Roadmap Subsampling for Changing Environments

Sean Murray[1, 3] George D. Konidaris[2, 3] Daniel J. Sorin[1, 3]

Abstract— **Precomputed roadmaps can enable effective multi-query motion planning: a roadmap can be built for a robot as if no obstacles were present, and then after edges invalidated by obstacles observed at query time are deleted, path search through the remaining roadmap returns a collision-free plan. However, large roadmaps are memory intensive to store, and can be too slow for practical use. We present an algorithm for compressing a large roadmap so that the collision detection phase fits into a computational budget, while retaining a high probability of finding high-quality paths. Our algorithm adapts work from graph theory and data mining by treating roadmaps as unreliable networks, where the probability of edge failure models the probability of a query-time obstacle causing a collision. We experimentally evaluate the quality of the resulting roadmaps in a suite of four motion planning benchmarks.**

I. Introduction

A roadmap [12] is a popular data structure for multi-query motion planning. The roadmap can be constructed once—as if the robot was surrounded by completely free space—and then runtime collision detection temporarily deletes those edges whose motions would cause collisions with obstacles currently in the environment [14]. Using a fixed roadmap in this way differs from planners that rely on the ability to sample additional nodes *ad infinitum* at runtime to attain probabilistic completeness and probabilistic optimality [12], [10]. A fixed roadmap will never return a path that is in collision, but could report failure even when a path exists. This can happen for two reasons. First, obstacles present in the environment may bisect the roadmap such that no collision-free paths remain from the start to the goal, even though paths may exist if considering the entire free configuration space. Second, the roadmap may not contain sufficient coverage in the areas that the current query demands; even if no obstacles are present, the roadmap may not have any nodes that satisfy the query's goal constraints. A successful fixed roadmap strategy must thus create roadmaps that find high quality paths in relevant environments with high probability.

While it is easy to obtain robust performance on roadmaps of unbounded size (such as those generated with PRM* [10]), the problem is more challenging when trying to achieve good performance on a budget. Larger graphs take longer to query, so smaller graphs are more desirable to reduce planning latency. Large roadmaps may also be impractical to store and transmit to mobile robots.

We present a novel approach to creating roadmaps that meet limited budgets while enabling the robot to effectively solve future queries. For a given (large) baseline roadmap, we show how to find sub-roadmaps that meet a given roadmap size budget while preserving three qualities: path quality, workspace coverage, and connectivity.

[1]Department of Electrical and Computer Engineering, Duke University
[2]Department of Computer Science, Brown University
[3]Realtime Robotics

II. Related Work in Robotics

We require roadmaps that are multi-query and not easily bisected. Thus, tree-based algorithms like RRT and RRT* [13] are unsuitable, and we focus on multi-query algorithms like PRM and PRM* [12]. These algorithms often produce exceedingly large roadmaps [17], because random sampling adds many nodes and edges in unimportant workspace regions, particularly when asymptotically trying to approach optimality [18]. We assume a PRM-like algorithm to generate a very large baseline roadmap followed by a process to reduce it to fit in a given budget. Two previously studied approaches for roadmap reduction include graph spanners and edge contraction.

A graph spanner is a subgraph which contains all the vertices of the original graph, and that maintains the original connected components with a subset of the original graph's edges [21]. A t-spanner is a special class of spanner with the property that for all pairs of vertices (u, v), the shortest path between u and v in the spanner is no more than t-times the shortest path between u and v in the original graph [2]. Unfortunately the problem of finding the smallest t-spanner of a graph is NP-hard [6]. Several researchers have applied graph spanner algorithms to extract subgraphs of a more reasonable size from very large roadmaps [17], [18], [5], [3], [16], [4]. While these works have similar goals to our own, they only address environments in which obstacle locations are known a priori. Instead, we are focused on the case where the environment around the robot can change, so we cannot assume known obstacle locations.

Shaharabani et al. [23] observe that if a pair of connected vertices (u, v) share multiple neighbors in common, several edges can be removed by contracting the edge between u and v. As with the work on graph spanners, this work assumes an unchanging obstacle environment. This technique also relies on vertices having many shared neighbors in common in order to achieve high space reductions, which means the initial roadmap must not only be large but quite dense to realize significant compression gains.

