Search-based Planning for Manipulation with Motion Primitives

Benjamin J. Cohen¹, Sachin Chitta², and Maxim Likhachev³

Abstract—Heuristic searches such as A* search are highly popular means of finding least-cost plans due to their generality, strong theoretical guarantees on completeness and optimality and simplicity in the implementation. In planning for robotic manipulation however, these techniques are commonly thought of as impractical due to the high-dimensionality of the planning problem. In this paper, we present a heuristic search-based manipulation planner that does deal effectively with the high-dimensionality of the problem. The planner achieves the required efficiency due to the following three factors: (a) its use of informative yet fast-to-compute heuristics; (b) its use of basic (small) motion primitives as atomic actions; and (c) its use of ARA* search which is an anytime heuristic search with provable bounds on solution suboptimality. Our experimental analysis on a real mobile manipulation platform with a 7-DOF robotic manipulator shows the ability of the planner to solve manipulation in cluttered spaces by generating consistent, low-cost motion trajectories while providing guarantees on completeness and bounds on suboptimality.

I. INTRODUCTION

Many planning problems in robotics can be represented as finding a least-cost (or close to least-cost) trajectory in a graph. Heuristic searches such as A* search [6] have often been used to find such trajectories. There are a number of reasons for the popularity of heuristic searches. First, most of them typically come with strong theoretical guarantees such as completeness and optimality or bounds on suboptimality [15]. Second, there exist a number of anytime heuristic searches that find the best solution they can within the provided time for planning [5], [19], [20], [12]. Third, there exist a number of incremental heuristic searches that can reuse previous search efforts to find new solutions much faster when previously unknown obstacles are discovered [17], [9], [13]. Finally, treating a planning problem as finding a good quality path in a graph is advantageous because it allows one to incorporate complex cost functions, complex constraints and represent easily arbitrarily shaped obstacles with grid-like data structures [4], [11]. Consequently, heuristic search-based planning has been used to successfully solve a wide variety of planning problems in robotics.

Despite the wide popularity of heuristic searches, they typically have not been used for motion planning for high-DOF robotic manipulators [2]. The main reason for this is the high-dimensionality of the planning problem. In this paper, we present a heuristic search-based planner for manipulation that combats effectively this high dimensionality by exploiting the following three observations. First, the solutions found in a low-dimensional manifold of the workspace can serve as highly informative heuristics and can therefore guide search in the joint angle state-space quite well. Second, the majority of complex motion plans can be decomposed into a small set of basic (small) motion primitives. These motion primitives can be used to construct a graph on which the heuristic search is executed. The graph is sparser compared with the N-dimensional grid resulting from the discretization of N joint angles for an N-DOF manipulator. As explained later, this motion primitive-based graph also allows us to optimize for smoothness in actions at a small increase in the size of the graph. Third, while finding a solution that is provably optimal is expensive, finding a solution of bounded suboptimality can often be drastically faster. To this end, we employ an anytime heuristic search, ARA* [12], that finds solutions with provable bounds on suboptimality and improves these solutions until allotted time for planning expires.

The paper is organized as follows. It first briefly describes some of the existing approaches to planning for manipulators including the widely popular sampling-based approaches. It then explains our heuristic search-based planner including the graph it builds, the heuristic it uses and the search it employs. Section IV mentions other extensions and optimizations we have developed for the planner including the extension that allows the planner to manipulate objects while satisfying constraints (e.g., carrying a cup without turning it upside down). Section V presents the experimental analysis of the planner in simulation and on a real mobile manipulation platform with a 7-DOF robotic manipulator (Figure 1). The experimental results shows the ability of the planner to solve manipulation in cluttered spaces by generating consistent, low-cost motion trajectories while providing guarantees on completeness and bounds on suboptimality.

