E-Graphs: Bootstrapping Planning with Experience Graphs

Mike Phillips, Benjamin Cohen, Sachin Chitta, Maxim Likhachev

***Abstract*—Human environments possess a significant amount of underlying structure that is under-utilized in motion planning and mobile manipulation. In domestic environments for example, walls and shelves are static, large objects such as furniture and kitchen appliances most of the time do not move and do not change, and objects are typically placed on a limited number of support surfaces such as tables, countertops or shelves. Motion planning for robots operating in such environments should be able to exploit this structure to improve its performance with each execution of a task. In this paper, we develop an online motion planning approach which learns from its planning episodes (experiences) a graph, an *Experience Graph*. This graph represents the underlying connectivity of the space required for the execution of the mundane tasks performed by the robot. The planner uses the *Experience graph* to accelerate its planning efforts whenever possible and gracefully degenerates to planning from scratch if no previous planning experiences can be reused. On the theoretical side, we show that planning with Experience graphs is complete and provides bounds on suboptimality with respect to the graph that represents the original planning problem. On the experimental side, we show in simulations and on a physical robot that our approach is particularly suitable for higher-dimensional motion planning tasks such as planning for single-arm manipulation and two armed mobile manipulation. The approach provides significant speedups over planning from scratch and generates predictable motion plans: motions planned from start positions that are close to each other, to goal positions that are also close to each other, are similar. In addition, we show how the Experience graphs can incorporate solutions from other approaches such as human demonstrations, providing an easy way of bootstrapping motion planning for complex tasks.**

I. Introduction

Motion planning is essential for robots operating in dynamic human environments. Tasks like picking and placing objects and mobile manipulation of larger boxes require the robot to approach and pick up (or place) objects with minimal collision with the rest of the environment. Fast performance of motion planning algorithms in such tasks is critical, to account for the speed of operation expected by humans and to account for sudden changes in the environment. This is especially true of tasks involving higher-dimensional configuration spaces, e.g. for a two-armed mobile manipulation system (Figure 1).

At the same time, many of mundane manipulation tasks such as picking and placing various objects in a kitchen are highly repetitive. It is therefore expected that robots should be capable of learning and improving their performance with every execution of these repetitive tasks. In particular, robots should be capable of exploiting learned knowledge about the underlying geometric structure in tasks and human environments. While human environments can be very dynamic, e.g. with people walking around, large parts of the environment are still static for significant periods of time. Similarly, tasks tend to have some form of spatial structure, e.g. objects are often found on support surfaces like tables and desks.

Fig. 1. Motion planning is often used to compute motions for repetitive tasks such as dual-arm mobile manipulation in a kitchen.

This work focuses on learning from experience for motion planning. Our approach relies on a graph-search method for planning that builds an *Experience Graph* online to represent the high-level connectivity of the free space used for the encountered planning tasks. New motion planning requests reuse this graph as much as possible, accelerating the planning process significantly by eliminating the need for searching large portions of the search-space. While previously encountered motion planning problems can speed up the planner dramatically, it gracefully falls back to searching the original search-space, adding the newly generated motion to the Experience graph. Planning with Experience graphs is therefore complete. Furthermore, we show that it provides bounds on sub-optimality with respect to the graph that represents the original planning problem. Planning with Experience graphs leads to consistent and predictable solutions for motion plans requested in similar (but not the same) scenarios, e.g when the goal states of the robot are close to each other. Our approach is particularly useful when the tasks are somewhat repeatable spatially, e.g. in moving a set of dishes off a particular counter into a dishwasher. Although the start and goal states would be different for each motion plan, the general motion of moving a dish from the counter to the washer would essentially be the same each time.

We provide experimental results demonstrating the use of our approach both in simulation and on a real robot. A full-body planner for the PR2 robot was developed for dual-arm mobile manipulation tasks using a search-based planner. We show how the use of Experience graphs improves the performance of the planner in this high-dimensional state space. We also present results comparing our planner against a sampling-based planning approach for tabletop manipulation tasks with the PR2 robot. Finally, we show a preliminary application of Experience graphs to learning by demonstration.

II. Related Work

Initial approaches to motion planning focused on planning from scratch, i.e. there was no reuse of the information from previous plans. Recently, there has been more work on reuse of previous information for motion planning, especially in the context of performance optimization for motion planning in realtime dynamic environments. Lien et. al. [9] presented an approach that involved constructing roadmaps for obstacles, storing them in a database, and reusing them during motion planning. Bruce et. al. [3] extended the traditional RRT approach to reuse cached plans and bias the search towards waypoints from older plans. Extensions of this approach can be found in [17, 5].

In [12], an evolutionary algorithm approach was used to bias RRT search for replanning in dynamic environments towards the edges of the explored areas, intended to reduce the time spent on searching parts of the space that have already been explored. In [17], workspace probability distributions were automatically learned for certain classes of motion planning problems. The distributions attempted to capture a locally-optimal weighting of workspace features.

Trajectory libraries have seen use for adapting policies for new situations [15], especially for control of underactuated systems and high-dimensional systems. In [1], new trajectories in a state space were generated by combining nearby trajectories, including information about local value function estimates at waypoints. In [11], a trajectory library was used in combination with an optimal control method for generating a balance controller for a two link robot. Transfer of policies across tasks, where policies designed for a particular task were adapted and reused in a new task, were discussed in [16].

