Speeding up heuristic computation in planning with Experience Graphs

Mike Phillips[1] and Maxim Likhachev[1]

***Abstract*— Experience Graphs have been shown to accelerate motion planning using parts of previous paths in an A* framework. Experience Graphs work by computing a new heuristic for weighted A* search on top of the domain's original heuristic and the edges in an Experience Graph. The new heuristic biases the search toward relevant prior experience and uses the original heuristic for guidance otherwise. In previous work, Experience Graphs were always built on top of domain heuristics which were computed by dynamic programming (a lower dimensional version of the original planning problem). When the original heuristic is computed this way the Experience Graph heuristic can be computed very efficiently. However, there are many commonly used heuristics in planning that are not computed in this fashion, such as euclidean distance. While the Experience Graph heuristic can be computed using these heuristics, it is not efficient, and in many cases the heuristic computation takes much of the planning time. In this work, we present a more efficient way to use these heuristics for motion planning problems by making use of popular nearest neighbor algorithms. Experimentally, we show an average 8 times reduction in heuristic computation time, resulting in overall planning time being reduced by 66%. with no change in the expanded states or resulting path.**

I. INTRODUCTION

Many of the tasks we would like robots to perform are highly repetitive, such as transporting objects from one place to another in a warehouse or clearing the table and putting dirty dishes in the sink. Many of these tasks also require motion planning in order to guarantee each motion is completed in a collision free and efficient manner. With tasks being relatively similar each time, it is inefficient to generate each motion from scratch. Instead, we would prefer for our planners to make use of prior experience to improve performance.

Planning with Experience Graphs (E-Graphs) was introduced in [1]. This method allows weighted A* based planners to incorporate previously generated paths and user demonstrations into the planning process. It does so while maintaining guarantees on resolution completeness (if a solution exists from the start to the goal within the user chosen discretization, the planner will find it) and bounded suboptimality (the planner's solutions will be within a user-chosen factor of the cost of an optimal solution). In motion planning, A* based methods search from the starting configuration toward the goal state (the reverse is also possible) by repeatedly choosing a state that has been discovered and *expanding* it (applying a set of motions to it and adding the resulting new states to the set of discovered). These methods use a heuristic function which estimates the remaining cost-to-go from any state to the goal, in order to focus search effort on states that are likely to lead to the goal quicker. Experience Graphs construct a special heuristic function for the A* planner by combining the original heuristic with previous paths. The result is a heuristic which is goal directed but biased toward getting there via reusing previous paths when relevant.

The E-Graph heuristic can build on top of almost any reasonable user chosen heuristic, but is particularly efficient to compute when combined with a heuristic computed using dynamic programming. These tend to be heuristics computed over lower dimensional versions of the original problem. For instance, in a robot navigation problem the configuration space might be position and heading (x, y, θ) and a heuristic might be the cost-to-go (ignoring θ) for each 2D cell computed via a single 2D search from the goal. Since the E-Graph heuristic is also computed by dynamic programming (Section III-B), it turns out that its computation can be folded into these dynamic programming-based heuristics with almost no additional overhead. In previous publications on Experience Graphs, these types of heuristics were always used.

However, many commonly used heuristics in robotics are not computed this way, such as euclidean and manhattan distance or heuristics based on the dubins car model. For regular A* searches (without E-Graphs) these types of heuristics are often efficient to use, in many cases having $O(1)$ computation time for a pair of states. However, the naive implementation of the E-Graph heuristic, on top of these type of heuristics is linear in the size of the E-Graph when evaluating *each* state's heuristic that the search encounters. This consumes much of the planning time and for large E-Graphs can be prohibitive.

In this paper, we speed up the E-Graph heuristic computation for general heuristics with no change in planner behavior, i.e. the planner expands the exact same states in the same order and produces the exact same paths. We make use of offline precomputation time and the availability of efficient nearest neighbor methods, which have also played a role in speeding up popular sampling-based motion planners (e.g. RRT, PRM).

We experimentally evaluate these improvements in a challenging full body motion planning domain for the PR2 robot. We use the Euclidean distance heuristic which previously could be less efficient when combined with E-Graphs. We observe an average speedup of 8 times over the previous heuristic computation method resulting in overall planning

*This research was sponsored by the NSF grant IIS-1409549.

[1]Robotics Institute, Carnegie Mellon University, Pittsburgh, PA

times taking 66% as long.

II. RELATED WORK

As mentioned earlier, Experience Graph are used to plan paths faster by reusing previously planned paths [1] or demonstrations [2]. There are a number of other approaches that use previous paths to speedup motion planning [3], [4], [5], [6]. However, the improvements being made in this paper are specific to the Experience Graph heuristic computation.