II. RELATED WORK

Sampling-based motion planners [8], [10], [1] have gained tremendous popularity in the last decade. They have been shown to consistently solve impressive high-dimensional motion planning problems. In addition, these methods are simple, fast and general enough to solve a variety of motion planning problems. Sampling-based methods have also been extended to support motion constraints through rejection sampling [18].
Our approach to motion planning differs from these algorithms in several aspects. First, sampling-based motion planner are mainly concerned with finding any feasible path rather than minimizing the cost of the solution. By sacrificing cost minimization, these approaches gain very fast planning speeds. Searching for a feasible path however may often result in the solutions of unpredictable length with superfluous motions, motions that graze the obstacles, and jerky trajectories that may potentially be hard for the manipulator to follow. To compensate for this, various smoothing techniques have been introduced. While often helpful, they may fail to help in cluttered environments. Second, sampling-based approaches provide no guarantees on the sub-optimality of the solution and provide completeness only in the limits of samples. In contrast, the search-based planning tries to find the solutions with minimal costs and provides guarantees on solution suboptimality w.r.t. the constructed graph. These aspects are valuable when solving motion planning problems for which the minimization of objective function is important and when consistent behavior is expected.

Several motion planning algorithms have been developed that also try to find solutions of minimal cost [7], [16]. One of the most recent algorithms in this category is Covariant Hamiltonian Optimization and Motion Planning, or CHOMP [16]. It works by creating a naive initial trajectory from the start position to the goal, and then running a modified version of gradient descent on the cost function. CHOMP offers numerous advantages over sampling-based approaches such as the ability to optimize trajectories for smoothness and to stay away from obstacles when possible. Our approach is similar to CHOMP in that we also recognize the importance of cost minimization but, in addition, we provide the guarantees on the global solution suboptimality.

III. ALGORITHM

The operation of our algorithm is based on constructing a motion primitive-based graph and searching this graph for a low-cost solution. In the following sections we explain the construction of the motion primitive-based graph, the cost function used to assign edge costs in the graph, the heuristic function that guides the graph search in finding the solution, and finally graph search itself. It is important to note that the actual graph construction is interleaved with graph search so that only the portion of the graph needed by search is actually stored in memory. This is important because the full graph is too large to store in memory and would be too computationally expensive even to construct.

A. Graph Construction

The graph structure we use was inspired by the success of lattice based planners in planning dynamically feasible trajectories [11]. Lattice-based representation is a discretization of the configuration space into a set of states, and connections between these states, where every connection represents a feasible path. As such, lattices provide a method for motion planning problems to be formulated as graph searches. However, in contrast to many graph-based representations (such as 4-connected or 8-connected grids), the feasibility requirement of lattice connections guarantees that any solutions found using a lattice will also be feasible. This makes them very well suited to planning for non-holonomic and highly-constrained robotic systems. Let us use the notation $G = (S, E)$ to denote the graph $G$ we construct, where $S$ denotes the set of states of the graph and $E$ is the set of transitions between the states. The states in $S$ are the set of possible (discretized) joint configurations. Similary to lattice-based representations, we confine the transitions in $E$ to be a predefined set of feasible motion primitives. \(^1\) We define a state $s$ as $n + 1$-tuple $(\theta_0, \theta_1, \theta_2, ..., \theta_n, m)$ for a manipulator with $n$ joints. In this definition, $m$ represents the index of the motion primitive used to reach the state $s$. This additional variable is used for path smoothing and will be explained later in the paper. The graph is dynamically constructed by the graph search as it expands states since pre-allocation of memory for the entire graph would be infeasible for an $n$-DOF manipulator with any reasonable $n$. Each motion primitive is a single vector of joint velocities, $(v_0, v_1, v_2, ..., v_n)$ for all of the joints in the manipulator. The set of primitives is the set of the smallest possible motions that can be performed at any given state. Therefore, a primitive is the difference in the global joint angles of neighboring states. This set is the same for any state at which it is executed which allows us to pre-evaluate and pre-compute motion primitives offline.

In our experiments with a 7-DoF manipulator, 14 basic motion primitives were used along with eight additional compound primitives. Seven basic motion primitives moved one joint each a pre-defined amount in the positive direction and the other seven moved the same amount in the negative direction. The eight additional motion primitives moved several joint angles at a time and provided better coverage.