III. Planning Roadmaps as Unreliable Graphs

Our goal is to start with a very large roadmap—one that provides good performance but is too large—and then reduce it to the best performing and most reliable roadmap that fits inside a given budget.

Abstractly, we have a graph (roadmap) where some of the edges may not be available during any given query (because the motion corresponding to an edge would collide with an obstacle present for that query). In graph theory, this is known as an *unreliable graph*. To the best of our knowledge, we are the first to leverage the similarities between roadmaps and unreliable graphs.

Our roadmap subsampling problem is closely related to the graph theory problem known as the Most Reliable Subgraph Problem (MRSP) [7]. The key to solving MRSP is identifying which connections are most important for the reliability of the system, and which ones can be discarded with minimal impact on reliability. System reliability is defined as the probability that a special set of vertices known as *terminal vertices* remain connected.

Formally, given an unreliable baseline graph $G = V, E$ and a set of terminals $T \subseteq V$, where each edge in E can fail with some probability, the goal of MRSP is to find the subgraph $H = V', E'$ with the properties that $V' \subseteq V$, $E' \subseteq E$, and $|E'| <= K$, where K is the edge budget. Lastly, it should maintain that $T \subseteq V'$, and it should maximize the probability that the terminals in T are connected [7]. This last property can be formalized by stating that for any other subgraph H' with at most K edges, $Rel(H) \geq Rel(H')$, where $Rel(G)$ is the reliability of graph G. This formulation is known as the k-terminal reliability problem; when $|T| = 2$ it is the two-terminal reliability problem, and when $|T| = |V|$ the all-terminal reliability problem.

There are two general approaches to solving MRSP: start with the complete baseline graph and prune it down, or start with a few useful paths from the baseline graph and incrementally add paths to it. We will use aspects of both approaches in our solution for roadmap construction.

A. Monte Carlo Pruning Algorithms

For the general case of a baseline graph with arbitrary topology, Hinstanen [7] provides a greedy polynomial-time heuristic for the two-terminal case, but with no performance guarantees. First, the two-terminal reliability of the original graph $G = V, E$ is estimated (since even determining two-terminal reliability is NP-hard). For each edge $e \in E$ there is an associated probability p_e that the edge has failed or not. Flipping a coin for each edge to determine its status comprises one trial. For each trial, the two terminals are checked for connectedness. Performing a number of trials gives an approximation of the reliability of the original graph. This process is then repeated for each edge e on $G' = V, E'$ where $E' = E \setminus e$. Any edge e whose removal does not impact reliability can be removed, since no acyclic path can be found between the terminals that uses e. After this step, the edge which affected reliability the least is removed. The process repeats, calculating the impact of removing each remaining edge and deleting the one with the least impact.

Although this algorithm is straightforward to implement, it has several deficiencies. Most importantly, it does not find the most reliable subgraph, and is not even guaranteed to find a good approximation of the optimal solution. It also

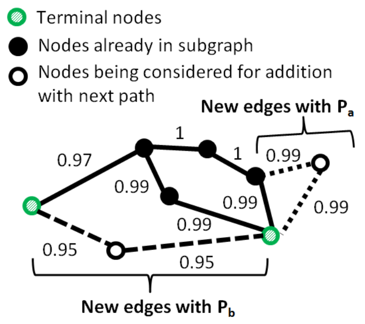

Fig. 1: Two possible paths that could be added to the subgraph in the next iteration. P_a has a higher path reliability, but does not add as much redundancy to the graph as P_b.

requires a very large number of Monte Carlo simulations. The complexity of the algorithm is $O(N|E|^2 + K(|E| + log|E|))$ where N is the number of trials needed during each iteration. For graphs with hundreds of thousands of vertices, N would need to be quite large.

