Methods for Improving Motion Planning Using Experience

by

David Thornton Coleman IV

B.S., Georgia Institute of Technology, 2010

M.S., University of Colorado Boulder, 2013

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2017

This thesis entitled:
Methods for Improving Motion Planning Using Experience
written by David Thornton Coleman IV
has been approved for the Department of Computer Science

Dr. Nikolaus Correll

Dr. Sriram Sankaranarayanan

Dr. Christoffer Heckman

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Coleman IV, David Thornton (Ph.D., Computer Science)

Methods for Improving Motion Planning Using Experience

Thesis directed by Dr. Nikolaus Correll

This thesis introduces new approaches to improve robotic motion planning by learning from past experiences especially suited for high-dimensional configuration spaces (c-spaces) with many invariant constraints. This *experience-based motion planning (EBMP)* paradigm reduces query resolution time, improves the quality of paths, and results in more predictable motions than typical probabilistic methods. Most previous approaches to motion planning have discarded past solution results and planned from scratch new solutions for every problem. A robot that is in operation for years will never get any better at its routine tasks. This thesis is novel in its focus on efficiently recalling previous motions the robot has performed and generalizing them to arbitrary new solutions even in the midst of changing obstacle environments.

Several key difficulties present themselves in the reuse of previous experiences: efficient storage given memory constraints, quick recall for new queries, verification given changing environments, and adaptation/repair. These challenges are largely addressed by the use of sparse roadmaps that provide theoretical guarantees for asymptotic-near optimality, and lazy collision checking which allows iterative search through a large roadmap of motions. Improved sparse roadmap data structures for experience storage are presented that are optimized for the L^1-norm metric and large c-spaces. The trade-offs of full preprocessing of an experience roadmap for invariant constraints is studied.

These new approaches are applied to the high-dimensional problems of humanoid whole body manipulation, dual-arm shelf picking, and multi-modal underconstrained Cartesian planning. Experiments are implemented in the MoveIt! Motion Planning framework and takeaways from developing robot-agnostic motion planning software are presented. Experimental results show two orders of magnitude speedups for solving difficult motion planning problems.

Dedication

This thesis is dedicated to the open source ROS, MoveIt!, and OMPL communities, for teaching me by example the tools and techniques to build robot hardware and software. This large body of work greatly inspired the direction of my research. May open source software help democratize the abundance the future of robotics holds for humanity.

Acknowledgements

I would like to thank my mentors around the world for guiding me in my robotics research journey. The late Dr. Mike Stilman, my favorite professor at Georgia Tech, whose mentorship and lightheartedness started me on my robotics career. My adviser Dr. Nikolaus Correll who convinced me to pursue a Ph.D., trusted my abilities, and casually suggested I focus on robotics arms—a topic to which I stubbornly adhered throughout my research. My Willow Garage mentors Dr. Ioan Sucan, Dr. E. Gil Jones, and Dr. Mac Mason for showing me the power of ROS and the field of manipulation. My OSRF mentors Dr. Nate Koenig, Dr. Morgan Quigley, and Dr. John Hsu for teaching me about simulation, controls, road biking, and Baxter hacking. My visiting researcher mentor at JSK, University of Tokyo, Dr. Kei Okada: for teaching me about humanoid whole body motion planning and IK. My OMPL summer of code mentor Dr. Mark Moll for continuously providing valuable feedback in my research direction and insights into motion planning. My Google Robotics internship mentors Dr. Advait Jain and Dr. Emily Cooper for showing me the importance of closed-loop control, in-hand sensing, and disdain for motion planning. My ROS Industrial mentors Shaun Edwards and Paul Hvass for their efforts in supporting and encouraging my MoveIt! maintenance and development. Colleague friends: Dr. Jonathan Bohren for routinely complaining with me about the many problems of ROS, Andy McEvoy for his dedication to the Amazon Picking Challenge and our weekly lunches, and Dr. Ioan Sucan for creating so much for the motion planning field, advising my MoveIt! software work and inspiring much of my research.

Finally Mom—for teaching me how to read and write and for reviewing all those papers since childhood—and Dr. Dad for inspiring me to be an engineer and scientist.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

1.1 Motivation

Motion planning is a maturing and central field in robotics [119] that addresses the problem of computing a set of inputs to a robot's actuators that move it from a start position to a goal position given various constraints such as avoiding collision. This could include navigation, such as walking through a building, manipulation, such as picking an item from a shelf, or other processes, such as welding along a line. All these problems are computationally difficult to solve due to various constraints: a humanoid robot must remain stable and has around 30 joints to solve concurrently, reaching into tight shelf compartments restricts the available set of feasible motions, and welding along a line requires computing a path through a potentially infinite set of underconstrained tip positions. Certain properties are desirable when solving these problems: first, the *faster* the robot can automatically generate the motions the more useful it is. Another desirable property is that the robot finds an efficient solution that *minimizes execution time* or effort. Finally, *consistency* is useful when working around humans because the generated paths are more predictable and therefore safer.

One of the promises of robotics is to eliminate repetitive tasks that have little variance, such as fulfilling warehouse orders by reaching into a shelf (Figure 1.1) or picking up randomly positioned and varying sized objects off a conveyor belt. In these problems much—but not all—of the environment remains constant, and the motions being performed by the robot are largely the same. Because of the variance in environment, the traditional approach of manually hard-coding

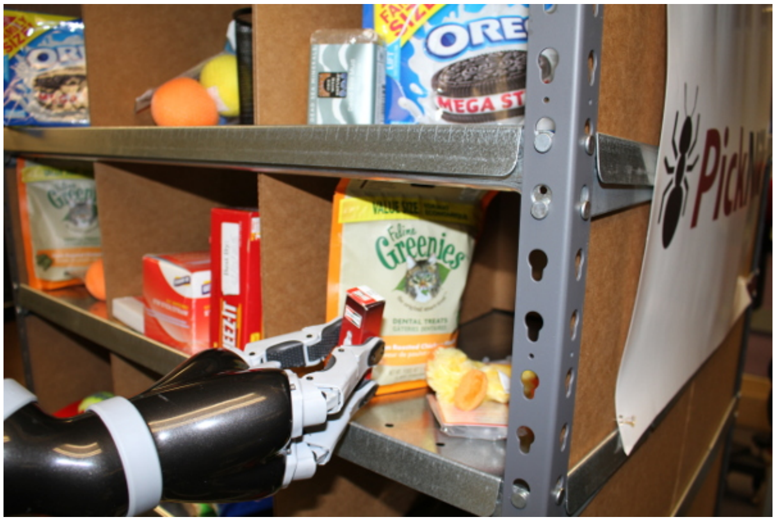


Figure 1.1: A storage shelf from an Amazon warehouse with a Kinova Jaco 2 arm reaching into a confined space for products used in the Amazon Picking Challenge [37]

exact paths to repeat do not apply. However, solving these queries each time from *scratch* is also inefficient because there is so much similarity from previous queries that could be reused. Yet most motion planning algorithms used in robotics today for non-hard coded trajectory generation are single-query planners [28] that solve the same problems from scratch over and over again without reuse of past experience. The emerging field of experience-based motion planning (EBMP) stores in memory some aspect of past solutions to improve future requests. EBMP is a form of lifelong learning [160], generalizing previous plans to speedup and improve future planning problems.

A simple motivating observation of this school of thought is the human ability to generate trajectories "amazingly" quickly [70]. Humans seem to require no time to motion plan but can execute complex trajectories instantly. From this observation, it has been suggested there exists in humans a *reactive trajectory policy* which maps the motion-relevant features of a situation to an entire trajectory [70]. This mapping, however, is extremely complex and must take into account collision avoidance, smoothness, and other criteria.

Despite continued improvements in the speed of computers, Moore's law has begun to slow down [17]—the methods in this thesis offer alternative speedups to only processing power by leveraging increased memory usage. EBMP could be considered just motion planning with caching, but many difficult problems remain with efficiently saving and recalling past solutions. As the dimensionality of the planning problem increases, the necessary size of a naive cache implementation is not feasible. Determining how many experiences should be saved and in which format to save them is one problem. Quickly recalling the best experience—if one exists—is another issue that becomes especially hard when there are varying constraints and collision environments.

Efficiently repairing a partial solution from recall is another challenge. As a robot moves from different tasks, such as working in a kitchen versus climbing a ladder, it is important that changing obstacle locations can efficiently be considered when recalling a path. Repairing a recalled solution to be valid in the current planning problem is an active area of research [2], and as such no consensus exists as to the best method for performing this.

The increasing expectations of robots to solve more complex problems is another reason to focus on improving the speed of motion planning algorithms. For example, humanoid robots with 30 or more degrees of freedom (DOF) require not only planning in the joint configuration space (c-space) but also must account for dynamics, which at a minimum doubles the size of the already large search space. Optimal planning with complex cost functions also increases the complexity, for example finding the shortest path while simultaneously maintaining a safe distance from nearby obstacles. The emerging field of cloud-based robotics motivates even more the practicality of building large datasets of experience [88].

To build an efficient experience database, it is our goal to have a roadmap that can capture all past homotopic classes a robot has experienced (Figure 1.2). This means that for difficult path planning problems, such as the narrow passage problem [64], at least one path is saved that traverses through this constrained region, and it can be later recalled and continuously deformed into a near-optimal solution.

EBMP is especially suited for robots with a large amount of invariant constraints, such as joint limits, self-collision, and stability constraints because these expensive checks can be inherently encoded in the experience database. Whenever a path is recalled, it is guaranteed to already satisfy the invariant constraints. *Variant* constraints are mainly collision checks with changing obstacles.

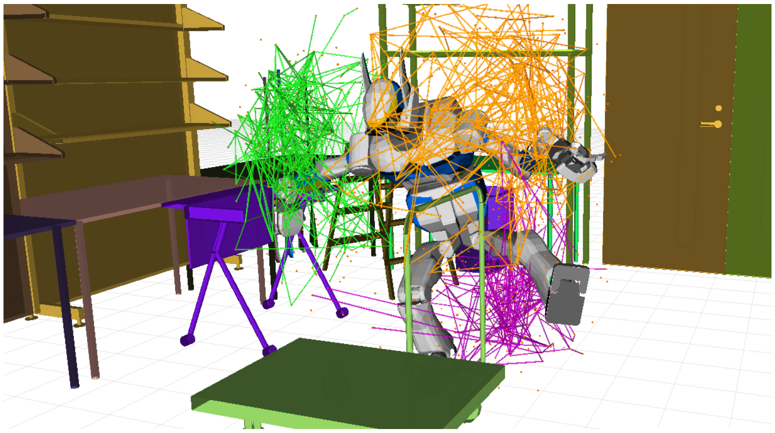


Figure 1.2: Visualization of an experience database for a humanoid robot after 10,000 runs with obstacles. The three different colored line graphs project a whole body state to a reduced image of each of the robot's two end effectors and free foot.

An example of a highly invariant-constrained robot is a biped humanoid, which although kinematically has a large range of motion, has only a limited free space that satisfies its stability constraints. Another example is a dual-arm robot, where a significant fraction of random configurations of the arms is invalid due to collision between the two arms.

A final motivation for EBMP is improved predictability of planned motions. Traditional probabilistic methods are less desirable in applications such as hardened industrial environments where safety and reliability are highly valued. An EBMP planner becomes more deterministic over time as it learns to solve repeated problems from recall. Understanding what a robot will do makes humans more comfortable when interacting with it.

1.2 Definition of Problems

Here we formally define the motion planning problem and the generation of sparse roadmaps problem.

The configuration space (c-space) C is the representation of a robotic system (sometimes referred to as a state space) as a set of values useful in the motion planning problem. The point $q \in C$ is able to fully capture a state of the system being planned. This typically includes the joint angles of a robot, the orientation and position of a robot's end effector, and/or the position of the robot with respect to a world coordinate system. The c-space can also include dynamics, such as derivatives of the dimensions (e.g. velocity). In this thesis the focus is on robotic arm manipulation, and as such the primary c-space is the d-dimensional joint angles of one or more robotic arms and legs. hereby referred to as the *joint space*.

The *free space* C_{free} is the subset of points in C that are valid configurations of the system. This is typically defined by a function $isValid$ that can contain any number of constraints including joint limits, collision checking, orientation constraints, and stability constraints. The converse of C_{free} is C_{obs}—the set of invalid states due to *obstacles* in C.

The start q_{start} and goal q_{goal} configurations can be any number of states in C_{free}. Often q_{start} is just a single state: the robot's current state. Conversely q_{goal} frequently has many configurations due to, for example, the infinite solution space of a redundant robotic arm for a particular end effector pose.

Path planning under Geometric Constraints The path planning problem under geometric constraints is the task of finding the shortest continuous path $\{q | q : [0, 1] \rightarrow C_{free}\}$ where $\pi(0) = q_{start}$, $\pi(1) = q_{goal}$. In practice, most motion planning algorithms actually find a path of the form $[q_0, ..., q_n] \in C_{free}$ where each q_i can be trivially connected to q_{i+1} using e.g. linear interpolation. A common assumption in this path planning problem is that the robot can move instantaneously in any direction; dynamics are ignored and only addressed in post-processing.

In most motion planning algorithms a metric function $d(q_1, q_2) \rightarrow \mathbb{R}$ is used in C to find the distance between two configurations. Throughout this thesis the L^1-norm metric function (*Manhattan distance*) is used, the rationale can be found in the introduction of Chapter 5. Additionally, our cost function is always the shortest path.

Representation of a Free Space The motion planning algorithms presented in this thesis generate a graph $G(V, E)$ where V is the set of vertices that correspond to valid $q \in C_{free}$ and E represent local valid paths $L(q_1, q_2)$ between two points in V. The local planner used throughout this thesis is straight-line interpolation between two points.

Generation of a sparse roadmap We wish to create a compact representation $G_S(V_S \in$

$V, E_S \in E$) where:

- All vertices $q_i \in G$ can connect with a vertex $q_j \in G_S$ with a valid local path $L(q_i, q_j)$.
- G_S has as many connected components as G
- All shortest paths in G are no longer than t times the corresponding shortest path in G.

Asymptotically near-optimal with additive cost The generated G_S shall be *asymptotically near-optimal with additive cost* if, for a cost function c with an optimal path of finite cost c^*, the probability a path will be found with cost $c \leq t \cdot c^* + \varepsilon$ for a stretch factor $t \geq 1$ and additive cost $\varepsilon \geq 0$, converges to 1 as the number of samples approach infinity [43]. The additive cost $\varepsilon \leq 4 \cdot \Delta$ is the connection cost for getting onto G_S.

1.3 Hypothesis and Contributions

In this thesis, we propose a framework for experience-based motion planning (EBMP) that improves performance by generalizing, storing, recalling, and repairing past motion plans. We show that for easy planning problems we can match or outperform current approaches, and for difficult problems we demonstrate two orders of magnitude improvements in planning time. This approach trades off memory for runtime performance to improve the search time, allowing more difficult problems with larger c-spaces and harder constraints to be solved in a computationally tractable manner. The key idea is that we generate a sparse roadmap that can properly capture the various *useful* homotopic classes in a space while still allowing for changing environments. This is different from a traditional probabilistic roadmap in that 1) we do not randomly attempt to connect the entire c-space but rather we only save paths/vertices/edges of motions we have actually used in the past and 2) we do not try to save all motions but only those within a certain resolution, using graph spanner guarantees [42]. By saving only past experiences and to only a certain resolution, we reduce the size of the roadmap to search for past experiences, which increases search time.

In some sense, this approach is more similar to search-based motion planning methods (e.g. grid-search) in that we are discretizing to a certain resolution of the c-space. But it is significantly different in that the discretization is randomly generated using probabilistic search methods similar to Rapidly Exploring Random Tree (RRT), and due to the sparse roadmap spanner guarantees, our resolution can become more fine-grained as needed for narrow passages. In this way, we can capture more homotopic classes without having an overly discretized graph.

This thesis presents the following contributions:

- An EBMP framework called *Thunder* that accelerates motion planning while providing asymptotically near-optimal guarantees on path length and completeness guarantees. It applies sparse roadmaps for efficiently storing and recalling experiences—focusing only on paths that have been used before. To maintain completeness, a secondary thread is run using the popular sampling-based single-query planner RRT-Connect [89].

- An alternative EBMP approach that fully preprocesses a roadmap for invariant constraints, called *Bolt*. This approach has the advantage that no Planning from Scratch (PFS) is likely necessary for future arbitrary planning queries, but has the disadvantage that in large c-spaces the datastructures grow cumbersome.

- An improved set of criteria for graph spanners that are optimized for the L^1 space rather than the L^2 Euclidean space. These criteria are able to generate roadmaps 77% smaller, a critical component for our fully preprocessed approach.

- A simplified set of criteria for graph spanners that are fast enough to make full preprocessing of a high DOF c-space computationally tractable with the trade-off of no asymptotically near-optimal guarantees.

- An experience-optimized RRT-Connect variant we call Experience-RRT-Connect (ERRT-Connect) that leverages our experience database to grow its dual trees through narrow passageways more efficiently. It is based on the observation that for manipulation tasks the most confined areas are typically near the start and goal states, and therefore feeding samples from the experience database that are from this area of the c-space have a high probability of growing the tree faster.

- A multi-modal underconstrained Cartesian planner that is able to plan in one unified motion planning problem a free space approach path, a Cartesian path, and finally a free space retreat path. This planner uses a discrete task dimension to plan through the two subgoals, and leverages our efficient roadmap preprocessing work in *Bolt*.
- Takeaways from making the robot-agnostic motion planning and manipulation framework MoveIt! both powerful and easy to use.

The contributions of this thesis make previously computationally difficult, high-dimensional problems more efficient and consistent in real-world problems.

1.4 Organization of the Thesis

Chapter 2 of this thesis gives background on past and current approaches to solving the motion planning problem, including related techniques for EBMP. Chapter 3 describes the robot-agnostic motion planning framework MoveIt! that has greatly motivated this thesis research. Chapters 4, 5, and 6 describe varying approaches to EBMP. Chapter 7 demonstrates the use of a preprocessed experience roadmap to solve a novel a multi-modal underconstrained Cartesian problem. Chapter 8 presents final improvements to our Thunder algorithm. Chapter 9 is our conclusion and final remarks.

Chapter 2

Background

2.1 The Motion Planning Field

There are two prominent fields of motion planning: classic grid search methods and sampling-based methods. Classic Grid Search (CGS) motion planning discretizes the search space and searches for paths using traditional search algorithms like A* [60]. CGS has been popular with mobile robot navigation such as steering a robot through a building, but until recently the approaches have been thought too limited for motion planning problems beyond three dimensions. They have seen a recent comeback in popularity due to motion primitives and new low-dimensional projection heuristics, though the latter are hard to come up with for complicated robot geometries. Most CGS methods use heuristics to focus search in the form of approximations of the goal distances. This results in much faster solution time than uninformed search algorithms. However, finding an admissible heuristic for higher dimensional spaces can be a tough problem [92].

Many sampling-based approaches have been developed to address some of the limitations of CGS that have seen widespread use over the last decade, They have become very popular due to their overall simplicity and speed at which they can solve difficult high-dimensional problems. They typically use probabilistic approaches to solve the problem, which have been proven to provide probabilistically-complete guarantees on finding a solution if a solution exists as time goes to infinity. The major trade-off in most sampling-based algorithms, however, is that they are not optimal and can often produce unnatural and inefficient solutions. Sampling-based approaches are typically partitioned into two categories: multi-query and single-query. Most multi-query approaches are based on the Probabilistic Roadmap (PRM) [80, 81]. Single-query approaches are typically based on the Rapidly Exploring Random Tree (RRT) [96] algorithm and it's tree construction. RRTs are arguably the most popular motion planning algorithm today due to their simplicity.