\(^1\)The term “motion primitive” is sometimes used in planning literature to represent a higher level action such as opening a door, swinging a tennis racket, or pushing a button. This is different from our use of the term: we use “motion primitive” to denote a basic (atomic) feasible motion.
of the workspace based on the arm’s kinematics and joint limitations.

B. Cost Function

The cost function is designed to minimize the path length, maximize path smoothness and maximize the distance between the manipulator and the obstacles around it. The cost of traversing any transition between states \( s \) and \( s' \) in graph \( G \) can therefore be represented as:

\[
\begin{align*}
    c(s, s') &= c_{cell}(s') + w_{action} \cdot c_{action}(s, s') + w_{smooth} \cdot c_{smooth}(s, s').
\end{align*}
\]

The term \( c_{action} \) is used to assign a fixed cost for each motion primitives. In our experiments, we assigned uniform \( c_{action} \) costs. In other domains however, one may assign different motion primitive costs based on how motion primitives are easy to follow for example. The cost terms are weighted with \( w_1 \) and \( w_2 \). The weights can be chosen by a user to govern the amount of smoothness desired. A larger weight results in smoother paths at the expense of the trajectory length.

The term \( c_{smooth}(s, s') \) was used to penalize the non-smooth trajectories. One way to implement this cost would be to add the velocity of each joint to each state in the graph \( G \). Then, for a 7 DoF manipulator, instead of planning in a 7 dimensional state-space, \((\theta_1, \theta_2, ..., \theta_7)\), we would have to plan in a 14-dimensional state-space, that is, each state would be determined by the motion primitive index.

As described in the previous section, our solution to this is to augment each state with a single variable \( m \) which represents the index of the motion primitives that connects the previous state with the current. This is possible because of our use of a fixed motion primitive set. The smoothness cost, \( c_{smooth}(s, s') \), is a cost applied to the change in velocities between states \( s \) and \( s' \). The magnitude of the change in velocities can be represented by:

\[
\sum_{i=0}^{n} (s(v_i) - s'(v_i))^2
\]

where \( v_i \) are the joint velocities. The joint velocities of a state are determined by the motion primitive index.

C. Heuristic

The purpose of a heuristic function is to improve the efficiency of the search by guiding it in promising directions. A common approach for constructing a heuristic is to use the results from a simplified search problem (e.g. from a lower-dimensional search problem where some of the original constraints have been relaxed). For a heuristic function to be most informative, it must capture the key complexities associated with the overall search, such as mechanism constraints or the environment complexities. If a map of the planning environment is available at the start of the search, it is beneficial to create a heuristic lookup table that can be computed offline. Heuristic-based search algorithms require that the heuristic function, \( h \), be admissible and consistent. This is true when \( h(s_{start}) = 0 \) and for every pair of states \( s, s' \) such that \( s' \) is an end state of a single action executed at state \( s \), \( h(s) + c(s, s') \geq h(s') \), where \( h(s) \) is a heuristic of state \( s \), \( s_{start} \) is a state that corresponds to the end effector position and \( c(s, s') \) is the cost of the action that connects \( s \) to \( s' \). The cost \( c(s, s') \) of the action needs to be chosen carefully because an action cost that overestimates the true costs would result in an inadmissible heuristic. The planner that we are proposing is unique in that it performs a graph search in a state-space defined by joint angle configurations while the goal pose is defined by the position and orientation of the end effector. Therefore, it is necessary to use a heuristic function that is informative about the end effector position and orientation, \((x, y, z, r, p, y)\). Therefore, the heuristic of a state \( s \) can be represented by:

\[
\begin{align*}
    h(s) &= h_{xyz}(s) + w \cdot h_{rpy}(s).
\end{align*}
\]

1) \( h_{xyz} \): The ability to plan robustly in cluttered environments is the primary motivation of this research, and so a heuristic function that efficiently circumvents obstacles is necessary. Simplifying a planning problem by removing its complexity and removing some dimensionality is a standard technique in creating an informative heuristic function. In the same vein, we use a 3D Dijkstra search to find the shortest path from the end effector position \((x, y, z)\) state \( s \) to \( s_{goal} \). The end effector is represented in the 3D Dijkstra search as a single voxel on the twenty-six connected grid. The path cost computed proves to be an informative heuristic in directing the graph search around obstacles in a cluttered workspace.

