Published in the IEEE/RSJ International Conference on Intelligent Robots and Systems. :4621-4626 (2011)

On the Advantages of Task Motion Multigraphs for Efficient Mobile Manipulation

Ioan A. Şucan, Lydia E. Kavraki

Abstract— **This paper addresses the problem of computing the sequence of motion plans necessary for a mobile manipulator to execute a given task. In our previous work, we have demonstrated that computational advantages can be obtained when solving this problem by using the notion of a task motion multigraph (TMM). TMMs represent the state spaces that correspond to various hardware components of the robot, and they convey this information to the motion planning level. In this paper, we present and evaluate an algorithm that further exploits TMMs and explores multiple state spaces simultaneously. Since tasks to be performed by mobile manipulators often allow solutions that use only a subset of the robot's hardware components, motion plans can be found in lower dimensional state spaces. The resulting solutions tend to be shorter, more natural and faster to compute. We show that when planning under geometric constraints only, information gained while exploring lower dimensional spaces can be reused to obtain solutions in higher dimensional spaces, if necessary. The reuse of information implicitly provides the ability to compute decoupled motion plans. If solutions are not found while planning in a decoupled fashion, the algorithm resorts to planning in the robot's full state space. Our experiments indicate speedups of 200% and solutions up to four times shorter when compared to an analogous approach that does not employ TMMs.**

I. Introduction

In this paper, we focus our attention on robots capable of manipulating objects in human environments – robots that include at least one arm and a means of locomotion (e.g., tracks, wheels, legs). Robots such as these, typically referred to as mobile manipulators, are often complex, with many degrees of freedom (e.g., Honda Asimo, Willow Garage PR2). The complexity of mobile manipulators makes them versatile, capable of solving a variety of tasks. In fact, given a specific task, it is possible that a mobile manipulator can perform the task in a multitude of ways, depending on its choice of hardware components. For example, a mobile manipulator can open a door by simply extending its arm and pressing the door's handle. At the same time, it is possible for the robot to get closer to the door by moving its base and then pressing the handle with its arm. Furthermore, it is possible for the robot to use its arm and base simultaneously.

The development of a mobile manipulator poses many research problems. This paper discusses aspects related to planning the sequence of motions that a robot needs to perform in order to achieve its goal. We use the notion of a task motion multigraph (TMM), which we introduced in our previous work [1]. The key advantage that TMMs offer is that they represent both potential sequences of motion plans and the set of state spaces that could possibly be used for planning. The selection of the state space to plan in, which corresponds to the choice of hardware components to use, plays an important role in the computational effort of the motion planner: lower dimensional state spaces often lead to reduced computation times and more natural looking solutions. Previous work typically considered the full state space of the robot as the space to be searched for plans. TMMs make it possible to reason about the state spaces to possibly plan in. We previously showed a simple algorithm that achieved a speedup in the range of 20% by explicitly considering the planning options specified by the TMM [1].

In this paper, we present a new algorithm that relies on TMMs, but allows for significantly higher computational gains when planning under geometric constraints only. Speedup in the range of 200% and solutions that are on average four times shorter are observed. Intuitively, when motion planning is performed in the state space corresponding to some set of hardware components, we save computation time by reusing the information previously gained from the exploration of state spaces corresponding to subsets of the same hardware components.

II. Background and Related Work

A. Task and Motion Planning

It is typical for tasks to be represented as graphs, an example of which is shown in Figure 1-Left. This example encodes the task of transferring either a pen or a pencil to a specified destination. Even though this task is very simple, the robot must reason about the sequence of operations it needs to perform: reaching for either the pen or the pencil, closing the gripper, taking the object to its destination and then releasing it. Reasoning about the sequence of actions to be taken in order to achieve a goal is referred to as task planning. A task planner is an algorithm that generates directed acyclic graphs such as the one shown in Figure 1-Left, based on more concise representations of tasks (e.g., LTL [2], STRIPS-like [3]), and searches such graphs for solutions (task plans). The nodes in the task graph are associated with a set of robot states and the edges in the task graph correspond to primitive actions the robot can execute. Any leaf in the task graph is a goal for the robot. For simplicity, we assume that the only actions the robot can perform are `grip` (close gripper), `release` (open gripper) and `move_to` (plan a motion). This assumption is typical, as even with such a limited set of actions, it is possible to perform pick and place operations.