Beyond classic grid search-based and sampling-based methods, recent advancements have presented novel new approaches to motion planning. Contact-Invariant Optimization (CIO) [122] introduced a planning method for complex full body motions that automatically chooses contact points for an arbitrary number of end effectors. While utilizing a physics simulation to plan with dynamics, the optimization approach is computationally expensive (5-20 minutes preprocessing time) while producing non-physically realistic motions that penalize, but do not prohibit, violating joint limits, self collision, and disconnecting various links of the robot's bodies. The dynamics assumes a point mass in the middle of the robot's torso. The generated trajectory must have its waypoints count pre-specified and each waypoint has non-variant timestep size. Additional difficulties present themselves when applying the dyamical trajectory to a real robot due to modeling error and state estimation uncertainty, which was overcome in simple walking examples through the use of ensembles of perturbed models [121].

Applying deep learning to motion planning and manipulation is another recent advancement that holds future promise. In [100] a convolutional neural network is trained to predict the probability of success of grasps during hand-eye coordinated manipulation tasks. By using many robots over 800,000 grasp attempts, a diverse set of training data is generated that is independent of calibration or small hardware differences. While demonstrating great promise in the simple task of grasping from a flat surface, and despite requiring a tremendous amount of training data, the approach has not been shown to generalize well to complex environments such as picking from multiple shelves or around small obstacles. Additionally, the neural network approach is unable to choose which item to pick from the bin, only picking the item with the highest probability of success.

2.2 High-Dimensional Spaces

A major motivation for experience-based motion planning (EBMP) is the growing need for very large configuration spaces (c-spaces) as demands for robotic performance increase. The most common source of increased dimensionality is the number of degrees of freedom (DOF) a c-space contains. Simple, low-dimensional problems such as car steering that have a small c-space of $n = 3$ (e.g. x,y,θ) can be easily solved with traditional search algorithms. The term "high-dimensional" is a moving target as computers get faster and methods get better—in 1998 a "many DOF" robot had "5 or more DOF" [81], but currently one might more likely call a dual-arm fixed-base robot such as Rethink Robotics' *Baxter* [140] as high-dimensional as there are 14 DOF. Today's robots are getting even more complex, and biped humanoids such as Kawada Industries' HRP2 [77] has 28 joints and, when including its pose with respect to the world, has a total of 34 DOF.

Object manipulation can also add more DOF to the c-space, for example, the position and orientation of the object can be optionally taken into account in the c-space. Similar problems such as the manipulation of flexible objects can be solved by discretized the object and simulating them with flexible joints, adding even more DOF to the planning problem. In [23] the simulation of a "Y" shaped flexible object added 50 additional DOF.

Planning in just the kinematic space is not enough for many robot applications such as bipeds who need to walk or move with more than just quasi-static stability. Kinodynamic motion planning is another heavily researched field [45, 65, 92, 116] that typically involves finding the minimal-time trajectory of state and control curves that respects constraints on velocity and acceleration. Kinodynamics planning requires at least one additional dimension for every joint on the robot, which is used to represent velocity [92]. A third additional dimension can be added for every joint to represent acceleration.

Another challenge in of kinodynamic planning is the resulting nonholonomic property of a robot. In a generated probabilistic roadmap or optimal tree-based planner, it must be ensured that the graph is directed and the plan only moves forward with time. Directed graphs have received little attention in the EBMP field.

To make the planning problem even more difficult, it is often desired that various optimization criteria be considered. Distance to nearest obstacle, motion time, and jerk are all examples of cost functions that need to be minimized. This greatly increases the search space and planning time.

2.3 Experience-Based Planning Approaches

The many varying approaches to generating motion plans using past experience are now presented. Although the words used to describe them sometimes differ, there exists much overlap in their techniques and purpose. In the following, the term *experience* is used to indicate a generated *action*, *path*, or *planning solution* to a particular task. Additionally, the terms *experience database*, *experience graphs*, and *experience roadmaps* are sometimes used interchangeably, where they all are intended to mean methods for storing previous *experiences* and data for future reuse, but the later two refer specifically to types of graph structures for storage.

2.3.1 Categorization of Approaches

Here we present two approaches to categorizing EBMP:

2.3.1.1 Methods for Generating Experience

One way to categorize varying EBMP approaches is by the method experiences are generated. Approaches employing *incremental search* are myopic experience planning methods that reuse only the most recent past experiences and typically still require Planning from Scratch (PFS) for the first planning episode. They focus on replanning for problems with similar or the exact start and goals. Approaches that automatically pre-compute all possible inputs and outputs, to some discretization, and save the results in a library for future lookup, will be referred to as *lookup table* or *trajectory library* approaches. *Learning by Demonstration* approaches rely on human experts to train robots in their desired behavior and have the robots learn from those initial inputs. Finally, methods that *incrementally build* or *learn* their experience database during the lifetime of the robot begin with no previous experiences generated and only save experiences they actually need.

One might be tempted to assume that incremental learning only slows down the online performance of planners (while saving on offline preprocessing), but one major advantage of the approach is that the experience database will likely remain much leaner than a database that is precomputed for all possible actions. This advantage is from the intuition that there exist many actions that a particular robot may never use.

A leaner experience database will typically result in faster search times because of reduced node expansions during graph search. The challenges in this approach include the need to have a good fallback when no suitable previous experience is available and the ability to determine which new experiences to save to disallow the experience database from growing indefinitely.

2.3.1.2 Methods for Saving and Recalling Experience

Another way to organized EBMP approaches is by the *storage and retrieval* methods used, as opposed to the methods for *generating* experiences. These approaches will be organized from the most basic to the subjectively more advanced methods. This categorization method better captures the diverse approaches to EBMP.

2.3.2 Incremental Search for Dynamic Environments

One common form of EBMP is reuse of recent solutions for similar problems. Traditionally called *incremental search*, more recent works have called it *rapid replanning.* The term *dynamic environments* is also used to describe similar planning problems that have changing topology, edge costs, start states, and goal states. Unfortunately, the term “dynamics path planning” can be problematic since the term “planning with dynamics” also can also refer to planning with control dynamics, or “kinodynamic planning”.

Incremental search for dynamic environments (ISDE) is limited in our EBMP context by its reliance on similar start states, goal states, and environments. It is our hope that a true EBMP is applicable to solving much more diverse sets of planning problems and environments. Still, ISDE is motivated by the need for highly responsive algorithms and achieves this by exploiting temporal and spacial coherence and reusing information across a series of planning iterations, or *planning episodes*. Several possible events may occur between replanning iterations: 1) the robot may have moved, 2) new obstacles were discovered, 3) the goal may have moved, and 4) obstacles may have moved independent of any action by the robot.

Incremental search is historically common in applications like autonomous vehicle navigation but is arguably needed in almost all real-world robotics due to incomplete and changing environments. As with planning in general, two conventional approaches emerge in the incremental search field: classic grid-based search approaches that use a discretized c-space and sampling-based planning using a tree-like structure.

2.3.2.1 Classic Incremental Grid Search

Classic incremental grid search has been solved by search-based planning methods such as A* to find a valid path over a discretized c-space. These methods solve *dynamic shortest path problems* where the shortest path has to be determined repeatedly as the topology or edge costs of a graph changes. Many incremental search methods have been proposed in the algorithms literature [8, 50, 105, 46]. They differ in their assumptions on what level of dynamic-ness of the graph is allowed, but none are informed.

2.3.2.2 Anytime A* Algorithms

Anytime A* (ARA*) [104] greatly improves the speed of A* by sacrificing optimality for a quick initial solution, but then it is able to efficiently improve the initial solution as time allows. However, it is not able to replan for dynamic environments; it is only able to improve the optimality of the same planning problem without any changes to the environment. It accomplishes the anytime property by using an inflated heuristic that still provides bounds on a solution's sub-optimality, and a clever method for reusing past g-values (past path-cost) in A* search.

2.3.2.3 D* Algorithms

A major improvement to incremental search methods is the use of heuristic search to focus its search and achieve faster results. The first truly [85] incremental heuristic search method was D* [152] and the slightly improved Focused D* [153] methods. They are dynamic programming-based approaches that extend A* by updating the g-values from the previous searches to correct them when necessary for the current search and maintain an optimal path.

Lifelong Planning A* (LPA*) [85] is a much more simplified and theoretically proven incremental search algorithm similar to D* that combines the strengths of incremental search and heuristic search. It behaves exactly the same as A* on the first planning iteration, and thereafter performs much faster than PFS especially when the problem has only slightly changed, and the changes are mostly near the goal state.

LPA* was then extended to behave similarly to D*, resulting in D*Lite [84] algorithm that is at least as efficient as D*. D*Lite and D* are similar in that both search in reverse from the goal to the start state, both use heuristics to focus their search, both propagate cost changes in two waves, and both stop when the smallest key of all vertices in the priority queue is greater or equal to the key of the robot's current state. Yet D*Lite and D* work differently internally, particularly in that D*Lite guarantees that each vertex is expanded no more than once per replanning episode.

One limitation of these D* algorithms is the constraint that movement is limited along graph edges or discrete transitions between cells that produce paths that are not optimal. Both E* Interpolated Path Replanner [131] and Field D* [49] address this issue with methods for finding solutions in the *continuous* domain by interpolating between edges. This reduces or avoids the need for post-plan smoothing. D*Lite is arguably the most popular incremental search algorithm today but still suffers in high-dimensional spaces. This limitation is partly because of the difficulty in creating heuristics for complex c-spaces.

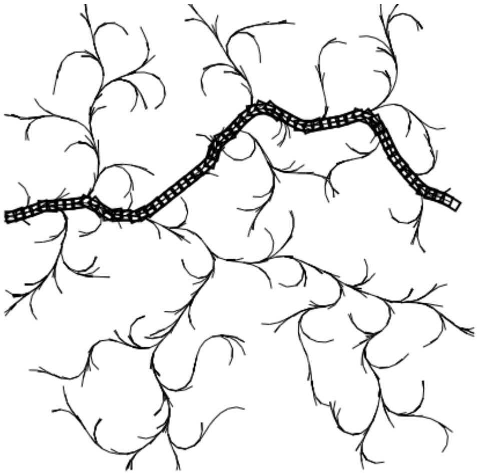


Figure 2.1: An example of a RRT tree in a 2D c-space with the solved path in bold.

2.3.2.4 Sampling-based Incremental Search

Sampling-based planning methods achieve incremental search by reusing parts of recent search trees for similar, but slightly modified, planning problems. In this section PRM-style multi-query sampling-based methods are excluded because the roadmaps that are being reused are useful not just for immediate queries but also for all future queries.

RRTs randomly grow one or more trees through the c-space as shown in Figure 2.1. They were originally developed to *not* be reused between search queries, the idea being that they are so much faster to generate, and so sparse, that there is no point to reuse them. However, their speed has proven to be useful in its own right, and new uses such as optimal RRT planning using RRT* [78] have motivated researchers to explore ways to rapidly replan with RRTs. A number of approaches exist for reusing old search trees in dynamics environments [20, 101, 48, 165]:

In Execution-extended RRT (ERRT) [20] the search tree is reused by maintaining a fixed-size waypoint cache of past states that are randomly replaced with new states as planning iterations are performed. During replanning, the waypoint cache is sampled with some probability to bias replanning towards previous solutions, working under the assumption that the world has not changed much between iterations. ERRT improves naive RRT when small changes occur to c-space, but the approach still requires that the RRT be rebuilt from scratch every time the current solution is invalidated, typically resulting in far more work being performed than necessary.

In Reconfigurable Random Forests (RRF) [101] a roadmap of the environment is created using several different RRTs rooted at different locations. The individual RRTs are checked periodically to see if they can be connected together, similar to the RRT-Connect algorithm [89]. When the environment changes, newly-invalidated edges in the forest are removed, resulting in new trees formed from the branches that were removed.

In Dynamic RRT (DRRT) [48], efficient pruning and repair of the planning tree is achieved similar to RRF, except only a single tree is maintained. Whenever the environment changes, invalid branches and all connected subtrees simply are removed, and then planning continues on the abridged tree. It is also suggested that planning occur in reverse—from goal to start—to allow more tree reuse when the robot location changes. In practice DRRT allows reuse of very large subtrees. This approach, however, does not work well when both the goal and start state change, and its simplicity ignores the cases where an obstacle directly in front of the robot results in the entire tree being discarded, resulting in complete replanning.

In Multipartite RRTs (MP-RRT) [165] the strength of biasing sampling towards previously useful states as in ERRT is combined with reuse of previous RRTs as in DRRT. Similar to RRF, a forest of disconnected RRTs is maintained that is created as invalidated edges disconnect the original RRT. When replanning, sampling is biased with some probability towards the roots of the disconnected subtrees to encourage reuse of disconnected components. Subtrees eventually are removed if they are not used after a certain number of replanning iterations. MP-RRT is able to effectively overcome some of the other limitations in dynamic planning approaches, but its use is still focused on reusing experience on only slightly changed environments and states.

2.3.2.5 Genetic and Evolutionary Incremental Search

More unique approaches to EBMP in dynamic environments take inspiration from genetic algorithms and evolutionary algorithms. In the Real-Time Adaptive Motion Planning (RAMP) [161] approach, no tree or roadmap is generated, but instead a diverse *population* of candidate trajectories is continuously maintained and scored for feasibility and optimality. The best scored trajectory is constantly chosen for use in the real-time control of the robot, allowing for dynamic obstacles to be handled. At every planning cycle, the trajectories are randomly modified and re-scored, giving the approach its "genetic" properties. RAMP utilizes recent experience by continuously updating its previous candidates' trajectories with respect to new start positions (as the robot moves) and new obstacle locations.

In the Rapidly Exploring Evolutionary Tree (RET) algorithm [115], an offline preprocessing phase uses an evolutionary algorithm to generate a smart sampler for prior known environments. This allows RRTs to be biased towards the edges of the explored areas for faster replanning but is limited to already known environments with little or no changes. While RET results in fewer tree nodes being visited, it is not general enough to be considered an effective EBMP planner.

2.3.3 Saving Experiences Independently

In the previous section, rapid replanning was discussed for similar queries, but the drawback is the inability to generalize those similar queries to much more diverse situations. EBMP is now expanded to using all past experiences by storing each past experience separately and recalling the most relevant experience from that set. With this approach, a new challenge emerges of identifying the best past experience from the set of all available experiences for a particular planning query despite changes in constraints such as the collision environment. Mapping from a current query to a particular past experience to recall must be efficient in practice.

From the artificial intelligence field, saving experiences independently, then finding the most relevant experience, is a form of *instance-based learning* [141] that compares new problem instances with instances previously seen in training. This allows these approaches to adapt their models to previously unseen data by simply storing a new instance that it has not encountered before. Many of these approaches use *instance reduction* to reduce the memory complexity of storing all training instances.

2.3.3.1 Trajectory Libraries and Lookup Tables

A simple approach to saving experiences independently is pre-computing all possible inputs and outputs, to some discretization, and saving the results into a table or database for future lookup. This building of discretized representations of state reachability and c-space volumes “has been a major research thrust in robot motion planning” [18]. These approaches make the assumption that it is feasible to save all possible trajectories, or that there is a clever method for adapting experiences to similar tasks.

2.3.3.2 Recall: Using High-Dimensional Descriptors

In [156] the footsteps of quadruped robot are pre-generated in a trajectory library to improve realtime planning. To make trajectories transfer across tasks, a feature matching approach is developed that utilizes properties of the state of the system with respect to the properties of the environment. To make feature matching faster, the approach also utilizes principle component analysis to project the feature vectors to a lower dimensional space.

In [70] the difficult mapping problem from planning query to past experience is improved and generalized by utilizing a large *high-dimensional descriptor* to improve mapping prediction. In this approach a 791-dimension vector with features consisting of robot joint angles, body parts, obstacle objects and target reach locations is used in various coordinate frames. A feature selection technique is then used to reduce the dimensionality of the descriptors to include only the aspects best for path prediction in new situations. This approach is unique in that it does not save joint angles of experiences but rather only end effector poses. It then maps the end effectors poses back to joint space using an inverse kinematic (IK) solver as needed. In [71, 72] the trajectory prediction approach is extended for use with laser point clouds using voxel occupancy grids (Figure 2.4) and tested on real hardware. The high-dimensional descriptors approach is promising but in its current form seems to lack generality for other tasks; it is constrained to the exact number and types of objects used when creating the sparse descriptors. It also assumes very simple object geometry that can easily be represented in the descriptor, and does not extend to difficult tasks.

2.3.3.3 Filtering: Choosing Which Experiences to Save

The problem of choosing which experiences, and how many, to save is most pronounced when saving experiences independently. If no thought is given to this aspect, an experience database will grow unbounded until it is intractable for use on today's hardware. This problem is addressed in [18] where a large set of paths is pruned such that the paths are optimized for maximum connectivity over all possible obstacle environments. Using their methods for calculating the exact probability of collisions a *path diverse* database of collision-free paths is obtained.

2.3.3.4 Lightning Framework

The Lighting Framework [11] is an impressively simple approach to saving experiences independently. In Lightning, a parallel module concept is utilized that splits the computation into two threads for each planning query. The first thread runs a Planning from Scratch (PFS) component, which attempts to solve the problem using a traditional sampling-based planner without any prior knowledge. It is used to populate the empty experience database when the robot first starts learning, removing the need for pre-computation discussed. PFS is also important for maintaining the same guarantees of probabilistic completeness as a traditional sampling-based planner.

The second thread runs a Retrieve Repair (RR) component, which uses an experience database to find a past solution that is similar to the current problem and then attempts to repair that solution as necessary to solve the current problem. The solution from the fastest threaded component to finish in Lightning is returned as the overall result, the other thread being immediately terminated.

The Lightning Framework implements the RR module by saving entire paths of past experiences individually into a database (Figure 2.2, left) for later reuse. Only paths that are planned from scratch or differ significantly from their original recalled parent are added to the database. When a new problem is presented, Lightning retrieves the most relevant and similar experience using two heuristics. The chosen experience is then repaired using a bidirectional RRT to attempt

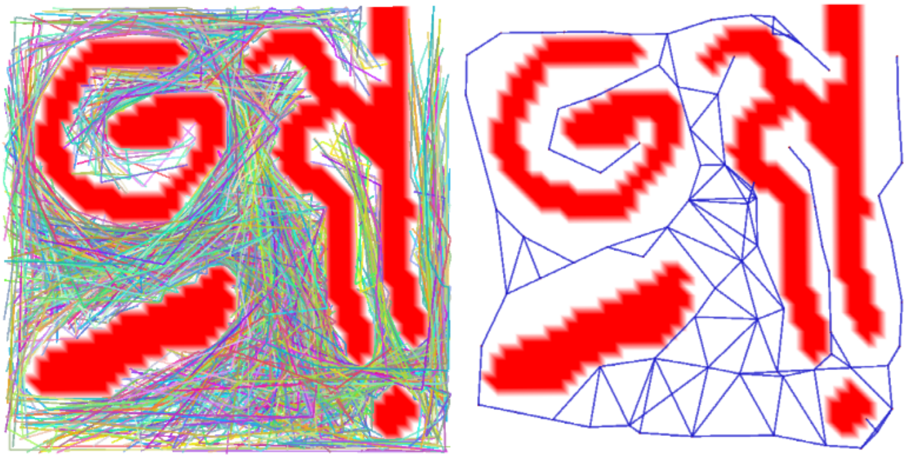


Figure 2.2: Visualization of 2000 experiences in a 2D world saved in an experience database. Left: Lightning saves many redundant edges densely. Right: sparse roadmap spanners efficiently covers the space within a stretch factor t.

to re-connect each set of end points of segments that has been disconnected by invalid regions. In many cases the computation of recalling and repairing a path, particularly difficult paths containing challenges like the narrow passage problem [64], is less than the computation required for PFS [11].

