Solving Task Space Problems with Multi-Step Planning

Max Pflueger and Gaurav S. Sukhatme

***Abstract*—Modern robotics research has developed a mature family of planners for solving robot motion planning problems, but task space problems (where we want to reason about objects) remain difficult. Other approaches for solving what are often referred to as combined task and motion planning problems rely on bringing logical problem structure into the design of the high level planner. While programming assumptions around logical problem structure cannot be avoided, we propose the multi-step planning approach which maintains high level generality while pushing these logical assumptions to a lower level of the program. Through experiments on a real robot and in simulation we demonstrate the ability of this architecture to solve real (though small) problems. We also demonstrate how carefully chosen heuristics can be key to making this approach faster.**

***Note to Practitioners:* This paper proposes techniques for using multi-step planning to solve problems defined in the way we would like objects to move in the space, rather than how we would like the robot to move. Successful applications of this technique will depend of the ability of the practitioner to define useful families of planning spaces, and provide algorithms that can plan and sample states in those families. Additionally, in order to get good performance, particularly on larger problems, the practitioner should provide an informative heuristic that matches the cost function used in the planning process. Our results suggest that with good heuristics, multi-step planning could be a practical technique for problems with relatively small search complexity, and suggest the potential for developing stronger heuristics to target larger problems.**

***Index Terms:* Planning, Manipulation, Heuristic Search, Combined Task and Motion Planning.**

I. Introduction

Robotic systems capable of dexterous manipulation are becoming increasingly common, driven by precipitous drops in the cost of sophisticated sensing technologies as well as ever more powerful computers. These new sensors allow robots to be used in unstructured settings where objects may arrive in unknown configurations and obstacles may exist. In order to use robots in these environments we need planning technology capable of dealing with these unknowns to achieve a manipulation task.

Techniques for robot motion planning have come a long way in the last decade or so, with powerful planning algorithms now available essentially 'off the shelf' through frameworks such as OMPL [1] and MoveIt [2]. These planners can effectively create plans to move robot arms to almost any position through most practical environments.

The authors are with the Computer Science Department in the Viterbi School of Engineering at the University of Southern California in Los Angeles. {pflueger, gaurav}@usc.edu. This work was supported in part by NSF grant IIS-1017134 and by DARPA grant W91CRBG-10-C-0136.

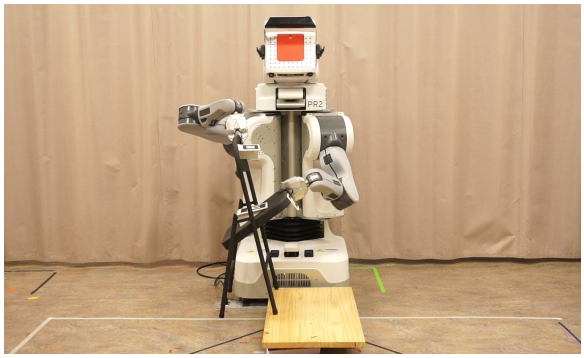


Fig. 1: PR2 robot in the process of folding a chair.

When thinking about practical robotic manipulation we are almost always dealing with 6 or 7 degree-of-freedom robot arms, mounted on a mobile base. These arms have complex kinematics that make analytic solutions to collision checking problems near impossible. Consequently, sampling-based planning algorithms are used to develop collision free trajectories. These planning algorithms have advanced to a point where 'off the shelf' algorithms exist that can produce practical, collision free, motion plans for many arms in diverse circumstances [1]. However, they leave something to be desired when we think about the objectives we actually care about in robotic manipulation problems.

When we talk about robotic manipulation, we are interested the ability of the robot to change the state of an object in its environment, and not so much in the state of the robot itself. From the perspective of a motion planner, robots tend to be well behaved, with fully actuated degrees of freedom, where as objects themselves are more difficult to plan for. Many effective techniques exist to develop plans for the kinematic motion of robots, but they tend to fall short of a complete solution when we try to develop plans for the object rather than the robot.

Traditional robotic motion planning problems focus on the state of the robot. A more useful (though more difficult) problem formulation would specify goal conditions in terms of the state of the environment. To wit, in many useful problems, the state of the robot is not intrinsically important. If we wish to move a bowl on a table by using a robot, we care that the bowl gets to where we want it. Usually (as long as the robot doesn't get in our way or run into anything) we don't care

where the robot goes to achieve the goal. Combined task and motion planning refers to planning through a combination of the task space (the state of the environment) and the motion of the robot.

Although it may seem like a simple matter of adding more degrees of freedom, combined task and motion planning problems are significantly different from the problems usually encountered in robotic motion planning. Holonomic robots are those that are capable of instantaneously moving in any free direction (such as a ground robot with omni-wheels, or a robot arm in joint space). Planning is easier for holonomic robots because it is easy to develop a local algorithm to move the robot from one waypoint to another. Planning for non-holonomic robots is more difficult because of restrictions in how waypoints can be connected (requiring a more sophisticated local planner).

We believe that a significant difficulty of planning in this domain comes from the non-holonomic nature of the problem, that is, it is not possible for the system to instantaneously move in any direction in its configuration space. In general, motion planners exist that can deal with non-holonomic problems (RRTs [3] and variants are a good example), however these sorts of planners were generally designed for a particular type of non-holonomic problem where even if it is not possible to move instantaneously in any direction, it is possible to get to any goal state that is nearby in a short amount of time. For example a differential drive robot can use a forward and back motion to slide to the side a small distance in a relatively small amount of time.

Put another way, these spaces have an easily computed or well defined sense of locality. That turns out not to be the case for manipulation problems,as the state of the objects in the environment cannot be changed until the robot is in a proper position to interact with them. For example, if our planner desired a small motion of an object on a table, the actual time to achieve that may vary widely (and be unbounded) depending on how easily the robot can reach the object.

Some researchers in the motion planning community have looked at the problem of planning for the state of objects along with the robot that moves them, sometimes referring to this problem as combined task and motion planning, or integrated task and motion planning. Some approaches have involved using logic based solvers [4] or pre-designed plan outlines supplied by an expert [5]. So far the problem is still very difficult, and without good generalized solutions.

The solution we have proposed for solving these combined task and motion planning problems is called multi-step planning. Multi-step planning has been applied previously by other researchers in some other domains, and we believe it is a strong technique for robotic manipulation problems as well. The principle behind multi-step planning is the combination of our known, well studied robot motion planners (also referred to in this work as continuous planners, as they operate on a continuous space) with a discrete planning algorithm able to choose the sequence of continuous plans that must be executed. It relies on the ability of the programmer to do two important things in the space: first, to find discrete points for breaking up plans, and second, to provide continuous planners that can generate plans at all of those discrete points. Our system is agnostic of how the programmer chooses to meet these constraints, but any modern kinematic motion planner should work well, as well as more specialized planners for specific tasks. (We discuss this process in more detail in section IV: Algorithm.)

