A Roadmap-based Planner for Fast Collision-free Motion in Changing Environments

by

Matthew Ralph Orton

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by ....
Andreas Hofmann
Research Scientist
Thesis Supervisor

Certified by ....
Brian Williams
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by ....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Roadmap-based Planner for Fast Collision-free Motion in Changing Environments

by

Matthew Ralph Orton

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the development of a roadmap-based planner to enable high-DOF robotic arms to accomplish tasks based around motion planning problems with motions that feel reactive and intuitive in changing environments. My approach to accomplish this is to combine a roadmap-based motion planner with a sequential, convex trajectory optimization library called TrajOpt. The roadmap is used to produce collision-free seed trajectories, which are then provided to TrajOpt for optimization based on path length and proximity to obstacles. The difficulty of this approach arises from how to quickly update the roadmap as the environment changes to ensure that the seed trajectory provided to TrajOpt is always collision-free. This difficulty is addressed with a few different innovations. The roadmaps used by this planner are relatively sparse, so they are faster to update and perform searches on. Next, the sparse roadmaps are constructed offline along wih a cache of shortest path solutions to minimize online search requirements. These solution caches are combined with an iterative search algorithm based around A* search with lazy collision checking. Finally, an adaptation of an incremental search algorithm, D* Lite, is developed to take advantage of the full environment knowledge assumed by my motion planner and the rapid optimization provided by TrajOpt while utilizing a lazier collision checking approach than the original algorithm.

Thesis Supervisor: Andreas Hofmann
Title: Research Scientist

Thesis Supervisor: Brian Williams
Title: Professor of Aeronautics and Astronautics

Acknowledgments

I would like to start out thanking everyone in the Model-based Embedded Robotic Systems Group (MERS) of the MIT Computer Science and Artificial Intelligence Laboratory. Every member of the lab is both motivated and kind, and as a collective, the members of the lab create a welcoming and productive research environment. In particular I would like to thank Sylvia Dai, Steve Levine, Shawn Schaffert, and Andreas Hofmann of the Humanoids Team in MERS. Sylvia developed testing environments that are used for every experiment in this thesis. Steve is incredibly friendly and patient, and I could not have integrated into the lab as quickly as I did without frequent guidance by Steve. Shawn is an invaluable mentor who oversaw the bulk of the software development I did as a member of the Humanoids Team. He provided great insight about what building a robust robotic software stack entails and how I could shape my work in that pursuit. Finally, I would not be here were it not for Andreas Hofmann. Despite my lack of experience developing robotic software and algorithms, Andreas saw enough else in me to see fit giving me a research position under his supervision. Along the way, he consistently pushed me to hold my work to a higher standard of scientific rigor and helped me distinguish between engineering and research problems. There are others not mentioned here who definitely contributed to my time in MERS and MIT as a whole, so I would like to conclude by giving thanks one last time to everyone who has made an impact on me, my research, and my education over my six years at MIT.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **17**

1.1 Motivation 17

1.2 Related Work 19

1.3 Problem Statement 22

1.4 Approach 23

1.4.1 Roadmap Framework 25

1.4.2 Offline Algorithms 27

1.4.3 Online Algorithms 27

1.5 Introduction Summary 28

2 Methods Developed **29**

2.1 An Interface for Interaction with Real and Simulated Robots 31

2.1.1 Development Environment 31

2.1.2 Pick and Place Demonstrations 32

2.2 A Roadmap Based Motion Planner with Shortest Path Solution Cache 34

2.2.1 Software Modules 34

2.2.2 Roadmap and Cache Construction 35

2.3 A Framework For Providing Collision Free Trajectories in Static Environments 37

2.4 Using Semantic Information to Extend the Ability of the Motion Planning Framework 38

2.5 Offline All-Pairs Shortest Path Strategies for Avoiding Dynamic Obstacles 40

2.5.1 Simple Strategy . . . 40
2.5.2 Realistic Obstacles . . . 42
2.5.3 Improved APSP Solution Cache . . . 43
2.6 Online Single Source Shortest Path Strategies for Avoiding Dynamic Obstacles . . . 44
2.7 An Incremental Search Strategy for Avoiding Dynamic Obstacles . . . 49

3 Experiment Plan 59
3.1 Description of the Robot and Testing Environments . . . 61
3.2 Development of and Characterization the Roadmap Framework . . . 64
3.2.1 Tuning Roadmap Hyper-Parameters . . . 64
3.2.2 Characterizing Roadmap Performance . . . 66
3.3 Semantic Sampling to Improve Roadmap Connectivity and Other Side Explorations . . . 67
3.4 Incorporating Dynamic Obstacles to Augment and Evaluate Roadmap Solution Caches . . . 69
3.5 Creating Experiments to Expose the Benefits of Incremental Planning 71

4 Experiment Results 75
4.1 Roadmap and TrajOpt Performance . . . 75
4.2 Semantic Sampling and Sorting Heuristic . . . 80
4.3 Training and Testing Results for Obstacle Insertion Experiments . . . 88
4.3.1 APSP Training Results . . . 88
4.3.2 Base Roadmap Results for A* Search . . . 92
4.3.3 Training Comparison . . . 95
4.3.4 A* Repair Results . . . 97
4.4 Performance Comparison for Different Incremental Execution Implementations . . . 102

5 Discussion 107
5.1 Looking Forward . . . 107

5.2 Revisiting the Problem Statement 111

A Additional Figures, Tables, and Graphs **113**

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

1-2 A basic outline of how a roadmap is constructed. The static environment is showin in (a). Roadmap nodes are randomly sampled in (b) and then nodes are checked for collisions in (c). Nodes in collision are pruned from the roadmap. Edges are generated between nearby nodes in (d) and then checked for collisions in (e). The final roadmap is shown in (f) after edges in collision have been removed. 24

2-1 The Barrett Whole-Arm Manipulator (WAM) shown in a ROS visualization of our hardware test-bed for pick and place tasks. 33

2-2 Yen's algorithm 37

2-3 A visualization of the end-effector poses within the shelf that were tested for valid IK solutions to add to the roadmap 40

2-5 Image (a) shows a roadamp with $p = 2$ shortest paths for a pair of start and points. Image (b) shows how a single obstacle can invalidate both of those paths while a valid shortest path exists in the roadmap (shown in red) but not in the solution cache. 41

2-6 The five obstacles used to obstruct the robot in ways that are representative of the environment 43

2-9 The four versions of A* Repair and helper functions. CheckPathCollisions checks the path for collisions, obtain pairs of nodes surrounding in-collision edges, and updates *validEdges* accordingly. 48

2-11 Illustration of a standard D* Lite implementation. A path has been found through initial search in (a). The robot begins to execute the trajectory in (b) and (c), and in (d), an obstacle is discovered inside the visibility range. In response, all affected edges are updated and a new plan is formed from the current state of the robot. The new trajectory is executed to to the goal in (e) and (f). 50

2-13 Illustration of Adapted D* Lite. A path has been found through initial search in (a) and is then optimized with TrajOpt in (b). The robot begins to execute the trajectory in (c), and in (d), an obstacle is discovered that invalidates the current trajectory. In response the roadmap checks edges near the colliding edge in (e) and forms a new plan from the current state of the robot in (f). The new trajectory is optimized in (g) and is executed to to the goal in (h) and (i). 54

2-15 The standard D* Lite algorithm 55

2-17 Adapted D* Lite main and helper functions 57

2-18 A* Repair with incremental execution and execution monitoring. Get-CollisionFreePath can be any of the four versions of A* Repair. . . . 58

3-1 Rethink Robotics Baxter 62

3-3 The four testing environments used for all experiments 63

3-4 Environment visualization of Tabletop with a Pole for incremental planning analysis 73

4-1 Roadmap performance assuming a static environment. No collision checks were performed on roadmap edges as a result of the assumption. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. 77

4-2 TrajOpt performance when seeded with roadmap trajectories. TrajOpt is provided a collision-free seed from the roadmap in all cases and a static environment is assumed with no collisions in the roadmap. . . . 78

4-3 Graphs showing the difference between a seed trajectory from a roadmap and an optimized trajectory from TrajOpt for three different cases. Values are only shown for the first 4 DOFs because the roadmap nodes have the same fixed values for the remaining DOFs. In each plot, the roadmap trajectory is shown in red and the optimized trajectory is shown in blue. TrajOpt is provided a pose target rather than a joint target for these experiments, so the end joint state for an optimized trajectory may differ from the corresponding roadmap trajectory. . . 79

4-4 Performance comparison between roadmaps with different sets additional points added to the same base 1000 node roadmap. Only modifications that add points specific to the shelf support queries where "shelf" is provided to guide attempted connections to the roadmap. Roadmap paths are checked for collisions before they are returned although the static environment will not push any roadmap edges into collision. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. 81

4-6 Roadmap performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap. For each roadmap, the number of cases that had a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. 85

4-8 TrajOpt performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap 87

4-9 APSP Training obstacle avoidance graphs. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. See the Appendix for for a breakdown of the graphs by obstacle 91

4-11 Performance comparison in a static environment examining the impact of checking the roadmap for collisions during A* search versus assuming all roadmap edges are collision-free. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. 94
4-14 A* Repair obstacle avoidance graphs. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. See the Appendix for for a breakdown of the graphs by obstacle. 101
4-15 Performance comparison of two incremental algorithms with both overlapping and non-overlapping (serial) replanning and execution. One based around D* Lite and the other based around A* Repair. Both heavily rely on lazy collision checking and have the roadmap solutions provided to TrajOpt before execution. 106

5-2 Scenario illustrating the shortcomings of D* Lite when an obstacle is detected near the goal. A collision-free path is shown for the static environment in (a). In (b), an obstacle is detected near the goal and updates are required for seven nodes in addition to the current state before computing a new shortest path. For contrast, the obstacle in (c) is detected closer to the current state and only updates are only required for two nodes in addition to the current state. Nodes that must be updated are shown in orange and the edges are in yellow that provide the shortest path to the goal for any one of these nodes. . . . 108

List of Tables

4.1 Roadmap performance assuming a static environment 77

4.3 TrajOpt performance when seeded with roadmap trajectories 78

4.5 Performance comparison between roadmaps with different sets additional points added to the same base 1000 node roadmap 81

4.6 Roadmap performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap 84

4.7 TrajOpt performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap 86

4.8 APSP Training obstacle avoidance data for 500 node roadmaps. Original Cached Solution Collides serves as a validation of the experiment because the same roadmap is used for every test in a given environment but with different solution caches. RRT Comparison describes the use of RRT in OMPL with 100 maximum iterations to find a solution when the original roadmap solution is in collision. Some cases were lost for the RRT Comparison due to non-deterministic failures with the OMPL planner. 90

4.9 A* search performance in a static environment examining the impact of checking roadmap edges for collisions when they are expanded during search. 93

4.10 Additional paths for solution caches developed with APSP Training . 96

4.11 Overview of solution caches developed with A* Training. Paths Removed is an estimate determined by tracking whenever a new solution is created for a pair of nodes that already have the maximum number of allowed solutions. 96
4.12 A* Repair obstacle avoidance data for 500 node roadmaps. The Control Roadmap results are identical to the APSP Training Control Roadmap results. 100
4.13 Performance comparison of two incremental algorithms with both overlapping and non-overlapping (serial) replanning and execution. Effective Planning Time is the difference between the Replanning and Execution Time metric and the Execution Time Only metric. 105

Chapter 1

Introduction

1.1 Motivation

Imagine a child working to assemble a LEGO model from instructions with help from a robotic arm on a stationary base. The child and robot are both following instructions that require finding one or more particular LEGO pieces in a pile at a step in the instructions. While the child is assembling pieces from the previous step, the arm can attempt to find and grab a piece required by the next step. When multiple pieces are required the robot can identify what piece the child is reaching for and reach for a different required piece.

This scenario requires that the motion planner for the robot arm plans its motions quickly enough that they are still relevant in the context of what the child is doing when the motions are being executed. It requires that motion plans can be terminated or modified during execution to account for changes in task-level goals or to avoid children as they move in and out of the path of the robot after the initial planning. This can be summarized to say the motion planner needs to be reactive.

Additionally, this scenario requires that the motion plans produced for the arm are intuitive. Intuitive motions from the standpoint of the child in this scenario are smooth and direct so the child can react to what it believes the robot is trying to accomplish with any given motion. In the context of motion planning, this means the motions are near-optimal.

Finally, it requires coordination with a task-level executive to receive motion planning goals that will move the robot towards accomplishing human understandable tasks such as picking up a LEGO block in the context of completing a larger plan such as a full LEGO instruction set, while accounting for any additional constraints that are required.

Other areas of robotics deal with similar types of problems. Today, many companies are heavily investing in the development of robotic solutions for product manufacturing. On most modern factory floors, you will see a variety of robots, each performing a set of well-defined tasks. However, these robot-operated manufacturing areas are often sectioned off with physical barriers to ensure human safety. These robots tend to repeatedly execute a single sequence of actions without any awareness of what is happening around them.

Different manufacturing robots, like those that transport material from one area of the factory to another, have more variability in terms of the tasks they are given which in turn require certain sensing capabilities and higher level decision-making than just controlling arm motion along a fixed trajectory. They can be given a task, create a full motion plan, and even pause execution and modify plans in the face of obstacles. However, robots in this scenario are supported by heavily constrained sets of possible motions, environmental factors, and tasks to execute. For different kinds of motion planning problems such as the previously described LEGO scenario, these existing motion planning and execution systems are not fast enough to react to dynamic environments.

My goal is to enable high-DOF robotic arms to accomplish tasks based around motion planning problems with motions that feel reactive and intuitive in changing environments. My approach to accomplish this is to combine a roadmap-based motion planner with a sequential, convex trajectory optimization library called TrajOpt. The roadmap is used to produce collision-free seed trajectories, which are then provided to TrajOpt for optimization based on path length and proximity to obstacles. The difficulty of this approach arises from how to quickly update the roadmap as the environment changes to ensure that the seed trajectory provided to TrajOpt is always

collision-free.

I address this difficulty with a few different innovations. The roadmaps I am constructing are relatively sparse, so they are faster to update and perform searches on. This makes the seed trajectories coarse relative to those produced by denser roadmaps, but this is offset by trajectory optimization. Next, I am precomputing a cache of shortest path solutions to minimize online search requirements. These solution caches are built to include multiple useful path solutions in addition to the all-pairs shortest path(APSP) solution set for the roadmap. The precomputed solution caches are combined with an iterative search algorithm based around A* search with a heavy reliance on lazy collision checking. This algorithm prevents repeating any collision checks performed checking cached solutions or those returned by previous iterations of the A* search. Finally, I have adapted an incremental search algorithm, D* Lite, to take advantage of the full environment knowledge assumed by the motion planner and the rapid optimization provided by TrajOpt while utilizing a lazier collision checking approach than the original algorithm.

1.2 Related Work

Most existing sampling-based motion planners plan from scratch for every problem, during what is referred to as online motion planning. A very popular example of one of these online planners is Rapidly-exploring Random Trees (RRT) [1]. This algorithm randomly samples robot states in an attempt to build a tree of states from the current state to a known goal state. It is popular in large part due to its ease of implementation and its ability to be adapted to a wide variety of motion planning problems. In a sense, online planners are making the assumption that the environment is static because every plan is for a particular static snapshot of the environment. This assumption leads to having to fully replan whenever the current plan becomes infeasible due to changes in the environment. For example, if the child in our LEGO scenario reaches for a brick near the brick that the robot arm is moving towards, the plan would be invalidated by the child colliding with a future state.

The robot arm would then stop and replan without any knowledge retained from the previous plan. For high-DOF manipulators in particular, these planners struggle with trade-offs between the planning speed and the optimality the plan returned.

Some existing sampling-based planners do save information between planning problems or store information during a prior offline planning phase. Many of these offline planners are variants of the Probabilistic Roadmap (PRM) algorithm [2]. Depending on the problem scenario, this algorithm can be adapted in a large number of ways to construct roadmaps that are optimized for certain criteria. These adaptations are often made to adhere to restrictions in terms of observability of the environment, computation time, and memory. Since offline computation is free other than memory consumed, computation time in this case refers to what is required to search the roadmaps during online motion planning and to update roadmaps for changing environments.

A distinct class of motion planners are optimization-based motion planners. Optimization-based robotic motion planners have become more popular in recent years in large part due to the increased complexity of robots and environments. Covariance Hamiltonian Optimization for Motion Planning (CHOMP) [3], [4], Stochastic Trajectory Optimization for Motion Planning (STOMP) [5], Incremental Trajectory Optimization for Real-time Replanning (ITOMP) [6] and TrajOpt [7], [8] are several state-of-the-art optimization-based planners. Our group has chosen to focus on TrajOpt for three reasons. First, the non-convex collision checking method used in TrajOpt can take accurate object geometry into consideration to enhance the ability of getting trajectories out of collision. In contrast, the distance field method used in CHOMP and STOMP consider the collision cost for each exterior point on a robot, which means two points might drive the objective in opposite direction. Second, the sequential quadratic programming method used in TrajOpt can better handle deeply infeasible initial trajectories than the commonly used gradient descent method [7]. Third, customized differential constraints, for example velocity constraints and torque constraints, can be incorporated in TrajOpt. Since the roadmap-based planner is purely kinematic and always produces a collision-free seed trajectory, these last two points

are less important for the work specific to this thesis. However, they are very important for other research conducted within my group using the same motion planner.

What is important about this class of planners is they can rapidly produce near-optimal trajectories that avoid environment collisions, but they are very dependent on the quality of the seed trajectories they are provided. In particular, they can struggle to produce a collision-free trajectory when the seed trajectory they are provided is in collision [9]. Our goal is to provide an optimization based planner a collision-free seed from a roadmap, so it can be quickly optimized while remaining collision-free.

There are existing approaches for rapidly validating whether or not a robot configuration is in collision that both do and do not make use of workspace representations of the environment. Leven and Hutchinson have established a framework for connecting a workspace voxelization to a configuration space roadmap for rapid collision checking [10]. Such systems are heavily dependent on an efficient 3D cell decomposition of the environment, but this can be enabled by existing open-source GPU voxel libraries [11]. Implementing such a system is not a focus of this research, so collision checking will be an accepted bottleneck for this system. Our system is instead focused on efficiently combining that collision information with precomputed shortest path solutions and state-of-the-art search algorithms to rapidly produce collision-free trajectories from the roadmap.

Extensive research has been done on finding optimized paths on a graph. The best known single-source shortest path(SSSP) algorithm is likely Dijkstra's algorithm [12], but within the SSSP domain, a lot of developments have been made in the last 50 years. Many newer algorithms can be classified as either heuristic, incremental, or both. Heuristic search algorithms use an approximate distance from the goal as additional knowledge to speed up the search. A* is an example of heuristic search that is simply Dijkstra's algorithm with a heuristic added [13]. Incremental search algorithms, on the other hand, use information from previous searches to speed up the current search. D* Lite is an example of an incremental search algorithm [14]. Both A* and D* Lite can be adapted in many ways to meet the individual requirements of a system.

1.3 Problem Statement

The problem solved by this research is to plan and execute motions for high-DOF robot arms in a reactive and inutitive manner while coordinating with a task-level executive to accomplish tasks based around motion planning problems in a changing environment. The environment changes in our motivating examples are often due to collaboration with a human that acts in a manner not known to the robot a priori. It is imperative for human-robot collaboration that the motion planner is reactive so as to not execute plans that collide with the human or other moving obstacles, but also to generate plans quickly enough that they are still relevant to the task at hand by the time they are executed. Plans created by the motion planner must also be intutive, so a human can determine what the robot is trying to accomplish and therefore not interfere with the execution. By our definition, intuitive also means near-optimal in accordance with objective function that can be specified to optimize for energy efficiency, robustness, speed, smoothness, and a variety of other objectives [15].

The motion plans for the problems I am addressing consist of trajectories containing a sequence of robot states in configuration space. I am limiting the modeled configuration space to the actuator positions, or joint state, of the robot arm. The motion planning in the scope of my thesis will ignore robot dynamics by assigning conservative time differences between robot states in a trajectory.

In addition to limiting the planner to purely kinematic motion planning, there are a number of key assumptions made for the development of this motion planner. The first assumption is that the manipulation workspace is characterized by a limited set of pregrasp poses. Next, the motion between a pregrasp and a grasp pose is assumed to be short and best handled by visual and force servoing loops rather than open-loop planners. Finally, the environments encountered by a robot using our motion planner are assumed to not be overly complex. Environments are assumed to consist of a small set of potential obstacles, some static such as a workpiece or a table, and some dynamic such as another robot or a human. The emphasis here is on achieving fast

performance in typical, practical situations [15].

1.4 Approach

The core of the motion planner is the combination of a roadmap-based motion planner with an optimization based motion planner, TrajOpt. These trajectories must avoid static obstacles observed during the construction of the roadmap as well as dynamic obstacles introduced during or after the roadmap construction. The roadmaps constructed for this thesis are relatively sparse (1000 nodes) for the high-DOF robotic arms that this research focuses around. Using a sparse roadmap for a high-DOF arm allows for fast search and updates to the roadmap as required by changes in the environment. Our system requires that roadmap trajectories are confirmed to be collision-free before they are provided to TrajOpt. TrajOpt is being used for its ability to quickly adjust a seed trajectory to minimize the required motion to get to the goal state while avoiding obstacles in the environment. TrajOpt also makes the roadmap-based planner more complete due to its complete coverage of the reachable workspace for a robot compared to the coarse coverage provided by a sparse roadmap alone.

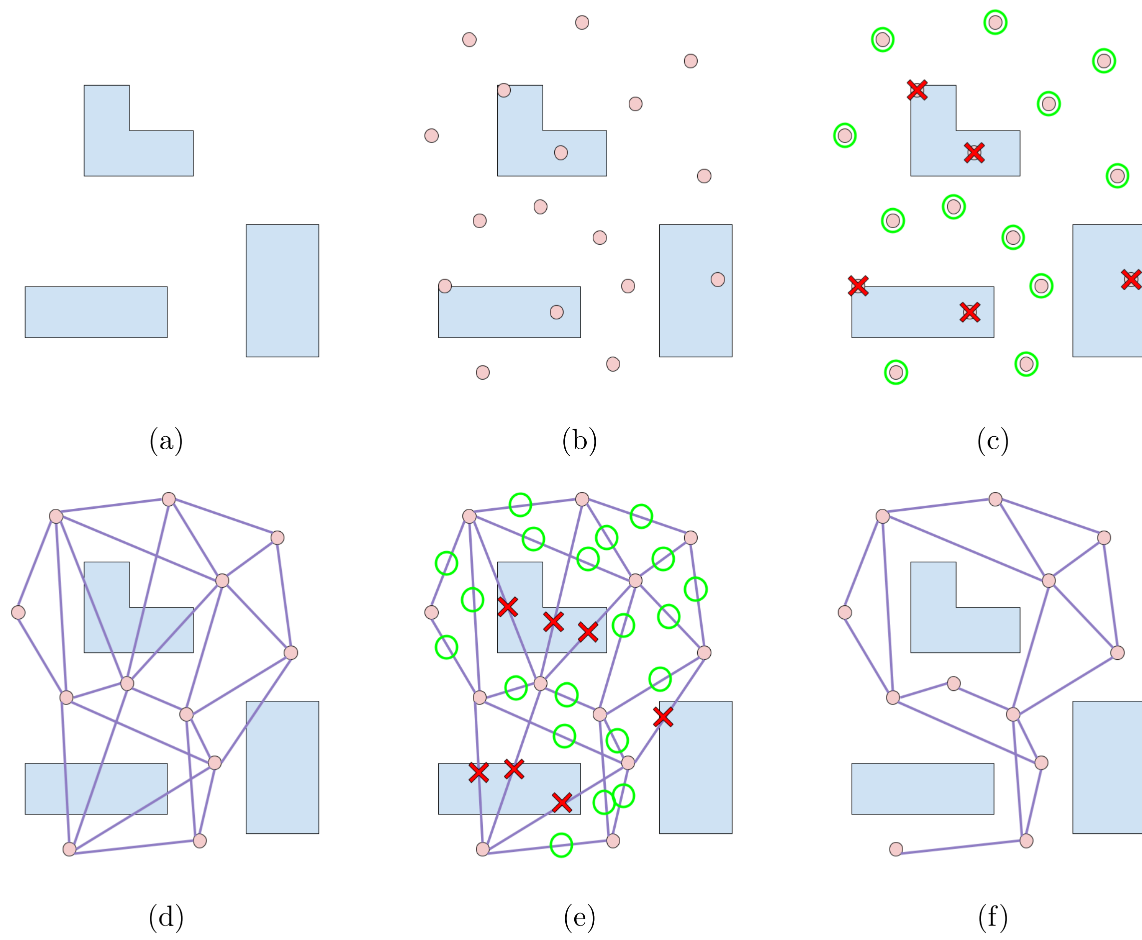


Figure 1-2: A basic outline of how a roadmap is constructed. The static environment is showin in (a). Roadmap nodes are randomly sampled in (b) and then nodes are checked for collisions in (c). Nodes in collision are pruned from the roadmap. Edges are generated between nearby nodes in (d) and then checked for collisions in (e). The final roadmap is shown in (f) after edges in collision have been removed.

Additionally, these roadmaps are coupled with a solution cache containing paths that are computed offline during roadmap construction. Precomputing paths for the sparse roadmap reduces the amount of online search required and therefore the time to produce a plan on average. Caching additional path solutions for a pair of roadmap nodes that stray from the shortest path in interesting ways can provide additional reduction in the online search required and overall planning time. These cached solutions can be coupled with online search to limit the amount of repeated work that is performed over the course of a planning problem.

The solutions that are produced by providing seeds from our sparse roadmaps to TrajOpt are more optimal than what can be obtained by planners of comparable speed and are produced more quickly than what can be obtained by planners of comparable optimality. My research investigates how to smooth the transition between offline and online planning approaches with obstacle information introduced after construction and how to synthesize solutions from a precomputed solution cache for a roadmap to avoid obstacle collisions that arise during the execution of a solution. The contributions are as follows:

1. A framework for the construction, augmentation and testing of roadmaps
2. Offline algorithms for precomputing useful paths for the solution cache for a roadmap
3. Online algorithms for combining a precomputed solution cache with online search and for otherwise performing fast online search

1.4.1 Roadmap Framework

The first stage of my research is the development of a module for constructing roadmaps for testing with TrajOpt in the testing environments constructed by Sylvia Dai of MERS. For a given environment, this module samples nodes for the roadmap in configuration space and discards a node if the robot forward kinematics are in collision with a static obstacle in the environment. Nodes are connected by edges

that are confirmed to be collision-free in the static environment after all nodes have been sampled. After the full roadmap has been constructed, a cache of solutions is computed and attached to the roadmap for storage. The justification for sampling in configuration space, the specific sampling approaches, and the subsequent edge and solution generation will be further explained in the Methods Developed chapter.

The difficulty in implementing a sparse roadmap has to do with selecting from different existing PRM implementation strategies and tuning different roadmap hyper-parameters in order to achieve good workspace connectivity for our roadmaps with a minimal set of configuration states. The surrounding software system has to allow for flexible configuration of the roadmap construction process while allowing for a variety of post-construction augmentations and remaining robust in the face of exhaustive testing.

Something that our group is interested in is hard-coding pregrasp poses into the roadmap for objects the robot will interact with. We are also interested in using semantic information to aid roadmap construction. In addition to using this information to increase roadmap connectivity in difficult sections of an environment, a goal for using semantic information is to allow our motion planner to provide trajectories to pregrasp poses that have been generated dynamically. From there, a separate controller will be used to handle the object interaction.

The early testing conducted for this thesis seeks to uncover an optimal roadmap resolution along with values for other roadmap hyper-parameters for maximizing the performance benefits provided by seeding TrajOpt with a non-optimal roadmap trajectory. The performance benefit is the improvement of a trajectory in terms of path length, proximity to obstacles, or any other costs modeled in the objective function for TrajOpt. This improvement is at the expense of the extra time taken by the optimization. The experiments hope to show that the extra time for optimization is offset by obtaining a collision-free seed from a sparse roadmap more rapidly than would be possible for a planner producing more optimal initial trajectories.

1.4.2 Offline Algorithms

The next stage of my research surrounds developing offline approaches for creating roadmap solution caches that contain multiple useful solutions for a pair of roadmap nodes. Using a precomputed all-pairs shortest paths solution cache is not a new idea. What are novel, however, are the approaches I have developed to cache additional solutions that are not the shortest paths between a pair of nodes for the static environment. That being said, the approaches I have developed surrounding precomputed solutions are based on a core assumption: A significant majority of the collisions that the sparse roadmaps for stationary manipulators will encounter can be represented with a relatively small set of objects and corresponding poses for those objects.

I believe this is a valid assumption because I believe the workspace coverage of my roadmaps is coarse enough that there is small number of paths between any pair of nodes that will avoid a significant majority of the collisions that can be avoided by a path between those two nodes in that roadmap. From there, I believe those paths can be discovered using a small set of objects and corresponding poses to obstruct the roadmaps.

Given that assumption, it is worth noting that if those paths are captured in the cache, TrajOpt will be provided a collision-free seed and produce a near-optimal trajectory without any online search in most cases. There will still be times where no collision-free solution exists in the cache due to obstacles in the environment. For when that occurs, I have implemented online search routines that that build off of the work that was performed checking the cached solutions to return a collision free solution whenever one exists in the roadmap.

1.4.3 Online Algorithms

This brings me to the final stage of my research: fast online planning for changing environments. Many of the algorithms developed in this stage of my research consist of A* variants that iteratively search the roadmap while heavily relying on lazy collision checking to minimize online computation. These algorithms utilize paths that already

exist in the solution cache. They also incorporate new paths found through online search back into the solution cache for use in later planning problems.

The last section of my online planning research moves away from precomputed paths and looks at incremental planning and execution. The primary exploration surrounds an implemented search algorithm that is based on D* Lite, but diverges from the established algorithm in interesting ways. These algorithmic divergences potentially help limit delays caused by online replanning due to moving obstacles while incorporating TrajOpt optimization into the incremental algorithm, but they also open up the implementation to new corner cases that can cause inconsistency in the search if not accounted for.