A learning based approach to reuse information from previous motion plans, the environment, and the types of obstacles was presented in Jetchev [6]. Here, a high-dimensional feature vector was used to capture information about the proximity of the robot to obstacles. After a dimensionality reduction in the feature space, several learning methods including nearest neighbor, locally weighted regression and clustering are used to predict a good path from the database in a new situation.

The work in [7] is most similar to our work and involves the use of a database of older motion plans. The approach uses a bi-directional RRT and tries to draw the search towards a path which is most similar to the new motion planning problem (based on distances to the start, goal and obstacles). The problem of culling a database of paths to find a set of paths that are most robust with respect to unknown obstacle configurations was treated in [2].

The use of a database of motion plans is a key feature of our approach. However, we differ from other approaches in that we attempt to use all the information from previous searches instead of attempting to pick the most similar or best path. Our approach can reuse parts of the Experience Graph even when the start and goal states change. Contrary to other approaches, our approach also gracefully degenerates to planning from scratch (with no reused information) to deal with new scenarios which might be completely different from the ones in the database. Our approach also does not rely on object or shape recognition and is thus agnostic to the representation of the environment, e.g. as a voxel grid or individual objects represented using meshes. Although our approach is also comparable to Probabilistic Roadmaps [8], a crucial difference is that Experience Graphs are generated from task-based requests instead of sampling the whole space. We are also able to provide a bound on the quality of the returned solution, which most sampling-based methods lack. Lastly, as we will demonstrate using an example, our approach can be bootstrapped by initial plans generated in different manners, e.g. using learning from demonstration.

III. Algorithm

A. Overview

An *Experience Graph* or E-Graph is a graph formed from the solutions found by the planner for previous planning queries or from demonstrations. We will abbreviate this graph as $G^{\mathcal{E}}$. The graph $G^{\mathcal{E}}$ is incomparably smaller than graph G used to represent the original planning problem. At the same time, it is representative of the connectivity of the space exercised by the previously found motions. The key idea of planning with $G^{\mathcal{E}}$ is therefore to bias the search efforts, using a specially constructed heuristic function, towards finding a way to get onto the graph $G^{\mathcal{E}}$ and to remain searching $G^{\mathcal{E}}$ rather than G as much as possible. This avoids exploring large portions of the original graph G. In the following we explain how to do this in a way that guarantees bounds on solution quality with respect to the original graph G.

B. Definitions and Assumptions

First we will list some definitions and notations that will help explain our algorithm. We assume the problem is represented as a graph where a start and goal state are provided (s_{start}, s_{goal}) and the desired output is a path (sequence of edges) that connect the start to the goal.

- $G(V^G, E^G)$ is a graph modeling the original motion planning problem, where V^G is the set of vertices and E^G is the set of edges connecting pairs of vertices in V^G.
- $G^{\mathcal{E}}(V^{\mathcal{E}}, E^{\mathcal{E}})$ is the E-Graph that our algorithm builds over time ($G^{\mathcal{E}} \subseteq G$).
- $c(u, v)$ is the cost of the edge from vertex u to vertex v
- $c^{\mathcal{E}}(u, v)$ is the cost of the edge from vertex u to vertex v in graph $G^{\mathcal{E}}$

Edge costs in the graph are allowed to change over time (including edges being removed and added which happens when new obstacles appear or old obstacles disappear). The more static the graph is, the more benefits our algorithm provides. The algorithm is based on heuristic search and is therefore assumed to take in a heuristic function $h^G(u, v)$ estimating the cost from u to v ($u, v \in V^G$). We assume h^G is admissible and consistent. An admissible heuristic never overestimates the minimum cost from any state to the goal. A

consistent heuristic is one that satisfies the triangle inequality, $h^G(s, s_{goal}) \leq c(s, s') + h^G(s', s_{goal})$.

C. Algorithm Detail

The planner maintains two graphs, G and $G^{\mathcal{E}}$. At the high-level, every time the planner receives a new planning request the *findPath* function is called. It first updates $G^{\mathcal{E}}$ to account for edge cost changes and perform some precomputations. Then it calls the *computePath* function, which produces a path π. This path is then added to $G^{\mathcal{E}}$. The *updateEGraph* function works by updating any edge costs in $G^{\mathcal{E}}$ that have changed. If any edges are invalid (e.g. they are now blocked by obstacles) they are put into a disabled list. Conversely, if an edge in the disabled list now has finite cost it is re-enabled. At this point, the graph $G^{\mathcal{E}}$ should only contain finite edges. A precomputeShortcuts function is then called which can be used to compute shortcut edges before the search begins. Ways to compute shortcuts are discussed in Section V. Finally, our heuristic $h^{\mathcal{E}}$, which encourages path reuse, is computed.

findPath(s_{start}, s_{goal})
1: $updateEGraph(s_{goal})$
2: $\pi = computePath(s_{start}, s_{goal})$
3: $G^{\mathcal{E}} = G^{\mathcal{E}} \cup \pi$

updateEGraph(s_{goal})
1: $updateChangedCosts()$
2: disable edges that are now invalid
3: re-enable disabled edges that are now valid
4: $precomputeShortcuts()$
5: compute heuristic $h^{\mathcal{E}}$ according to Equation 1

Our algorithm's speed-up comes from being able to reuse parts of old paths and avoid searching large portions of graph G. To accomplish this we introduce a heuristic which intelligently guides the search toward $G^{\mathcal{E}}$ when it looks like following parts of old paths will help the search get close to the goal. We define a new heuristic $h^{\mathcal{E}}$ in terms of the given heuristic h^G and edges in $G^{\mathcal{E}}$.

$$h^{\mathcal{E}}(s_0) = \min_{\pi} \sum_{i=0}^{N-1} min\{\varepsilon^{\mathcal{E}} h^G(s_i, s_{i+1}), c^{\mathcal{E}}(s_i, s_{i+1})\} \quad (1)$$

where π is a path $\langle s_0 \dots s_{N-1} \rangle$ and $s_{N-1} = s_{goal}$ and $\varepsilon^{\mathcal{E}}$ is a scalar ≥ 1.