We start by demonstrating how to use multi-step planning to solve the non-trivial problem of manipulating a folding chair (work previously published in [6]). The folding chair problem demonstrates the power of this technique on an interesting planning problem with a relatively small search space. This problem is in some ways easier that other multi-step problems (only one degree of freedom outside the robot), but also demonstrates the power of this approach by showing how it can integrate a specialized planning algorithm in some steps of the plan where it is necessary (two arm manipulation of the articulated object), while also using more simple planning algorithms when the robot is differently constrained (standard sampling based planners for free arms).

We will then extend the technique by showing how to improve the discrete search component of the process to provide a significant speed up in planning time as well as shrinking the size of the explored search tree. We evaluate the magnitude of this speed up by running our planner on four simulated configurations of our chair folding problem, and collecting a variety of metrics about the planning process. Additionally, with the increased performance of this new discrete planner, we are able to demonstrate how this technique can cross applications to a simple tabletop manipulation problem. Although the tabletop problem is more common and appears simple, it actually exhibits significantly higher combinatorial complexity than the chair folding problem.

II. Related Work

Most robot motion planning can be classified as either dynamic (or policy based) planning, or kinematic planning. Our work is entirely based on the kinematic planning model in which the result of a planner is a viable trajectory through state space that the robot is expected to follow. Kinematic planners are not well suited for applications where the robot must deal with large uncertainty in state transitions, and so for that reason we have to allow the robot to move slowly so it can stay close to its trajectory.

Our work relies on the A* search algorithm for the discrete search component. A* was originally proposed by Hart and Nilsson [14] and has become a standard algorithm since then. Other heuristic search algorithms exist such as D* [15] and its variants [16] [17], and multi-heuristic A* [18]. Examining the properties of these algorithms applied to multi-step planning could be a promising future area of research.

A. Continuous Space Planners

Motion planning problems for modern, high degree-of-freedom robots, such as arms and mobile manipulators, are most frequently solved using one of a group of sampling based motion planning algorithms. This includes techniques such as probabilistic road maps [7] and RRTs [3].

RRTs in particular are well suited to the well behaved non-holonomic problems often encountered in robot navigation (discussed earlier), and many variants have been proposed, including RRT* [8], RRT-connect [9] and others. However, RRTs rely on a notion of locality that works well in these well behaved spaces (specifically, a meaningful distance metric for finding nearest nodes), but tends to break down in manipulation problems such as the type we address with multi-step planning and other combined task and motion planners.

While RRT and its variants fundamentally look like search processes, there are also approaches to motion planning based on the idea of trajectory optimization, where we start with some hypothesis trajectory and deform it according to some cost function with techniques such as CHOMP [19] or STOMP [20]. Cost functions can be built to incorporate the need for collision avoidance, as well as other cost metrics.

B. Combined Task and Motion Planning

Other researchers have used a variety of techniques to approach the combined task and motion planning problem. Nedunuri et. al. [5] developed an approach that relied on plan outlines supplied by an expert. Barry et. al. [21] have demonstrated a technique that constructs plans in a hierarchical manner, starting with object motions, then moving up to robot motions to satisfy the object trajectory. Srivastava et. al. [4] developed a planner based on a new interface layer between known motion planning algorithms and known PDDL based task planning algorithms.

In contrast to some of the above approaches, our work does not use specialized task planners, but instead divides the world state into qualitative classes based on the nature of the robot's interaction with the environment, and asks the programmer to provide motion planners that are valid in each class of world states. We believe our approach is more generalized than others in the field, making very few assumptions about the nature of the world, and instead relying on the implementer to build the necessary constraints into the provided planners. In a practical sense, this requirement would exist with any attempt at combined task and motion planning, but we think it makes sense to push it to the lowest level possible, rather than building structural constraints into the high level algorithm. This approach may come with a computational cost (at least until we can develop improved heuristics), however we are bullish on future improvements in computational power. As an example of its versatility, our formulation allows us to provide planners for specialized situations such as two hand manipulations of articulated objects.

Multi-step planning has been used before in some other specialized applications, Bretl et. al. [12] used it to plan a path for a wall climbing robot. In their experiment a robot climbs a wall using rock climbing holds similar to what would be seen in a rock climbing gym. In this context the discrete states represented the set of holds the robot is currently using to attach to the wall, and state transitions have to account for whether the robot can reach a new hold without falling off the wall. Hauser et. al. [13] demonstrated an application of machine learning to improve the efficiency of the multi-step planner for the same robot.

The formulation developed by Siméon et. al. [22] is very similar to our own in that they also look at the space as a high dimensional configuration space including both objects and the robot. They use families of 'transit' and 'transfer' paths to break up the space into subdimensional manifolds where the robot either holds an object or it does not. These 'transit' and 'transfer' paths then form a graph that can be used with a probabilistic roadmap (PRM) planner. In our own work we do not distinguish algorithmically between paths where the robot gripper is occupied or not, instead allowing the programmer to provide whatever planners (and thus, subdimensional manifolds) they choose. But, more importantly, we use a heuristic search process, rather than a PRM, which has different performance characteristics and guarantees.

C. Object Perception and Manipulation

Outside of simulated or highly structured environments, any manipulation planning system relies on some form of sensing and perception to detect the presence and position of the objects with which to interact. Our experimental system relies on fiducial markers and knowledge of the structure of the object. At a practical level it can be difficult to supply structured information about articulated objects when a CAD model is not handy. Sturm et. al. studied ways to model and learn the configuration of articulated objects [10]. Katz et. al. have done similar work using a robot to interact with the scene and detect articulated structure [11].

III. Problem Formulation

The full planning space of the problem combines the full pose of the robot along with the poses of all objects in the scene. Planning in that space in a raw form would normally be very difficult due to the high dimensionality of the space, and the need to model state transitions that take into account the physical interaction of objects. We simplify this space by assuming that the scene (non-actuated degrees of freedom) is fixed, and only moved under special circumstances. In this case, objects in the scene only move when the robot is holding on to them.

A limitation of this approach is that it does not directly model the physics of objects in the world. For example, if the robot deposits a small tabletop object into a location in free space, the object tends to fall to the ground (Robonaut [23] will be allowed to disagree). This can be fixed with a constraint that objects may only be released in stable positions, but it requires the programmer to codify these stable positions.

A. A Toy Problem

The need for this solution approach is easy to demonstrate by a simple, 2D toy problem. Consider a robot and object in a 1 dimensional world, such as Fig. 2. Suppose we wish to achieve a final state as in Fig. 3. To visualize a plan we can see the whole state space represented in 2D in Fig. 4.

We observe that since the cart is not actuated, only certain forms of motion in the state space are possible. A plan must either move along the diagonal line, as when the robot is

Fig. 2: Initial conditions for a 1D toy world, with a robot on the left, and cart on the right.

Fig. 3: Final conditions for the 1D toy world.

attached to the cart, or it must move vertically under the diagonal line, as when the robot is unattached. We note that most easy notions of locality or distance metrics break down in this problem as two states may be very close, if the cart is near its goal position, but if the robot is not nearby, the actual manipulation distance may be much larger.