1.5 Introduction Summary

With the completion of this thesis research, I hope to provide a motion planner with the ability to quickly produce motion plans through optimizing collision-free seed trajectories from sparse roadmaps of robot states. The roadmaps used by this motion planner can be quickly updated as dynamic obstacles move around the environment. When the robot is already executing a trajectory, it will monitor execution and form a new plan when the current plan becomes invalid. This new plan builds off information learned during the generation and execution of the previous plan. The result is a motion planner that enables high-DOF robotic arms to accomplish tasks based around motion planning problems with reactive and intuitive motions.

Chapter 2

Methods Developed

The main software architecture developed for this thesis is the roadmap-based motion planner. This motion planner consists of a PRM-style roadmap with robot poses as nodes and and the trajectories to traverse between those poses as edges connecting the nodes. Additionally, the motion planner has some representation of shortest path solutions for the roadmap that it either develops offline, produces as necessary online, or some combination thereof. In this chapter we will explain the different augmentations applied to the motion planner within this core structure of a roadmap and shortest path solutions.

While developing this motion planner, it was important to keep in mind that the system is meant to be used on a real robots solving real motion planning problems. With that consideration, one of the first software systems built for this thesis was a means to executing trajectories on the real robots used for the roadmap based motion planner. The MERS labspace includes a testbed containing a Rethink Robotics Baxter [16] and a Barrett Whole-Arm Manipulator (WAM) [17] to be used for hardware demonstrations and testing.

The motion planning capabilities explored here are mostly agnostic to task or activity level goals. These capabilities have been developed with regards to solving the large robot motions in between more activity specific tasks. That being said, this motion planner has been integrated into an activity level planning and execution system to allow for more compelling demonstrations of the motion planner [18].

Once the basic roadmap-based motion planner was integrated with the robot controllers and an activity level planning and execution system, all development that followed could be focused on addressing the research questions laid out in the Problem Statement. The first of these questions regards the optimal configuration of hyper-parameters that dictate the construction of the roadmap for a given environment. The goal here is to develop roadmaps that effectively cover the reachable workspace without being a burden to interact with from a space and computation standpoint. Next, methods are investigated for utilizing semantic information about an environment to more effectively cover the workspace around objects of interest or within more constrained sections of the environment.

After the performance of the roadmap-based motion planner is sufficient in static environments, the focus shifts to the main innovation of this thesis: collision avoidance in changing environments. The research conducted to address this problem is broken into three main arcs. The first of those is to use a cache of all-pairs shortest path solutions with multiple solutions for each pair of points. The hope with this approach is to find one solution in the set that is collision-free for any given motion planning problem and layout of dynamic obstacles. In the second arc, path searches are conducted online using a single-source shortest path algorithm and checking for collisions in the solutions as they are produced. From there, I explore combinations of the first two arcs where paths produced online are cached for later motion planning problems. The third and final arc is very similar to the second, but involves using incremental algorithms to continuously update knowledge of the environment while producing shortest path solutions.

Finally, I return to one of our primary considerations: developing systems that solve real motion planning problems with real robots. In order to adequately address this, it is important to develop testing procedures in both simulation and hardware that reflect the scenarios that we expect these robots to encounter.

2.1 An Interface for Interaction with Real and Simulated Robots

2.1.1 Development Environment

Early on in the inception of this thesis, it was determined that the robot environment for the motion planner would be managed in OpenRAVE. This decision was made because OpenRAVE supports TrajOpt whereas ROS MoveIt! does not. While TrajOpt is not directly necessary for the roadmap-based motion planner, we were interested in its capabilities when provided seed trajectories of varying quality. In that sense, we used the quality of optimized trajectories produced by TrajOpt as a means of judging the quality of the seed trajectories produced by the roadmap-based motion planner. How the quality of a trajectory was determined is explained in the Experiment Plan chapter.

The issue with managing the environment in OpenRAVE is that the two robots used for this research, Baxter [16] and WAM [17], both have their controllers wrapped in ROS nodes. The Baxter SDK developed by Rethink exposes Baxter's capabilities through ROS topics and the WAM Joint Controller, developed by Steve Levine of MERS, is a ROS node that wraps around the API provided by Barrett Technologies to control the WAM.

ROS MoveIt! was the existing system used for receiving planning requests and developing trajectories to be executed on the on the robots via ROS. This means an interface needed to be developed to receive the planning requests over ROS, conduct the planning problem in OpenRAVE, convert the plan output back to a ROS format, and execute the newly formatted plan on the desired robot via ROS. This interface allows the trajectory execution modules to behave the same if what they are executing comes from the roadmap based motion planner, from TrajOpt, or from any other standard motion planner, like those in OMPL.

2.1.2 Pick and Place Demonstrations

For hardware demonstrations, this interface is connected to a task planning and dispatch module, also developed by Steve Levine, that performs pick and place tasks. The pick and place module previously used MoveIt! to plan an entire pick or place task as well as to execute the plan on the desired robot. These plans consist of a few differnt components that will be illustrated for the task of picking up a block. For more involved planning problems, an activity-level executive develops goals and constraints from human instructions. This executive can also receive a temporal plan network or qualitative state plan as input, both containing human-readable commands [18]. The executive will then dispatch actions necessary to complete the user-defined goals to the appropriate agents and monitors action execution. For our case, the actions received correspond to the aforementioned pick and place tasks which are further broken down into goal poses for the motion planner to generate a full motion plan.

To pick up a block, a plan for large arm motion to a pregrasp poses is developed using an online sampling-based motion planner, like RRT. This is followed by a call directly to the robot to open its gripper. Then a Cartesian planner is used to move from the pregrasp pose to a pose where the open gripper can pick up the block by closing. The plan produced by this Cartesian planner is a straight-line trajectory in workspace for the robot gripper. To complete the task, the gripper closes around the block and uses the Cartesian planer to return to the pregrasp pose. From there a place task could then begin with the planning of the large arm motion to a pregrasp pose corresponding to where the held block should be placed.

In this description of pick and place tasks, the roadmap-based motion planner replaces MoveIt! for planning the large arm motion, but MoveIt! is still used for the Cartesian planner. This is facilitated by a wrapper developed for the roadmap-based motion planner to mimic the API of the MoveIt! motion planner. The wrapper dispatches planning requests to the appropriate planner, so the API is identical from the vantage of the pick and place module. This separation is to ensure that the scope of this thesis does not creep into problems involving grasp planning.

Additionally, I implemented ROS nodes to simulate the behavior of the two robots used for testing. These simulated robots receive requests on the ROS topics used for trajectory execution and publish information that mimics what the real robots would be publishing during trajectory execution. Additional simulation functionality has been added to these nodes as needed, which include ROS services for operating the gripper attached the WAM. These simulations make it possible to use RVIZ to visually confirm the execution of activity level plans that have been broken down into motion planning problems without having to execute them on the real hardware.

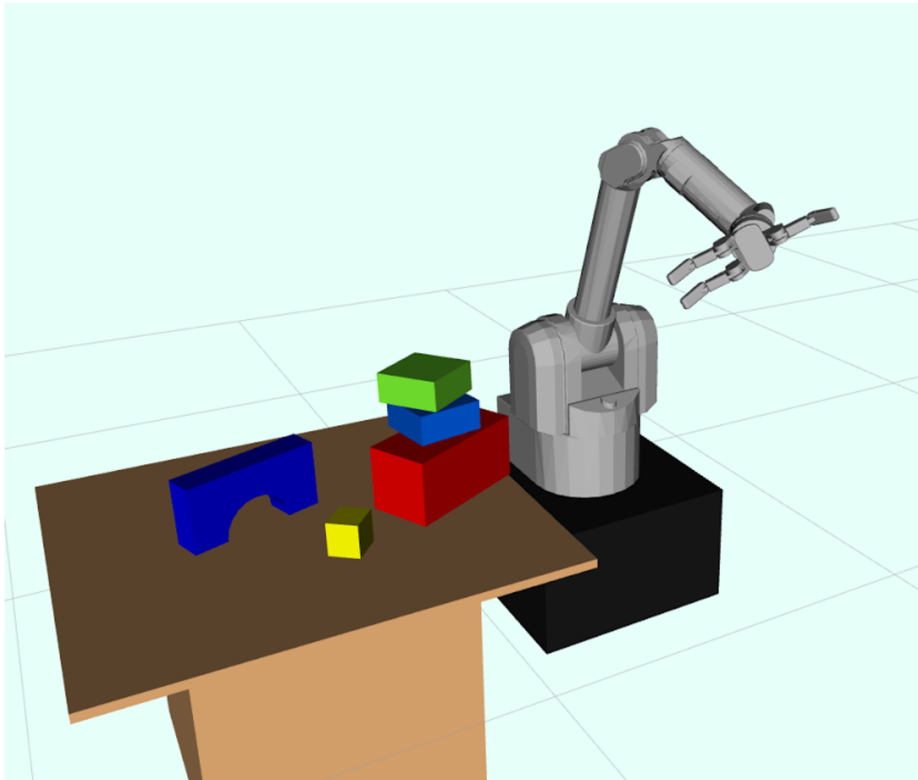


Figure 2-1: The Barrett Whole-Arm Manipulator (WAM) shown in a ROS visualization of our hardware test-bed for pick and place tasks.

2.2 A Roadmap Based Motion Planner with Shortest Path Solution Cache

2.2.1 Software Modules

The implementation of the roadmap-based motion planner is a Python class called RoadmapManager, so the roadmap-based motion planner will be referred to as the RoadmapManager from here on out. The RoadmapManager wraps around a MATLAB interface for building, storing, and loading roadmaps so they can be queried and augmented in Python. Initially MATLAB was used for querying the roadmap for shortest path solutions during online motion planning. This approach spent an unnecessary amount of time loading the solution cache for each query, so later iterations load the entire shortest paths cache into Python from MATLAB, leaving MATLAB unused after the initialization of the RoadmapManager for a given roadmap.

The decision to use MATLAB within the RoadmapManager was made largely to support the potential integration of existing software systems that had been developed in MATLAB. However, this decision has also allowed us to leverage MATLAB's easy to use parallel processing tools for roadmap construction, specifically for the calculation of the all-pairs shortest path solution caches.

The module that provides the communication between the RoadmapManager and the simulated or real robot is the RobotClient. For a specific robot, "Robot" is replaced by the name of the robot used, WAMClient for example. The first iteration of RobotClient that could call to MATLAB for a seed trajectory from a roadmap existed before the roadmap interface was standardized with the RoadmapManager. The roadmap it used had a fixed interval grid of nodes in configuration space with edges between adjacent nodes in each dimension but only changing one dimension per edge. For reference, a 900 node roadmap constructed for the 4 most proximal joints of the WAM using this strategy for node and edge placement had 22082 edges. The corresponding APSP cache took a week to generate using a non-optimal single-source shortest path algorithm. This roadmap was overly dense in terms of the number

of nodes and the edges could not skip over nodes. This led to an all-pairs shortest paths solution cache that was time consuming to calculate and memory bloated for its performance capabilities.

Ensuing development of the RoadmapManager was guided by a handful of questions regarding offline roadmap construction as well as online adjustment and querying of the roadmap and shortest paths solution cache. With regards to construction, the main interests were the roadmap size in terms of number of sampled nodes, the sampling resolution of the robot state, and how sampled nodes would be connected by edges. For online behavior, the primary concern was how to incorporate obstacles into the roadmap after construction. This involves how to efficiently invalidate parts of the roadmap as well as how to use the solution cache given that precomputed solutions may be partially or fully in collision.

2.2.2 Roadmap and Cache Construction

The roadmap construction is modeled loosely after the k-nearest variant of sPRM [1]. The similarity to sPRM over the original PRM comes from sampling all roadmap nodes before attempting any edge connections. Where our algorithm differs from k-nearest sPRM algorithm is in edge generation. For a given node, rather than attempting to connect edges to the k-nearest neighbors in the roadmap, the algorithm iterates over its neighbors in order of increasing distance while attempting to connect an edge to each neighbor from that node. This iteration terminates when k collision-free edges have been found or N neighbors have been tested. Edges were determined to be collision free by checking for collisions along a linear interpolation of the edge at fixed intervals determined by the edge length. All of my roadmaps were constructed with $k = 10$ and $N = 100$.

This approach was taken in favor of a standard k-nearest approach or the ball radius used by the original PRM algorithm to encourage roadmap connectivity while limiting unnecessary edges. Given a configuration space roadmap with states in close proximity to workspace obstacles, ensuring the roadmap is fully connected would be more difficult using a fixed number of neighbors or a fixed distance to check within.

It is important for testing that roadmaps are fully connected.

Even though we are concerned with workspace connectivity and collisions, roadmap nodes are sampled in configuration space for a couple reasons. First of all, TrajOpt requires seed trajectories to be composed of joint states. Second, the roadmaps are primarily being developed for redundant manipulators, so a point in workspace would have multiple corresponding solutions in configuration space for our robots. Since our group will eventually be incorporating dynamic constraints into the roadmap-based motion planner, it is important to have a singular robot state for each node in the roadmap.

Additionally, when constructing roadmaps for Baxter and WAM,which are both 7-DOF manipulators, random values are only sampled for the 4 most proximal joints, while fixed values are assigned to the remaining 3 joints. This approach was implemented to improve workspace coverage for a roadmap with a minimal number of sampled joint states. The assumption is that for many-DOF manipulators, the position of the end-effector is most greatly impacted by the more proximal joints, while the more distal joints have a greater impact on the end-effector orientation. Furthermore, we assume that when connecting to the roadmap for a query, it is easier to change orientation than position while avoiding collision, so it is more important to have an existing roadmap node with a nearby position than a nearby orientation.

The algorithm used to construct the all-pairs shortest paths solution caches is a Dijkstra's variant called Yen's Algorithm [19]. Yen's Algorithm is a single-source algorithm that finds the p-shortest paths between two nodes in a graph for a given p. I use this algorithm between every pair of nodes in a roadmap in order to construct the full solution set. Since this algorithm has to be run n^2 times for n roadmap nodes, the ability to run it on parallel processors for different node pairs greatly speeds up the cache construction time. While some of my experiments did compare different values of p, the majority utilized this algorithm with $p = 1$ so other single-source shortest path algorithms could have been used in its place. Pseudocode for Yen's Algorithm is shown below.

```
1:  function ComputePShortestPaths(G, s_start, s_goal, P)
2:      A[0] = Dijkstra(G, s_start, s_goal);
3:      B = ∅;
4:      for p from 1 to P do
5:          for i from 0 to Size(A[p − 1]) − 2 do
6:              spurNode = A[p − 1].Node(i);
7:              rootPath = A[p − 1].Nodes(0, i);
8:              for each path in A do
9:                  if rootPath = path.Nodes(0, i) then
10:                     G.Remove(path.Edge(i, i + 1));
11:             for each node in rootPath except spurNode do
12:                 G.Remove(node);
13:             spurPath = Dijkstra(G, spurNode, s_goal);
14:             totalPath = rootPath + spurPath;
15:             B.Append(totalPath);
16:             restore edges to G;
17:             restore nodes in rootPath to G;
18:         if B is empty then
19:             break;
20:         B.Sort();
21:         A[p] = B[0];
22:         B.Pop();
23:     return A;
```

Figure 2-2: Yen's algorithm

2.3 A Framework For Providing Collision Free Trajectories in Static Environments

Once a roadmap has been constructed along with its corresponding APSP solution cache, it can then be used to handle motion planning queries. A motion planning query consists of a start point and an end point, both in configuration space. The first thing that must happen for a successful query is that the the start and end points, or query points, must be connected to existing nodes in the roadmap.

For each query point, check the nearest m roadmap nodes to that query node in order of increasing euclidean distance in configuration space. At each of the m nodes, check for a collision-free edge from the query point to that node. Stop the iteration for that query point when the first collision-free edge is found or return a failure for the query if no such edge is found to any of the m nodes. I used $m = 100$ for all of my experiments to allow a failure to be returned for a query within about one second of unsuccessful collision checking.

Some additional exploration was done with using different sorting heuristics to modify the order of the roadmap nodes that the query points attempt to connect to. Namely, analysis was performed where the m nearest nodes were sorted by either a weighted or unweighted euclidean distance to both query points rather than just the query point attempting to connect to the m nodes. This meant that the roadmap nodes connected to by the query points tended to be more along the way to one another.

If a pair of roadmap nodes is successfully connected to by the pair of query points, the node pair is used to obtain a corresponding set of cached solutions. Full paths are then made for the query by bookending the cached solutions with the start and end points for the query. From here, there are a few options depending on what assumptions are made for the received queries.

If the environment is assumed to have not changed from the static environment that the roadmap was constructed for, then the shortest cached path can be returned with no collision checking. If that assumption is not made about the environment, then the shortest cached path that is confirmed to be collision-free at the time of the query is returned. Finally, if no cached paths are collision-free for the roadmap nodes connected to at the time of the query, then there are steps that can be taken to return a collision-free path that has been modified from from a cached solution. The different approaches taken to modifying cached solutions will be explained later in this chapter.

2.4 Using Semantic Information to Extend the Ability of the Motion Planning Framework

Something that is of interest to our group is how to effectively use semantic information about objects in the environment to improve the performance of our roadmap-based motion planner. The primary way we are currently using semantic information in our end to end planning and execution demonstrations is in the use of pregrasp

poses. For objects in the environment that we expect our robot to manipulate, like blocks or drawers, sets of end-effector poses are generated relative to the object that would allow the robot to grasp the object. If the object is expected to be in a finite number of states, like an open or closed drawer, then these poses can be added to the roadmap as joint states using inverse kinematics. For objects like blocks, which can exist in a wide variety of states in an environment, these poses are only used at runtime to connect corresponding joint states to existing nodes in the roadmap.

The other use of semantic information that we have explored to some degree is how to use information about objects in the environment to improve connectivity in cluttered areas where we expect the robot to interact often. This exploration was conducted in the "Shelf with Boxes" environment in an attempt to improve connectivity in and around the narrow shelves. The approach taken was as follows. For a given object of interest, in our case the shelf, obtain a planar grid of positions at some fixed resolution relative to the object. The different planar grids I tested were just outside the shelf, just within the shelf, and halfway inside the shelf.

For a given planar grid of positions, iterate over each position and at each position iterate over a range of orientations. For each resulting pose, attempt to find a collision-free inverse kinematic solution for the robot. Stop iterating over orientations when the first collision-free solution is found for a given position and move on to the next position. The collection of inverse kinematic solutions found are added to a base roadmap for the environment and then new edges are generated for the augmented roadmap along with the APSP solution cache. Although it will not be covered in this thesis, our group hopes to extend this exploration to generating three-dimensional grids of positions for some volume related to a given object of interest.

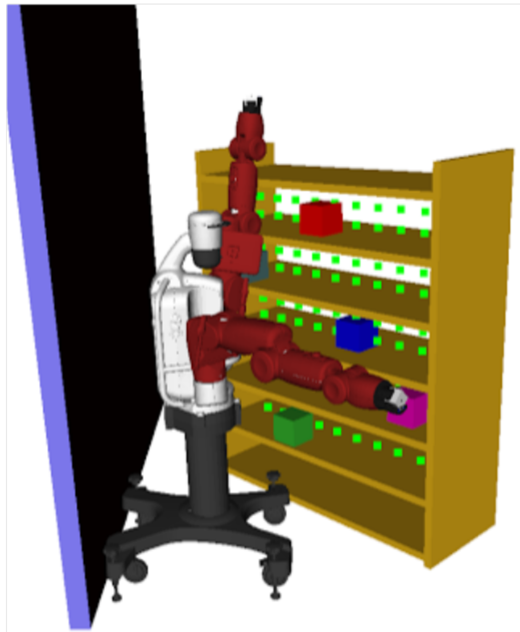
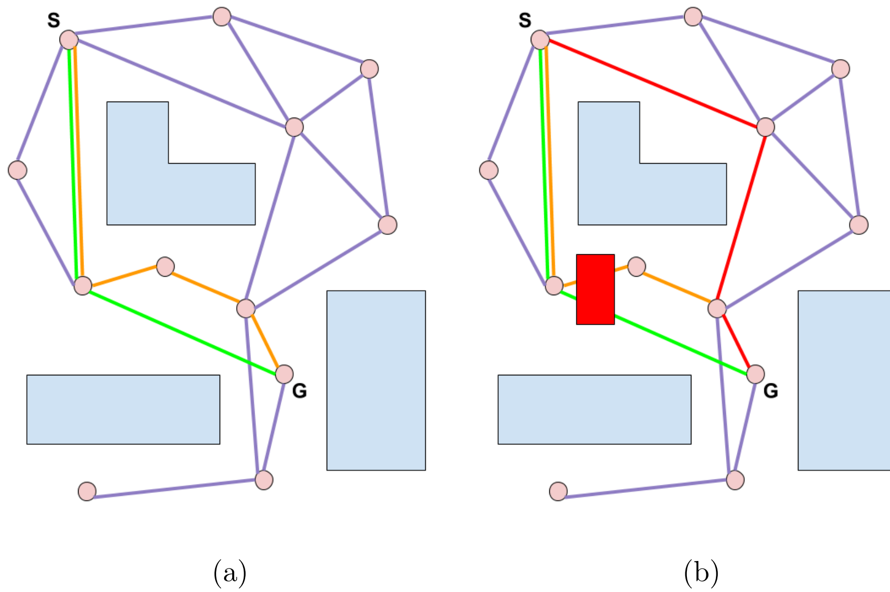


Figure 2-3: A visualization of the end-effector poses within the shelf that were tested for valid IK solutions to add to the roadmap

2.5 Offline All-Pairs Shortest Path Strategies for Avoiding Dynamic Obstacles

2.5.1 Simple Strategy

Offline APSP strategies refer to steps taken to modify an initial APSP solution cache before the roadmap is used to handle any motion-planning queries. The main idea is to precompute additional path solutions for roadmap nodes to handle cases where the shortest path between two nodes is in collision at the time of the query. The simple strategy to leverage the existing software systems being used is to increase p for Yen's Algorithm. For the case where $p = 2$, this would find the shortest path and the second shortest path between every pair of nodes in the roadmap. This strategy



(a) (b)

Figure 2-5: Image (a) shows a roadamp with $p = 2$ shortest paths for a pair of start and points. Image (b) shows how a single obstacle can invalidate both of those paths while a valid shortest path exists in the roadmap (shown in red) but not in the solution cache.

can be used for arbitrary values of p provided that there are enough distinct paths between a pair of nodes.

The issue with this simple strategy is demonstrated below for the case of $p = 2$. Very often, the difference between the shortest path and the second shortest will only be a few edges. As a result, an obstacle that obstructs the shortest path will very likely obstruct the second shortest path. This will also hold true for the n_{th} shortest path and the $(n+1)_{th}$ shortest path. The takeaway from this is that storing a collection of paths for a given pair of roadmap nodes is is more useful if the paths in that collection differ from one another significantly enough to be able to avoid obstacles that obstruct other paths in the collection.

2.5.2 Realistic Obstacles

With that in mind, I looked at how to invalidate parts of the roadmap prior to computing shortest path solutions in order to create many different sets of all-pairs shortest path solutions. The idea I settled on is to use a set of generated obstacles and poses for those obstacles to simulate collisions. Of course you cannot simulate all of the infinite possibilities of of obstacle collisions for a given robot. As mentioned in the Problem Statement, this approach is based on an assumption that a sparse set of objects and corresponding poses can be representative of a significant majority of the obstacle collisions the robot will encounter when solving real world planning problems.

The obstacles I have chosen as representative of these collision scenarios are shown below along with their dimensions in meters. For each of the five obstacles, I have selected eight poses for each environment that I believe will obstruct the robot in interesting or representative ways for that obstacle in the real world. The cup, thermos, and monitor are arranged in a variety of ways on the flat surfaces in the environment whereas the bent and straight arms are in different floating orientations around the robot to simulate a person reaching around or just generally interacting with the robot. The five obstacles with eight poses each create forty total object-pose tuples for each environment. I will refer to the set of obstacles I have chosen as realistic obstacles from here on.

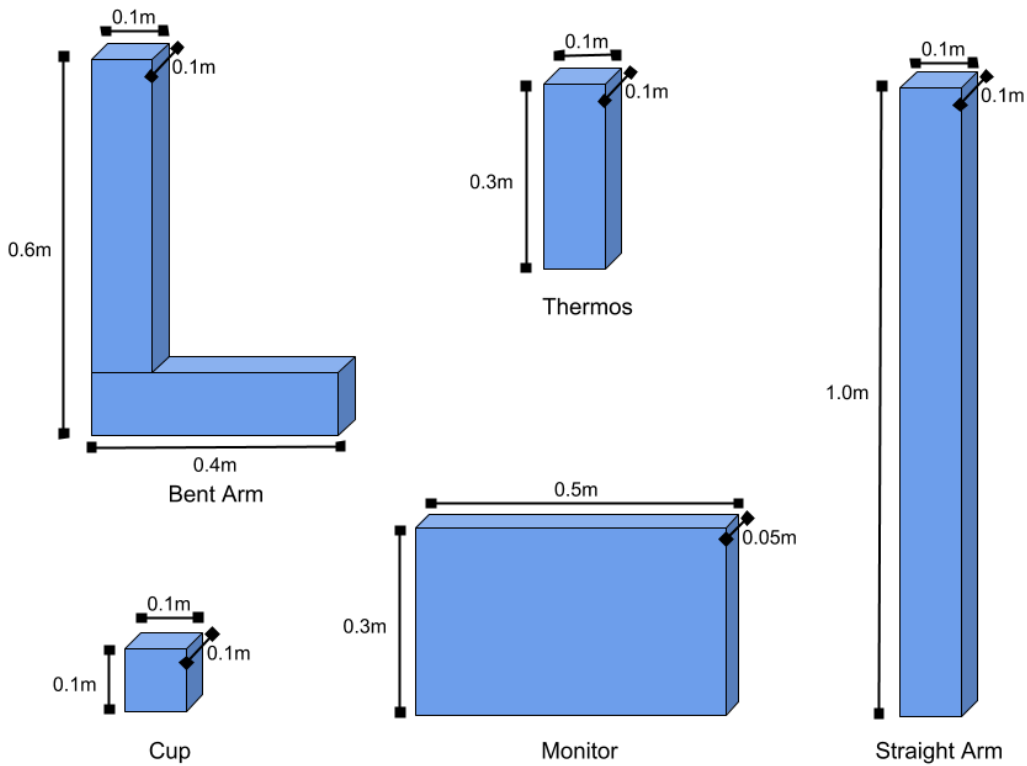


Figure 2-6: The five obstacles used to obstruct the robot in ways that are representative of the environment

2.5.3 Improved APSP Solution Cache

The offline approach developed to utilize the realistic obstacles is a means of computing multiple solutions for a pair of nodes in the APSP solution cache that are significantly different from one another. This is in contrast to the simple strategy where the multiple solutions are fairly similar due to the nature of being the p shortest paths for a pair of nodes. The steps taken to make these roadmaps with improved APSP solution caches is as follows:

1. Construct a roadmap by sampling nodes and connecting them with edges for the static environment. Then use $p = 1$ for generating the initial APSP solution cache.

2. Randomly select a subset of object-pose tuples and insert that subset into the

static environment. The subset is sampled from the full set using a binomial distribution with 0.1 probability of selecting any one of the forty object-pose tuples. This leads to scenarios where obstacles overlap with one another, but for our purposes, that creates more difficult obstacle configurations to navigate around.

3. Temporarily remove any edges from the roadmap that are in collision with the obstacles that have been inserted into the environment. This requires individually checking every edge for collisions.

4. Generate a new APSP solution cache for the current state of the roadmap using $p = 1$ without discarding the existing cache. Add any newly generated solutions to the existing cache that are not already stored in the cache.

5. Remove all inserted obstacles from the environment. Repeat steps 2, 3, and 4 to generate additional solutions.

Roadmap solution caches have been constructed with this approach using 5, 10, 15, 20, and 25 iterations of inserting obstacles and computing new solutions. Moving forward, this approach will be referred to as APSP Training.

2.6 Online Single Source Shortest Path Strategies for Avoiding Dynamic Obstacles

The online single-source shortest path strategies developed for avoiding dynamic obstacles all wrap around an implementation of A* developed for the RoadmapManager. This A* implementation attempts to connect query nodes to the roadmap as with the standard shortest path query. When expanding roadmap nodes, it can either check for edge collisions as it expands nodes in the search, or it can just rely on an initial set of edges known to be in collision and assume all others are collision-free. For the approaches outlines below, the latter of the two collision checking approaches is used.

This lazy collision checking approach allows for minimal collision checking by only checking edges that the A* implementation believes are part of a collision-free solution and then iteratively providing A* a more accurate knowledge of edges in collision. This occurs until A* eventually returns a solution that is in fact collision free at the time of the query or it returns no roadmap solution at all indicating that none exist for the current state of the environment, so a failure is returned for the query. While the following are referred to as multiple approaches, it is important to note that each step is a development iteration on the step before. Each of the following approaches are for a shortest path query after the two query points have been connected to existing nodes in the roadmap:

1. For the pair of roadmap nodes connected to, check the shortest cached path for collisions. If it is in collision, use A* with the lazy collision checking approach described above to find a collision-free solution between the pair of roadmap nodes.

2. Once again, check the shortest cached path for collisions. Now if it is in collision, identify each series of edges in collision and the corresponding pairs of collision-free nodes that bookend the colliding edges. Then use A* to find a collision-free solution between pair of collision-free nodes to construct a new collision-free solution between the original pair of roadmap nodes. This approach repairs the cached solution rather than replacing it all together as in approach 1.

3. This iteration incorporates roadmaps with multiple cached solutions for a pair of nodes. Instead of just checking the shortest cached solution for collisions, check each cached solution for collisions in order of increasing distance. The first solution that is collision-free is returned, or if none are collision-free, the solution with the smallest percentage of edges in collision is identified and repaired using approach 2.

4. The final approach approach takes solutions that have been repaired during online motion planning and incorporates them back into the cache over the

lifetime of the particular roadmap. Every solution in the cache has a whole number attached to it that corresponds to the number of times that solution has been returned for an online motion planning query since that roadmap was first constructed. That is to say the solution was used without needing to be repaired. Additionally, a roadmap is allowed to have a maximum of s solutions between any pair of roadmap nodes. If a pair of roadmap nodes are connected to during a query and none of the existing s solutions are collision-free, a repaired solution found using the approach from the previous iteration will replace the solution from the s cached solution with the fewest uses and will be assigned $uses = 1$.

