Efficient Multi-Step Planning for Robotic Manipulation

Max Pflueger and Gaurav S. Sukhatme

Abstract—Multi-step planning is a promising technique for solving complex manipulation problems directly in the state space of the object, however, previous formulations were too slow to be practical except on problems with small search spaces. In this paper we propose and demonstrate two optimizations for this technique that together yield a significant speed-up on a previously used benchmark problem. The optimizations are 1. using the A* discrete search algorithm with proposed heuristics, and 2. a data structure for tracking the search graph. Depending on initial configuration, our results show a 2.7x-12.8x speed up in planning time as well as significant reductions in other search tree metrics. This faster planning algorithm also allows us to demonstrate the multi-step planning process on a combinatorially more complex tabletop planning problem.

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. However, 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.

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. 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, 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, 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. (We discuss this process in more detail in section IV: Algorithm.)

In our previous work [6] we demonstrated how multi-step planning can be used to solve the non-trivial problem of manipulation of a folding chair. In this work we will build upon that 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 in 1968 by Hart and Nilsson [7] and has become a standard algorithm since then. Other heuristic search algorithms exist such as D* [8] and its variants [9] [10], and multi-heuristic A* [11]. Examining the properties of these algorithms applied to multi-step planning could be a promising future area of research.

A. Continuous Space Planners

In addition to the discrete planner, the other important piece of multi-step planning is the continuous space motion planners. In our work we are only concerned with kinematic motion planners. Some of the most popular motion planners for robot arms are based on rapidly-exploring random trees (RRTs) [3] and variants such as [12] [13], or probabilistic roadmaps (PRMs) [14]. There also approaches based on trajectory optimization such as [15] [16].

B. Combined Task and Motion Planning

Multi-step planning has been used before in other domains of robotics, notably by Tim Bretl, Kris Hauser, and others in work with a wall climbing robot [17] [18]. Beyond that other researchers have used a variety of techniques to approach the combined task and motion planning problem. Some have relied on a plan outline supplied by an expert such as in [5], where as others have designed ways to interface logic based task planners with physical constraints from the world [4]. Barry et. al. [19] have demonstrated a technique based on manipulation with highly diverse actions. The formulation developed by Siméon et. al. [20] shares many similarities with our own.

III. PROBLEM FORMULATION

We formally 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 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.

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. In this work we only consider minimizing the number of steps (stance transitions).

IV. ALGORITHM

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 also refers to a general heuristic function. This can be any admissible heuristic for our application that also matches the cost calculation for \( g \) values. Note that in the algorithm below, we are calculating \( g \) values as the number of steps, though this could be easily modified for other metrics such as ones based on the trajectories returned by reachable. In this paper we are proposing the minSteps heuristic outlined in Algorithm 4. In the Experiments section we will show a comparison of how the minSteps heuristic compares to a null heuristic (always returns 0).

A. Reachable

The reachable function tells us whether or not a continuous plan exists to connect two states. 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. 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 final multi-step plan at the memory cost of storing trajectories for each valid edge in our search graph.

B. Adjacent

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
Algorithm 1 getPlan

1: \textbf{Input:} world states \emph{init} and \emph{goal}
2: \textbf{Output:} trajectory, if one exists
3: Let \emph{state_queue} be a Priority Queue
4: \emph{closed_set} $\leftarrow \emptyset$
5: \emph{state_queue}.insert(\emph{init})
6: \textbf{while} \emph{state_queue} is not empty \textbf{do}
7: \hspace{1em} \emph{current} $\leftarrow$ \emph{state_queue}.pop()
8: \hspace{1em} \emph{closed_set}.insert(\emph{current})
9: \hspace{1em} \textbf{if} \emph{current} == \emph{goal} \textbf{then}
10: \hspace{2em} trace steps backwards to construct \emph{trajectory}
11: \hspace{1em} \textbf{end if}
12: \hspace{1em} \textbf{for all} \emph{s} $\in$ adjacent(\emph{current})\\setminus\emph{closed_set} \textbf{do}
13: \hspace{2em} \textbf{if} \emph{s} is reachable from \emph{current} \textbf{then}
14: \hspace{3em} \emph{h} $\leftarrow$ heuristic($\emph{s}$, \emph{goal})
15: \hspace{3em} \emph{g} $\leftarrow$ \emph{current}.g+1
16: \hspace{3em} \emph{f} $\leftarrow$ \emph{g} + \emph{h}
17: \hspace{2em} \textbf{if} \emph{s} is not in \emph{state_queue} \textbf{then}
18: \hspace{3em} insert \emph{s} in \emph{state_queue} with priority \emph{f}
19: \hspace{2em} \textbf{else}
20: \hspace{3em} \textbf{if} \emph{f} < current priority for \emph{s} in \emph{state_queue} \textbf{then}
21: \hspace{4em} update \emph{s} in \emph{state_queue} with priority \emph{f}
22: \hspace{2em} \textbf{end if}
23: \hspace{2em} \textbf{end if}
24: \hspace{1em} \textbf{end if}
25: \hspace{1em} \textbf{end for}
26: \hspace{1em} \textbf{end while}
27: \textbf{return} \emph{trajectory}