B. Incremental Construction Algorithms

Another class of algorithms iteratively grows a subgraph by adding useful components from the baseline graph, instead of iteratively pruning. The "Best Paths Incremental" (BPI) algorithm tries to find the most probable/reliable paths between two terminals [8]. These paths are sorted in terms of reliability, and combined in decreasing order to build up a subgraph. Once the number of edges in the subgraph exceeds the desired edge budget, Monte-Carlo pruning can be used for several iterations to satisfy the roadmap size budget [8]. BPI can be relatively efficient because finding the most reliable paths between terminals is far easier than finding a most reliable subgraph. The probability p_e associated with each edge is simply transformed into weight $w_e = -\log(p_e)$. From here, there are many algorithms that can produce the k-shortest paths with a polynomial time complexity [22]. The BPI algorithm has a complexity of $O(K(k^2|V|^2 + k|V||E|)\log(k|V|))$, where k is the number of best paths needed to create a subgraph of desired size K.

Hintsanen et al. [9] propose a solution to the k-terminal version of MRSP using a framework like BPI as the inspiration. They note that BPI's main drawback is that each incremental path is greedily added based solely on its individual reliability, and not the effect adding the edge would have on the whole subgraph's reliability. An example of this distinction is shown in Figure 1. This figure shows that even though candidate path P_a has a reliability of 0.95 (calculated by $\prod_{e \in P_a} p(e)$), which is higher than the 0.90 reliability of P_b, P_b has a much higher effect on subgraph reliability since it adds more independent links.

The authors approximate the problem by dividing their solution into *path sampling* and *subgraph construction* phases [9]. Given a current set of paths C (which form a subgraph), the challenge in path sampling is finding the candidate path P that maximizes $Rel(C \cup P)$. They iterate over all the paths currently in the subgraph, producing "realizations" of its edges until the path fails. A realization is produced by flipping a coin for each edge and determining if it is available for the current query. The assumption of independence between edges is necessary for algorithmic efficiency but does not truly hold in motion planning roadmaps. Once all paths have failed, the best path from among the un-failed edges of the original graph is added. Subgraph construction involves selecting from among the sampled paths in C a subset of paths C' that meet the edge budget, while maximizing reliability [9]. Testing all possible path combinations is infeasible, so the authors use heuristics.

The main difference between the solutions for the two terminal and k-terminal problems is that instead of the first phase sampling candidate paths, it samples candidate spanning trees that connect all terminals [11]. To do this, first $(k^2 - k)/2$ candidate trees are initialized, each with the most reliable pairwise path between two of the query nodes.

IV. Applying Unreliable Graph Theory to Motion Planning Roadmaps

We divide roadmap generation into two phases.

1) Generating the baseline roadmap: We need an initial roadmap G large enough to serve as an ideal benchmark. The size depends on the application; challenging scenarios require more edges.
2) Finding the best sub-roadmap: The baseline roadmap G is likely too large for a practical roadmap budget. We seek to generate a roadmap from G that meets an edge budget K. Given G and K, the algorithm outputs a graph $G' = V', E'$ where $|E'| \leq K$.

A. Baseline Graph Generation

We experimented with two strategies to generate large baseline roadmaps. Both first model the static portion of the environment, so that computation is not wasted considering workspace regions that are always occupied.

The first strategy is to simply run a conventional sampling-based planner such as PRM or PRM* until the graph reaches a specified size. While this provides uniform coverage in configuration space, it may not equate to uniform coverage in the 3D workspace. Even worse, it cannot leverage semantic knowledge about the desired task, resulting in (a) many poses and motions in areas that are irrelevant and (b) insufficiently dense coverage in critical areas.

The second strategy is to create a roadmap (or augment a roadmap created using the first strategy) using knowledge of the robot's task. For example, if a robot is being installed to spot weld a part, and the part has known variability in its presenting location, then you can select an assortment of tool poses covering the expected range of part locations. The number of random samples required to achieve equivalent density in the same area would be enormous.

We compared the two strategies experimentally by generating two baseline graphs. One was generated by running the PRM algorithm until the roadmap contained 100,000 edges. The other first generated 250 nodes corresponding to grasp poses chosen above a table for a pick-and-place task, after which PRM was run until this graph also had 100,000 edges. From both baseline roadmaps we generated smaller roadmaps using the Monte Carlo pruning approach. Specifically, we generated 10,000 pick-and-place environments. For each of these environments, the roadmap was queried to find the shortest collision-free path to the goal. We pruned the roadmap by deleting edges that were never used or used infrequently. This process was done iteratively, profiling and pruning to create subgraphs of a range of sizes.