A large part of our improvements will make use of exact nearest neighbor methods which build a data structure that allows them to avoid looking at all members of the dataset during queries. The KD-tree (k-dimensional tree) assumes the data is a set of k-dimensional points and recursively splits the data in half using axis-aligned hyperplanes in the vector space [7]. The VP-tree (vantage point tree), makes fewer assumptions. It works by selecting a pivot from within the dataset and then choosing a "median radius" around it such that half of the datapoints are inside and half are outside according to the distance metric. It then recurses on the two sets [8]. Here the data is not required to be in a vector space like the KD-tree, instead there are only contraints on the distance metric itself. The main constraint is that it satisfies the triangle inequality. A similar method is the GH-tree (generalized hyperplane tree) which works by choosing two pivots from the dataset and splitting the data in half according to how close they are to both points (roughly, for each datapoint, is it closer to the first pivot or the second) and then recursing [9]. The GH-tree makes the same assumptions on the distance metric as the VP-tree. LSH (locality-sensitive hashing) is a method which uses a hash function to group nearby datapoints into the same bin with high probability [10].

Identifying nearest neighbors quickly is a crucial component to sampling-based motion planners such as RRTs (rapidly-exploring random tree) [11] and PRMs (probabilistic roadmap) [12]. These methods repeatedly choose random samples in configuration space and then attempt to connect them to one or more of the nearest neighbors in this randomly generated tree or graph. In [13] KD-trees are applied to such planners to improve performance.

III. BACKGROUND

A. Definitions

E-Graphs assume the motion planning 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.
- $G^E(V^E, E^E)$ is the E-Graph ($G^E \subseteq G$). It is typically a colleciton of previously planned paths or demonstations.
- $c(u, v)$ is the cost of the edge from vertex u to vertex v
- $c^E(u, v)$ is the cost of the edge from vertex u to vertex v in graph G^E and is always equal to $c(u, v)$ when it exists. If edge (u, v) does not exist in E^E, $c^E(u, v) = \infty$.

The algorithm is based on heuristic search and therefore uses a heuristic function $h^G(u, v)$ estimating the cost from u to v ($u, v \in V^G$). E-Graphs assume $h^G(u, v)$ is admissible and consistent for any pair of states $u, v \in V^G$. An admissible heuristic never overestimates the minimum cost from any state to any other state. A consistent heuristic $h^G(u, v)$ is one that satisfies the triangle inequality, $h^G(u, v) \leq c(u, s) + h^G(s, v)$ and $h^G(u, u) = 0$, $\forall u, v, s \in V^G$ and $\forall (u, s) \in E^G$.

B. Experience Graphs

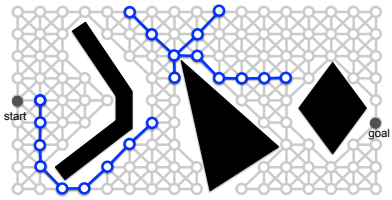
This section provides a brief description of how E-Graphs work. For more details see the prior work where Experience Graphs were introduced [1].

An Experience Graph G^E is a collection of previously planned paths or demonstrations (experiences). Planning with Experience Graphs uses weighted A* (A* with heuristics inflated by $\varepsilon > 1$) to search the original graph G (which represents the planning problem) but tries to reuse paths in G^E in order to minimize the exploration of the original graph G (which is significantly larger than G^E). This is done by modifying the heuristic computation of weighted A* to drive the search toward paths in G^E, that appear to lead towards the goal. Essentially, this comes down to computing a new heuristic distance from the state in question to the goal where traveling off of G^E is penalized but traveling on edges of G^E is not. The new heuristic h^E is defined in terms of the original heuristic h^G and edges in G^E for all states s_0 in the original graph.

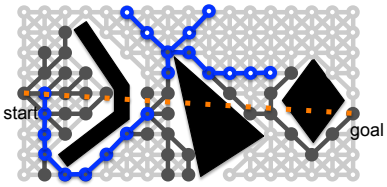
$$h^E(s_0) = \min_{\pi} \sum_{i=0}^{N-2} min\{\varepsilon^E h^G(s_i, s_{i+1}), c^E(s_i, s_{i+1})\} \tag{1}$$

where π is called the *heuristic path* $\langle s_0 \dots s_{N-1} \rangle$ and $s_{N-1} = s_{goal}$ and ε^E is a scalar ≥ 1. As shown in [1], the heuristic is ε^E-consistent and therefore guarantees that the solution cost will be no worse than ε^E times the cost of the optimal solution when running A* search to find a path. More generally, planning with Experience Graphs using weighted A* search inflates the entire h^E heuristic by ε. Consequently, the cost of the solution is bounded by $\varepsilon \cdot \varepsilon^E$ times the optimal solution cost.