There are several drawbacks in the Lightning Framework: it is highly likely that redundant information is stored in the form of similar segments of motion due to experiences being saved disjointly in separate paths. If two experiences have significantly different start or goal states but share overlapping subsegment paths, those overlapping paths constitute redundant information. Another downside is that in changing collision environments it is assumed that a path with small invalid segments will be easy to repair, but it could also be the case that a different experience with slightly more invalid segments is in a less cluttered area of the c-space. Last, the Lightning framework suffers from limited dataset sizes, or conversely, significant memory usage due to unbounded growth.

2.3.3.5 Generation: Learning by Demonstration

In [117] it is argued that the most difficult motion planning aspects, such as navigating through narrow passages and manipulating objects, are best trained using a human with a joystick in order to achieve motions desirable to humans. They achieve push-manipulation in a 3D environment by first training a robot offline in suitable motions for manipulation. Those motions are saved and used as building blocks for a sampling-based planner that ensure safe and achievable motions. [117] combines traditional RRT planning for open spaces, with object-specific manipulation motions that are saved from the human operator and used to connect to the search trees. It is possible, however, that similar results could also be obtained from other methods such as denser sampling of motions around targeted objects for manipulation or using autonomous experience generation.

The approaches in this section all save experiences independently, which almost always result in redundant information being stored and extra memory being used. An idea explored in the following sections is to attempt to use all the information from previous searches instead of picking just the best singular previous experience.

2.3.4 Classic Grid Search Reuse

Methods to reuse classic search-based graphs also have been developed. In [132] the results of searching in a fully discretized c-space are saved in an *Experience Graph* (E-Graph). E-Graphs are populated online and grow with each task-based request. For each planning query, the search is biased towards finding a way to connect to the E-Graph, so that search can be sped up with pre-build solutions. The idea is to remain searching on the E-Graph as much as possible. The E-Graph can handle dynamic environments by an update phase that fixes edge cost changes such as collision checking.

In [133] the E-Graphs approach is extended to learning from demonstration and applied to constrained manipulation, i.e. opening doors/cabinets/drawers. While populating the experience graph from user input seems simple, their method actually requires that a new dimension be added that corresponds to the dimension of the object being manipulated, for example, how open the drawer is. Additionally, a special heuristic is developed that guides the search towards the objects that needs to be manipulated and then guides it in how to manipulate the object.

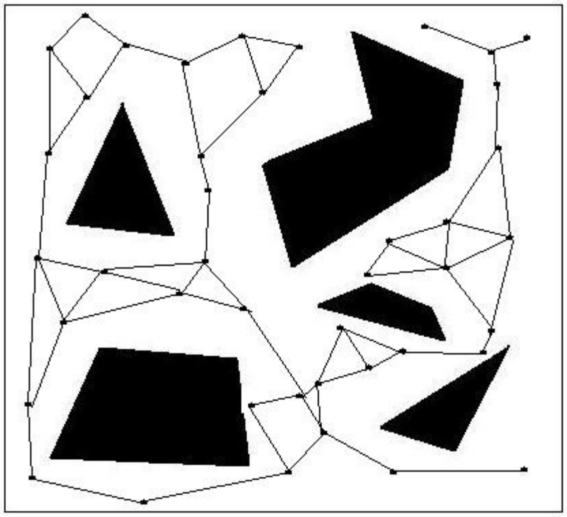


Figure 2.3: A probabilistic roadmap showing efficient coverage of a space around obstacles

2.3.5 Probabilistic Roadmap Reuse

The PRM algorithm was developed to build a roadmap, as shown in Figure 2.3, that originally was intended to be reused over multiple search queries, so it is in many senses a EBMP algorithm. The naive implementation, however, is still specific to a particular environment and when it changes (which is typically often) the entire roadmap must be recalculated. A simple modification to PRM, that also makes it more efficient in general, is to delay collision checking.

Collision checking is one of the most computationally intense components of most motion planners, and as such, reducing the amount of required collision checks is a major challenge of multi-query sampling-based incremental search. Lazy collision checking, as demonstrated in LazyPRM [14], is a popular approach that delays collision checking until after a solution through a c-space is found. If the path is found invalid, only the invalid edges and vertices are removed or marked as invalid, and search is restarted through the updated graph. Because a LazyPRM's roadmap is constructed without consideration for obstacles, it is general enough to be considered an EBMP planner. However, more efficient methods for PRM reuse are discussed in the next section and in this thesis.

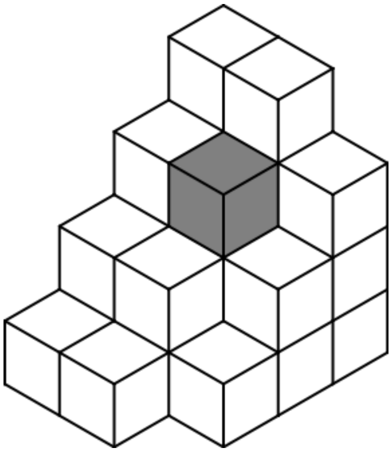


Figure 2.4: A voxel represents a value on a regular grid in three-dimensional space, useful for many applications such as Dynamic Roadmaps to create a mapping from an area in the workspace to states in the c-space.

2.3.5.1 Dynamic Roadmaps

Dynamic Roadmaps (DRM) [98, 99, 76, 144, 107, 162, 91, 31] are PRM-based approaches that heavily use preprocessing to improve handling of unknown and dynamic (moving) obstacles. Unlike with a typical PRM, DRMs can handle changing environments so "cannot exploit the premise that planning will occur many times in the same environment" [99]. DRMs are computed without *any* obstacles in the roadmap: only invariant constraints such as self-collision and stability are encoded in the roadmap. This allows the roadmap to be fully reusable in any environment but also requires that every configuration in the roadmap be checked for collision during online planning. To speed up this expensive step, DRMs use simple but clever *mappings* from the Cartesian workspace to the joint c-space of the robot.

The idea of using a mapping between a workspace and a c-space is motivated by evidence that humans similarly maintain egocentric spatial relationships between sensory signals and motor commands [150]. In [99] this mapping is encoded efficiently using compression schemes that exploit redundancy. However, [99] admits that lookup is faster without encoding so [76] does away with the complicated compression methods all together.

The mapping is achieved by reducing the 3D collision environment workspace into a uniform rectangular decomposition of voxels, similar to classic grid search methods, as shown in Figure 2.4. Each voxel maps to a subset of the robot's c-space that is invalidated when that voxel is occupied, e.g. when it contains a collision object. For each planning query, before the roadmap is searched for a path, all currently occupied voxels are used to invalidate regions of the robot's roadmap that are in collision.

DRMs are similar to E-Graphs in that they store multiple experiences in a PRM, and they are similar to LazyPRM in that they delay collision checking until after the graph is built.

The DRM method "has not yet been widely adopted" [144] because performance is highly dependent on the size of the roadmap and on the workspace size and resolution. To address this, practical implementations details have been further discussed in [76, 107, 162, 91, 31, 144]. In [76] DRMs are applied to 3D workspaces and fixed-base humanoids, and the results are benchmarked against online planning alone. In [162] the roadmap is implemented using a non-uniform 2^m hierarchical data structure that allows multi-resolution search strategies but requires grid-based search instead of using sampling.

In [91] bottlenecks are identified, and the approach is implemented on real robots that have the reaction speed twice as fast as a human's. Their focus is on avoiding collision checking as much as possible using techniques such as voxelization. In [144], Parallel Dynamic Roadmaps (PDRMs) are presented which achieve incredible performance gains on modern graphic processing units (GPUs). PDRMs also use motion primitives from [31] to allow inherent compression techniques.

A special case of the DRM method is presented in [107] where attached collision objects are considered e.g. when a robot is holding something. They argue that to compensate for the enlarged collision area, the workspace to configuration mapping must be updated, which is a slow process. To speed it up, only nodes are mapped, while edges are ignored in the DRM-type collision checking. To account for collision checking of edges, lazy collision checking is used.

In [111] an approach similar to DRM that reuses roadmaps is especially adapted for moving obstacles with known trajectories. In their work they use multiple *critical roadmaps* that move with obstacles, and they use special calculations to determine time of collision so that they can only update plans when a critical change occurs. Similarly, in [102] local roadmaps for obstacles are stored in an experience database and quickly recalled whenever similar obstacles are found. The method composes a global roadmap based on all the obstacle roadmaps, which are updated as obstacles move.

These DRM approaches are a powerful technique to speed up collision checking large pre-computed roadmaps, but they do not address the challenge of creating the roadmaps themselves. This is one of the main contributions of this thesis, and the DRM approach can be applied to our work as well.

2.3.5.2 Sparse Roadmaps

The performance of the many dynamic roadmaps approaches that use PRMs presented above suffer from large roadmap sizes. A key requirement of fast planning times is keeping the size of roadmaps sparse. To address this, the work of [44] presents the Sparse Roadmap Spanners (SPARS) algorithm that is able to compactly represent a graph while maintaining the original optimality to within a stretch factor t. For example, if the t-stretch factor is 1.1, then the maximum length a path can be from its optimal solution is 10%. The SPARS algorithm is powerful in that it is probabilistically complete, asymptotically near-optimal, and the probability of adding new vertices to the roadmap converges to zero as new experiences are added. It is a key component of this thesis. Previous works similar to SPARS suffered from a lack of compactness [79] or the inability to only remove edges but not vertices [114]. SPARS uses graph spanners to create subgraphs where the shortest path between two vertices is no longer than stretch factor t times the shortest path on the original graph. This allows theoretical guarantees on path quality to be upheld while filtering out unnecessary vertices and edges from being added to the graph.

In order to have the asymptotic optimality guarantees as PRM within a t-stretch factor, a number of checks is required to determine which potential vertices and edges (experiences) should be added to have coverage across a robot's free space. The only configurations that are saved are those that are useful for 1) coverage, 2) connectivity, or 3) improving the quality of paths on sparse roadmap relative to the optimal paths in the c-space. Two parameters t and the sparse delta visibility radius (Δ_{sparse}) control the sparsity of the graph.

More recently, an improved version called Sparse Roadmap Spanners 2 (SPARS2) [42] relaxes the requirement for a dense graph to be maintained alongside the sparse roadmap. This greatly reduces the memory requirements of the graph, making it more practical for higher DOF c-spaces to be easily maintained in memory. A trade-off in new path insertion time and memory is made for this relaxation through a local sampling process and some bookkeeping information. Still, SPARS2 significantly reduces the size required for an experience database to cover all homotopic classes.

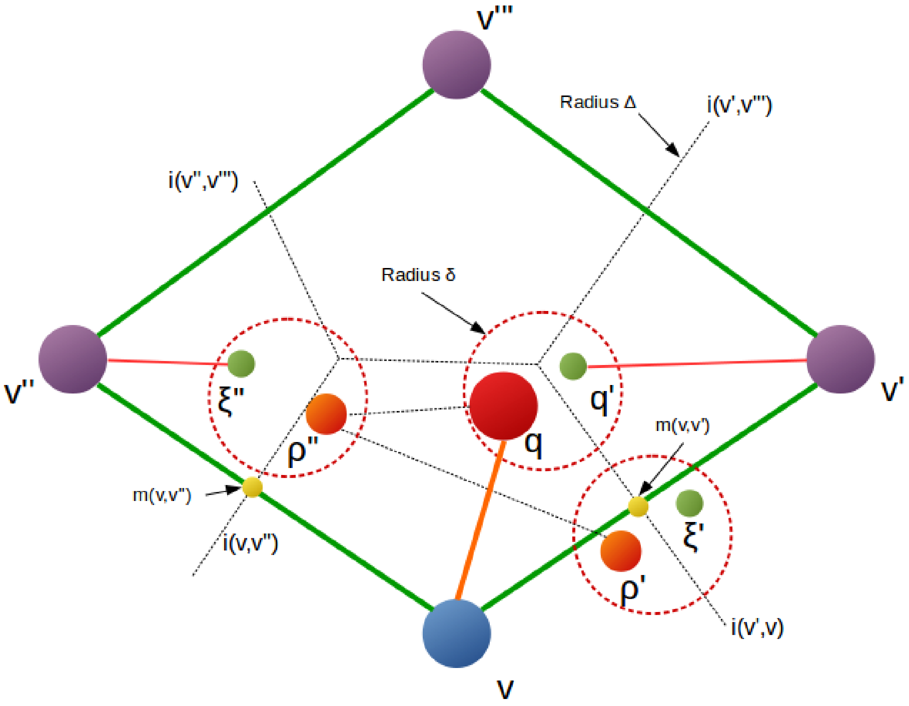


Figure 2.5: Diagram of 2D c-space explaining SPARS quality criteria: $[v, v', v'', v''']$ and the thick green lines are respectively the vertices and edges of the sparse roadmap. The black dotted lines labeled e.g. $i(v, v'')$ represent interfaces between visibility ranges Δ_{sparse} of the vertices. States ξ, ρ and q, q' represent an estimate of the interface between the vertices, bounded within the red dotted line radius δ. Midpoints $m(v, v')$ are used to calculate the worse case path distance through the graph.

The SPARS algorithm operates by uniformly sampling random states q and attempting to insert them into the sparse roadmap, criteria allowing. The algorithm terminates when M failed insertions have occurred, at which point the probabilistic estimation of the percentage of free space not covered by vertices is $\frac{1}{M}$. The sample q is tested against the following criteria:

Coverage Criterion: This check adds a vertex to the roadmap whenever no neighboring vertex within a radius Δ_{sparse} is *visible*, where visibility is defined as the existence of a collision free path using the local planner L. The local planner is typically the straight-line interpolation. This criterion is adapted from Visibility-based PRM (V-PRM) [146].

Connectivity Criterion: This check determines if any neighboring vertices within the Δ_{sparse} radius are in different *connected components* (subgraphs). If they are, q is added to the roadmap and edges to the neighboring vertices from q are created.

Interface Criterion: If q reveals the existence of an interface (boundary) between two vertices $[v_1, v_2]$ on the roadmap, an edge is added between the two vertices. This occurs when the two nearest neighbors to q are both visible to q. If no direct edge is possible between $[v_1, v_2]$, q is added and edges are used to *bridge* the connection between $[v_1, v_2]$.

Quality Criterion: Distance measurements using the midpoint between each neighboring vertex determines if the local path through the region obeys the t-stretch spanning factor to guarantee asymptotic near-optimality. If the local path does not, a new path is smoothed and added to the graph to ensure the quality criterion is upheld.

In [135] the SPARS algorithm is used to generate a sparse roadmap offline that is used to perform quick online manipulation planning using a LazyPRM-like special collision checking technique. In that work, they observed that when a path is found invalid by LazyPRM, there is a good probability that any nearby paths in the roadmap will also be in collision. However, by default LazyPRM will check all of those paths as well before moving on to planning in valid regions. To speed up planning, nearby vertices of an invalid point receive a cost bump which depends on the square distance to the invalid point. This small vertex cost adjustment heuristic for LazyPRM is an alternative to DRM-like workspace to c-space mappings.

2.3.6 Other Experience-Based Techniques

In the following section EBMP methods that speed up the planning problem using past experience in more unique ways are presented.

2.3.6.1 Guiding Sampling Using Attractors

Attractor Guided Planners (AGP) [73] bias sampling-based motion planning to previous samples along a path that are useful for a similar problem. AGP maps a current planning query to a similar problem path based on start, goal, and obstacle similarity. Once a previous experience is identified, its trajectory's waypoints are used to bias new planning towards areas that are likely to be useful again. If previous samples fail, the experience bias is reduced so that the planner can accommodate highly dynamic and difficult environments, eventually falling back to regular sampling-based planning. A similar attractors approach discussed earlier in this chapter is [20].

2.3.6.2 Configuration Space Approximations of Constraints

Using experience to sample in highly constrained environments such as when there exists close kinematic chains, orientation constraints, or stability constraints is addressed in [158]. They generate a *Approximation Graph* that caches valid samples of the constraint manifold, and saves it for future use. During online planning, these samples are used as c-space approximations to quickly generate a roadmap using traditional sampling methods with a bias towards the approximations.

2.3.6.3 Dynamic Motion Primitives

To speed up motion planning, various forms of *Dynamic Motion Primitives* (DMP) have been used to provide larger segments or "steps" of motion that can efficiently guide a planner. In [62] these DMPs are used to provide more natural motion of a walking humanoid robot and speed up query times.

In [128] DMPs are used with task and context labels to recall the proper DMP for a given problem. The DMP in this work is a non-linear differential equation that is used to represent a trajectory. This allows the method to better account for perturbations in the recalled trajectory. The experiences are populated using learning by demonstration.

In [109] instead of recalling a planning tree, roadmap, or trajectory, they use DMPs that are dynamic equations able to guide control algorithms towards a goal. These equations can be a path, spline trajectory, or an open/closed-loop controller. Unioning these primitives together can be sufficient to cover most frequently occurring tasks in experience.

In [110] DMPs are tested on a humanoid robot and benchmarked against Planning from Scratch (PFS). From their study, DMPs are found to result in less movement variability and lower computation, but PFS generalized better to different tasks.

2.4 Underconstrained Cartesian Planning

In this section we give background on another problem in the motion planning field: *underconstrained* Cartesian planning. While typically not related to EBMP, the exact problem we tackle in part of this thesis—multi-modal dual-arm Cartesian planning—is difficult enough to warrant the need for our experience techniques, as we will show in Chapter 7.

There are many applications for underconstrained Cartesian planning, including industrial processes such as routing, grinding, or wiping down surfaces. In these applications the end effector is required to follow a series of waypoints, but each waypoint has some tolerance for the angle in which the task is performed. In industry today most of these tasks are still achieved with manual off-line programming approaches [126]. Solving these problems with two arms is even more advantageous for high-speed industrial tasks that require fast cycle time.

The problem of generating a trajectory along *fully*-constrained paths has been studied extensively [159] and is often referred to as *joint trajectory generation* [58]. However, there is little literature on approaches for following *underconstrained* paths except for an approach that iteratively solves the problem by generating a graph of redundant IK solutions (*Descartes* [47]). Combining that approach in a unified free-space planning problem for multi-armed robots is part of the novel work presented later in this thesis.

A related dual-arm Cartesian-based planning approach is presented in [29] where a graph of joint solutions are generated that obeys orientation constraints for two arms holding a serving tray. In that work, they achieve their results by reducing the dimensionality of the problem by representing it's c-space with the position of the tray, yaw orientation of the tray, and a redundant joint value for each arm. The states are connected with motion primitives and a graph is generated dynamically as it is explored.

Similar relevant work utilizes a precomputed roadmap to quickly generate a trajectory obeying Cartesian and other constraints that is then sent to a trajectory optimization planner as a seed path [127]. It returns several solution paths by exploiting the redundancy in a Cartesian goal point and smooths them in parallel.

Representing the underconstrained Cartesian path as an implicit constraint manifold and planning over that using probabilistic methods is an alternative approach that could be utilized. The CBiRRT [12] algorithm uses random sampling and projection methods to handle multiple constraints that have infinitesimal volumes in the c-space. CBiRRT2 [13] extends this approach to use chains of task space regions. Unlike our approach, it is not straight-forth to represent our multiple complex constraints as a manifold. Similar approaches such as AtlasRRT [69] have problems such as assuming the manifold is smooth everywhere and do not take into account singularities. They also have expensive Newton procedures to solve at every sample and approximate nearest neighbor search with joint space distances. Still, these CBiRRT and AtlasRRT methods for dealing with highly constrained systems could be complimentary to the contributions of thesis, using experience-based planning to compensate for the expensive computation required be these approaches.