The resulting roadmaps were tested by querying each with 1,000 additional random scenarios. Results shown by Murray et al. [20] demonstrate that both strategies can extract effective subgraphs (greater than 90% query success rate) that have fewer than 1% of the original number of edges. However, for a given edge budget, the second strategy consistently achieves higher success rates, and we use it hereafter.

B. Extracting Reliable Subgraphs

Our strategy draws heavily on Hintsanen et al.'s work on the k-terminal subgraph problem [11], and we make several modifications to their algorithm in adapting it to the motion planning domain.

1) Estimating Edge Reliability: Let the baseline graph be $G = V, E$. Each edge $e \in E$ in the graph has an associated probability $0 \leq p_e \leq 1$.[1] We use a generator of sample environments, W, to estimate p_e for each edge by sampling obstacle sets (with $10,000$ samples). We collision check each $e \in E$ and compute the fraction of collisions. If W is an unbiased generator of obstacles, then this is an unbiased estimator of p_e.

2) Defining Source/Sink Terminal Nodes: One fundamental aspect where the motion planning problem differs from prior work on MRSP is in the definition of terminal nodes. Prior work considers only a single class of terminals, but motion planning often has terminals that are sources or sinks.

Having multiple terminal types makes the problem much more tractable. When considering only a single set R of terminal nodes, $R \subset V, \quad k = |R|$, the number of pairwise reliabilities the algorithm must maximize is $\binom{k}{2}$. In motion planning, we may have a single source terminal but require 500 sink terminals to handle variation in goal location. With only a single class of terminals, this would require over 100,000 pairwise combinations. However, we do not need to maximize reliability between sink terminals, and only need to maintain $source \rightarrow sink$ reliability. This can be achieved with only 500 pairwise combinations. We assume two sets of terminal nodes S and D, with the following invariants:

- $S \subset V$ and $D \subset V$
- $S \cap D = \varnothing$ and $S \cup D = R$
- When all edges in G are present, then for each pairwise combination u, v, where $u \in S$ and $v \in D$, a path exists from u to v through G

[1]In Hintsanen's work, these probabilities represent uncertainties that different genes are connected. Here, they are each edge's likelihood of not being in collision for a given query.

3) Sampling Candidate Trees: A major difference between our application and Hintsanen is we must consider path quality in addition to reliability. Each edge now has an associated weight or cost reflecting how long it takes the robot to traverse that edge. We accommodate this additional optimization dimension by modifying the algorithm to search for shortest paths using different cost functions at different phases of the algorithm. While Hintsanen considered only $Rel(e) = -log(p_e)$, we introduce $Dist(e)$ which is the cost to traverse the edge.

As seen in Algorithm 1 lines 1 to 3, initialization occurs in the same way as Hintsanen [11], except that instead of starting a new tree for every terminal pair, we include only pairs from source terminals to sink terminals. For each pair, the most reliable path is used to initialize a tree. We then enter the main loop of the tree sampling phase, which terminates once we have built a specified number of "complete" trees. A tree is complete if it contains all the source and sink terminals.

In each iteration of the sampling phase, we first produce a realization of the uncertain baseline graph, and label each edge as available or failed for this iteration (line 6). After producing a realization, we search for an "intact" tree to extend (line 8). A tree is intact if all its edges lie in the available set (E_a) in this realization. Extending only intact trees biases the algorithm to produce robust trees.

The process of extending a tree is in Algorithm 2. If there are source terminals absent in the tree, then a random source terminal out of the missing set is chosen, along with a random destination terminal out of the set already present in the tree; if all source terminals are already in the tree, then one is randomly chosen along with one of the absent destination terminals (lines 1 to 5). When extending trees, we select best paths among the available set of edges using their distance-based cost function, instead of reliability. Even though we do not consider reliability, the resultant subgraphs still have high reliability. (Due to the stochasticity of the algorithm, only edges with high reliability are often in the available set.) Before searching for a best path, we temporarily set the cost of each edge that is *already* in the tree being extended to zero. This biases the algorithm to extend the tree using paths that add fewer edges.