Equation 1 is a minimization over a sequence of segments such that each segment is a pair of arbitrary states $s_a, s_b \in G$ and the cost of this segment is given by the minimum of two things: either $\varepsilon^{\mathcal{E}} h^G(s_a, s_b)$, the original heuristic inflated by $\varepsilon^{\mathcal{E}}$, or the cost of an actual least-cost path in $G^{\mathcal{E}}$, provided that $s_a, s_b \in G^{\mathcal{E}}$.

In Figure 2, the path π that minimizes the heuristic contains a sequence of alternating segments. In reality π can alternate between h^G and $G^{\mathcal{E}}$ segments as many or as few times as needed to produce the minimal π. When there is a $G^{\mathcal{E}}$ segment we can have many states s_i on the segment to connect two states, while when on a h^G segment a single pair of points suffices since no real edge between two states is needed for h^G to be defined.

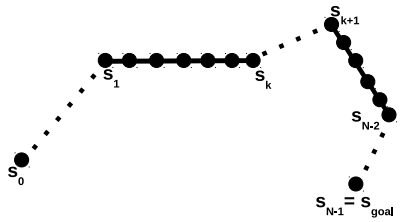
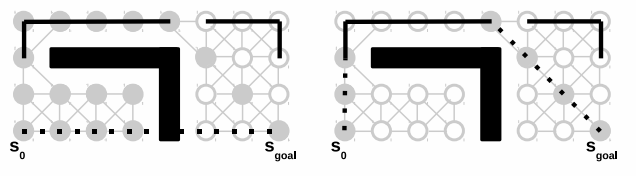
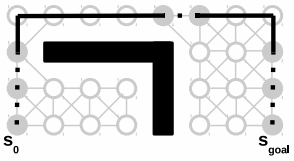


Fig. 2. A visualization of Equation 1. Solid lines are composed of edges from $G^{\mathcal{E}}$, while dashed lines are distances according to h^G. Note that h^G segments are always only two points long, while $G^{\mathcal{E}}$ segments can be an arbitrary number of points.



(a) $\varepsilon^{\mathcal{E}} = 1$ (b) $\varepsilon^{\mathcal{E}} = 2$



(c) $\varepsilon^{\mathcal{E}} \to \infty$

Fig. 3. Shortest π according to $h^{\mathcal{E}}$ as $\varepsilon^{\mathcal{E}}$ changes. The dark solid lines are paths in $G^{\mathcal{E}}$ while the dark dashed lines are the heuristic's path π. Note as $\varepsilon^{\mathcal{E}}$ increases, the heuristic prefers to travel on $G^{\mathcal{E}}$. The light gray circles and lines show the graph G and the filled in gray circles represent the expanded states under the guidance of the heuristic.

The larger $\varepsilon^{\mathcal{E}}$ is, the more we avoid exploring G and focus on traveling on paths in $G^{\mathcal{E}}$. Figure 3 demonstrates how this works. As $\varepsilon^{\mathcal{E}}$ increases, it becomes more expensive to travel off of $G^{\mathcal{E}}$ causing the heuristic to guide the search along parts of $G^{\mathcal{E}}$. In Figure 3a, the heuristic ignores the graph $G^{\mathcal{E}}$ because without inflating h^G at all, following real edges costs (like those in $G^{\mathcal{E}}$) will never be the cheaper option. In the other parts of Figure 3 we can see as $\varepsilon^{\mathcal{E}}$ increase, the heuristic uses more and more of $G^{\mathcal{E}}$. This figure also shows how during the search, by following old paths, we can avoid obstacles and have far fewer expansions. The expanded states are shown as filled in gray circles, which change based on how the $h^{\mathcal{E}}$ is biased by $\varepsilon^{\mathcal{E}}$.

The *computePath* function runs weighted A* without re-expansions [13, 10]. Weighted A* uses a parameter $\varepsilon^w > 1$ to inflate the heuristic used by A*. The solution cost is guaranteed to be no worse than ε^w times the cost of the optimal solution and in practice it runs dramatically faster than A*. The main modification to Weighted A*, is that in addition to using the edges that G already provides (getSuccessors), we add two additional types of successors: *shortcuts* and *snap motions*. The only other change is that instead of using the

heuristic h^G, we use our new heuristic $h^{\mathcal{E}}$.

computePath(s_{start}, s_{goal})
1: $OPEN = \emptyset$
2: $CLOSED = \emptyset$
3: $g(s_{start}) = 0$
4: $f(s_{start}) = \varepsilon^w h^{\mathcal{E}}(s_{start})$
5: insert s_{start} into $OPEN$ with $f(s_{start})$
6: **while** s_{goal} is not expanded **do**
7: remove s with the smallest f-value from $OPEN$
8: insert s in $CLOSED$
9: $S = getSuccessors(s) \cup shortcuts(s) \cup snap(s)$
10: **for all** $s' \in S$ **do**
11: **if** s' was not visited before **then**
12: $f(s') = g(s') = \infty$
13: **end if**
14: **if** $g(s') > g(s) + c(s, s')$ and $s' \notin CLOSED$ **then**
15: $g(s') = g(s) + c(s, s')$
16: $f(s') = g(s') + \varepsilon^w h^{\mathcal{E}}(s')$
17: insert s' into $OPEN$ with $f(s')$
18: **end if**
19: **end for**
20: **end while**

Shortcut successors are generated when expanding a state $s \in G^{\mathcal{E}}$. A shortcut successor uses $G^{\mathcal{E}}$ to jump to a place much closer to s_{goal} (closer according to the heuristic). This shortcut may use many edges from various previous paths. The shortcuts allow the planner to quickly get near the goal without having to re-generate paths it has produced before. Possible shortcuts are discussed in Section V.

