8, and unavoidable collisions in Section 10. Then, Section 9 focuses on geometric dependencies chains, which is not a culprit detection technique, but a way of selecting subsequences of actions from the symbolic plan.

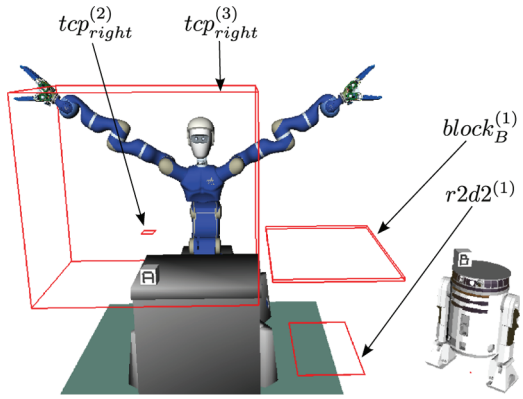


Fig. 14. Representation of the spatial constraints as bounding boxes.

8. Culprit detection with spatial constraints

This section describes the first test performed by the geometric reasoner: the "consistency check" (see Figure 5, layer (1)). The problem is relaxed by representing the poses of objects/robots by a set of bounding boxes. A network of linear constraints is built, from which inconsistencies are detected using linear programming techniques. These inconsistencies reveal violations of kinematic constraints or reachability problems. It is crucial that the bounding boxes always cover a *larger* space than the space actually occupied by objects/robots, in order to guarantee that only unfeasible geometric states are rejected by the constraints.

8.1. Building the linear constraint network

The spatial constraints are the geometric counterparts of the spatial relations introduced in Section 6.4. Hence, we also define three types of spatial constraints: *grasp, placement,* and *dock* (see Figure 9). These constraints are automatically generated from the symbolic plan. One can see them as a set of bounding boxes, which encompass all the possible poses in which each object/base/TCP can be. For instance, a *placement* constraint can be visualized as a polyhedral region encompassing the location in which the center point of the object has to be after the corresponding *place* action has been executed. The placement constraint on the pose of

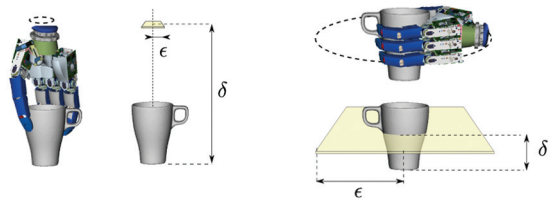


Fig. 15. Polyhedral region for the TCP relative to the object during a *grasp* action.

an object o_i with respect to a *fixed* location at step j can thus be written as a linear inequality

$$\boldsymbol{a}^{(j)} \leq \boldsymbol{p}_i^{(j)} \leq \boldsymbol{b}^{(j)} \tag{2}$$

with $\boldsymbol{a}^{(j)} = (loc_{xmin}, loc_{ymin}, loc_{zmin})$
and $\boldsymbol{b}^{(j)} = (loc_{xmax}, loc_{ymax}, loc_{zmax})$

where $\boldsymbol{a}^{(j)}$ and $\boldsymbol{b}^{(j)}$ define a bounding box around the location in the world frame. In most cases, the constraints are between two objects that can move, e.g. a *grasp* constraint between an object at pose $\boldsymbol{p}_k$ and a TCP at pose $\boldsymbol{p}_i$ at step j can be written as

$$\boldsymbol{p}_k^{(j)} + \boldsymbol{c}^{(j)} \leq \boldsymbol{p}_i^{(j)} \leq \boldsymbol{p}_k^{(j)} + \boldsymbol{d}^{(j)} \tag{3}$$

with $\boldsymbol{c}^{(j)} = (-\epsilon, -\epsilon, \delta)$
and $\boldsymbol{d}^{(j)} = (\epsilon, \epsilon, \delta)$

where ϵ and δ are some parameters that can be extracted from the grasp frame (see Figure 15). Note that depending on the type of constraint, the bounding box is not necessarily centered around the object. The poses of unmovable objects and the initial poses of movable objects are modeled as variables subject to unary equality constraints, e.g.

$$\boldsymbol{p}_i^{(0)} = \boldsymbol{p}_{init} \tag{4}$$

where $\boldsymbol{p}_{init}$ is a constant.

The linear constraint network is initialized with the initial poses of objects/bases/TCPs, and built by iterating over the actions of the symbolic plan. For each action, one or several constraints are added. A new set of variables is created for each object/base/TCP that is moved. From now on, we use the term "variable" to denote the translation $\boldsymbol{p}$ of an object, which actually consists of three variables (x, y, z). Let us describe this process with an example. Consider for instance the symbolic plan:

A_1: *dock (r2d2, left_base)*
A_2: *pick (right, top, block_a, table)*
A_3: *stack (right, top, block_b, z1, block_a)*

In the initial state s_0, the bounding box of each variable is a *point* corresponding to the initial pose. After the *dock* action A_1, r2d2 is moved, and so is block B, which is placed on r2d2[5], hence two variables $r2d2^{(1)}$ and $block_B^{(1)}$ are created. A *dock* constraint is posted to the constraint network, which imposes the new variable $r2d2^{(1)}$ to be within a bounding box centered around $left\_base^{(0)}$. $left\_base^{(0)}$ is a variable representing the pose of the first link of the left arm of Justin. A *placement* constraint $\mathcal{P}^{(1)}$ is created between r2d2 and block B, since the block is `connected` to the robot (this is known from the symbolic state). After the *pick* action A_2, a new variable $tcp_{right}^{(2)}$ is created since the right TCP is moved. A *grasp* constraint is added to the network, that constrains the right TCP to be within a bounding box located above block A (see Figure 14). The exact size and position of this bounding box is determined using the predefined grasp frames of this object class. Finally, after the *stack* action A_3, the right TCP and block A are moved, hence new variables are created for both. Two constraints are created: a *grasp* constraint $\mathcal{G}^{(3)}$ between the TCP and the object, and a *placement* constraint $\mathcal{P}^{(3)}$ between block B and block A. The kinematic constraints for manipulators ($\mathcal{K}^{(2)}$ and $\mathcal{K}^{(3)}$) can be modeled as a box centered on the first joint of the manipulator with dimensions depending on the length of the manipulator, although a better approximation is possible (Lagriffoul et al., 2012). The resulting constraint network is shown in Figure 16.

We define the vector of the variables representing the poses of all objects/bases/TCPs in the problem

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_N)$$

The bounding boxes are represented by a set of intervals that define an upper bound and a lower bound on these variables, which we call the domain $\mathcal{D}$ of the problem

$$\mathcal{D} = \langle [\underline{x_1}, \overline{x_1}], [\underline{x_2}, \overline{x_2}], \ldots, [\underline{x_N}, \overline{x_N}] \rangle$$

The set of all linear constraints of the problem

$$\mathcal{C} = \{\mathcal{G}^{(j)}, \mathcal{P}^{(j)}, \mathcal{D}^{(j)}, \mathcal{K}^{(j)}\},\ j \in \{1, \ldots, n\}$$

can be expressed as

$$\boldsymbol{D}\boldsymbol{x} \leq \boldsymbol{e} \tag{5}$$

where $\boldsymbol{D}$ and $\boldsymbol{e}$ aggregate all the spatial constraints of the problem (see expressions (2), (3) and (4)).

8.2. Culprit detection in the linear program

Identifying a culprit subset of constraints in a constraint network is a difficult problem in general. In the case of linear programming, there exists efficient methods (implemented in most solvers) to compute a so called Irreducible Infeasible Set (IIS). An IIS is an infeasible subset of constraints, from which removing one constraint makes the unfeasible problem feasible. IISs are useful for diagnosing a potential cause of infeasibility in simple cases, but often a problem has many IISs (potentially an exponential number) and finding the actual cause of failure requires a tedious find-and-repair process. IISs are essentially a tool for finding *modeling errors*, which is not useful here. Imagine for

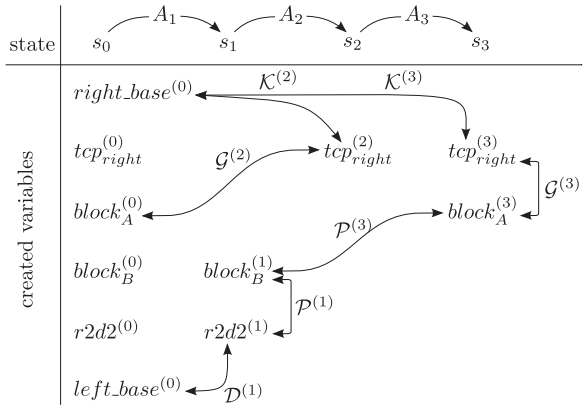


Fig. 16. The *spatial constraints graph* for the plan $\langle A_1, A_2, A_3 \rangle$ from Section 8.1. The arrows represent the constraints created for each action: grasp constraints ($\mathcal{G}$), placement constraints ($\mathcal{P}$), dock constraints ($\mathcal{D}$), and kinematic constraints ($\mathcal{K}$). The variables in the figure represent the three components of the translation x, y and z. The nodes indexed by 0 are constants.

instance that the problem illustrated in Figure 14 gives rise to an inconsistency, i.e. the right TCP cannot reach r2d2 in order to stack block A on block B. In this case, an IIS would tell us that the inconsistency could be removed if the docking area was larger, or if the right arm of Justin was longer, etc. These constraints are not modeling errors, but the real constraints of the problem. Rather, what is useful here is to determine the *culprit set* of constraints which causes inconsistency. The solution to this problem is to compute a set of constraints which contains at least one constraint from each IIS in the model (Chinneck, 1996). This problem, referred to as the IIS set covering problem, is known to be NP-hard (Chakravarti, 1994). This problem is related to computing the maximum cardinality feasible subsystem (see, e.g. Parker and Ryan (1996)). In the present work, we take advantage of the specific structure of the constraint network to devise a simpler technique.

8.3. Identifying the culprit set in a line-network

While constraints are added to the network, a hypergraph of the constraint network is built, in which the edges represent constraints of type bounding box, and the nodes contain the variables of the pose of an object at a certain time step (see Figure 16). We call this graph the *spatial constraints graph*. Each time a constraint $\mathcal{C}$ is posted, a consistency check is performed. Therefore, when an inconsistency is detected, we know that $\mathcal{C}$ belongs to the set of culprit constraints. Then, the graph is used for tracing back all the constraints in relation with $\mathcal{C}$, until a unary equality constraint (4) is reached. When such a constraint is reached, there is no need to continue the process, since the associated variable is constant, hence its value cannot be affected by other constraints. We call the constraint network resulting from this process the *candidate culprit set*. Let us consider the example in Figure 14. Imagine that an inconsistency is detected

$$
\begin{aligned}
c_0 &: \quad \boldsymbol{p}_0 = \boldsymbol{p}_{init} \\
c_1 &: \quad \boldsymbol{p}_0 - \boldsymbol{c}^{(1)} \leq \boldsymbol{p}_1 \leq \boldsymbol{p}_0 + \boldsymbol{d}^{(1)} \\
c_2 &: \quad \boldsymbol{p}_1 - \boldsymbol{c}^{(2)} \leq \boldsymbol{p}_2 \leq \boldsymbol{p}_1 + \boldsymbol{d}^{(2)} \\
&\ldots \\
c_{i-1} &: \quad \boldsymbol{p}_{i-2} - \boldsymbol{c}^{(i-1)} \leq \boldsymbol{p}_{i-1} \leq \boldsymbol{p}_{i-2} + \boldsymbol{d}^{(i-1)}
\end{aligned}
$$

Fig. 17. Linear inequalities resulting from splitting a line-constraint network in two parts. $\boldsymbol{p}_{init}, \boldsymbol{c}^{(1)}, \boldsymbol{d}^{(1)}, \ldots, \boldsymbol{c}^{(i-1)}, \boldsymbol{d}^{(i-1)}$ are constants.

when the placement constraint $\mathcal{P}^{(3)}$ is posted. Then, one can trace back the following candidate culprit set

$$S = \{\mathcal{K}^{(3)}, \mathcal{G}^{(3)}, \mathcal{P}^{(3)}, \mathcal{P}^{(1)}, \mathcal{D}^{(1)}\}$$

Tracing the candidate culprit set is a preliminary step before finding the culprit set, which eliminates irrelevant constraints ($\mathcal{K}^{(2)}$ and $\mathcal{G}^{(2)}$ in this example). In order to prove that the candidate culprit set S is the *culprit set*, we have to prove that there is no smaller subset of S causing inconsistency. This may require many consistency checks, because it requires checking all the subsets in the power set of S. Another approach consists in proving that all the subsets of cardinality $n - 1$ are consistent. This proves that S is the smallest inconsistent set, hence the culprit set. In other words, it is only needed to show that removing *any one* of the constraints in S removes the inconsistency. We propose a simple way to proceed in case the candidate culprit set is a *line-network*, i.e. a network of which the topology is a tree with branching factor equal to 1.

Proposition 1: *In an inconsistent line-network with constraints of type bounding box, removing any one of the constraints removes the inconsistency.*

Proof: Let $s = \{c_0, \ldots, c_n\}$ be a line constraint network. Removing a constraint c_i from s always results in two line-networks $\{c_0, \ldots, c_{i-1}\}$ and $\{c_{i+1}, \ldots, c_n\}$, where c_0 and c_n are unary equality constraints of type (4) (because they correspond to the initial pose of an object), and the constraints $c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{n-1}$ are binary constraints of type "bounding box" (see equation (2) or equation (3)). Figure 17 represents the linear inequalities composing such a line-network

The two subsets of constraints are trivially feasible because it is always possible to recursively construct a solution $(\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{i-1})$ (see Figure 18)

$$\text{with } \boldsymbol{p}_k = \frac{1}{2}(\boldsymbol{d}^{(k)} - \boldsymbol{c}^{(k)}) + \boldsymbol{p}_{k-1}, \quad k = 1, \ldots, i-1 \qquad (6)$$

■

Therefore, if the candidate culprit set is a line-network, then it is necessarily the culprit set.

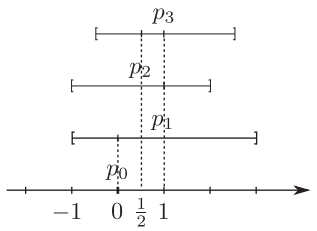


Fig. 18. Example of a trivial solution with real intervals instead of bounding boxes: $p_1 = \frac{1}{2}(3-1)+0 = 1, p_2 = \frac{1}{2}(1-2)+1 = \frac{1}{2}, p_3 = \frac{1}{2}(2-1)+\frac{1}{2} = 1$. But if a unary constraint is imposed on p_3, the system may be inconsistent, for example with $p_3 = x, x < -10$ or x > 10.