One major modification we make is introducing epochs to the sampling phase. Because the average reliability of our edges is significantly higher than those investigated by Hintsanen, only a subset of edges were being added to many candidate trees. There were slightly more costly alternatives that would have added useful redundancy. We counter this effect by allowing edges from the baseline graph to be used in extensions once per epoch. Once used, they are temporarily removed from the baseline graph (line 14). If an extension fails to find a path through the reduced graph in a given realization, then the epoch is reset, and all original edges are restored to the baseline graph (line 11). This change to the algorithm increases the number of unique edges in complete trees by 18%.

If no trees are intact, Hintsanen initializes a new tree with a path that is intact in the current realization. However, since we have many more terminals than Hintsanen, our trees are much larger (in the thousands of edges), and so there is much less chance that a tree is intact in a given realization. We also begin the algorithm with many more trees, so there is less need to start new trees. Thus, when we have no intact trees, instead of creating new trees, we "repair" the oldest incomplete tree. If the oldest incomplete tree still has paths between all its source and destination terminals, we consider it already repaired and extend it. Otherwise, we find paths through the available edges from the source to sink terminals, add the corresponding edges to the tree, and extend it. Note that after being "repaired", the growing subgraphs are no longer trees, since additional paths between disconnected terminals have been added. For consistency with the original body of work by Hintsanen, we continue to refer to the algorithm steps as ExtendTree, SelectTrees, etc.

After extending a tree, we check if it contains all the terminals from the original baseline graph. If so, the tree is considered complete and moved to the $CompleteTrees$ set. Once this set reaches the size specified as input, the incomplete trees are discarded, and the complete trees are used in the next phase of the algorithm. We swept the parameter space and found no additional benefit in sampling more than $|S| * |D|$ complete trees.

4) Selecting Trees to Construct a Subgraph: The next step is to select a subset of the trees sampled. This phase of the algorithm takes as input an edge budget along with the collection of complete trees sampled, and outputs a subgraph. Hintsanen et al. [11] focus only on maximizing reliability, so we present a modified algorithm that we find maintains reliability while also selecting trees with high-quality paths.

We initialize the subgraph with the tree that contains the fewest number of unique edges. Because each candidate tree is complete, the subgraph begins already having (unreliable) connections to every terminal. During each iteration of the selection phase, we produce a realization of the edges from the baseline graph. We then search through the remaining candidate trees to find the one that maximizes an incremental quality metric. During this search, we also remove any candidate tree that has no unique edges compared to the growing subgraph (lines 5 to 7). The quality metric is calculated by iterating over each pair of source and destination terminals. The cost of the best path between them in this realization is calculated both through the current tree as well as the subgraph. If the current tree offers a cost improvement in the cost of the current path, the amount of the improvement is added to a running sum. The total improvement in all the paths is divided by the number of unique additional edges this tree would bring to the subgraph if they were merged.

After finding the tree that maximizes this quality metric, we remove it from the set of complete trees, and add it to the subgraph. The algorithm terminates when either the edge budget has been exceeded, or the set of complete trees has been exhausted. If the latter occurs, the sampling phase can be re-run with tuned inputs intended to provide a larger number of candidate trees and unique edges.

Algorithm 1 SampleTrees

```
Let E(G) provide edges of G, and V(G) vertices of G, whether G is a path, tree, or graph.
Let E_a, E_f ← RealizeEdges(E) indicate producing a possible world by drawing from a uniform distribution for each edge, placing failed links in E_f and available links in E_a.
Let P ← PathSearch(E, u, v, func()) indicate searching for shortest u → v path through edges E using cost function as metric for search, placing resultant path in P.
Let E ← ResetEpoch() indicate restoring all original edges from baseline graph to E.
Input: Baseline graph G = (V, E), Source terminals S, Sink terminals D, Cost function Rel() that considers reliability of each edge, Cost function Dist() that considers cost to traverse each edge, Number of complete trees to sample N.
Initialize: $Trees \leftarrow \emptyset$, $CompleteTrees \leftarrow \emptyset$
1:  for each pair of terminals $< u, v >$ where $u \in S, v \in D$ do
2:      $P \leftarrow$ PathSearch$(E, u, v, Rel())$
3:      Add $P$ as a new candidate tree to $Trees$
4:  end for
5:  while $|CompleteTrees| < N$ and $|Trees| > 0$ do
6:      $E_a, E_f \leftarrow$ RealizeEdges$(E)$
7:      for each $T \in Trees$ do
8:          if $\mathbf{E}(T) \subset E_a$ then
9:              ExtendTree$(T, S, D, E, E_a)$
10:             if $S \subset \mathbf{V}(T)$ and $D \subset \mathbf{V}(T)$ then
11:                 remove $T$ from $Trees$ and place in $CompleteTrees$
12:             end if
13:             continue at line 5
14:         end if
15:     end for
16:     RepairTree$(Trees[0], S, D, E, E_a)$
17:     ExtendTree$(Trees[0], S, D, E, E_a)$
18:     if $S \subset V(Trees[0])$ and $D \subset V(Trees[0])$ then
19:         remove $Trees[0]$ from $Trees$ and place in $CompleteTrees$
20:     end if
21: end while
22: return $CompleteTrees$
```