Finally, for environments that can support it, we introduce *snap motions*. Sometimes, the heuristic may lead the search to a minimum at the "closest" point to $G^{\mathcal{E}}$ with respect to the heuristic, but it may not be a state on $G^{\mathcal{E}}$. For example, in (x, y, θ) navigation, a 2D (x, y) heuristic will create a minimum for 2 states with the same x,y but different θ. A problem then arises because there isn't a useful heuristic gradient to follow, and therefore, many states will be expanded blindly. We borrow the idea of *adaptive motion primitives* [4] to generate a new action which can snap to a state on $G^{\mathcal{E}}$ whenever states s_i, s_j have $h^G(s_i, s_j) = 0$ and $s_j \in G^{\mathcal{E}}$ and $s_i \notin G^{\mathcal{E}}$. The action is only used if it is valid with respect to the current planning problem (e.g. doesn't collide with obstacles). As with any other action, it has a cost that is taken into account during the search.

IV. Theoretical Analysis

Our planner provides a guarantee on completeness with respect to G (the original graph representation of the planning problem).

Theorem 1. *For a finite graph G, our planner terminates and finds a path in G that connects the s_{start} and s_{goal} if one exists.*

Since no edges are removed from the graph (we only add) and we are searching the graph with Weighted A* (a complete planner), if a solution exists on the original graph, our algorithm will find it.

Our planner provides a bound on the sub-optimality of the solution cost. The proof for this bound depends on our heuristic function $h^{\mathcal{E}}$ being $\varepsilon^{\mathcal{E}}$-consistent.

Lemma 1. *If the original heuristic function h^G is admissible, then the heuristic function $h^{\mathcal{E}}$ is $\varepsilon^{\mathcal{E}}$-consistent.*

From Equation 1

$$h^{\mathcal{E}}(s) \leq min\{\varepsilon^{\mathcal{E}} h^G(s, s'), c^{\mathcal{E}}(s, s')\} + h^{\mathcal{E}}(s')$$

$$h^{\mathcal{E}}(s) \leq \varepsilon^{\mathcal{E}} h^G(s, s') + h^{\mathcal{E}}(s')$$

$$h^{\mathcal{E}}(s) \leq \varepsilon^{\mathcal{E}} c(s, s') + h^{\mathcal{E}}(s')$$

The last step follows from h^G being admissible. Therefore, $h^{\mathcal{E}}$ is $\varepsilon^{\mathcal{E}}$-consistent.

Theorem 2. *For a finite graph G, the planner terminates, and the solution it returns is guaranteed to be no worse than $\varepsilon^w \cdot \varepsilon^{\mathcal{E}}$ times the optimal solution cost in graph G.*

Consider $h'(s) = h^{\mathcal{E}}(s)/\varepsilon^{\mathcal{E}}$. $h'(s)$ is clearly consistent. Then, $\varepsilon^w h^{\mathcal{E}}(s) = \varepsilon^w \cdot \varepsilon^{\mathcal{E}} h'(s)$. The proof that $\varepsilon^w \cdot \varepsilon^{\mathcal{E}} h'(s)$ leads to Weighted A* (without re-expansions) returning paths bounded by $\varepsilon^w \cdot \varepsilon^{\mathcal{E}}$ times the optimal solution cost follows from [10].

V. Implementation Detail

In this section we discuss how various parts of the algorithm could be implemented.

A. Heuristic

Some heuristics h^G are derived using dynamic programming in a lower dimensional state space, such as a 2D Dijkstra search for an (x, y, θ) navigation problem. Alternatively, some heuristics h^G can be computed in O(1) upon request (e.g. euclidean distance).

In the first case, we can compute $h^{\mathcal{E}}$ by running a Dijkstra search on the low-dimensional projection of G, with additional edges from $G^{\mathcal{E}}$ connecting the low-dimensional projection of the states in $G^{\mathcal{E}}$. This can be computed with similar efficiency to the original heuristic h^G, so it doesn't hurt the planning times (this is what we used in our experiments).

In the second case, the $h^{\mathcal{E}}$ can be computed by constructing a fully connected graph in the state space of G with h^G edges between them all as well as the edges from $G^{\mathcal{E}}$ and then running Dijkstra's algorithm on it.

In our implementation, we compute the heuristic $h^{\mathcal{E}}$ in an on-demand fashion. Our computation runs Dijkstra's algorithm just up until the heuristic of the requested state is computed and then suspends until another un-computed heuristic is requested.

B. Shortcuts

Shortcuts accelerate the search by allowing the search to bypass retracing an old path (re-expanding the states on it) in $G^{\mathcal{E}}$. The algorithm works with or without the shortcuts. Basically, the shortcuts are pre-computed edges that connect all states in $G^{\mathcal{E}}$ to a very small set of states in $G^{\mathcal{E}}$. Shortcut

successors can only be generated when expanding a state $s \in G^{\mathcal{E}}$. There are several obvious choices for choosing this subset. For example, it can contain all states s in $G^{\mathcal{E}}$ that are closest to the goal within each connected component of $G^{\mathcal{E}}$. The closeness can be defined by h^G or $h^{\mathcal{E}}$. In our experiments we used h^G. Other ways can also be used to compute this subset of states. It is future work to explore these options.