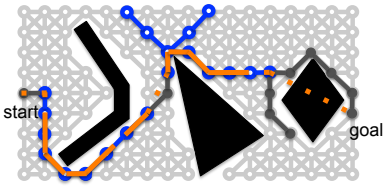
Equation 1 is computed by finding the shortest heuristic path (Dijkstra's algorithm) from s_0 to the goal in a simplified version of the planning problem where there are two kinds of edges. The first set (the first term in Eqn. Equation 1). are edges that represent the same connectivity as h^G in the original planning problem but their cost is inflated by ε^E. So if the original heuristic is euclidean distance then there are edges between all pairs of states. However, if the original heuristic is some lower dimensional search then we use the edge set from that. The second set of edges (the second term in Eqn. Equation 1) are from G^E with their costs c^E



(a) G^E



(b) $\varepsilon^E = 1$



(c) $\varepsilon^E \to \infty$

Fig. 1. Effect of ε^E. The light gray circles and lines show the original graph. The highlighted states and edges in (a) show the E-Graph. In (b) and (c) the dark gray circles show states explored by the planner in order to find a solution. The light dashed line shows the heuristic path from the start state. Notice that when ε^E is large, this path travels along the E-Graph and avoids most obstacles (there are few explored states). On the other hand when ε^E is small, the heuristic (in this case euclidean distance) drives the search into several obstacles and causes many more expansions. It should be noted that $\varepsilon > 1$ is used in these examples.

(∞ if the edge is not in G^E). As ε^E increases, the heuristic computation goes farther out of its way to use helpful E-Graph edges.

Figure 1 shows the effect of varying the parameter ε^E. As it gets large, the heuristic is more focused toward E-Graph edges. It draws the search directly to the E-Graph, connects prior path segments, and only searches off of the E-Graph when there are not any useful experiences (such as around the last obstacle). There are very few expansions and much of the exploration of the space is avoided. As ε^E approaches 1 (optimality) it ignores G^E and expands far more states.

IV. ALGORITHM

A. Naive approach

As described earlier, the minimum *heuristic path* computed to find $h^E(s)$ is composed of alternating path segments (those using the original heuristic h^G and those using E-Graph edges).

In the naive implementation (for a general heuristic) after the goal is given, h^E is computed for all E-Graph vertices upfront. To make the writing clearer we will use H^E instead of h^E for these precomputed values for E-Graph vertices and goal state (the two represent the same quantity, but when you see H^E you know it is the h^E value of an E-Graph vertex or goal state). Then during the search, when $h^E(s)$ is queried, there are two cases, s is either an E-Graph vertex (or goal state) in which case, the heuristic value was precomputed and we return $H^E(s)$ or s is not on the E-Graph. In this case, we recognize that the heuristic path for s will be the heuristic path of one of the E-Graph vertices (or goal) plus 1 additional segment which directly connects s to that vertex. In particular, it will use the vertex that minimizes $h^E(s)$. More formally,

$$h^E(s) = min_{s' \in V'} \left(\varepsilon^E h^G(s, s') + H^E(s') \right) \qquad (2)$$

Where $V' = \{V^E \cup s_{goal}\}$. Computing the H^E values can be performed using a single Dijkstra search from the goal state through a full connected graph $G'(V', E')$ where, $E' = \{(u, v) | \forall u, v \in V'\}$. The edge weights are defined as $w(u, v) = min \left(\varepsilon^E h^G(u, v), c^E(u, v) \right)$. Recall that if edge $(u, v) \notin E^E$ then $c^E(u, v) = \infty$. Essentially, edge weights in the fully connected graph are minimum of the inflated original heuristic and the cost of the corresponding E-Graph edge.

Briefly, Dijkstra's algorithm find the shortest cost from a source node in a graph to all other nodes the graph. All nodes are put into a priority queue based on their current shortest cost from the source (initially the source itself has cost 0 while all other nodes have infinite cost). The algorithm repeatedly removes the lowest cost node v from the queue and "expands" it. This means it tries to reduce the costs of all of its neighbors (if the cost of v plus the edge cost to a neighbor u is less than the current cost of u, then the cost of u is updated). When a node is removed from the queue, its cost is known to be optimal. The algorithm terminates when the queue is empty (or when all remaining nodes in the queue have infinite cost, indicating that the remaining nodes are not reachable from the source).

Algorithm 1 shows all of the naive method. ONSTARTUP refers to computation done before we are told the start and goal, when the planner initializes. At this point, only the E-Graph edges and the parameter ε^E are known. These are true precomputations that do not impact planning times. The naive method does not make use of this. The AFTERGOAL method is called at the start of the planning episode once the goal has been given. This is where the naive method computes Dijkstra search on G' with s_{goal} as the source. Notice that this takes $O(|V'|^2 log|V'|)$ since the graph G' is fully connected (using a binary heap for the Dijkstra priority queue). Finally, the GETHEURISTIC function is called on every state encountered by the search. For the naive method this implements Equation 2. Notice that each call to GETHEURISTIC takes $O(|V'|)$ which is expensive as the E-Graph gets larger.