V. Benchmarks and Experimental Evaluation

One of the challenges in evaluating planning algorithms is a well-known lack of standard benchmarks [15], [1]. We have developed four distinct scenarios for evaluating planning algorithms. Each one uses a different robot to accomplish a different task. We parameterize each scenario and provide a random environment generator so that users can produce unique training and test sets for each scenario. These environment generators have been bundled as a ROS package for distribution to the robotics community[2].

[2]http://irl.cs.brown.edu/mplan_benchmarks

Algorithm 2 ExtendTree

```
Input: $T, S, D, E, E_a$
1:  $S' \leftarrow S \setminus \mathbf{V}(T)$
2:  if $|S'| > 0$ then
3:      Randomly select $u \in S'$ and $v \in D \cap \mathbf{V}(T)$
4:  else
5:      Randomly select $u \in S$ and $v \in D \setminus \mathbf{V}(T)$
6:  end if
7:  Set $Dist(e)$ to 0 for all $e \in \mathbf{E}(T)$
8:  $P \leftarrow$ PathSearch$(E_a, u, v, Dist())$
9:  Restore original $Dist(e)$ for all $e \in \mathbf{E}(T)$
10: if $P == \emptyset$ then
11:     $E \leftarrow$ ResetEpoch$()$
12: else
13:     Add the nodes and edges in path $P$ to $T$
14:     $E \leftarrow E \setminus \mathbf{E}(P)$
15: end if
```

Algorithm 3 RepairTree

```
Input: $T, S, D, E, E_a$
1:  for each pair of terminals $< u, v >$ where $u \in S \cap \mathbf{V}(T), v \in D \cap \mathbf{V}(T)$ do
2:      if PathSearch($E_a \cap \mathbf{E}(T)$, u, v, Dist()) == $\emptyset$ then
3:          $P \leftarrow$ PathSearch$(E_a, u, v, Dist())$
4:          if $P == \emptyset$ then
5:              $E \leftarrow$ ResetEpoch$()$
6:              continue at line 1
7:          end if
8:          Add the nodes and edges in path $P$ to $T$
9:          $E \leftarrow E \setminus E(P)$
10:         $E_a \leftarrow E_a \setminus E(P)$
11:     end if
12: end for
```

- Machine tending: A UR5 robot reaches into a machine and places stock at a randomly chosen location. Success is defined as bringing the work part to within 1 cm and 5 degrees of the indicated location and orientation.
- Dish grasping: A Jaco robot reaches into a dishwasher to grasp a plate. The robot is randomly placed along any of the three sides of the dishwasher, and one plate is randomly chosen as the goal plate. Success is defined as the palm of the robot facing any part of the rim of the goal plate, with the palm 0.5-4 cm from the rim.
- Shelf placement: A Fetch robot places a grasped can on a cluttered shelf. Success is defined as bringing the can to within 2 cm of the goal, oriented vertically.
- Plug insertion: A UR3 robot inserts a grasped plug into a power strip that is randomly populated with other plugs. Success is defined as bringing the grasped plug to 1-3 cm above the goal outlet.