C. Pre-Computations

Some of the computations on $G^{\mathcal{E}}$ can be done before the goal is known. In particular, there are several places in our algorithm where we need to know the costs of least-cost paths between a pair of states in $G^{\mathcal{E}}$. One example is $shortcuts(s)$ (line 9 of *computePath*). If state $u \in G^{\mathcal{E}}$ is being expanded and has a shortcut to the state v on the same component in $G^{\mathcal{E}}$ then we need to assign an edge cost $c(u, v)$. In order to do that we need to know the cost of a least-cost path on $G^{\mathcal{E}}$ from u to v. These costs can be computed before knowing the goal by using an all-pairs shortest path algorithm like Floyd-Warshall. This can be done in a separate thread between planning queries (as well as adding the path from the previous query into $G^{\mathcal{E}}$). To make Floyd-Warshall run faster and to save memory, we can also exploit the fact that most of the paths in $G^{\mathcal{E}}$ don't intersect each other in many places. We can therefore compress it into a much smaller graph containing only vertices of degree $\neq 2$ and run Floyd-Warshall on it. Then, the cost of a path between any pair $x, y \in G^{\mathcal{E}}$ is given by $\min_{x_i, y_i}\{c(x, x_i)+c(x_i, y_i)+c(y_i, y)\}$. Where $x_i \in \{x_1, x_2\}$ and $y_i \in \{y_1, y_2\}$. x_1 and x_2 are states with degree $\neq 2$ that contain the path on which x resides. y_1 and y_2 are defined similarly.

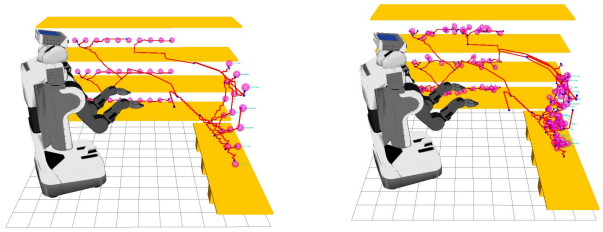
VI. Experimental Results

A variety of experiments were run in multiple domains to verify the performance gains produced by our algorithm. We compared our approach against Weighted A* without using $G^{\mathcal{E}}$ as well as with a randomized planner [14]. The domains in which this approach was tested includes planning for a 7 degree of freedom arm as well as full-body planning for the PR2 robot (two arms, the telescoping spine and the navigation motion of the base).

A. Full Body Planning

Planning in higher-dimensional spaces can be challenging for search-based planners, e.g. full-body planning with a robot like the PR2 can involve up to 18 degrees of freedom. A full-body planning scenario is thus a good test of the capabilities developed in this work. Our test scenario involves the PR2 carrying objects in a large environment. We restrict the objects to be upright in orientation, a constraint that often presents itself in real-world tasks like carrying large objects, trays, or liquid containers. We assume that the two end-effectors are rigidly attached to the object. The state space is 10 dimensional:

$$\langle x_e, y_e, z_e, \theta_e, \phi_L, \phi_R, x_b, y_b, \theta_b, z_s \rangle$$



(a) Bootstrap goals (b) One of the test sets

Fig. 4. Full-body planning in a warehouse

Here x_e, y_e, z_e, θ_e define the position and yaw of the object in a frame attached to the robot. The planner operates directly in this workspace (instead of joint space) and uses inverse kinematics to map plans or paths back onto the joint angles of both arms. ϕ represents the redundant degree of freedom in each arm. In the PR2, choosing this redundant degree of freedom as the upper arm roll joint angle restricts the number of valid inverse kinematics solutions for a given ϕ to one. x_b, y_b, θ_b define the robot's position and yaw in the map frame and z_s represents the extension of the telescoping spine.

A 3D Dijkstra heuristic is used to plan (backwards) for a sphere inscribed in the carried object from its goal position to the start position (this is the low-dimensional projection for the heuristic). The heuristic is useful in that it accounts for collisions between the object and obstacles. However, it does not handle the complex kinematic constraints on the motion of the object due to the arms, spine,and base and does not account for collisions between the body of the robot and the environment. In all experiments, $\varepsilon^w = 2$ and $\varepsilon^{\mathcal{E}} = 10$ resulting in a sub-optimality bound of 20. We chose these values for the parameters (manually) as they provided a good combination of speed-up and sub-optimality bound. In future work, we will look into ways to automatically reduce the sub-optimality bound as planning time allows. Setting $\varepsilon^{\mathcal{E}}$ to 10 greatly encourages the search to go to $G^{\mathcal{E}}$ if it looks like following it can get it closer to the goal. Setting ε^w to 2 inflates the whole heuristic including the part of the $h^{\mathcal{E}}$ using paths in $G^{\mathcal{E}}$. This encourages the planner to use shortcuts as soon as they become available, preventing it from re-expanding an old path. The results were compared against regular Weighted A* with $\varepsilon^w = 20$ so that both approaches would have the same sub-optimality bound.