We will accelerate the both AFTERGOAL and GETHEURISTIC functions in the following sections.

B. Replacing the Dijkstra Search

The Dijkstra search computed at the start of the planning episode can be expensive (as it scales with the size of the E-Graph). By doing some extra work on planner initialization,

Algorithm 1 Naive method

1: **procedure** ONSTARTUP()
2: **procedure** AFTERGOAL()
3: Run Dijkstra's Algorithm on G' with source s_{goal}
4: $H^E(s)$ = the cost of s in the Dijkstra search
5: **procedure** GETHEURISTIC(s)
6: $h^E(s) = min_{s' \in V'} \left(\varepsilon^E h^G(s, s') + H^E(s')\right)$

we can reduce the time spent at the start of each planning episode. When the planner initializes, we assume we already know the E-Graph and the parameter ε^E. If this is not true, than the following procedure will have to be repeated whenever either changes.

On planner initialization (ONSTARTUP in Algorithm 2), we will be computing the all-pairs shortest paths on the graph G'' which is the same as G' but without the goal vertex (since it is not known yet). To do this we create an adjacency matrix with the two types of edges that are used in G'. We then run the Floyd-Warshall algorithm to compute $d(u, v) \forall u, v \in V^E$, i.e. the shortest path from any u to v in G''. This runs in $O(|V^E|^3)$, which is large, but since it just happens once on planner initialization, it does not affect planning times. In fact, this could be computed once offline and the $d(u, v)$ values could be written to file for future use.

When a planning episode starts (and we are given the goal), we need to compute $H^E(s)$ but we will make use of the precomputed $d(u, v)$ values to do this more efficiently than the Dijkstra search from the naive method. Instead, each $H^E(s)$ is computed by finding the E-Graph node the goal connects to first along its heuristic path to s. More formally,

$$H^E(s) = min_{s' \in V^E} \left(\varepsilon^E h^G(s_{goal}, s') + d(s', s)\right), \forall s \in V^E \tag{3}$$

The computation of all H^E values now takes $O(|V^E|^2)$ instead of the naive $O(|V^E|^2 log|V^E|)$.

1) Dijkstra with an unsorted array: An alternative to the above precomputation approach is to run Dijkstra's algorithm using a different data structure. While Dijkstra's is often run with a binary heap, resulting in the $O(|V^E|^2 log|V^E|)$ runtime. For graphs with high connectivity (ours is fully connected) it is actually faster to use an unsorted array instead. It results in a runtime of $O(|V|^2)$. This will make AFTERGOAL take the same asymptotic runtime as the method we just posed, without having to run the all-pairs shortest path precomputation (ONSTARTUP can go back to being empty). However, in practice, we still found the version that uses the precomputations to run slightly faster, and therefore, present those results in our experiments.

C. Making GETHEURISTIC sub-linear

Each time the naive method evaluates the heuristic for a state, it looks at every node in the E-Graph to find the one that minimizes Eqn. 2, which takes $O(|V^E|)$ time.

We will accelerate this process by using popular nearest neighbor methods. The methods we will be considering are ones which can return exact nearest neighbors. In general,

Algorithm 2 Improved method

1: **procedure** ONSTARTUP()
2: Build adjacency matrix for E-Graph vertices
3: Run Floyd-Warshall to compute $d(u, v) \forall u, v \in V^E$
4: **procedure** AFTERGOAL()
5: $H^E(s_{goal}) = 0$
6: $H^E(s) = min_{s' \in V^E} \left(\varepsilon^E h^G(s_{goal}, s') + d(s', s)\right), \forall s \in V^E$
7: Build nearest neighbor data stucture NN
8: **procedure** GETHEURISTIC(s)
9: $v = getNearestNeighbor(NN, s)$
10: **return** $\varepsilon^E h^G(s, v) + H^E(v)$

these methods require the distance function to be a metric $\mathcal{F}$ and therefore must satisfy three constraints $\forall u, v, w$.