The idea behind the final approach is that as more planning is performed with a particular roadmap, the solutions that will remain in the cache are the shortest paths in the roadmap that are able to avoid collisions that other cached or repaired solutions are not. For an uncluttered environment, the shortest path will be collision-free and will have its uses incremented, but for obstacle configurations that collide with the shortest path, there will be paths in the cache that typically avoid those collisions while still being fairly short. Moving forward I will refer to the first three of these approaches as A* Repair 1, A* Repair 2, and A* Repair 3. The final approach will be referred to as A* Training. Pseudocode for the four approaches is shown below.

```
1: function GetCollisionFreePath(G, s_start, s_goal, validEdges)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     path = GetShortestCachedPath(s_start, s_goal);
5:     success = True;
6:     while success do
7:         collisionFree, _ = CheckPathCollisions(path, validEdges);
8:         if collisionFree then return path;
9:         success, path = ComputeShortestPath(G, s_start, s_goal, validEdges);
10:    return Failure;
```

A* Repair 1

```
1: function GetCollisionFreePath(G, s_start, s_goal, validEdges)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     originalPath = GetLeastCollidingCachedPath(s_start, s_goal);
5:     collisionFree, nodePairs = CheckPathCollisions(originalPath, validEdges);
6:     if collisionFree then
7:         Increment the number of uses for the path;
8:         return originalPath;
9:     success = True;
10:    while success do
11:        success, path = RepairPath(G, originalPath, validEdges, nodePairs);
12:        collisionFree, _ = CheckPathCollisions(path, validEdges);
13:        if collisionFree then
14:            if Size(G.Paths(s_start, s_goal)) < maxPaths then
15:                remove longest path with minimum uses
16:            G.Paths(s_start, s_goal)).Append(path);
17:            Set number of uses for the new path to 1;
18:            return path;
19:    return Failure;
```

A* Repair 4

```
1: function GetCollisionFreePath(G, s_start, s_goal, validEdges)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     originalPath = GetShortestCachedPath(s_start, s_goal);
5:     collisionFree, nodePairs = CheckPathCollisions(original_path, validEdges);
6:     if collisionFree then
7:         return originalPath;
8:     success = True;
9:     while success do
10:        success, path = RepairPath(G, originalPath, validEdges, nodePairs);
11:        collisionFree, _ = CheckPathCollisions(path, validEdges);
12:        if collisionFree then
13:            return path;
14:    return Failure;
```

A* Repair 2

```
1: function GetCollisionFreePath(G, s_start, s_goal, validEdges)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     originalPath = GetLeastCollidingCachedPath(s_start, s_goal);
5:     collisionFree, nodePairs = CheckPathCollisions(originalPath, validEdges);
6:     if collisionFree then
7:         return originalPath;
8:     success = True;
9:     while success do
10:        success, path = RepairPath(G, originalPath, validEdges, nodePairs);
11:        collisionFree, _ = CheckPathCollisions(path, validEdges);
12:        if collisionFree then
13:            return path;
14:    return Failure;
```

A* Repair 3

1: **function** ComputeShortestPath(G, s_{start}, s_{goal}, $validEdges$)
2: $evaluated = \emptyset$;
3: $discovered = \emptyset$;
4: $parentMap = \emptyset$;
5: **for all** $s \in G.States$ **do**
6: $f(s) = g(s) = \infty$;
7: $g(s_{start}) = 0$;
8: $f(s_{start}) = h(s_{start}, s_{goal})$;
9: **while** $discovered$ is not empty **do**
10: $s_{current} = discovered.Pop()$;
11: **if** $s_{current} = s_{goal}$ **then**
12: $path$ = ReconstructPath($parentMap$, $s_{current}$)
13: **return** True, $path$;
14: $discovered.Remove(s_{current})$;
15: $evaluated.Add(s_{current})$;
16: **for** $neighbor \in G.Neighbors(s_{current})$ **do**
17: **if** $neighbor \in evaluated$ **then**
18: $continue$;
▷ This check is the only deviation from standard A*
19: **if** $G.Edge(s_{current}, neighbor) \notin validEdges$ **then**
20: $continue$;
21: **if** $neighbor \in evaluated$ **then**
22: $continue$;
23: **if** $neighbor \notin evaluated$ **then**
24: $discovered.Add(neighbor)$;
25: $score = g(s_{current}) + cost(s_{current}, neighbor)$;
26: **if** $score \geq g(neighbor)$ **then**
27: $continue$;
28: $parentMap(neighbor) = s_{current}$;
29: $g(neighbor) = score$;
30: $f(neighbor) = g(neighbor) + h(neighbor, s_{goal})$;
31: **return** False, []

A* search algorithm adapted to incorporate knowledge of in-collision edges for an inputted graph

1: **function** ReconstructPath($parentMap$, $current$)
2: $path = [current]$;
3: **while** $current \in parentMap.Keys$ **do**
4: $current = parentMap(current)$;
5: $path.append(current)$;
6: **return** $path$;

7: **function** RepairPath(G, $path$, $validEdges$, $nodePairs$)
8: $repairedPath = []$;
9: $pathIndex = 0$;
10: **for** $(u, v) \in node_pairs$ **do**
11: $repairedPath.Extend(path[pathIndex : path.Index(u)])$;
12: $success, segment$ = ComputeShortestPath($G, u, v, validEdges$);
13: **if** $success \neq True$ **then**
14: **return** Failure;
15: $repairedPath.Extend(segment)$;
16: $pathIndex = path.Index(v) + 1$;
17: **return** $repairedPath$;

ReconstructPath for A* Search and RepairPath for A* Repair

Figure 2-9: The four versions of A* Repair and helper functions. CheckPathCollisions checks the path for collisions, obtain pairs of nodes surrounding in-collision edges, and updates $validEdges$ accordingly.

2.7 An Incremental Search Strategy for Avoiding Dynamic Obstacles

The offline and online approaches described above both seek to minimize the amount of online search and roadmap validation required for motion planning problem. In particular, collision checking in configuration space is a bottleneck of the system, so the developed approaches avoid repeated checks of any nodes or edges in the roadmap over the course of a planning problem. Something important to consider is how to modify, or revalidate knowledge of the environment collected during planning as the environment changes during plan execution. This is not accounted for by the aforementioned approaches, which would have to fully replan from scratch in cases where a plan becomes invalid during execution.

This scenario is one of the motivation for incremental search algorithms. According to the authors of the D* Lite incremental search algorithm, these methods "use heuristics to focus their search and reuse information from previous searches to find solutions to series of similar search tasks much faster than is possible by solving each search task from scratch," [14]. At a high level, D* Lite consists of the following steps:

1. Search for a plan from the goal to the current state of the robot
2. Move from the current state to the state that brings the robot closest to the goal and update the current state accordingly
3. Check if knowledge of obstacles has changed within a scan radius of the robot
    - If so, update all edges with changed costs and update the shortest path from the goal to the new current state
4. Repeat steps 2 and 3 until the goal has been reached

The incremental search strategy taken here is to adapt the D* Lite algorithm to a high-dimensional configuration space rather than the low-dimensional grid world often

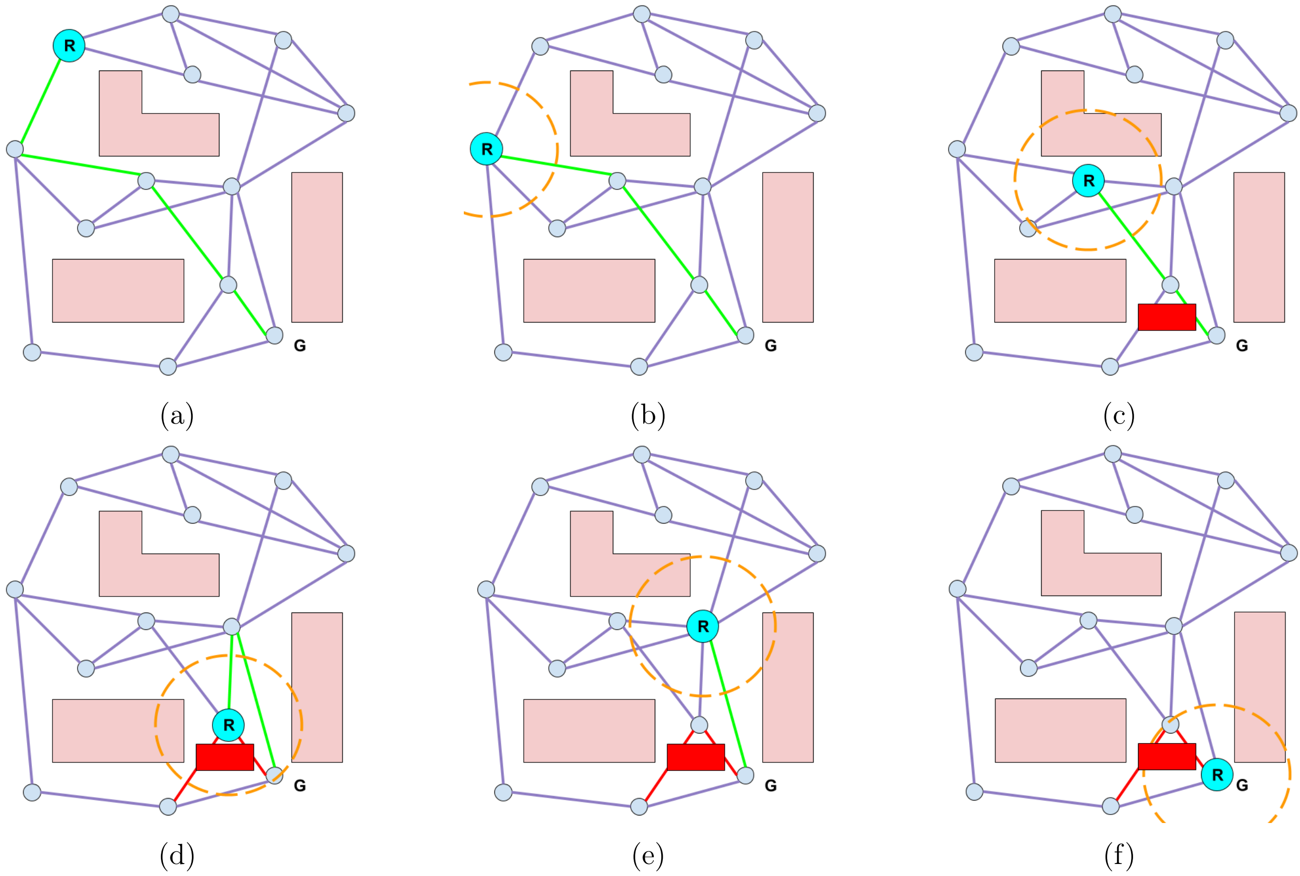


Figure 2-11: Illustration of a standard D* Lite implementation. A path has been found through initial search in (a). The robot begins to execute the trajectory in (b) and (c), and in (d), an obstacle is discovered inside the visibility range. In response, all affected edges are updated and a new plan is formed from the current state of the robot. The new trajectory is executed to to the goal in (e) and (f).

used to illustrate the benefits of the algorithm. Additionally, the configuration space is represented by a sparse roadmap coupled with trajectory optimization that requires a collision-free seed trajectory for the full path. Finally, D* Lite is often demonstrated in examples where the robot has limited observability of the environment. While our system can fully observe the environment, that would not be tenable for the collision checking used and planning times required.

The differences from the standard D* Lite implementation create implementation requirements for the adaptation. The adaptation employs a heuristic-based scan for dynamic obstacles as the execution proceeds. During execution, if the trajectory is a full solution for the motion planning problem, only the executed trajectory is checked for collisions until a collision is found in that trajectory. The difference in scanning approach creates inconsistencies in the cost to goal and estimated path costs for roadmap nodes during successive searches required to repair an initial path. These inconsistencies are addressed with a method that checks for cycles and invalid cost values when nodes are expanded and updated during the path search.

In order to incorporate trajectory optimization into the adaptation, a few things are required. The first requirement is a path reconstruction method similar to A* except it uses the minimum cost to goal for a node instead of the predecessor for a node in the search in order to build the path. After the roadmap path is reconstructed, it is checked for collision and a collision-free subpath is identified starting from the current state of the robot. This path reconstruction and subsequent collision check allow TrajOpt to optimize as much of a solution as could be guaranteed to be collision-free from the path search. After a trajectory has been optimized, a method is required to map the optimized trajectory back to the seed trajectory TrajOpt was provided. This allows the optimized trajectory to be incrementally executed in segments corresponding to the roadmap trajectory. Incremental trajectory execution in this case means that the trajectory is broken up into segments that are sent to the robot controller one at a time rather than sending the whole trajectory at once. This allows execution monitoring to assess if the remaining trajectory to execute is still valid at each step or if replanning needs to be performed.

Something important to highlight about this adaptation of D* Lite is how it interleaves planning and execution. If a collision is discovered in optimized trajectory, but not in the next segment of that trajectory, the next segment is sent to the robot for execution. Immediately after the segment is sent for execution, the necessary collision checking and replanning takes place using the end of the segment as the new current state. This creates stretches of time that trajectory execution overlaps with planning a repaired roadmap trajectory. This approach can be extended to allow more of the optimized trajectory to be executed in parallel with replanning, but the current implementation only executes the next segment. A similar adaptation has also been implemented to equip the different A* Repair algorithms with interleaved planning and execution.

The adapted D* Lite implementation can be reduced to the following high level steps:

1. Search the roadmap for an initial path

2. Identify the collision-free subpath and delay other collision checking

3. Smooth the collision-free roadmap subpath with trajectory optimization

4. Move along the optimized trajectory

5. Scan for collisions along the optimized trajectory

    - If a collision is identified in the optimized trajectory, or the collision-free subpath did not contain the full roadmap path, replan from the current state of the robot and smooth the collision-free subpath within the result.

    - Before replanning, map the collision in the optimized trajectory to roadmap nodes in the collision-free subpath. If the optimized trajectory is collision-free, identify the last node in the collision-free subpath. In either case, perform collision checks on all edges for the identified nodes. Then update end nodes for edges for whom collision status has been updated by the checks.

6. Repeat steps 4 and 5 until the goal has been reached

Full pseudocode for D* Lite and our adaptation are shown below.

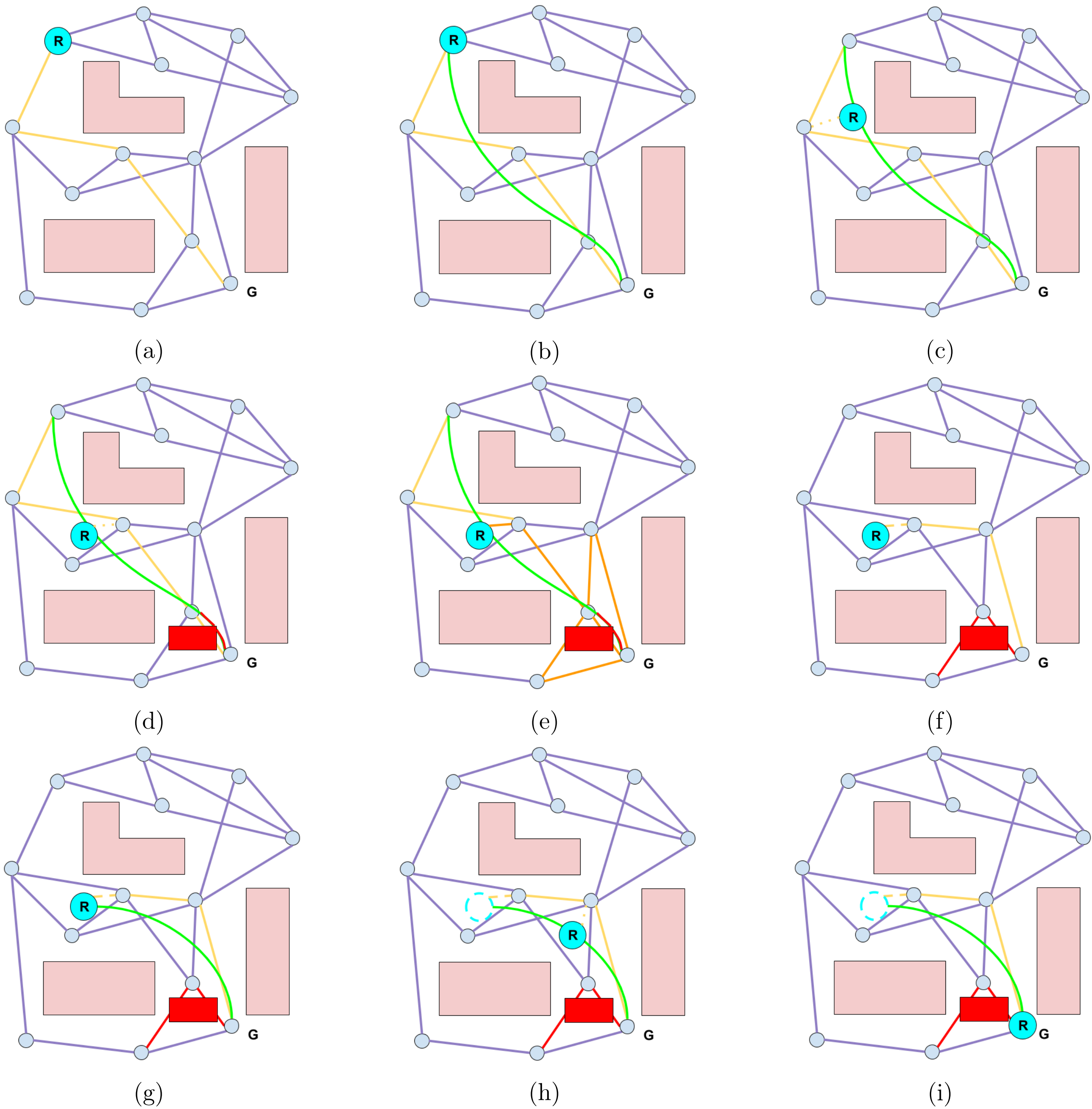


Figure 2-13: Illustration of Adapted D* Lite. A path has been found through initial search in (a) and is then optimized with TrajOpt in (b). The robot begins to execute the trajectory in (c), and in (d), an obstacle is discovered that invalidates the current trajectory. In response the roadmap checks edges near the colliding edge in (e) and forms a new plan from the current state of the robot in (f). The new trajectory is optimized in (g) and is executed to to the goal in (h) and (i).

```
1:  function CalculateKey(s)
2:      return [min(g(s), rhs(s) + h(s_start, s) + k_m);
              min(g(s), rhs(s))];

3:  function Initialize()
4:      U = ∅;
5:      k_m = 0;
6:      for all s ∈ S do
7:          rhs(s) = g(s) = ∞;
8:      rhs(s_goal) = 0;
9:      U.Insert(s_goal, CalculateKey(s_goal));

10: function UpdateVertex(u)
11:     if u ≠ s_goal then
12:         rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
13:     if u ∈ U then
14:         U.Remove(u);
15:     if g(u) ≠ rhs(u) then
16:         U.Insert(u, CalculateKey(u));

17: function ComputeShortestPath()
18:     while U.TopKey() < CalculateKey(s_start) or
              rhs(s_start) ≠ g(s_start) do
19:         k_old = U.TopKey();
20:         u = U.Pop();
21:         if k_old < CalculateKey(u) then
22:             U.Insert(u, CalculateKey(u));
23:         else if g(u) > rhs(u) then
24:             g(u) = rhs(u);
25:             for all s ∈ Pred(u) do
26:                 UpdateVertex(s);
27:         else
28:             g(u) = ∞;
29:             for all s ∈ Pred(u) ∪ u do
30:                 UpdateVertex(s);
```

```
1:  function Main()
2:      s_last = s_start;
3:      Initialize();
4:      ComputeShortestPath();
5:      while s_start ≠ s_goal do
6:          ▷ if g(s_start) = ∞ then there is no known path
7:          s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') +
                g(s'));
8:          Move to s_start;
9:          Scan graph for changed edge costs;
10:         if any edge costs changed then
11:             k_m = k_m + h(s_last, s_start);
12:             s_last = s_start;
13:             for all directed edges (u, v) with changed
                 edge costs do
14:                 Update the edge cost c(u, v);
15:                 UpdateVertex(u);
16:             ComputeShortestPath();
```

Figure 2-15: The standard D* Lite algorithm

```
1: function CalculateKey(s)
2:     return [min(g(s), rhs(s) + h(s_start, s) + k_m); min(g(s), rhs(s))];

3: function UpdateVertex(v)
4:     if u ≠ s_goal then
5:         filteredNeighbors = [];
6:         for neighbor ∈ G.Neighbors(v) do
7:             if G.Edge(v, neighbor) ∈ validEdges and CheckValidNeighborExpansion(v, neighbor) then
8:                 filteredNeighbors.Append(neighbor);
       ▷ Reset our knowledge of the node if it is not reachable in the current environment configuration as determined by having no collision-free edges to neighbors other than those discovered via that node
9:         if filteredNeibors is empty then
10:            g(v) = ∞;
11:            rhs(v) = ∞;
12:            parent.Remove(v);
13:            return
14:        minSucc = arg min_{n' ∈ filteredNeighbors}( c(v, n') + g(n'));
15:        rhs(v) = g(minSucc) + c(v, minSucc);
16:        parentMap(v) = minSucc;
17:    if v ∈ U then
18:        U.Remove(v);
19:    if g(v) ≠ rhs(v) then
20:        U.Insert(v, CalculateKey(v));
```

```
1: function CheckValidNeighborExpansion(v, neighbor)
2:     nextNode = neighbor;
3:     while nextNode ≠ 0 do
4:         if nextNode = v then
5:             return False
6:         if nextNode ∉ parentMap then
7:             return False
8:         nextNode = parentMap(nextNode);
9:     return True

10: function Initialize()
11:    U = ∅;
12:    k_m = 0;
13:    for all s ∈ G.Nodes do
14:        rhs(s) = g(s) = ∞;
15:    for all e ∈ G.Edges do
16:        validEdges.Add(e);
17:    rhs(s_goal) = 0;
18:    parentMap(s_goal) = 0;
19:    U.Insert(s_goal, CalculateKey(s_goal));

20: function ComputeShortestPath()
21:    while U.TopKey() < CalculateKey(s_start) or rhs(s_start) ≠ g(s_start) do
22:        k_old = U.TopKey();
23:        v = U.Pop();
24:        if k_old < CalculateKey(v) then
25:            U.Insert(v, CalculateKey(v));
26:        else if g(v) > rhs(v) then
27:            g(v) = rhs(v);
28:            for all s ∈ G.Neighbors(v) do
29:                UpdateVertex(s);
30:        else
31:            g(v) = ∞;
32:            for all s ∈ G.Neighbors(v) ∪ v do
33:                UpdateVertex(s);
```

```
1: function MAIN(G, s_goal, overlapExecution)
2:     s_last = s_current;
3:     INITIALIZE();
4:     COMPUTESHORTESTPATH();
   ▷ Same path reconstruction as in A* except each edge is checked for collision and reconstruction
             stops when an invalid edge is found in the path so the return is a collision-free
             subpath of the full path from search. A boolean is returned to indicate if the
             full path was reconstructed.
5:     path, isFull = RECONSTRUCTPATH(parentMap, s_current)
6:     optimizedPath, pathMap = OPTIMIZEPATH(path);
7:     pathIndex = 0;
8:     while s_current ≠ s_goal do
9:         nextSegment = optimizedPath[pathMap[pathIndex] : pathMap[pathIndex + 1]];
10:        excuteSegment = validPath = True;
11:        invalidNodes = ∅;
12:        if nextSegment is in collision then
13:            excuteSegment = validPlan = False;
14:            invalidNodes.Add(path[pathIndex]);
15:            invalidNodes.Add(path[pathIndex + 1]);
16:        for i from pathIndex + 1 to Size(path) − 1 do
17:            segment = optimizedPath[pathMap[i] : pathMap[i + 1]];
18:            if segment is in collision then
19:                validPath = False;
20:                invalidNodes.Add(path[i]);
21:                invalidNodes.Add(path[i + 1]);
22:        if overlapExecution = False and (validPlan = False or isFull = False) then
23:            excuteSegment = False;
24:        if excuteSegment = True then
25:            Move the robot along nextSegment and update s_current;
26:            k_m = k_m + h(s_last, s_current);
27:            s_last = s_current;
   ▷ Only scan for changes if their is not a valid full path to the goal
28:        if validPlan = False or isFull = False then
29:            Perform collision checks on all edges for nodes in invalidNodes and for the last node
               in path if isFull = False;
30:            for all edges (u, v) with changed collision status do
31:                UPDATEVERTEX(u);
32:                UPDATEVERTEX(v);
33:            COMPUTESHORTESTPATH();
```

Figure 2-17: Adapted D* Lite main and helper functions

```
1: function Main(G, s_goal, overlapExecution)
2:     for all e ∈ G.Edges do
3:         validEdges.Add(e);
4:     success, path = GetCollisionFreePath(G, s_current, s_goal, validEdges);
5:     if success = False then
6:         return False
7:     optimizedPath, pathMap = OptimizePath(path);
8:     pathIndex = 0;
9:     while s_current ≠ s_goal do
10:        nextSegment = optimizedPath[pathMap[pathIndex] : pathMap[pathIndex + 1]];
11:        excuteSegment = validPath = True;
12:        if nextSegment is in collision then
13:            excuteSegment = validPlan = False;
14:        for i from pathIndex + 1 to Size(path) − 1 do
15:            segment = optimizedPath[pathMap[i] : pathMap[i + 1]];
16:            if segment is in collision then
17:                validPath = False;
18:        if overlapExecution = False and (validPlan = False or isFull = False) then
19:            excuteSegment = False;
20:        if excuteSegment = True then
21:            Move the robot along nextSegment and update s_current;
    ▷ Only scan for changes if their is not a valid full path to the goal
22:        if validPlan = False then
23:            success, path = GetCollisionFreePath(G, s_current, s_goal, validEdges);
24:            if success = False then
25:                return False
26:            optimizedPath, pathMap = OptimizePath(path);
27:            pathIndex = 0;
```

Figure 2-18: A* Repair with incremental execution and execution monitoring. GetCollisionFreePath can be any of the four versions of A* Repair.

Chapter 3

Experiment Plan

The overarching goals for the experiments here are twofold. First, we want the experiments to guide the development of the roadmap-based motion planner. Second, we want to refine and verify our hypotheses regarding using a roadmap-based motion planner and precomputed solution cache in conjunction with an optimization-based motion planner. As a reminder, the planner should produce motions that are reactive and intuitive in changing environments. For our testing, we define reactive and intuitive to be to rapidly providing near-optimal, collision-free trajectories in a large majority of typical motion planning problems. The motion planner does not account for the motion of obstacles directly. Instead, it reacts to changing environments by taking a snapshot of the environment and computing a full plan quickly enough that the snapshot is still relevant. The motion planner should be able to solve most cases that will be encountered rather than to be able to solve every possible case on a longer timeline. To test this, realistic environments have been constructed along with test cases in those environments that are feasible for the motion planner to solve. With the environments characterized and the test cases for them developed, the first iteration of the motion planner can be constructed.

The core of the motion planner is a roadmap of robot states connected by edges containing the trajectories to traverse between the states. For each test environment, a separate roadmap must be constructed and maintained for modifications and path queries. The roadmaps built for these tests do not have any nodes or edges in collision

with the obstacles in the environment, and there are no dynamic obstacles. There are many hyper-parameters associated with the construction of these roadmaps, so it is important to have a good framework for comparing and conducting experiments on multiple roadmaps. This first iteration of the roadmap-based motion planner is measured by its ability to quickly provide high quality seed trajectories to an optimization-based motion planner for as many test cases as possible. What high quality means will be explained later in this chapter.

Our group is also interested in exploring the use of semantic information about the environment to guide the selection of nodes in the roadmap. The goal for this exploration is to improve the likelihood of solving motion planning problems that involve certain objects of interest or that require avoiding obstacles in a constrained environment. The experiments conducted for this thesis regarding semantic information are preliminary, and it is unclear when or if future research will be pursued for this topic.

The next focus for my experiments will be testing how well motion planner avoids dynamic obstacles in the environment. For these experiments, the dynamic obstacles referred to are stationary obstacles inserted into the environment after the roadmap has been constructed for the initial or static environment. Offline and online strategies for the development and repair of shortest path solutions for the roadmap go hand in hand with the development of the experiments to test these shortest path strategies.

As these strategies are developed, the roadmaps will in a sense be trained to solve planning problems for the obstacle configurations used. In order to avoid, or at least be aware of, over-fitting the roadmap solution caches to the realistic obstacles, it is important to have different caches that have been trained across a range of values. The experiments are then conducted on each cache to observe trends in the relevant performance metrics as a solution cache is trained more.

The final collection of experiments for this thesis will surround demonstrating the the capabilities of the incremental search algorithm, D* Lite. What separates incremental search algorithms from more traditional search algorithms, like A*, is that the node expansion, or in my case collision checking, is interleaved with the

execution of the trajectory that is being actively being modified with new knowledge of the environment. In that regard, experiments must be developed to showcases improvements brought by the incremental search algorithm particularly in cases where the environment changes in the middle of execution.

For the experiments regarding dynamic obstacles in particular, the goal is to test the hypothesis that offline computation and caching of roadmap paths results in improved performance online. Additionally, the tests should show the benefit provided by reusing information both within a single planning problem and between planning problems. The hope is to observe noticeable differences in the success rate of the planner being able to produce a collision-free trajectory and the average duration of a motion planning query depending on the algorithm and solution cache used.

3.1 Description of the Robot and Testing Environments

The robot used for all experiments developed and conducted in simulated environments for this thesis is the Rethink Robotics Baxter. This is in spite of having developed controllers and surrounding ROS infrastructure to demonstrate the roadmap based motion planner on a Barrett WAM as well. The reason for limiting testing to one robot is to remove another element of variability when trying to compare results from different tests. We expect the findings uncovered through our experiments with Baxter to be generalizable to other robotic systems.