1) Warehouse scenario: Our first scenario is modeled on an industrial warehouses where the robot must pick up objects off a pallet and move them onto a shelving unit (Figure 4). The goals alternate between pallet and shelves. Since our planner improves in performance with repeated attempts in a particular spatial region of space, we first bootstrap it with 45 uniformly distributed goals (split between the pallet and the shelves). The bootstrap goals and the resultant $G^{\mathcal{E}}$ after processing them are shown in Figure 4a.

A set of 100 random goals (with varying positions and yaws of the object) alternating between the pallet and the shelves

TABLE I
E-GRAPHS ON WAREHOUSE ENVIRONMENT (100 GOALS PER SET)

Set	mean time(s)	std dev time(s)	mean expands	mean cost
1	1.08	1.77	103	7933
2	1.26	3.18	150	7806
3	1.53	4.58	178	7804
4	1.80	4.70	221	7775
5	1.16	1.97	142	7351

TABLE II
WEIGHTED A* ON WAREHOUSE ENVIRONMENT (100 GOALS PER SET)

Set	mean time(s)	std dev time(s)	mean expands	mean cost
1	12.12	35.11	1883	4589
2	8.22	23.14	1211	4321
3	134.12	806.11	21792	4590
4	14.25	70.21	2527	4539
5	9.59	37.67	1495	4221

were then specified to the planner. This entire process was repeated 5 times with $G^{\mathcal{E}}$ cleared before running each new set of 100 goals. The statistics from 5 different sets of 100 random goals are shown in Table I. On average, 94% of the edges on a path produced by our planner were recycled from $G^{\mathcal{E}}$. The mean time to update $G^{\mathcal{E}}$ (add the new path and compute Floyd-Warshall) was 0.74 seconds.

Statistics for the same 5 sets using Weighted A* (with no learning) are shown in Table II. The data indicates that the average planning time for our approach tends to be an order of magnitude smaller, and the standard deviation is significantly smaller as well. $G^{\mathcal{E}}$ tends to have a vertex near a new goal. Therefore, planning time is mostly dominated by having to plan for matching the yaw of the object at the goal, which the heuristic provides no information about. It should be noted that in Set 3, there was one very difficult goal that the Weighted A* approach struggled to solve. The number of expansions in these tables reflects the trend in planning times. Finally, on average, the path cost from our approach tends to be higher than from the Weighted A* approach, but no more than twice as expensive.

Table III compares the two approaches further by examining the ratio between planning times for Weighted A* and the E-Graph approach. The data indicates that E-Graph planning times are almost 20 times faster (except in Set 3). The expansion ratio is higher than the planning time ratio, i.e. more time per expand is spent in the E-Graph approach, since it has to compute shortcut and snap successors.

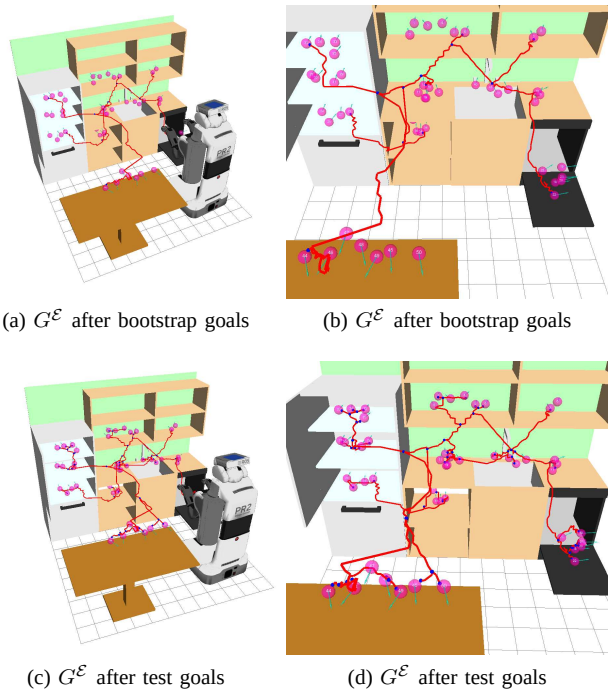
Table IV compares the 10% hardest goals (the ones that took the Weighted A* the longest to solve) showing that our approach is over 100 times faster on such difficult goals.

TABLE III
WEIGHTED A* TO E-GRAPH RATIOS ON WAREHOUSE ENVIRONMENT (100 GOALS PER SET)

Set	mean time(s)	std dev time(s)	mean expands	mean cost
1	18.40	39.94	206	0.63
2	17.74	51.24	231	0.60
3	224.83	1674.12	1562	0.63
4	24.30	155.42	142	0.64
5	16.69	72.41	193	0.62

TABLE IV
10% MOST DIFFICULT CASES ON WAREHOUSE ENVIRONMENT (10 GOALS PER SET)

Set	1	2	3	4	5
Mean Time Ratio	122.68	128.12	2157.14	200.80	112.17



(a) $G^{\mathcal{E}}$ after bootstrap goals (b) $G^{\mathcal{E}}$ after bootstrap goals

(c) $G^{\mathcal{E}}$ after test goals (d) $G^{\mathcal{E}}$ after test goals

Fig. 5. Full-body planning in a kitchen scenario

2) Kitchen Environment: A second set of tests was run in a simulated kitchen environment. 50 goals were chosen in locations where object are often found (e.g. tables, countertops, cabinets, refrigerator, dishwasher). 10 representative goals were chosen to bootstrap our planner, which was then tested against the remaining 40 goals. Figure 5 shows $G^{\mathcal{E}}$ both after bootstrapping and after all the goals have been processed.