.

With the culprit set, one can automatically generate a logical constraint for the task planner, using the mapping between geometric constraints and spatial relations:

$\mathcal{G}^{(3)} \rightarrow$ `relation(right, blockA, grasp, top, 3)`
$\mathcal{P}^{(3)} \rightarrow$ `relation(blockB, blockA, placement, z1, 3)`
$\mathcal{P}^{(1)} \rightarrow$ `relation(r2d2, blockB, placement, z1, 1)`
$\mathcal{D}^{(1)} \rightarrow$ `relation(r2d2, left base, dock, 1)`

and the mapping between variables and symbolic objects:

right_base$^{(0)} \rightarrow$ `right_base`
left_base$^{(0)} \rightarrow$ `left_base`

The logical constraint is constructed as a conjunction of terms, which simply are the spatial relations associated to the geometric constraints. Besides, using **Proposition 1**, we observe that the inconsistency in the culprit set can be removed if either c_0 or c_n are removed. This means that the inconsistency can be avoided if one of the objects that are in their initial pose was moved. This information is included into the constraint as well (in the last two lines). Finally, the following expression is generated:

```
:- relation(right, blockA, grasp, top, t)
relation(blockB, blockA, placement, z1, t)
relation(r2d2, blockB, placement, z1, t)
relation(r2d2, left_base, dock, t)
not 1{moved(left_base, 1..t-1); moved(right_base,
1..t-1)}
```

In natural language, this means that the ASP solver cannot return a plan in which r2d2 is docked at the left base, *while* B is placed on r2d2 and A is placed on B, *while* the right TCP is grasping B, *unless* the left base *or* the right base is moved in a previous time step.

Note that in this constraint, the relations are indexed by the same variable `t`, whereas the corresponding constraints have been posted in the network at different time steps. This is not wrong, since the spatial relations *persist* at the symbolic level until they are destroyed according to the rules defined in Section 6.4. Therefore, this constraint prevents the task planner from returning the plan $\langle A_1, A_2, A_3 \rangle$, as well as all the plans that cause these relations to be true at the same time. This is another reason why spatial constraints are effective: they make abstraction of *time* to a certain extent, hence they prune out entire families of plans regardless of the specific ordering of actions.

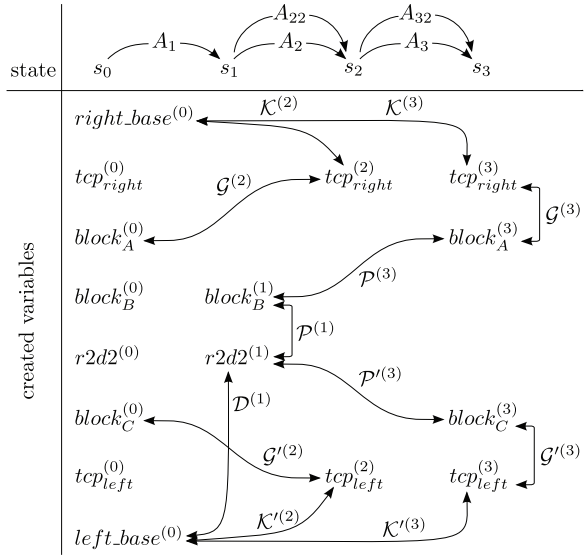
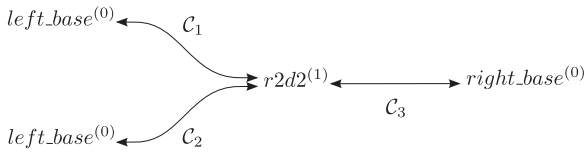


Fig. 19. The *spatial constraints graph* for the sequence of actions $\langle A_1, A_2, A_{22}, A_3, A_{32} \rangle$.



$\mathcal{C}_1 = \mathcal{D}^{(1)}$
$\mathcal{C}_2 = \{\mathcal{P}'^{(3)}, \mathcal{G}'^{(3)}, \mathcal{K}'^{(3)}\}$
$\mathcal{C}_3 = \{\mathcal{P}^{(1)}, \mathcal{P}^{(3)}, \mathcal{G}^{(3)}, \mathcal{K}^{(3)}\}$

Fig. 20. The *simplified spatial constraint graph* after tracing back and simplification.

8.4. Identifying the culprit set in a tree

In more complicated cases, the *candidate culprit set* is not a line-network, but a tree[6]. In this case, there may be several possible culprit sets. Consider for instance the following sequence of actions:

A_1: *dock (r2d2, left_base)*
A_2: *pick (right, top, block_a, table)*
A_{22}: *pick (left, top, block_c, table)*
A_{32}: *place (left, top, block_c, z1, r2d2)*
A_3: *stack (right, top, block_b, z1, block_a)*

This is the same plan as in the previous example, with the addition that in parallel with actions A_2 and A_3, Justin picks and places another block C from the table onto r2d2 with the left arm (actions A_{22} and A_{32}). The corresponding constraint network is given in Figure 19. If the constraints associated to A_3 are posted before the constraints associated to A_{32}, then an inconsistency is detected and we can trace back a set of constraints which is a line-network, as in the previous

example. Otherwise, the following candidate culprit set is traced back

$$S' = \{\mathcal{K}^{(3)}, \mathcal{G}^{(3)}, \mathcal{P}^{(3)}, \mathcal{P}^{(1)}, \mathcal{D}^{(1)}, \mathcal{P}'^{(3)}, \mathcal{G}'^{(3)}, \mathcal{K}'^{(3)}\}$$

Figure 20 represents a simplification of the candidate culprit set. Note that the variable $left\_base^{(0)}$ can be seen as two different leave nodes because it is a constant. In this example, the culprit subset is obviously $\{\mathcal{C}_1, \mathcal{C}_3\}$ because the actions A_{22} and A_{32} do not resolve the kinematic problem of the right manipulator being unable to reach r2d2. But automatically finding the culprit set is not easy in the general case. It requires finding the smallest inconsistent subset, i.e. performing consistency checks on the power set of the set of constraints, considering subsets of increasing size. In this way, the inconsistent set with the smallest possible cardinality can be found. We refer to this set as the *optimal* culprit set, but there may be other culprit sets, which we refer to as *minimal* culprit sets. These sets are minimal in the sense that removing any constraint from them removes the inconsistency, but their cardinality is not minimal.

Finding a culprit set in a tree constraint network is computationally expensive, but in all the scenarios addressed in this paper, the structure of the problem allows us to use a simpler technique. The reason is that in the problems we address, *the culprit set is always a line-network*. Indeed, the topology of the constraint network maps to the kinematic relations between robots/objects at the time of inconsistency. This means that a culprit constraint network consisting of three or more branches, would result from a situation in which three or more robots are *simultaneously* interacting with the same object, which never occurs, because it is not allowed by the symbolic domain.

Using the assumption that the culprit set is a line-network, we can use **Proposition 1** and iterate over all the constraints in order to test if they belong to the culprit subset or not, hence isolating the culprit set from the candidate culprit set. We use the following procedure

Given a candidate culprit set $S = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ *with bounding box constraints, for each* $\mathcal{C}_i \in S$*:*
if $S \setminus \{\mathcal{C}_i\}$ *remains inconsistent, then* $S \leftarrow S \setminus \{\mathcal{C}_i\}$

which requires n consistency checks. The constraint network resulting from this process is a minimal culprit set. Once a minimal culprit set is found, a logical constraint is automatically generated as previously explained, and returned to the ASP solver. Note that there may exist several culprit sets depending on the order in which the constraints are tested. It is possible to enumerate all of them and send them in bulk to the ASP solver, or simply return the first one and detect the other ones during the next iterations.

In summary, detecting the culprit spatial relations is achieved by detecting inconsistencies in a constraint network representing the spatial constraints of the problem. This is done efficiently for two reasons. First, inconsistencies are detected using linear programming, for which efficient methods with polynomial worst-case complexity exist (Boyd and Vandenberghe, 2004). The second reason is that we maintain a graph representing the constraint network, and take advantage of its structure to trace back a candidate culprit set. If the candidate culprit set is a line-network, it is the culprit set itself. If it is a tree, the culprit set can be found with a simple procedure with linear worst-case complexity.

9. Culprit detection in geometric dependencies chains

This layer of the geometric reasoner also works on a relaxation of the problem, but unlike the "Spatial relations" layer, the actions are evaluated with their exact kinematic constraints, and a search in the space of grasps and placements (geometric backtracking) is performed. The relaxation consists of (i) isolating subsequences of actions in the symbolic plan, (ii) not performing motion planning (only the final configurations resulting from actions are considered, i.e. grasp/release positions for *pick/place* actions, or final pose of the robot for *dock/reach* actions). First, we introduce the geometric backtracking process.

9.1. Geometric backtracking

Geometric backtracking is a search process which allows us, when an action fails, to reconsider the choices made at the geometric level for previous actions (Bidot et al., 2015; Karlsson et al., 2012). In the present work, we reconsider the choices made for grasps and placements. It is also possible to reconsider the choices of inverse kinematic solutions for manipulators, but we found that few problems need this feature to be solved. Algorithm 1 implements geometric backtracking in a systematic depth-first search manner, but it can be implemented in different ways, e.g. by combining several probabilistic roadmaps (Cambon et al., 2009).

The function is initially called with the initial *configuration* (see Section 7.2) which gives a geometric description of the initial scene, a symbolic plan (from the ASP solver), and an empty list *sol*. At each call, the function takes the first action A in the sequence $\mathcal{S}$, and keeps the remaining list $\mathcal{T}$ for the recursive call (line 9). The resolution r of an action is the number of ways a symbolic action can be geometrically instantiated. Then, the function iterates over all the possible *action indexes k* (see Section 7.2) for the action A. The function *resultConfig*( ) (line 5) returns the configuration resulting from applying the geometric action ${}^{k}a$ on the configuration c, or *null* if the action is unfeasible. If the action is feasible, the temporary solution sol' is appended with the current action index (line 7). In case some actions remain to be evaluated ($\mathcal{T} \neq \emptyset$), the function is recursively called with the new configuration and the list of remaining actions. In case of failure, the next action index k is tried. If all of them fail (line 14), the *null* value is returned to the calling function through line 9, and the calling function tries the next action index. If the last action is reached

Algorithm 1: GeometricBacktracking

```
Function GeometricBacktracking(c, S, sol)
  input : c: a configuration
          S: a sequence of symbolic actions
          sol: a list of action indexes
1  A = S.head()
2  T = S.tail()
3  r = A.resolution
4  for k ← 1 ... r do
5      c' = resultConfig(^k a, c)
6      if c' ≠ null then
7          sol' ← sol ∪ k
8          if T ≠ ∅ then
9              temp = GeometricBacktracking(c', T, sol')
10             if temp ≠ null then
11                 return temp
12         else
13             return sol'
14 return null
```

($\mathcal{T} = 0$, line 13), the solution is returned to the initial calling function through line 11 in the form of a list of action indexes, which indicates which grasp/placement to use for each symbolic action.

GeometricBacktracking() is a depth-first search algorithm with no heuristic to guide the search. Although some work has been initiated in this direction (Bidot et al., 2015; Lagriffoul et al., 2012), geometric backtracking remains a difficult problem because of the large branching factor and because geometric computations such as motion planning are not computationally reducible. In practice, *GeometricBacktracking()* cannot complete in reasonable time if the depth exceeds 4-5 actions. Therefore, a cutoff time has to be used. However, if the problem contains few *geometric dependencies*, then less geometric backtracking is needed, and it is possible to find a solution for a symbolic plan containing dozens of actions. Next, we define different types of geometric dependencies and the concept of *geometrically ground sequence of actions*, which is used to isolate subsequences of actions to be separately evaluated.

9.2. Geometric dependencies

This section refers to the notion of *Geometric Reachable Set* and *Geometric dependency between two actions*, which are formally defined in Appendix B. Next, we will consider three types of geometric dependencies in particular:

(A) dependencies based on reachability;
(B) dependencies based on body connection;
(C) dependencies based on collisions.

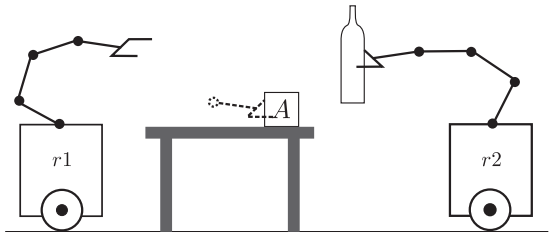


Fig. 21. Illustration of a part of the symbolic plan $\langle A_1, \ldots, A_n \rangle$.

We illustrate these through an example. Consider for instance the following subsequence of symbolic actions, illustrated in Figure 21. We assume that the plan contains some other actions performed by other robots on other objects, and that block A was not manipulated prior to action A_{c5}:

...
A_{c1}: *pick (r2, side, bottle, cellar)*
...
A_{c2}: *reach (r2, table)*
A_{c3}: *place (r2, bottle, z1, table)*
...
A_{c4}: *reach (r1, table)*
A_{c5}: *pick (r1, side, block_a, table)*
...

The first type of geometric dependency (A) exists between actions A_{c2} and A_{c3}, or A_{c4} and A_{c5}. The geometric reachable set for the actions A_{c3} and A_{c5} is affected by the geometric instance chosen for the actions A_{c2} and A_{c4}, because the poses that the TCP can *reach* depend on the pose of the base relative to the table. The second type of geometric dependency (B) exists between actions A_{c1} and A_{c3}, because the set of poses in which the bottle can be placed on the table depends on how the bottle has been grasped, even if the action A_{c2} is performed in the same way. This depends on the *connection* between the gripper and the bottle. The third type of geometric dependency (C) exists between actions A_{c3} and A_{c5} because placing the bottle on the table may cause *collisions* between the bottle and $r1$, which may change the geometric reachable set of the action A_{c5}. We denote a geometric dependency of type T between A_i and A_j by: $A_i \overset{T}{\rightsquigarrow} A_j$.

According to **definition 2**, geometric dependencies of types A and B are *direct* geometric dependencies, since the geometric reachable set is only affected because of kinematic issues. Note also that in a direct geometric dependency $A_i \overset{dir.}{\rightsquigarrow} A_j$, A_j cannot be geometrically instantiated if A_i is not geometrically instantiated. Consider for example the relation $A_{c1} \overset{B}{\rightsquigarrow} A_{c3}$: without a geometric instance for A_{c1}, the position of the bottle within the gripper is unknown, hence no geometric instance can be defined for A_{c3}. Similarly with $A_{c4} \overset{A}{\rightsquigarrow} A_{c5}$, the geometric instantiation of the *pick* action requires the pose of the robot to be defined.