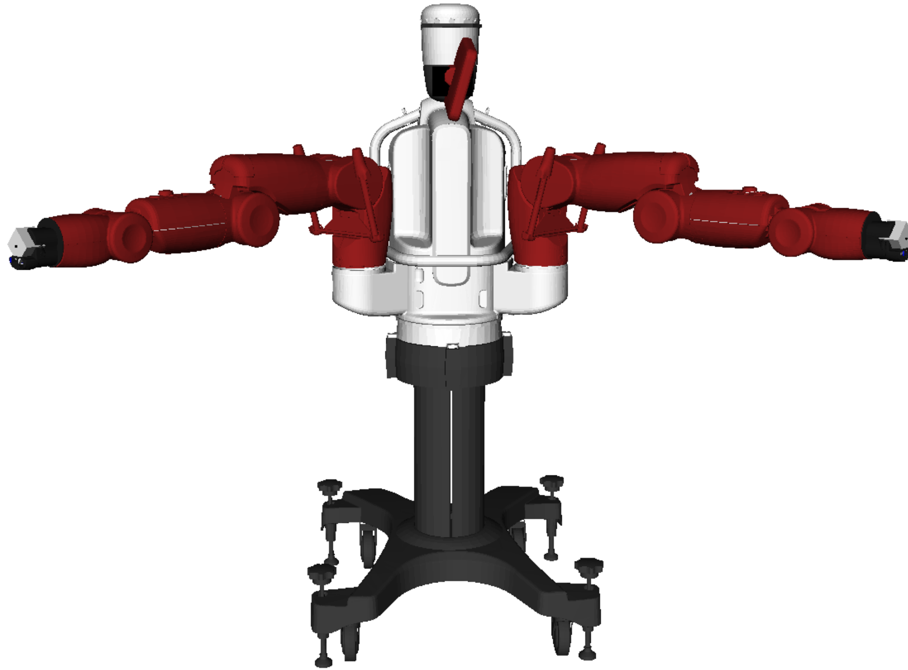
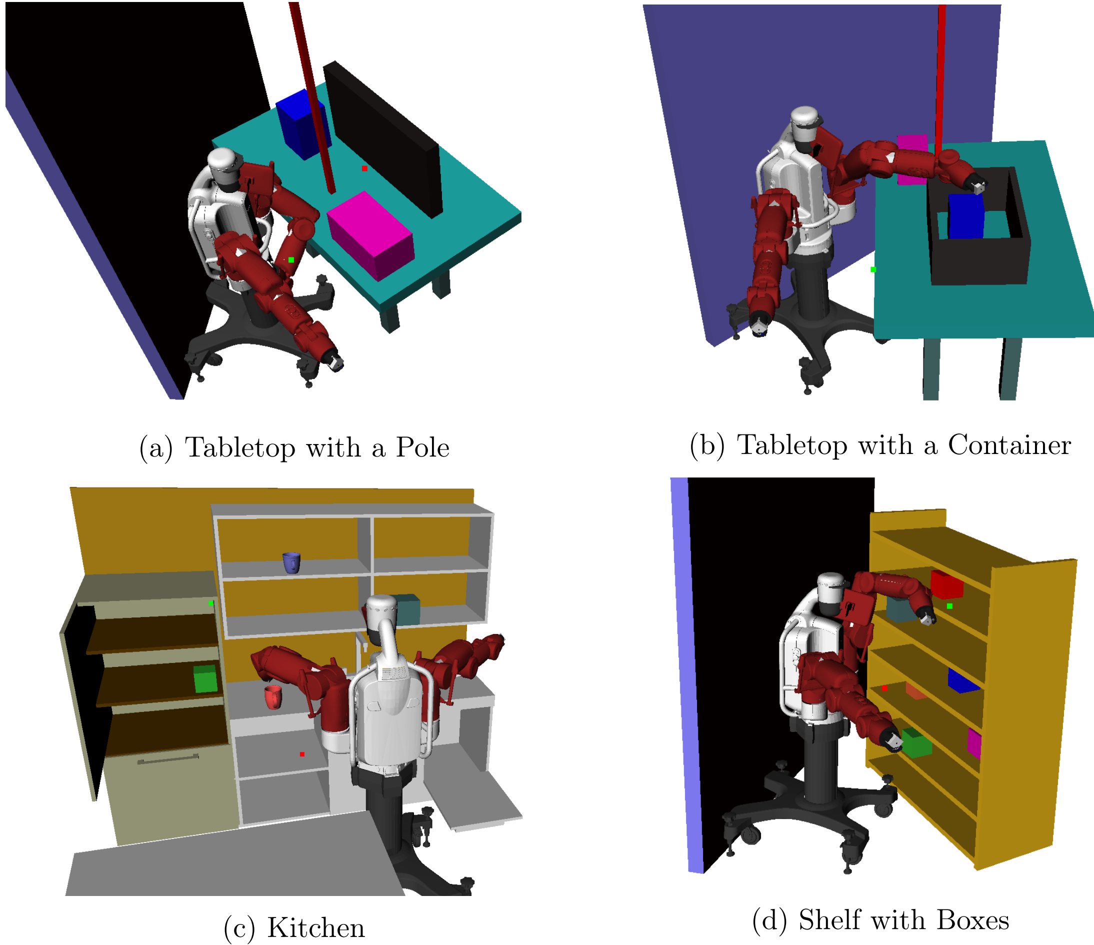


Figure 3-1: Rethink Robotics Baxter

Four practical environments have been developed for testing the performance of the roadmap-based motion planner [9]. The goal for the environments selected was to mimic settings the robot may be placed in as far as its position relative to large furniture objects, ie. a table or a shelf. From there, the environments are filled with obstacles of realistic size, shape, and orientation relative to the large furniture objects and one another, ie. placing a box on a tabletop. Three of these environments, Tabletop with a Pole, Tabletop with a Container, and Shelf with Boxes, were developed in OpenRAVE using collision objects by Sylvia Dai. The fourth, Kitchen, is provided by the TrajOpt package, but additional objects were placed in the stock environment for testing.

For the remainder of this thesis, I will order the environments by relative difficulty as determined by the experiments that will be explained in this chapter. That order is Tabletop with a Pole, Tabletop with a Container, Kitchen, and Shelf with Boxes. Visualizations of the environments can be seen below. Although this has not been specifically confirmed by experiments, it is generally believed that within our four



(a) Tabletop with a Pole

(b) Tabletop with a Container

(c) Kitchen

(d) Shelf with Boxes

Figure 3-3: The four testing environments used for all experiments

environments, difficulty is strongly correlated with how much surrounding space the robot has to maneuver around any individual collision object.

In addition to the development of environments for testing, 5000 test cases have been created for each of the four environments [9]. Each of these test cases has a feasible solution as determined by solving the cases with existing planners, like RRT. Therefore, we can reasonably assess the performance of the roadmap based motion planner in one of the four environments using the 5000 feasible cases corresponding to that environment.

3.2 Development of and Characterization the Roadmap Framework

The roadmap framework, implemented as the RoadmapManager, was developed to allow convenient modification of different roadmap hyper-parameters. This allows testing a range of values for a given hyper-parameter in order to find inflection points along the range for the different performance metrics used. The roadmap hyper-parameters initially considered relevant for testing were the number of nodes in the roadmap, the number of edges that connect each node in the roadmap, and the number of solutions between every pair of roadmap nodes in the all-pairs shortest path solution cache. As development and testing of the RoadmapManager ensued, it became clear that number of roadmap nodes used for attempting to connect query points to the roadmap and the number of interpolations to perform on an edge for collision checks are two additionally important hyper-parameters.

3.2.1 Tuning Roadmap Hyper-Parameters

The first hyper-parameter experiments focused on was the number of roadmap nodes. For each of the four testing environments, roadmaps were constructed with 250, 500, and 1000 nodes. These roadmaps were constructed using $k = 10$ when forming edges by connecting nodes to their k-nearest neighbors and $p = 1$ for creating the APSP solution cache with Yen's Algorithm [19]. Additionally, 2000 node roadmaps were constructed for the Kitchen and Shelf with Boxes environments after they were determined to be the two most difficult for connecting query points to the roadmap.

Across these different roadmaps, the number of edges scaled fairly linearly with the number of nodes, and at no point were there indications of having too many or not enough edges connecting nodes in the roadmap. As a result, no experiments were conducted comparing roadmaps with different values of k, and $k = 10$ has been used for all roadmaps constructed for this thesis. Similarly, having more than one path for a pair of roadmap nodes is only useful if the state of the environment when

handling roadmap queries is different than when the roadmap was constructed, so the majority of the roadmaps constructed for testing use $p = 1$ when constructing the APSP solution cache with Yen's Algorithm.

As for the additional two hyper-parameters, no series of experiments was targeted at testing either of them directly, but information gained from other experiments impacted how the hyper-parameters were treated. Take, for example, deciding how many roadmap nodes to attempt connections to when trying to connect a query point to the roadmap. The trade-off at hand is to limit the amount of time spent performing collision checking before returning a failure for a query versus limiting the number of false negatives where a query node could have been connected to the roadmap had more nodes been attempted. Early experiments made it clear that only checking the nearest $m = 10$ roadmap nodes yielded too many false negatives, but allowing all roadmap nodes to be checked yielded cases for the 1000 node roadmaps where the RoadmapManager would spend more than 5 seconds collision-checking before returning a failure. From these observations it was determined that $m = 100$ would be the maximum number of roadmap nodes that a query point would attempt to connect to before a failure would be returned for a query.

Determining how densely to interpolate edges when checking them for collisions is a little more complicated of a matter. For many of my roadmap experiments, online collision checking is a bottleneck that slows down the overall times for the RoadmapManager by an order of magnitude. My research group is aware that collision checking in the context of our motion planning problems is highly parallelizable and would lend itself to GPU programming quite well. However, collision checking is not within the scope of our research interests, so at this point in time, my group has no plans to implement such a collision checking system.

As a result, simpler fixes were developed for speeding up our sequential collision checking for online motion planning queries. The first of these is to check an edge sparsely for collision before checking it densely. This allows the RoadmapManager to quickly move on from edges that are very evidently in collision. This approach is used when generating edges during roadmap construction, when attempting to

connect query points to roadmap nodes with collision-free edges, and when validating that edges in the roadmap are collision-free at the time of a query. The sparse check performs a collision check at each of 10 interpolated joint states for the end nodes of a given edge, while the dense check performs collision checks at 1000 interpolated joint states during offline roadmap construction and at 100 interpolated joint states during online motion planning problems.

Over the course of the experiments evaluating roadmap performance in the static environments, a small number of cases arose where an edge in the roadmap would be improperly marked collision-free at the time of a query. This bug was soon identified to be due to the fact that all roadmap edges were interpolated the same number of times for collision checks, meaning longer edges had larger spaces in between collision checks and could potentially miss a collision in one of those spaces. This was addressed by modifying edge collision checking to interpolate an edge 10, 100, and 1000 times per radian of euclidean distance traversed by the edge for sparse, dense online, and dense offline checks respectively. With this modification, no additional cases have been observed of improperly marked collision-free during online motion planning queries.

3.2.2 Characterizing Roadmap Performance

In order to gain insight into the impact of different values for roadmap hyper-parameters, it was important to establish a set of performance metrics to compare roadmaps that reflect the goals for the motion planner. Again, the planner should be able to solve most typical motion planning problems with motions that feel reactive and intuitive.

With that in mind, the first performance metric used to compare roadmaps is the rate of failure to produce a collision-free roadmap solution. Often this is caused by the lack of a straight-line, collision-free connection from the start or end points in a motion planning query to existing nodes in the roadmap. For experiments run in the unchanged static environments, this is the only way for this kind of failure to occur because all roadmap edges are collision-free for the static environment. However for experiments with changing environments, this can also occur when there is no collision-free solution in the cache for the pair of nodes connected to. Depending on

the experiment being performed, the RoadmapManager may return an in-collision solution or no solution at all for these kinds of failures.

The next two metrics are the average duration of a roadmap query and the average euclidean length of solutions returned by the RoadmapManager. The bulk of the time for a roadmap query is consumed by collision checking, so comparisons for average query duration between different roadmaps or query strategies are mainly exploring the trade-off of how many edges to check before returning a failure for a query. Path length of a roadmap solution is fairly straightforward metric that is primarily impacted by the density of the roadmap, but experiments have also been conducted to show that it is impacted by the choice of roadmap nodes to connect to start and end query points. While the path length of the roadmap seed is not necessarily reflected in the resulting trajectory after optimizing with TrajOpt, our experiments have indicated that there is correlation between the length of a seed trajectory and the amount of time TrajOpt takes to optimize that seed.

The last two metrics used to compare roadmap performance are obtained after the RoadmapManager has produced a seed trajectory. Those metrics are the average duration of the optimization performed by TrajOpt and the average euclidean length of the resulting optimized trajectory. While TrajOpt is expected to terminate fairly quickly when provided a collision-free seed, its duration is important to track because it contributes to the total query duration for our motion planner. Finally, the length of the optimized trajectory is important because that is the trajectory that is actually being outputted by our motion planner and executed by the robot.

3.3 Semantic Sampling to Improve Roadmap Connectivity and Other Side Explorations

The primary experimental explorations regarding offline methods surround tuning roadmap parameters and developing solution cache augmentation and repair strategies, but there are a couple tangential explorations containing smaller sets of ex-

periments in order to potentially provide directions for future research. The first of these explorations is the aforementioned use of semantic information to improve the connectivity of a roadmap. Each roadmap tested is an augmentation on the same base 1000 node roadmap for the "Shelf with Boxes" environment. This exploration was only conducted for the most difficult of the four environments. The different augmentations are as follows:

- Baseline Control: The base 1000 node roadmap with 3 nodes pruned for being disconnected from the main subgraph giving the roadmap 997 nodes
- Just Outside the Shelf: End-effector poses in a planar grid just outside the shelf were checked for valid IK solutions and resulted in 123 joint states added giving the roadmap 1120 nodes
- Just Within the Shelf: Poses in a planar grid just within the shelf were checked with 61 valid joint states added giving the roadmap 1058 nodes
- Halfway in the Shelf: Poses in a planar grid halfway in the shelf were checked with 42 valid joint states added giving the roadmap 1039 nodes
- Additional Samples Control: An additional 42 valid nodes were randomly sampled and added to the base roadmap giving the roadmap 1039 nodes

These roadmaps were each tested for the static environment using the 5000 feasible cases. Nodes added to the base roadmap were demarcated with an abbreviated label for the augmentation performed to obtain those points. When these augmented roadmaps are queried, a label is provided to indicate to the RoadmapManager that if no connection can be made from a query point to the nearest m roadmap nodes, roadmap nodes containing that label will be additionally checked regardless of whether or not they are in the nearest m. The idea is that roadmap connectivity can be improved if a roadmap contains nodes specific to objects in the environment and then the RoadmapManager is provided indication that the query involves a specific object.

The second of these explorations, and the less involved of the two, regards the implementation of a sorting heuristic when attempting to connect query points to

roadmap nodes during a motion planning query. When the nearest m nodes are obtained for connecting to either the start or the end point for a query, they are normally in order of increasing distance to whichever point is being connected to, start or end. The implemented sorting heuristic takes the nearest m nodes and sorts them again by the sum of their distances to both the start and end points for the query. The idea behind this is that the query points will first attempt to connect to roadmap nodes that are more along the way to one another, so the resulting roadmap trajectory will be shorter.

The associated experiments did not involve any roadmap augmentation. All that was different is a flag indicating that the sorting heuristic should be used for a query. The performance metrics particularly important to compare to the control of not performing an additional sort are the average query duration and roadmap trajectory length. However, all performance metrics used for comparing roadmaps with different hyper-parameters are recorded for these experiments as well.

3.4 Incorporating Dynamic Obstacles to Augment and Evaluate Roadmap Solution Caches

A significant portion of the methods developed for this thesis involve strategies for augmenting the solution cache of a roadmap to better account for obstacles introduced into the environment after the roadmap has been constructed for the static environment. These strategies are largely focused on using a set of realistic obstacles that are assumed to be representative of the obstructions the robot is expected to encounter in the real world despite being a fairly sparse set of geometries and poses. These realistic obstacles are inserted into the environment to block roadmap edges, so paths can be computed between pairs of nodes that hopefully differ significantly from the shortest path in the static environment.

This brings up the issue of how to effectively evaluate these roadmaps while using the same sparse set of obstacles. To generate experimentally meaningful results, it is

important to apply the same set of tests to each roadmap. This means that there can be no probabilistic aspect to how the environments are modified. For contrast, when training roadmap solution caches with repeated generations of an APSP solution set, the environment is modified at each iteration by inserting a set of object-pose tuples using a binomial distribution for sampling.

The test set for evaluating the ability of a roadmap to plan around an obstacle inserted before the start of planning is also based around the 5000 feasible cases for each of the four practical environments. Each case is given to the RoadmapManager as a planning query for the static environment. If a plan is returned, it means that the start and end joint states can be connected to the roadmap. Each of these successful cases is then repeatedly tested with a different object-pose tuple inserted into the environment at each iteration for a total of 40 additional iterations for the case with a single added obstacle in the environment for each test.

The solution cache training only involves a couple different cache augmentation approaches but ranges on the number of rounds used to train the solution cache for a given roadmap. Additionally, there is a lot of similarity between the environment configurations used for training and evaluating the performance of the different trained solution caches. Because of this, what is most interesting in the results for these experiments are the trends and inflection points across a range for a given roadmap and augmentation strategy. In particular, the trends are on how many cases the roadmap is still able to find or produce a collision-free solution and how long on average does it take to produce that solution or return a failure. Some thought was given to inserting multiple obstacles for a single test, but the decision was made to constrain the testing to hopefully make the trends more apparent.

3.5 Creating Experiments to Expose the Benefits of Incremental Planning

All of the experiments described up to this point measure motion planning performance when planning from scratch in some environment configuration. For many of the experiments, the environment configuration is the static configuration, so the constructed roadmaps are entirely collision-free. The others have involved inserting realistic obstacles to obstruct the roadmap in ways that are representative of what would be encountered in the real world. For those experiments, the goal is to demonstrate how offline computation and caching of roadmap paths can benefit online motion planning performance even when faced with dynamic obstacles.

However, the perceived benefits are demonstrated through fast planning in configurations that differ from the static environment without any regard for replanning in cases where an initial plan becomes invalid. For contrast, as previously mentioned, the goal for incremental search algorithms like D* Lite is to reuse information from a previous search to speed up subsequent searches as the search result is executed and knowledge of the environment changes [14]. Because of that, it does not make sense to compare an incremental search algorithm to a non-incremental search algorithm in experiments that surround planning from scratch in different environment configurations.

Instead, the experiment to asses performance of an incremental search algorithm should force replanning by invalidating an initial plan during execution. This idea is realized by obtaining an initial plan from a RobotClient method wrapping a given search algorithm with with incremental trajectory execution and execution monitoring. The plan obtained from the RobotClient is used to identify the end-effector pose at a waypoint that is halfway along the trajectory as determined by the total euclidean distance of the trajectory in configuration space. A cube with a 10cm side-length is then inserted into the environment with its center at the identified end-effector pose. This is guaranteed to invalidate the initial plan, which in turn forces replanning by the search algorithm in use.

As mentioned in the Methods Developed chapter, one of the innovations for these implemented adaptations of incremental algorithms is the interleaving of replanning with execution. In accordance, it was important to establish a metric that highlights that innovation. That metric, "Effective Planning Overhead", is the difference between the total measured time of planning and execution and the estimated time of just execution. Planning and execution time is measured from when the initial plan is completed to when the final trajectory segment has been executed. Execution time alone is estimated from the joint displacements between consecutive waypoints of executed trajectory segments and the known maximum joint velocities of the robot controller. The final metric of interest for these tests is the length of the executed trajectory measured by euclidean distance in configuration space.

The two algorithm implementations evaluated for these experiments are Adapted D* Lite and A* Repair 1. For each of these implementations, a flag can be used to toggle the interleaving of replanning and execution. When interleaving is disabled and replanning is required, the implementations will not send the next segment of the previous trajectory to be executed even when the segment is still collision-free. For each of the two implementations, serially replanning and then executing a trajectory is used as the de facto base case to illustrate the benefits of interleaving planning and execution.

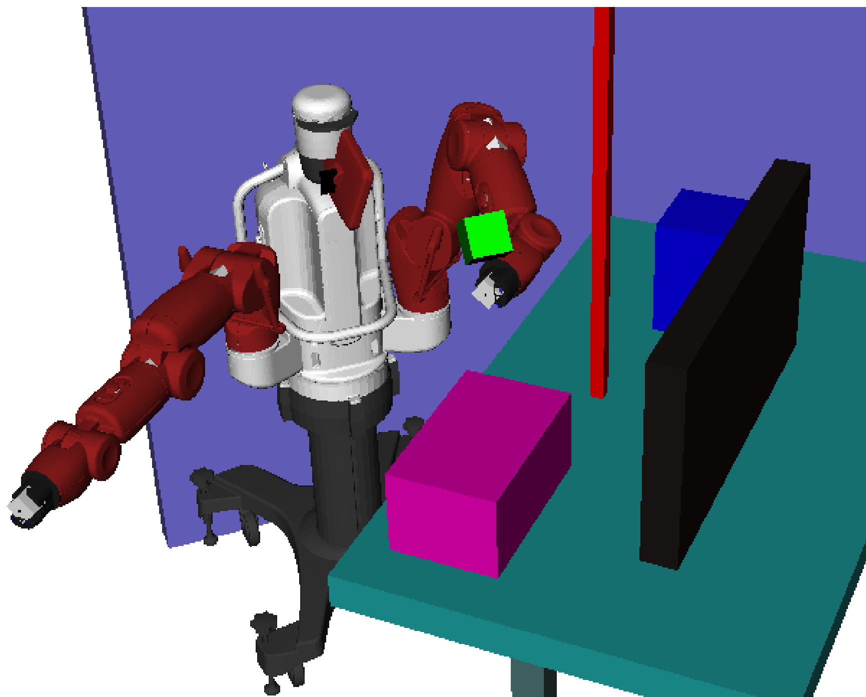


Figure 3-4: Environment visualization of Tabletop with a Pole for incremental planning analysis

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Experiment Results

4.1 Roadmap and TrajOpt Performance

The first set of experiment results evaluates roadmap performance in each of the four static environments. For these tests, no collision checks are performed on roadmap edges because the environment is assumed to be unchanged from the static environment. Since the roadmaps are constructed to be collision-free in the static environment, all nodes and edges in the roadmap will be collision-free for these tests. This means all failures in these experiments are caused by not being able to make a straight-line, collision-free connection from the start or end point for a case and an existing node in the roadmap. Additionally, all paths returned by the roadmap come from a precomputed solution cache, so the majority of the time for a roadmap query is consumed by establishing the collision-free connections to the roadmap.

What is of particular interest for these tests is how roadmap performance changes with the number of nodes in the roadmap. From the Figure 4-1, a clear trend emerges that the roadmap can be connected to as more nodes are sampled for a given roadmap. This should not come as a surprise, but what is surprising is the percentage of successful cases for the sparser roadmaps. With the exception of the Shelf with Boxes environment, a 250 node roadmap can connect to more than 95% of tested cases. This supports the hypothesis that a sparse roadmap can be used to cover enough of the reachable workspace for a robot to account for a significant majority of typical

planning problems.

Other trends in the data are less apparent, so the other metrics are likely less dependent on the size of a roadmap. However, it should be noted from Table 4.3 that the average path length of of roadmap seed trajectories decreases as the size of the roadmap increases in most cases as does average length of the optimized trajectories.

Environment	Number of Roadmap Nodes	Failure Rate	Average Runtime (s)	Average Path Length (rad)
Tabletop with a Pole	250	0.22%	0.1522	1.260
	500	0.18%	0.1596	1.284
	1000	0.18%	0.1434	1.238
Tabletop with a Container	250	1.46%	0.1846	1.354
	500	1.40%	0.2054	1.310
	1000	0.76%	0.1806	1.320
Kitchen	250	2.58%	0.3832	1.282
	500	2.80%	0.4248	1.289
	1000	1.92%	0.3792	1.285
	2000	1.58%	0.3978	1.280
Shelf with Boxes	250	18.34%	0.4052	1.316
	500	15.50%	0.4456	1.308
	1000	12.06%	0.3876	1.302
	2000	10.20%	0.3434	1.283

Table 4.1: Roadmap performance assuming a static environment

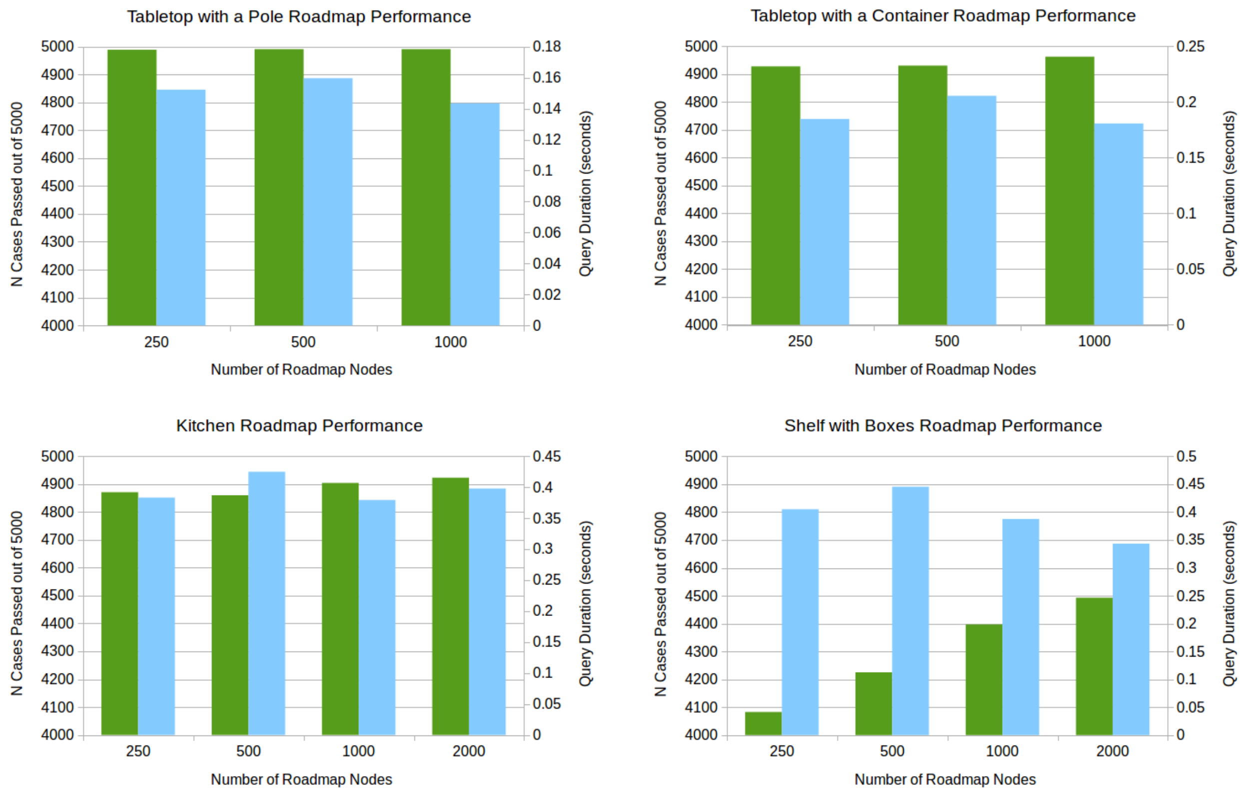


Figure 4-1: Roadmap performance assuming a static environment. No collision checks were performed on roadmap edges as a result of the assumption. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue.

Environment	Number of Roadmap Nodes	Average Roadmap Path Length (rad)	Average TrajOpt Runtime (s)	Average Optimized Path Length (rad)
Tabletop with a Pole	250	1.260	0.4348	0.873
	500	1.284	0.5029	0.853
	1000	1.238	0.4456	0.821
Tabletop with a Container	250	1.354	0.5096	1.042
	500	1.310	0.5471	1.012
	1000	1.320	0.5185	1.017
Kitchen	250	1.282	0.6943	0.8505
	500	1.289	0.713	0.8597
	1000	1.285	0.7039	0.8572
	2000	1.280	0.8836	0.8343
Shelf with Boxes	250	1.316	0.6072	1.034
	500	1.308	0.6187	1.028
	1000	1.302	0.6138	1.020
	2000	1.283	0.9056	0.9677

Table 4.3: TrajOpt performance when seeded with roadmap trajectories

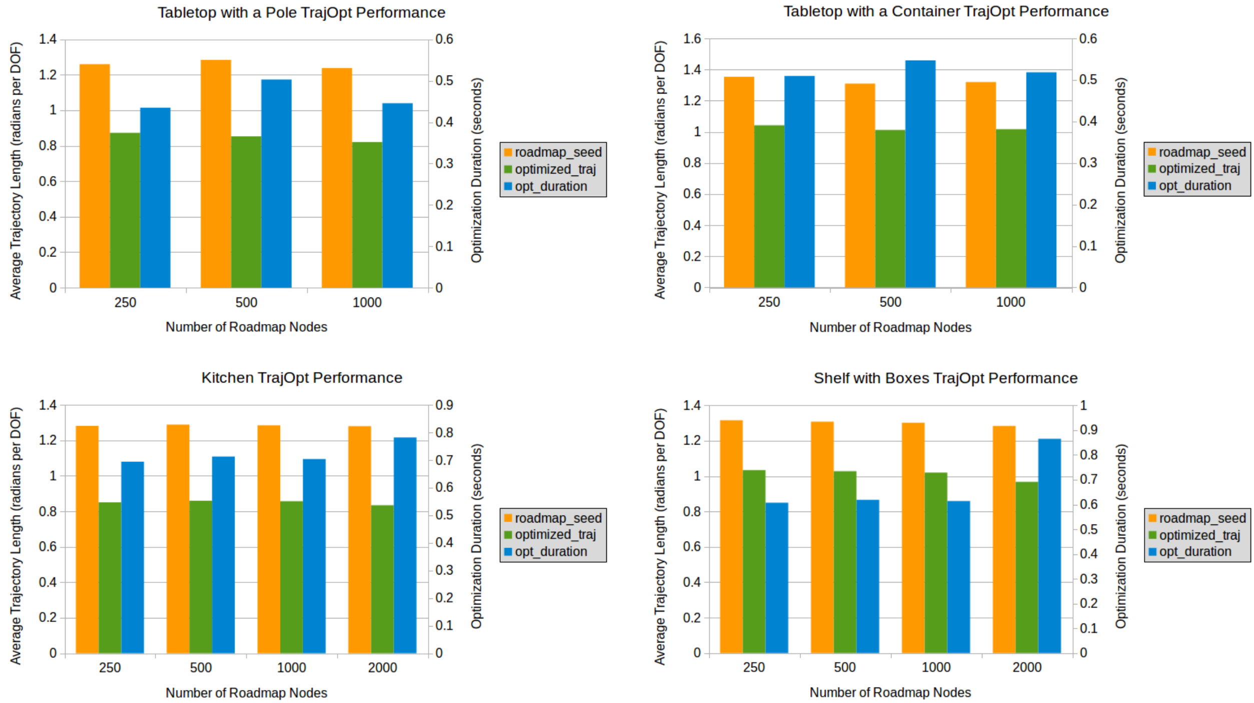


Figure 4-2: TrajOpt performance when seeded with roadmap trajectories. TrajOpt is provided a collision-free seed from the roadmap in all cases and a static environment is assumed with no collisions in the roadmap.