The `grip` and `release` actions are simple ones and do not include the computation of grasp poses. When grasp poses are necessary, we assume they are included in the task graph's nodes. The computation of such grasp poses is possible using grasp reasoning systems (e.g., [4]).

The move_to actions are referred to as *motion planning actions* because they require motion planning – the computation of a motion plan that connects two given robot states. In this work however, we are planning under geometric constraints only, and we assume that motion plans are represented as sequences of motion segments, and that each motion segment can be represented as a pair of robot states. It is a well known fact that when a sequence of motion plans needs to be computed as part of a solution to a task, it is possible that the way certain motion plans are computed influences the feasibility of subsequent motion plans. For this reason, a significant amount of work has been done towards the development of algorithms that compute task and motion plans simultaneously (e.g., [5]–[10]).

B. Previous Work

A generic description of a large body of previous work is that a task graph is automatically constructed based on some specification (e.g., LTL, STRIPS-like), and motion plans are computed with a sampling-based motion planner [11], [12]. Sampling-based motion planners are typically preferred due to their ability to quickly compute motion plans in high-dimensional spaces. In some of the previous work, the focus is on quickly computing task plans. For example, a hierarchical representation of tasks can be used to reduce the time taken to generate task graphs [9], or the robot can quickly commit to actions before having a complete plan [10]. In those works, a motion planner is used to implement primitive actions akin to our definition of move_to. In other works [7], [8], roadmaps [13] are used for motion planning, and task and motion planning are interleaved. The information gained from the exploration of the roadmaps is used at the task planning level to determine potentially feasible sequences of actions. Ideas related to interleaving motion planning with the computation of discrete plans have also been explored (e.g., [14]).

One idea related specifically to the improvement discussed in this paper is that of constructing a "relative roadmap" [8]. In that work, similar motion planning queries are solved by reusing exploration information at different locations in the environment and applying geometric transformations. A relative roadmap is constructed in a virtual environment that includes only an object to grasp. Multiple plans that achieve the grasp are computed and stored. When the particular object needs to be grasped, the previously computed plans are transformed to match the location of the object in the real environment and are then used to bootstrap the search for valid grasping plans. This is different from our work because we reuse information across spaces of different dimensionalities and we do not use geometric transformations.

C. Task Motion Multigraphs

In recent work [1] we introduced the concept of a task motion multigraph (TMM). This is a data structure that represents the possible sequences of motion planning actions (move_to, for the purpose of this paper), in a multigraph. In addition to the information in a task graph, a TMM also represents the set of possible state spaces in which motion plans could be computed. The set of available planning spaces corresponds to the possible combinations of hardware components, and hence, degrees of freedom (DOF), that the robot may use to perform its task.

Example: Figure 1-Right shows a TMM corresponding to the task graph in Figure 1-Left. Graph edges that correspond to grip and release actions are contracted and each graph edge that corresponds to a move_to action is replaced by a multiset of TMM edges, each TMM edge being labeled with the set of joints to be actuated along that edge. The removal of grip and release actions does not mean these actions are not to be performed, but only that they are ignored for the purpose of motion planning.

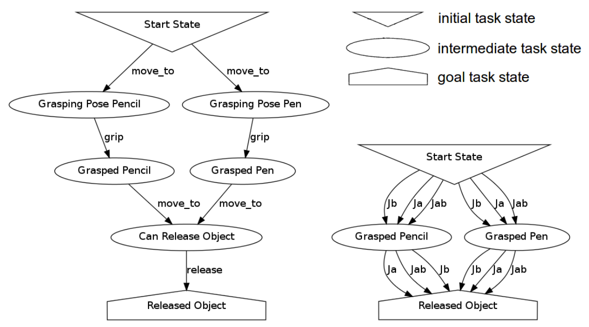


Fig. 1. Left: The task graph for retrieving either a pen or a pencil. Right: The task motion multigraph (TMM) – only actions that require motion planning are kept. For every action, multiple TMM edges are created, one for each set of joints potentially used for motion planning. Jb = joints in base, Ja = joints in arm, Jab = joints in arm and base.