The multi-modal aspect of our approach that we will present is similar to manipulation graphs that express connectivity among manifolds of varying c-spaces [4]. In [63] complex manipulation tasks are achieved for humanoids using multi-modal approaches. The conventional PRM planning approach was applied to multi-task manipulation planning in [147].

2.5 Engineering Challenges of Motion Planning Frameworks

In this section we provide background on the complexity and usability of software engineering for the underlying framework, MoveIt!, used for the research in this thesis. We will also present other Motion Planning Frameworks predating MoveIt!.

Much work has addressed the software engineering challenges of complex robotic frameworks, but typically the identified design goals have emphasized the need for features such as platform independence, scalability, real-time performance, software reuse, and distributed layouts [93, 35, 87]. In [87]'s survey of nine open source robotic development environments, a collection of metrics was used which included documentation and GUIs, but no mention was made of components important to us: setup time, barrier to entry, or automated configuration.

A focus on component-based design of motion planning libraries similar to MoveIt! was addressed in [21]. The challenges of software reuse, combining various algorithms, and customizations are discussed, but the work falls short of addressing the initial ease of use of these robotics frameworks. The difficulty of creating good component abstractions between hardware and algorithms is addressed in [82].

The importance of an open source robotics framework having a large number of researchers and engineers motivated to contribute code and documentation is emphasized in the OROCOS framework [22].

Human-robot interaction (HRI) has also been a popular area of research, but HRI's focus has been on the runtime behavior of robots and not on the difficulties of human users applying software frameworks to robot hardware [151, 164, 57]. For example, in [75], an effective user interface is presented for teleoperation of rescue robots, but no thought is given to making it robot agnostic or to its configuration.

In [27] a set of tools was presented that allowed the Arm Navigation software framework (the precursor to MoveIt!) to be easily configured within a short amount of time for a new robotic system. Chapter 3 extends and improves that work, focusing specifically on the difficulties of setting up and configuring robotics software.

2.5.1 Existing Motion Planning Software

Many open source software projects for motion planning exist whose intent is to provide a platform for testing and developing novel path planning algorithms and other motion planning components. We will distinguish them from a motion planning *framework* due to their exclusion of actual hardware perception and control. All offer varying degrees of modularity, and all have a basic visualization window for viewing motion plans of 3D geometries. A brief review of them is presented here.

Both LaValle's Motion Strategy Library (MSL) [95], 2000, and Latombe's Motion Planning Kit (MPK) [145], 2003, have scopes limited to only simulation and therefore are not frameworks in our definition. The MSL is configured manually using six required text files and up to fifteen optional files, depending on the planning problem. It has a GUI for tweaking parameters and controlling the visualization of plans. The MPK is able to load robots with varying geometry without recompiling code and provides a scene format that is an extension of a conventional 3D graphics toolkit. It does not have a fully interactive GUI but rather allows control only through keyboard shortcuts. Neither MSL or MPK provides assistance for setting up a new robot and has little to no documentation on this process.

The Karvaki Lab's Object-Oriented Programming System for Motion Planning (OOPSMP) [134], 2008, is a predecessor to the Open Motion Planning Library (OMPL) [157], 2010, both of which are collections of planning algorithms and components whose scope also excludes hardware execution and perception tasks. OOPSMP is XML-based for configuration, scene definitions, and robot geometry. An additional SketchUp interface provides a quick way to build environments. It has some GUIs that assist in visualization. OMPL differs in its handling of environments and robots in that it abstracts that notion into a black box and instead operates in various c-spaces. OMPL includes a benchmarking GUI called OMPLApp and a web interface that can test simple planning problems. Neither OOPSMP nor OMPL provides the high level GUIs for configuration with full robotic systems.

Diankov's OpenRave [41], 2010, is a fully featured motion planning framework with many high level capabilities, some GUIs, and the ability to connect to hardware controllers and sensors. It uses the Collada format [1], as well as its own proprietary format, to define robots and environments. Its main interface is through simple Python scripting, and it utilizes a plugin interface to provide extensibility of the framework. It too falls short of providing easy to setup tools for new robots.

Willow Garage's ROS Arm Navigation framework [27], 2010, is the predecessor of MoveIt! and provides much of the same functionality of MoveIt! and OpenRave but also includes a Setup Wizard that provides a GUI for helping new users setup arbitrary robots into the framework. It was the inspiration for the Setup Assistant described in the following chapter.

Chapter 3

Building A Reusable Motion Planning Framework

Developing a framework for experience-based motion planning (EBMP) is a complex task that requires many separate components of robotic functionality to work together. The work presented in this thesis is built on top of pre-existing inverse kinematic (IK) solvers, collision checkers, motion planners, visualizers, communication middle ware, among many other components. In this chapter we present our takeaways in helping develop the popular motion planning framework MoveIt! [38] and robot software in general. We will present best practice principles for lowering the barrier to entry to robotic software using MoveIt! as our case study.

This thesis author developed one of the cornerstone's of MoveIt! usability: the MoveIt! Setup Assistant. He also has been a long time code contributor and documentation writer to many aspect of the project. The core project was developed at Willow Garage by Sachin Chitta, Ioan Sucan, and many others.

3.1 Overview

Managing the increasing complexity of modern robotic software is a difficult engineering challenge roboticists face today. The size of the code bases of common open source robotic software frameworks such as ROS [137], MoveIt! [38] and OROCOS [22] continues to increase [113], and the required breadth of knowledge for understanding the deep stack of software from control drivers to high level planners is becoming more daunting. As it is often beyond the capabilities of any one user to have the necessary domain knowledge for every aspect of a robot's tool chain, it is becoming increasingly necessary to assist users in the configuration, customization, and optimization of the various software components of a reusable robotic framework.

3.1.1 User Interface Design Principles

The user interface design principles required in the emerging field of robotics software is similar to other more mature software engineering fields, and much can be learned from them. There have been many examples of software, such as computer operating systems, that have historically required many installation and configuration steps whose setup process has since improved. Still, the user interface design principles for robotics are unique in 1) the degree to which software interacts with hardware and real world environments compared to consumer-level software, 2) the large variety in complexity and scale of robotic platforms, and 3) the long-term desire to increase the autonomy of robotics systems by reducing reliance on GUIs and increasing high level robotic intelligence.

3.1.2 Barriers to Entry

The term *barriers to entry* is used in the context of robotic software engineering to refer to the time, effort, and knowledge that a new user must invest in the integration of a software component with an arbitrary robot. This can include, for example, creating a virtual model of the robot's geometry and dynamics, customizing configuration files, choosing the fastest algorithmic approach for a specific application, and finding the best parameters for various algorithms.

Powerful robotics software generally requires many varying degrees of customization and optimization for any particular robot to operate properly. Choosing the right parameters for each utilized algorithm, software component, and application typically involves expert human input using domain-specific knowledge. Many new users to a software package, particularly as robotics becomes more mainstream, will not have the breadth of knowledge to customize every aspect of the tool chain. When the knowledge of a new user is insufficient for the requirements of the software, the barriers to entry become insurmountable and the software unusable. One of the emerging requirements of robot agnostic frameworks is implementing mechanisms that will automatically setup and tune task pipelines for arbitrary robots.

Another motivation for lowering the barrier to entry of complex robotics software is the *paradox of the active user*. This paradox explains a common observation in many user studies that *users never read manuals* but start attempting to use the software immediately [26]. The user's desire to quickly accomplish a task results in their skipping the reading of any provided documentation or gaining deeper understanding of the system and instead diving right into completing their task. The *paradox* is that the users actually would save time in the long run if they learned more about the system before attempting to use it, but these studies showed that in reality people do not tend to invest time upfront into learning a new system.

Even experts in the area of the associated robotics software will become frustrated with robotics software if all initial attempts to setup and configure the framework fail, and no progress is made. Most researchers and engineers typically do not have the time or ability to completely understand the entirety of robotics software before they start using it. It is important for the user's initial experience with a piece of software to be positive to ensure its continued use.

3.1.3 Benefits of Larger User Base

The need to lower the barrier to entry is beneficial to the software itself in that it enables more users to utilize the framework. If the software framework is being sold for profit, the benefits of a larger user base are obvious. If instead the software is a free open-source project, as many successful robotic frameworks currently are [113], lowering the barrier to entry is beneficial in that it creates the *critical mass of skilled contributors* that has been shown to make open source projects successful [22]. As the number of users increases, the speed in which bugs are identified and fixed increases [142]. It is also typically hoped that development contributions to the code base increases, though this correlation is not as strong [142]. One of the key strengths of a larger community for an open source project is increased participation of users assisting with quality assurance, documentation, and support [143].

Another benefit of lowering the barrier to entry is that it allows the robotics software to become an educational tool for robotics. Not only is the software accessible for academic research and industrial applications, but graduate, undergraduate, and even primary-level students can use it to learn some of the higher level concepts of robotic applications as has been demonstrated in [36, 119, 59].

Beyond the motivation of success for an individual software project, broadening access to robotics software development increases the number of creative minds working on solving today's challenging robotics problems. Making the accessibility of robotic development more like mobile device development and web development might increase the speed of innovation in robotics similar to that experienced by phone apps and the Internet [15].

Target *users* for these robotic frameworks are engineers, scientists, students, and hobbyists with a general aptitude for software and robotics but who are not necessarily experts in either of those fields. The hope remains that human-robotic interaction for the general population in the future will be based on more natural methods and that software configuration and graphical user interfaces (GUIs) are only necessary for the robot developers themselves [164].

3.1.4 Difficulty of Robot Agnostic Software

The software engineering challenges faced in making reusable, robot agnostic code are hard and are different from those in other reusable software frameworks. In [148], Smart argues there are three main factors that make general-purpose robotic software difficult: heterogeneity of robotics, limited resources (computational or otherwise), and the high rate of hardware and software failures. The variety of different tasks and task constraints imposed upon robots is another challenge for robot agnostic software [82].

The heterogeneity of robots is of primary concern to us in this chapter—accounting for different types of actuators, sensors, and overall form factors is a difficult task. To some users a robot is a robust and precise industrial arm, to others a robot is simply a mobile base with wheels and a computer, and to others a robot is a fully anthropomorphic biped. Creating reasonable abstractions for these large amounts of variation requires many trade-offs to be made that almost always lead to a sub-optimal solution for all robots. It is more difficult to create a hardware abstraction for a robot than a standard computer. When robotic software must interact with physical devices through an abstraction, it gives up specific knowledge of the hardware that then requires much greater reasoning and understanding of its configuration [148].

Operational requirements are another challenge in making reusable software for a wide range of robotic platforms. Some users require hard real-time constraints, while others can tolerate "fast enough" or "best effort" levels of performance. Variable amounts of available computational resources such as processing power or the "embeddedness" of the system also makes it difficult to design robot agnostic code that can run sufficiently on all robots. The amount of required error checking and fault tolerance varies by application area, for example, there are significant differences between a university research robot and a space exploration rover or a surgical robot.

In making robotic agnostic software, many time-saving shortcuts employed for single-robot software must be avoided. This includes hard coding domain-specific values for "tweaking" performance and using short-cutting heuristics applicable to only one hardware configuration. Instead, reasonable default values or automatically optimized parameters must be provided as discussed later.

On top of these challenges, packaging reusable software into an easy to setup experience for end users requires creating tools that automate the configuration of the software.

3.2 Motion Planning Frameworks

The software development of a motion planning framework (MPF) is challenging and involves combining many disparate fields of robotics and software engineering [129]. We refer to the software as a *framework* in this context because it abstracts the various components of motion planning into generic interfaces as discussed later.

One of the most important features of a MPF is providing the structures and classes to share common data between the different components. These basic data structures include a model of

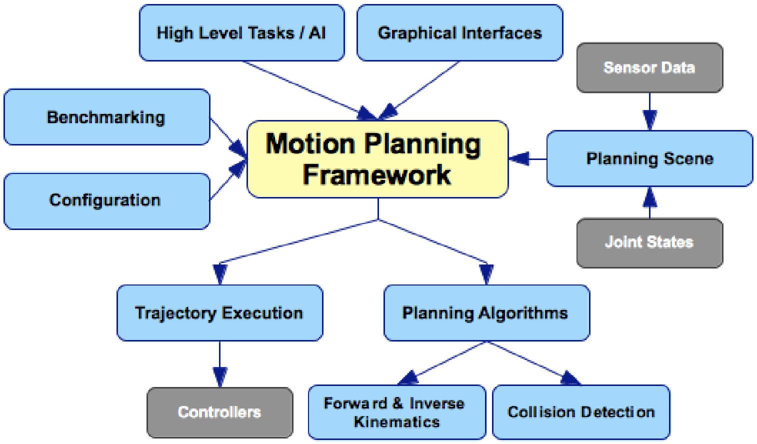


Figure 3.1: High-level diagram of various planning components (blue boxes) in a Motion Planning Framework (MPF). Gray boxes represent external input and output.

the robot, a method for maintaining a representation of the state of the robot during planning and execution, and a method for maintaining the environment as perceived by the robot's sensors (the "planning scene").

In addition to the common data structures, a MPF requires many different interacting software components, henceforth referred to as the *planning components*. A high level diagram of the various planning components is shown in Figure 3.1. The planning component that actually performs motion planning includes one or more algorithms suited for solving the expected problems a robot will encounter. The field of motion planning is large, and no one-size-fits-all solution exists yet, so a framework that is robot agnostic should likely include an assortment of algorithms and algorithm variants.

Other planning components include a collision checking module that detects the potential intersection of geometric primitives and meshes in the planning scene and robot model. A forward kinematics solver is required to propagate the robot's geometry based on its joint positions, and an IK solver is required when planning in the Cartesian space of the end effector for some planning techniques. Other potential constraints, such as joint/velocity/torque limits and stability requirements, require additional components.

Secondary components must also be integrated into a powerful MPF. Depending on what configuration space (c-space) a problem is solved in, the generated motion planning solution of position waypoints must be parameterized into a time-variant trajectory to be executed. A controller manager must decide the proper low level controllers for the necessary joints for each trajectory. A perception interface updates the planning scene with recognized objects from a perception pipeline as well as optional raw sensor data.

Higher level applications are built on top of these motion planning components to coordinate more complex tasks, such as pick and place routines. Other optional components of a MPF can include benchmarking tools, introspection and debugging tools, as well as the user-facing GUI.

3.2.1 MoveIt! Motion Planning Framework

MoveIt![38] is the primary software framework for motion planning and mobile manipulation in ROS and has been successfully integrated with many robots including the PR2 [163], Robonaut [6], and DARPA's Atlas robot. MoveIt! is written entirely in C++ but also includes Python bindings for higher level scripting. It follows the principle of software reuse as advocated for robotics in [113] of not tying itself exclusively to one robotic framework—in its case ROS—by creating a formal separation between core functionality and robotic framework-dependent aspects (e.g., communication between components).

MoveIt! uses by default the core ROS build and messaging systems. To be able to easily swap components, MoveIt! uses plugins for most of its functionality: motion planning plugins (default: Open Motion Planning Library (OMPL)), collision detection (default: Fast Collision Library (FCL) [124]), kinematics plugins (default: OROCOS Kinematics and Dynamics Library (KDL) [149] for forward and inverse kinematics for generic arms as well as custom plugins). The ability to change these default planning components is discussed in Section 3.4.3. MoveIt!'s target application is manipulation (and mobile manipulation) in industrial, commercial and research environments. For a more detailed description of MoveIt!, the interested reader is referred to [38].

3.3 Entry Barrier Design Principles

In designing the configuration process that enables MoveIt! to work with many different types of robots with almost any combination of planning components, several contending design principles for lowering the barrier to entry emerged. These requirements were drawn partially from standard HCI principles [54], from work on MoveIt!'s predecessor, and from an iterative design process where feedback was gained from internal users at Willow Garage during development. We believe these *entry barrier design principles* transcend motion planning and can be applied to most robotic software:

Immediate: The amount of time required to accomplish the most primitive task expected from the robotic software component should be minimized. This is similar to the time-honored "Hello World" demo frequently used by programming languages and typical Quick Start guides in documentation. Immediacy is essential for the *paradox of the active user* as it provides cursory feedback to the user that the software works and is worth investing further time.

Transparent: The configuration steps being performed automatically for the user and the underlying mechanisms utilized in the software components should be as visible as possible. For example, transparency is important so that users can later understand what parameters — such as in motion planning *configuration space search radius* — are specific to their robot and know how to customize the aspects they desire. A "layered" approach to presenting information can offer a good balance of separating the required knowledge for a user's immediate goals from the "useful later" information needed to prevent the user from being hindered in the future.

Intuitive: The need to read accompanied documentation and the amount of required documentation should be minimized. A well-designed user interface, be it graphical or command line, should be as intuitive as possible by following standard design patterns and providing interface context clues. An ideal GUI for configuration would not require any documentation for most users.

Reconfigurable: The automatically generated parameters and default values for the initial setup of a robot should be natural for the user to modify at a later time. Typically, these parameters and values are chosen to work for the largest number of robots possible but are not optimal for any particular robot. Providing easy methods to reconfigure the initial setup is important for allowing better performance.

Extensible: The user should be enabled to customize as many components and behaviors as possible within the reasonable scope of the software. Providing the means to extend the software with custom solutions for a particular application makes the software far more powerful and reusable for varying use-cases. A typical solution for this is providing a plugin interface.

Documented: The amount and organization of reference material explaining how to use the software should be maximized for as many aspects and user levels as possible. Even the most intuitive software requires documentation for various aspects of the operation or modification of the software itself. Different types of documentation are needed for different users—for example developers and end users—though in robotics these groups are frequently the same. Documentation is arguably the most important factor in reducing the barrier to entry of new software [52].

These principles are additionally applicable to computer software in general, but a greater focus on hardware variance and the needs of developers has been applied. *Reconfigurability*, or personalization, is common in computer software as well, but in our application we use it mainly in reference to parameters that require customization for different physical geometries and hardware designs. Similarly, *extensibility* and *transparentness* are design principles that aid robotic developers in applying their software to specific hardware. Whereas today most computer hardware is fairly standardized and shares similar capabilities, robotics hardware still has large variability in design and capability. For this reason, *transparency* is particularly important since many robotic researchers and developers need to understand the software enough to adapt it to their unique hardware.

Many of these *entry barrier design principles* have opposing objectives that require a balance to be found between them. For example, the desire for *transparency* in the underlying mechanisms often leads to slower setup times (lack of *immediacy*) and more complicated configuration steps (lack of *intuitiveness*). The need for extensibility of various components in the software often results in far more complicated software design as more abstraction is required, resulting in a less *intuitive* code base and difficult *documentation*. Nevertheless, compromises can be made between these principles that result in a superior user experience, as will be demonstrated in the next section.

3.4 Methods to Lower The Entry Barrier

One of the unique features of MoveIt! is the ratio of its power and features to the required setup time. Beginners to motion planning can take a model of their robot and with little effort execute motion plans in a virtual environment. With a few additional steps of setting up the correct hardware interfaces, one can then execute the motion plans on actual robotic hardware.

The *entry barrier design principles* discussed above were applied to MoveIt! to address the challenges faced for new users to this complex software framework. Developing these solutions required difficult software challenges to be overcome as discussed in the following case study.

3.4.1 Basic Motion Planning Out of the Box

To address the entry barrier design principle of *immediacy*, a streamlined “Quick Start” for MoveIt! was created that consists of a series of fairly trivial steps, relative to our target users. The most challenging of these steps—creating a *robot model*—is not directly related to the configuration of MoveIt! but rather is a prerequisite of using the software framework. Nevertheless, we will discuss this important prerequisite before proceeding to the more directly-related configuration steps.