For this toy problem it is easy to see how to find a solution: travel up to the diagonal line (grab the cart), traverse the diagonal line until vertically aligned with the goal state (move the cart to its goal position), then move vertically down to the goal state (release the cart and move the robot to its own goal position). Of course, this is an ad hoc solution that would only work for this particular problem, below we describe the multi-step planning algorithm for finding solutions in these sorts of spaces.

B. General Formulation

We formulate this problem as one of kinematic planning in a high dimensional configuration space that includes all dimensions of both the robot and any objects. Instead of planning directly through the entire space, we define subdimensional manifolds in the space that we will refer to as *stances*. The key properties of a stance are that (1) we have a planner that can plan on the manifold, and (2) we have a way to sample states that exist at the intersection of different stances. Sets of disjoint stances can be combined to form a planning space of higher dimensionality. A planning space would be the set of stances that all use the same planner. This might be visualized as a stack of sheets.

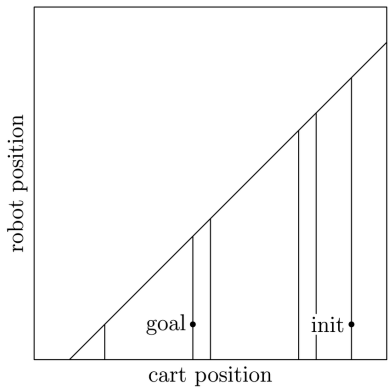


Fig. 4: The full 2D state space for the 1D toy world. Note that motion is only possible along the diagonal line (robot holding the cart) or vertically under the diagonal line (robot unattached to the cart).

An example of this could be seen in a tabletop problem with objects A and B. The robot is holding object A and object B is on the table. The robot is in a planning space that includes all cases of holding and moving object A, and the specific stance in that space is parameterized by the position of object B on the table, and which grasp point the robot is using to hold object A. The robot could then sample intersections with the planning space of holding no objects by sampling places to set down object A.

The search problem becomes one of finding the sequence of trajectories along stance manifolds and stance transition points that minimize the distance metric of our choice.

IV. Algorithm

Multi-step planning is, fundamentally, a layering of a discrete search algorithm on top of specialized continuous planners. We consider the full state of our world to include all degrees of freedom of our robot, as well as all the degrees of freedom of any objects we will interact with. Some dimensions of our world can be directly controlled (i.e. robot joints), where as others cannot (e.g. the position of an object).

The core of our algorithm is the getPlan function outlined in Algorithm 1. We use the A* discrete search algorithm, with a dynamically generated graph. A* is a well known and studied algorithm, but when using it on the dynamically generated graph that is essential to multi-step planning, the behavior of a couple of important functions becomes key to its performance. We discuss those functions below.

Algorithm 1 depends on two implementation specific functions, `reachable` and `heuristic`, as well as the `adjacent` function which is domain specific. The `heuristic` function can be any admissible and consistent heuristic for our application that also matches the cost calculation for g values. In this paper we will use the number of steps as a distance metric with the matching `minSteps` heuristic outlined in Algorithm 4.

A. Reachable

The reachable function is designed to tell the discrete search algorithm if a valid path exists from one world state to another. To do this it has to verify that any changes in the state are valid (i.e. objects cannot move unless the robot is holding them), and use a continuous planner to verify that a feasible path exists for the robot.

As we noted earlier, the multi-step planning algorithm depends on the programmer to supply a family of planning algorithms that can generate plans in all of the various stances that may exist in our problem. `reachable` will read the stance and then make a call to the appropriate planner to see if the states can be connected, returning a boolean value about the reachability of the new state, as well as (optionally, depending on the implementation) a distance to the new state. As a small optimization, if a valid trajectory is found then it will be returned so it can be stored until the end of the planning process, thus preventing the need to replan all the steps in the

Algorithm 1 getPlan

Input: world states $init$ and $goal$
Output: trajectory, if one exists
Let $state\_queue$ be a Priority Queue
$closed\_set \leftarrow \emptyset$
$state\_queue$.insert($init$)
while $state\_queue$ is not empty **do**
  $current \leftarrow state\_queue$.pop()
  $closed\_set$.insert($current$)
  if $current$ == $goal$ **then**
    trace steps backwards to construct $trajectory$
  end if
  for all $s \in$ adjacent($current$)$\backslash closed\_set$ **do**
    if s is reachable from $current$ **then**
      $h \leftarrow$ heuristic(s, $goal$)
      $g \leftarrow current$.g_val+ distance($current$, s)
      $f \leftarrow g + h$
      if s is not in $state\_queue$ **then**
        insert s in $state\_queue$ with priority f
      else
        if $f <$ current priority for s in $state\_queue$ **then**
          update s in $state\_queue$ with priority f
        end if
      end if
    end if
  end for
end while
return $trajectory$

final multi-step plan at the memory cost of storing trajectories for each valid edge in our search graph.

B. Adjacent

An important abstraction here is the idea of an adjacency. Two stances are considered to be adjacent if there exists a world state that is valid in both stances. The implementation of this algorithm requires a way to sample world states that are valid at the intersection of two stances.

In order to do multi-step planning we need the ability to find stances adjacent to our current stance. For manipulation problems we define stances by the set of grasps the robot is using, so finding adjacent stances is a matter of looking at each gripper and either releasing the current grasp or taking a new one, which provides a set of adjacent stances. It is then necessary to sample world states in the intersection of each of those stances with our base stance. This process is detailed in Algorithm 2. In the next section we will discuss the state sampling process.

C. State Sampling

Algorithm 3 outlines our state sampling process. The generation of individual samples is a random, application specific process that must be supplied by the programmer for a given application.

Two other aspects of this function are worth noting. First is the $sampled\_states$ data structure. This is a structure that

Algorithm 2 adjacent

Input: world state a
Output: set of adjacent world states adj
$adj \leftarrow \emptyset$
$adj\_stance \leftarrow \emptyset$
for g in robot grippers **do**
  if g is holding object **then**
    $release \leftarrow a.stance$ with g released
    $adj\_stance \leftarrow adj\_stance \cup release$
  else
    for i in available grasp points **do**
      $grab \leftarrow a.stance$ with gripper g grabbing i
      $adj\_stance \leftarrow adj\_stance \cup grab$
    end for
  end if
end for
for $stance$ in adj_stance **do**
  $adj \leftarrow adj \cup$ sampleStates($a.stance$, $stance$)
end for
return adj

provides an efficient way to store all previously sampled states along with all of the stances of which they are members. Furthermore, it can be queried with a pair of stances to efficiently return all already sampled states that are in both stances. In the next section we will briefly discuss how we implemented a data structure with these specifications.

Second, we have a parameter, $sample\_count$ that tells us how many samples to take before returning. This parameter can have a very large effect on the running time and solution quality of the planner. If it is set too high it will grow the branching factor of the search tree to make it too large. If it is set too low the planner may not be able to find a high quality, or any, solution.