Notation: Let the mobile manipulator consist of a set of joints J. The full state space of the robot, $\mathcal{X}_J$, is implicitly defined by J. Let $\mathbf{J} = \{J_1, J_2, \ldots, J_h | J_i \subseteq J, i = 1, \ldots, h\}$ contain the (not necessarily disjoint) user defined sets of joints that correspond to the hardware components we are interested in potentially planning motions for. In the general case, J_i may depend on what controllers are available. For example, for a mobile manipulator with two arms and an omni-directional base, the sets of joints $J_i, i \in \{1, 2, 3\}$ could correspond to the joints in the left arm, the joints in the right arm and the special SE(2) joint corresponding to the base, $J = \bigcup_i J_i$.

For convenience, we restate the following definition [1]:

Definition: A *task motion multigraph* (TMM) is a directed acyclic multigraph $G_M = (V_M, E_M)$ such that:

• $V_M = \{v | Q(v) \subset \mathcal{X}_J\}$ is a set of vertices. Every vertex v is associated with a set of robot states $Q(v) \subseteq \mathcal{X}_J$; $Q(v)$ can be explicitly specified as a set of states or implicitly specified in a manner that allows sampling.

Let $A = \{a = (v_i, v_j) | v_i \neq v_j,\ v_i, v_j \in V_M\}$ be the set of pairs of nodes between which motion planning actions exist. Let $label(a) = (Act, Env_a)$. For the purposes of this paper, Act is always move_to. At the start of the action, the robot is at a state $x \in Q(v_i)$ and at the end of the action, the robot is at a state $x' \in Q(v_j)$. Env_a defines the environment in which motion plans between v_i and v_j are to be computed.

• E_M is a multiset of edges representing all the motion

planning options between pairs of nodes $(v_i, v_j) \in A$,
$E_M = \bigcup_{a \in A} E_{M,a}$, where $E_{M,a}$ is a multiset. For each $a = (v_i, v_j) \in A$ and $label(a) = (Act, Env_a)$, $E_{M,a} = \{e_k = (v_i, v_j) \mid k \in \{1, \ldots, 2^{|\mathbf{J}|} - 1\}\}$ and $label(e_k) = (Act, Env_a, J_{e_k})$,
$J_{e_k} = \bigcup_{j \in \mathbf{b}(k)} j$, $\mathbf{b} : \{1, \ldots, 2^{|\mathbf{J}|} - 1\} \to 2^{\mathbf{J}} \setminus \{ \emptyset \}$ is a bijection, $2^{\mathbf{J}}$ is the power set of $\mathbf{J}$.

Intuitively, for every motion planning action, the TMM contains an edge for every state space that could possibly be used to plan for that action. For example, the TMM in Figure 1-Right conveys that for every action, it could be possible to plan in the state spaces that correspond to the base, the arm, or the Cartesian product of the base and the arm. In the worst case, the TMM will have $(2^h - 1) \cdot k$ edges, where k is the number of motion planning edges in the input task graph and h is the number of hardware components considered (the cardinality of $\mathbf{J}$). Given a particular robot, it is usually possible that with domain knowledge, some of the edges in the TMM can be pruned. For example, it is unlikely that planning a turn for the robot's head requires that the spaces corresponding to the robot's base or arms need to be considered individually. For completeness purposes, it is recommended that the full state space of the robot, $\mathcal{X}_J$, always be included as an option. For the remainder of the paper we will assume a TMM is available as input. For a discussion on how TMMs can be constructed, please see our previous work [1].

III. Multi-Space Exploration

In our previous work we showed that given a TMM, even a simple algorithm can lead to reduced computational effort when computing the sequence of motion plans necessary for a robot to perform its task [1]. The reduced computation came from the preference towards planning in lower dimensional spaces. In this paper, we show how to further leverage the information contained in the TMM to decrease computational effort even more. At the same time, we obtain shorter solutions. When planning in higher dimensional spaces is required to find solutions, information from the exploration of lower dimensional spaces is reused. Samples generated while exploring lower dimensional projections $\mathcal{X}_{J''}$ are lifted to the full state space of the robot and to higher dimensional projections $\mathcal{X}_{J'}$, $J'' \subseteq J'$. Thus, when reverting to planning in higher-dimensional spaces, the motion planner does not start from scratch. An additional feature of our approach is that it can implicitly perform decoupled planning [12], as discussed later.