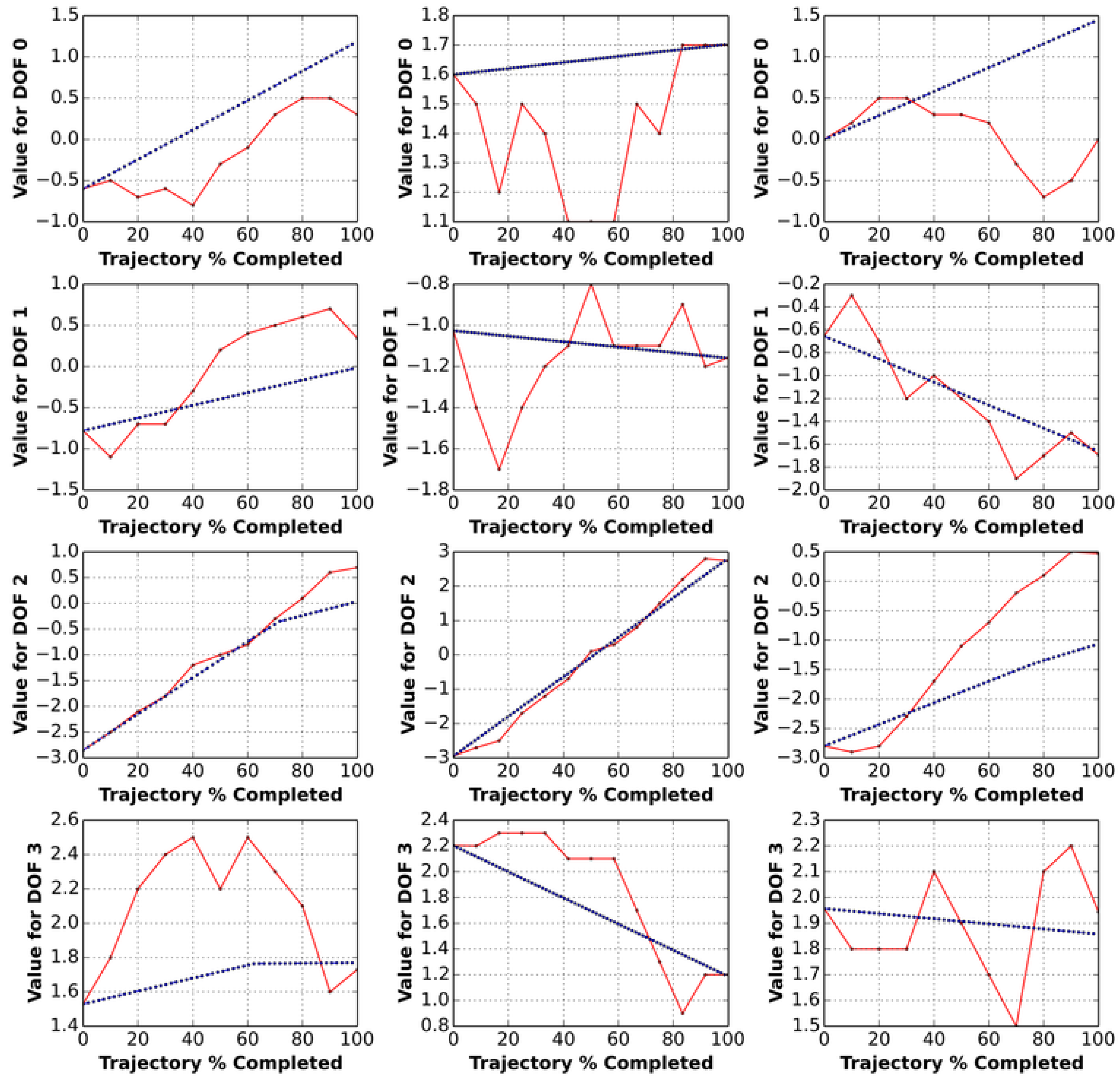


Figure 4-3: Graphs showing the difference between a seed trajectory from a roadmap and an optimized trajectory from TrajOpt for three different cases. Values are only shown for the first 4 DOFs because the roadmap nodes have the same fixed values for the remaining DOFs. In each plot, the roadmap trajectory is shown in red and the optimized trajectory is shown in blue. TrajOpt is provided a pose target rather than a joint target for these experiments, so the end joint state for an optimized trajectory may differ from the corresponding roadmap trajectory.

4.2 Semantic Sampling and Sorting Heuristic

The first table and graph pair shown is for an experiment testing the benefit of adding a small number of points to the roadmap that were selected using semantic information from the environment. Specifically, the different sets of these points were selected due to their proximity to the shelf in the Shelf with Boxes environment. This is after it was hypothesized that this environment was the most difficult of the four due to much of the reachable workspace being divided by the shelves and then further cluttered by collections of boxes that must be maneuvered around. As a reminder, the sets of additional points were developed by testing a grid of end-effector poses near the shelf for valid IK solutions. An additional control roadmap was also constructed by randomly sampling additional points for the base roadmap.

Despite finding the fewest number of valid IK solutions when testing the grid of poses halfway in the shelf, these points provided the largest increase in additional cases passed compared to the control roadmap. This supports the hypothesis that many of the failures were caused by an inability to interact around the shelf in close proximity as well as the hypothesis that a small number of well selected points can have a dramatic improvement on roadmap performance.

It should be noted that the dramatic increase in average query duration for these results when compared with the previous section is due to checking the full roadmap solutions for collisions before they are returned. These collision checks will be performed for all experiments moving forward to guarantee the satisfaction of the requirement that TrajOpt must be provided a collision-free.

Modification Description	Number of Roadmap Nodes	Succeeded with Standard Query	Succeeded when Providing Object Name	Average Query Duration (s)
Control with Disconnected Subgraphs Pruned	997	4397	4397	1.566
42 Randomly Sampled Nodes added to Control Roadmap	1039	4405	4405	1.662
42 Nodes added Halfway in the Shelf	1039	4480	4512	1.716
61 Nodes added Just Within the Shelf	1058	4464	4482	1.745
123 Nodes added Just Outside the Shelf	1120	4427	4456	1.744

Table 4.5: Performance comparison between roadmaps with different sets additional points added to the same base 1000 node roadmap

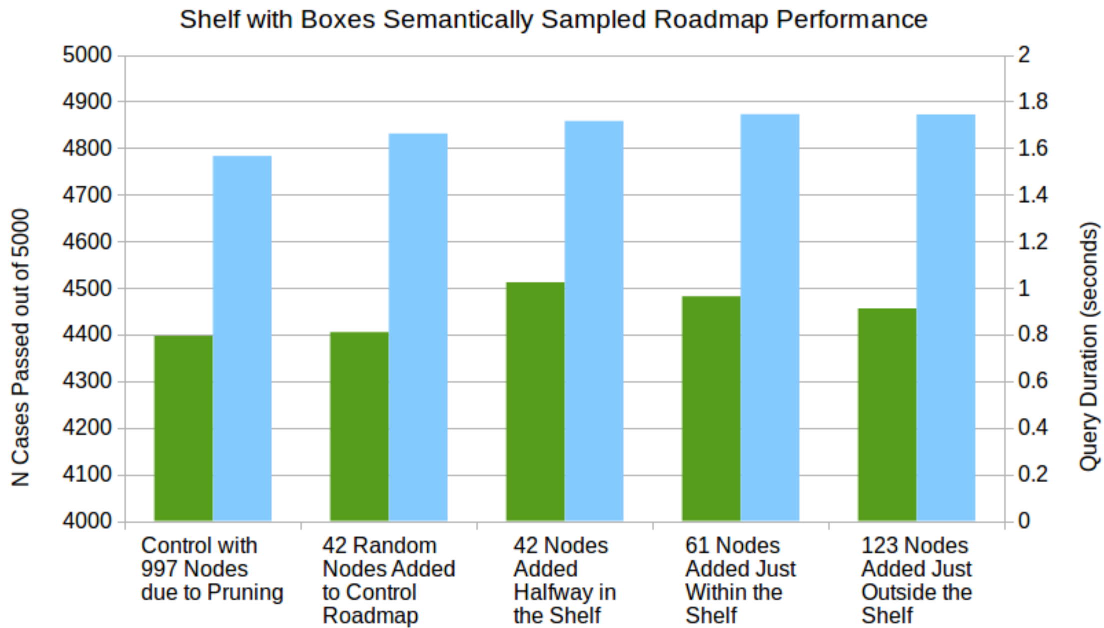


Figure 4-4: Performance comparison between roadmaps with different sets additional points added to the same base 1000 node roadmap. Only modifications that add points specific to the shelf support queries where "shelf" is provided to guide attempted connections to the roadmap. Roadmap paths are checked for collisions before they are returned although the static environment will not push any roadmap edges into collision. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue.

The goal for the sorting heuristic experiments was to identify if approaches should be explored for performing additional work at the start of a query to guide how the query is handled. The sorting heuristic orders roadmap nodes in terms of their summed distance from both the start and end points for a query instead of either the start point or the end point. This decreases the length of the roadmap seed trajectories provided to TrajOpt at the expense of the time taken to sort the roadmap nodes for each attempted connection. The roadmap nodes connected to when using the sorting heuristic will tend to be farther away from the point attempting to connect to the roadmap since the roadmap nodes will no longer be checked in order of increasing distance from the point. This could result in delays caused by having to check more nodes on average in order to establish the collision-free connection.

Looking at the average durations in Figure 4-6, we can see that the use of a sorting heuristic increases the average duration for the two tabletop environments but decreases the average duration for the Kitchen and Shelf with Boxes environments. I believe that the increase in query duration for the tabletop environments is caused by the reasons stated in the previous paragraph. The results for the Kitchen and Shelf with Boxes environments are contrary to expectation, but I believe uncover a valid result. I believe that the decrease in average query duration is the result of the decrease in average path length for roadmap seed trajectories. Collision checking in these two environments is more time consuming than in the tabletop environments, so while their is an increase in average duration caused by sorting and potential extra connection attempts, this is offset by the overall decrease in required collision checking due to a shorter average path length.

That being said, the difference in optimization duration for TrajOpt could lend itself to a different interpretation of the results. While these differences are smaller than for the average roadmap query duration, they have the same trend across the four environments. This could indicate a similar narrative about collision checking with the speculative addendum that TrajOpt took longer when using the sorting heuristic in the tabletop environments due to increased proximity to obstacles for more aggressive seed trajectories. However, this could also indicate that the computer

used for testing was under greater stress for the experiments with increased TrajOpt duration, which would imply that conclusions should not be drawn from the average duration results.

The shorter paths produced by the roadmaps when using the sorting heuristic are likely the primary cause for the shorter optimized paths produced by TrajOpt. The differences in average path length for the optimized paths are not pronounced enough that the sorting heuristic should be considered universally beneficial, but the results here by and large indicate that there are likely scenarios where it is beneficial to perform additional work at the start of a roadmap query to guide how it is handled.

Environment	Roadmap Description	Failure Rate	Average Runtime (s)	Average Path Length (rad)
Tabletop with a Pole	500 Node Roadmap Control	0.18%	1.033	1.284
	500 Node Roadmap with Sorting Heuristic	0.18%	1.241	1.073
	1000 Node Roadmap Control	0.18%	1.001	1.238
	1000 Node Roadmap with Sorting Heuristic	0.18%	1.182	1.024
Tabletop with a Container	500 Node Roadmap Control	1.40%	1.170	1.310
	500 Node Roadmap with Sorting Heuristic	1.40%	1.413	1.142
	1000 Node Roadmap Control	0.80%	1.159	1.320
	1000 Node Roadmap with Sorting Heuristic	0.80%	1.372	1.132
Kitchen	500 Node Roadmap Control	2.84%	2.441	1.289
	500 Node Roadmap with Sorting Heuristic	2.84%	2.186	1.059
	1000 Node Roadmap Control	1.94%	2.279	1.285
	1000 Node Roadmap with Sorting Heuristic	1.94%	2.089	1.059
Shelf with Boxes	500 Node Roadmap Control	15.56%	1.569	1.308
	500 Node Roadmap with Sorting Heuristic	15.56%	1.299	1.139
	1000 Node Roadmap Control	11.98%	1.566	1.302
	1000 Node Roadmap with Sorting Heuristic	11.98%	1.282	1.134

Table 4.6: Roadmap performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap

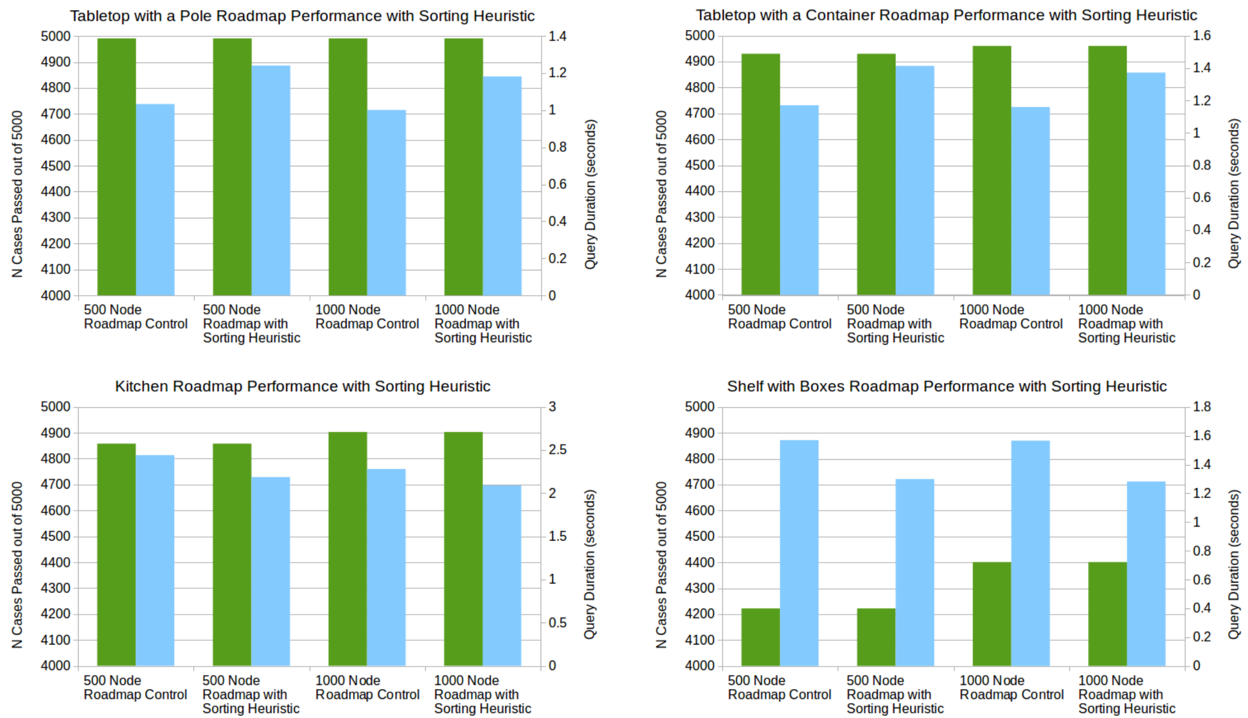


Figure 4-6: Roadmap performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap. For each roadmap, the number of cases that had a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue.

Environment	Roadmap Description	Average Roadmap Path Length (rad)	Average TrajOpt Runtime (s)	Average Optimized Path Length (rad)
Tabletop with a Pole	500 Node Roadmap Control	1.284	0.5672	0.8219
	500 Node Roadmap with Sorting Heuristic	1.073	0.5967	0.7554
	1000 Node Roadmap Control	1.238	0.5662	0.7694
	1000 Node Roadmap with Sorting Heuristic	1.024	0.6002	0.6887
Tabletop with a Container	500 Node Roadmap Control	1.310	0.6868	0.9586
	500 Node Roadmap with Sorting Heuristic	1.142	0.7860	0.8812
	1000 Node Roadmap Control	1.320	0.7315	0.9533
	1000 Node Roadmap with Sorting Heuristic	1.132	0.8014	0.8734
Kitchen	500 Node Roadmap Control	1.289	0.7587	0.8514
	500 Node Roadmap with Sorting Heuristic	1.059	0.5912	0.7783
	1000 Node Roadmap Control	1.285	0.7732	0.8457
	1000 Node Roadmap with Sorting Heuristic	1.059	0.5982	0.7789
Shelf with Boxes	500 Node Roadmap Control	1.308	0.8415	0.9915
	500 Node Roadmap with Sorting Heuristic	1.139	0.6329	0.9076
	1000 Node Roadmap Control	1.302	0.8639	0.9813
	1000 Node Roadmap with Sorting Heuristic	1.134	0.6474	0.8959

Table 4.7: TrajOpt performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap

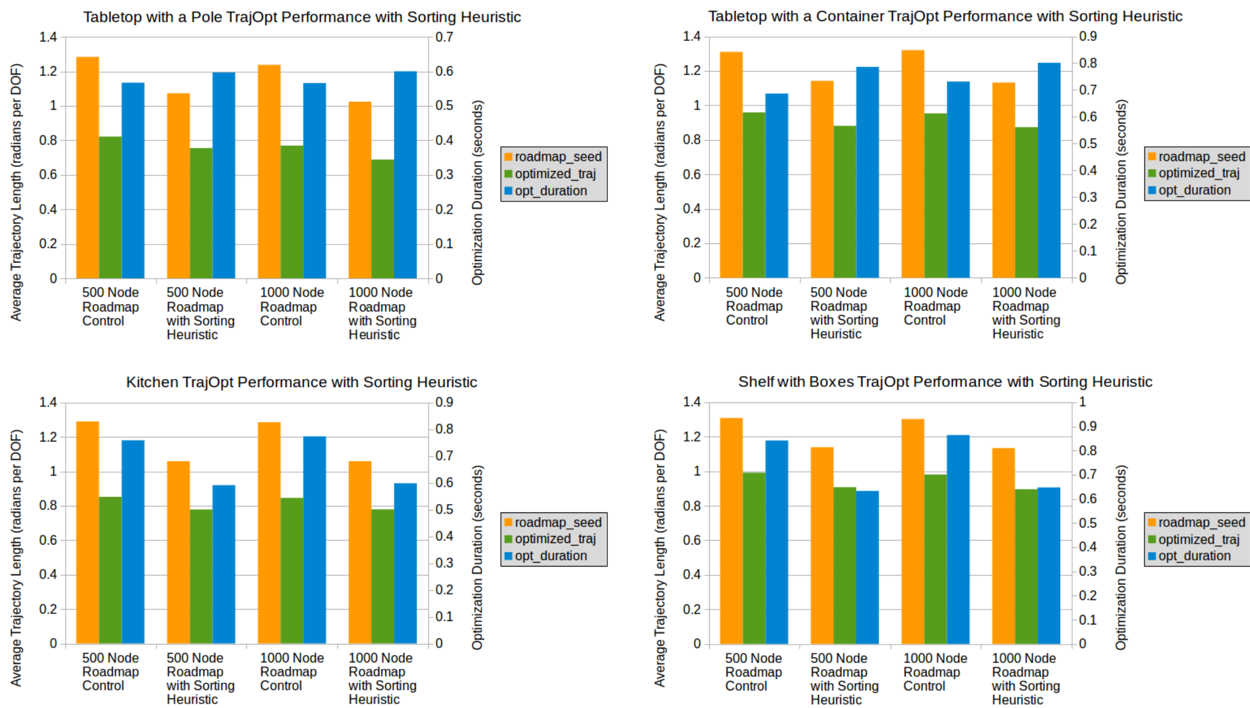


Figure 4-8: TrajOpt performance comparison to determine the effects of using a sorting heuristic to guide connection to the roadmap

4.3 Training and Testing Results for Obstacle Insertion Experiments

4.3.1 APSP Training Results

The first experiments involving the insertion of the realistic obstacles are to evaluate the solution caches trained different amounts using the APSP Training approach. These experiments are exclusively concerned with whether or not a collision-free solution can be found and how long the RoadmapManager takes to make that decision. For testing solution caches constructed with APSP Training, that first metric is evaluated only by checking solutions in the cache, but for experiments later in this section involving A*, that metric will be evaluated by whether or not a collision-free solution exists in the roadmap at all.

In Table 4.8, Original Solution Collides is tracked both to verfiy that the experiment runs properly and to provide context for the adjacent metric, Collision-free Solution Found. All experiments in a given environment use the same roadmap but with different amounts of training for the solution caches, so the original cached solution should be the same path for all caches. When that solution is invalidated by the insered obstacle, the RoadmapManager is queried for a collision-free solution in the updated environment configuration. The Control Roadmap is still able to find collision-free solutions in many of these cases because the start and points for the query connect to different roadmap nodes than in the static environment case. The RRT Comparison uses the RRT implementation provided by OMPL [20] capped at 100 iterations for time, and some cases were lost due to non-deterministic failures in the off the shelf planner.

Unsurprisingly, as solution caches are trained more with the realistic obstacles, they in general are then more able to avoid the same obstacles during testing. What is interesting to note is that Average Runtime does not significantly increase with the number of training rounds in all instances. This leads me to believe that the extra collision checking performed on edges for the additional cached solutions impacts the

query duration less than the time required to initially connect the start and end points for the query to nodes in the roadmap. What is interesting to note is that the greatest payoff in increased performance is in the first five rounds of APSP Training and performance effectively levels off after 15 rounds of APSP Training.

Environment	Number of Training Rounds	Original Cached Solution Collides	Collision-free Solution Found	Average Runtime (s)
Tabletop with a Pole	RRT Comparison	84085	22129	2.781
	0 Rounds (Control Roadmap)	84086	9881	0.5041
	5 Rounds	84086	28360	0.5851
	10 Rounds	84086	31344	0.5653
	15 Rounds	84086	34987	0.6242
	20 Rounds	84086	34955	0.6396
	25 Rounds	84086	32912	0.6886
Tabletop with a Container	RRT Comparison	85859	17355	3.035
	0 Rounds (Control Roadmap)	85860	9118	0.4982
	5 Rounds	85860	20956	0.4894
	10 Rounds	85860	22607	0.4965
	15 Rounds	85860	27517	0.5710
	20 Rounds	85860	27567	0.5698
	25 Rounds	85860	28343	0.5684
Kitchen	RRT Comparison	78810	21764	2.874
	0 Rounds (Control Roadmap)	78848	6499	0.9681
	5 Rounds	78848	17472	0.9071
	10 Rounds	78848	20810	1.020
	15 Rounds	78848	23527	1.078
	20 Rounds	78848	22907	1.110
	25 Rounds	78848	23616	1.086
Shelf with Boxes	RRT Comparison	64807	5756	1.821
	0 Rounds (Control Roadmap)	64832	4096	0.6046
	5 Rounds	64832	10300	0.5139
	10 Rounds	64832	14091	0.5918
	15 Rounds	64832	13355	0.5934
	20 Rounds	64832	14761	0.6235
	25 Rounds	64832	14968	0.6254

Table 4.8: APSP Training obstacle avoidance data for 500 node roadmaps. Original Cached Solution Collides serves as a validation of the experiment because the same roadmap is used for every test in a given environment but with different solution caches. RRT Comparison describes the use of RRT in OMPL with 100 maximum iterations to find a solution when the original roadmap solution is in collision. Some cases were lost for the RRT Comparison due to non-deterministic failures with the OMPL planner.

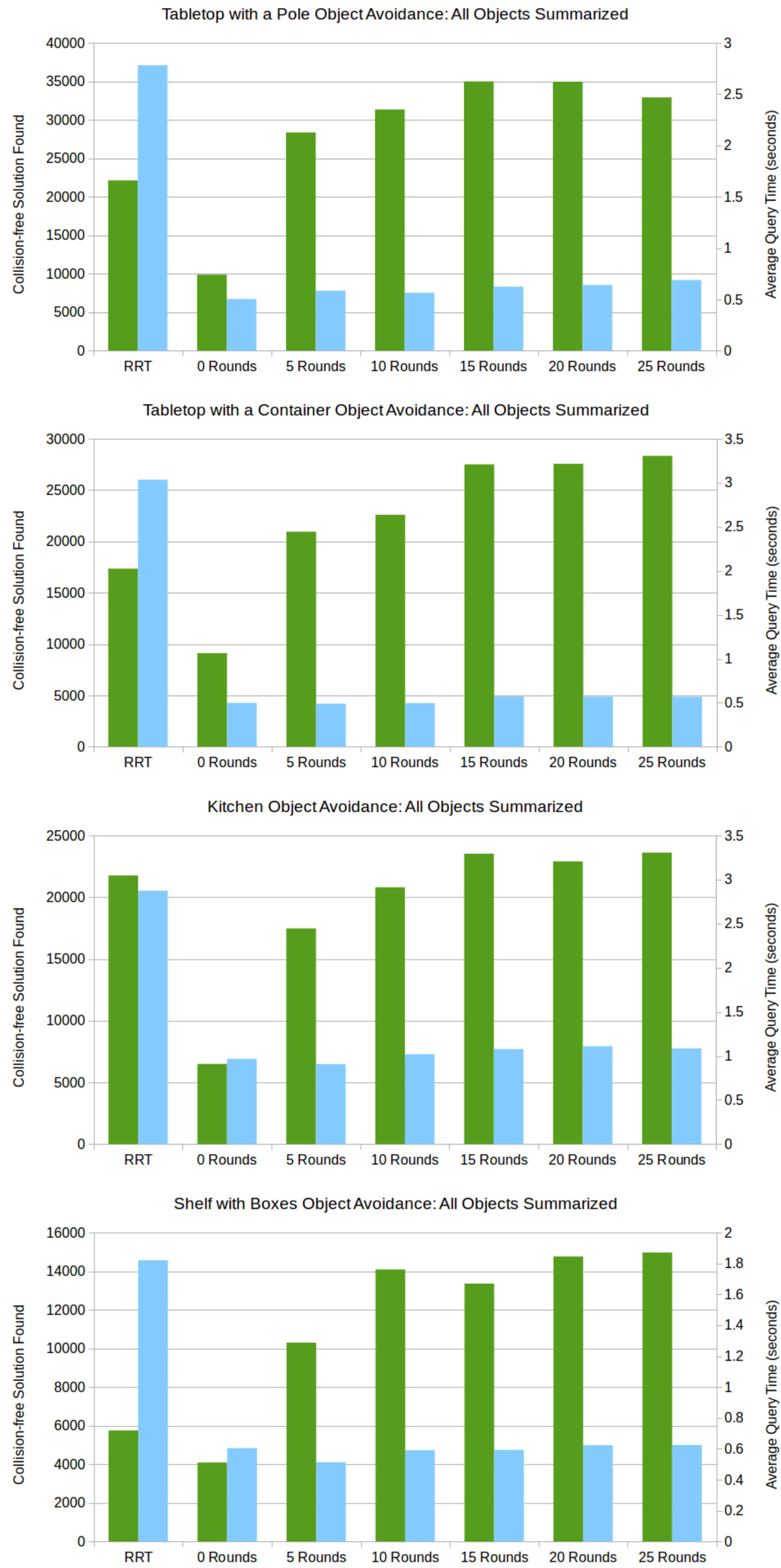


Figure 4-9: APSP Training obstacle avoidance graphs. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. See the Appendix for for a breakdown of the graphs by obstacle

4.3.2 Base Roadmap Results for A* Search

The online algorithms developed for avoiding dynamic obstacles are all based around A* search in varying forms [13]. In order to understand how A* search should be incorporated into the roadmap-based planner and solution cache, it is important to first understand how the search algorithm performs in the static environment with a roadmap absent of the solution cache. As with previous testing in static environments, an important distinction to make is whether the roadmap is assumed to be collision-free and therefore does not require collision checking for its edges. This is a particularly important distinction for searching the roadmap online because edges are expanded during the search that do not end up in the final solution, so checking those edges for collision would be time validating parts of the roadmap that are not relevant to the motion planning problem at hand. As a result, whether or not edges are checked for collision when expanded during A* search is the primary comparison these experiments examine.

Table 4.9 shows that checking the roadmap for collisions during A* search does not impact the rate of failure for producing a solution or the length of the solutions produced. This is expected because every collision check should validate the edge checked as collision-free. Average Runtime, however, is dramatically impacted by these collision checks. The takeaway from these results is that checking the roadmap for collisions edge by edge during online search is not a viable for avoiding dynamic obstacles for a reactive motion planner. As a result, all subsequent experiments involving online search will heavily rely on lazy collision checking.

Environment	Roadmap Description	Failure Rate	Average Runtime (s)	Average Path Length (rad)
Tabletop with a Pole	500 Node Roadmap No Edge Collision Checks	0.18%	0.7798	1.284
	500 Node Roadmap Checking Edges on Expansion	0.18%	11.48	1.284
	1000 Node Roadmap No Edge Collision Checks	0.18%	0.7656	1.238
	1000 Node Roadmap Checking Edges on Expansion	0.18%	12.20	1.238
Tabletop with a Container	500 Node Roadmap No Edge Collision Checks	1.40%	0.8052	1.310
	500 Node Roadmap Checking Edges on Expansion	1.40%	13.03	1.310
	1000 Node Roadmap No Edge Collision Checks	0.80%	0.8100	1.320
	1000 Node Roadmap Checking Edges on Expansion	0.80%	23.04	1.320
Kitchen	500 Node Roadmap No Edge Collision Checks	2.84%	1.059	1.289
	500 Node Roadmap Checking Edges on Expansion	2.84%	27.93	1.289
	1000 Node Roadmap No Edge Collision Checks	1.94%	1.041	1.285
	1000 Node Roadmap Checking Edges on Expansion	1.94%	37.24	1.285
Shelf with Boxes	500 Node Roadmap No Edge Collision Checks	15.56%	0.8464	1.308
	500 Node Roadmap Checking Edges on Expansion	15.56%	16.09	1.308
	1000 Node Roadmap No Edge Collision Checks	11.98%	0.8680	1.302
	1000 Node Roadmap Checking Edges on Expansion	11.98%	24.30	1.302

Table 4.9: A* search performance in a static environment examining the impact of checking roadmap edges for collisions when they are expanded during search.