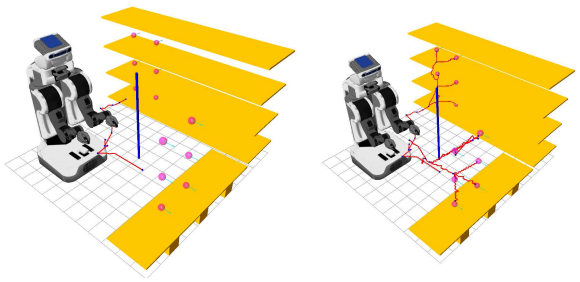
Table V shows the results of our experiments. The average planning time for our approach is significantly lower than the Weighted A* approach and again, has a lower standard deviation. This is also reflected in the average number of expansions. Our planner also provides a speed-up of about 20.0. The paths from our approach are only a little longer than that those in the Weighted A*. On average, 82% of the edges on a path produced by our planner were recycled from $G^{\mathcal{E}}$. The mean time to update $G^{\mathcal{E}}$ (add the new path and compute Floyd-Warshall) was 0.35 seconds. Finally, the average Weighted A* to E-Graph time ratio on the 10% hardest cases was 84.53.

B. Learning by Demonstration

The high dimensionality of the full-body domain makes it very easy for the planner to get stuck in large local minima. This is especially true when the heuristic is misleading, such

TABLE V
RESULTS ON KITCHEN ENVIRONMENT (40 GOALS)

	mean time(s)	std dev time(s)	mean expands	mean cost
E-Graphs (E)	2.12	6.55	351	5646
Weighted A* (W)	11.54	20.74	2639	4236
Ratio (W/E)	22.29	38.09	357	0.79



(a) A demonstrated path (b) $G^{\mathcal{E}}$ after 12 goals

Fig. 6. Learning by demonstration in a more difficult warehouse scenario

TABLE VI
RESULTS ON LEARNING BY DEMONSTRATION (12 GOALS)

	mean time(s)	std dev time(s)	mean expands	mean cost
E-Graph	1.08	0.63	40	6465

as when the heuristic implies that the carried object can pass between 2 objects, but the robot's body won't fit throught this space. Figure 6a shows an example scenario where, after picking up an object at the pallet, the goal is to move it to a shelf. The heuristic tries to guide the search around the right side of the pole, but the robot's body can't fit through and therefore, the search will take incredibly long. Without a path getting past the pole, our algorithm will not be able to remedy this problem (in our experiments, the regular planner was unable to solve this scenario in 20 minutes).

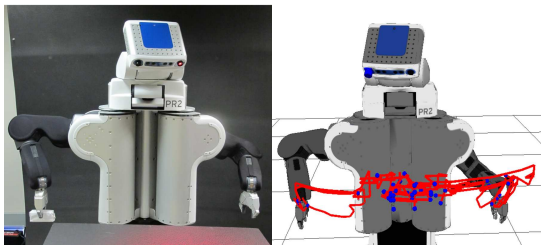
However, if a path were demonstrated, and then added to $G^{\mathcal{E}}$, our approach could harness this information to reach all the desired goals. We demonstrated such a path through teleoperation in the simulated world. We added this demonstrated path to $G^{\mathcal{E}}$ by having each recorded waypoint be a vertex with a cost represented by the cost function used in our approach. Figure 6a shows $G^{\mathcal{E}}$ containing this one demonstrated path.

After the demonstration, our approach was able to plan to all 12 goals quickly, as shown in Table VI. The resulting graph $G^{\mathcal{E}}$ is shown in Figure 6b. On average, 71% of the edges on a path produced by our planner were recycled from $G^{\mathcal{E}}$. The mean time to update $G^{\mathcal{E}}$ (add the new path and compute Floyd-Warshall) was 0.44 seconds.

While learning by demonstration is not the focus of this work, these preliminary results show it as a promising application. It is also interesting to note that any path can be demonstrated regardless of how sub-optimal the quality is. Our planner will use parts of it if possible and return a solution with a bound on the resulting solution quality.

C. Single Arm Planning

The planner's performance was also tested for tabletop manipulation using the PR2 robot (Figure 7a). A search-based planner [4] generates safe paths for each of the PR2's 7-DOF arms separately. The goals for the planner are specified as the position and orientation of the end-effector. Our implementation builds on the ROS grasping pipeline to pick up and put down objects on the table in front of the robot. Our approach is used during both the pick and place motions to plan paths



(a) Grasping pipeline setup (b) $G^{\mathcal{E}}$ partway through the experiment

Fig. 7. Tabletop manipulation experiments

TABLE VII
RESULTS ON TABLETOP MANIPULATION (411 GOALS)

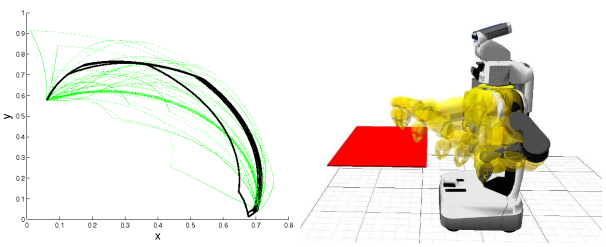
	mean time(s)	std dev time(s)	mean expands	mean cost
E-Graphs (E)	0.13	0.07	4	117349
Weighted A* (W)	0.26	0.10	145	109297
SBL (S)	0.24	0.09	N/A	N/A
Ratio (W/E)	2.50	1.23	66	1.03
Ratio (S/E)	2.44	1.50	N/A	N/A

for the robot's arms (individually).