Some actions require several actions in order to be instantiated, then we denote it by $\{A_{i_1}, \ldots, A_{i_k}\} \overset{dir.}{\rightsquigarrow} A_j$. In contrast, when two actions have a non-direct geometric dependency, the second action may be instantiated even if the first action is not. Consider $A_{c3} \overset{C}{\rightsquigarrow} A_{c5}$ for instance: A_{c5} can be geometrically instantiated without the pose of the bottle on the table being defined.

We say that an action is *independent* (with respect to the plan containing it) if it has no direct dependency with other actions. In our example, A_{c2} and A_{c4} are independent because they only depend on the position of the table, which cannot be moved (although they may have non-direct dependencies with other actions, because of collisions).

Definition 3 *Geometrically ground subsequence of actions* Let $\mathcal{P} = \langle A_1, \ldots, A_n \rangle$ be a sequence of actions, and $\mathcal{Q} = \langle A_{i_1}, \ldots, A_{i_q} \rangle$ a subsequence extracted from $\mathcal{P}$, i.e. $1 \leq i_1 < \cdots < i_q \leq n$. The subsequence $\mathcal{Q}$ is *geometrically ground* iff $\forall A_j \in \mathcal{Q}$,

A_j is independent or

$\forall A_p \in \mathcal{P}$ such that $A_p \overset{dir.}{\rightsquigarrow} A_j$, we have $A_p \in \mathcal{Q}$.

In other words, a ground sequence of actions is "self-contained", and can be geometrically instantiated even if it is a subsequence of actions extracted from a larger sequence. Examples of ground subsequences of action are given in the next subsection. For the sake of brevity, we simply use the term "ground" in the remainder of the article.

9.3. Finding culprit subsequences

In order to find a culprit subsequence of actions in a symbolic plan $\langle A_1, \ldots, A_n \rangle$, one needs to find a minimal subsequence of actions $\langle A_{c1}, \ldots, A_{cq} \rangle$ which is unfeasible. In order to prove that a subsequence is not feasible, we deactivate collision detection for objects which are not manipulated in the subsequence, and use the algorithm *GeometricBacktracking()*, which searches a possible geometric instantiation of that subsequence. If it returns *false*, then the subsequence would a fortiori be unfeasible if executed within the original plan, and therefore is a culprit one.

The difficulty is that one must work with the power set of $\{A_1, \ldots, A_n\}$, i.e. checking all the subsequences with one action, two actions, three actions, etc. However, the basic problem is not about the combinatorial, but rather about finding subsequences of actions which are ground, because a subsequence which is not ground cannot be geometrically instantiated in the first place. Consider again the problem in Figure 21 for example. The plan fails because wherever the bottle is placed on the table, the bottle prevents $r1$ to grasp block A. Intuitively, the culprit subsequence seems to be $\langle A_{c3}, A_{c5} \rangle$. In order to prove it, one needs to show that all the possible combinations of the geometric actions ${}^{k}a_{c3}$ and ${}^{k'}a_{c5}$ are unfeasible. The problem is that $\langle A_{c3}, A_{c5} \rangle$ is not ground because it cannot be geometrically instantiated without the actions A_{c1}, A_{c2} and A_{c4}.

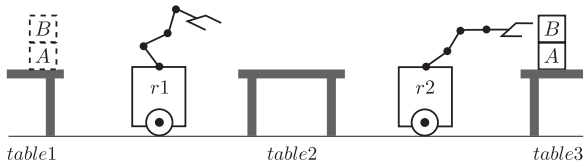


Fig. 22. A manipulation problem with two robots. The goal state is to have both blocks stacked on *table1* (dashed line).

Therefore, in order to detect culprit subsequences, one must first find the ground subsequences, i.e. identifying the *direct* geometric dependencies that exist between the actions of the plan. Consider the actions of the symbolic domain and their parameters (we limit ourselves to *pick, place*, and *reach* for clarity):

pick (Robot, Grasp_type, Object, Location)
place (Robot, Object, Axis, Location)
reach (Robot, Location)

The *direct* dependencies can be determined by mapping the parameters of actions in the symbolic plan with domain knowledge provided by the user about how actions affect object reachability and body connection (see Tables 2 and 3). Using this information, the *graph of direct geometric dependencies* $\mathcal{G}_{dir}$ can be automatically constructed.

We illustrate the construction of this graph with an example. Consider the problem illustrated in Figure 22. The goal is to have block B stacked on block A on *table*1. A possible symbolic solution plan for this problem could start with the following action sequence:

A_1: *reach (r1, table2)*
A_2: *reach (r2, table3)*
A_3: *pick (r2, top, block_b, block_a)*
A_4: *reach (r2, table2)*
A_5: *place (r2, block_b, z1, table2)*
A_6: *pick (r1, top, block_b, table2)*
A_7: *reach (r2, table3)*
A_8: *reach (r1, table1)*
A_9: *pick (r2, top, block_a, table3)*
A_{10}: *reach (r2, table2)*
A_{11}: *place (r1, block_b, z1, table1)*
...

The first step in constructing $\mathcal{G}_{dir}$ is to determine for each action, using Table 3 and the parameters of the symbolic actions, which objects are moved, and which connections between bodies are created (see Figure 23). Then, for each action A_j, using the table of dependencies (Table 2), the previous actions are listed for finding the last action(s) that changed what A_j depends on (arrows in Figure 23). Note that this process runs in time quadratic in the number of actions of the plan.

As an example, the action A_5 *place(r2, block_b, z1, table2)* depends on action A_4 because A_4 moved the robot's base (type (A) dependency) and on action A_3 because A_3 created a connection between the TCP and the object (type

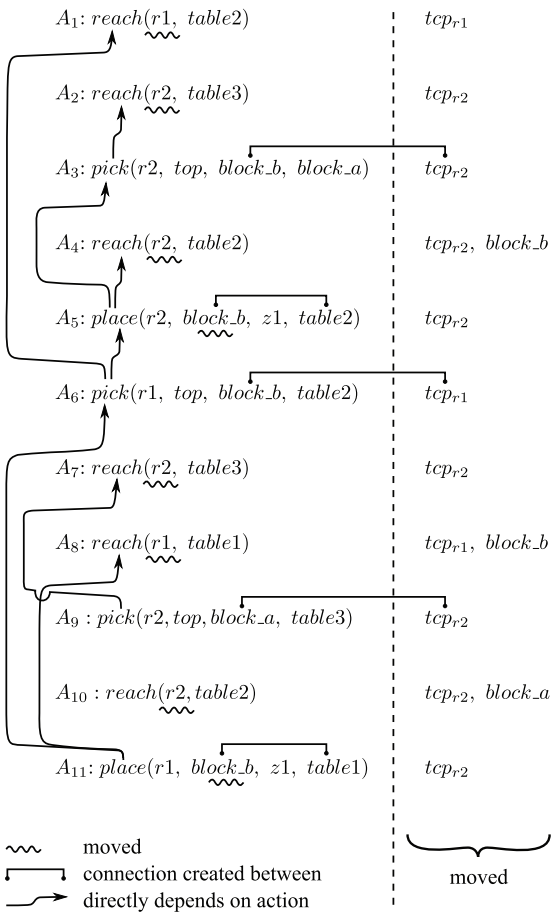


Fig. 23. Graph of *direct* geometric dependencies. The column on the right indicates the objects/bodies that are moved because of their connection with a mobile base.

Table 2. Actions and their associated dependencies.

Action	Depends on pose of	Depends on connection of
pick	*Robot_base, Object*	*Location* and *Object*
place / stack	*Robot_base, Location*	*TCP* and *Object*
reach / dock	*Location*	∅

Table 3. Actions and their effects on pose and connection.

Action	Changes the pose of	Creates a connection between
pick	*TCP*	*TCP* and *Object*
place / stack	*TCP, Object*	*Location* and *Object*
reach / dock	*Robot_base*	∅

(B) dependency). A_5 also depends on the pose of the location *table*2, but in this plan, *table*2 was not moved in the previous actions, hence no dependency is represented. Then, a set of ground subsequences $\mathcal{S}_{ground}$ can be built by tracing back the direct dependencies from each action:

$\mathcal{S}_{ground} = \{$
$\langle A_1 \rangle,\ \langle A_2 \rangle,\ \langle A_2, A_3 \rangle,\ \langle A_4 \rangle,\ \langle A_2, A_3, A_4, A_5 \rangle,$
$\langle A_1, A_2, A_3, A_4, A_5, A_6 \rangle,\ \langle A_7 \rangle,\ \langle A_8 \rangle,$
$\langle A_7, A_9 \rangle,\ \langle A_{10} \rangle,$
$\langle A_1, A_2, A_3, A_4, A_5, A_6, A_8, A_{11} \rangle\}$

As an example, the subsequence $\langle A_7, A_9 \rangle$ is ground because it can be instantiated regardless of other actions (it depends on the pose of *table3* which is unmovable, and it depends on the pose of *block_a* which has not been moved yet). In contrast, the subsequence $\langle A_8, A_{11} \rangle$ is not ground because action A_{11} requires the prior grasp of *block_b*.

In practice, since geometric backtracking is computationally expensive, only ground subsequences containing up to four actions are evaluated. It is possible to construct more (and longer) ground subsequences by combining the elements of $\mathcal{S}_{ground}$ with each other, but this is not done for the same reason. Culprit subsequences are detected by running the algorithm *GeometricBacktracking()* on each ground subsequence in $\mathcal{S}_{ground}$. Let us denote by $\langle A_{g1}, \ldots, A_{gm} \rangle$ a ground subsequence. In case of failure, the depth d at which the failure occurs is recorded. The subsequence $\langle A_{g1}, \ldots, A_{gd} \rangle$ is therefore a culprit subsequence.

During the evaluation of a subsequence, different types of collision checks are performed (see Section 10) in order to identify the objects that *always* collide. If the subsequence is unfeasible, a constraint is returned to the task planner, for example:

```
:- action1 (param11, param12, t1),
action2 (param21, param22, t2),
action3 (param31, param32, param33, t3),
...
t1<t2, t2<t3, ...,
not moved(colliding_object1, 1..t2-1); not
moved(colliding_object2, 1..t3-1); ...}.
```

The culprit subsequence is defined by a *partial order* on the actions. This representation is general, hence a large number of symbolic plans can be ruled out by this constraint. However, this type of constraint is weaker than the constraints returned by the "Spatial relations" layer, in which *no order* on the actions is imposed.

In this section, we have described the second layer of the geometric reasoner, which detects culprit subsequences of actions within a symbolic plan. It uses the direct geometric dependencies between actions in order to extract ground subsequences which can be tested independently. In the next section, we explain in more details the different types of collision checks that are performed in both the first and second layers.

10. Different types of collisions checks

Unavoidable collisions checks are performed in layers (1) and (2) of the geometric reasoner (see Figure 5). Unavoidable collisions are a common cause of infeasibility in CTAMP problems: when *all* the geometric instantiations ${}^k a_j$ of an action A_j in the plan result in a collision. We detect three types of unavoidable collisions, which return constraints of different strength:

- *strong unavoidable collisions*: collisions with fixed obstacles only (see Figure 1);
- *weak unavoidable collisions*: collisions with objects that have not yet been moved in previous time steps;
- collision with *unavoidable volumes*, i.e. regions of space which are necessarily occupied at some time step.

These three types of collision detection checks are performed both in the "Spatial relations" layer (1) and in the "Geometric dependencies chains" layer (2) (see Figure 5). In both cases, the same types of collision checks are performed, but the way the geometric configurations are sampled is different. In this section, we describe how this is done in the "Spatial relations" layer, i.e. using bounding boxes.

10.1. Strong unavoidable collisions

When this test is performed, a set of bounding boxes ($\mathcal{D}$) has been computed for each object/base/TCP at different time steps (see Section 8). Algorithm 2 is run for each of them, following the chronological order of the steps. First of all, Algorithm 2 deactivates collision detection for each object/base/TCP, except o_i (line 1). This means that only the collisions between o_i and the fixed obstacles are considered during the collision detection phase. Then, it retrieves from $\mathcal{D}$ the bounding box *bbox* associated to the object/base/TCP o_i (line 2). From this box, a discrete set of positions is uniformly sampled, as well as a set of orientations (line 4), and the pose of the object/base/TCP is updated accordingly (line 5). The orientation γ represents the rotation around a reference axis applied to a template transformation, as described in Section 7.1. The function *collide()* returns a list of colliding objects, if any (line 6). If one sampled pose is collisions-free (line 7), this means that collisions are not unavoidable and the function returns false (line 8). Otherwise, a list of objects causing the collisions is populated at line 10.

The process is demanding when an unavoidable collision exists because all the samples need to be tested. In such case, a logical constraint is automatically generated as explained earlier. The candidate culprit set is found by back-tracing from the node $o_i^{(j)}$ in the *spatial constraints graph* (see Section 8.3). For instance, the problem depicted in Figure 1 would result in the following constraint:

```
:- relation(p1, block_a, placement, z1, t)
relation(block_a, block_b, placement, z1, t)
relation(block_b, block_c, placement, z1, t)
relation(block_c, block_d, placement, z1, t)
relation(gripper, block_d, grasp, top, t)
not 1{moved(p1, 1..t-1)}
```

10.2. Weak unavoidable collisions

This test is similar to the *strong unavoidable collisions* test, but it also includes collisions with the movable objects

Algorithm 2: DetectStrongUnavoidableCollisions

Function **DetectStrongUnavoidableCollisions**($\mathcal{D}, o_i, j$)

input : $\mathcal{D}$: a domain
o_i: an object/base/TCP
j: a step in the symbolic plan

1 **deactivate** all movable objects but o_i
2 $bbox = \{[\underline{x_{o_i}}^{(j)}, \overline{x_{o_i}}^{(j)}], [\underline{y_{o_i}}^{(j)}, \overline{y_{o_i}}^{(j)}], [\underline{z_{o_i}}^{(j)}, \overline{z_{o_i}}^{(j)}]\} \subset \mathcal{D}$
3 $unavoidable = \emptyset$
4 **forall the** $(x, y, z) \in bbox, \gamma \in [0, 2\pi]$ **do**
5 **setPose**(o_i, x, y, z, γ)
6 $collisions =$ **collide**()
7 **if** $collisions = \emptyset$ **then**
8 **return** *false*
9 **else**
10 $unavoidable \leftarrow unavoidable \cup collisions$
11 **return** *unavoidable*

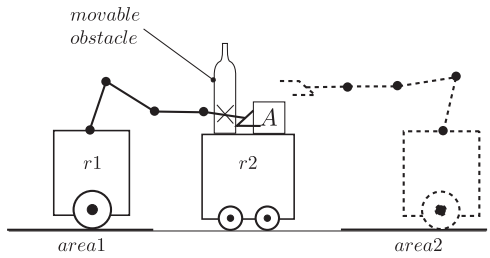


Fig. 24. Example of *weak unavoidable collision* between the TCP and the bottle.

that have not been moved yet (according to the ordering of actions). If an object consistently causes such collisions, a logical constraint is generated using the *spatial constraints graph* as previously explained. In addition, the constraint enforces that the colliding object(s) have to be moved during a previous time step. Consider for instance the situation illustrated in Figure 24. The bottle prevents $r1$ from grasping block A. In this case, a *weak unavoidable collision* is detected between the gripper and the bottle, and the following constraint is returned to the task planner:

```
:- relation(area1, r1, dock, t)                              (a)
relation(block_a, gripper, grasp, side, t)                   (b)
relation(r2, block_a, placement, z1, t)                      (c)
not 1{moved(area1, 1..t-1); moved(r2, 1..t-1)}               (d)
not moved(bottle, t2)                                        (e)
t2<t
```

Like in the case of a spatial inconsistency, the extremities of the culprit chain are used to determine that moving *area*1 or $r2$ may relax the linear constraints, hence possibly avoid the collision (d).