- $\mathcal{F}(u, u) = 0$
- $\mathcal{F}(u, v) = \mathcal{F}(v, u)$
- $\mathcal{F}(u, v) \leq \mathcal{F}(v, w) + \mathcal{F}(w, u)$

For a state s' in the E-Graph we compute the heuristic upfront when getting the goal, as described in the previous section. These states are then put into the nearest neighbor data structure NN as the following vector.

$$\tilde{u} = \begin{bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \end{bmatrix} = \begin{bmatrix} s' \\ H^E(s') \end{bmatrix}$$

During planning, when we call GETHEURISTIC on a state s we need to find a state s' such that it minimizes Eqn. 2. To do this we represent s as vector

$$\tilde{v} = \begin{bmatrix} \tilde{v}_1 \\ \tilde{v}_2 \end{bmatrix} = \begin{bmatrix} s \\ 0 \end{bmatrix}$$

and do a look up in NN according to the following distance metric:

$$\mathcal{F}(\tilde{u}, \tilde{v}) = \varepsilon^E h^G(\tilde{u}_1, \tilde{v}_1) + |\tilde{u}_2 - \tilde{v}_2| \tag{4}$$

This metric satisfies all the required conditions since the two terms both respect the triangle inequality (recall h^G is consistent).

Now that we have a metric, we can employ a wide variety of exact nearest neighbor methods such as VP-tree, GH-tree, and KD-tree. All of these work by recursively splitting the dataset in half based on a pivot element or other criteria. These methods can achieve exact nearest neighbor lookups in logarithmic time under certain contitions regarding the distribution of the data. Though in general they perform faster than linear search mostly on large datasets.

At the end of the AFTERGOAL function of Algorithm 2, the nearest neighbor data structure NN is built using the metric $\mathcal{F}$. Then in the GETHEURISTIC function, we simply query NN for the nearest neighbor to s and return the distance.

D. Using optimized KD-trees

Some optimized implementations of KD-trees require the distance metric to be of the form:

$$dist(x, y) = f_1(x_1, y_2) + f_2(x_2, y_2) + \ldots + f_n(x_n, y_n) \tag{5}$$

where $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ are all scalars.

FLANN (fast library for approximate nearest neighbors) is a popular nearest neighbor library that does this [14].

Euclidean distance is supported under this form by squaring it and therefore effectively removing the square root (which does not affect the order of nearest neighbors). However, our metric $\mathcal{F}$ does not fit the form if h^G is euclidean distance since we then have:

$$dist(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 \ldots} + |x_{H^E} - y_{H^E}| \tag{6}$$

and the first term couples $x_1, x_2, \ldots$.

To support these optimized libraries, we develop an additional method for handling these constraints.

We assume that the original heuristic h^G can be written in the form of Eqn. 5. Then we will build the KD-tree using only h^G (note this means the KD-tree can be built in ONSTARTUP in Algorithm 3). Then in GETHEURISTIC the nearest neighbor (according to h^G) is found using the KD-tree and in a post-processing step, we make a linear pass through the E-Graph nodes to find the one that minimizes h^E. The trick is that we will use the knowledge that we have about the minimal elements according to h^G to terminate the linear pass early. In practice, the linear pass only looks at a small fraction of the E-Graph nodes before deciding we have the optimal one.

More specifically, in function GETHEURISTIC of Algorithm 3, we will be maintaining four variables

- h^E_{best}: is the best (lowest) value of $h^E(s)$ computed so far (from E-Graph vertices inspected)
- H^E_{low}: a lower bound on H^E among E-Graph vertices we have not yet inspected
- h^G_{low}: a lower bound on $h^G(s,s')$ among E-Graph vertices s' we have not yet inspected
- H^E_{max}: a stopping condition, when $H^E_{low} \geq H^E_{max}$ it is no longer possible for any remaing E-Graph nodes to reduce h^E_{best}

The first thing in $GetHeuristic$ is to evaluate $h^E(s)$ using s_{goal} and initialize h^E_{best} to this value. This also initializes H^E_{low} to 0, the H^E value of s_{goal}. We then find the K nearest neighbors in the KD-tree (recall the KD-tree only uses h^G) and evaluate $h^E(s)$ using each, and updating h^E_{best} as needed. The E-Graph nodes returned from the KD-tree are sorted, so nn_1 has the lowest h^G distance to s of all E-Graph nodes. We now set h^G_{low} using nn_K knowing that all remaining E-Graph nodes we inspect must have an h^G distance to s that is at least as large as the one from nn_K. Therefore, it is beneficial to set $K > 1$ in order get a better lower bound (though not as large as $|V^E|$ since that would defeat the point).

We then set the stopping condition $H^E_{max} = h^E_{best} - h^G_{low}$, which basically says that for an uninspected E-Graph state b to provide a lower h^E_{best}, $H^E(b)$ must be lower by at least h^G_{low} to make up for the fact that we know $h^G(s,b)$ is at least that large. We prove the correctness of this stopping condition shortly.