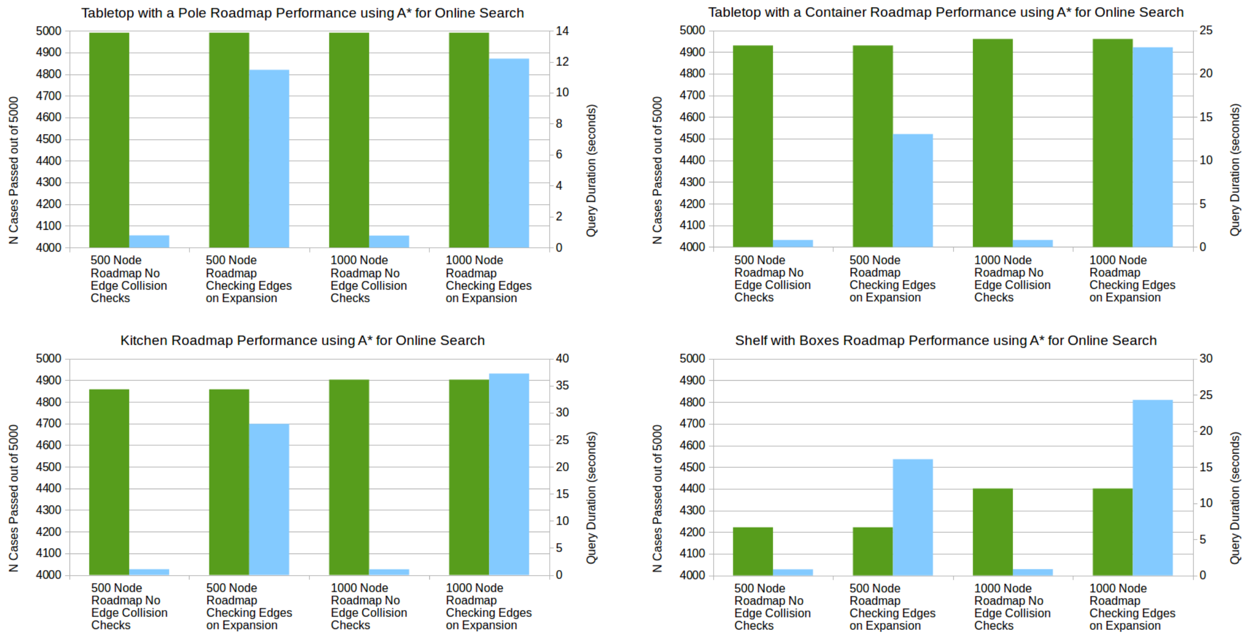


Figure 4-11: Performance comparison in a static environment examining the impact of checking the roadmap for collisions during A* search versus assuming all roadmap edges are collision-free. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue.

4.3.3 Training Comparison

The range of training rounds for APSP Training was selected based on a hypothesis that by 25 rounds, the solution cache would be saturated with paths that would avoid every object-pose tuple from the set of realistic obstacles. The resolution of 5 rounds was chosen to be as large as possible while still revealing trends in the range. This decision was made because APSP Training is computationally intensive through its use of parallel processing to generate the APSP solution set for each round, so training limited the ability for other experiments to be run.

Training a roadmap solution cache with 25 rounds of APSP training takes between two and three days to complete. As a comparison, the range of training rounds for A* Training was selected so the maximum number of rounds, 100000, also takes between two and three days to complete. Since A* Training does not utilize parallel processing, a finer resolution, 10000 rounds, could be selected relative to the range without inhibiting surrounding experiments. With this distinction, A* Training produces significantly fewer path solutions for a comparable training duration, so the solution caches for A* Training ware capped at a maximum of five solutions for a pair of roadmap nodes. This cap lends itself to a training metric, "Estimated Paths Removed", that provides insight as to when a solution cache may be saturated with path solutions to avoid every object-pose tuple. It does not appear that any roadmap solution caches are approaching saturation after 100000 rounds of A* Training.

Estimated Paths Removed is incremented when the pair of roadmap nodes for which a solution is returned have the maximum number of cached solutions and the returned solution has the minimum number of uses, indicating it is a newly found solution. Valid Rounds are training rounds where a solution exists in the roadmap for the given environment configuration, and Paths Added is the difference in the number of paths at the end and start of training.

Environment	Number of Training Rounds	Number of Paths Added
Tabletop with a Pole	0	0
	5	176540
	10	241756
	15	397286
	20	448964
	25	486812
Tabletop with a Container	0	0
	5	153506
	10	250194
	15	438846
	20	433287
	25	536634

Environment	Number of Training Rounds	Number of Paths Added
Kitchen	0	0
	5	219588
	10	314759
	15	482592
	20	489732
	25	625298
Shelf with Boxes	0	0
	5	144287
	10	407702
	15	335305
	20	445083
	25	521494

Table 4.10: Additional paths for solution caches developed with APSP Training

Env Name	Training Rounds	Valid Rounds	Paths Added	Paths Removed
Tabletop with a Pole	10000	3989	2284	42
	20000	8072	3990	116
	30000	12091	5216	203
	40000	16376	6484	310
	50000	20140	7310	424
	60000	24194	8388	491
	70000	28081	8878	621
	80000	32116	9468	723
	90000	36283	10230	835
	100000	40398	10914	927
Tabletop with a Container	10000	3357	1922	3
	20000	6745	3462	20
	30000	9993	4870	51
	40000	13510	6304	69
	50000	16797	7090	123
	60000	20091	8204	164
	70000	23275	8930	194
	80000	26715	9800	262
	90000	30156	10566	314
	100000	33277	11370	387

Env Name	Training Rounds	Valid Rounds	Paths Added	Paths Removed
Kitchen	10000	3494	1932	3
	20000	7011	3652	23
	30000	10388	5070	42
	40000	14019	6218	100
	50000	17522	7330	158
	60000	21083	8330	196
	70000	24358	9044	274
	80000	27544	9830	362
	90000	31342	10574	425
	100000	34650	11220	481
Shelf with Boxes	10000	2873	1222	0
	20000	5593	1986	2
	30000	8361	2826	13
	40000	11272	3616	21
	50000	14128	4296	28
	60000	17057	4872	52
	70000	19696	5466	71
	80000	22370	6018	110
	90000	25160	6516	115
	100000	27859	6712	125

Table 4.11: Overview of solution caches developed with A* Training. Paths Removed is an estimate determined by tracking whenever a new solution is created for a pair of nodes that already have the maximum number of allowed solutions.

4.3.4 A* Repair Results

The experiments conducted to evaluate the performance of the different combinations of algorithms and solution caches utilizing A* search are the same set of experiments used to evaluate the solution caches developed with APSP Training. To provide points of comparison, the first three entries in each graph in Figure 4-14 also exist in Figure 4-9. In general, the trade-off of concern is how often does an approach yield a collision-free solution versus how long on average does a query take with that approach. The first result of note is that all entries that use A* search in some form are able to find a collision-free solution in the roadmap whenever one exists. This makes Average Query Time the most important metric for these experiments.

The first two A* Repair approaches only use the shortest cached path to guide the search, and as a result, they both take longer on average than A* Repair 3 with the various solution caches. What is surprising about this, however, is that A* Repair 2 does not outperform A* Repair 1. As a reminder, after checking the shortest cached path for collisions, A* Repair 1 iteratively performs an end-to-end A* search on the roadmap while A* Repair 2 performs A* searches using pairs of nodes that bookend collisions in the original solution. The thought is that A* Repair 2 would have to search less of the roadmap to produce a full collision-free solution and therefore would finish faster. Since that is not the case, it is possible that A* Repair 2 yields solutions that are significantly longer than the shortest possible collision-free solution due to backtracking of some form. It is also possible that searching in the proximity of obstacles, as A* Repair 2 does, leads to more collision checks performed on edges that do not end up in the final solution because it checks nearby edges that collide with the same obstacle. However, the most likely cause is that the small size of the roadmaps result in paths that are too short for there to be any benefit in breaking up the path into collision-free segments and searching to connect the segments.

The most significant result from these experiments comes from the unification of trained solution caches and online search provided by A* Repair 3. By checking a cache of useful path solutions and then performing online search only when necessary,

the RoadmapManager is able to return a collision-free solution whenever one exists in the roadmap while handling queries in almost the same time as just using the solution cache. The explanation for that result is two-fold. Comparing A* Repair 3 with 15 rounds of APSP Training to only the APSP Training, one can see that the majority of cases where a collision-free solution is found that solution exists in the solution cache. This means that online search is not used for those cases and therefore does not slow down the query at all. The cases where online search is used will have slower query times, but collision checking for the A* Repair has likely already been performed in large part from checking the cached solutions.

Finally, Figure 4-14 provides a comparison of using solution caches developed with APSP Training and A* Training for guiding subsequent A* searches. It is unclear from the data collected how substantial the diminishing returns are for A* Training at 100000 rounds, but at least for the Shelf with Boxes environment, there is not much improvement in average query time from 50000 rounds. Additionally, 15 rounds of APSP Training reduces average query times more than 100000 rounds of A* Training in all four environments. However, this should not be surprising considering more than 40 times as many paths were added to the solution cache through 15 rounds of APSP Training. What is perhaps more surprising is how close their respective performance is considering the discrepancy in the number of paths added, but the results should be interpreted with the understanding that A* Training more directly trains solution caches for the experiments conducted.

Environment	Roadmap and Algorithm Description	Original Cached Solution Collides	Collision-free Solution Found	Average Runtime (s)
Tabletop with a Pole	Control Roadmap	84086	9881	0.5041
	A* Repair Approach 1	84086	37773	0.8118
	A* Repair Approach 2	84086	37773	0.7929
	A* Repair Approach 3 - 5 Rounds of APSP Training	84086	37770	0.6203
	A* Repair Approach 3 - 15 Rounds of APSP Training	84086	37773	0.5612
	A* Repair Approach 3 - 10000 Rounds of A* Repair Training	84086	37762	0.6891
	A* Repair Approach 3 - 50000 Rounds of A* Repair Training	84086	37760	0.6155
	A* Repair Approach 3 - 100000 Rounds of A* Repair Training	84086	37759	0.5895
Tabletop with a Container	Control Roadmap	85860	9118	0.4982
	A* Repair Approach 1	85860	30610	0.6903
	A* Repair Approach 2	85860	30610	0.7562
	A* Repair Approach 3 - 5 Rounds of APSP Training	85860	30609	0.6196
	A* Repair Approach 3 - 15 Rounds of APSP Training	85860	30609	0.5318
	A* Repair Approach 3 - 10000 Rounds of A* Repair Training	85860	30610	0.6652
	A* Repair Approach 3 - 50000 Rounds of A* Repair Training	85860	30608	0.5971
	A* Repair Approach 3 - 100000 Rounds of A* Repair Training	85860	30608	0.5576

Environment	Roadmap and Algorithm Description	Original Cached Solution Collides	Collision-free Solution Found	Average Runtime (s)
Kitchen	Control Roadmap	78848	6499	0.9681
	A* Repair Approach 1	78848	26752	1.377
	A* Repair Approach 2	78848	26752	1.341
	A* Repair Approach 3 - 5 Rounds of APSP Training	78848	26751	1.143
	A* Repair Approach 3 - 15 Rounds of APSP Training	78848	26752	1.038
	A* Repair Approach 3 - 10000 Rounds of A* Repair Training	78848	26752	1.223
	A* Repair Approach 3 - 50000 Rounds of A* Repair Training	78848	26752	1.106
	A* Repair Approach 3 - 100000 Rounds of A* Repair Training	78841	26744	1.012
Shelf with Boxes	Control Roadmap	64832	4096	0.6046
	A* Repair Approach 1	64832	16015	0.7939
	A* Repair Approach 2	64832	16011	0.8665
	A* Repair Approach 3 - 5 Rounds of APSP Training	64832	16011	0.7166
	A* Repair Approach 3 - 15 Rounds of APSP Training	64832	16012	0.6185
	A* Repair Approach 3 - 10000 Rounds of A* Repair Training	64832	16011	0.7513
	A* Repair Approach 3 - 50000 Rounds of A* Repair Training	64832	16010	0.6775
	A* Repair Approach 3 - 100000 Rounds of A* Repair Training	64832	16008	0.6457

Table 4.12: A* Repair obstacle avoidance data for 500 node roadmaps. The Control Roadmap results are identical to the APSP Training Control Roadmap results.

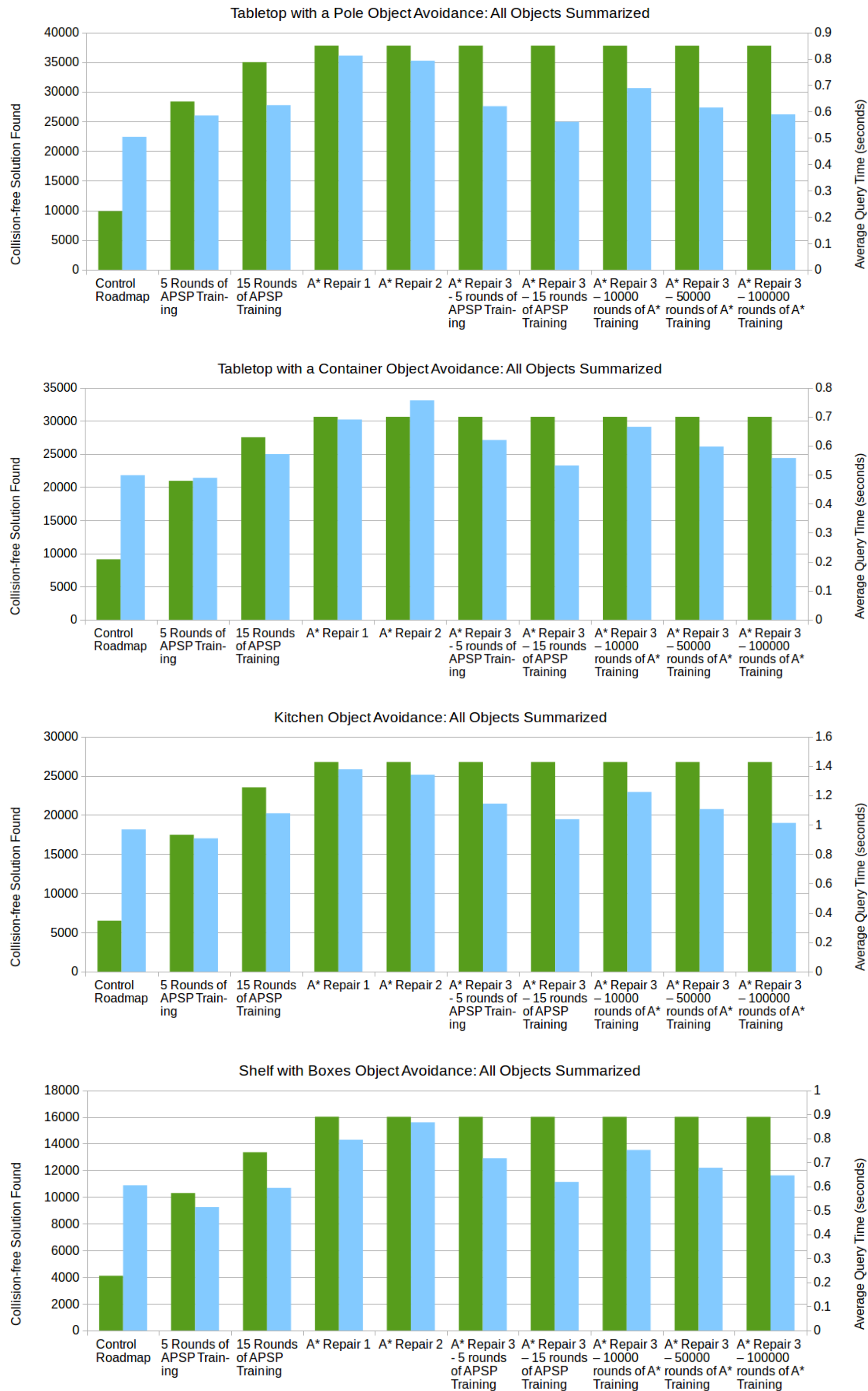


Figure 4-14: A* Repair obstacle avoidance graphs. For each roadmap, the number of cases that have a valid roadmap seed is shown in green and the average duration for a roadmap query is shown in blue. See the Appendix for for a breakdown of the graphs by obstacle.

4.4 Performance Comparison for Different Incremental Execution Implementations

There are two goals for the analysis of the incremental search algorithms implemented for the RoadmapManager. The first is to compare the performance of Adapted D* Lite and A* Repair 1. The second goal is to examine how each algorithm implementation benefits from interleaving replanning and trajectory execution. As a reminder this occurs when the previously found plan becomes invalid. At that point, if the next segment of the trajectory from the previous plan is still valid, the segment is sent to the robot controller for execution while replanning occurs. Effective Planning time is the metric created to capture the benefit gained by interleaving replanning and trajectory execution. It is defined as the difference between the Replanning and Execution Time metric and the Execution Time Only metric.

All tested algorithm implementations incorporate trajectory optimization, incremental trajectory execution, and execution monitoring. The discrepancies in Solution Found between the different algorithm implementations likely have a few causes. The most apparent is indicated by the fact that, in general, solutions are found less often by the overlapping implementations. The suspected cause is that executing the additional trajectory segment while replanning can cause the robot to move too close to an obstacle to then connect to an existing roadmap node. This is supported by the fact that TrajOpt covers the entire reachable workspace and therefore can move the robot into positions that are not accessible by a given roadmap.

Figure 4-15 contains graphs highlighting the important metrics from Table 4.13, namely Executed Path Length, Replanning and Execution Time, and Effective Planning Time. For all environments and roadmap sizes, the A* Repair implementation is substantially faster than the Adapted D* Lite. While the paths produced by A* Repair are on average shorter leading to shorter execution times, the difference is more prominent in Effective Planning Time. This difference can likely be traced to how both D* Lite and A* have been adapted to address the collision checking bottleneck presented by the surrounding software system. A* Repair only checks roadmap edges

for collisions after they are returned as part of a full roadmap path, whereas Adapted D* Lite identifies the roadmap edge in collision from the previous plan and then checks all neighboring edges for collisions before replanning. Additionally, the small size of the tested roadmaps likely results in a very short A* search, so the possible benefit to be had from performing that search during execution is fairly minimal. For both algorithms and roadmap sizes, Effective Planning Time is reduced by interleaving replanning with execution, but the difference is much larger for Adapted D* Lite. This result indicates that interleaving replanning and execution is a viable strategy for a reactive motion planner. This is especially the case for the scenarios outlined in the introduction because motion delay caused by replanning would be more heavily penalized than sub-optimality in the executed trajectory.

Environment	Roadmap and Algorithm Description	Solution Found	Executed Path Length (rad)	Replanning and Execution Time (s)	Execution Time Only (s)	Effective Planning Time (s)
Tabletop with a Pole	500 Node Roadmap with Overlapping D* Lite	3497	6.827	10.164	5.396	4.768
	500 Node Roadmap with Serial D* Lite	3595	6.448	10.871	5.137	5.734
	500 Node Roadmap with Overlapping A* Repair	3390	6.447	7.346	5.064	2.282
	500 Node Roadmap with Serial A* Repair	3635	6.061	7.161	4.781	2.380
	1000 Node Roadmap with Overlapping D* Lite	3598	6.379	9.883	5.061	4.822
	1000 Node Roadmap with Serial D* Lite	3666	6.014	10.872	4.824	6.048
	1000 Node Roadmap with Overlapping A* Repair	3687	6.065	7.165	4.794	2.371
	1000 Node Roadmap with Serial A* Repair	3762	5.665	6.939	4.493	2.446
Tabletop with a Container	500 Node Roadmap with Overlapping D* Lite	4068	6.963	10.138	5.439	4.699
	500 Node Roadmap with Serial D* Lite	4111	6.844	10.870	5.383	5.487
	500 Node Roadmap with Overlapping A* Repair	4131	6.619	7.431	5.148	2.283
	500 Node Roadmap with Serial A* Repair	4179	6.351	7.386	4.953	2.433
	1000 Node Roadmap with Overlapping D* Lite	3965	6.692	9.669	5.293	4.376
	1000 Node Roadmap with Serial D* Lite	4014	6.576	11.110	5.242	5.868
	1000 Node Roadmap with Overlapping A* Repair	4072	6.457	7.435	5.075	2.36
	1000 Node Roadmap with Serial A* Repair	4123	6.193	7.463	4.884	2.579

Environment	Roadmap and Algorithm Description	Solution Found	Executed Path Length (rad)	Replanning and Execution Time (s)	Execution Time Only (s)	Effective Planning Time (s)
Kitchen	500 Node Roadmap with Overlapping D* Lite	3945	6.407	12.217	5.012	7.205
	500 Node Roadmap with Serial D* Lite	4052	6.225	13.113	4.907	8.206
	500 Node Roadmap with Overlapping A* Repair	4096	6.249	7.919	4.871	3.048
	500 Node Roadmap with Serial A* Repair	4176	5.970	7.845	4.682	3.163
	1000 Node Roadmap with Overlapping D* Lite	3843	6.254	11.969	4.918	7.051
	1000 Node Roadmap with Serial D* Lite	3921	5.974	13.014	4.763	8.251
	1000 Node Roadmap with Overlapping A* Repair	4081	6.099	8.001	4.780	3.221
	1000 Node Roadmap with Serial A* Repair	4150	5.829	8.027	4.596	3.431
Shelf with Boxes	500 Node Roadmap with Overlapping D* Lite	3534	7.002	11.366	5.575	5.791
	500 Node Roadmap with Serial D* Lite	3547	6.923	12.533	5.556	6.977
	500 Node Roadmap with Overlapping A* Repair	3662	6.769	7.920	5.361	2.559
	500 Node Roadmap with Serial A* Repair	3694	6.582	8.037	5.229	2.808
	1000 Node Roadmap with Overlapping D* Lite	3566	6.747	10.877	5.398	5.479
	1000 Node Roadmap with Serial D* Lite	3513	6.624	12.180	5.339	6.841
	1000 Node Roadmap with Overlapping A* Repair	3778	6.514	7.891	5.177	2.714
	1000 Node Roadmap with Serial A* Repair	3808	6.357	8.059	5.066	2.993

Table 4.13: Performance comparison of two incremental algorithms with both overlapping and non-overlapping (serial) replanning and execution. Effective Planning Time is the difference between the Replanning and Execution Time metric and the Execution Time Only metric.

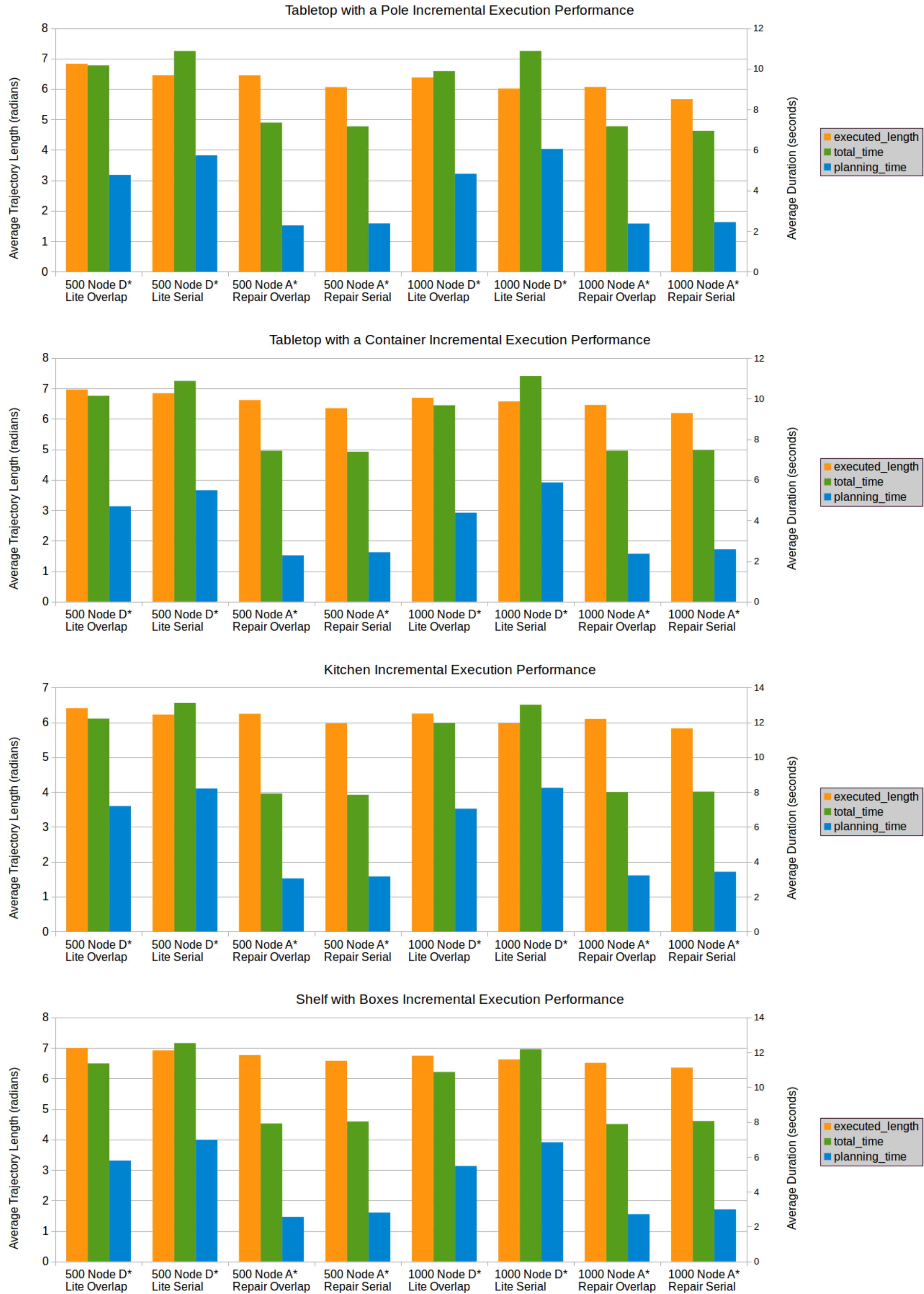


Figure 4-15: Performance comparison of two incremental algorithms with both overlapping and non-overlapping (serial) replanning and execution. One based around D* Lite and the other based around A* Repair. Both heavily rely on lazy collision checking and have the roadmap solutions provided to TrajOpt before execution.

Chapter 5

Discussion

5.1 Looking Forward

Roadmap-based motion planning and fast online search are both active areas of research in the robotics community, so it should come as no surprise that there are open questions remaining at the close of this research. The first question that must be looked at is what are the limitations of this roadmap-based motion planner as it stands in its current configuration. Over the course of the research presented here, but particularly during the development of online search algorithms, collision checking proved to be a bottleneck that had to be continually tip-toed around with lazy collision checking. Collision checking is both the reason that looking up cached solutions in a static environment is not near-instant, and it is likely the reason that the adaptation of the more recent D* Lite incremental search algorithm is slower during replanning than the more established A* heuristic search algorithm. Our group currently has plans to address these limitations with a GPU-based collision checking approach.

The question that follows is if this roadmap-based planner were equipped with faster collision checking, how should the algorithmic approaches associated with the planner be modified. The adaptations made to D* Lite for this planner were focused around limiting collision checking with a lazy approach and incorporating trajectory optimization. With faster collision checking, the first of these adaptations can be

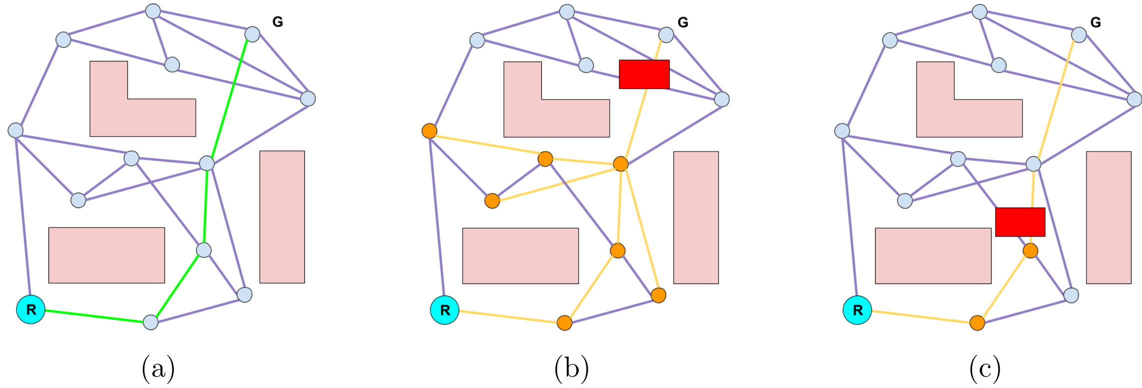


Figure 5-2: Scenario illustrating the shortcomings of D* Lite when an obstacle is detected near the goal. A collision-free path is shown for the static environment in (a). In (b), an obstacle is detected near the goal and updates are required for seven nodes in addition to the current state before computing a new shortest path. For contrast, the obstacle in (c) is detected closer to the current state updates are only required for two nodes in addition to the current state. Nodes that must be updated are shown in orange and the edges are in yellow that provide the shortest path to the goal for any one of these nodes.

replaced with the approach used by the standard implementation of D* Lite. However, D* Lite was selected for these adaptations because its structure lends itself to limited collision checking more than other state of the art search algorithms. In practice, D* Lite does not perform well when changes to the search graph are close to the current location of the robot, as is often the case with robot arms. Motion planning in the context of robot arms also tends towards scenarios where the full state of the environment can be observed, so changes to the roadmap can occur anywhere with respect to the current state of the robot. D* Lite does not perform well when changes occur near the goal configuration for a planning problem because these changes often invalidate work done in previous search iterations. This leaves the question of whether or not D* Lite is the right search algorithm for our problem scenarios.

Since the invention of D* Lite over 15 years ago, other researchers have investigated alternative search strategies that have been demonstrated to improve upon the results produced by D* Lite. Adaptive A*(AA*) was developed around 2006 [21]. AA* runs A* and then starts executing the returned path. If the cost of the path increases during execution, it updates all observable edge costs, and again performs A* search.