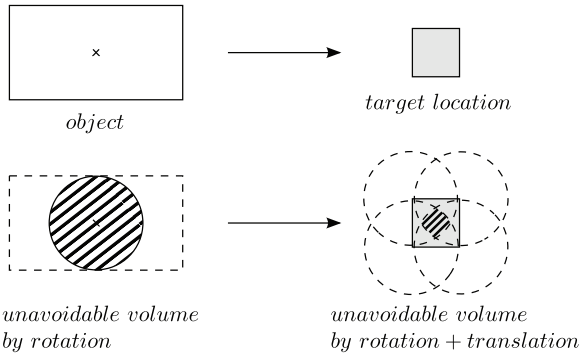


Fig. 25. 2D example of construction of an UV: the center of a rectangular object is to be placed on a square target location (top). First, the unavoidable volume by rotation is computed (disc at bottom left). Then the intersection of all possible translations of that disc on the target location (bottom right).

This "occlusion problem" is often mentioned in similar works. A common strategy is to use an ad hoc "occluding" predicate in order to artificially trigger the removal of the object. Here, occlusions are just a special case of *weak unavoidable collision*. The advantage is that the ASP solver is not tied to a predefined strategy, and can find other ways to solve the problem by using the logical constraint and the inference mechanisms of the solver. Removing the object is not the only possibility. Basically, any plan that makes one term of the constraint (a, b, c, d, e) false resolves the problem. For example, without changing the length of the plan, the solver could decide to choose a different docking area (a = *false*), or use a top grasp (b = *false*). If no solution is found this way, some actions can be added to the plan, e.g. moving block A to another location with another robot (c = *false*), moving the robot $r2$ (d = *false*), or picking the bottle up and placing it away (e = *false*).

10.3. Unavoidable volumes

An unavoidable volume (UV) represents a region of space which is necessarily occupied at a given time step. It is constructed by intersecting the volumes of an object in all the possible poses it can occupy as the result of an action. Figure 25 depicts how UVs can be geometrically constructed. In our implementation, these volumes are not computed in this way. We use a predefined set of cylinders with different radius and height, which are selected with ad hoc rules when needed.

According to the example in Figure 25, UVs can only be computed when the pose of a large object is constrained to be inside a small region. However, UVs can be computed in many situations, e.g. during stacking actions or grasping actions, because both the TCP and the grasped object are confined in a small region. Unavoidable volumes are computed after a symbolic plan is found. The bounding boxes are used to determine the size of the regions that each object occupies at each time step. A data structure $\mathcal{L}_{uv}$ is used to store which UVs are occurring at each time step, and which objects these UVs correspond to.

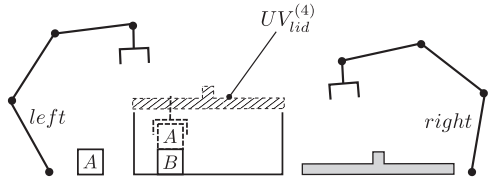


Fig. 26. Example of collision between the gripper and the unavoidable volume of the lid of the box.

As an example, consider the situation in Figure 26, from which the following sequence of actions is to be executed:

A_1: *pick (left, top, block_a, table)*
A_2: *stack (left, top, block_a, z1, block_b)*
A_3: *pick (right, top, lid, table)*
A_4: *stack (right, top, lid, z1, box)*

Regardless of how block A is stacked on block B, this sequence is doomed to fail at step 4, when the lid is stacked on top of the box. This problem can be detected as an unavoidable collision between the left TCP and the UV of the lid. Note that this problem cannot be detected either by the strong unavoidable collision check (because it only considers fixed obstacles), or by the weak unavoidable collision check (because the left TCP is moved before the lid is stacked, hence it is deactivated).

Collisions with UVs are detected together with weak unavoidable collisions, using specific rules for activating UVs at the correct time step. For instance in our example, the UV of the lid must be activated because the position of the left TCP at step 2 is not changed until the lid is stacked, at step 4. This can be determined using the list $\mathcal{L}_{uv}$ and the symbolic plan $\mathcal{P}$. $\mathcal{P}$ contains the information that the left TCP is not moved after step 2, and $\mathcal{L}_{uv}$ indicates that an UV exists for the lid at step 4.

The logical constraint is generated as for weak unavoidable collisions: a culprit set is found by back-tracing from the node $o_i^{(j)}$ ($left\_tcp^{(2)}$) in the spatial relation graph. But in addition, another culprit set is back-traced from the node corresponding to the object associated to the colliding UV ($lid^{(4)}$). The resulting constraint is built as a conjunction of the terms of both culprit sets. In our example, this process results in the following constraint:

```
:- relation(left_tcp, block_a, grasp, top, t)
relation(block_b, block_a, placement, z1, t)
relation(box, block_b, placement, z1, t)
not 1{moved(left_base,1..t-1); moved(box,1..t-1)}
relation(right_tcp, lid, grasp, top, t)
relation(box, lid, placement, z1, t)
not 1{moved(right_base,1..t-1);
moved(box,1..t-1)}
```

Note that UVs are also useful during geometric backtrack search, because in some problems, they prevent from

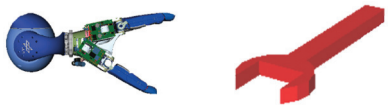


Fig. 27. Geometric bodies attached to the TCP for Justin (left) and Fabot (right).

placing an object in a pose which may compromise a future action.

The collision checks described in this section are done in all the layers of the geometric reasoner. For layers (1) and (2), in the special case where the tested body o_i is a mobile robot, only the base is used for collision detection. If o_i is a gripper, only the body attached to the TCP (see Figure 27) is considered, i.e. the links of the manipulator are ignored. None of these restrictions apply in layer (3).

11. Experimental evaluation

In this section, we evaluate our approach on three different scenarios, which present different difficult aspects of CTAMP. Scenario 1 is based on the introductory example. The difficulty lies in the peculiar geometric configuration which requires a culprit detection mechanism in order to avoid repeatedly encountering the same failure. This scenario is used for evaluating the scalability of our approach. Scenario 2 illustrates the generality of our approach, by addressing a problem which usually requires ad hoc algorithms to be solved. Scenario 3 shows an example of a problem where the causes of failure need to be generalized in order to solve the problem efficiently. Scenario 4 shows a limitation of our approach: geometric constraints cannot be fed back to the ASP solver, because symbolic aspects (allocation of tasks to robots) play a predominant role. Sample videos of the solutions to these problems can be found online[7].

The linear programs are solved with Gurobi[8], and collision detection with the library V-Collide (Hudson et al., 1997). Motion planning (for Justin's manipulators, Fabot's manipulator and Fabot's base) is done with bi-directional RRT (LaValle, 2006) implemented in Java. The ASP combined grounder-solver is clingo 4.2.1[8]. The rest is implemented in Java, and all the experiments are conducted on a MacBook Pro with Intel Core 2 duo i5, 2.4 GHz.

For a better understanding of the results, we present in Table 4 an order of magnitude of the time spent during the culprit detection checks, for a plan containing 30 actions. These checks are performed in sequence, and if a check fails, the latter ones are not performed. It may seem counter-intuitive to do the consistency check before the unavoidable collisions checks which are faster. The reason is that the consistency check is also used to compute the bounding boxes necessary for the subsequent checks. The resolutions used for discretizing actions are those presented in Table 1. Note that when the geometric backtracking check succeeds, a valid solution is found.

Table 4. Average times for the different checks performed by the geometric reasoner, assuming a plan with 30 actions.

Check (layer)	Success	Failure
Consistency (1)	3 s	7 s
Unavoidable collisions (1)	0.07 s	0.15 s
Unavoidable collisions (2)	2 s	3 s
Geometric backtracking (3)	45 s	1 min

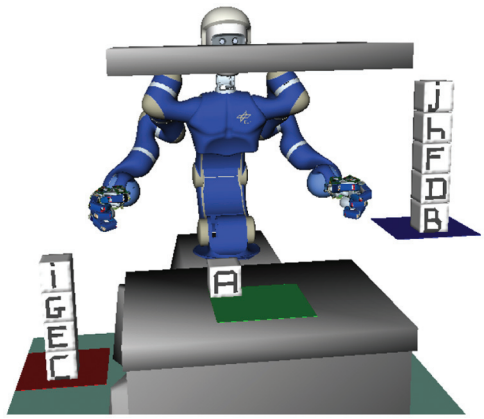


Fig. 28. The 3D version of the introductory example. The picture shows the experiment with 10 blocks. For the experiments with fewer blocks, the top-most blocks are removed from the piles.

The planning times given in the results do not include the path smoothing time. When a solution is found, the motion plans consist of raw RRT paths that guarantee feasibility, but which need to be smoothed. We do not include this time in the results because we consider that smoothing can be done during execution (except for the first action), with an average time of 2 seconds per action. We also use the term *iteration*, which refers to the fact that a failure has been detected and that a new plan is generated by the ASP solver (see Figure 4).

11.1. Scenario 1

This experiment is inspired by the introductory scenario as shown in Figure 28. The aim of the experiment is to show the scalability of our approach on a combinatorial task planning blocks-world problem with a non-trivial geometric problem. The task is to build a pile by stacking all the blocks in a randomly chosen order. The location of the pile is not specified, it may be on any of the modeled locations (the table or one of the three trays).

A fixed obstacle is located above the table at a distance such that it is impossible to build a pile with more than two blocks on the table, otherwise the TCP would collide with this obstacle. The right-most tray is not covered by the obstacle, but its higher position prevents the robot from stacking more than six blocks on it, because of kinematic constraints. These constraints are not logically encoded in the symbolic domain, the ASP solver will receive them

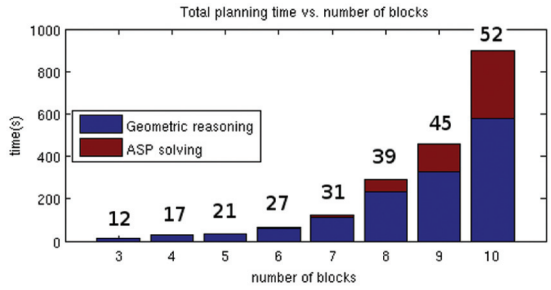


Fig. 29. Total average planning time with respect to the number of blocks. The numbers above the bars correspond to the average number of actions in the solution plans.

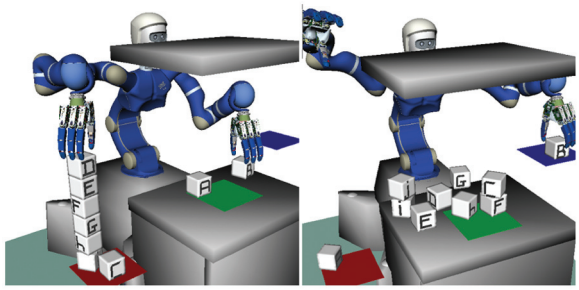


Fig. 30. Two pathological cases making the problem unfeasible or difficult. On the left, a pile is built which prevents grasping block C later on. On the right, blocks are placed on the table in an intermediate position, and due to clutteredness, some actions become difficult to perform.

from the geometric reasoner. If these two problems are not detected, the problem is intractable because the planner gets trapped in trying different plans which all lead to the same failures, i.e. collision with the obstacle or kinematic problem.

The problem was scaled up by increasing the number of blocks from three to ten. The initial configuration resembles what is depicted in Figure 28. Each problem was run with four different initial positions for the tray on the table, and with block A located on the tray. We did not change the position of the piles, because of reachability issues for blocks C and J. In total, 96 runs were conducted. For the experiments with nine and ten blocks, it happens that no solution is found because the planner ends up in a pathological situation where block C is occluded by the pile under construction (see Figure 30, left). As a result, 91.7% of the runs were solved.

The global results of the scalability experiment are presented in Figure 29. The trend is exponential, reaching up to 15 minutes average planning time for ten blocks (52 actions). Nevertheless, the planner is able to find a solution in reasonable time (less than 1 minute) for problems up to six or seven blocks, with plans containing around 30 actions. The time spent by the ASP solver on computing the symbolic plan(s) increases faster than the time spent on geometric reasoning. This is owing to the fact that the algorithms used in the geometric reasoner run either in polynomial time, or have a cutoff time.

The detailed results for the time spent on geometric reasoning are given in Figure 31. Detection of unavoidable collisions all together is fast, and increases linearly with the number of actions, while most of the time is spent on geometric backtracking. Figure 32 shows the average number of iterations needed to solve the problems, and the proportion of each type of failure encountered during culprit detection. For problems up to six blocks, all the failures are detected by some strong unavoidable collision checks in the "spatial relations" layer. It takes on average 4-5 iterations for the geometric reasoner to detected that it is impossible to build any pile on the table with more than two blocks, using the left or right TCPs. Then a valid plan is found,

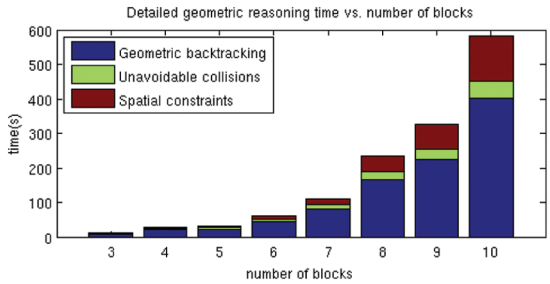


Fig. 31. Details for the average time spent on geometric reasoning. "Unavoidable collisions" refers to the three types of unavoidable collisions performed in layers (2) and (3) in Figure 5.