Preliminaries

Let $G_M = (V_M, E_M)$ be the TMM given as input. For every edge $e = (v_a, v_b) \in E_M$, $label(e) = (Act, Env, J_e)$, we define the following operators:

- $M(e) = \{(v'_a, v'_b) \in E_M | v_a = v'_a, v_b = v'_b\}$, the multiset of edges that connect the same pair of nodes as e,
- $Joints(e) = J_e$, the set of joints e corresponds to,
- $Space(e) = \mathcal{X}_{J_e}$, the state space J_e implicitly defines.

Let $\mathcal{X}_J$ be the full state space of the robot. Given $x \in \mathcal{X}_J$ and $y \in \mathcal{X}_{J'}$, $J' \subset J$, we define $Lift(y, x) \in \mathcal{X}_J$ be the state that has the same values as y for the joints in J' and the same values as x for the joints in $J \backslash J'$. The inverse operation of $Lift()$ is $Project()$. For $x \in \mathcal{X}_J$, $Project(x, \mathcal{X}_{J'}) = y \in \mathcal{X}_{J'}$, where y has the same values as x for the joints in J' (y is an orthogonal projection of x).

A. Generic Planning with TMMs

The overall structure of our method is shown in Algorithm 1. This structure follows our previous work [1], and we repeat it here for convenience. Algorithm 1 proceeds iteratively until a solution is found or a maximum allowed time ($MaxT$) is exceeded. There are two main steps in the iteration. The first step [lines 2-4] aims to find motion plans for TMM edges that are closer to the goal (greedily selected). If no progress towards the goal is made, the second step [lines 5-7] is executed. The second step aims to find motion plans for TMM edges selected stochastically, so that probabilistic completeness can be achieved.

Algorithm 1 TMM-Computation($G_M = (V_M, E_M)$)

```
1: while $timeSpent < MaxT$ do
2:     $P \leftarrow$ ShortestPath($G_M$)
3:     $edge \leftarrow$ SelectEdgeFromPath($P$)
4:     ($edge'$, $sol$) $\leftarrow$ TMM-MotionPlan($edge$, $\Delta t$)
5:     if $sol$ = nil then
6:         $nextEdge \leftarrow$ SelectEdge($E_M \backslash M(edge)$)
7:         ($edge'$, $sol$) $\leftarrow$ TMM-MotionPlan($nextEdge$, $\Delta t$)
8:     if $sol \neq$ nil then
9:         RecordSolution($edge'$, $sol$)
10:    if HaveFullDimensionalSolution($G_M$) then
11:        return ExtractSolution($G_M$)
12: return nil
```

Algorithm 1 begins with the computation of the shortest path in the TMM [line 2]. The cost of a TMM edge e, $label(e) = (Act, Env_e, J_e)$ is

$$cost(e) = \exp\left(1 + \frac{dim(\mathcal{X}_{J_e})}{\max_J dim(\mathcal{X}_J)}\right) \cdot scost(e)$$

$$scost(e) = \begin{cases} 1 & \text{if } sol \\ s \cdot (1+t) \cdot \left(1 + \frac{d_L(e)}{d_R(e) + d_L(e)}\right) & \text{if not } sol, \end{cases}$$

where $dim(\cdot)$ is the dimension of a space, s represents the number of times e was selected for motion planning (starts at 1), t is the number of seconds already spent planning motions along e, $d_L(e)$ represents the number of TMM edges from e to the nearest leaf, $d_R(e)$ represents the number of TMM edges from e to the root, and sol is a flag indicating whether any motion plans have been found for e [1].

SelectEdgeFromPath() selects the edge that is closest to a leaf in the TMM and that has no motion plan associated to it [line 3]. A call to *TMM-MotionPlan()* [line 4] is then made, which replaces the direct call to a motion planner we had in our previous work, with a more complex implementation described later in Algorithm 2. If no solution is found, *TMM-MotionPlan()* is called again on an additional TMM edge returned by *SelectEdge()*. When *TMM-MotionPlan()* finds a solution, *RecordSolution()* is called. *TMM-MotionPlan()* may choose to switch the TMM edge it is planning for, so it also returns the edge it computed a solution for, when a solution is found. For a TMM edge $e = (v_a, v_b)$, *RecordSolution()* adds the reached robot state to a set of reached states $R(v_b) \subset Q(v_b)$. Initially $R(v) = \emptyset$ for all $v \in V_M$ except for the root: $R(root) = Q(root)$. For more details on Algorithm 1, please see our previous work [1].