When constructing a path after finishing A* search, it maintains a dicitionary is maintained mapping a node in the path to the next node in the path. This is the revserse of the dictionary developed during A* search that maps each node to the node it was expanded from during search. When increased edge costs are observed, AA* removes the nodes from the next node dictioanry. After each iteration of a full A* search, heuristic values are updated for all expanded nodes based off of the path it found. Generalized Adaptive A* (GAA*) was developed around 2008 and builds off AA* by adding extra methods to reestablish consistency in the heuristic values for cases when edge costs decrease as well [21].

Multipath Adaptive A* (MPAA*) was developed around 2014 and is very similar to the implementation of AA* [22]. The only difference is in the goal condition. Instead of just returning success if the state to be expanded is the goal state, MPAA* will also return success if the state can be traced to the goal state through the dictionary mapping a node to the next node in a previously found path. This allows the search to terminate when it reaches a state that the goal state is known to be reachable from. Multipath Generalized Adaptive A* (MPGAA*) was developed around 2015 and makes the same ajustment to MPAA* that GAA* makes to AA* [23]. This adaptation requires slightly more from an implementation standpoint than the adaptation for GAA* due to the modified goal condition.

These four algorithms require updating all states that have been affected by changed edge costs within a visbility range along with establishing consistency in the heuristic values that have been affected. This can require a substantial update for a given environment change, which is the reason these algorithms were not implemented for the research presented here. In fact, the the authors of MPAA* and MPGAA* address how MPGAA* performance suffers when presented with extned visibility ranges due to the number of heuristic updates required. That being said, developments have been made as recently as 2017 to address this limitaion [24].

The implementation of A* Repair 1 has similarities to AA* in terms of handling A* in an iterative nature. The main difference is AA* helps successive searches by updating the heuristics for all nodes within a visibility range while A* Repair 1

only updates nodes that are perceived to be in a shortest path via a lazy collision checking approach. As such, should not be difficult to modify A* Repair 1 to more closely align with AA*. From there, the implementation could be further adapted to resemble MPGAA*.

However, the problem scenarios addressed by this research use a full visibility range, which could slow down an implementation of MPGAA* or its improved variants. The MPGAA* authors talk about this slowing down the algorithm because of the time it takes to reestablish consistency in the heuristics, but for our system in its current state, it would also require a lot of collision checking which would slow down the implementation. The improved MPGAA* addresses the slowdown from reestablishing consistency, and as mentioned earlier, a GPU-based collision checking approach could address that system bottleneck. In summary, there is likely novel research to be explored by combining improved MPGAA* with fast collision checking of the full environment, but implementing any of the above algorithms without fast collision-checking would yield results that are no better than what is currently obtained with A* Repair 1.

Finally, our research group hopes to adapt the roadmap-based planner developed here to be used for the Toyota Human Support Robot (HSR). The HSR presents a unique challenge in that it has a robot arm on a moving base. This means environment configurations are expected to change dramatically with respect to the base, and therefore larger portions of the reachable workspace for the HSR can be expected to move in and out of collision over the course of accomplishing a task-level plan. It is possible that this can be addressed by constructing a roadmap for an empty environment and then updating the roadmap with the lazy approach established in A* Repair 1. Implementing a GPU-based collision checking approach could make these lazy updates fast enough to satsify the requirements of the problem scenario. If that is not the case, another possible solution is the implementation of a Probabilisitc Roadmap for Changing Environments (PRMCE) developed around 2002 [10]. This algorithm requires a workspace cell decomposition in order to perform all collision checking in workspace. The innovation of PRMCE is an efficient mapping of the

workspace collision information back to the configuration-space roadmap. However, this approach would still be dependent on fast collision checking, so the GPU-based approach would likely be required regardless.

5.2 Revisiting the Problem Statement

The goal for this research was to develop a system that can plan and execution motions for high-DOF robot arms in a reactive and intuitive manner. This motion planner must coordinate with a task-level executive to accomplish tasks surrounding motion planning problems in a changing environment. The motion planner must be reactive so executed plans do not collide with a human or other moving obstacles, but also so that the motion plans are still relevant to larger task-level plan when the motions are executed. Additionally, the motion planner must be intuitive, which we define as near-optimal, so humans or other agents can determine what the robot intends to accomplish while it is executing a motion.

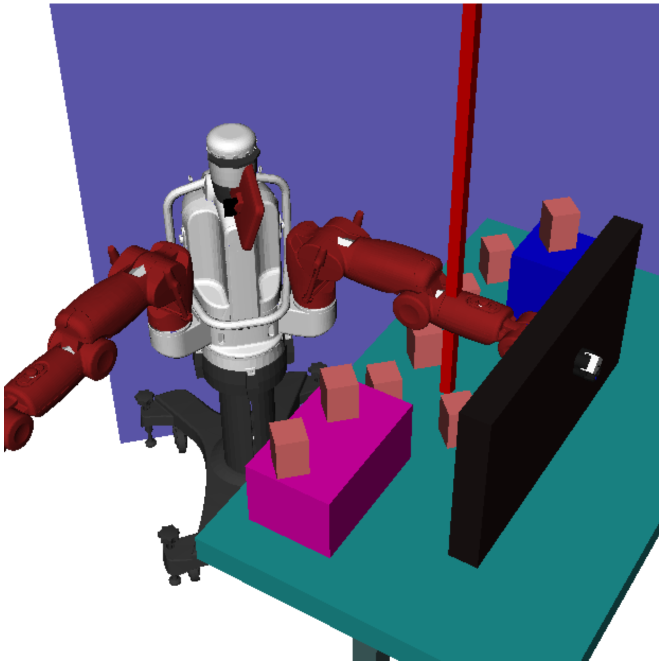
The sparse roadmap-based planner developed for this research demonstrated its capability to rapidly produce collision-free seed trajectories in static environments. The roadmap-based planner is able to achieve short planning times in large part due to a precomputed cache of shortest path solutions. Roadmap seed trajectories are provided to TrajOpt for trajectory optimization that leaves the resulting trajectory near-optimal. To address changing environments, offline approaches were developed to provide useful paths to the solution cache in addition to the all-pairs shortest path solution set for the static environment. For environment configurations where no collision-free solution exists in the cache, online search algorithms were developed that build off the information stored in the cache to minimize repeating work during online motion planning that has been performed offline. Finally, incremental search algorithms were developed to provide fast replanning when a motion plan becomes invalid during execution.

The combination of these developments for the roadmap-based planner with trajectory optimization result in a motion planner that rapidly produces near-optimal,

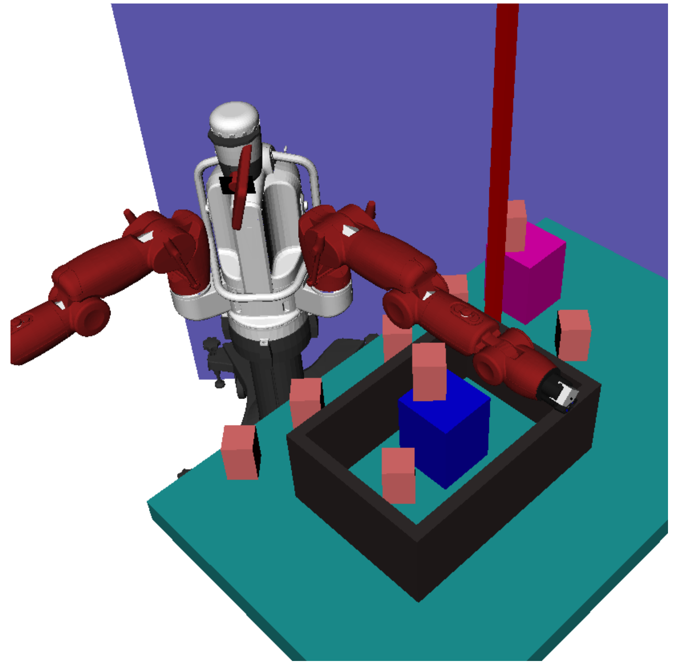
collision-free trajectories for high-DOF robot arms in a significant majority of typical planning problems. By rapidly producing collision-free trajectories in changing environments in coordination with a task-level executive, the motion planner demonstrates itself to be reactive. By producing trajectories that are near-optimal with respect to any objective function, the motion planner demonstrates itself to be intuitive. By satisfying the requirements of being reactive and intuitive, this roadmap-based motion planner demonstrates itself to be capable of accomplishing motion-based tasks in the context of human-robot collaboration.

Appendix A

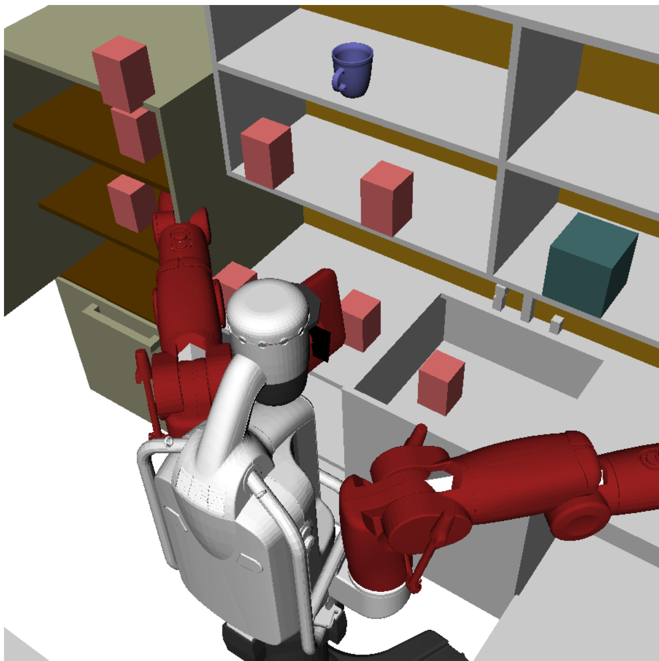
Additional Figures, Tables, and Graphs



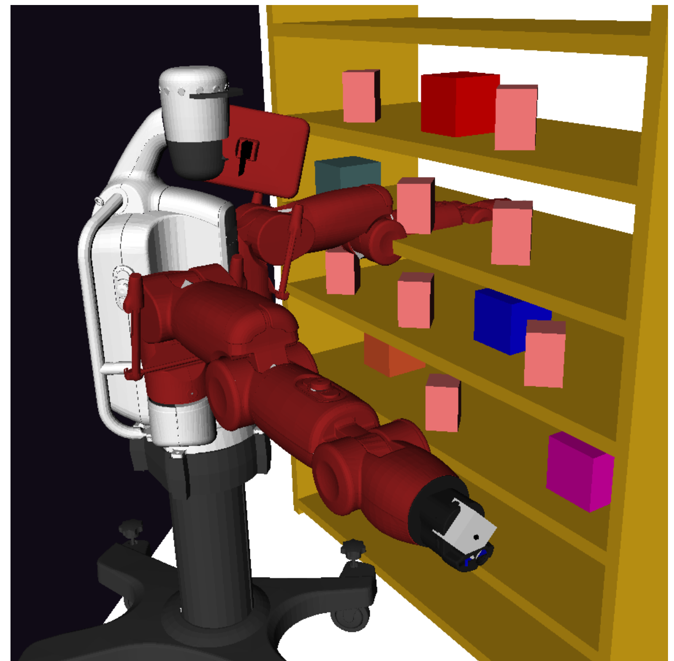
(a) Tabletop with a Pole



(b) Tabletop with a Container

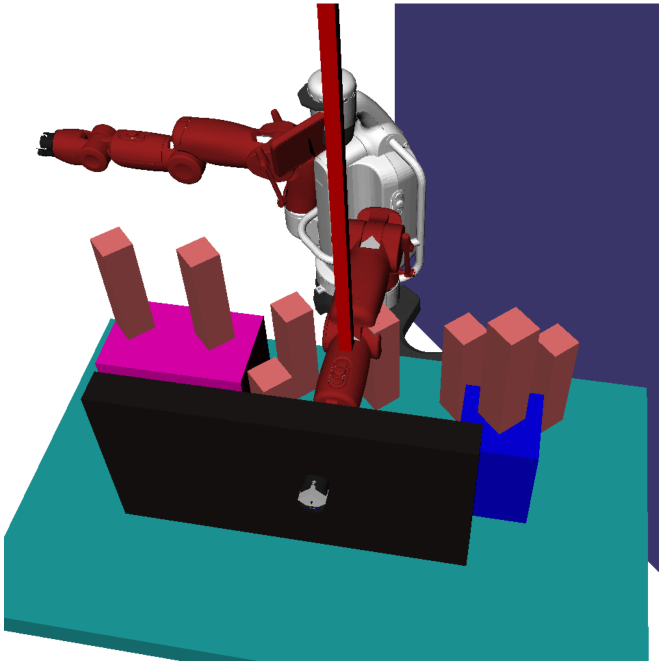


(c) Kitchen

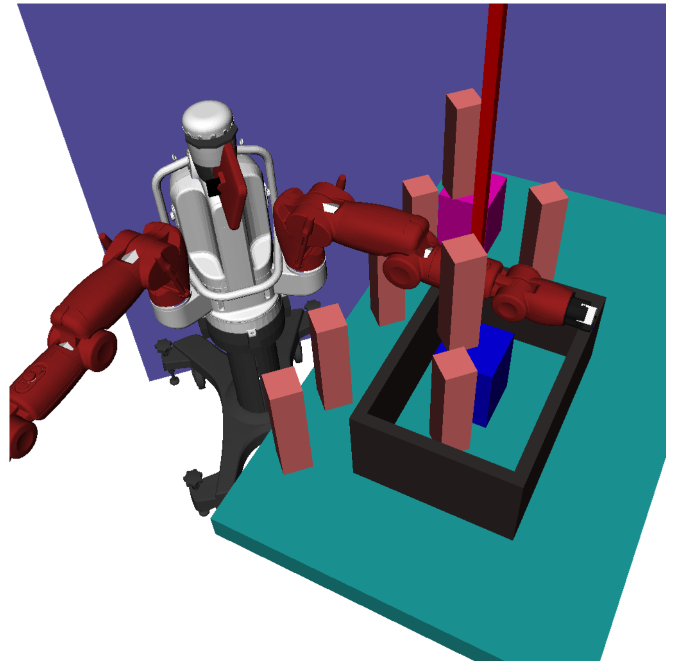


(d) Shelf with Boxes

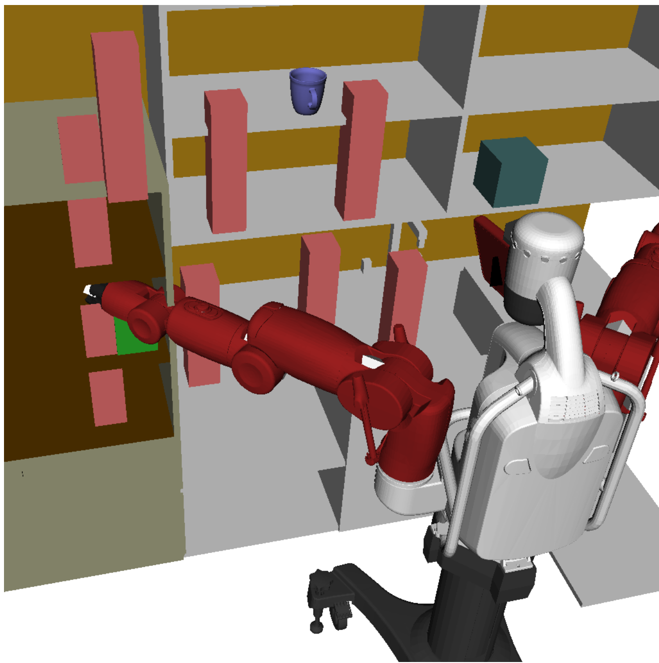
Figure A-2: Cup obstacle in its eight poses for each of the four environments



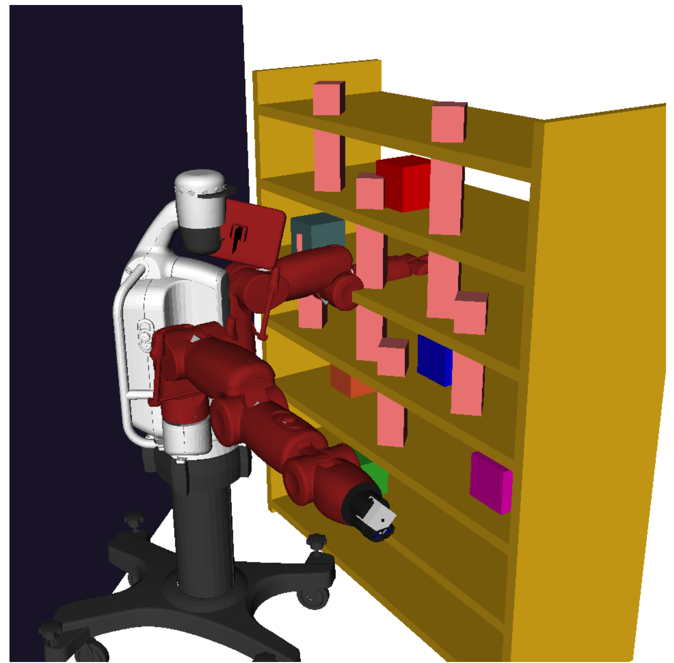
(a) Tabletop with a Pole



(b) Tabletop with a Container

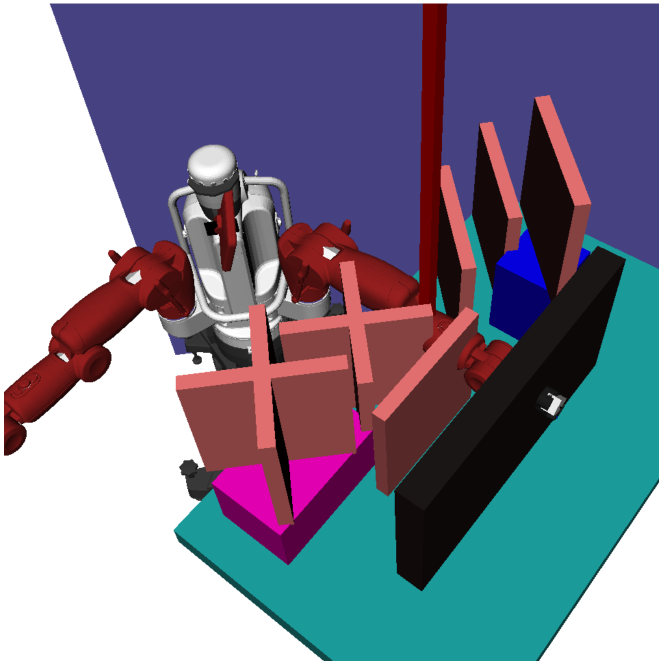


(c) Kitchen

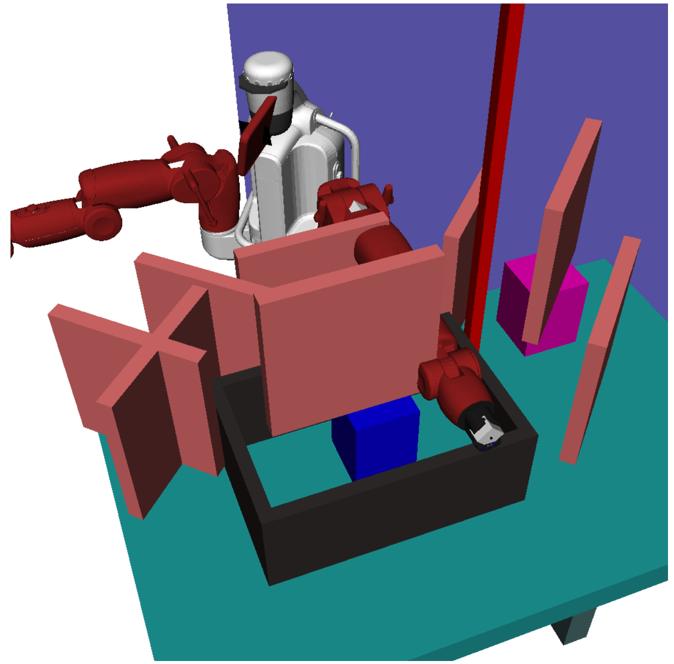


(d) Shelf with Boxes

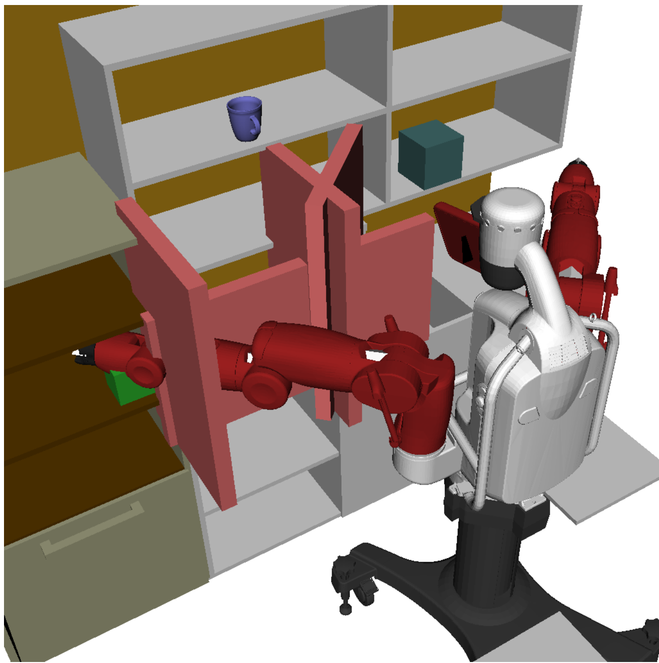
Figure A-4: Thermos obstacle in its eight poses for each of the four environments



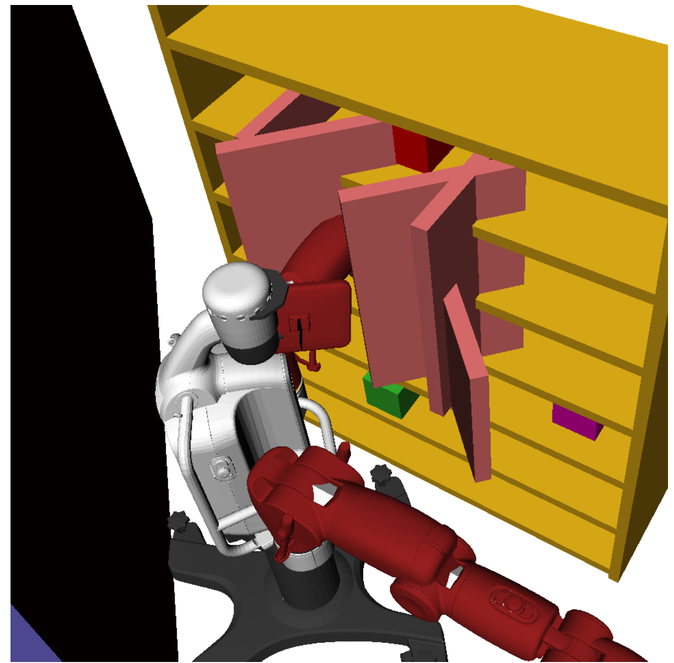
(a) Tabletop with a Pole



(b) Tabletop with a Container

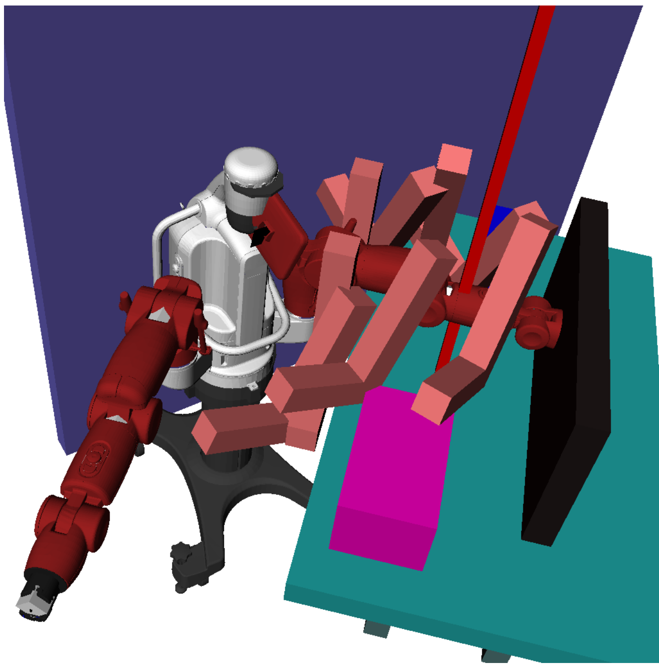


(c) Kitchen

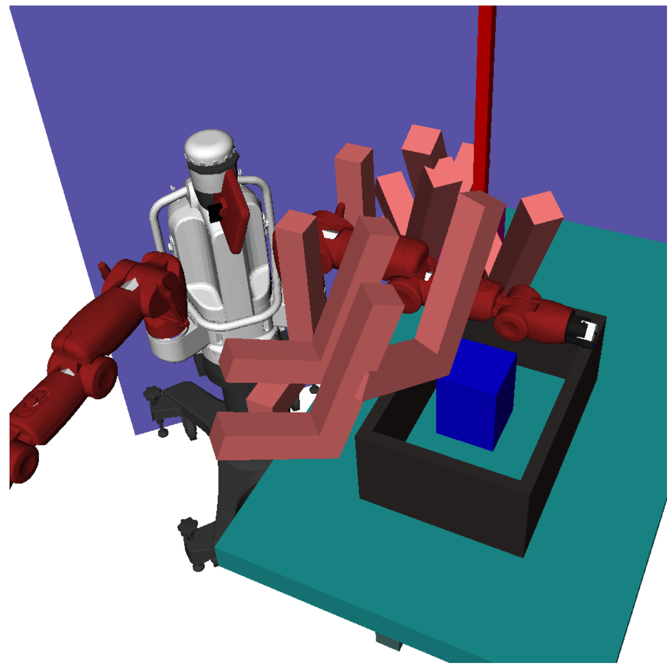


(d) Shelf with Boxes

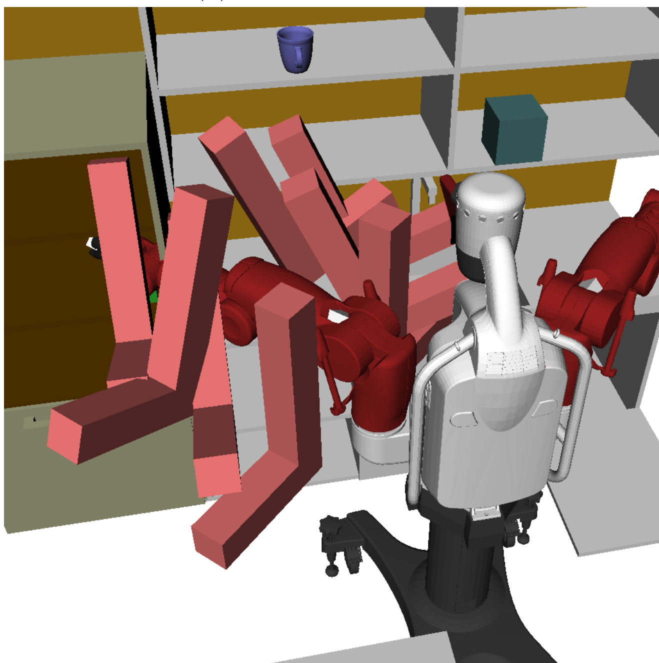
Figure A-6: Monitor obstacle in its eight poses for each of the four environments