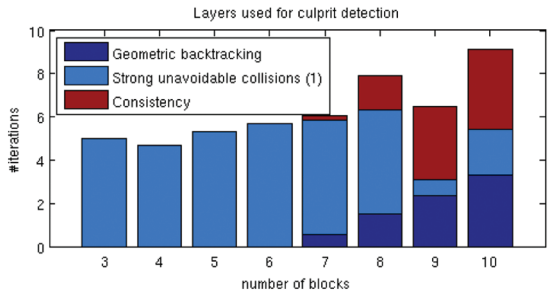


Fig. 32. This figure shows which checks in the geometric reasoner detected the failures during the iteration process (see Figure 4). (1) refers to the "Spatial relations" layer in Figure 5.

by building the pile on one of the two trays which are not occluded by the obstacle.

For problems with more than six blocks, the consistency check is triggered, because the planner finds solution plans in which the pile is built on the right-most tray. Then an inconsistency is detected because it is not possible to stack more than six blocks for kinematic reasons. For nine and ten blocks, the planner attempts to build the pile on the table, which is also impossible with respect to kinematic constraints. That is why we observe two extra failed consistency checks. Note that this is also not possible because of the obstacle, but recall that in the sequence of checks, the consistency check is performed *before* the

collision checks. These consistency checks lead the planner to choose the left-most tray as the only possible location to stack the blocks, which explains why less unavoidable collision checks are observed for nine and ten blocks.

Even when the main causes of failure have been detected, the planner needs to iterate over several solution plans, because it fails at the geometric level for reasons that are not detected by any specific check. This happens when the choice made at the geometric level for an action A_i causes occlusions or motion planning failures for another action A_j. If A_i and A_j are too far from each other in the plan, the geometric backtracking layer cannot reconsider A_i and returns failure. These situations typically occur because of peculiar configurations, or when the number of objects increases (see Figure 30). This explains why the number of geometric backtracking checks increases for more than six blocks (Figure 32). Consequently, the geometric backtracking time increases as well (see Figure 31), secondarily because the plans are longer, but mainly because the geometric backtracking check is performed multiple times. Another problem of geometric backtracking failures is that they are not informative for the ASP solver, because they just prevent it from returning one specific plan.

In summary, our approach shows a decent performance for plans up to 30 actions, although two factors appear as a limitation for larger problem instances. (i) Task planning is a difficult problem in general. (ii) At the geometric level, intricate situations occur more frequently for larger problem instances. Often, they cannot be solved by geometric backtracking, and lead to failures that do not guide the ASP solver. Therefore, the planner has to iterate over different plans until a plan without intricacies is found *by chance*. This is time-consuming because geometric backtrack search has to be done several times.

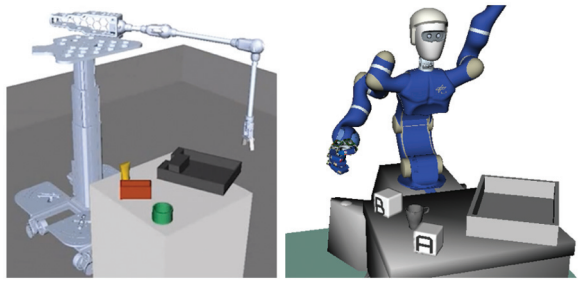


Fig. 33. Erdem et al. (2011) experiment on rearrangement planning of multiple objects (left), and our setup (right). The difference is that they use a mobile manipulator whereas in our setup Justin remains in a fixed position.

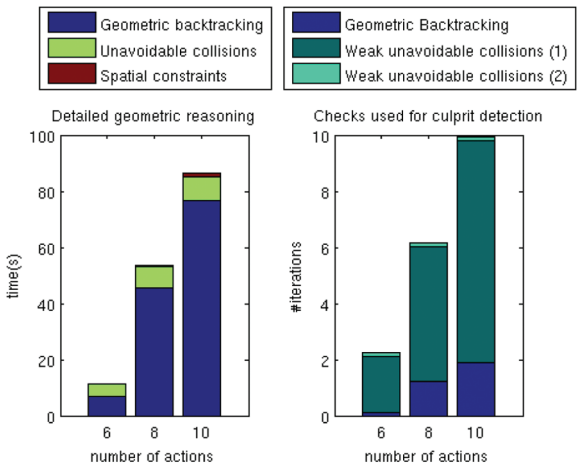


Fig. 34. Left: Details for the average time spent on geometric reasoning. Right: Checks in the geometric reasoner used to detect the failures during the iteration process (see Figure 4). (1) Refers to the "Spatial relations" layer and (2) refers to the "Geometric dependencies chains" layer in Figure 5.

11.2. Scenario 2

In this scenario, we replicate the experiment by Havur et al. (2014) on rearrangement planning of multiple objects. The aim of this experiment is to show the generality of our approach, by applying it to a problem which usually requires specific techniques to be solved. Rearrangement planning is a variation of navigation among movable obstacles (Stilman et al., 2007). These problems are known to be complex, therefore a common assumption applied for addressing them is to restrict the space of solutions to *monotone plans*, i.e. plans in which objects are moved at most once, which is an incomplete approach. We refer the reader to Havur et al. (2014) for the related work. They propose an approach with multiple stages, including the gridization of the continuous plane and hybrid planning combining ASP and geometric reasoning.

The scenario is illustrated in Figure 33. The goal is to swap the position of the cup with the position of the tray (meaning the center of the cup with the center of the tray). Blocks A and B have to be moved in order to free some space for the tray. The cup has to be moved to a temporary position, and placed at its final position after the tray has been moved. Note that it is also possible to move the tray first. The limited space on the table, plus the fact that Justin can only use one manipulator makes the task difficult. It is also forbidden to stack the objects on each other, otherwise the problem is less challenging. At the task level, the problem is simple if the objects that need to be moved are identified. At the geometric level however, the temporary poses of objects need to be carefully chosen because the space is limited.

We conducted 100 runs, with randomized initial positions for all objects (ensuring that they are reachable). 100% of the runs were solved. Depending on the initial configuration, the solution plans contain six, eight, or ten actions in respectively 29%, 46%, and 25% of the problem instances. When the problem is solved with six actions (which is the simplest case), the plan consists in moving the cup (respectively the tray) to an intermediate position, moving the tray to the position of the cup (respectively the tray), and then

moving the cup to the position of the tray (respectively the cup). Two or four extra actions are used when blocks A and/or B need to be moved. The results are summarized in Figure 34.

The time spent on ASP solving is negligible, hence the chart on the left in Figure 34 practically represents the total planning time: on average 12, 54, and 87 s for respectively six, eight, and ten actions. Although most of the time is spent on geometric backtracking, the chart on the right shows that the type of failure which is the most frequently detected is weak unavoidable collisions in the first layer. These checks have not much impact on the geometric reasoning time because they are fast (Table 4), and occur early in the sequence of checks (see Figure 5).

Let us illustrate the iteration process with an example where block B is close to the cup, and therefore has to be moved. We use two virtual locations (5×5 cm squares), symbolically labeled *target1* and *target2*, corresponding to the initial centers of respectively the tray and the cup. The goal is defined as:

```
:- not connected(target1, cup, t), horizon(t)
:- not connected(target2, tray, t), horizon(t)
```

Even though the initial state is randomized at the geometric level, it remains symbolically the same for each run. Therefore, the first plan returned by the ASP solver is always:

A_1: pick(right, border, z1, tray, table)
A_2: stack(right, border, target2, z1, tray)
A_3: pick(right, top, z1, cup, table)
A_4: stack(right, top, target1, z1, cup)

For action A_2, the geometric reasoner detects an unavoidable collision with block B. The collision is detected as "weak" since block B has not been moved yet. It is detected in the first layer, i.e. by sampling all the positions of the tray in a bounding box centered on *target2*. The following constraint is returned:

```
:-relation(tray, right, grasp, border, t)
relation(target2, tray, placement, z1, t)
not 1{moved(block_b, 1..t-1); moved(cup, 1..t-1)}
```

The second plan avoids this problem by starting to move the cup instead of the tray. But a similar problem occurs because of a collision with the tray. Therefore a weak unavoidable collision with the tray is detected and the following constraint is returned:

```
:-relation(cup, right, grasp, top, t)
relation(target1, cup, placement, z1, t)
not 1{moved(tray, 1..t-1)}
```

Now, there are no more solutions within plans of length 4. The solver increases the length to 5, for which there is no solution, and then searches for a plan of length 6. The first constraint enforces to move block B *or* the cup before placing the tray on *target2*. The third plan takes this constraint into account by moving the cup in action A_4:

A_1: pick(right, border, z1, tray, table)
A_2: place(right, border, table, z1, tray)
A_3: pick(right, top,z1, cup, table)
A_4: stack(right, top, target1, z1, cup)
A_5: pick(right, border, z1, tray, table)
A_6: stack(right, border, target2, z1, tray)

But a problem remains because of block B that still prevents the tray from being placed on *target2*. Again, a weak unavoidable collision is detected in the first layer and the following constraint is returned:

```
:-relation(tray, right, grasp, border, t)
relation(target2, tray, placement, z1, t)
not 1{moved(block_b, 1..t-1)}
```

The ASP solver returns the fourth plan which satisfies all the constraints, i.e. block B is moved before the tray is placed, and the tray is moved before the cup is placed:

A_1: pick(right, top, z1, block_b, table)
A_2: place(right, top, table, z1, block_b)
A_3: pick(right, border, z1, tray, table)
A_4: stack(right, border, target2, z1, tray)
A_5: pick(right, top, z1, cup, table)
A_6: stack(right, top, target1, z1, cup)

But the problem remains that the cup has to be removed before the tray is placed. Note that this problem was detected at the first iteration, but since several colliding objects were detected, the constraint contained a disjunction with respect to the objects to be moved (block B *or* cup). Through several iterations however, this disjunction is incrementally resolved. Finally, another weak unavoidable collision is detected and this constraint is returned:

```
:-relation(tray, right, grasp, border, t)
relation(target2, tray, placement, z1, t)
not 1{moved(cup, 1..t-1)}
```

With this constraint, the ASP solver cannot find a solution plan with six actions, but it finds one with eight actions, by inserting two actions at the beginning of the fourth plan that move the cup onto the table.

We showed through an example how our system solves a particular problem instance. Five iterations are necessary to detect which objects have to be moved, and in which order. This is achieved quickly because the geometric checks involved are fast, and the problem is simple at the symbolic level. The difficulty is at the geometric level, in the choice of the intermediate poses for the objects. They have to be chosen in a way that does not compromise any future action, which is not trivial because of the limited space together with the kinematic constraints of the manipulator. These geometric choices are facilitated by the *unavoidable volumes* (see Section 10.3), which can be computed because the tray is a large object to be placed on a small area. Nevertheless, the UV of the tray is a cylinder which is smaller than the actual tray (see Figure 25), hence the possibility remains that some objects are placed in positions occupied by the tray in the next steps. Consequently, a significant

effort remains to be spent on geometric backtrack search in order to find appropriate intermediate poses for all objects.

11.3. Scenario 3

This scenario demonstrates the capacity of our approach to generalize from the detected failures, i.e. after detecting an inconsistent configuration with two particular blocks, the planner is able to prune out the plans leading to the same failure with another combination of blocks. In the initial configuration, an open box is located on the right side of Justin, containing a pile with six blocks and a bottle, and the lid of the box is set on the table (see Figure 35). The goal is to have the six blocks inside the box and the box closed with the lid:

```
:- not on_location(block_a,cylbox,t), horizon(t).
:- not on_location(block_b,cylbox,t), horizon(t).
:- not on_location(block_c,cylbox,t), horizon(t).
:- not on_location(block_d,cylbox,t), horizon(t).
:- not on_location(block_e,cylbox,t), horizon(t).
:- not on_location(block_f,cylbox,t), horizon(t).
:- not connected(cylbox,cylbox_lid,t), horizon(t).
```

There is no predicate to represent that an object is "inside" the box, because we did not implement an action able to achieve this geometric effect. Instead, we use the predicate `on_location`, which represents the fact that an object is directly connected to the box, or connected to a pile of objects located in the box (see Section 6.2). We use the following rule:

```
on_location(Object, Location, t) :-
connected(Location, Object, t).
```

and a rule that ensures the transitivity:

```
on_location(Object2, Location, t) :-
on_location(Object1, Location, t),
connected(Object1, Object2, t).
```

From these rules, there is a large number of combinations that satisfy the goal: 1 pile with 6 blocks, 2 piles with 1 and 5 blocks, 2 and 4 blocks, or 3 and 3 blocks, etc., modulo all the possible orderings of the blocks. We added a symbolic constraint stating that no more than 3 piles can be made inside the box, otherwise the problem gets difficult, due to the size of the robot's hands with respect to the size of the box. Geometrically, it is not possible to close the box if the bottle is inside, nor if more than 2 blocks are stacked on each other. Therefore the only solution is to create 3 piles with 2 blocks each.

The mobile robot Fabot was used in this scenario. It can grasp all the objects with a side grasp, and unlike Justin, it can grasp an object which is not clear, i.e. it can grasp a whole pile of objects. With this additional robot, the problem can be solved in fewer steps, since some actions can be done in parallel. On the other hand, parallel actions sometimes lead to intricate situations where both robots interfere with each other (see Figure 37), which triggers costly GBT.

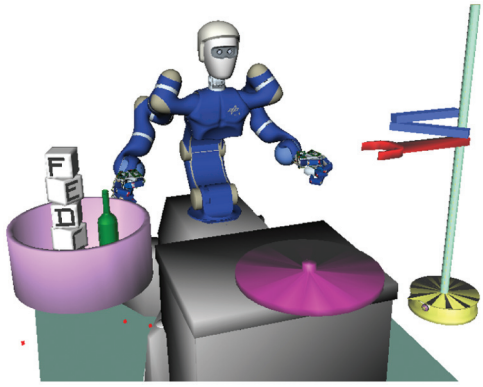


Fig. 35. A complex scenario combining several difficulties.