Algorithm 3 sampleStates

Input: stances a and b
Output: set of world states at the intersection of a and b
$states \leftarrow sampled\_states$.atIntersection($a$, b)
while $states$.size $< sample\_count$ **do**
  $sample \leftarrow$ randomly selected state in intersection of a and b
  $states \leftarrow states \cup sample$
  $sampled\_states$.insert($sample$ in a and b)
end while
return $states$

D. State Storage and Reuse

In order to facilitate storing and retrieving world states by stance intersection, we created a data container we call a `VennSet` defined by two operations:

- **insert**($node$, set_list): Insert a node that is associated with a list of possibly preexisting sets.
- **intersect**(set_a, set_b): Return all nodes associated with both set_a and set_b.

Internally, the nodes are stored in a dynamically resizable array (C++ `std::vector` in our implementataion), and sets are stored as a mapping from from the set to an array of node IDs for the nodes in that set (we use the C++ `std::map`, which is usually implemented as a red-black tree). Thus `insert` can be implemented by adding the new node to the end of the node array, and then adding it's ID to the end of each array for each set it belongs to. The `intersect` query is implemented then by first sorting in place the arrays of node IDs for each set, then it is easy to linearly pass through to get the intersection. This may be slightly expensive the first time it is called for a given set, but by using a sorting algorithm that runs faster with 'mostly sorted' data, this cost can be minimized.

This is not the most asymptotically efficient structure possible for these queries (using hash tables for internal structures could provide an asymptotic advantage), but we found it to be sufficiently fast for our application.

E. Heuristic Function

Algorithm 4 lists the procedure for our proposed `minSteps` heuristic. This heuristic is based on a simple intuition that objects must move to the goal if they are not already there, and they do not move unless the robot moves them. As such, if an object is out of place, the state is at least one step away from a solution. Taking this one step further (with the inner **if** statement), we check if the robot is holding the object that must move. If not then we are one extra step from the goal as the robot must first reach to grab the object before it can be moved.

Ultimately, this is a relatively simple heuristic, though, as we will show in our experiments, it provides a significant speed-up.

Algorithm 4 minSteps

```
1:  Input: world states a and b
2:  Output: a lower bound on steps to get from a to b
3:  steps ← 0
4:  for object o in object list do
5:    if a.object_state[o] != b.object_state[o] then
6:      steps ← steps + 1
7:      if robot is not holding o in a then
8:        steps ← steps + 1
9:      end if
10:   end if
11: end for
12: return steps
```

F. Optimality

Since our algorithm relies on the A* search algorithm, it is able to inherit from it the associated guarantees on on optimal solution length for the search graph as sampled. However, the random sampling of the search graph prevents us from making a general optimality claim for the multi-step planning algorithm. We mentioned earlier that tuning the `sample_count` parameter can have a significant impact on solution quality. It is potentially an area of future research to look into ways to ramp up the intersection sampling density until a satisfactory solution is found.

In our experiments we use step count as the measure of solution quality. This works well for our problem since most paths have similar execution time, and there is a significant and constant time cost associated with taking and releasing grasp points. Solutions for other problems might reasonably like to choose measures of travel time, end effector travel distance, or some other metric of plan cost. Problems with a mobile robot in particular may benefit from different cost metrics. In this case it would also be necessary to develop a new heuristic specific to the cost function being used.

G. Heuristic Inflation

A common approach to accelerating a heuristic search process is to apply an inflation factor to the heuristic. If starting with an admissible heuristic this technique comes at the cost of optimality, however, the sub-optimality of the final solution is bounded by the factor used to expand the admissible heuristic.

We have experimented with using a very small inflation factor (1.05) to prevent the search process from needlessly expanding nodes with identical cost. It should be observed that since we use a cost metric of number of steps, and that cost is discrete, for problems with optimal solutions less than 20 steps, the factor 1.05 still guarantees an optimal solution.

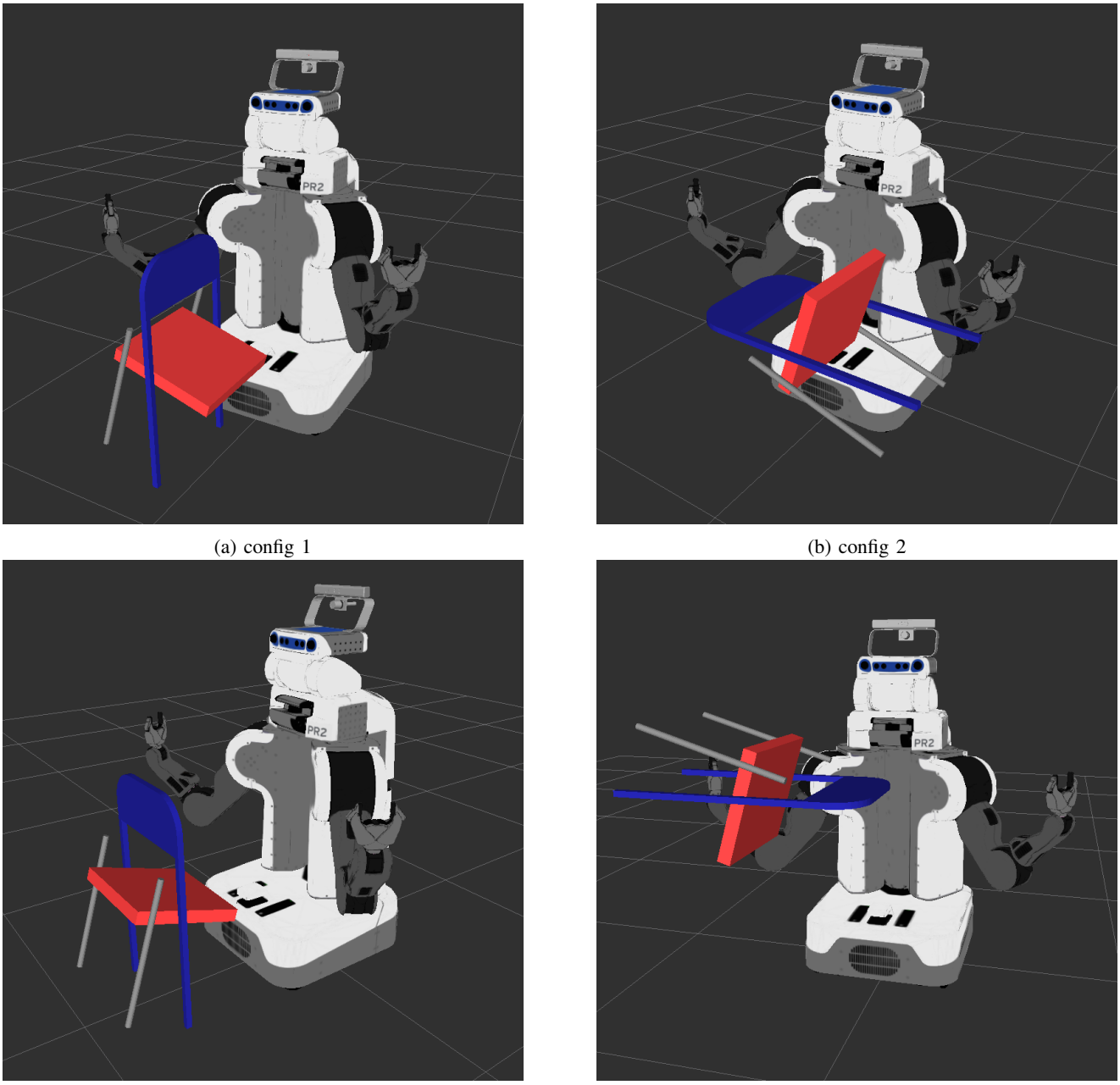
V. Simulation

We have implemented the multi-step planning algorithm for a simulated folding chair identical to the real chair used in our earlier work [6], as well as a simulated tabletop manipulation problem.