We created a baseline roadmap for each benchmark, generated 10,000 environments to produce data for running the algorithm, and a separate 1,000 environments for testing. We defined a subset of the nodes in each baseline roadmap

Algorithm 4 SelectTrees

```
Let $P, C \leftarrow$ **PathSearch**$(E, u, v, func())$ indicate searching for shortest $u \rightarrow v$ path through edges $E$ using specified cost function as metric for search, and placing resultant path in $P$, the cost of shortest path in $C$
Input: $CompleteTrees$, $E$, $S$, $D$, $Dist()$, Edge Budget $K$
Initialize: $SubGraph$ with the tree in $CompleteTrees$ with fewest edges
1: while $|E(Subgraph)| < K$ and $|CompleteTrees| > 0$ do
2:     $E_a, E_f \leftarrow$ RealizeEdges$(E)$
3:     $BestScore \leftarrow 0$, $BestTree \leftarrow \emptyset$
4:     for each $T \in CompleteTrees$ do
5:         if $E(T) \subset E(Subgraph)$ then
6:             Remove $T$ from $CompleteTrees$
7:             continue on line 4
8:         end if
9:         $sum \leftarrow 0$
10:        for each pair of terminals $< u, v >$ where $u \in S, v \in D$ do
11:            $P, C \leftarrow$ PathSearch$(E(T) \cap E_a, u, v, Dist())$
12:            $P', C' \leftarrow$ PathSearch$(E(SubGraph) \cap E_a, u, v, Dist())$
13:            if $(C' - C) > 0$ then
14:                $sum \leftarrow sum + (C' - C)$
15:            end if
16:        end for
17:        $score \leftarrow sum / |E(T) \setminus E(Subgraph)|$
18:        if $score > BestScore$ then
19:            $BestScore \leftarrow score$
20:            $BestTree \leftarrow T$
21:        end if
22:    end for
23:    remove $BestTree$ from $CompleteTrees$ and add its edges and nodes to $SubGraph$
24: end while
25: return $SubGraph$
```

to act as source and sink terminals. We performed collision detection on the 10,000 training environments to calculate a reliability for each edge. The results are in Figure 2.

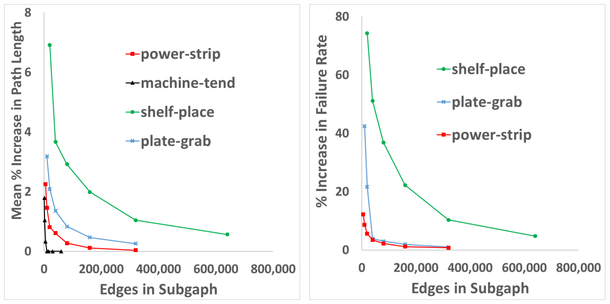

Fig. 2: Path Length (left) and Failure Rate (right)

The *plate-grab* benchmark has a baseline roadmap of 2 million edges, 5,000 terminals, and a 78% success rate, and it is well-suited for our algorithm. The open space above the dishwasher means that there is an area that can be pruned without damaging path feasibility. We only see dramatic increases in failure rates when the edge budget decreases to the point where the "approach edges" that enable direct connections to terminal nodes begin to be excluded from the subgraph. This increases the likelihood that terminal nodes will be bisected by the other clutter in the dishwasher and decreases success rate. Effects on path quality remain modest until reducing roadmap size below 40,000.

The *shelf-place* benchmark was quite challenging, because often the goal location was behind or very close to other objects on the shelves. The baseline roadmap of 5 million edges (11,000 terminals) had only a 64% success rate. This benchmark also suffered the most from increases in path failure rate. Trying to have a single roadmap cover the entire shelf assembly may be infeasible. Hardware accelerators may benefit from the ability to be reprogrammed with different roadmaps; using this feature, we could divide the shelf assembly into 10-15 regions, and program an accelerator for the specific region relevant for the query [19].

The *power-strip* benchmark was relatively easy. The baseline roadmap of 2,000,000 edges (3,000 terminals) was successful for just over 95% of the test set, and even sampling a subgraph of just 20,000 edges from the baseline had less than a 6% effect on failure rate (an increase to 5.3% from 5%), and an average path length increase of less than 5%.

The outlier benchmark was *machine-tend*. The UR5 is not robust to inserting obstacles in between it and the opening to the machine. For this reason, the machine-tend scenario is the only one without dynamic obstacles, having only the goal location vary between queries, and thus the effective reliability of all edges in the baseline graph (100,000 edges and 400 terminals) is 1. The subgraph problem then reduces to simply finding subgraphs with the fewest number of edges that contain the highest quality paths between terminals. Path quality is unchanged until we decrease the edge budget below the number of unique edges in the union of the shortest paths between each pair of source/sink terminals.