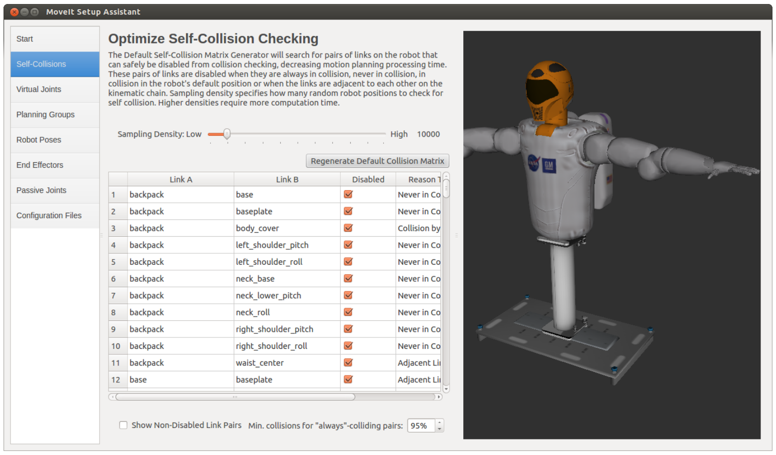
Robot Model Format: The robot model is the data structures and accompanying file format used to describe the three-dimensional geometric representation of a robot, its kinematics, as well as other properties relevant to robotics. These other properties can include the geometric visualization meshes, courser-grained collision geometry of the robot used for fast collision checking, joint limits, sensors, and dynamic properties such as mass, moments of inertia, and velocity limits. Often the robot’s joint and link relationships are represented by a kinematic tree, though this approach is problematic when a robot has a closed chain. In our application, as well as most state of the art MPFs, we will restrict our definition of modeled robots to arbitrarily articulated rigid bodies.

Extensible robotics software requires using a standardized format that can express the intricacies of varying hardware configurations. An additional design requirement for this standardized format is that it is *intuitive* for users to setup. There are a few options for representing robots, and in MoveIt! it was accomplished by using the Unified Robotic Description Format (URDF [55]) Document Object Model. This data structure is populated by reading human-readable (*transparent*) XML schemas, both URDF-formatted files (different from the datastructure), as well as the industry standard Collada [1] format.

Creating an accurate model of a robot can be a difficult task. URDF models for many robots already exist, so often users can avoid this problem. However, when a custom robot requires a new robot model, the URDF model in ROS was found to be the most appropriate to use since the user can also take advantage of tools in ROS for working with the URDF. In particular, there are tools for verifying the validity of the XML, for visualizing it, and for converting a SolidWorks CAD model of a robot directly into URDF format.

MoveIt! Setup Assistant: The main facility that provides out of the box support for beginners is the MoveIt! Setup Assistant (SA). The SA is a GUI that steps new users though the initial configuration requirements of using a custom robot with the motion planning framework (Figure 3.2). It accomplishes the objective of *immediacy* for the user by automatically generating the many configuration files necessary for the initial operation of MoveIt!. These configurations include a self-collision matrix, planning group definitions, robot poses, end effector semantics, virtual joints list, and passive joints list.

The GUI consists of 1) a large navigation pane on the left that allows the user to move back and forth through the setup process as needed (providing quick *reconfigurability*), 2) the middle settings window that changes based on the current setup step being performed by the user, and 3) a right side visualization of the three-dimensional model of the robot as it is being configured. The right side visualization increases the *immediacy* of results and *transparency* of the configuration by highlighting various links of the robot during configuration to visually confirm the actions of the



Optimize Self-Collision Checking

The Default Self-Collision Matrix Generator will search for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. Sampling density specifies how many random robot positions to check for self collision. Higher densities require more computation time.

Sampling Density: Low High 10000

	Link A	Link B	Disabled	Reason T
1	backpack	base	☑	Never in Co
2	backpack	baseplate	☑	Never in Co
3	backpack	body_cover	☑	Collision by
4	backpack	left_shoulder_pitch	☑	Never in Co
5	backpack	left_shoulder_roll	☑	Never in Co
6	backpack	neck_base	☑	Never in Co
7	backpack	neck_lower_pitch	☑	Never in Co
8	backpack	neck_roll	☑	Never in Co
9	backpack	right_shoulder_pitch	☑	Never in Co
10	backpack	right_shoulder_roll	☑	Never in Co
11	backpack	waist_center	☑	Adjacent Li
12	base	baseplate	☑	Adjacent Li

□ Show Non-Disabled Link Pairs Min. collisions for "always"-colliding pairs: 95%

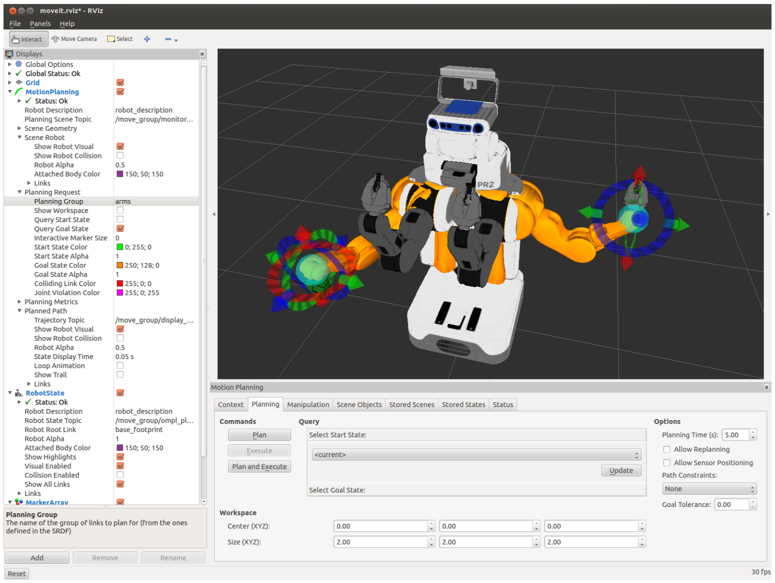
Figure 3.2: MoveIt! Setup Assistant GUI with the NASA Robonaut loaded on the self-collision matrix screen.

user.

Using a properly formatted robot model file with the SA, MoveIt! can automatically accomplish many of the required tasks in a MPF. If one desired, the steps within the SA could almost entirely be automated themselves, but they have been kept manual to 1) increase *transparency* and 2) provide *extensibility* for edge cases and unusual customizations. For example, automated semantical guesses of where an arm ends and an end effector begins can sometimes be incorrect.

MoveIt! Motion Planning Visualization GUI: The details of the automated configuration are left for the next section, but after the steps in the SA are completed, a demo script is created that automatically starts up a visualization tool with the new robot loaded and ready to run motion planning algorithms in a non-physics based simulation. A typical demo task would be using the computer mouse to visually drag 3D interactive arrows situated on the robot's end effector from a start position to a goal position around some virtual obstacle. The demo can then quickly plan the arm in a collision-free path around the obstacle and visualize the results within the GUI.

This user interaction is accomplished with the MoveIt! Motion Planning Visualization (MMPV) [38], an additional GUI that allows beginning users to learn and experiment with a large subset of the functionality provided by MoveIt! (Figure 3.3). While the long-term goal of



Global Options	
✓ Global Status: Ok	
Grid	☑
MotionPlanning	☑
✓ Status: Ok	
Robot Description	robot_description
Planning Scene Topic	/move_group/monitor...
Scene Geometry	
Scene Robot	
Show Robot Visual	☑
Show Robot Collision	☐
Robot Alpha	0.5
Attached Body Color	150; 50; 150
Links	
Planning Request	
Planning Group	arms
Show Workspace	☐
Query Start State	☐
Query Goal State	☑
Interactive Marker Size	0
Start State Color	0; 255; 0
Start State Alpha	1
Goal State Color	250; 128; 0
Goal State Alpha	1
Colliding Link Color	255; 0; 0
Joint Violation Color	255; 0; 255
Planning Metrics	
Planned Path	
Trajectory Topic	/move_group/display_...
Show Robot Visual	☑
Show Robot Collision	☐
Robot Alpha	0.5
State Display Time	0.05 s
Loop Animation	☐
Show Trail	☐
Links	
RobotState	☑
✓ Status: Ok	
Robot Description	robot_description
Robot State Topic	/move_group/ompl_pl...
Robot Root Link	base_footprint
Robot Alpha	1
Attached Body Color	150; 50; 150
Show Highlights	☑
Visual Enabled	☑
Collision Enabled	☐
Show All Links	☑
Links	
MarkerArray	☑

Planning Group
The name of the group of links to plan for (from the ones defined in the SRDF)

Select Start State: <current>

Select Goal State:

Workspace

Center (XYZ):	0.00	0.00	0.00
Size (XYZ):	2.00	2.00	2.00

Options

Planning Time (s): 5.00
☐ Allow Replanning
☐ Allow Sensor Positioning
Path Constraints: None
Goal Tolerance: 0.00

Figure 3.3: MoveIt! Motion Planning Visualization GUI with the PR2 planning with both arms to goal positions with interactive mouse-based tools

robotics is to provide more autonomous solutions to motion planning and human-robot interactions [164], the MMPV fulfills the immediate needs of direct operation for easily testing and debugging the framework's capabilities. This interface is a vital component of MoveIt!'s strategy to provide *immediate* results for motion planning with a robot that does not require any custom coding. Once the user is comfortable with the basic feature set and functionality of MoveIt!, *extensibility* is provided via varying levels of code APIs for more direct, non-GUI, access to the robot's abilities.

The MMPV provides a large number of features and visual tools for motion planning. Using the MMPV, visualizations are provided of: 1) Start and goal configurations of the robot for planning, 2) Current robot hardware configuration, 3) Animated planned path before execution, 4) Detected collisions, 5) Sensor data and recognized objects, 6) Pick and place data such as grasp positions, 7) Attached bodies such as manipulated objects, and 8) Planning metrics.

Additionally, the MMPV contains many other non-visualization tools such as: 1) Connecting to a database of planning scenes, 2) Adjusting IK settings, 3) Changing the utilized planning algorithm, 4) Adjusting the workspace size, 5) Adjusting goal tolerance and planning time, 6) Tweaking manipulation plans, 7) Loading and moving collision objects, 8) Exporting/importing scenes and states, and 9) Viewing the status of MoveIt!.

Hardware Configuration and Execution: Once the user is comfortable with the basic tools and features provided by MoveIt!, the next step is to configure their robot's actual hardware actuators and control interfaces to accept trajectory commands from MoveIt!. This step is not as easy and requires some custom coding to account for the specifics of the robot hardware—the communication bus, real-time requirements, and controller implementations. At the abstract level, all MoveIt! requires is that the robot hardware exposes its joint positions and accepts a standard ROS trajectory message containing a discretized set of time-variant waypoints including desired positions, velocities, and accelerations.

3.4.2 Automatic Configuration and Optimization

The size and complexity of a feature-rich MPF like MoveIt! requires many parameters and configurations of the software be automatically setup and tuned to improve the MPF's *immediacy*. MoveIt! accomplishes this in the 1) setup phase of a new robot, using the Setup Assistant, 2) during the runtime of the application, and 3) using benchmarking and parameter sweeping [30].

Self-Collision Matrix: The first step of the SA is the generation of a self-collision matrix for the robot that is used in all future planning to speed up collision checking. This collision matrix encodes pairs of links on a robot that never need to be checked for self-collision due to the kinematic infeasibility of there actually being a collision. Reasons for disabled collision checking between two links includes:

- Links that can never intersect due to the reachability kinematics of the robot
- Adjacent links that are connected and so are by design in collision
- Links that are always in collision for any other reason, including inaccuracies in the robot model and precision errors

This self-collision matrix is generated by running the robot through tens of thousands of random joint configurations and recording statistics of each link pair's collision frequency. The algorithm then creates a list of link pairs that has been determined to never need to be collision checked. This reduces future motion planning runtimes because it reduces the number of required collision checks for every motion planning problem. The algorithm is incomplete because in probabilistically rare cases a pair of links will be disabled for collision checking when it should not be. For this reason, the number of tests needs to be very high.

Configuration Files: The other six steps of the SA all provide graphical front ends for the data required to populate the Semantic Robotic Description Format (SRDF) and other configuration files used by MoveIt!. The SRDF provides *reconfigurable* semantic metadata of the robot model. It is data useful to motion planning but not relevant to the URDF because it does not describe physical properties of the robot. The SRDF information includes which set of joints constitutes an arm and which set of links is considered part of the end effector. It is one of the main components that allows MoveIt! to be robot agnostic and to avoid dependencies on specific robots [38]. Requiring the user to configure all the semantic information by hand in a text editor would be tedious and more difficult than using a GUI. The GUI populates the available options for each input field in list boxes and guides the user through filling in the necessary fields with buttons and graphical feedback.

The last step of the SA is to generate all launch scripts and configuration files. This step outputs to file the information collected from the user during the step-by-step user interface, as well as generates a series of default configuration and launch scripts that are automatically customized for the particular robot using the URDF and SRDF information. These defaults include velocity and acceleration limits for each joint, kinematic solvers for each planning group, available planning algorithms, and projection evaluators for planning. Default planning adapters are setup for pre- and post-processing of motion plans. Default benchmarking setups, controller and sensor manager scripts, and empty object databases are all generated using launch scripts, which essentially allow one to start different sets of MoveIt! functionality that are already put together.

These configuration files easily can be modified later from their default values by simply editing the text-based configuration files. The format of the files is based on ROS standards, which were chosen for their widespread acceptance, readability, and simplicity. For the launch files, an XML-based format custom to launching ROS applications was utilized. For all other configuration files, the open source YAML data serialization format was used.

Automatic Runtime Tuning: MoveIt! is designed to simplify solving planning problems by reducing the number of hard-coded parameters and so called "magic numbers." Sampling-based planning algorithms in particular require such parameters as input. MoveIt! uses heuristics from OMPL to automatically choose good values for certain parameters to reduce the amount of expert domain knowledge required and make MoveIt! extensible to a larger set of problems.

An example of automatic runtime tuning is the resolution at which collision checking is performed; it is defined as a fraction of the space extent. The space extent is the lowest upper bound for the distance between the two farthest configurations of the robot. This distance depends on the joint limits, the types of joints, and the planning groups being used. Using the same information, projections to Euclidean spaces also can be defined. These projections are used to estimate coverage during planning. For example, the projections for robot arms are orthogonal ones, using the joints closer to the shoulder, as those most influence the position of the end-effector.

Benchmarking: For applications that require more tuning and optimization than those afforded by automatically generated parameters and default values, MoveIt! provides the ability to configure and switch out different planning components and and specify their configuration. However, this capability is much less useful without the ability to quantify the results of different approaches. Optimization criteria such as path length, planning time, smoothness, distance to nearest obstacle, and energy minimization need benchmarking tools to enable users and developers to find the best set of parameters and planning components for any given robotic application.

MoveIt! lowers the barrier to entry to benchmarking by providing a command line-based infrastructure and benchmarking configuration files that allows each benchmark to easily be set up for comparison against other algorithms and parameters [30, 120].

Choosing the best combination of planning components and parameters for any particular robot and problem is a daunting task even for experts due to the number of choices that must

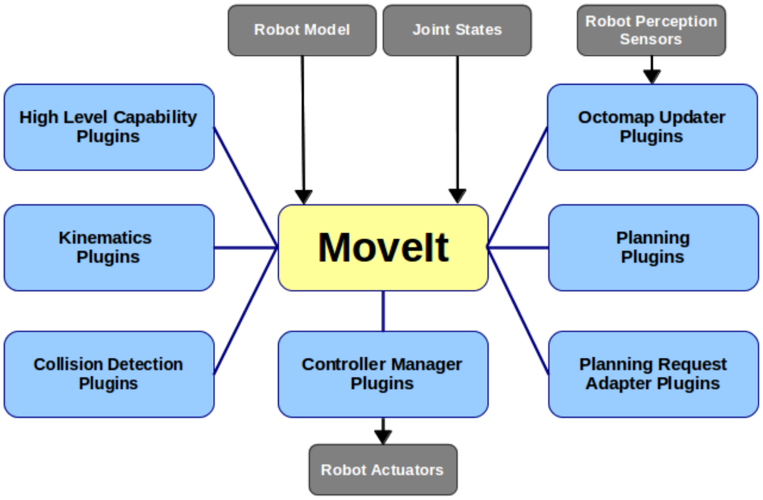


Figure 3.4: Available planning component plugins for easily extending the functionality of MoveIt!. Gray boxes represent external input and output.

be made [30]. A conventional method to optimize an algorithms performance is to perform single and multivariable parameter sweeps during benchmarking. MoveIt! provides an interface for this in its benchmarking infrastructure by allowing an upper, lower, and increment search values to be provided by the user. Results can be output into generic formats for use in different plotting tools for analysis of which combination of parameters performed the best.

Attempting to fine-tune the functionality of MoveIt! with benchmarking and parameter sweeping is a feature for expert users, and it is generally not required for entry-level users.

3.4.3 Easily Customize Framework Components

MoveIt! lowers the barrier to entry by not requiring users to provide their own implementation of any of the components in the motion planning framework. The default planning components are based on OMPL, FCL, and KDL. However, these default components are limiting to more advanced users who have their own application or research-specific needs to fulfill. We briefly describe here how MoveIt! uses a plugin-based architecture and a high-level interface to address these *extensibility* issues (interested readers should refer to [38] for more detailed explanations).

Plugins: MoveIt! is designed to be *extensible* by allowing its various planning components to be customized through a lightweight plugin interface [38]. This is accomplished by using C++ shared objects that are loaded at run time, reducing dependency complexities. This plugin-centric framework, as seen in Figure 3.4, provides interfaces for forward and IKs, collision detection, planning, planning request adapters, controllers, perception, and higher level capabilities. Almost all aspects of MoveIt!'s functionality can be extended using plugins.

A particular strongpoint of MoveIt!'s feature set is its kinematics plugins that it can automatically generate using the input URDF. The default KDL plugin uses numerical techniques to convert from a Cartesian space to joint c-space. A faster solution can be achieved for some robots by utilizing OpenRave's IKFast [40] plugin that analytically solves the IK problem. A combination of MoveIt! scripts and the IKFast Robot Kinematics Compiler can automatically generate the C++ code and plugin needed to increase the speed of motion planning solutions by up to three orders of magnitude [40].

Essentially, MoveIt! provides a set of data sharing and synchronization tools, sharing between all planning components the robot's model and state. The *extensibility* of MoveIt!'s framework is greatly enhanced by not forcing users to use any particular algorithmic approach.

High Level Interfaces: High level custom task scripting is easily accomplished in MoveIt! with both a C++ and Python interface that abstracts away most of the underlying mechanisms in MoveIt!. Users who do not wish to concern themselves with how the various low level planning components are operating can focus instead on the high level application tasks, such as picking up an object and manipulating it. Python in particular is a very easy scripting language that enables powerful motion planning tasks to be accomplished with little effort.