A. The Chair

Configuration files provide the planner with a full kinematic model of the chair, as well as a list of all the valid grasp points on the chair. Configuration files also provide information about how finely to discretize the states of the chair during the planning process. The experiments in this paper use 7 grasp points on the back of the chair, 10 on the seat, and discretize the joint into 5 states.

For the purposes of this planner, we assume the pose of the back of the chair is fixed in space. This is reasonable in some configurations, such as when the chair is resting on the ground in such a way that robot will not need to support the full weight of the chair while grasping it, as we demonstrated in our earlier work. In our simulated planning problems we are less concerned with the practical realities of such things as gravity (though it poses interesting challenges for future work), and so, as one can see in our Experiments sections, we will consider the planning problem with the chair in some less conventional configurations.



(a) config 1

(b) config 2

(c) config 3

(d) config 4

Fig. 5: Four initial configurations of the chair used for testing the multi-step planner.

B. Continuous Planners

As per the requirements of multi-step planning, we must provide a family of continuous planners capable of generating robot trajectories for the various stances that may exist. In this problem there are 3 such planners. A 2 free arm planner and a 1 free arm planner account for 2 of those, and are essentially identical except that in the 1 free arm planner the constrained arm is held stationary. The third planner is application specific to the closed kinematic chain formed when the robot grabs 2 links on the same object that have a joint between them, and works by interpolating through the joint positions of the object.

C. Tabletop Problem

Our tabletop manipulation problem involves two cylindrical objects on a table within reach of the robot that both need to be repositioned on the table. A single grasp point (from the top) is provided for each object, and the objects can be placed anywhere on the table in a grid of potential poses (we use a grid resolution of 5cm). Similar continuous planners are provided as with the chair folding problem, except for the closed kinematic chain which does not exist in this problem. Visualizations of the start and goal configurations are shown in Figures 6 and 7 respectively.

VI. Folding Chair Implementation

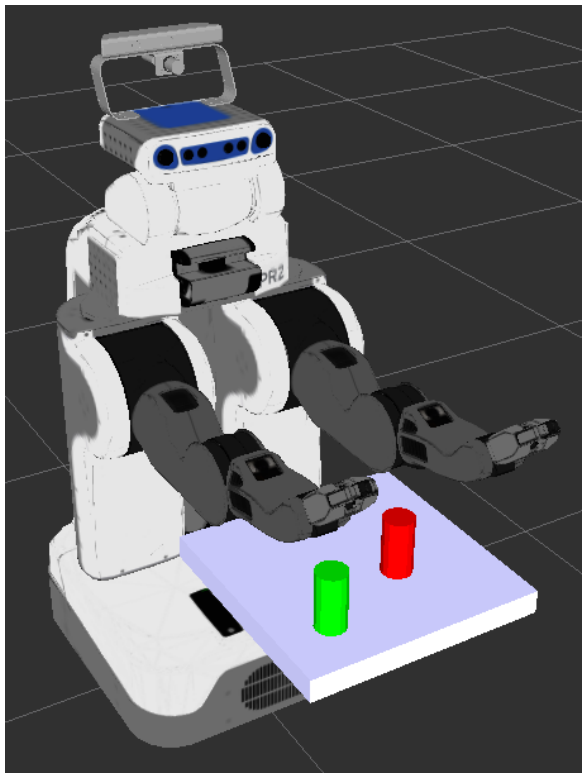


Fig. 6: Initial conditions for a 2 object tabletop manipulation problem.

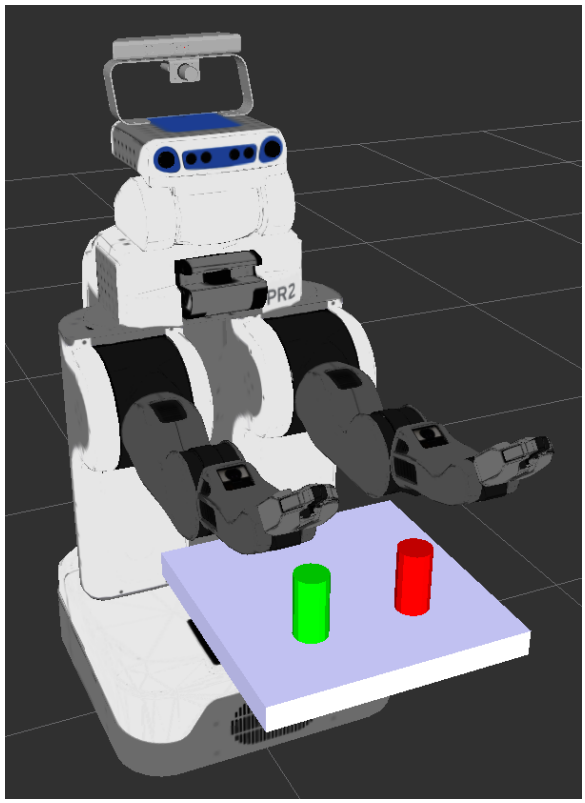


Fig. 7: Goal conditions for a 2 object tabletop manipulation problem, both objects have been repositioned.

The implementation of this algorithm involves resolving a number of specific technical challenges. Some of these are specific to the nature of the object we are working with, others are not. The sections below will cover the significant technical challenges and assumptions made by our implementation for the physical folding chair demonstration.

A. The Chair

We examine the problem of a robot that folds (or unfolds) a typical folding chair. The initial and goal states will be specified as poses for the chair, leaving the planner to decide where to hold the chair and how to move. We allow ourselves full prior knowledge of the shape and kinematics of the chair, as well as a list of available grasp points on the chair.

We perform this manipulation with the PR2 robot from Willow Garage, as a practical matter the robot is not strong enough to manipulate the chair arbitrarily, so we add an extra constraint to the state of the chair that requires two legs of the chair to be on the ground at all times, and we assume they will not slip. In most practical configurations this will ensure the robot does not have to support any significant forces with its arms. In this initial implementation we have also assumed that the pose of the chair base link (the back of the chair) will not change.

B. Problem Specifications

We allow the algorithm to start with a complete kinematic model of a real chair, supplied in the standard URDF format. We also provide a list of valid grasp points on the chair. Each grasp point is tied to a particular link on the chair, so it will continue to be valid through manipulation actions. Fig. 8 shows a visualization of available grasp points being evaluated during the planning process, note that some of the grasp points are shown for hypothetical configurations of the chair. Grasps are shown green when a valid kinematic solution could be found to perform a grasp, and red when one could not be found, or because that grasp is considered in collision.