VI. Conclusions

We have presented a new algorithm for creating fixed roadmaps for multiple motion planning queries. The algorithm starts with a large baseline roadmap and subsamples it to meet a given budget while finding high quality paths with high probability. Experimental results on a newly developed benchmark suite show that the roadmaps can achieve results similar to the baseline at greatly reduced size.

Acknowledgements

The authors would like to thank Rebecca McCabe for her assistance creating example robot scenarios. This research was supported in part by DARPA under agreement number D15AP00104. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The content is solely the responsibility of the authors and does not necessarily represent the official views of DARPA. Disclosure: George Konidaris is the Chief Roboticist, and Dan Sorin the Chief Architect of Realtime Robotics, a robotics company that produces a specialized motion planning processor.

References

[1] G. Antonelli, "Robotic research: Are we applying the scientific method?" *Frontiers in Robotics and AI*, vol. 2, p. 13, 2015.

[2] E. Cohen, "Fast algorithms for constructing t-spanners and paths with stretch t," *SIAM Journal on Computing*, vol. 28, no. 1, pp. 210–236, 1998.

[3] A. Dobson and K. E. Bekris, "Improving sparse roadmap spanners," in *2013 IEEE International Conference on Robotics and Automation*.

[4] ——, "Sparse roadmap spanners for asymptotically near-optimal motion planning," *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 18–47, 2014.

[5] A. Dobson, A. Krontiris, and K. E. Bekris, "Sparse roadmap spanners," in *Algorithmic Foundations of Robotics*, 2013, pp. 279–296.

[6] M. Elkin and D. Peleg, "Strong inapproximability of the basic k-spanner problem," in *International Colloquium on Automata, Languages and Programming*, 2000, pp. 636–648.

[7] P. Hintsanen, "The most reliable subgraph problem," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2007, pp. 471–478.

[8] P. Hintsanen and H. Toivonen, "Finding reliable subgraphs from large probabilistic graphs," *Data Mining and Knowledge Discovery*, vol. 17, no. 1, pp. 3–23, 2008.

[9] P. Hintsanen, H. Toivonen, and P. Sevon, "Fast discovery of reliable subnetworks," in *Advances in Social Networks Analysis and Mining, 2010*. IEEE, 2010, pp. 104–111.

[10] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[11] M. Kasari, H. Toivonen, and P. Hintsanen, "Fast discovery of reliable k-terminal subgraphs," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2010, pp. 168–177.

[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] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Technical Report, Tech. Rep., 1998.

[14] P. Leven and S. Hutchinson, "A framework for real-time path planning in changing environments," *The International Journal of Robotics Research*, vol. 21, no. 12, pp. 999–1030, 2002.

[15] R. Madhavan, E. W. Tunstel, and E. R. Messina, *Performance evaluation and benchmarking of intelligent systems*. Springer, 2009.

[16] J. D. Marble and K. E. Bekris, "Towards small asymptotically near-optimal roadmaps," in *2012 IEEE International Conference on Robotics and Automation*, pp. 2557–2562.

[17] ——, "Computing spanners of asymptotically optimal probabilistic roadmaps," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 4292–4298.

[18] ——, "Asymptotically near-optimal is good enough for motion planning," in *2011 International Symposium on Robotics Research*, pp. 419–436.

[19] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, "A programmable architecture for robot motion planning acceleration," in *2019 IEEE International Conference on Application-Specific Systems, Architectures and Processors*.

[20] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, "Robot motion planning on a chip," in *Robotics: Science and Systems*, 2016.

[21] D. Peleg and A. A. Schäffer, "Graph spanners," *Journal of graph theory*, vol. 13, pp. 99–116, 1989.

[22] L. Roditty, "On the k-simple shortest paths problem in weighted directed graphs," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[23] O. Salzman, D. Shaharabani, P. K. Agarwal, and D. Halperin, "Sparsification of motion-planning roadmaps by edge contraction," *The International Journal of Robotics Research*, vol. 33, pp. 1711–1725, 2014.