3.4.3.1 Documentation and Community

Though commonplace in open source software projects [22], it should be mentioned for completeness that MoveIt! addressed the *entry barrier design principle* of *documentation* by providing extensive online wiki pages, a mailing list for questions, and an issue tracker for bug reports and

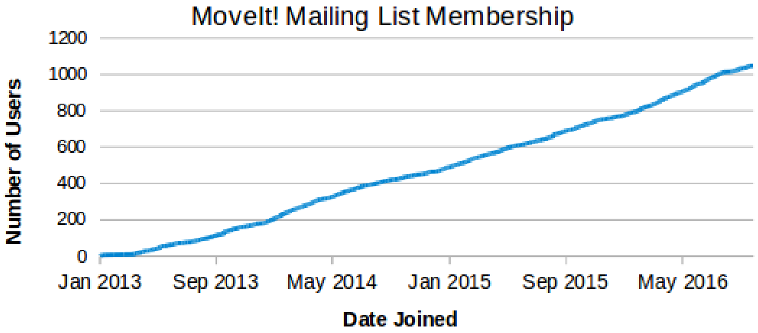


Figure 3.5: MoveIt! mailing list statistics though November 2016. MoveIt! was alpha released in May 2013

feature requests.

3.5 Community Impact

The success of MoveIt!'s efforts to lower its barrier to entry to new users through the application of the barrier to entry principles is quantified in the following. Its adoption rate, community activity, contributors, and results from a user survey are used as indicators of its progress.

3.5.0.1 Statistics

MoveIt! was officially alpha released on May 6th, 2013—about 3 and a half years prior to this writing. One method to quantify its popularity is the total number of binary and source code installations that have been performed. Though not exactly representative of this data, MoveIt!'s website has an "Installation" page that receives an average of 1,300 unique page views per month [38]—a large fraction of that number can be assumed to represent unique installations.

There are currently 1062 members on the MoveIt! mailing list as shown over time in Figure 3.5.

There have been a total of 130 contributors to the MoveIt! code base since its initial development began in 2011, shown in Figure 3.6. According to statistics gathered by the website Ohloh.com, which tracks activity in open source projects, MoveIt! is "one the largest open-source

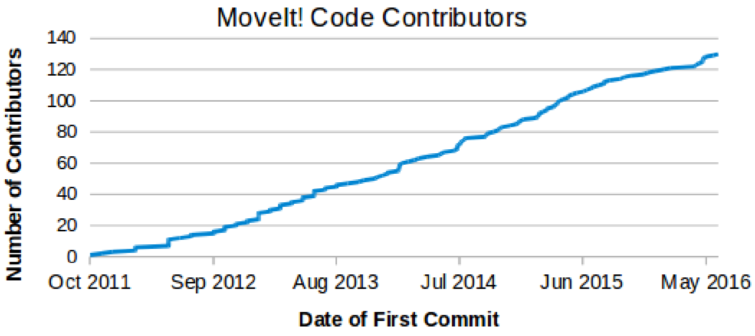


Figure 3.6: Growth of the number of involved developers (code contributors) in the MoveIt! source repositories through July 2016

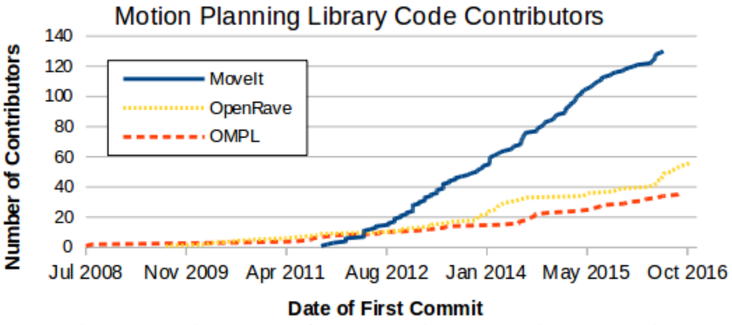


Figure 3.7: Comparison of the growth of the number of involved developers (code contributors) between MoveIt!, OMPL, and OpenRave through November 2016

teams in the world and is in the top 2% of all project teams on Ohloh" [123].

3.5.0.2 Comparison

A brief comparison with MoveIt! to OMPL and OpenRave is shown in Figure 3.7. In this diagram, the total number of code contributors is plotted with respect to time, as reported from the projects' respective version control system (VCS). No other software projects discussed in this chapter had VCSs available publicly for comparison.

3.5.0.3 Survey

A survey on users' experience with MoveIt! was administered on the MoveIt! and ROS mailing lists. There were a total of 105 respondents; graduate students represented by far the largest

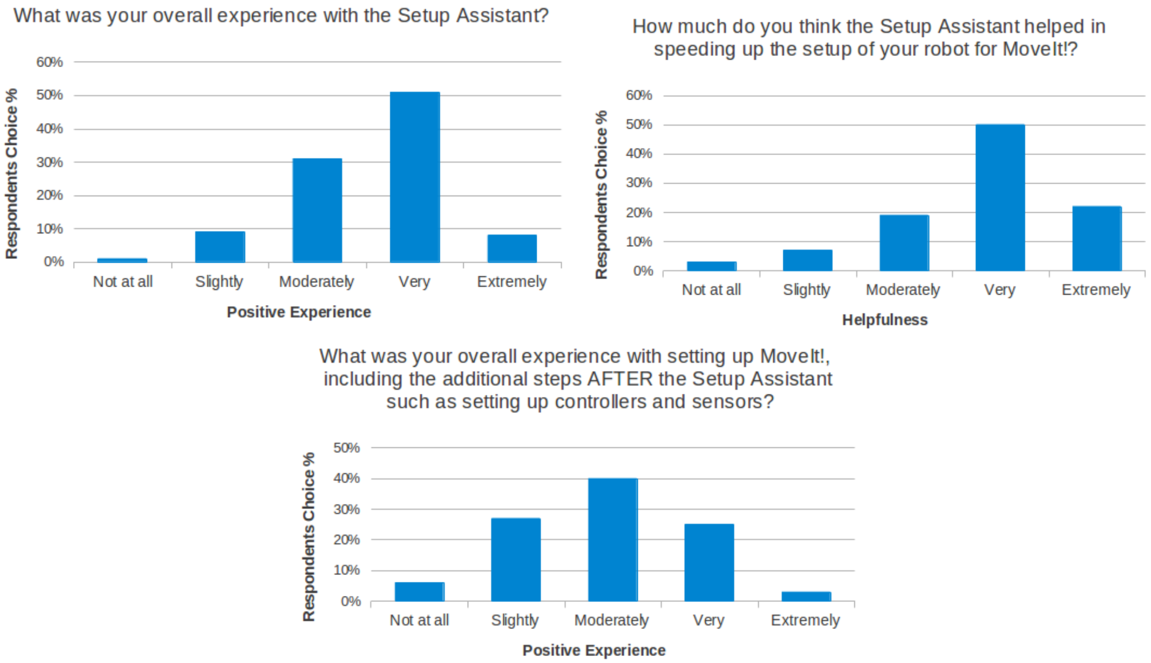


Figure 3.8: Survey data of 105 respondents on the MoveIt! and ROS mailing lists.

group of respondents (39%), while faculty/post-docs (18%) and industry R&D users (17%) represented the next biggest groups (see [38] for the full survey results). Relevant results corresponding to the use of the MoveIt! Setup Assistant are shown in Figure 3.8.

Respondents were asked to rate their overall experience with using the MoveIt! SA, and asked how much the SA helped in speeding up setup of a robot in MoveIt!. For both questions, ninety percent had “moderately” to “extremely” positive experiences with the SA, and for both questions over half rated their experience as “very good.”

Respondents then were asked what their overall experience was with setting up and configuring MoveIt!, including any additional steps they had to take after the SA, such as setting up controllers and sensors. In this question, the results were less positive. Forty percent had a “moderately” positive experience, but only 28% had a “very” or “extremely” positive experience.

An additional question asked respondents “how many minutes would you estimate you spent going from a URDF to solving motion plans using the MoveIt! Motion Planning Visualizer?” The responses to this question had a large amount of variance, with the mean time taking users 1.5

hours with a standard deviation of 2 hours.

3.6 Discussion

MoveIt! Setup: We believe the barrier to entry for MoveIt! is easier than most, if not all, open source motion planning software available today as discussed in Section 2.5.1. As a result, MoveIt! quickly has become popular in the robotics community as a powerful MPF that is extensible to most users' needs for their robot application. The adoption rate of MoveIt! since its official release three and a half years ago has been impressively positive in comparison to the size of the worldwide robotics community. With 1062 users on the mailing list since MoveIt!'s release, a new member has joined the project at a rate of nearly one per day. These numbers indicate a healthy usage and popularity of this open source software project.

Community effort to improve MoveIt! has been better than expected given the large number of code contributors during MoveIt!'s existence. The comparison of two other robotics software projectss all-time contributors in Figure 3.7 makes it quite evident that MoveIt! is a popular robotics project relative to others. Ohloh's ranking of MoveIt! as one of the largest open source teams in the world confirms our belief that by making complex software more accessible, more developers will be able to report and fix issues.

The results of the survey on MoveIt! indicated most people have found the cornerstone of our approach to lowering the barrier to entry, the Setup Assistant, to be very or extremely helpful in saving them time during setup (72% of respondents). Additionally, their overall experience was very or extremely positive with the SA (59%). However, in asking respondents their overall configuration experience with MoveIt! beyond just the SA, their ratings were lower, with only 28% saying they had a very or extremely positive experience setting up MoveIt!. This indicates that improvement can be made in the overall integration process and that adding more steps and features to the SA could reduce even further the entry barrier to MoveIt!.

From the lower results from this last survey question, it is clear that the setup and configuration process of MoveIt! still can be improved. A popular response from some of the free-form questions in the survey is that setting up the hardware controllers also can be a difficult task for non-experts, and the MoveIt! setup process does not yet document and provide example code as well as it could. It is likely this step will continue to require some custom coding to account for arbitrary hardware interfaces and communication methods, but based on the feedback we have received from actual users, this is certainly an area of improvement for the MoveIt! Setup Assistant to address.

The estimated setup time from taking a URDF and using MoveIt! to solve motion plans shows a large range of variance and likely indicates that wide range of experience levels in MoveIt! users. Although users averaged 1.5 hours to configuring MoveIt! from scratch, 31% reported it taking them 15 minutes or less to setup MoveIt! for a new robot. Creating software powerful but simple enough for all skill levels of users is a challenging task that MoveIt! will continue to tackle.

Though not exactly within the scope of MoveIt!, creating the robot model itself is a difficult task that typically requires a lot of trial and error in configuring the links and joints properly. This process could be improved by a better GUI for making arbitrary robot models, better tools for attaching the links together correctly, and more documentation.

Finally, although MoveIt! is very extensible with its plugin-based architecture, modifying the actual code base of MoveIt! can be intimidating due to its large size. MoveIt! contains over 170 thousand lines of code across all its various packages. Due to the need for computational speed and power, the layout of the code sometimes can seem complicated and abstracted.

We would like to emphasize the effect of a quick setup process and *Getting Started* demo on a new user unaccustomed to MoveIt! or motion planning in general. The positive reinforcement of a quick initial success encourages novices to continue to use the software and enables them to begin going deeper into the functionality and code base. If the entry barrier is too high, that is to say if it is too complex and error-prone, a new user will likely give up and turn to other frameworks or custom solutions. Attempting to blindly fix software that a new user has not had any success with is a daunting task.

MoveIt Development: Finding the balance between the opposing objectives of the *entry* *barrier design principles* was a difficult task in developing MoveIt!. *Immediacy* was given one of the highest priorities, such that we focused on users being able to go from robot model to planning feasible motion plans with very few steps. To allow this, the GUI streamlined the entire process and only presented the most important and *intuitive* configuration options. For example, the concept of defining the parts of a robot that make up an arm is *intuitive.* This focus on *immediacy* sacrifices *transparency* in that once users get this initial virtual demonstration working, they have not learned much on how to extend or dive deeper into the MPF. At this point *documentation* is necessary to the user.

Another pair of conflicting principles are *extensibility* and *intuitiveness.* The powerful plugin framework that MoveIt! provides allows custom components to be loaded and swapped out at runtime. However, this requires many layers of abstraction and inheritance, and results in overall convoluted code that is difficult for new developers. The balance is attempted by providing documentation and code examples for plugins that allows users to build new components without worrying about the underlying framework.

Integrating components from different sources, such as third party libraries of robotic software from other research groups, presents challenges as discussed in [21]. During MoveIt! development, the plugin interfaces were required to be general enough to work with many different implementation methods and choices of data structures. This was accomplished by providing "wrapper" packages that connect together the standard MoveIt! plugin interfaces with the third party software API. For example, MoveIt is currently setup to work with at least three planning libraries—OMPL, SBPL [103], and CHOMP [138]. Although they represent fairly different approaches to motion planning and use different datastructures, each has a wrapper component that harmonizes it to work together in MoveIt!. It should be noted that as is true with any external dependency, maintaining compatibility with these wrappers has proven challenging.

Robotic Software: The techniques utilized in lowering the barrier to entry for MoveIt! easily can be applied to robotics software in general. Almost all robotics software requires customizations specific to a particular hardware and kinematic configuration. Reducing the difficulty of performing these customizations should be the goal of robotic software engineers who desire to create useful tools for a large audience.

For example, perception applications such as visual servoing require similar kinematic models to those being used in motion planning. Frame transforms must be specified for the location of the camera and the location of the end effector with respect to the rest of the robot's geometry [67]. Often this is a difficult and tedious task. Automating the setup and calibration of these transforms lowers the barrier to entry to new users to robotic vision software and makes the vision software useful to more users. In general, automating the sequence of configuration steps necessary for performing particular tasks is a useful strategy; the users will not have to think about whether they have missed steps or whether they have performed the necessary steps in the correct order.

The *entry barrier design principles* of immediacy, transparency, intuitiveness, reconfigurability, extensibility, and documentation present a set of guidelines for other open source robotic software projects to reduce their barriers of entry to users. In fact, many existing robotics projects already follow subsets of these principles but typically to a lesser extent and fervor.

Creating a GUI such as the Setup Assistant is a time-consuming process that many robotics developers avoid in favor of hard-coded or command-line based configuration, thereby neglecting the opportunity to attract non-expert users. Between two developers, the various GUIs and configuration tools in MoveIt! took about three months of development time. We believe that the trade-off in the time invested is worthwhile for the higher adoption rates and creation of a larger community willing to contribute to the software's development.

3.7 Chapter Summary

Beyond the usual considerations in building successful robotics software, an open source project that desires to maintain an active and large user base needs to take into account the barriers of entry to new users. By making robotic software more accessible, more users have the ability to utilize and contribute to robotics development who previously could not have. The entry barrier design principles are guidelines for robotic software engineers to improve the usefulness and usability of their work to others as demonstrated in this chapter with the case study of MoveIt!.

As robotic algorithms become more complicated and the number of interacting software components and size of the code base increases, configuring an arbitrary robot to utilize robotic software becomes a daunting task requiring domain-specific expertise in a large breadth of theory and implementation. To account for this, a quick and easy initial configuration, with partially automated optimization and easily extensible components for future customization is becoming a greater necessity in motion planning and in robotic software engineering in general.

Chapter 4

Experienced-Based Planning Framework

This chapter will present our approach to experience-based motion planning (EBMP) using sparse roadmaps in a framework called *Thunder*, detail its implementation, and show experimental results comparing how our improvements perform against previous methods. Our framework is tested on a whole body humanoid in variable obstacle environments. In Chapter 8 we will update these results using takeaways from upcoming chapters in an improved and simplified version.

4.1 Method

The Thunder Framework is built on the parallel module concept of the *Lightning Framework* [11] (the inspiration for Thunder's name) presented in Section 2.3.3.4. As in Lightning, computation is split into two modules. The first module runs the Planning from Scratch (PFS), which attempts to solve the problem using a traditional sampling-based planner without prior knowledge or experience. The second module runs the Retrieve Repair (RR) planner, which uses an experience database to find parts of past solutions that are similar to the current problem and then attempts to repair them as necessary to solve the current problem. The solution from the component first to finish is returned as the overall result, while the other module is immediately terminated, as shown in Figure 4.1.

The PFS component maintains the same guarantee of probabilistic completeness as a traditional sampling-based planner [28]. It is used to initialize the empty experience database so that no human training is required, allowing a robot, which typically has different kinematics and joint

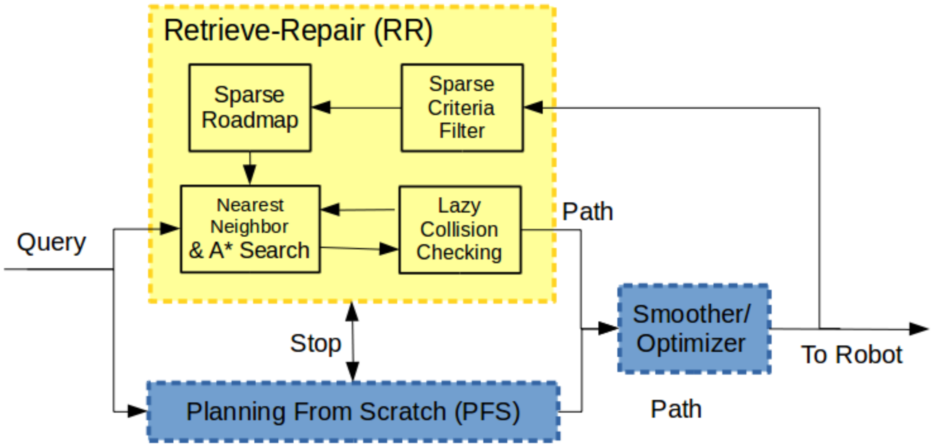


Figure 4.1: Diagram of the Thunder framework pipeline

limits than a human, to find solutions that would be difficult to specify for human trainers.

Our approach to store past experiences builds upon Sparse Roadmap Spanners (SPARS) [44] presented in Section 2.3.5.2, though we use the improved version, called Sparse Roadmap Spanners 2 (SPARS2) [42], which relaxes the requirement for a dense graph to be maintained alongside the sparse roadmap. This greatly reduces the memory requirements of the graph, making it practical for high degrees of freedom (DOF) configuration spaces (c-spaces) to be maintained in memory.

Collision checking is one of the most computationally intense components of most motion planners, and as such reducing the number of required collision checks is pivotal in increasing speed. Lazy collision checking, as demonstrated in LazyPRM [14], is a classic approach used in Thunder that delays checking until after a candidate path through a c-space is found. Once a path is found, it is checked for validity against the changing collision environment. Invalid segments are removed or marked as invalid, and search continues if necessary.

4.1.1 Recording Experiences

In the remainder of this thesis we will refine the meaning of experience database as a *graph* by using the term *experience roadmap* where appropriate. In our approach, a robot's experience

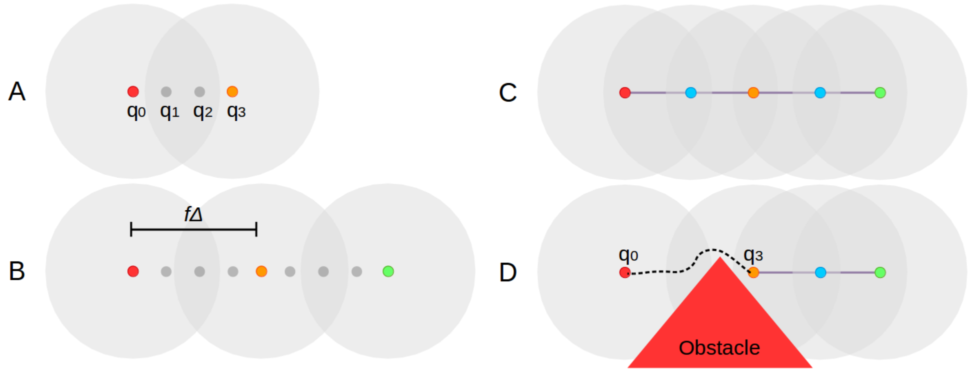


Figure 4.2: Vertex types: start (green), goal (red), *coverage* guards (orange), *connectivity* guards (blue). Gray circles indicate visibility regions and the dotted line indicates a disconnected segment. A) demonstrating in-order insertion that fails to create an edge. B) evenly spaced out guard placement. C) guards are connected with edges D) potential issue where spacing does not work out with constraints in c-space.

roadmap is initially empty. As solutions are generated from both the PFS and RR planners, they are fed back into the experience roadmap. An example of this experience roadmap for a humanoid is visualized in Figure 1.2. In the case of paths that have been generated from recall, reinsertion into the database is still beneficial because all generated paths are first randomly smoothed in a post-processing step before being sent to the robot. This stochastic element of the experience planning pipeline allows the experience roadmap to actually improve over time; there is a probability that a faster path will be found that is beyond the asymptotically near-optimal graph spanner guarantees, and new vertices and edges will be added that improve future queries.