In the course of the experiments, statistics for 411 planning requests were recorded using Weighted A*, our approach, and a randomized planner (SBL [14]). The results are shown in Table VII and Figure 7b shows $G^{\mathcal{E}}$ part-way through the experiment. We set $\varepsilon^w = 2$ and $\varepsilon^{\mathcal{E}} = 50$ for a sub-optimality bound of 100. We ran the regular Weighted A* planner with $\varepsilon^w = 100$.

Table VII shows that we have a speed increase of about 2.5 over both methods. The heuristic computation time (0.1s) dominates the planning times for both our approach and the Weighted A* approach, resulting in a smaller speedup than expected by the ratio of expansions. On average, 95% of the edges on a path produced by our planner were recycled from $G^{\mathcal{E}}$. The mean time to update $G^{\mathcal{E}}$ (add the new path and compute Floyd-Warshall) was 0.12 seconds.

These results show that our approach is competitive with sampling-based method in terms of planning times. However, our approach provides explicit cost minimization and therefore, some notion of consistency (for similar goals we provide similar paths). Figure 8a shows the paths of the end effector for 40 paths that had similar starts and goals



(a) 40 end effector trajectories with similar starts and goals. Our approach is in black, while the sampling approach is in dotted green.

(b) One visualized path

Fig. 8. E-Graphs provide similar solutions to similar problems

TABLE VIII
LENGTH OF 40 SIMILAR QUERIES IN TABLETOP MANIPULATION

	mean length (m)	std dev length (m)
E-Graphs	1.378	0.012
SBL	1.211	0.178

(within a 3x3x3cm cube). The dotted green paths are from the randomized approach while the black are from our approach. Notice that the randomized approach produces highly varying paths even after shortcutting. On the other hand our approach consistently provides almost the same path each time (we also applied a simple shortcutter to eliminate artifacts from the discritized environment). Table VIII shows that our approach has only a slightly longer path length (for the end effector distance) but a significantly lower standard deviation. While our planner's cost function is actually trying to minimize the change in joint angles, our average end effector path length is still relatively competitive with the sampling-based approach. Figure 8b shows one of these paths visualized in more detail.

VII. CONCLUSION

In this paper we have presented planning with Experience Graphs, a general search-based planning method for reusing parts of previous paths in order to speed up future planning requests. Our approach is able to do this while still providing theoretical guarantees on completeness and a bound on the solution quality. The paths our planner uses can be fed back from each planning episode in an online fashion to allow the planner to get better over time. The paths can also be demonstrated by a human (or other planner) to allow our planner to improve (while still providing a bound on solution quality). We provided experiments in the robotics domains of planning for a 7 DoF arm in tabletop manipulation tasks and full-body planning for the PR2 robot performing mobile manipulation tasks in warehouse and kitchen scenarios. Our comparison against the same search-based planning method without E-Graphs showed a speed-up of 1 to 2 orders of magnitude. Our comparison against a state of the art sampling based approach showed that we are competitive in terms of speed, but we yield more consistent paths.

As future work, we would like to introduce some heuristics for pruning $G^{\mathcal{E}}$ as it gets large, as well as taking a deeper look at applications in the field of learning by demonstration. We are also interested in making this into an anytime search so that the solution could approach optimality as more time is allowed.

REFERENCES

[1] C. Atkeson and J. Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

[2] Michael S. Branicky, Ross A. Knepper, and James J. Kuffner. Path and trajectory diversity: Theory and algorithms. In *IEEE International Conference on Robotics and Automation*, 2008.

[3] J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[4] B.J. Cohen, G. Subramanian, S. Chitta, and M. Likhachev. Planning for manipulation with adaptive motion primitives. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011.

[5] D. Ferguson, N. Kalra, and A. T. Stenz. Replanning with rrts. In *IEEE International Conference on Robotics and Automation*, May 2006.

[6] Nikolay Jetchev and Marc Toussaint. Trajectory prediction: Learning to map situations to robot trajectories. In *IEEE International Conference on Robotics and Automation*, 2010.

[7] Xiaoxi Jiang and Marcelo Kallmann. Learning humanoid reaching tasks in dynamic environments. In *IEEE International Conference on Intelligent Robots and Systems*, 2007.

[8] L. E. Kavraki, P. Svestka, J.-C Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[9] J. Lien and Y. Lu. Planning motion in environments with similar obstacles. In *Proceedings of the Robotics, Science and Systems Conference*, 2005.

[10] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS) 16*. Cambridge, MA: MIT Press, 2003.

[11] C. Liu and C. G. Atkeson. Standing balance control using a trajectory library. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[12] S. Martin, S. Wright, and J. Sheppard. Offline and online evolutionary bi-directional rrt algorithms for efficient replanning in environments with moving obstacles. In *IEEE Conference on Automation Science and Engineering*, 2007.

[13] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:219–236, 1970.

[14] Gildardo Sanchez and Jean claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *International Symposium on Robotics Research*, 2001.

[15] M. Stolle and C. Atkeson. Policies based on trajectory libraries. In *IEEE International Conference on Robotics and Automation*, 2006.

[16] M. Stolle, H. Tappeiner, J. Chestnutt, and C. Atkeson. Transfer of policies based on trajectory libraries. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.

[17] M. Zucker, J. Kuffner, and M. Branicky. Multipartite rrts for rapid replanning in dynamic environments. In *IEEE International Conference on Robotics and Automation*, 2007.

We thank Willow Garage for their support of this work. This research was also sponsored by ARL, under the Robotics CTA program grant W911NF-10-2-0016.