2) \( h_{rpy} \): In order to achieve the proper end effector orientation at the goal pose, we use a heuristic function that computes a cost representing the difference in orientation to the goal pose. An effective parameterization for describing the difference between the end effector orientation of state \( s \) and the orientation of \( s_{goal} \) is through the axis-angle representation of a rotation [14]. Thus, \( h_{rpy}(s) \) is equal to the angle of rotation about a fixed axis specified by the axis-angle representation of the rotation between the end effector orientation of state \( s \) and \( s_{goal} \). A more efficient search is achieved when \( h_{rpy} \) is included in the heuristic function during the entire search when planning in an empty workspace. However, in a cluttered environment it is first necessary to compute a path to the goal region before
constraining the end effector orientation. Thus, we multiply a weight, $w$, to $h_{rpy}$ that is inversely related to the feasible distance to the goal. The feasible distance to the goal is effectively the path cost determined by the Dijkstra search. Therefore, the orientation constraint is relaxed on the end effector to allow for the search to circumvent any obstacles and approach $s_{goal}(x, y, z)$. As the search gets closer to $s_{goal}(x, y, z)$, $h_{rpy}$ is factored into the heuristic function accordingly.

D. Search

Any standard graph search algorithm can be used to traverse the graph $G$ that we construct. Given its size, however, optimal graph search algorithms such as A* [6] are infeasible to use. Instead we employ an anytime version of A* - - Anytime Repairing A* (ARA*) [12]. This algorithm generates an initial, possibly suboptimal solution quickly and then concentrates on improving this solution while deliberation time allows. The algorithm guarantees completeness for a given graph $G$ and provides a bound $\epsilon$ on the suboptimality of the solution at any point of time during the search. ARA* speeds up the typical A* search by inflating the heuristic values by a desired inflation factor, $\epsilon$. This is effective at rushing the graph traversal towards the goal state at the cost of solution optimality. An $\epsilon$ greater than 1.0 will produce a solution guaranteed to cost no more than $\epsilon$ times the cost of an optimal solution. If the path is found within the time allotted, then a followup search is executed with a lower $\epsilon$ weight applied to the heuristic. The search is continuously repeated while decrementing the epsilon with every iteration, until either the search time is up or $\epsilon$ has reached a value of 1. In doing so, ARA* gains an additional efficiency by not re-computing the states that it already computed in its previous search iterations.

IV. EXTENSIONS & OPTIMIZATIONS

A. Features

1) Path Constraints: Many motion planning tasks not only require that the end effector ultimately finds its way to the goal pose, but also require that the manipulator adheres to certain constraints along the way. A common example of a planning task that requires path constraints is picking up a glass filled with liquid and putting it elsewhere. Manipulating a filled glass requires that the end effector remains upright throughout the whole manipulation so as to not spill the glass’s contents. Path constraints such as an upright end effector (roll & pitch angles of zero) can be taken into consideration when planning with this planner. Path constraints can be expressed as a bounds on the position or orientation of a specific joint or link. Path constraints can also be defined as a desired joint position of a specific joint.

All path constraints are implemented as validity checks on all expanded nodes. During the expansion of a node, if one of its successor nodes does not meet the specified constraints, then it is considered invalid and not placed in the open list. It is common that path constraints can provide a decent speedup to the planner by effectively shrinking the state space.

2) Multiple Goals: Another feature of the motion planner we are presenting is the ability to handle multiple goal poses as input and return a path to the goal pose with the lowest path cost overall. This can prove useful when a grasp planner finds multiple feasible grasp poses for manipulating an object. The motion planner will compute a path to the grasping pose with the least cost path.

B. Additional Path Smoothing

In addition to the smoothing cost mentioned above, where we apply a cost to change joint velocities along graph edges, the planned path is passed through a short circuit smoothing function [3]. The short circuit smoothing function iterates through the waypoints in the path and tries to connect each waypoint to the furthest waypoint in the path in which there is a direct collision free path between them. If a direct path exists, then the intermediate waypoints are removed from the path and the algorithm continues with the next waypoint.