Every solution path is discretized into an ordered set of vertices and incorporated into the experience roadmap. SPARS' theoretical guarantees decide which parts of past experiences to save. In [44] a proof is provided that SPARS decreases the rate of vertex addition over time.

In practice, care must be taken when incorporating a discretized path into an experience roadmap while maintaining the properties of SPARS. The difficulty stems from the SPARS checks that each inserted vertex must pass. The naive approach of inserting a new path into the experience roadmap linearly—inserting each vertex in order from start to goal—will almost never yield a single connected component in the graph early on.

As a simple example, take an empty graph in a c-space devoid of obstacles and a discretized candidate path as shown in Figure 4.2A. Initially a vertex q_0 is attempted to be inserted in a free space. Because there are no other nearby vertices, it is added for coverage. The next vertices, q_1 and q_2, in the ordered set of discretized path states likely still will be within the sparse delta visibility radius (Δ_{sparse}) of q_0 and will be visible to q_0 because there are no obstacles. These vertices are not added because the sparse roadmap criteria determine that q_0 can serve as their representative guard. Next, vertex q_3 will be attempted to be inserted into the graph spanner. Because it is no longer visible to q_0, it also will be added as a guard of type *coverage*. However, no edge will be created connecting coverage vertices q_0 to q_3 because edges are added only when 1) a candidate vertex is visible to two other surrounding vertices that are not already part of the same connected component, 2) when they are on an *interface* or border between the visibility regions of two guards, or 3) a series of quality checks determines that adding the edge is required to ensure the optimality guarantees of nearby guards. In this simple demonstration, a graph of two disconnected components is created that fails to bridge a connected path.

Having a disconnected path is not detrimental because future similar paths have a probability of eventually reconnecting the components with their candidate states. In the meantime, however, the retrieval component of Thunder suffers from a large number of disconnected components that do not contribute to speeding up planning. The size of a high-dimensional c-space can make the probability of two similar paths being inserted rare. Therefore, it is advantageous to ensure that a candidate path is fully inserted and connected in order to greatly increase the rate at which a Thunder experience roadmap learns and becomes useful.

To improve connectivity of inserted paths, a straight-path heuristic that orders the insertion of interpolated states into SPARS is utilized. The heuristic is built on the premise that two *coverage* guards must be within a certain distance d of each other. That distance is $\Delta \cdot f_{low} < d < \Delta \cdot f_{high}$ where the Δ_{sparse} scaling factor ranges f must be between 1.0 and 2.0 as required by SPARS' visibility criterion, shown in Figure 4.2B. The best value for f for our candidate path is chosen such that guards are evenly spaced to prevent any of the *coverage* guards from being inserted:

$$f = \frac{l}{n \cdot \Delta} \qquad n = \lfloor \frac{l}{\Delta \cdot f_{low}} \rfloor$$

where l is the total length of the path, n is the number of guards, and the second relation is rounded down. Using this result, a guard is inserted every $f \cdot \Delta$ distance away from the last guard and has an even coverage of the path as shown in Figure 4.2B. To then connect those *coverage* guards with edges, new vertices must be inserted in-between each pair of *coverage* guards to create the necessary *connectivity* guards. For a straight line path with no obstacles, a graph like the one shown in Figure 4.2C will emerge.

In the aforementioned insertion method, it was assumed that the path was straight, and there were no obstacles in the environment. However, adversarial cases—such as shown in Figure 4.2D when a path has a small curve (dotted line) around an obstacle—can cause a path to become disconnected. The utilized f in this example is not able to capture this curve because of the chosen resolution, and no edge will connect q_0 to q_3 because it lacks visibility. Similar scenarios exist even without obstacles when paths are convoluted, such as a zig-zag shape.

To mitigate these edge cases, all remaining path states that were not already attempted to be added to the sparse roadmap are added in random order. While still no guarantees exist for full connectivity, the probability is much higher since an informed insertion heuristic already has achieved most of the necessary connections. We will improve this insertion method in Chapter 8.

The path discretization resolution was set as the same used in collision checking, which results in many unnecessary vertex insertion attempts but greatly improves the chances of a fully connected path being added to the database. In practice, a path that is attempted to be inserted into a sparse roadmap achieves start-goal connectivity about 97% of the time.

4.1.2 Retrieving and Repairing Experiences

Finding a valid experience in the Thunder experience roadmap is similar to the standard Probabilistic Roadmap (PRM) and SPARS approach of using A* to search for the optimal path in the experience roadmap. The main difference is that lazy collision checking is utilized so that the framework can handle changing collision environments.

Our search method is as follows. For both the start and the goal states q_{start} and q_{goal}, all saved states within a Δ_{sparse} are found. Every combination of pairs of nearby candidate start/goal vertices $q_{nearStart}$ and $q_{nearGoal}$ is checked for visibility to its corresponding q_{start} and q_{goal} states. If both q_{start} and q_{goal} are successfully connected to the graph, A* searches over the graph to find an optimal path. In some cases the roadmap contains disconnected components, in which case a path may not be found. However, in our experiments even with a 30 DOF robot this rarely is the case.

Once a path is found, each edge is checked for collision with the current environment. Although the path is guaranteed to be free of invariant constraints, it still is necessary to check that no new obstacles exist that may invalidate our path. If a candidate solution is found to have invalid edges, those edges are disabled, and A* searches again until a completely valid path is found, or it is determined that no path exists from past experiences.

If a valid path is found, the path is smoothed and sent to the robot for execution. If no path is found, other pairs of nearby candidate start/goal vertices $q_{nearStart}$ and $q_{nearGoal}$ are attempted. If still no path is found, the candidate path with the fewest invalid segments is repaired, if one exists. To repair the invalid path, the repair technique used in the Lightning Framework is employed, such that a bi-directional RRT (RRT-Connect) attempts to reconnect disconnected states along the candidate path. If the PFS module finds a solution before a path can be retrieved and repaired, the recall request is canceled.

4.2 Results

We ran a series of benchmarks to determine the efficacy of the Thunder Framework using the Open Motion Planning Library (OMPL) [157] and MoveIt! [33] with a rigid body humanoid. All tests were implemented in C++ and run on a Intel i7 Linux machine with 6 3.50GHz cores and 16 GB of RAM.

First, we implemented and benchmarked the original Lightning Framework for a baseline

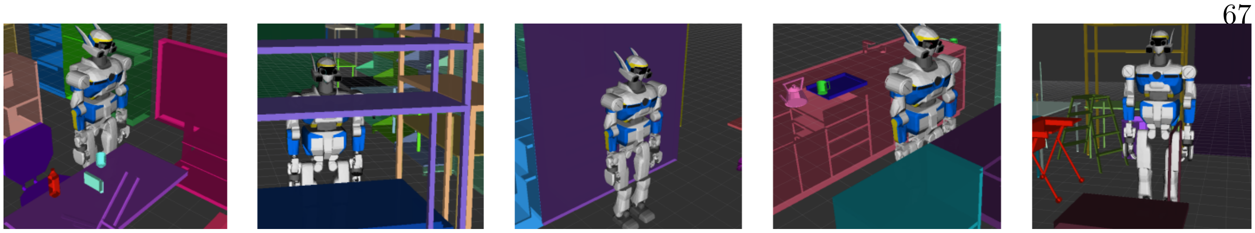


Figure 4.3: The five collision environments for HRP2 used to test repairing experiences

comparison. The parameters of Lightning were tested and tweaked, optimizing performance: the dynamic time warping threshold used for scoring path similarity was set to 4 and the number of closest candidate paths to test set to 10 in all experiments. Thunder then was implemented using a modified version of SPARS2 as our experience roadmap with the Δ_{sparse} fraction set to 0.1 and t set to 1.2. The lazy collision checking module was integrated and the insertion and recall methods incorporated as described in Section 4.1.1 and 4.1.2.

Planning from Scratch (PFS), Lightning, and Thunder frameworks were compared. For all three methods, the bi-directional planner RRT-Connect [89] was used to facilitate comparison with the original Lightning implementation. RRT-Connect is advantageous because of its ability to explore the c-space while retaining an element of "greediness." PFS was tested using two threads for a fair comparison with the experience frameworks, which both use two threads.

MoveIt! was modified to allow whole body motion planning for bipeds such as the 30 DOF HRP2 humanoid [77]. The results presented here are from simulation. The robot was placed in five different collision environments to test its ability to repair past invalid experiences, as shown in Figure 4.3.

A single contact point, in our case the robot's left foot, is fixed to the ground, and the rest of the robot balances on that foot. During PFS, all of the humanoid's joints are sampled randomly, and each candidate robot state is checked for stability, self-collision, and collision with the environment. Quasi-static stability is checked by maintaining the center of mass within the foot's support polygon. The area of the support polygon was reduced by 10% to account for modeling and mechanical error.

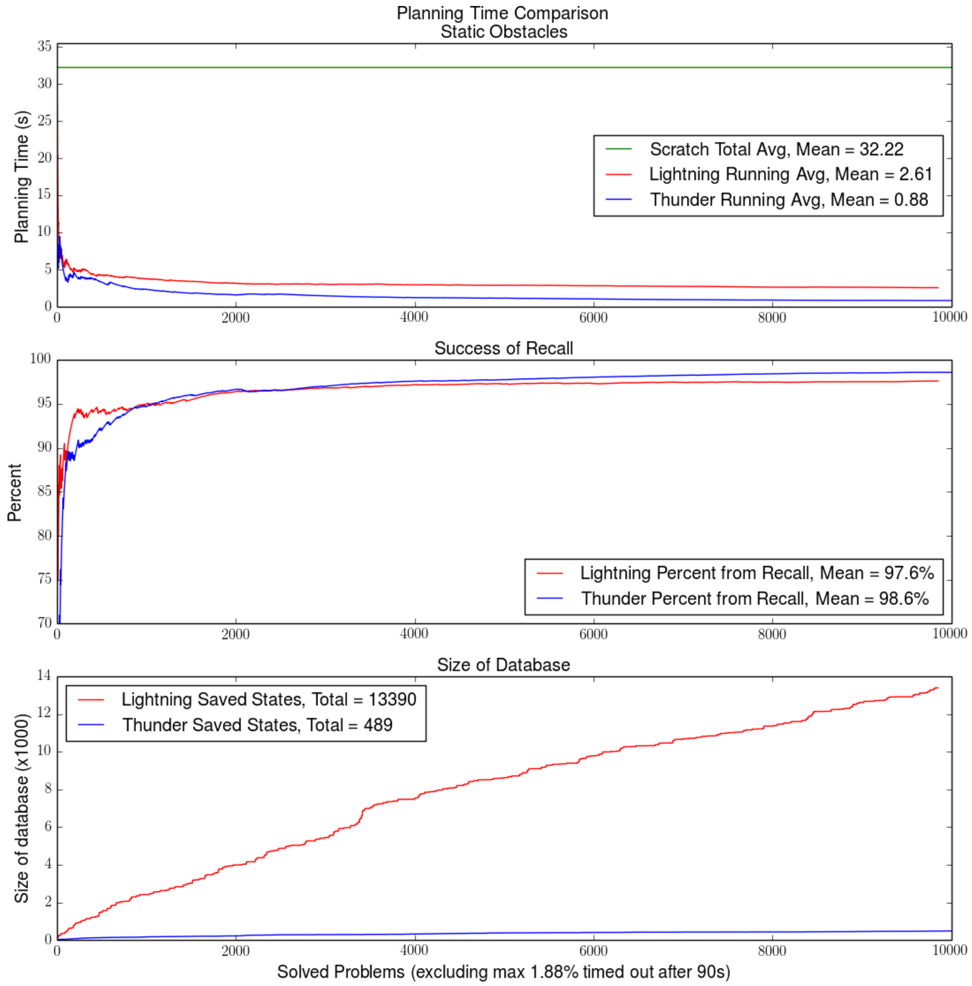


Figure 4.4: Comparison of methods with a static collision environment.

In the first test, the performance of experience planning is tested in a static, unchanging collision environment: the kitchen shown in the fourth image in Figure 4.3. The start state is always the same, and the goal state random. Planning timed out after 90 seconds and problems that had not been solved after that time are discarded, which accounted for fewer than 1.88% of runs for each method. Ten thousand runs are performed to observe how the database grew over time. Results comparing planning time, frequency of using recalled paths, and growth of the databases are shown in Figure 4.4.

The second test makes the planning problem more realistic and difficult by randomly choosing an obstacle environment for each planning problem from the five shown in Figure 4.3. This increased

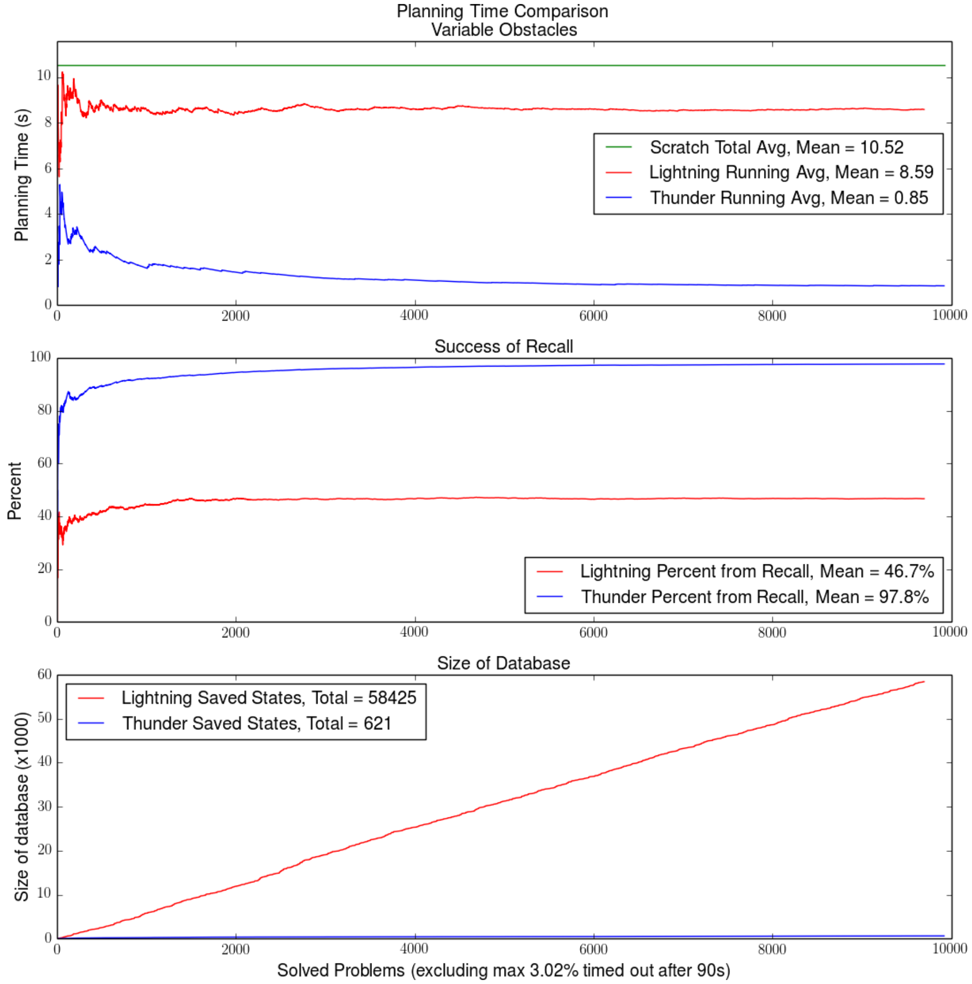


Figure 4.5: Comparison of methods with varying collision environments.

difficulty resulted in 3.02% of runs for each method being discarded due to timeouts. Results after 10,000 motion plan trials are shown in Figure 4.5.

We also compare the size difference between the experience databases in Thunder and Lightning in the second experiment. Thunder's database uses 235 kB, and Lightning uses 19,373 kB, meaning Thunder uses only 1.2% of the memory Lightning uses. Similarly, Thunder had stored 621 states and Lightning 58,425 states. Finally, the variance in response time is shown in Figure 4.6 for the three methods.

The average insertion time of new paths into the databases was 1.41 seconds for Thunder and 0.013 seconds for Lightning. Path insertions are performed in the background and do not affect

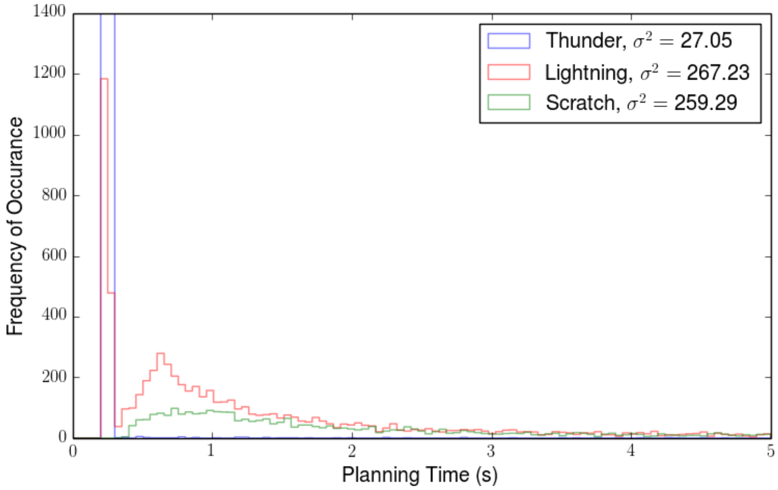


Figure 4.6: Histogram of planning time across 10,000 tests. Note that Thunder frequency has been reduced by 1/4 to improve readability.

planning time.

4.3 Discussion

Our results demonstrate that planning from past experience, using either of the Thunder or Lightning approaches, provides vastly improved query response times in hard motion planning problems. The difference is most notable in problems with a large amount of invariant constraints, such as in the top of Figure 4.5. From this we see that after 10,000 runs, Thunder outperforms Lightning by a factor of 10 and PFS by a factor of 12.3. Thunder on average solves problems 1231% faster than PFS. However, PFS sometimes can be faster for simple problems, such as when the start and goal states are nearby.

The variance in response time also improves with a graph-based experience roadmap, as shown in Figure 4.6. A path-centric experience approach requires far more repair planning using traditional sampling methods, which decreases the deterministic response time of the path. Using Thunder after it was properly trained also has the added benefit of returning paths that are more predictable for a repeated query.

In addition to response time, memory usage also is vastly improved in our approach. Because a path-centric approach is unable to reuse subsegments, it is forced to store many similar and redundant paths, which result in huge amounts of wasted memory. A path-centric experience planner suffers from the ability to connect only at the start and end of its stored paths. After 10,000 runs of random start-goal planning problems, the Lightning framework had grown to a database two orders of magnitude (98.8%) larger than Thunder's, and it appears that it will continue to increase linearly with time despite having a path similarity filter.