Algorithm 3 Using optimized KD-trees

```
1:  procedure OnStartUp()
2:      Build adjacency matrix for E-Graph vertices
3:      Run Floyd-Warshall to compute d(u,v) ∀u,v ∈ V^E
4:      Build KDtree of E-Graph vertices with metric h^G
5:  procedure AfterGoal()
6:      H^E(s_goal) = 0
7:      H^E(s) = min_{s'∈V^E} (ε^E h^G(s_goal, s') + d(s', s)), ∀s ∈ V^E
8:      Create S a list of EGraph nodes sorted by H^E
9:  procedure GetHeuristic(s)
10:     H^E_low = 0
11:     h^E_best = ε^E h^G(s, s_goal)
12:     nn_1 ... nn_K = getNN(KDtree, K, s)
13:     h^E_best = min(h^E_best, min_{i=1...K} (ε^E h^G(s, nn_i) + H^E(nn_i)))
14:     h^G_low = ε^E h^G(s, nn_K)
15:     H^E_max = h^E_best − h^G_low
16:     for i = 1, 2, ..., |S| do
17:         h^E_temp = ε^E h^G(s, S(i)) + H^E(S(i))
18:         if h^E_temp < h^E_best then
19:             h^E_best = h^E_temp
20:             H^E_max = h^E_best − h^G_low
21:         H^E_low = H^E(S(i))
22:         if H^E_low ≥ H^E_max then
23:             break
24:         if h^E_best ≤ ε_KD(h^G_low + H^E_low) then
25:             break
26:     return h^E_best
```

At this point we start iterating through S which contains all E-Graph nodes sorted by their H^E values in increasing order (computed in AFTERGOAL). At each iteration we see if we can update h^E_{best} using the next E-Graph node. If so, we also update H^E_{max} according to its definition, which will lower the stopping condition, making it possible to terminate ever earlier.

Then we update the H^E_{low} based on the current E-Graph vertex since we know we are iterating through H^E values in increasing order, every E-Graph vertex remaining to be inspected has an H^E at least as large as the vertex we just inspected. We then see if it is possible to terminate early. We have an optimal termination condition $H^E_{low} \geq H^E_{max}$ and a bounded suboptimal condition $h^E_{best} < \varepsilon_{KD}(h^G_{low} + H^E_{low})$. For $\varepsilon_{KD} > 1$ we can often terminate earlier knowing that we are within a factor of ε_{KD} times the optimal h^E_{best}.

We will now prove that the two termination conditions are optimal and bounded suboptimal, respectively.

Theorem 1 (Optimal termination): When the main loop of GETHEURISTIC(S) terminates due to $H^E_{low} \geq H^E_{max}$, h^E_{best} is minimal.

Proof: Assume for sake of contradiction that after terminating the loop using the optimal condition, there is some E-Graph node b^* which would have led to h^{E*}_{best} lower than the h^E_{best} found using b, the node that provided the current best value. Since we terminated,

$$H^E_{low} \geq H^E_{max}$$

Since we iterate through the E-Graph nodes in increasing order by H^E, $H^E(b^*) \geq H^E_{low}$. So,

$$H^E(b^*) \geq H^E_{max}$$

Since $H^E_{max} = h^E_{best} - h^G_{low}$,

$$H^E(b^*) \geq h^E_{best} - h^G_{low}$$
$$H^E(b^*) + \varepsilon^E h^G(s, b^*) \geq h^E_{best} - h^G_{low} + \varepsilon^E h^G(s, b^*)$$
$$h^{E*}_{best} \geq h^E_{best} - h^G_{low} + \varepsilon^E h^G(s, b^*)$$

Since b^* has not been evaluated, it cannot be among the nearest neighbors found by the KDtree and therefore, $-h^G_{low} + \varepsilon^E h^G(s, b^*) \geq 0$. Therefore, we can safely remove these two terms from the right hand side.

$$h^{E*}_{best} \geq h^E_{best}$$

Contradiction. ■

If instead of finding the best E-Graph node which minimizes h^E_{best}, we find a vertex that only computes h^E_{best} within a user-chosen factor of minimal, the algoritm often terminates significantly faster. The chosen factor will affect the theoretical bound on the costs of solutions found by the planner.

Theorem 2 (Bounded suboptimal termination): When the main loop of GETHEURISTIC(S) terminates due to $h^E_{best} \leq \varepsilon_{KD}(h^G_{low} + H^E_{low})$, $h^E_{best} \leq \varepsilon_{KD} h^{E*}_{best}$, where h^{E*}_{best} is the minimal possible value of h^E_{best}.