Currently the list of grasp points passed to the algorithm is hand specified, however it would be sensible to do this process automatically. The running time of the algorithm, as well as the existence and quality of a solution, can depend heavily on how many and where grasp points are provided, so having a principled process here would be quite useful.

The initial state of the object (chair) is observed at the beginning of the planning process, the final state is specified as the state of the object and robot. Currently we specify the pose of the root link of the object to be unchanged, and only allow the robot to move the articulated joint of the object.

C. Chair State Estimation

Perceiving the exact state of the chair is also a challenging problem, however a general solution to that problem is outside the scope of this work. We have attached AR Tag style markers to the chair that can be used with a calibrated 2D camera to perceive the pose of the parts of the chair in the frame of the robot. We used the `ar_pose` ROS package to perceive

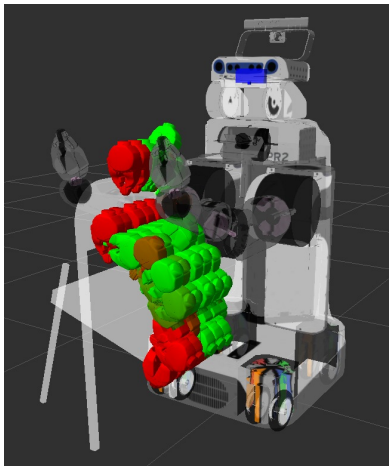


Fig. 8: Visualization of hypothetical grasp points during planning for different chair configurations. Grasps are shown as PR2 grippers, green for valid, red for invalid. Note that some grasps are shown for hypothetical chair positions.

these tags. In ROS we treat the chair as another robot with a URDF file, so it is necessary to extract from multiple observed tracking tags the joint angles of the chair. We calculate the observed rotation between two links of the chair, then project that rotation in quaternion space onto the constraint created by the rotational joint in the URDF specification of the chair. Note that an axis of rotation constraint becomes a hyperplane passing through the origin in 4D quaternion space, so we can use standard linear algebra techniques for this.

D. Adjacent Stance Expansion

For this robotic manipulation task stances can be defined by the state of the robot grippers with respect to what (if anything) they are holding and where they are holding it. This makes it relatively easy to sample stance intersections since they will always be characterized by a change of state for a gripper (grabbing or releasing something).

We maintain an index of sampled world states for every stance intersection, and by choosing an upper limit on how many states may by sampled for a given stance intersection we avoid the issue of oversampling an intersection if it is revisited.

The sampled world states are maintained in a data structure that associates each world state with the stances it is in, and allows efficient queries of stance intersections to find all world states in that intersection.

E. Reachability Checking

Reachability checking involves determining when a state transition is feasible in 1 step. We have to verify three things, kinematics, collisions, and path validity.

To check kinematics we look at the initial and final states wherever the robot is holding an object. If the robot is holding an object, we use inverse kinematics to verify that a valid kinematic solution exists for the configuration of the object in that state.

Currently we check for collision only by specifically excluding grasp-configuration combinations that are known to cause collisions, however it makes sense to use a more generalized collision checker here.

Checking path validity can be done by a standard robot arm path planner for whatever change in robot joint state is necessary from initial to final conditions. We are currently using joint space interpolation for unconstrained arms, but with further work we plan to incorporate a standard collision avoidance path planner here. Here we add some additional constraints that the configuration of the object may not change unless the robot is holding it with both arms on different links (this is a constraint specific to the chair, but it is conceptually simple to generalize to other objects, a joint cannot be moved unless the robot is holding both the links). When the robot does change the configuration of the object, this creates some special constraints as a closed kinematic chain, discussed below.

F. Closed Kinematic Chains

Solving kinematics for closed chains in general can be difficult. We deal with this problem by prioritizing the pose of the object. When a path must be found where the robot changes the configuration of the object, we perform a linear interpolation in the joint space of the object, and then solve inverse kinematics for the robot to hold the object at points along that path. We keep the joint states of the robot smooth by seeding each IK solver with the solution from the previous state.

G. Path Execution

We assume that our environment is very predictable and so after finding a solution trajectory we simply execute it without sensor feedback.

VII. Experiments

We have been able to demonstrate the multi-step planning algorithm running both in simulation and on the real robot, with successful planning and execution results.

A. Simulated Chair

We ran the planner for four different chair configurations and for each chair configuration we tested with 3 heuristics. A null heuristic that always returns 0, thus producing a naive breadth-first search, the `minSteps` heuristic discussed in the Algorithm section, and the `minSteps` heuristic inflated by a factor of 1.05. Fig 5 shows the four chair initial configurations we used. The chair can be seen with a blue back, red seat, and grey rear legs.

Table I shows the differences between the two search processes. Each configuration-planner combo has been run 5 times, and results are shown as the average of those 5 plans; standard deviation is shown in parentheses. All of the planning results were run on an Intel Core i7-950 desktop with 8 GB of ram. We evaluate the planners on 3 metrics:

- **planning time:** cpu time spent in the `getPlan` function.
- **search tree nodes:** total number of reachable nodes added to the search tree.

TABLE I

	null heuristic		
Config	Time (s)	Nodes	Branching Factor
1	19.37 (1.15)	497 (26)	2.16 (0.06)
2	24.02 (0.60)	968 (19)	2.11 (0.06)
3	78.45 (0.53)	1830 (15)	1.42 (0.01)
4	15.86 (0.37)	821 (15)	2.63 (0.05)
	minSteps		
	Time (s)	Nodes	Branching Factor
1	4.39 (0.26)	174 (2.9)	2.84 (0.15)
2	1.96 (0.03)	111 (1.4)	18.27 (0.23)
3	28.61 (2.27)	985 (46)	1.63 (0.07)
4	2.13 (0.07)	112 (2.2)	15.89 (0.32)
	minSteps with inflation		
	Time (s)	Nodes	Branching Factor
1	6.25 (0.44)	261 (20)	3.32 (0.16)
2	1.88 (0.02)	94 (0.8)	15.57 (0.13)
3	25.66 (0.67)	1010 (13)	1.98 (0.03)
4	3.53 (0.18)	308 (37)	9.29 (0.06)

Running time, search graph nodes, and branching factor for the 4 initial configurations with the null heuristic, the minSteps heuristic, and the minSteps heuristic with a 1.05 inflation factor. Each data point is an average of 5 runs, with the standard deviation shown in parentheses.

- **branching factor:** average number of reachable successors from each expanded search node.

Table II shows the speed-up factor of both non-null heuristics over the null heuristic.