Algorithm 2 adjacent

\textbf{Input:} world state $a$
\textbf{Output:} set of adjacent world states $adj$
\begin{itemize}
\item $adj \leftarrow \emptyset$
\item $adj\_stance \leftarrow \emptyset$
\end{itemize}
\textbf{for} $g$ in robot grippers \textbf{do}
\begin{itemize}
\item \textbf{if} $g$ is holding object \textbf{then}
\hspace{1em} release $\leftarrow$ $a$.stance with $g$ released
\hspace{1em} $adj\_stance \leftarrow adj\_stance \cup$ release
\item \textbf{else}
\hspace{1em} \textbf{for} $i$ in available grasp points \textbf{do}
\hspace{2em} grab $\leftarrow a$.stance with gripper $g$ grabbing $i$
\hspace{2em} $adj\_stance \leftarrow adj\_stance \cup$ grab
\hspace{1em} \textbf{end for}
\item \textbf{end if}
\end{itemize}
\textbf{end for}
\textbf{for} stance in $adj\_stance$ \textbf{do}
\hspace{1em} $adj \leftarrow adj \cup$ sampleStates($a$.stance, stance)
\textbf{end for}
\textbf{return} $adj$

Algorithm 3 sampleStates

\textbf{Input:} stances $a$ and $b$
\textbf{Output:} set of world states at the intersection of $a$ and $b$
\begin{itemize}
\item $states \leftarrow$ sampled_states.atIntersection($a$, $b$)
\end{itemize}
\textbf{while} $states$.size $<$ sample\_count \textbf{do}
\hspace{1em} sample $\leftarrow$ randomly selected state in intersection of $a$ and $b$
\hspace{1em} $states \leftarrow states \cup$ sample
\hspace{1em} sampled_states.insert(sample in $a$ and $b$)
\textbf{end while}
\textbf{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:
\begin{itemize}
\item \textbf{insert}(node, set\_list): Insert a node that is associated with a list of possibly preexisting sets.
\item \textbf{intersect}(set\_a, set\_b): Return all nodes associated with both set\_a and set\_b.
\end{itemize}

Internally, the nodes are stored in a dynamically resizable array (C++ std::vector in our implementation), 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

Input: world states a and b
Output: a lower bound on steps to get from a to b
steps ← 0
for object o in object list do
  if a.object_state[o] != b.object_state[o] then
    steps ← steps + 1
  if robot is not holding o in a then
    steps ← steps + 1
  end if
end if
end for
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 implementation we use step count as the measure of solution quality, regardless of how long each step is. Other implementations might reasonably like to choose solutions of shortest total time or some other metric of plan distance. 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. IMPLEMENTATION SPECIFICS

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.

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. In this application none of our planners do geometric collision checking, though we do specifically exclude joint position to grasp pairs that are known to cause the gripper to collide with the chair.

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.

VI. EXPERIMENTS

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 1 shows the four chair initial configurations we used. The chair can be seen with a blue back, red seat, and grey rear legs. Although in this paper we are running on a modified code base, the null heuristic is functionally comparable to our earlier paper in [6].

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 func-
TABLE I

<table>
<thead>
<tr>
<th>Config</th>
<th>null heuristic</th>
<th>minSteps</th>
<th>minSteps with inflation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Nodes</td>
<td>Branching Factor</td>
</tr>
<tr>
<td></td>
<td>(s)</td>
<td>(s)</td>
<td>(s)</td>
</tr>
<tr>
<td>1</td>
<td>19.37 (1.15)</td>
<td>497 (26)</td>
<td>2.16 (0.06)</td>
</tr>
<tr>
<td>2</td>
<td>24.02 (0.60)</td>
<td>968 (19)</td>
<td>2.11 (0.06)</td>
</tr>
<tr>
<td>3</td>
<td>78.45 (0.53)</td>
<td>1830 (15)</td>
<td>1.42 (0.01)</td>
</tr>
<tr>
<td>4</td>
<td>15.86 (0.37)</td>
<td>821 (15)</td>
<td>2.63 (0.05)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Nodes</th>
<th>Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.39 (0.26)</td>
<td>174 (2.9)</td>
</tr>
<tr>
<td>2</td>
<td>1.96 (0.03)</td>
<td>111 (1.4)</td>
</tr>
<tr>
<td>3</td>
<td>28.61 (2.27)</td>
<td>985 (46)</td>
</tr>
<tr>
<td>4</td>
<td>2.13 (0.07)</td>
<td>112 (2.2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Nodes</th>
<th>Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.25 (0.44)</td>
<td>261 (20)</td>
</tr>
<tr>
<td>2</td>
<td>1.88 (0.02)</td>
<td>94 (0.8)</td>
</tr>
<tr>
<td>3</td>
<td>25.66 (0.67)</td>
<td>1010 (13)</td>
</tr>
<tr>
<td>4</td>
<td>3.53 (0.18)</td>
<td>308 (37)</td>
</tr>
</tbody>
</table>

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.

- **search tree nodes**: total number of reachable nodes added to the search tree.
- **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.

### A. 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.

### B. 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
higher branching factors in the 15-19 range.

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.

C. 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. 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.

D. 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.

VII. DISCUSSION AND FUTURE WORK

Based on our results 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 [11]. 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 open the opportunity for using multi-step planning on more complex problems with more steps or bigger dimensionality.

VIII. CONCLUSION

In this work have proposed heuristic search techniques as a significant improvement over our previous implementation of multi-step planning, and demonstrated the performance of this strategy with our new 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