Proof: Suppose that b^* is the node which computes h^{E*}_{best}. Also assume that b^* is found in the main loop since if it is s_{goal} or one of the neighbors returned by the KDtree, then we are optimal and trivally satisfy the bound. We know that $h^G_{low} \leq \varepsilon^E h^G(s, b^*)$. We also know that if we have not actually evaluated b^* yet then $H^E_{low} \leq H^E(b^*)$. Therefore if we have terminated using the bounded suboptimal condition,

$$h^G_{low} + H^E_{low} \leq \varepsilon^E h^G(s, b^*) + H^E(b^*)$$
$$h^G_{low} + H^E_{low} \leq h^{E*}_{best}$$
$$\varepsilon_{KD}(h^G_{low} + H^E_{low}) \leq \varepsilon_{KD} h^{E*}_{best}$$
$$h^E_{best} \leq \varepsilon_{KD}(h^G_{low} + H^E_{low}) \leq \varepsilon_{KD} h^{E*}_{best}$$
$$h^E_{best} \leq \varepsilon_{KD} h^E_{best}$$

■

Terminating with bounded suboptimal termination means that the heuristic values returned may overestimate the true minimal distance to the goal by an additional ε_{KD}. This will then raise the overall suboptimality bound of planning with Experience Graphs to $\varepsilon \cdot \varepsilon^E \cdot \varepsilon_{KD}$.

V. EXPERIMENTAL RESULTS

Planning with E-Graphs has already been compared extensively against other approaches in [1]. In this paper, we focus on merely improving its performance when using general heuristics especially those not computed by dynamic programming. Therefore, in these experiments we tested the improved heuristic computation using euclidean distance. This is one of the most common heuristics used in robot motion planning and before the algorithms presented in this paper, its use with E-Graphs was avoided due to its inefficient computation, especially as the E-Graph gets large.

The domain we chose is 11 DoF (degree of freedom) full body planning for the PR2 robot. The planner has control of the robot's base movement (x, y, θ), prismatic spine for adjusting the height of the upper half the robot, and 7 DoF control over the right arm.

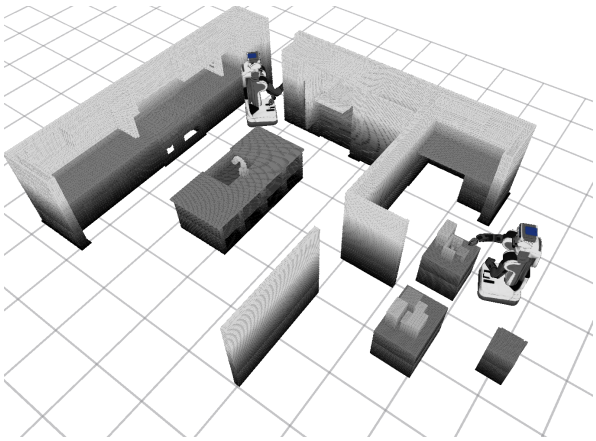


Fig. 2. An example start and goal state in the kitchen domain

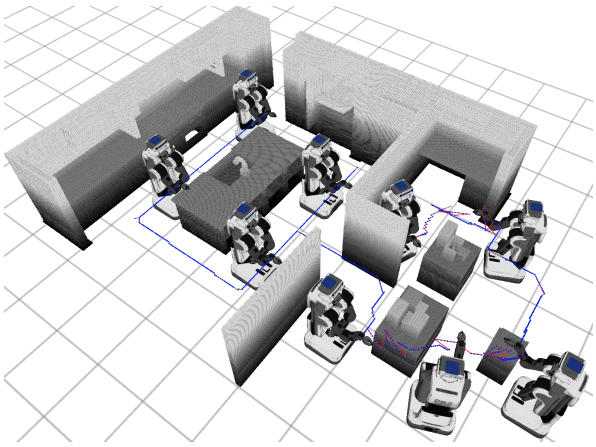


Fig. 3. The E-Graph used for the experiments. Several configurations are shown. To not crowd the image, the rest of the E-Graph is shown as the red and blue line which shows where the gripper of the robot at each state.

Both the start and goal states are fully specified configurations randomly placed throughout a simulated kitchen environment with two rooms and a narrow doorway connecting them (Fig. 2). Random goal states always have the right hand reaching onto the surface of a cluttered table. We initialized the E-Graph with five demonstrations that visit the main areas of the environemnt (Fig. 3). The resulting E-Graph had 942 vertices in it. We then ran 100 planning episodes to compare the naive heuristic computation against the improved version with different nearest neighbor data structures. Specifically, we ran the VP-tree, GH-tree, and KD-tree. The KD-tree implementation we used is from FLANN [14] and therefore we had to use the method described in Section IV-D (we used $K = 5$ nearest neighbors). All of these methods are running the same E-Graph planner, with the only difference being how the heuristic is being computed. The naive method follows Algorithm 1, VP-tree and GH-tree follow Algorithm 2, and the KD-tree approaches

follow Algorithm 3).

The E-Graph parameters used were $\varepsilon = 2$ and $\varepsilon^E = 10$ providing a suboptimality bound of 20 (the theoretical bound is higher by a factor of ε_{KD} for KD-tree methods using the bounded suboptimal termination criteria). While the theoretical bound is quite large, in practice the solutions costs are much lower than this. While computing true optimal paths in such a high-dimensional space (for the purpose of comparison) is very difficult, we estimate the found solutions have costs no worse than 2 or 3 times the optimal cost.