B. Exploring Multiple Spaces Simultaneously

The core of our proposed improvement is in Algorithm 2. Given a TMM edge $edge = (v_a, v_b)$ and an amount of time Δt, Algorithm 2 computes a valid motion between v_a and v_b along a TMM edge in $M(edge)$ within the amount of time Δt, or terminates with failure. Our approach assumes the availability of a bi-directional motion planning algorithm. An instance of such an algorithm ($edge$.mp) and storage for its generated exploration information are associated to every edge in the TMM. Sampling-based planners would be typically used for $edge$.mp, but the only needs for an algorithm to be usable are that it must allow: *1)* access to the valid motion segments it generates in its exploration (*readNextValidMotionSegment()* function used in Algorithm 2), and *2)* a means of incorporating information about new valid motion segments that are computed externally (*addValidMotionSegment()* function used in Algorithm 2).

Algorithm 2 manages the exploration information generated by the motion planning instances associated to the TMM's edges. Significant computational gains can be obtained by sharing exploration information between the planning instances. The overall memory consumption of our approach is not affected negatively, as we will show later. Our approach requires essentially no changes to the underlying motion planner: the sharing of exploration information is managed completely by Algorithm 2.

Algorithm

The first time *TMM-MotionPlan()* is called for $edge$, input states are added for all edges in $M(edge)$ [lines 1-3 Algorithm 2, Algorithm 3]. *ActivatePlanner()* starts the motion planner corresponding to $edge$, and stops any other running motion planner instance, if one is active [line 4]. We further assume the motion planner is automatically deactivated when *TMM-MotionPlan()* terminates.

The body of Algorithm 2 is a three part iterative process. At every iteration, the *readNextValidMotionSegment()* function is called [line 5] to obtain a pair of states (x_p, x) that represent a new valid motion segment discovered by the motion planner in use. Information gained at each iteration is propagated to higher dimensional spaces in the first part of the algorithm. When solutions are found in lower dimensional spaces, part two decides whether to report a solution or to switch to planning in different state spaces. Part three switches to planning in higher dimensional spaces if slow progress is detected. More details on these parts follow.

1) Part one of Algorithm 2 [lines 7-13] shares information gained from the exploration of $Space(edge)$ with motion planners that could potentially be called for other edges in $M(edge)$. The goal is to reuse information between planning instances, so that if planning in higher dimensional spaces is needed, the planner does not start from scratch.

If the motion segment between states $x_p \in Space(edge)$ and $x \in Space(edge)$ is valid, an equivalent motion segment from $y_p \in Space(e')$ to $y \in Space(e')$ can be constructed for some edges $e' \in M(edge)$. The condition on edges e' is that $Joints(edge) \subset Joints(e')$. For example, for the TMM in Figure 1-Right, exploration in the space $\mathcal{X}_{J_{arm}}$ would lead to progress in the state space $\mathcal{X}_{J_{arm+base}}$ as well. To compute the equivalent motion segment, the states x_p and x first need to be lifted to the full state space of the robot, $\mathcal{X}_J$. This can always be done because a plan from the original input state to x_p and x is known. All joint values are known for the original input state, so states x_p and x can be lifted to $\mathcal{X}_J$ by filling in the missing joint values with the ones from the input states. The lifted states can then be projected to $Space(e')$, yielding a valid motion segment that can be added to the motion planner instance exploring $Space(e')$ [lines 10-13]. In the worst case scenario, there could be $2^{|\mathbf{J}|-1} - 1$ sets J' such that $Joints(edge) \subset J'$. While the number of state spaces to keep track of is exponential, the validity of a motion (collision checking) is evaluated only once. Furthermore, when the validity check is performed, all the robot parts need to be checked for collision, and the full robot state is actually already constructed. The process of lifting and projecting that state can be made very efficient.

2) Part two of Algorithm 2 [lines 14-24] handles the construction of solution plans and the possibility of decoupled planning. When a solution is found in $Space(edge)$, it is possible that not all of the joints for the input start and goal states are matched, since a plan was found only for a subset of the joints of the robot (e.g., a plan for the base only, will not ensure that the arm has moved to its correct state). In that case, planning is subsequently attempted for a subset of the unmatched set of joints.