B. Planning Difficulty

We can see based on the planning times and other metrics that not all the configurations were equally difficult. Specifically, 3 appears to be quite difficult, where as 2 and 4 are easier. Qualitatively we note that configurations 1 and 3 require solutions where the robot starts by grasping the seat of the chair from the edge to start the motion, and then has to regrasp from underneath to close the chair. This is due to the grasp from underneath being kinematically impossible in the initial conditions, but the grasp from the edge being in collision with the chair in the final conditions. Motion of this type was also required in the solution present in our previous work with a real robot and chair. By contrast, configurations 2 and 4 are able to grasp the seat from underneath in the initial conditions and can thus solve the problem in fewer steps.

C. Speed-Up Factors

Based on the table we can see that while all configurations got a significant speed-up from the `minSteps` heuristic, some gained a lot more than others. Specifically, our 'easier' problems (2 and 4) got a much bigger speed up than the harder problems. We suspect this is related to the fact that we are using a relatively simple heuristic. Intuitively, a simple heuristic will be very effective at directing search graph expansion in areas of the problem where decision making is easy, but will be less useful in areas where decision making is hard. Under this intuition, configurations with more 'hard' decisions to make will not be able to shed as much of their planning time. We think this also suggests that there may be further headroom for improvement on difficult problems with more sophisticated heuristics. Heuristic inflation seemed to produce inconsistent results, resulting in better runtimes in some problems, and worse in others.

TABLE II

Speed-Up Factor

Config	minSteps	minSteps w/ inflation
1	4.4	3.1
2	12.3	12.8
3	2.7	3.1
4	7.4	4.5

Speed-up factor shows how many times faster the search is with each heuristic over the null heuristic, in terms of running time.

D. Branching Factors

We measure branching factor by counting the number of times the planner successfully evaluates the reachability of another state (about line 14 in Algorithm 1) and dividing that by the size of the closed set when planning is complete. Note that this is not the average number of adjacencies, as already closed states are not counted, and neither are calls to `reachable` that return false. A property of this process is that the branching factor starts out quite large as the early states have many reachable successors, but as states move into the closed set the average branching factor shrinks, to the point that if the planner terminates without finding a plan, it will show a branching factor approaching 1 (though possibly higher, depending on how cyclic the search graph is). This is an interesting metric because it gives us a measure of something related to the exponential difficulty of the problem, as well as how much of the graph is being left unexpanded by our heuristic.

Looking at the branching factors we notice that all configurations with the null heuristic have branching factors around 2, and with the `minSteps` heuristic configurations 1 and 3 (our 'hard' problems) also have a branching factor around 2 (though in each case higher than their corresponding null heuristic search), but configurations 2 and 4 have radically higher branching factors in the 15-19 range.

We think this shows that in these simpler configurations the `minSteps` heuristic has been extremely successful in cutting out unnecessary exploration, leaving behind a large branching factor. This also corresponds to the significant decreases in computation time for these problems.

E. Folding Chair Physical Demonstration

The physical demonstration of the algorithm runs on a slightly different code base from the simulation results, but is comparable to the simulation using a null heuristic. It was previously presented in [6].

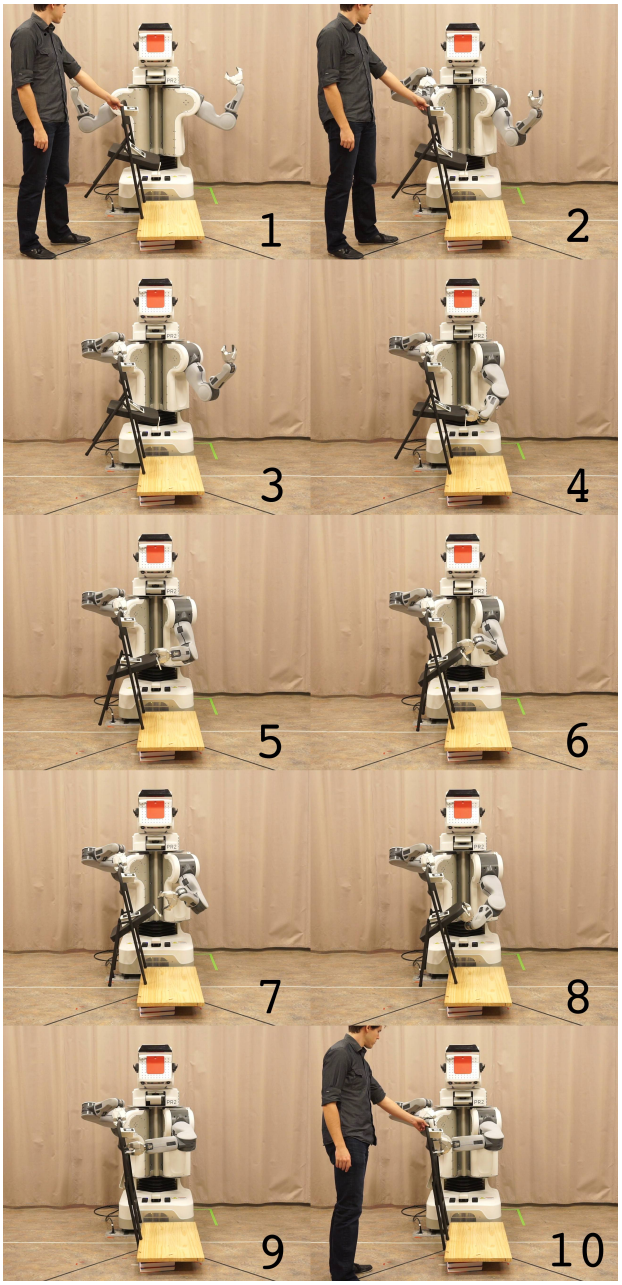


Fig. 9: A sequence of robot states from a plan executed in the lab. The initial state of the chair is shown in tile 1, the goal state is reached in tile 10. Observe how the robot moves the seat part way, then transitions to a different grip (tiles 7-8) from which it is possible to finish the manipulation.

Fig. 9 shows a sequence of states while executing a resulting plan on the real robot. The chair starts in an unfolded state with a goal configuration to have it folded up. The plan is executed blind and is susceptible to perception errors in the initial state of the chair, or small miscalibrations in the sensors, so as a result some undesired motion of the chair is observed. Despite this the plan executes successfully and moves the chair into a folded state. Of particular interest should be the state where the planner has naturally discovered the need to reposition the gripper part way through the motion. This is a result of the fact that in order of fully close the chair the robot must grasp the seat from below to avoid a collision. However, those grasp poses are not kinematically feasible in the unfolded state, so it is necessary to start with different grasp, fold the chair part way, then change grasp point.

The full video of the test can be seen here: [http://youtu.be/cmL4UpjyMng].

TABLE III

Tabletop Planning Problem

Heuristic	Time (s)	Nodes	Branching Factor
minSteps	31.56 (0.19)	1991 (5)	210 (38)
minSteps * 1.05	47.12 (0.10)	2870 (4)	161 (15)

Planning results for our 2-object tabletop planning problem. Results are averaged over 5 runs and standard deviations shown in parentheses.