TABLE I
PLANNING TIMES USING DIFFERENT HEURISTIC COMPUTATION METHODS

	mean planning time	mean h^E time	median planning time	median h^E time	% time on h^E
naive	1.09 ± .72	0.39 ± .25	0.19	0.09	47%
vp	0.73 ± .50	0.05 ± .03	0.115	0.01	12%
gh	0.78 ± .53	0.08 ± .05	0.13	0.03	23%
kd(1)	0.84 ± .55	0.15 ± .10	0.14	0.055	31%
kd(2)	0.77 ± .54	0.06 ± .03	0.125	0.015	16%
kd(3)	0.73 ± .50	0.05 ± .03	0.115	0.01	13%

In Table I we present the results of our experiments. All methods succeeded on 94 of the 100 trials given a 30 second time limit. The rows show the different methods. For the KD-tree, the number afterward indicates the suboptimality bound used (ε_{KD}). Naive, VP-tree, GH-tree, and KD-tree(1) are all optimal and therefore result in the planner expanding the exact same states (mean number of expands was 492) and producing the same path. The KD-tree using suboptimality bounds of 2 and 3, resulted in negligible difference in the number of expansions (less than 1% change; mean number of expands was 491). The table presents the mean and median of total planning time (columns 2 and 4) as well as the mean and median of the planning time that went toward computing heuristics (columns 3 and 5). We also report the average percentage of planning time that went toward computing heuristics (column 6). The confidence intervals shown with the means are at the 95% confidence level. We can see that the VP-tree performs the best in all measures among the optimal methods. Though if the KD-tree method is given a suboptimality bound of 3, it performs just as well. Overall we can see that the amount of planning time that goes toward heuristic computation drops from 47% using the naive method down to 12% when using the VP-tree. Heuristic computation time was reduced by a factor of 8 and on average planning times dropped by 33%.

It is important to note that these improvements to heuristic computation time and planning times come at no cost (other than a slightly more complicated implementation and larger memory footprint). The exact same states are expanded, and the same paths are found, just in less time.

The methods that perform precomputation (all but naive) took an additional 1.8s for the planner to initialize. This is a one time cost when the planner is launched (all of the 100 trials could then be run). While this time is not particularly large, it could become worse if the E-Graph were much bigger. Additionally, any time the E-Graph changes (e.g. a new path is added or the environment changes), the precomputations would need to be re-run. If the precomputations would have to be run often, we suggest not using the precomputations and instead running the Dijkstra search in AFTERGOAL with an unsorted array (Section IV-B.1). We found that this makes all methods slower by 0.02s but it avoids the need for precomputation.

VI. CONCLUSIONS

In this work we presented a more efficient way to compute general heuristics for E-Graphs, especially for those which are not computed using dynamic programming. Planning with Experience Graphs can now be done more efficiently with heuristics like euclidean distance, which are ubiquitous in robot motion planning. Our improvements make use of nearest neighbor methods which have also been used to speed up sampling-based motion planners. Future work is to experiment with other heuristics like the Dubins car model.

REFERENCES

[1] M. Phillips, B. J. Cohen, S. Chitta, and M. Likhachev, "E-graphs: Bootstrapping planning with experience graphs," in *Robotics: Science and Systems*, 2012.

[2] M. Phillips, V. Hwang, S. Chitta, and M. Likhachev, "Learning to plan for constrained manipulation from demonstrations," in *Robotics: Science and Systems*, 2013.

[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] N. Jetchev and M. Toussaint, "Trajectory prediction: Learning to map situations to robot trajectories," in *IEEE International Conference on Robotics and Automation*, 2010.

[5] X. Jiang and M. Kallmann, "Learning humanoid reaching tasks in dynamic environments," in *IEEE International Conference on Intelligent Robots and Systems*, 2007.

[6] D. Berenson, P. Abbeel, and K. Goldberg, "A robot path planning framework that learns from experience," in *ICRA*, 2012.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, no. 3, pp. 209–226, Sept. 1977.

[8] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '93, 1993, pp. 311–321.

[9] J. K. Uhlmann, "Satisfying general proximity / similarity queries with metric trees," *Information Processing Letters*, vol. 40, no. 4, pp. 175 – 179, 1991.

[10] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99, 1999, pp. 518–529.

[11] J. J. K. Jr. and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *ICRA*, 2000.

[12] 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*, vol. 12, no. 4, pp. 566–580, 1996.

[13] A. Atramentov and S. LaValle, "Efficient nearest neighbor searching for motion planning," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 1, 2002, pp. 632–637 vol.1.

[14] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Application VISSAPP'09)*. INSTICC Press, 2009, pp. 331–340.