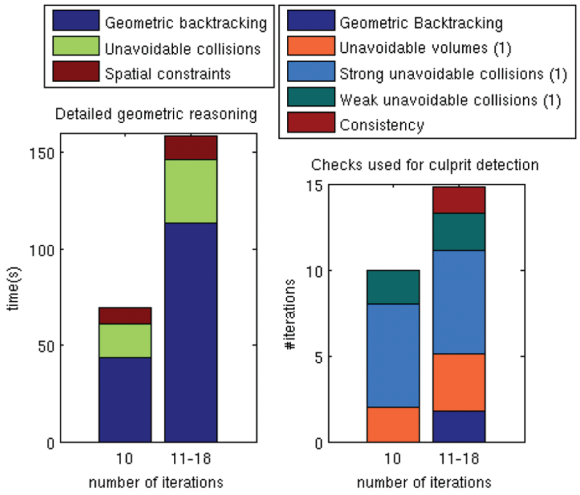


Fig. 36. Left: Details for the average time spent on geometric reasoning. Right: Checks in the geometric reasoner used to detect the failures during the iteration process (see Figure 4). (1) Refers to the "Spatial relations" layer in Figure 5.

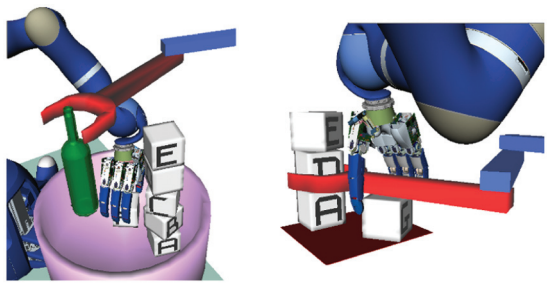


Fig. 37. Examples of interference between both robots during geometric backtracking. One of the robots cannot move its arm after a pick action because the other robot's arm is above (left), or less common: Fabot's TCP is trapped between Justin's hand and the grasped object (right).

The initial positions of the bottle and the pile were randomized, and 100 runs were conducted. A solution was found in 75% of the runs using a cutoff time of 5 minutes. The average time for finding a solution is 131 s (standard

deviation 59 s). The plans consist of 9 or 10 steps, containing 16 to 19 actions. The detailed results are presented in Figure 36. We aggregated the results into two groups. In the first group (28 runs out of 75), the problem was solved in 72 s on average (standard deviation 9.6 s). In the second group (47 runs out of 75), the problem was solved in 163 s on average (standard deviation 49 s). The ASP solving time is negligible: respectively 2.9 s and 4.5 s. The time spent on geometric reasoning is dominated by geometric backtracking, although not as strongly as in the previous scenarios.

Both groups share the same three types of checks. Unavoidable volumes are mainly used to detect that it is not possible to place a block (or stack 2 blocks) in the box, and leave the hand of Justin there while the box is closed. They also detect that it is not possible to have a pile with 3 blocks and the lid on the box. Strong unavoidable collisions detect that Fabot cannot pick/place a block in the box, or stack a block on 1 or 2 other blocks in the box, without the TCP colliding with the rim of the box (the TCP of Fabot is by construction always horizontal). Weak unavoidable collisions detect that it is not possible to place the lid on the box without moving block C and/or D. Whether D needs to be moved or not depends on the position of the pile: the handle of the lid collides with block D if the pile is located in the middle of the box. Once these culprits have been detected, the problem is basically solved, unless the initial position of the pile and the bottle are prone to interference, as illustrated in Figure 37 on the left, which explain the 25% of failures, because GBT does not complete.

In order to explain why the second group of runs takes twice the time to solve the problem, we need to analyze what happens during the first iterations of the algorithm. The first symbolic plan is always the same: putting the lid on the box. Consequently the geometric reasoner feeds back a constraint saying that this is not possible unless the bottle or blocks C and/or D are moved. At the second iteration, the ASP solver finds a plan in which the bottle is moved, by stacking the bottle on block F with the right arm and putting the lid on the box with Fabot. Depending on the initial configuration, an inconsistency may be detected, because the neck of the bottle may be too high to be reachable by the right arm. But in some cases, no inconsistency is detected: the TCP of Justin grasping the bottle is inside its bounding box, although the exact kinematics do not allow that movement. If it is detected, the problem is "easy", otherwise the system needs more iterations to figure out that stacking the bottle on block F is not a good option. During these iterations, extra consistency checks and geometric backtracking checks are performed as indicated in Figure 36, which increases the planning time.

This scenario contains another difficulty which does not appear in the presented results. There is a large number of possible arrangements of the blocks to achieve the goal at the symbolic level, from which only a small subset is geometrically feasible, i.e. 3 piles with 2 blocks each. This is

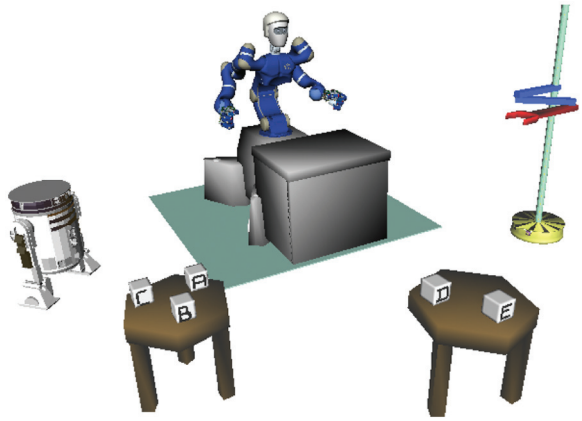


Fig. 38. A transportation scenario with two mobile robots.

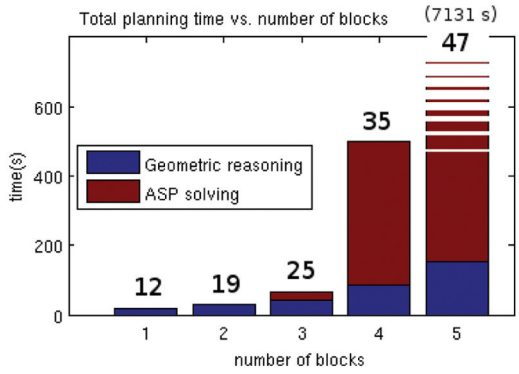


Fig. 39. Total average planning time with respect to the number of blocks. The numbers above the bars correspond to the average number of actions in the solution plans.

achieved thanks to the *generalized constraints with types* as explained in Section 6.5. Once the system has tried to create a pile in the box, e.g. F-E-D, and detected that this prevents from placing the lid on the box, it generates a constraint with object types, that prevents all possible combinations of 3 blocks in the box with the lid on the box. Moreover, the ASP solver can infer that it is impossible to build a pile with 4, 5 or 6 blocks, because this is logically impossible without first building a pile with 3 blocks. Therefore, it is only possible to build 3 piles with 2 blocks each (remember that we imposed a constraint of maximum 3 piles in the box for space reasons). This example shows that the ASP solver is not only used for planning action sequences, but also for reasoning about spatial relations between objects.

11.4. Scenario 4

In this scenario, the task is to clean a set of objects. In the initial state, objects are `dirty` and located at their respective tables (see Figure 38). In the goal state, they have to be `clean`, and back to their initial position. Cleaning an object is simulated by the action of placing that object on the table in front of Justin. Note that this problem is not solvable by

multimodal planning techniques, since symbolic reasoning is needed to achieve the goal.

A simple solution consists of Fabot moving back and forth between Justin's table and the smaller tables, transporting each object one after the other, which requires 6-7 actions for each object[9]. But since parallel actions are allowed, since r2d2 can carry several objects, and since Fabot can manipulate piles of objects, better solutions exist, in which robots wisely cooperate with each other.

The problem is geometrically simple, i.e. there are no narrow passages, and only weak geometric dependencies between actions. Therefore, the symbolic plan returned at the first iteration is always geometrically feasible, and the geometric reasoning time increases linearly with the number of actions (see Figure 39). At the symbolic level however, 4 robots can manipulate the objects, which leads to a large number of combinations. Remember that the ASP solver only increases its search horizon when it has proven that no valid plan exists for the current length. The advantage is that the plans are optimal (in terms of number of steps), but the drawback is that the computational cost increases exponentially with the horizon length, as shown in Figure 39. This type of problem would be more efficiently solved with state-space heuristic planning approaches (Dornhege et al., 2009; Srivastava et al., 2014), or the approach by Kaelbling and Lozano-Pérez (2011) for long-horizon problems.

It is interesting to compare these results with the results of scenario 1, in which plans with 52 actions (for the 10-blocks problem instance) are found in much less time. Setting aside the fact that both problems are different, we hypothesize the following explanation: in scenario 1, a solution is found after 9 iterations. During these iterations, the ASP solver finds several shorter plans, which are unfeasible, but which enable to compute 9 logical constraints that can be used for pruning the symbolic search space. This does not occur in the present scenario, i.e. the ASP solver has to solve a difficult problem from the start, and no logical constraint is used for pruning, since only one iteration is performed.

Note that this poor performance owes to the fact that an optimal plan is sought. The performance is dramatically improved if the ASP solver is asked for a solution within more steps. For instance, the 5-blocks problem instance is optimally solved with 26 steps in 2 hours. Solving the same problem within 50 steps takes only 28 s, although the plan contains useless actions (see the videos for a comparison of the optimal versus non-optimal plan).

11.5. Comparative experiments

11.5.1. Experiments without culprit detection In order to assess the impact of the culprit detection mechanisms, the same experiments have been done, with layers (1) and (2) deactivated. Layer (3) was maintained for geometric evaluation of symbolic plans, and forcing the ASP solver to return a different plan in case of failure. With this setup, the system becomes equivalent (in terms of type of information fed back to the ASP solver) to the approaches by Erdem et al. (2011) and Aker et al. (2012).

Scenario 1: No problem instance solved
Without culprit detection mechanisms, *none* of the problem instances could be solved, not even those instances with three blocks. Let us consider the simpler three-blocks case in detail. A solution for this problem requires six steps, for instance (some parameters have been omitted for concision):

1. pick(right, block_c) pick(left, block_a)
2. place(right, block_c, table) place(left, block_a, blue_tray)
3. pick(left, block_b)
4. stack(left, block_b, block_a)
5. pick(left, block_c)
6. stack(left, block_c, block_b)

However, without geometric feedback, a logically feasible plan consists in leaving *block_a* in place, and stacking *block_b* and *block_c* on top, which can be done in three steps. There exists four such plans of length 3, 99 plans of length 4, and 1193 plans of length 5. Since the ASP solver increases its horizon after exhausting all the solutions, the system therefore needs to geometrically evaluate 1296 plans before considering solutions of length 6, which is not possible within the allotted time (20 min).

Scenario 2: Time increased by one order of magnitude
For this scenario, the average solving time is increased by a factor 10, and not all instances could be solved. Problems requiring six and eight actions were all solved, on average in 135 and 606 s respectively. For problems requiring ten actions, only two problem instances were solved (out of 25), in 17 and 19 min, the rest being cut off. The problems requiring six actions could be solved because after trying the two possible 4-actions plans (swapping the cup and the tray), the ASP solver enumerates 6-actions plans, which consist of swapping the cup and the tray, plus moving an extra object. If by chance the extra object is the cup or the tray, a solution is found. Problems requiring eight actions present more combinations, but a plan moving the occluding object can be found by chance within 20 minutes. Few plans requiring ten actions were solved because reaching to that search horizon requires exhausting all the plans of length 6 and 8, which is rarely possible within the allotted time.

Scenario 3: No problem instance solved
The problem in scenario 3 is similar to the case of scenario 1. Logically speaking, the problem is feasible within two steps: since the blocks are `on_location` with respect to the box in the initial state, it is sufficient to pick the lid and place it on the box for achieving the goal. However, the actual feasible plans consist of at least nine steps. Considering the number of possible arrangements of blocks, and the combinations for deciding which robot manipulates which

blocks, the problem is clearly intractable without feedback from the geometric level.

Scenario 4: Similar results

Similar results were observed for scenario 4 because one iteration is sufficient to solve all problem instances. The time spent on geometric reasoning was not significantly reduced, because the time spent on geometric backtracking dominates the time spent on culprit detection.

11.5.2. Comparison with heuristic planning approaches

In this experiment, we evaluate how approaches based on state-space heuristic planning, which interleave symbolic and geometric reasoning, e.g. Dornhege et al. (2009); Srivastava et al. (2014), would perform on scenario 1. Since their setup cannot be exactly replicated, we instead emulated geometric reasoning in order to address a simpler problem, and therefore assess a lower bound on the results that may be obtained for solving the same problem with these types of approaches. The tested hypothesis is that for some problems, heuristic state-space planners may perform poorly because the geometric constraints of the problem are not captured by the heuristic function.

An equivalent pick-and-place domain was implemented in PDDL (Ghallab et al., 1998), and the forward state-space planner FF (Hoffmann and Nebel, 2001) was used. No detailed geometric representations were used, i.e. only one symbol for each location, one type of grasp, were used. The fact that the red tray (respestively blue tray) is only reachable by the right (respectively left) arm, was hardcoded in the domain. In order to emulate geometric reasoning, we implemented a function $\mathcal{F}_{stop}$ which symbolically evaluates if a pile of more than two blocks exists on the table or on the green tray (which always causes a collision between the gripper and the obstacle). $\mathcal{F}_{stop}$ is called for each visited symbolic state, and the state is not expanded if such a pile is detected. The running time of this function is negligible compared to the time for computing the heuristic, therefore the timings obtained are a lower bound on the time that would be taken if actual geometric reasoning was performed.

The native heuristic function of FF (h^{FF}) estimates the goal distance by building a planning graph and computing a plan ignoring the negative effects of actions. Instead, we used the heuristic function $f = g + h^{FF}$ (where g is the length of the partial plan at hand), otherwise the planner gets trapped in a heuristic plateau owing to the lack of geometric information, as explained next.

The results are presented in the second line of Table 5, and confirm the hypothesis. In absence of information about the obstacle above the table, the heuristic considers that bringing a block towards the green tray (or the table) requires two actions, whereas bringing a block from one side to another requires four actions. Hence, it estimates that stacking all blocks on the green tray requires less actions (say, n_{green} actions) than stacking all blocks on the red tray or the blue tray (say, n_{side} actions). Therefore, the planner does not reconsider moving *block_a* to the blue tray or the red tray as long as the heuristic "sees" solutions with less than n_{side} actions. Although the function $\mathcal{F}_{stop}$ prevents the planner from exploring all such unfeasible plans, there remains many states to visit from which a plan with less than n_{side} actions seems possible. For instance, the blocks from the initial piles can be unstacked in different orders or to different locations, and worse, $n_{side} - n_{green}$ irrelevant actions can be inserted without making the current state less promising than moving *block_a* to the red tray or the blue tray. The planner cannot escape this local minimum for problem instances with more than five blocks. If the native heuristic function of FF is used, the planner is trapped in a heuristic plateau for the same reasons, and never escapes from it. These results cannot be generalized to other types of problem, since scenario 1 was constructed with the aim of highlighting this issue. Nevertheless, it suggests that similar pitfalls may be encountered each time geometric constraints are not well captured by the heuristic function.