C. Optimizations

The most common problem that arises when applying search based algorithms to high dimensional planning problems is sluggish performance which leads to the inability to plan in realtime. Aware of that fact, we set out to exploit all of the provided knowledge of the system to precompute as much as possible. As mentioned earlier, having a complete map of the environment allows the planner to precompute the heuristic, essentially providing a lookup table to the planner for computing the cost of a successor to a node. The algorithm also precomputes a smoothing cost table that stores the cost of traversing from one action to another. Another time consuming task that is coupled with planning for complex high dimensional systems results from the processing required to compute the forward kinematics for the system. In addition to computing the end effector position in the
environment, the forward kinematics are required to compute the complete robot state if a mobile platform is used. The robot state is needed to perform collision avoidance with the world and with the robot itself. The forward kinematics are computed for all of the successors of an expanded node. The positions of the joints and orientation of the end effector are solved once and stored in the hash table for future access. Our implementation uses the DH convention for forward kinematics calculations. The DH convention is a common method for solving for the transformation between prismatic and/or revolute joints with just four parameters that describe the geometric relationship between them. Of the four parameters, three are fixed for a given link, which leaves, theta, the joint angle, as the only variable in the transformation. A large performance gain was achieved in the algorithm by using the known relationships between the joints of the system to precompute all of the transformation matrices between each joint with 1 degree resolution. The time it takes to create the transformation table is under one second and 360 transformation matrices for 8 coordinate frames requires only 740 KB. The lookup table is used to fetch the frame to frame transformation matrices which are then multiplied together to get the position and orientation of the end effector and the other joints in the global frame. During the search, computing the forward kinematics of a given joint configuration is performed in under 2.5 ms.

In our experimentation, we planned on a voxel grid with one centimeter resolution because we were limited by the resolution of the collision map received from the laser scanner. This accuracy was adequate when planning to objects during tabletop manipulation. To accelerate our collision checking function, we used an occupancy grid with 2 cm resolution to check if a successor state was valid. Thus, two occupancy grids were maintained at all times, allowing the planner to use the higher resolution grid to check if the search has reached the goal, and the lower resolution grid for collision checking. Once the planner receives the goal state from the user, the planner precomputes the cost from the start position(s) to every voxel in the grid. Thus, every time the heuristic function needs to compute Dijkstra’s shortest path cost to a voxel, a mere table lookup is all that is necessary. In addition to precomputing the breadth first search on the entire grid, voxel, a mere table lookup is all that is necessary. In addition to precomputing the breadth first search on the entire grid, voxel, a mere table lookup is all that is necessary. In addition to precomputing the breadth first search on the entire grid, voxel, a mere table lookup is all that is necessary.

In our experimentation, we planned on a voxel grid with one centimeter resolution because we were limited by the resolution of the collision map received from the laser scanner. This accuracy was adequate when planning to objects during tabletop manipulation. To accelerate our collision checking function, we used an occupancy grid with 2 cm resolution to check if a successor state was valid. Thus, two occupancy grids were maintained at all times, allowing the planner to use the higher resolution grid to check if the search has reached the goal, and the lower resolution grid for collision checking.

Once the planner receives the goal state from the user, the planner precomputes the cost from the start position(s) to every voxel in the grid. Thus, every time the heuristic function needs to compute Dijkstra’s shortest path cost to a voxel, a mere table lookup is all that is necessary. In addition to precomputing the breadth first search on the entire grid, we gain additional speedup by using a lower resolution grid for the heuristic computation. The lower resolution grid finished in a third of the time demanded by the larger grid used for planning. The precomputed Dijkstra heuristic may be less necessary in an open workspace when a direct path can be found with very few expansions, however the design of our planner is motivated by household cluttered environments in which good collision avoidance is necessary.