The *FullDimensionalSolution()* routine checks if the obtained solution covers all the dimensions of $\mathcal{X}_J$ [line 16]. If $Space(edge) = \mathcal{X}_J$, *FullDimensionalSolution()* will return true. If an incomplete solution is found, more planning needs to be done, perhaps in a different state space. The *NextPlanningEdge()* routine decides which edge to switch to. A constraint at this point is that no edge is active more than once (mechanism implemented with the $edge$.used variable) to avoid infinite recursion. *NextPlanningEdge()* identifies a TMM edge e' whose corresponding joint values differ between the start and goal states, such that the dimension of $Space(e')$ is minimal. For example, if planning for the base, left arm and right arm, it may be necessary to switch to the space corresponding to the left arm after having succeeded at planning a motion for the base alone. This is because even though an SE(2) plan for the robot's base was found, the arm may not be at the desired state. In order to reuse the incomplete solution found while planning in $Space(edge)$, additional input states are added [line 22, 23]. Because the motion planner we use is bi-directional, a connection state $x_c \in Space(edge)$ along the solution exists such that x_c is connected to both a starting state and a goal state. The state x_c can be lifted to $\mathcal{X}_J$ in two ways, using either of the input states it is connected to [lines 20,21]. Although $x_c^s \neq x_c^g$, they do not differ for the joints in $Joints(edge)$. Subsequent planning in $Space(e')$ may quickly lead to a solution. This is in fact a form of decoupled planning [11], [12]. For example, a motion for the base could be planned first, and a motion for the arm could be planned subsequently. This approach may lead to faster computation of solutions, but it is not a complete approach.

Algorithm 2 TMM-MotionPlan($edge = (v_a, v_b)$, Δt)

```
1: if not AddedInputStates($edge$) then
2:     TMM-AddInputStates($edge$, $R(v_a)$, $Q(v_b)$)
3: ActivatePlanner($edge$.mp)
4: while $timeSpent < \Delta t$ do
5:     $(x_p, x) \leftarrow$ $edge$.mp.readNextValidMotionSegment()
Part 1: // share information between planning spaces
6:     if $x \neq$ nil then
7:         for $e' \in M(edge)$ do
8:             if $Joints(edge) \subset Joints(e')$ then
9:                 $spc \leftarrow Space(e')$
10:                $y_p \leftarrow Project(Lift(x_p, Root(x_p)), spc)$
11:                $y \leftarrow Project(Lift(x, Root(x)), spc)$
12:                $e'$.mp.addValidMotionSegment($y_p$, $y$)
Part 2: // report solution or plan for remaining dimensions
13:    if $edge$.mp.haveSolution() then
14:        $(sol, x_c) \leftarrow$ $edge$.mp.getSolution()
15:        if $FullDimensionalSolution(sol)$ then
16:            return $(edge, sol)$
17:        $edge$.used $\leftarrow$ True
18:        $e' \leftarrow$ NextPlanningEdge($M(edge)$)
19:        $x_c^s \leftarrow Lift(x_c, StartRoot(x_c))$
20:        $x_c^g \leftarrow Lift(x_c, GoalRoot(x_c))$
21:        TMM-AddInputStates($e'$, $\{x_c^s\}$, $\{x_c^g\}$)
22:        return TMM-MotionPlan($e'$, $\Delta t - timeSpent$)
Part 3: // if slow progress, switch to higher dimensional spaces
23:    if SlowProgress() and $Space(edge) \neq \mathcal{X}_J$ then
24:        for $e' \in M(edge)$ do
25:            if $Joints(edge) \subset Joints(e')$ then
26:                return TMM-MotionPlan($e'$, $\Delta t - timeSpent$)
27: return nil
```

3) Part three of Algorithm 2 ensures that *TMM-MotionPlan()* eventually degrades to simply calling the motion planner for $\mathcal{X}_J$ [lines 25-28]. If slow progress is detected, a switch is made to a strictly higher dimensional space that requires planning for a larger set of joints. The condition we use for detecting slow progress is that the distance between the set of states connected to starting states and the set of states connected to goal sates does not decrease for two thousand iterations. Due to this degradation policy, if the underlying motion planner is (probabilistically) complete, the same property is maintained for *TMM-MotionPlan()*.