Table 5 also includes the results obtained with the ASP solver on the same problem instances (see Figure 29). A fair comparison is difficult: the ASP solver did not have the information computed by the FF heuristic, and the FF planner did not have the logical constraints computed by the culprit detection mechanisms. It is also inappropriate to compare optimal and non-optimal planning. The question is how would the FF planner perform if the information computed with culprit detection mechanisms was integrated in the computation of the heuristic. This raises the issue of the feasibility of such integration in the first place. Although the work by Garrett et al. (2014) presents a method for including information about occluding objects in the FF heuristic, extending this approach to the complex logical expressions returned by our geometric reasoner appears as a very challenging problem.

Scenario 1 is also an example in which information about occluding objects may not be useful, since the occluding object cannot be moved. This supports our initial claim that information about occluding objects or unfeasible paths is not sufficient to efficiently guide the task planner in all situations, and advocates for using more informative diagnosing methods, such as the proposed culprit detection mechanisms. Assuming that such mechanisms could be implemented in the heuristic function, one may expect performance issues since the heuristic function is called for each expanded node. This problem does not arise with our approach because symbolic and geometric reasoning are weakly coupled.

11.6. Discussion

In the first scenario, the scalability of the planner was evaluated. The experiments indicate an exponentially increasing planning time, which is the rule for planning problems. Nonetheless, the planning time remains below 1 min for

Table 5. Average task planning time (s) for scenario 1, ASP solving time versus FF planning time. Dashes represent cutoff time (30 min) or insufficient memory.

Number of blocks	3	4	5	6	7	8	9	10
ASP	0.2	0.36	1.04	3.9	12.9	56.1	134.7	319
FF	1.3	10.2	132	–	–	–	–	–

problems requiring 30 actions, which is a decent performance with respect to the complexity of the problem. Scenario 1 also demonstrates the ability of our approach to deal with combinatorial task planning problems, which is possible because of the weak coupling between the symbolic and geometric levels.

In scenario 2, the challenge is at the geometric level. After a few iterations, the objects that need to be moved are known by the task planner. The difficulty is then to carefully choose intermediate positions for these objects. Although the performance is not impressive, the experiments showed that all instances were solved. This demonstrates the ability of our approach to solve, without specific heuristics or strategies, a problem which usually requires ad hoc algorithms.

The third scenario, although it seems simple, hides several difficulties. The symbolic goal state can be achieved in many ways, although few of them are geometrically feasible. Solving the problem requires to detect geometric constraints on specific objects instances and generalize them to other objects of the same type.

The last scenario points out a limitation of our approach, i.e. when a difficult symbolic problem is to be solved in the first place. Then, it is not possible to iterate through simpler solutions, that offer opportunities for adding geometric constraints which can help solving the symbolic problem. This is the drawback of decoupling symbolic and geometric search spaces.

The experiments also point out another limitation of our approach, i.e. when the symbolic solution plan leads to intricate geometric configurations (see Figure 30 and 37), which cannot be resolved by geometric backtracking. Since the geometric reasoner cannot explicitly express the cause of failure into a logical constraint, the system needs to iterate over symbolic plans until the problem disappears by chance, which is very inefficient. A possible approach to address this issue is discussed in the next section.

Comparative experiments have shown that culprit detection mechanisms are the backbone of our approach, in particular for solving intricate problems. In scenario 2, the difficulty was mainly at the geometric level, while in scenario 4, the difficult part was the symbolic problem. On both scenarios, removing the culprit detection mechanisms did not affect the results much. On scenarios 1 and 3, where symbolic and geometric aspects are more intricate, using culprit detection mechanisms makes the difference.

Finally, a brief comparison with state-space heuristic planning approaches on a specific problem points out potential local minimum problems if the heuristic does not completely capture the geometric constraints of the problem.

12. Conclusion

We presented an approach for combining task and motion planning which included two culprit detection mechanisms in order to feed back rich information from the geometric level to the symbolic level. The first mechanism works on a relaxed version of the geometric problem, in which the poses of robots and objects are approximated by a set of bounding boxes represented by a network of linear constraints. We proposed techniques to detect spatial inconsistencies within this network in polynomial time. These bounding boxes are also used to detect different types of unavoidable collisions. The second mechanism relies on the construction of a graph of the geometric dependencies between actions. From this graph, shorter subsequences of actions can be extracted and independently evaluated.

The failures detected by these mechanisms efficiently guide the ASP solver because they do not simply report a "local" failure, but rather a context (expressed with spatial relations) or a culprit subsequence of partially ordered actions (with geometric dependencies chains). Since the task planner is based on logic programming, the detected failures are fed back by simply adding logical constraints to the problem. Therefore, the planning problem and the geometric constraints are homogeneously integrated in the same search space. ASP is not strictly required for the task planning part though: other logic-programming languages, or satisfiability-based planners could be used as well. One could also think of partial order planning as a suitable candidate, since the logical constraints returned by the geometric reasoner can be interpreted in terms of constraints on partially instantiated plans, and therefore they could be mapped to a small number of decision points in the plan-space. It is less obvious though how this could be integrated with a state-space planner.

The experiments have demonstrated the capacity of our system to solve various types of problems. Thanks to the weak coupling between symbolic and geometric search spaces, challenging task planning problems can be addressed, provided that geometric information can be used to cut the search space. Thanks to geometric backtracking, intricate geometric problems can be solved, although

this may be expensive in some cases. The proposed culprit detection mechanisms are effective: in most cases, a dozen iterations is sufficient to symbolically capture the main causes of geometric failures in a problem. This allows the ASP solver to prune out large parts of the search space, and quickly reach to a feasible plan. Last but not least, our system produces plans optimal in the number of steps, and supports for parallel actions, which few other approaches do. Nevertheless, we found limitations that need to be addressed in order to apply this approach to a wider range of problems.

(i) Path planning failures are not explained. The proposed culprit detection mechanisms only consider the configurations reached by a robot after completion of an action. As mentioned, this limitation is not of major concern for the proposed scenarios, but it would have a greater impact if the robots had to operate in heavily cluttered environments. In order to handle path planning failures, one needs to prove that a path does not exist, which remains a difficult and unsolved problem. The work by Hauser (2014) on the Minimum Constraint Removal Problem is promising in this respect. It could plug this gap in our approach, by feeding back to the ASP solver minimal explanations for path planing failures, indicating which objects need to be moved away.

(ii) Geometric backtrack search is difficult. In the experiments, some of the runs could not be solved because the symbolic plan led to an intricate configuration (see Figure 30), and since the culprit decision was made early, geometric backtracking is not able to backtrack up to that decision before the cutoff time is reached. In related work (Bidot et al., 2015), we proposed a heuristically guided geometric backtrack search algorithm to address this issue. It turned out to be difficult to find heuristics that can handle all situations, because geometric dependencies result from subtle interactions between the kinematics of the robot and the configuration space of obstacles.

In most such cases however, we observed that intricate situations only concern one or two actions, while the rest of the plan can be geometrically instantiated without difficulty, since the geometric reasoner has already detected the main possible causes of failure. In most cases, one could circumvent such problems by simply switching the order of two actions, or by inserting actions that move undesirable objects away, while preserving the rest of the plan. We will therefore focus our future efforts on integrating local symbolic plan repair strategies to the current approach.

Another possibility which could be explored is to relax some constraints during geometric backtracking, which is for now inflexible with respect to kinematic constraints and collisions, and to perform the needed adjustments at execution time by local reasoners (Scioni et al., 2015; Winkler et al., 2012). In other words, delegating some of the problems encountered offline during geometric backtracking to online execution processes, which would also provide more robust execution.

(iii) Task planning with ASP can be improved. The combined ASP grounder-solver clingo offers a general declarative framework for incorporating heuristics into the solving procedure (Gebser et al., 2013). In our future work we plan to utilize this framework to specify a heuristic for task planning problems in order to firstly reduce the solving time of the ASP solver and, secondly to guide the ASP solver such that the task plan found is adjusted to the geometric solver, e.g. avoiding unnecessary actions, spreading objects over available locations, etc. This would further reduce the overall solving time and would allow to approach more challenging problems.

Acknowledgments

We also thank Lars Karlsson and Alessandro Saffiotti for their insightful suggestions and help in improving this article.

Funding

This work was partially supported by EU FP7 project "Generalizing Robot Manipulation Tasks" (GeRT) [contract number 248273].

Notes

1. Geometrically, manipulators are moved away from their "natural" workspace, e.g. for Justin, away from the space in front of the torso. Bases are moved to a circular region one meter away from the current position.
2. The meaning of this symbol is explained in the next section.
3. This could be improved by sampling with in a ring or a disk, but a circular domain proved suffcient in all the experiments.
4. https://www.youtube.com/user/MRLabSweden
5. This is known from the symbolic state: since r2d2 and block B are connected, if r2d2 is moved, then block B is moved by ramification.
6. Because new variables are created each time an object is moved, no cycles can be created, but this is out of the scope of this article.
7. http://aass.oru.se/~fll/videos_ijrr.html
8. Gurobi Optimization, Inc (2013) Gurobi optimizer reference manual. http://www.gurobi.com
9. Potassco (2014) Potassco, the Potsdam answer set solving collection. http://potassco.sourceforge.net/
10. reach small_table1, pick block_a, reach table, clean, pick block_a, reach small_table1, place block_a, pick block_b,...

References

Aker E, Patoglu V and Erdem E (2012) Answer set programming for collaborative housekeeping robotics: Representation, reasoning, and execution. *Intelligent Service Robotics* 5(4): 275–291.

Bidot J, Karlsson L, Lagriffoul F et al. (2015) Geometric backtracking for combined task and motion planning in robotic systems. *Artificial Intelligence* (Online).

Boyd S and Vandenberghe L (2004) *Convex Optimization*. New York: Cambridge University Press.

Bylander T, Allemang D, Tanner MC et al. (1991) The computational complexity of abduction. *Artificial Intelligence* 49(1–3): 25–60.

Cambon S, Alami R and Gravot F (2009) A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research* 28(1): 104–126.

Chakravarti N (1994) Some results concerning post-infeasibility analysis. *European Journal of Operational Research* 73(1): 139–143.

Chinneck JW (1996) An effective polynomial-time heuristic for the minimum-cardinality IIS set-covering problem. *Annals of Mathematics and Artificial Intelligence* 17(1-2): 127–144.

Choi J and Amir E (2009) Combining planning and motion planning. In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. Piscataway: IEEE Press, pp. 238–244.

Cortés J and Siméon T (2004) *Sampling-based motion planning under kinematic loop-closure constraints*. Zeist: IEEE.

Dechter R and Frost D (2002) Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence* 136(2): 147–188.

Dornhege C, Eyerich P, Keller T et al. (2009) Semantic attachments for domain-independent planning systems. In: *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*. pp. 114–121.

Erdem E, Haspalamutgil K, Palaz C et al. (2011) Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. pp. 4575–4581.

Garrett CR, Lozano-Pérez T and Kaelbling LP (2014) Heuristic search for task and motion planning. In: *Proceedings of the Workshop on Planning and Robotics (PlanRob)*. pp. 148–156.

Gebser M, Kaufmann B, Otero R et al. (2013) Domain-specific heuristics in answer set programming. In: *Proceedings of AAAI*, Springer, pp. 350–356.

Gelfond M and Lifschitz V (1998) Action languages. *Electronic Transactions on AI* 3, Royal Swedish Academy of Science, pp. 193–210.

Ghallab M, Howe A, Knoblock C et al. (1998) PDDL—The planning domain definition language.

Guitton J and Farges JL (2009) Taking into account geometric constraints for task-oriented motion planning. In: *ICAPS Workshop on Bridging the Gap Between Task And Motion Planning, BTAMP'09*, pp. 26–33.

Hauser K (2014) The minimum constraint removal problem with three robotics applications. *International Journal of Robotics Research* 33(1): 5–17.

Hauser K and Latombe JC (2010) Multi-modal motion planning in non-expansive spaces. *International Journal of Robotic Research* 29(7): 897–915.

Hauser K, Ng-Thow-Hing V and González-Baños HH (2007) Multi-modal motion planning for a humanoid robot manipulation task. In: *Proceedings of the International Symposium on Robotic Research*, Springer, pp. 307–317.

Havur G, Ozbilgin G, Erdem E et al. (2014) Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach. In: *Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA 2014)*, IEEE, pp. 445–452.

Hoffmann J and Nebel B (2001) The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14(1): 253–302.

Hudson TC, Lin MC, Cohen J et al. (1997) V-collide: Accelerated collision detection for vrml. In: *Proceedings of VRML*. Monterey: ACM, pp. 119–125.

Kaelbling LP and Lozano-Pérez T (2011) Hierarchical task and motion planning in the now. In: *Proceedings of International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 1470–1477.

Karlsson L, Bidot J, Lagriffoul F et al. (2012) Combining task and path planning for a humanoid two-arm robotic system. In: *TAMPRA: ICAPS Workshop on Combining Task and Motion Planning for Real-World Applications*, pp. 13–20.

Kautz H and Selman B (1992) Planning as satisfiability. In: *IN ECAI-92*. Vienna: Wiley, pp. 359–363.

Kautz HA, McAllester D and Selman B (1996) Encoding plans in propositional logic. In: *Proceedings of KR96*, Morgan Kaufmann, pp. 374–384.

Kavraki L, Svestka P, Latombe J et al. (1996) Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In: *Proceedings of International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 566–580.

Kuipers L and Niederreiter H (1974) *Uniform distribution of sequences*. New York: Wiley.

Lagriffoul F, Dimitrov D, Bidot J et al. (2014) Efficiently combining task and motion planning using geometric constraints. *The International Journal of Robotics Research* 33(14): 1726–1747.

Lagriffoul F, Dimitrov D, Saffiotti A et al. (2012) Constraint propagation on interval bounds for dealing with geometric backtracking. In: *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 957–964.

LaValle S (2006) *Planning Algorithms*. Cambridge: Cambridge University Press.

Lifschitz V (2002) Answer set programming and plan generation. *Artificial Intelligence* 138: 2002.

Lifschitz V (2008) What is answer set programming. In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. Chicago: AAAI, pp. 1594–1597.