V. EXPERIMENTAL RESULTS

A. Simulation

The algorithm was tested in simulation using an open source simulator called Gazebo. Gazebo is a three dimensional simulator that can demonstrate the dynamics and limitations of the modeled manipulator using the Open Dynamics Engine. The robot model used in simulation is a fairly accurate representation of the PR2, the robot built by Willow Garage, that we eventually used for our experimentation. The simulated robot contains all of the sensors used by the actual robot. An open source software framework called ROS was used for inter-process communication. The PR2 is described in greater detail in the following section.

### Table I

<table>
<thead>
<tr>
<th></th>
<th>avg</th>
<th>std</th>
<th>ci</th>
</tr>
</thead>
<tbody>
<tr>
<td>expansions</td>
<td>16580.120</td>
<td>32957.950</td>
<td>5406.48</td>
</tr>
<tr>
<td>cost</td>
<td>85698.400</td>
<td>25966.624</td>
<td>3516.36</td>
</tr>
<tr>
<td>time</td>
<td>3.499</td>
<td>6.986</td>
<td>0.764</td>
</tr>
</tbody>
</table>

Simulation Results for randomly generated test environments. The planner was executed with \( \epsilon = 15 \). The confidence intervals are for 95% intervals.

Above are some preliminary results we computed. We generated 470 test environments with kinematically feasible goals. In each situation we randomly placed between 3 and 9 cubic obstacles in the workspace of the manipulator. The dimensions of the cubic obstacles range randomly between 10 cm and 20 cm. 323 of the environments, 68% computed valid solutions with 40 seconds. Of the remaining test cases, it is unknown whether the goals are occluded or not.

B. PR2 Experiments

Experiments were also conducted on a mobile manipulation platform called the PR2. The PR2 is a mobile robot designed for mobile manipulation, which makes it an appropriate test bed for our motion planner.

The robot has plenty of processing power on board, with two eight core computers available. For sensing it has six cameras as well as two Hokuyo laser scanners, one of which is mounted on a servo that provides a tilt scanning ability. In our experiments we decided to use the tilt laser scanner for perception of the environment. The tilt laser scanner provided the robot with a dense point cloud of the environment with 270 degrees of coverage around the sensor. Since manipulation generally occurs within one meter of the torso, the accuracy provided by the laser at that distance was quite good. The only drawback to using the tilt scanner is that a full scan of the environment takes around four seconds to complete a full sweep.

The PR2 has two arms with seven degrees of freedom each. However, the specific robot that we used for testing is an experimental model with just one arm. Each arm has a payload of two Kg and enough torque in the wrist to accomplish household chores. The seven joints in the arm are shoulder pan, shoulder lift, upper arm roll, elbow flex, forearm roll, wrist pitch and wrist roll. The gripper has an additional degree of freedom (a two-fingered claw) that we did not take into account during planning. The wrist roll and
forearm roll joints are continuous joints. The joint limitations of the arm are taken into account when planning. ROS was the software framework used for development, communication and interfacing to the hardware on the PR2. The software framework for the PR2 includes a controller library that provides an interface to a set of trajectory controllers for the different parts of the robot including the arms. The planner interfaces to the controllers using a ROS service call. The motion planner was implemented with an interface using a service that gets called when a motion plan for the arm is desired. The experiments conducted on the robot involved planning from different start positions to a set of goal positions in relatively cluttered environments. Snapshots of these experiments are shown in Figure 4.

VI. CONCLUSIONS

We have presented a heuristic search-based manipulation planner. We have shown that by using a motion primitive-based graph coupled with informative heuristics and anytime graph search, the planner can deal effectively with the high-dimensionality of the problem. In addition to its explicit cost minimization, the approach provides strong guarantees on completeness and suboptimality. Our experimental analysis on a real mobile manipulation platform with a 7-DOF robotic manipulator shows the ability of the planner to solve manipulation in cluttered spaces by generating consistent, low-cost motion trajectories while providing guarantees on completeness and bounds on suboptimality. In future, we intend to explore other, more sophisticated, approaches to generating and possibly learning online motion primitives. We would also like to develop other informative heuristic functions for the mobile manipulation tasks.

VII. ACKNOWLEDGMENTS

The authors gratefully acknowledge all of the support of everyone at Willow Garage.

REFERENCES