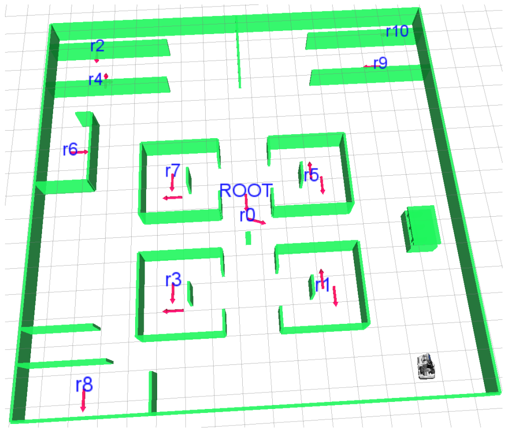
IV. Experiments

A. Experimental Setup

To test our proposed approach, we defined an office-like environment, show in Figure 2-Left. The task plan to be executed is shown as a task graph in Figure 2-Right. The robot we used for our simulations is the PR2 from Willow Garage. The robot components we defined were the left arm (7 DOF), the right arm (7 DOF) and the base (3 DOF). Thus, for every edge in the task graph, seven ($2^3 - 1$) corresponding edges were constructed in the TMM. This represents the worst case scenario, as not all edges would be necessary in practical applications. For example, the edge corresponding to the left and right arms is usually unnecessary. Furthermore, the regions marked in Figure 2-Left always differ in all the robot's components, so planning for all 17 joints is always necessary. This is an artificial example that makes the original version of our algorithm [1] always fail when attempting to find solutions in lower dimensional spaces. As a result, simply planning in the full 17 DOF state space is faster. We used RRT-Connect [15] as the underlying planner in Algorithm 2 because it is a simple and well established algorithm. However, other algorithms could be used as well. Planning is done under geometric constraints only and the implementation used is from OMPL [16].

Algorithm 3 TMM-AddInputStates($edge$, s, g)

```
1: for $e' \in M(edge)$ do
2:     $e'$.mp.addInputStates($Project(s, Space(e'))$,
                           $Project(g, Space(e'))$)
```



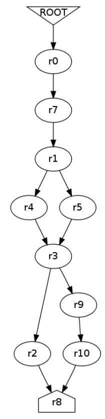


Fig. 2. Left: The environment in which the PR2 is operating. The robot is represented in the bottom right corner of the environment, for an impression of scale. The regions (rx) to be visited are marked by arrows indicating the robot's pose. Multiple poses per region are possible. Right: The sequence of motion plans to be computed (ROOT is start, r8 is the goal).

We compare our work to the approach that uses only a graph. We use Algorithm 1, but give it as input a graph in which edges correspond to the full state space of the robot. Furthermore, instead of of calling *TMM-MotionPlan()*, we call RRT-Connect directly. Although the implementation of our TMM-based approach is easily parallelizable, we use a single threaded implementation in our experiments.

B. Experimental Results

Figure 3 shows the results of computing a solution for the task described in Figure 2 using both our TMM approach with Algorithm 2 and the graph approach. The results are averaged over 30 runs. The success rate is 100% for both approaches. The only parameter we varied was Δt, the amount of time spent within a call to a *TMM-MotionPlan()*, and RRT-Connect, respectively.

As we can see, speedup of approximately 200% is consistently obtained (irrespective of Δt), as well as solutions that are four times shorter. The length of a solution is the sum of the lengths of its segments, and the length of a motion segment is the distance between its endpoints:

$$d(x, x') = d_2(P(x, J_{base}), P(x', J_{base})) \cdot 0.01 +$$
$$d_2(P(x, J_{left}), P(x', J_{left})) + d_2(P(x, J_{right}), P(x', J_{right})),$$

where $P(x, J_c)$ stands for $Project(x, \mathcal{X}_{J_c})$ and d_2 stands for the L2 norm. The factor 0.01 was used for the base to compensate for the size of the environment: 20m by 20m. The angles of the joints in the arms were measured in radians.

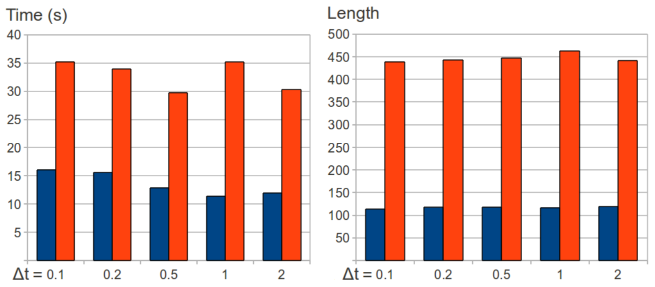


Fig. 3. Left: Runtime of our approach when using TMMs with Algorithm 2 (blue) versus the graph approach (red). Right: Length of solutions when using TMMs with Algorithm 2 (blue) versus the graph approach (red).