Lozano-Pérez T and Kaelbling LP (2014) A constraint-based method for solving sequential manipulation planning problems. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, pp. 3684–3691.

Luna R, Lahijanian M, Moll M et al. (2014) Fast stochastic motion planning with optimality guarantees using local policy reconfiguration. In: *IEEE International Conference on Robotics and Automation (ICRA)*, IEEE. To appear.

Nau D, Ghallab M and Traverso P (2004) *Automated Planning: Theory & Practice*. San Francisco: Morgan Kaufmann Publishers Inc.

Ott C, Eiberger O, Friedl W et al. (2006) A humanoid two-arm system for dexterous manipulation. In: *Proceedings of International Conference on Humanoid Robots (Humanoids)*, IEEE, pp. 276–283.

Parker M and Ryan J (1996) Finding the minimum weight IIS cover of an infeasible system of linear inequalities. *Annals of Mathematics and Artificial Intelligence* 17(1-2): 107–126.

Plaku E and Hager G (2010) Sampling-based motion planning with symbolic, geometric, and differential constraints. In: *Proceedings of International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 5002–5008.

Rossi F, van Beek P and Walsh T (2006) *Handbook of Constraint Programming*. New York: Elsevier.

Scioni E, Borghesan G, Bruyninckx H et al. (2015) Bridging the gap between discrete symbolic planning and optimization-based robot control. In: *2015 IEEE International Conference on Robotics and Automation*, IEEE. Accepted for publication.

Silva JaPM and Sakallah KA (1996) Grasp: A new search algorithm for satisfiability. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96. IEEE Computer Society, pp. 220–227.

Simeon T (2004) Manipulation planning with probabilistic roadmaps. *The International Journal of Robotics Research* 23(7-8): 729–746.

Simons P, Niemelä I and Soininen T (2002) Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2): 181–234.

Srivastava S, Fang E, Riano L et al. (2014) Combined task and motion planning through an extensible planner-independent interface layer. In: *IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 639–646.

Stallman RM and Sussman GJ (1977) Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9(2): 135–196.

Stilman M and Kuffner J (2008) Planning among movable obstacles with artificial constraints. In: *Algorithmic Foundation of Robotics VII*, *Springer Tracts in Advanced Robotics*, volume 47. Los Angeles: SAGE, pp. 119–135.

Stilman M, Schamburek JU, Kuffner J et al. (2007) Manipulation planning among movable obstacles. In: *Robotics and Automation, 2007 IEEE International Conference on*. pp. 3327–3332.

Toussaint M (2015) Logic-geometric programming: An optimization-based approach to combined task and motion planning. In: *International Joint Conferences on Artificial Intelligence*, AAAI.

Ullman J (1988) *Principles of Database and Knowledge-Base Systems*. New York: Computer Science Press.

Winkler J, Bartels G, Mösenlechner L et al. (2012) Knowledge Enabled High-Level Task Abstraction and Execution. *First Annual Conference on Advances in Cognitive Systems* 2(1): 131–148.

Appendices

A. Answer set programming

The reader may be interested in consulting a brief introduction to Answer Set Programming (Lifschitz, 2008) for completing the reading of this section.

A *rule r* is of the following form

$$H \leftarrow B_1, \ldots, B_m, \sim B_{m+1}, \ldots, \sim B_n$$

We use $headr = H$ and $body(r) = \{B_1, \ldots, B_m, \sim B_{m+1}, \ldots, \sim B_n\}$ to denote the *head* and the *body* of r, respectively, where "$\sim$" stands for default negation.[10] The head H is an atom a belonging to some alphabet $\mathcal{A}$, the falsum $\bot$, or a *count* constraint $L\,\{\ell_1, \ldots, \ell_k\}\,U$. In the latter, $\ell_i = a_i$ or $\ell_i = \sim a_i$ is a *literal* for $a_i \in \mathcal{A}$ and $1 \leq i \leq k$; L and U are integers providing a lower and an upper bound. Either or both of L and U can be omitted, in which case they are identified with the (trivial) bounds 0 and ∞, respectively. If $bodyr = \emptyset$, r is called a *fact*, and we skip "$\leftarrow$" when writing facts below. A rule r such that $headr = \bot$ is an *integrity constraint*, one with a *count* constraint in the head is a *choice rule* because it amounts to choosing some literals among $\ell_1, \ldots, \ell_k$ subject to the encompassing bounds. Each body component B_i is either an atom or a *count* constraint for $1 \leq i \leq n$.

We adhere to the definition of answer sets provided in Simons et al. (2002). A *count* constraint holds with respect to a set X of atoms if $L \leq |\{a \mid \ell_j = a, 1 \leq j \leq k, a \in X\} \cup \{\sim a \mid \ell_j = \sim a, 1 \leq j \leq k, a \notin X\}| \leq U$. A body literal B_i (or $\sim B_i$) holds with respect to X if B_i holds (or does not hold) with respect to X, where an atom a holds if $a \in X$. A rule r is satisfied with respect to X if some body literal of r does not hold with respect to X, H is *count* constraint holding with respect to X, or $H \in X$. Note that an integrity constraint is unsatisfied if all literals in its body hold with respect to X.

A ground logic program Π is a set of ground (i.e. variable-free) rules. A set X of atoms is a model of Π if each $r \in \Pi$ is satisfied with respect to X. An answer set of Π is a model X of Π such that every atom in X is derivable from Π. Roughly speaking, the latter means that, for each $a \in X$, Π contains a rule r with head $H = a$ or H being a *count* constraint comprising a such that all body literals of r hold with respect to X. We still note that programs are required to be *safe* (Ullman, 1988), that is, each variable must occur in a positive body literal. Predicates such as p and variables such as Y in "$p(\,Y)$" are written as lowercase or uppercase strings, respectively. Default negation $\sim$ is written as "`not`".

B. Definitions

Definition 1 *Geometric reachable set*

Let us consider a sequence of actions $\mathcal{P} = \langle A_1, \ldots, A_n \rangle$. We denote a geometric instantiation of $\mathcal{P}$ applied on the geometric state s by

$$\pi(s) = \langle {}^{i1}a_1, \ldots, {}^{i_n}a_n \rangle$$

where $i_1, \ldots, i_n$ represent the action indexes of each action, and we denote the resolution used for discretizing an action A_j by r_j (see Section 7.2). Let $k_\pi(s)$ (respectively $c_\pi(s)$) be the boolean function returning *true* if all actions in $\pi(s)$ are kinematically valid (respectively collision-free), and *false* otherwise. We define the Geometric Reachable Set from the geometric state s by an action $A_q \in \mathcal{P}$ as:

$$\begin{aligned}\mathcal{R}(A_q, s, \mathcal{P}) = \{a_q \in \pi(s) \mid\ & \pi(s) \text{ is a geometric instance of } \mathcal{P},\\ & a_q \text{ is a geometric instance of } A_q,\\ & k_\pi(s),\ c_\pi(s)\},\end{aligned}$$

and similarly, we define the Geometric *kinematically* Reachable Set from the geometric state s by an action $A_q \in \mathcal{P}$, which does not take collisions into account, as:

$$\begin{aligned}\mathcal{R}_{kin}(A_q, s, \mathcal{P}) = \{a_q \in \pi(s) \mid\ & \pi(s) \text{ is a geometric instance of } \mathcal{P},\\ & a_q \text{ is a geometric instance of } A_q,\\ & k_\pi(s)\}.\end{aligned}$$

Definition 2 *Geometric dependencies between two actions* Let $A_p, A_q \in \mathcal{P} = \langle A_1, \ldots, A_n \rangle$ be two symbolic actions such that $p < q$. Let s_0 be the initial geometric state on which $\mathcal{P}$ applies. Let $\pi(s_0)$ and $\pi'(s_0)$ be two geometric instantiations of the symbolic subsequence $\langle A_1, \ldots, A_p \rangle$ which only differ with respect to the geometric instance chosen for A_p

$$\pi(s_0) = \langle {}^{i1}a_1, {}^{i2}a_2, \ldots, {}^{i_p}a_p \rangle$$
$$\pi'(s_0) = \langle {}^{i1}a_1, {}^{i2}a_2, \ldots, {}^{i'_p}a_p \rangle,\ i_p \neq i_{p'}$$

and s_p, s'_p, the geometric states resulting from $\pi(s_0)$ and $\pi'(s_0)$, respectively. A_p is geometrically dependent on A_q iff $\exists \pi(s_0), \pi'(s_0)$ such that

$$\mathcal{R}(A_q, s_p, \langle A_{p+1}, \ldots, A_q \rangle) \neq \mathcal{R}(A_q, s'_p, \langle A_{p+1}, \ldots, A_q \rangle)$$

which we denote by $A_p \rightsquigarrow A_q$. In other words, A_p is geometrically dependent on A_q if changing the geometric instance of A_p can lead to a different geometric reachable set for A_q. Similarly, A_p is *directly* geometrically dependent on A_q iff $\exists \pi(s_0), \pi'(s_0)$ such that:

$$\mathcal{R}_{kin}(A_q, s_p, \langle A_{p+1}, \ldots, A_q \rangle) \neq \mathcal{R}_{kin}(A_q, s'_p, \langle A_{p+1}, \ldots, A_q \rangle),$$

which we denote by $A_p \overset{dir}{\rightsquigarrow} A_q$.

C. ASP encoding of the planning problem

```
%-------------------------------------------%
%        some geometric predicates          %   (1)
%-------------------------------------------%
% moved
moved(Component,t) :- do(Component,_,t).
moved(Movee,t) :- moved(Mover,t), holds(connected(Mover,Movee),t-1).

% reachable
reachable(ObjectLocation,Manipulator,t) :- holds(relation(ObjectLocation,Manipulator,dock),t).
reachable(ObjectLocation,Manipulator,t) :- reachable(ObjectLocation,Base,t), architecture_child(Base,Manipulator).
reachable(Object,Manipulator,t) :- reachable(Location,Manipulator,t), on_location(Object,Location,t).

% on_location
on_location(Object,Location,t) :- holds(connected(Location,Object),t), object(Object), location(Location).
on_location(Object,Component,t) :- holds(connected(Component,Object),t), object(Object), component(Component).
on_location(OtherObject,Location,t) :- on_location(Object,Location,t) , holds(connected(Object,OtherObject),t).

%-------------------------------------------%
%        persistence of connected           %   (2)
%-------------------------------------------%
holds(Attribute,t) :- holds(Attribute,t-1), not stop(Attribute,t).

% during pick action
holds(connected(Component,Object),t) :- do(Component,action(pick,Object,_),t).
stop(connected(Location,Object),t) :- do(Component,action(pick,Object,_),t),holds(connected(Location,Object),t-1).

% during place action
holds(connected(Location,Object),t) :- do(Component,action(place,_,_,Location),t),
                                       holds(connected(Component,Object),t-1).
stop(connected(Component,Object),t) :- do(Component,action(place,_,_,_),t),
                                        holds(connected(Component,Object),t-1).

%-------------------------------------------%
%                pick action                %   (3)
%-------------------------------------------%
% defines when pick is possible in general
action(pick,Object,GraspType) :- graspable(Object,GraspType).

% defines preconditions for pick at time t
:- do(Component,Action,t), not possible(Component,Action,t).

possible(Component,action(pick,Object,GraspType),t) :-
   possible(action(pick,Object,GraspType)),
   reachable(Object,Component,t-1),
   on_location(Object,Location,t-1),
   holds(oriented(Object,Orientation),t-1),
   allow_manipulate(Orientation,GraspType,Component),
   not moved(Location,t),
   not holds(connected(Component,_),t-1).

% consequences of the pick action
holds(relation(Component,Object,grasp,GraspType),t) :- do(Component,action(pick,Object,GraspType),t).
holds(connected(Component,Object),t) :- do(Component,action(pick,Object,_),t).
manipulated(Component,Object,t) :- do(Component,action(pick,Object,_),t).
stop(connected(Location,Object),t) :- do(Component,action(pick,Object,_),t),
                                      holds(connected(Location,Object),t-1).

% cannot pick a pile of objects if not skilled   (3.a)
:- do(Component,action(pick,Object,_),t), holds(connected(Object,_),t-1), not skilled(Component).

% cannot simultaneously grasp the same object by two robots   (3.b)
:- do(Component,action(pick,Object,_),t), holds(relation(OtherComponent,Object,grasp,_),t),
   Component != OtherComponent.

% cannot grasp by top an object which is not clear   (3.c)
:- do(_,action(pick,Object,top),t), holds(relation(Object,_,placement,_),t).
```

```
%-------------------------------------------%
%       relations persistence rules         %   (4)
%-------------------------------------------%
% breaking the placement relation
stop(relation(ObjectLocation,Object,placement,Orientation),t) :-
   holds(relation(ObjectLocation,Object,placement,Orientation),t-1),
   1{moved(Object,t) ; moved(ObjectLocation,t)},
   not holds(connected(ObjectLocation,Object),t-1).

% breaking the grasp relation
stop(relation(Component,Object,grasp,GraspType),t) :-
   holds(relation(Component,Object,grasp,GraspType),t-1),
   1{moved(Object,t) ; moved(Component,t)},
   not holds(connected(Component,Object),t-1).

% breaking the dock relation
stop(relation(Object,Base,dock),t) :-
   holds(relation(Object,Base,dock),t-1), 1{moved(Object,t) ; moved(Base,t)}.

%-------------------------------------------%
%                 Justin                    %   (5)
%-------------------------------------------%
component(left_base).
component(right_base).
component(left_arm).
component(right_arm).

% if Justin is mobile, replace left_base and right_base by torso,
% and connect torso to a mobile base.
base(left_base).
base(right_base).

able(left_arm,manipulating).
able(right_arm,manipulating).

architecture_child(left_base,left_arm).
architecture_child(right_base,right_arm).

% connects abilities to actions
:- do(Component,action(Action,_,_),t), requirement(Action,Requirement), not able(Component,Requirement).

requirement(dock,moving).
requirement(dock,loadable). % for r2d2
requirement(reach,moving).
requirement(pick,manipulating).
requirement(place,manipulating).
requirement(stack,manipulating).

%-------------------------------------------%
%      Manipulation rules for Justin        %   (6)
%-------------------------------------------%
orientation(z1).
orientation(z2).

% not allowed to pick/place an object in upright position by the bottom
allow_manipulate(z1,top,left_arm).
allow_manipulate(z1,top,right_arm).

% not allowed to pick/place an object in upside-down position by the top
allow_manipulate(z2,bottom,left_arm).
allow_manipulate(z2,bottom,right_arm).

% block_a cannot be grasped by the side
graspable(block_a,top).
graspable(block_a,bottom).
```