F. Tabletop Planning

In the tabletop planning problem we do not have comparisons with a null heuristic since the problem is significantly more complex. We can, however, observe some comparisons to the chair folding problem. The planning time here is generally significantly larger, and the branching factors are an order of magnitude larger than any we have seen on the chair folding problem. The large branching factors in particular suggest that the improvement here from a useful heuristic is even more important. Intuitively, the larger branching factor is an expected result of the robot having more choices at each adjacency in the tabletop problem. The chair only had one free dimension, where as our 2 can problem has 2 dimensions per can, greatly multiplying the available world configurations. In our state sampler we discretized the table into a grid with a resolution of 5cm. Finer discretizations are desirable for completeness, however, in addition to multiplying the number of reachable states from a given state, they also decrease the probability of the search graph revisiting old nodes, which further pushes up the branching factor.

VIII. Discussion and Future Work

Although we believe this multi-step planning approach is quite powerful, there are some situations that present challenges. Without enough samples at a stance intersection, the probability of getting within an acceptable distance of the goal position may become very small. As such, a coarse discretization, particularly for more subtle manipulation tasks, could be difficult to use. Also, as we increase the degrees of freedom of the object, this will multiply the number of samples necessary to achieve the same coverage of a stance intersection, and thus multiply the branching factor of the search tree (in nodes where the object can be moved).

A limitation of this algorithm is that once a plan is generated it must be executed blind, without any updates based on sensor data. This is usually how traditional motion planners are operated, but when multi-step motion involves interacting with objects, the system becomes less predictable. If a robot gripper is slightly ill-positioned then a grasp may fail, or it may succeed but move the object into an unexpected pose. We observed this behavior in our own experiments. Extensions to

this algorithm might provide the opportunity for replanning or adjusting plans based on new sensor data.

A reasonable extension of this work could also include an algorithm for generating grasp points on the object. The same branching factor issues would apply as above, but a finer grained selection of grasp points may be necessary for problems with tighter kinematic constraints.

Based on our results in simulation with the 'easy' configurations, we think that further development of the heuristic search process may lead to stronger improvements with the 'hard' configurations. One area that we think is particularly promising in this regard is the use of the multi-heuristic A* algorithm [18]. Because multi-heuristic A* can remain optimal with inadmissible heuristics, this opens the door to other forms of heuristic generation that may not be able to guarantee admissibility. Furthermore, significant speed-ups in the discrete search process may create opportunity for using multi-step planning on more complex problems with more steps or bigger dimensionality.

IX. Conclusion

Combined task and motion planning problems represent a new frontier in robotic planning tasks. They allow the programmer to no longer be concerned with the details of where a robot holds an object, or the order of operations necessary for larger problems. In this paper we presented a new approach to combined task and motion planning problems using multi-step planning. We demonstrated the efficacy of the approach by implementing it for a complex manipulation problem involving an articulated object, in this case a folding chair.

In this work we have also proposed heuristic search techniques as a significant development in multi-step planning, and demonstrated the performance of this strategy with our `minSteps` heuristic for our multi-step chair folding problem as well as a new tabletop problem. Using multiple initial configurations we have shown between 2.7x and 12.8x speedup in the planning procedure, depending on the configuration. We believe that heuristic search techniques will be very powerful for any future implementations of multi-step planning, and we are looking toward future work that will use these speed ups to expand the applicability of this technique to more complex problems.

References

[1] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[2] I. A. Sucan and S. Chitta, "Moveit!" *Available Online http://moveit.ros.org*, 2013.

[3] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[4] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *IEEE Intl. Conference on Robotics and Automation*, 2014.

[5] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "Smt-based synthesis of integrated task and motion plans from plan outlines," in *IEEE Intl. Conference on Robotics and Automation*, 2014.

[6] M. Pflueger and G. S. Sukhatme, "Multi-step planning for robotic manipulation," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 2496–2501.

[7] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, 1996.

[8] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[9] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *IEEE Intl. Conf. on Robotics and Automation*, vol. 2. IEEE, 2000, pp. 995–1001.

[10] J. Sturm, C. Stachniss, and W. Burgard, "A probabilistic framework for learning kinematic models of articulated objects," *Journal on Artificial Intelligence Research (JAIR)*, vol. 41, pp. 477–626, August 2011.

[11] D. Katz, A. Orthey, and O. Brock, "Interactive perception of articulated objects," in *12th International Symposium on Experimental Robotics*, Delhi, India, Dec 2010.

[12] T. Bretl, S. Lall, J.-C. Latombe, and S. Rock, "Multi-step motion planning for free-climbing robots," in *Algorithmic Foundations of Robotics VI*. Springer, 2005, pp. 59–74.

[13] K. Hauser, T. Bretl, and J.-C. Latombe, "Learning-assisted multi-step planning," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2005, pp. 4575–4580.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[15] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1994, pp. 3310–3317.

[16] ——, "The focussed d* algorithm for real-time replanning." in *International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 95, 1995, pp. 1652–1659.

[17] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

[18] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a*," in *Robotics: Science and Systems (RSS)*, 2014.

[19] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2009, pp. 489–494.

[20] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2011, pp. 4569–4574.

[21] J. Barry, L. P. Kaelbling, and T. Lozano-Pérez, "A hierarchical approach to manipulation with diverse actions," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2013, pp. 1799–1806.

[22] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 729–746, 2004.

[23] M. A. Diftler, J. Mehling, M. E. Abdallah, N. A. Radford, L. B. Bridgwater, A. M. Sanders, R. S. Askew, D. M. Linn, J. D. Yamokoski, F. Permenter *et al.*, "Robonaut 2-the first humanoid robot in space," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2011, pp. 2178–2183.

Max Pflueger received his B.S. degree in Engineering from Harvey Mudd College and his M.S. in Compute Science from the University of Southern California (USC). He has spent time working as a professional software engineer at Yahoo! Inc. and he is currently a Ph.D. student in Computer Science at USC.

His research interests focus on using robotic motion planning to solve complex problems, and developing techniques and applications for using machine learning to solve robotic planning and manipulation problems.

Gaurav S. Sukhatme is a Professor of Computer Science (joint appointment in Electrical Engineering) at the University of Southern California (USC). He received his undergraduate education at IIT Bombay in Computer Science and Engineering, and M.S. and Ph.D. degrees in Computer Science from USC. He is the co-director of the USC Robotics Research Laboratory and the director of the USC Robotic Embedded Systems Laboratory which he founded in 2000. His research interests are in multi-robot systems and sensor/actuator networks. He has published extensively in these and related areas. Sukhatme has served as PI on numerous NSF, DARPA and NASA grants. He is a Co-PI on the Center for Embedded Networked Sensing (CENS), an NSF Science and Technology Center. He is a fellow of the IEEE and a recipient of the NSF CAREER award and the Okawa foundation research award. He is one of the founders of the Robotics: Science and Systems conference. He was program chair of the 2008 IEEE International Conference on Robotics and Automation and is program chair of the 2011 IEEE/RSJ International Conference on Robots and Systems. He is the Editor-in-Chief of Autonomous Robots and has served as Associate Editor of the IEEE Transactions on Robotics and Automation, the IEEE Transactions on Mobile Computing, and on the editorial board of IEEE Pervasive Computing.