From a storage point of view, our approach is characterized by the memory needed for the stored robot states and the number of edges stored in the RRT-Connect exploration data structures. With respect to the approach that uses RRT-Connect directly, our approach requires only 78% of the memory for robot states and 120% of the number of edges in the exploration data structures (averaged over 10 runs). The reduced memory for the storage of states is a result of the reduced runtime of our approach. The total storage necessary for edges is significantly smaller than that for states, so the overall memory consumption of our approach is in fact reduced.

We also measured the amount of time spent collision checking, averaged over 10 runs. The computation of a task plan using the TMM approach with Algorithm 2 spends on average 89% of the time evaluating collisions, which shows that the overhead of managing the multi-space exploration is small (within the remaining 11%). The approach that uses RRT-Connect directly spends on average 79% of the time evaluating collisions. This percentage is smaller because planning in the full state space of the robot takes more time, more samples are generated by RRT-Connect, slowing down the nearest-neighbor computations. Furthermore, as more time is spent in the computation of a task plan, more shortest path computations need to be performed, as such computations occur every Δt seconds.

V. Conclusions and Future Work

We present an algorithm that uses task motion multigraphs (TMMs) to compute the sequence of motion plans necessary for a mobile manipulator to execute a given task. Our approach manages the exploration of multiple state spaces simultaneously by reusing information between edges in the TMM. When planning under geometric constraints only, experimental results indicate that our approach leads to a computational speedup of 200% and solutions that are four times shorter than those produced by an analogous approach that does not employ TMMs.

Although this version of the algorithm is a improvement over our previous work, further enhancements are possible. For example, the sharing of information between state spaces when considering more than just geometric constraints would require more sophisticated mechanisms, and in some cases it could be done in a lazy fashion, i.e., lift valid motions to higher dimensional spaces only when actually planning in those spaces.

Acknowledgements

Work on this paper by Ioan Şucan and Lydia Kavraki was supported in part by NSF IIS 0713623, NSF DUE 0920721, NSF CCF 1018798, a Sloan Fellowship to LK and Rice University funds. The experimental part of this work was supported in part by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc.

References

[1] I. A. Şucan and L. E. Kavraki, "Mobile manipulation: Encoding motion planning options using task motion multigraphs," in *Intl. Conf. on Robotics and Automation*, Shanghai, 2011, pp. 5492–5498.
[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, 2000.
[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 2003.
[4] A. Miller and P. K. Allen, "GraspIt!: A versatile simulator for robotic grasping," *IEEE Robotics and Automation Mag.*, vol. 11, no. 4, 2004.
[5] C. L. Nielsen and L. E. Kavraki, "A two level Fuzzy PRM for manipulation planning," in *Intl. Conf. on Intelligent Robots and Systems*, Takamatsu, Japan, 2000, pp. 1716–1722.
[6] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *Intl. Journal of Robotics Research*, vol. 23, pp. 729–746, 2004.
[7] K. Hauser and J.-C. Latombe, "Integrating task and PRM motion plannning," in *Intl. Conf. on Automated Planning and Scheduling*, 2009, Bridging the Gap between Task and Motion Planning Workshop.
[8] S. Cambon, R. Alami, and F. Gavrot, "A hybrid approach to intricate motion, manipulation and task planning," *Intl. Journal of Robotics Research*, vol. 28, pp. 104–126, 2009.
[9] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation," in *Intl. Conf. on Automated Planning and Scheduling*, 2010, pp. 254–258.
[10] L. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *Intl. Conf. on Robotics and Automation*. Anchorage: Workshop on Mobile Manipulation, 2010.
[11] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, June 2005.
[12] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.
[13] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.
[14] E. Plaku and G. D. Hager, "Sampling-based motion and symbolic action planning with geometric and differential constraints," in *Intl. Conf. on Robotics and Automation*, Anchorage, 2010, pp. 5002–5008.
[15] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Intl. Conf. on Robotics and Automation*, San Francisco, California, April 2000, pp. 995–1001.
[16] "The Open Motion Planning Library," http://ompl.kavrakilab.org.