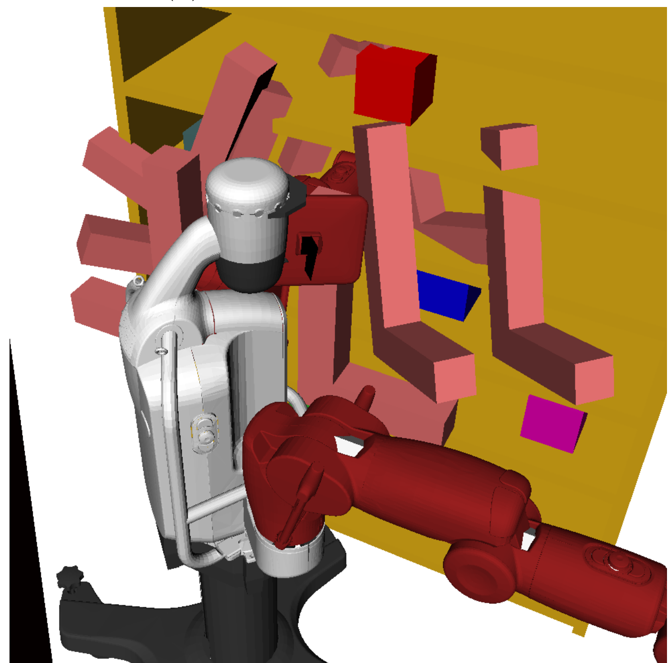
(a) Tabletop with a Pole



(b) Tabletop with a Container

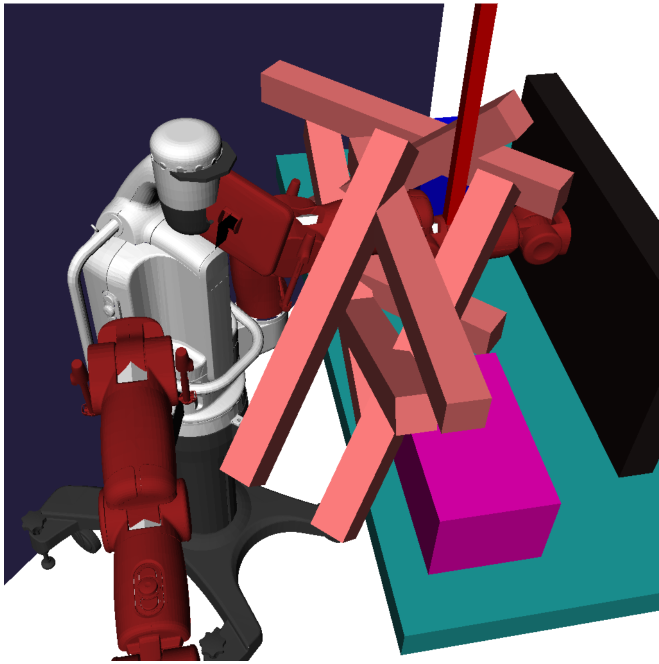


(c) Kitchen

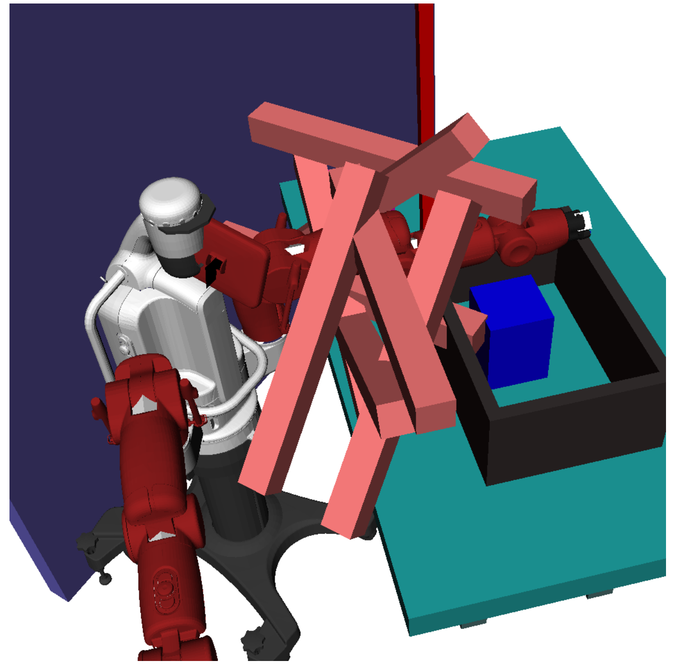


(d) Shelf with Boxes

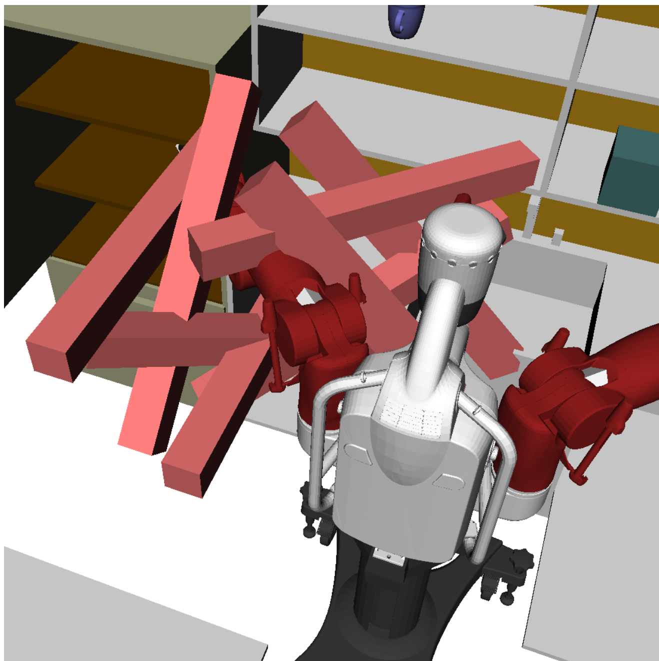
Figure A-8: Bent Arm obstacle in its eight poses for each of the four environments



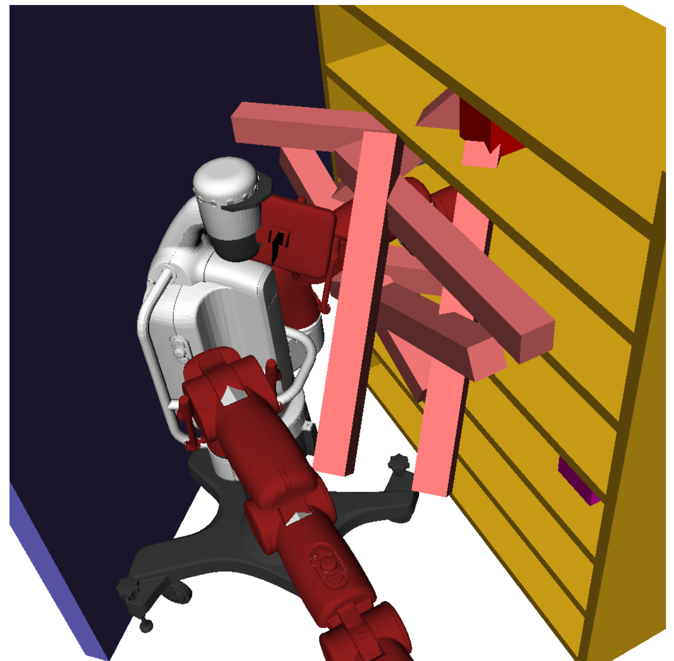
(a) Tabletop with a Pole



(b) Tabletop with a Container



(c) Kitchen



(d) Shelf with Boxes

Figure A-10: Straight Arm obstacle in its eight poses for each of the four environments

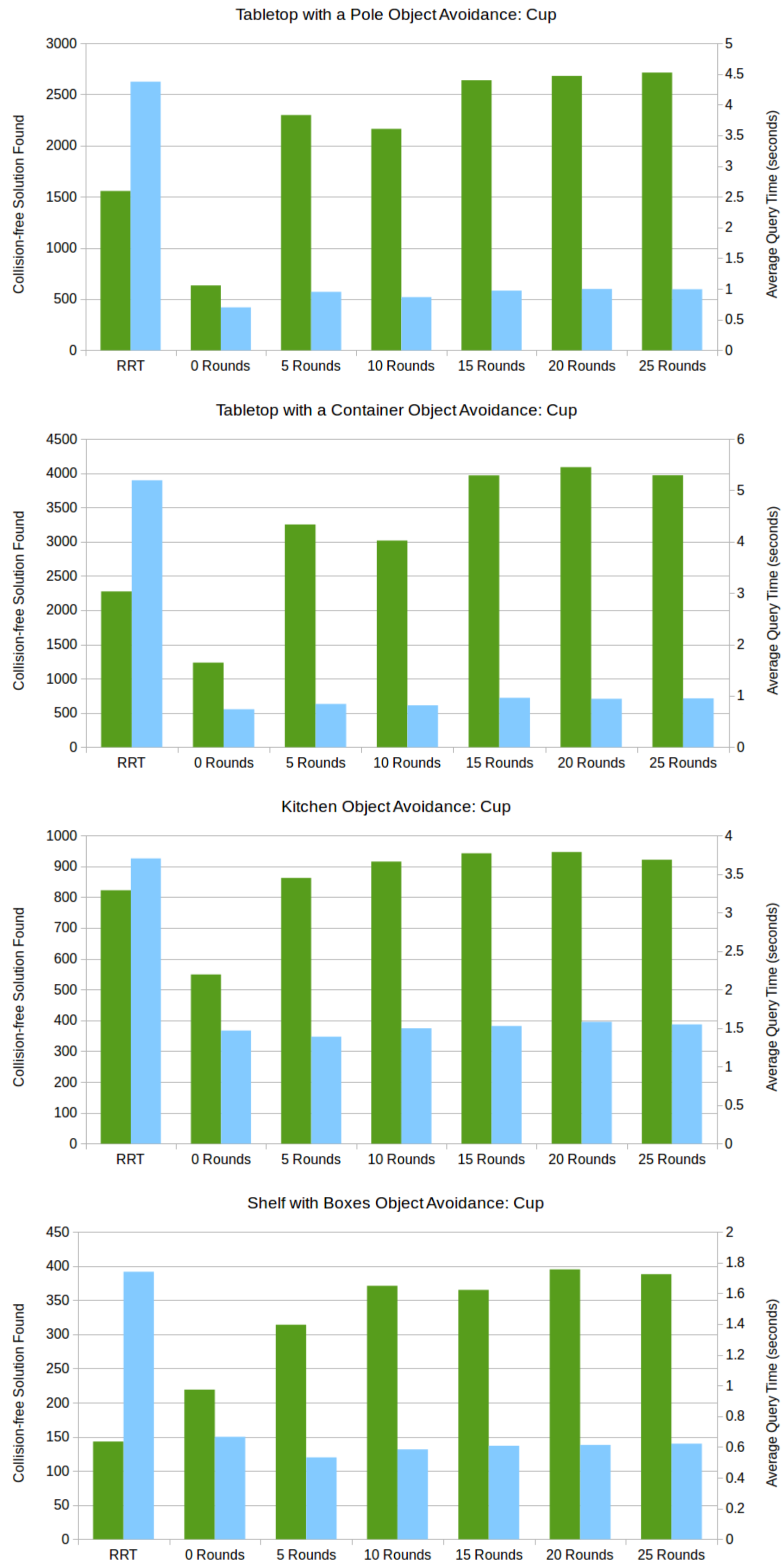


Figure A-11: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Cup obstacle for all four environments

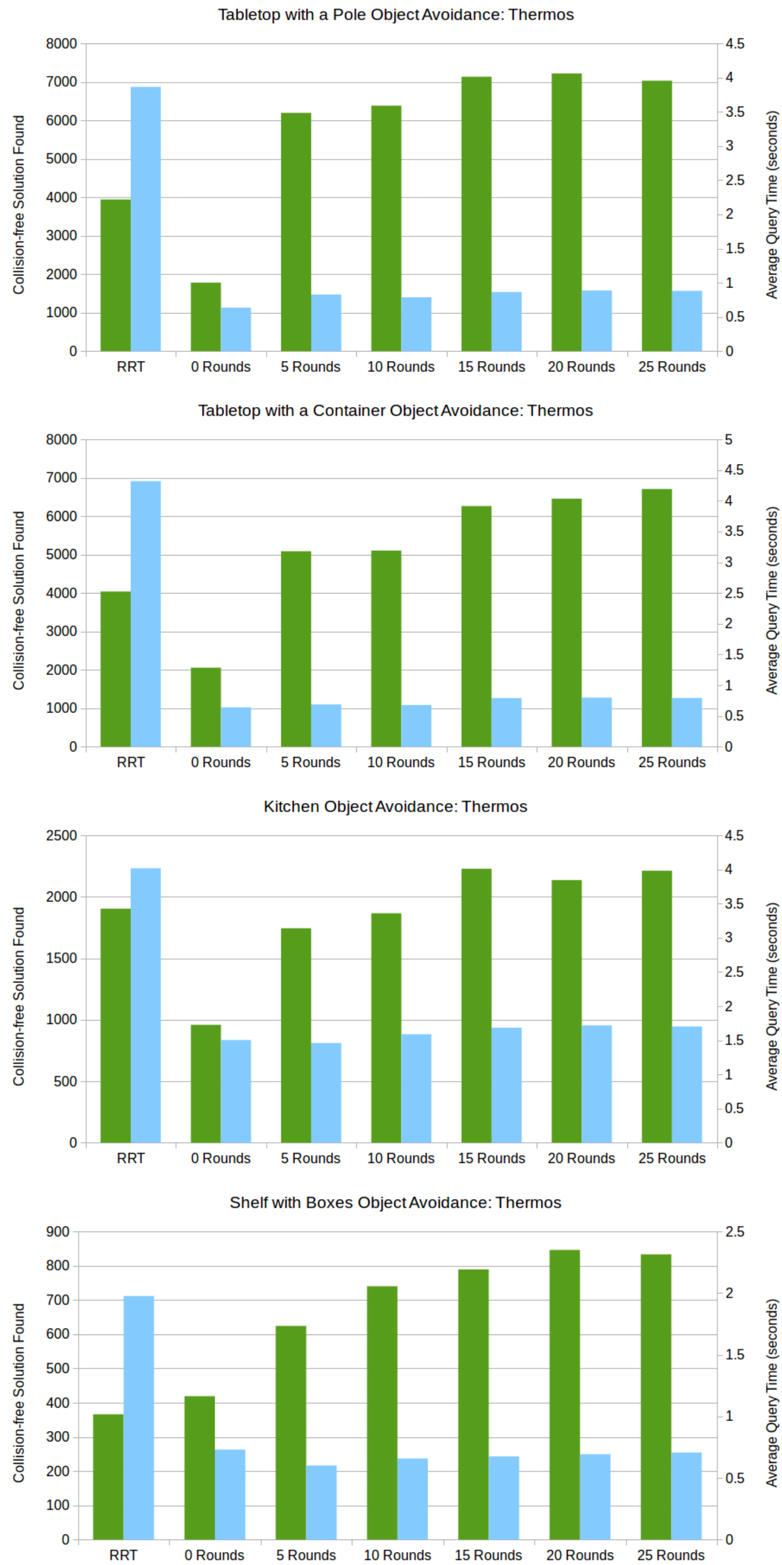


Figure A-12: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Thermos obstacle for all four environments

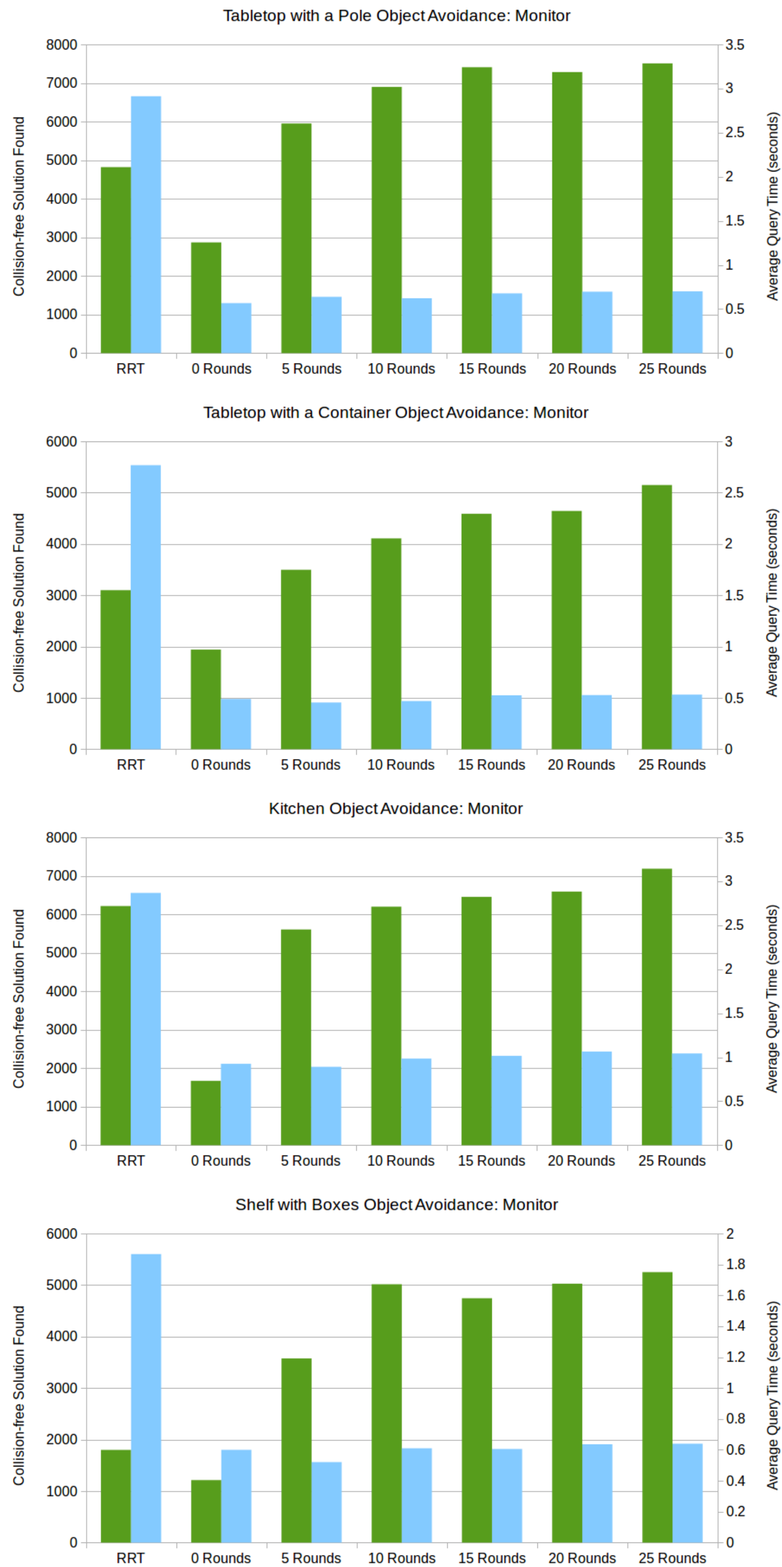


Figure A-13: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Monitor obstacle for all four environments

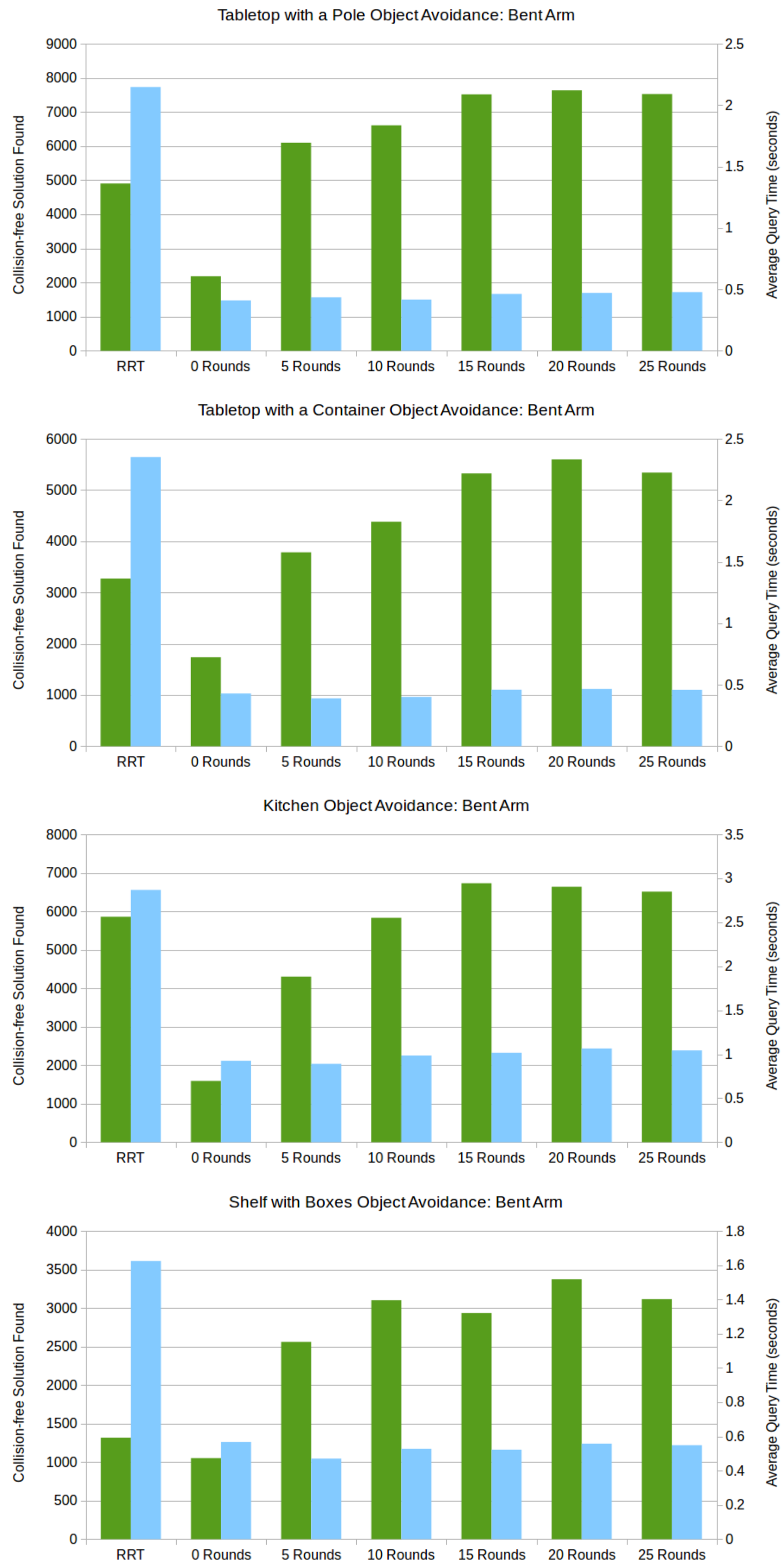


Figure A-14: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Bent Arm obstacle for all four environments

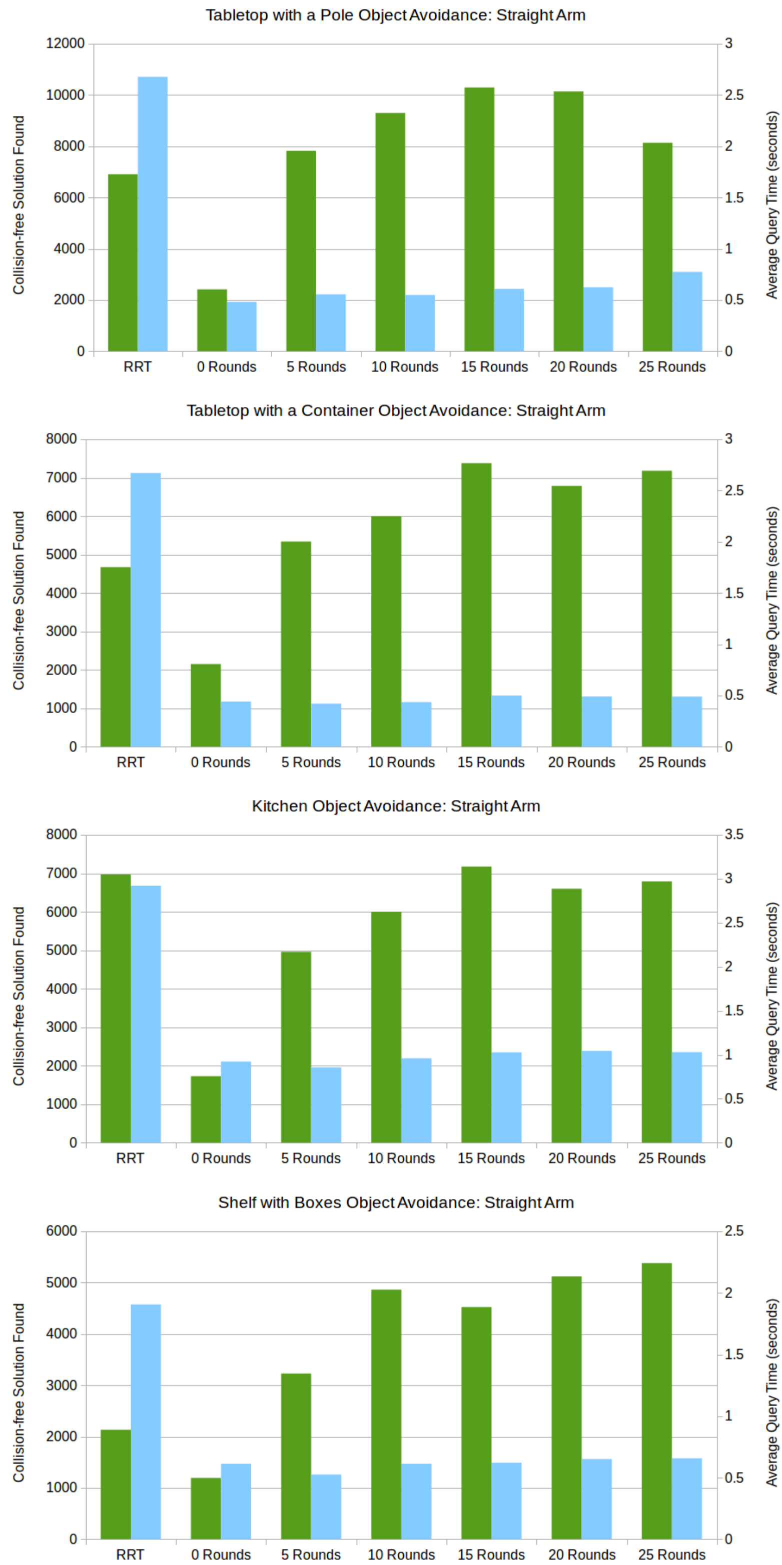


Figure A-15: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Straight Arm obstacle for all four environments

Figure A-16: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Cup obstacle for all four environments

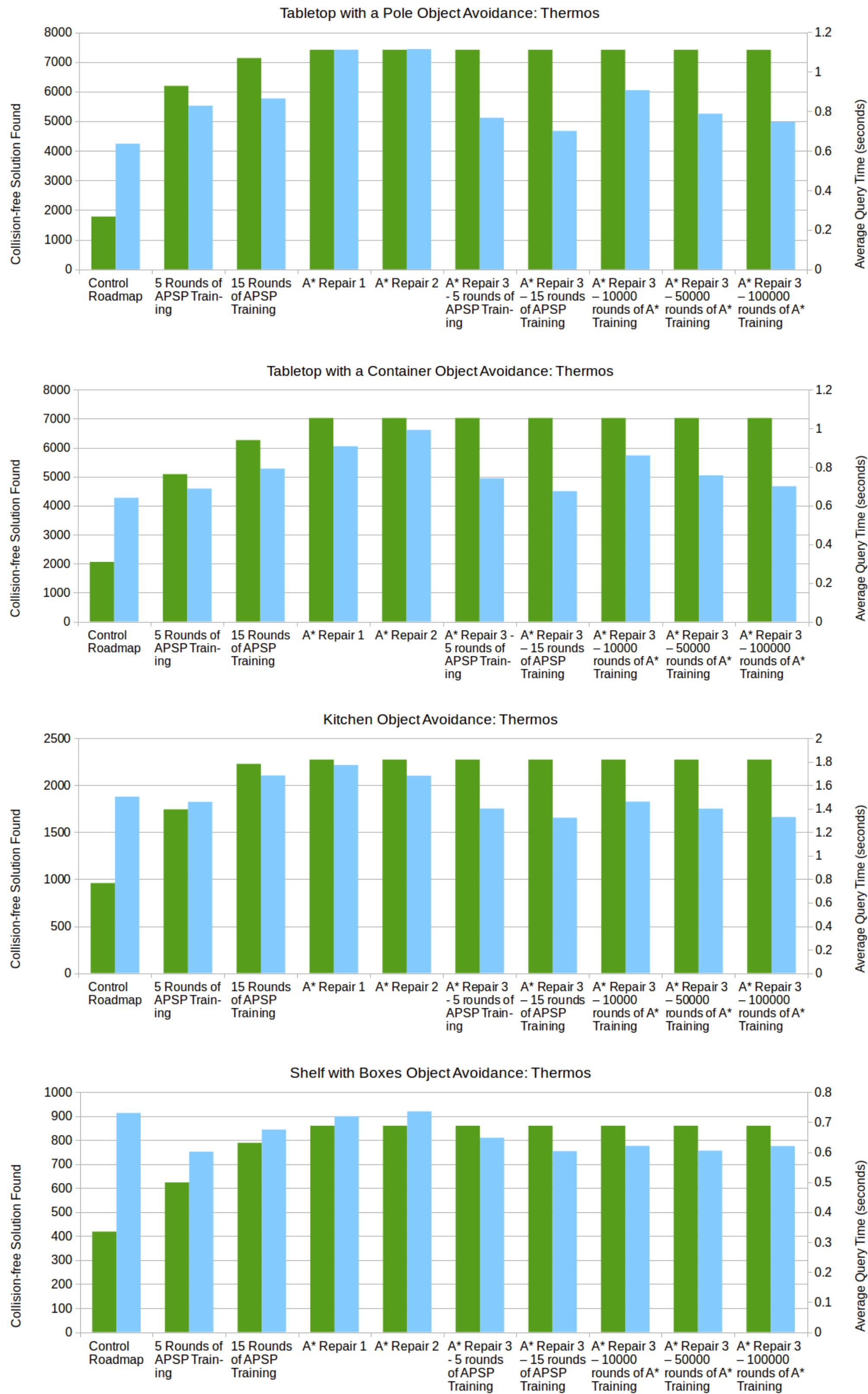


Figure A-17: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Thermos obstacle for all four environments

Figure A-18: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Monitor obstacle for all four environments

Figure A-19: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Bent Arm obstacle for all four environments

Figure A-20: Performance of solution caches developed with APSP Training in obstacle insertion experiment with Straight Arm obstacle for all four environments

Bibliography

[1] 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.

[2] 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.

[3] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on.* IEEE, 2009, pp. 489–494.

[4] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa, "Chomp: Covariant hamiltonian optimization for motion planning," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1164–1193, 2013.

[5] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "Stomp: Stochastic trajectory optimization for motion planning," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on.* IEEE, 2011, pp. 4569–4574.

[6] C. Park, J. Pan, and D. Manocha, "Itomp: Incremental trajectory optimization for real-time replanning in dynamic environments." in *ICAPS*, 2012.

[7] J. Schulman, J. Ho, A. X. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization." in *Robotics: science and systems*, vol. 9, no. 1. Citeseer, 2013, pp. 1–10.

[8] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.

[9] S. Dai, M. Orton, S. Schaffert, A. Hofmann, and B. Williams, "Improving trajectory optimization using a roadmap framework." International Conference on Intelligent Robots and Systems, 2018.

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

[11] F. R. C. for Computer Science at the Karlsruhe Institute of Technology, "Gpu voxels," http://www.gpu-voxels.org/author/gpuvoxels/.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[13] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[14] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, 2002, pp. 476–483.

[15] A. Hofmann, E. Fernandez, J. Helbert, S. Smith, and B. Williams, "Reactive integrated motion planning and execution." AAAI Press/International Joint Conferences on Artificial Intelligence, 2015.

[16] RethinkRobotics, "Baxter," http://www.rethinkrobotics.com/baxter/.

[17] B. Technology, "The wam arm," https://www.barrett.com/wam-arm/.

[18] S. J. Levine and B. C. Williams, "Concurrent plan recognition and execution for human-robot teams," *International Conference on Automated Planning and Scheduling*, 2014.

[19] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[20] K. Lab, "The open motion planning library," https://ompl.kavrakilab.org/.

[21] X. Sun, S. Koenig, and W. Yeoh, "Generalized adaptive a*," in *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1.* International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 469–476.

[22] C. Hernandez, J. A. Baier, and R. J. A. Acha, "Making a* run faster than d*-lite for path-planning in partially known terrain," 2014.

[23] C. Hernandez, R. Asin, and J. Baier, "Reusing previously found a* paths for fast goal-directed navigation in dynamic terrain," 2015.

[24] ——, "Improving mpgaa* for extended visibility ranges," 2017.