The SPARS theoretical claim that the probability of adding new vertices goes to zero with time has been empirically verified; during the last 1,000 trials of the second test, new vertices were being added to the Thunder database at a rate of 0.03 states per problem while Lightning was still increasing at a rate of 5.61 states per problem.

Another performance difference observed between frameworks is the frequency that a recalled path is used as shown in the middle graph of Figure 4.5. In Thunder, after building a sufficient experience roadmap, the number of problems solved by recall is 96.7%. In contrast, Lightning remains at a recall rate of 46.9% despite having a much larger database, indicating that the heuristics used in finding good paths to repair could be improved.

It was observed during experimentation that smoothing of PFS paths is important before inserting an experience into the database to reduce the chances that unnecessary curves break connectivity of an inserted path.

One major drawback from using SPARS as our database is the dramatic difference in insertion time. A solution path must first be finely discretized into a set of states, and each state must be tested against the graph spanner properties discussed in [44]. This verification step is computationally intensive, and therefore to save time, in Thunder only experiences that were planned from scratch are candidates for insertion. Still, the average insertion time of Lightning is two orders of magnitude faster than Thunder's. We do not include this computational time in our benchmarks, however, because this step easily can be done offline or while the robot is idle.

Another drawback to Thunder over PFS is that often once one solution, or homotopic class, is discovered and saved for a problem, shorter or lower cost paths can be overlooked in future, similar queries. This is because better solutions only can be discovered by PFS, and typically the recalled path will be found first, and the scratch planning will be canceled. This problem is most common when the free c-space changes, such as an obstacle being removed.

4.4 Chapter Summary

Replanning without taking advantage of past knowledge has been demonstrated to be less computationally efficient than leveraging EBMP. Our approach of using SPARS2 has shown that much less memory can be used in saving experiences for large c-spaces. This has the added benefit that multiple experience roadmaps easily can be maintained in memory, allowing task or environment-specific graphs to be stored and recalled. Next we improve our sparse roadmap criteria by optimizing it for L^1 metric spaces, making it smaller.

Chapter 5

Creating Sparser Sparse Roadmaps

In this chapter we present improvements to the Sparse Roadmap Spanners 2 (SPARS2) [42] acceptance criteria that further improve the sparsity of experience roadmaps. Following the naming convention of Thunder [34], and Lightning, we name this improvement *Bolt*—an algorithm that computes compact representations for shortest paths in continuous configuration spaces (c-spaces) in the form of roadmaps that are optimized for the L^1-norm metric space and that maintains asymptotically-near optimal theoretical guarantees on path quality. Unlike Thunder, Bolt does a one time per robot *preprocessing* step the free c-space with invariant constraints such as self-collision checking, allowing less planning from scratch that typically slows down solution time. In the results section we apply these improvements to 2D and 3D c-spaces.

Aiming at combining the best of graph search-based and sampling-based planning, Bolt uses a hybrid approach to generating a roadmap. We first apply a state space lattice pre-sampling step that inserts vertices into the graph at uniform increments of the c-space, followed by a random sampling step for filling in narrow passages. Unlike similar past work [3], we discretize in joint space rather than Cartesian work-space. The advantage of working in joint space is that it allows us to easily encode the redundancy in inverse kinematic (IK) solutions directly into the graph, with edges between vertices representing exact motions rather than underdefined end effector poses. Working in joint space also allows us to avoid creating complex and specially-tuned heuristics typical of discrete graph-search planners such as [3]. The downside is the added dimensionality and space requirements of a larger c-space, the focus of this work. One advantage of our hybrid approach

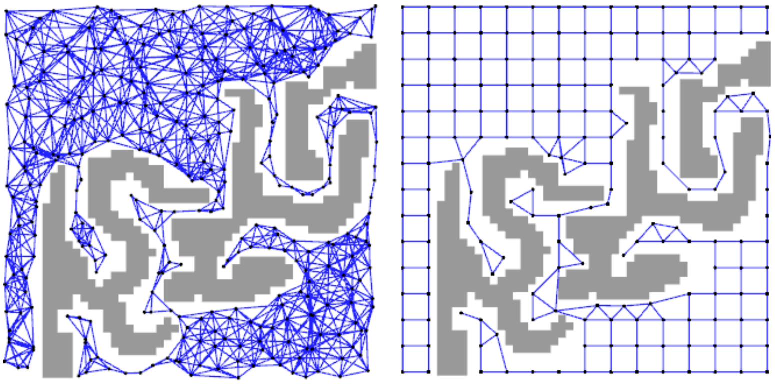


Figure 5.1: Comparison of (left) SPARS2 roadmap: *vertices: 334, edges: 1392* with (right) Bolt roadmap *vertices: 173, edges: 262*. Both graphs return paths of the same quality in L^1 metric spaces.

is that even coarse state space lattices can capture the necessary detail via the secondary random sampling preprocessing step, combined with asymptotically near-optimal quality path guarantees.

Choosing a proper distance function for joint space (e.g. $\mathbb{R}^6$) is non-intuitive due to the kinematic constraints of a robotic arm's geometry. Most literature has focused on SE3 spaces and variations of the L^2 Euclidean distance [5]. Distance calculations are one of the most numerous operations in a Probabilistic Roadmap (PRM) [5], so for computational motivations the L^1-norm metric function (*Manhattan distance*) is often used, such as in the MoveIt! Motion Planning Framework [33] that was used in this work. Using the L^1-norm usually will return less smooth paths, but these easily can be smoothed in post-processing. Additionally, the L^2 distance between two points can be bounded by the L^1 distance. It is not clear whether using L^2 or alternative distance functions [5] is a necessarily better measurement than others, but the remainder of this thesis will assume L^1 is used for the computational and space advantages.

Despite creating sparse roadmaps, sparse roadmap spanner's original specification still is inefficient in its graph size for various reasons presented here. SPARS2 is “slightly denser” [42] than the original Sparse Roadmap Spanners (SPARS) algorithm, sacrificing graph density for lower initial graph construction memory requirements. Bolt is built upon SPARS2 [42] and addresses those shortcomings with the trade-off of slower preprocessing times.

Solution time in high dimension c-spaces such as dual-arm robots is a goal of the work in this chapter—our planner searches the generated roadmap using the A* algorithm. A*'s time complexity is $O(|E|) = O(b^D)$, where b is the *branching factor* or average number of edges connected to each vertex, and D is the depth of the search. Therefore, reducing the number of edges in a graph will also reduce retrieval time for a path. Another often overlooked component for solving motion planning is the nearest neighbor (NN) search: reducing the number of vertices results in important NN speedups.

Bolt was developed with goal of removing the need for the Planning from Scratch (PFS) component, instead relying fully on a preprocessed and highly efficient roadmap of the entire c-space. Like Thunder, we reuse our roadmap multiple times, not only for a given environment, but for all environments. Bolt can also be used for multi-modal task planning, allowing multi-goal motion planning problems to be quickly solved by duplicating the entire joint-based Bolt roadmap for every discrete planning step and adding transition edges.

Our hybrid approach of combining discretized lattices with random sampling is similar to the extensive work on the subject in [97]. They surprisingly found that deterministic sampling methods are superior to the original PRM, noting that by definition a "collection of pseudo-random samples should have too many points in some places, and not enough in others." Our approach is most similar to their proposed subsampled grid search (SGS), where discretized vertices along a grid are coarsely spaced, and a local planner is used to collision check the edges between the grid. The unique aspect of our approach is that we size the grid optimally for the requirements of the spanning graph and additionally perform random sampling as a second step. To the best of our knowledge, this hybrid approach is unique in the literature.

Six modifications to the original method of SPARS are presented in this chapter that result in an average roadmap size reduction of 79% in 3D and an average planning time speedup of 35%. These improvements include (and their corresponding edge reduction):

- A discretization pre-sampling step to efficiently cover large areas of free c-space: 13%.

- An exact method for choosing the t-stretch factor optimized for the L^1 metric space: 15%.
- Methods to reduce outdated and redundant edges: 40%.
- An extra check for the connectivity criterion to determine if a vertex addition can be avoided: 36%.
- An additional smoothed quality path criterion: 12%.
- Modification of quality criterion for L^1 norm: 27%.

We formally describe the proposed method and demonstrate its performance in a variety of 2D and 3D environments.

5.1 Methods For Creating Sparser Sparse Roadmaps

5.1.1 Hybrid Discretization and Sampling

Here we present the hybrid approach to generating a roadmap that combines graph search-based planning and sampling-based planning. Discrete graphs avoid redundancy by equally spacing vertices and edges through the c-space, but can miss narrow passages due to resolution coarseness or consume too much memory and search time. Whereas sampling-based planners address these issues, they are not efficient if the goal is to create a graph that provides the near-asymptotically optimal properties in the fewest vertices and edges possible.

In Bolt, we initially cover the c-space with a discrete graph, providing efficient coverage of free space, and then sample to allow coverage of the more complex areas of space that interface with invalid regions. We must ensure that no vertex or edge added to the graph is in violation of the near-asymptotically optimal guarantees. In fact, with the following discretization method, and in the absence of any obstacles or other constraints, the random sampler with sparse roadmap criteria are unable to add any extra vertices beyond those added by the discretization step.

We discretize our space using a standard d-cubic honeycomb pattern, which in 3D is a homogeneous grid of squares. The discretization size β is calculated through a geometric formula

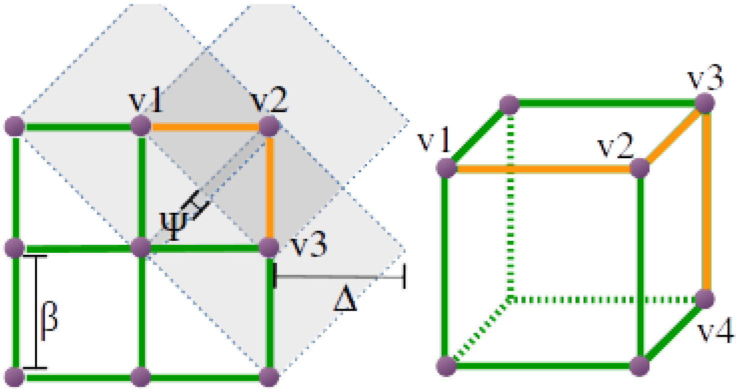


Figure 5.2: Demonstration of how the optimal discretization factor β is chosen for the L^1 norm in a 2D discretized grid (left) and 3D discretized grid (right). The gray transparent diamonds around the vertices $v1$, $v2$, $v3$ represent the visibility regions of Δ_{sparse}. The overlap of those regions is the region overlap, labeled Ψ. β is visualized as the length of the solid green line between two vertices. The two orange lines are highlighted to demonstrate that $dist(v_1, v_3) = dist(v_1, v_2) + dist(v_2, v_3)$

that leverages the known properties of graph spanner combined with the space's metric function and number of dimensions d. We use the sparse delta visibility radius (Δ_{sparse}) that specifies the coverage a vertex provides over the c-space. We introduce the concept of *region overlap* Ψ that defines the amount of penetration between two neighboring vertex's visibility regions. Ψ should be some small value greater than zero, with the trade-off that the smaller the value, the lower the probability of an edge being created between two vertices with a shared *interface*, but the larger the value, the more vertices are required in the graph. Here, an *interface* $i(v_1, v_2)$ between two vertices v_1 and v_2 is the shared boundary of their visibility regions, as defined in [146]. The visibility regions are illustrated in Figure 5.2 where the vertices v are each configurations of two dimensions with values (x, y).

We desire to find a value for β that distributes the vertices such that, lacking any constraints, would provide complete coverage for a given Δ_{sparse} across the c-space. To achieve this, we must find the maximum distance across any two discretized vertices that share an interface. Because we are using the L^1-norm and the d-cubic honeycomb pattern, the max distance in 2D is:

$$\begin{aligned} dist_{max} &= distL^1(v_1, v_3) \\ &= (x_3 - x_1) + (y_3 - y_1) \\ &= 2 \cdot \beta \\ &= d \cdot \beta \end{aligned} \tag{5.1}$$

The last line generalizes the result to d dimensions, for example the 3D case demonstrated in the right side of Figure 5.2. To ensure complete coverage of the space, the furthest vertices with shared interfaces must have slightly overlapping visibility regions given their Δ_{sparse}. Therefore, the two vertices with a shared interface with max distance apart need to have a distance of:

$$dist_{max} = 2 \cdot \Delta \tag{5.2}$$

where the constant 2 is invariant for all dimensions and represents the two vertices in question. For example, in Figure 5.2a, v_1 and v_3 share an interface and have slightly overlapping visibility regions labeled Ψ.

Combining these two functions, adding the necessary Ψ, and solving for the discretization level:

$$\beta = \frac{2 \cdot \Delta}{d} - \Psi \tag{5.3}$$

that is, for a given Δ_{sparse}, this equation specifies what β to use to have the minimum number of vertices and edges in the graph. As an aside, in our experiments we also used the L^2-norm for testing, which requires a slightly different formula based on the ubiquitous L^2 distance function:

$$\begin{aligned} dist_{max} &= distL^2(v_1, v_3) \\ &= \sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2} \\ &= \sqrt{d \cdot \beta^2} \end{aligned} \tag{5.4}$$

Which, similar to the L^1 version, results in a discretization size:

$$\beta = \sqrt{\frac{4 \cdot \Delta^2}{d}} - \Psi \tag{5.5}$$

In the rest of the chapter we will continue to assume the L^1 norm is used. It is desirable to generalize to L^1 because many intuitive assumptions about graph connectivity are invalidated, allowing many redundant edges to be eliminated in a roadmap. This is because it is possible for two vertices that share an interface with each other (but no edge) to have the same length path via another vertex than by a direct edge between them.

5.1.2 Exact Method for Choosing t-Stretch Factor

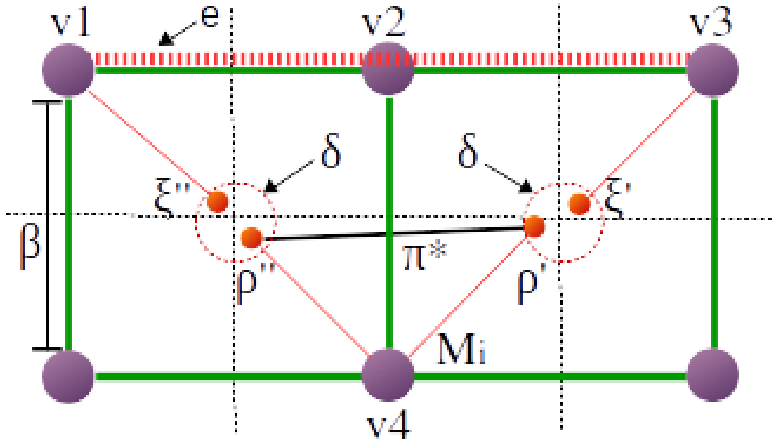


Figure 5.3: Demonstration of a geometric method for choosing t-stretch factor. The red dashed line is the candidate edge that should be avoided from being added. The orange vertices ξ, ρ are the interface vertices that represent an interface between the vertices $[v_1, v_2]$. The black dashed lines represent the exact interfaces between the vertices.

The SPARS algorithm path quality guarantees are largely based on the t-stretch factor. The value of t critically affects the number of edges added to the graph and thus the performance of search. Rather than arbitrarily choose a value, we develop a formula to calculate this stretch factor to remove redundant edges added by the default SPARS method. The geometry we chose to avoid is the overlap of *double* edges—edges that span the length of three vertices instead of two as pictured in Figure 5.3. In this figure we show that candidate edge $e = L(v_1, v_3)$ duplicates the two adjoining edges $L(v_1, v_2)$, $L(v_2, v_3)$. If the stretch factor is too small, the SPARS quality criterion algorithm will add duplicate edges like this one. That criterion, as explained in [42], will add an edge if:

$$t \cdot \pi^* < M_i \tag{5.6}$$

where π^* is the optimal path between the interior interface vertices (ρ', ρ'') and M_i is the midpoint path $[\rho'', v_4, \rho']$. We want to find the worst-case shortest length of π^*, which is when each pair of representative vertices $[\xi', \rho']$ is at its maximum distance apart. This length is δ by definition. Since we know the discretized distance β between vertices and that we are using the L^1 metric function, we can solve for these two line segment lengths:

$$\begin{aligned} M_i &= \frac{1}{2}(2\beta + 2\beta) \\ &= d\beta \\ \pi^* &= \beta - 4\delta \end{aligned} \tag{5.7}$$

Solving Equation 5.6 with Equation 5.7 gives us our minimum t-stretch factor to prevent these double edges from occurring:

$$t > \frac{d\beta}{\beta - 2\delta} \tag{5.8}$$

5.1.3 Methods to Reduce Outdated/Redundant Edges

Often during roadmap construction, edges are added to the graph based on the interface or quality criterion that are later no longer needed after newer vertices have changed the visibility regions of the c-space. Two techniques were added in Bolt to remove these unnecessary edges:

First, a simple delay is added in utilizing the SPARS quality criterion until the c-space has a high probability of having full vertex coverage. This means most, if not all, of the free c-space is within the Δ_{sparse} of a vertex. In practice, random samples are added without checking against the quality criterion until some number of failures $M_{coverage}$ occur. In our testing we used 5000 failures. After this threshold is reached, we continue to add random samples but with the quality criterion enabled.

Second, we clear all nearby edges within a Δ_{sparse} of any new vertex added. This causes all edges in that visibility region to be re-created taking into account the new vertex, which results in some previous edges never being recreated.

5.1.4 Improving Connectivity Criterion

In the original SPARS implementation, we found that more vertices than necessary were being added to satisfy the connectivity criterion. Whenever a new sample q was able to connect at least two vertices w_1, w_2 within distance Δ_{sparse} that are otherwise disconnected, the sample q is added to the graph with corresponding edges to the disconnected neighbors w_1, w_2. While this is sometimes necessary due to constraints, often an edge e_{direct} can be added that directly connects w_1, w_2 and avoids adding q. This reduces the number of overall vertices in the graph. This new edge e_{direct} still upholds the property that all $\forall e \in E_g : |e| \leq 2\Delta$ because all searched neighboring vertices of new sample q are of distance $\leq \Delta$. Therefore, two neighbors of q on opposite extremum of its visibility region are at most 2Δ apart.

5.1.5 Improving the Quality Path Criterion

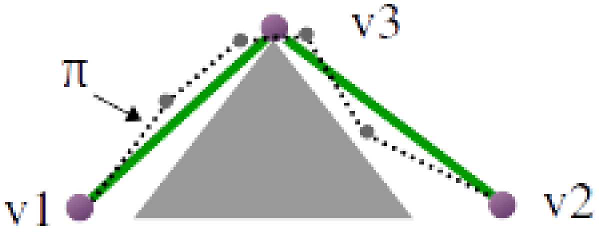


Figure 5.4: Example of a smoothed path π that does not improve the path length between v_1 and v_2 but only increases the graph size when added.

An improvement is presented that alters the logic for connecting two vertices $[v_1, v_2]$ for the quality criterion. If the connecting edge $e \in L(v_1, v_2)$ is found necessary, but no direct edge can be added due to obstacles, the SPARS algorithm will create a new path with configurations supporting the interfaces $i(v_1, v_3)$ and $i(v_3, v_2)$ and then “try to smooth the remaining path as much as possible” [43]. However, there are often cases, especially in L^1, where the current graph already has the optimal path around the obstacles, and the new smooth path does not improve path length between $[v_1, v_2]$, as shown in Figure 5.4. As proposed in SPARS, the graph will still add unnecessary vertices and edges from the smoothed path and increase the size of the graph. In our improvement, we add an extra check requiring that the distance of the newly smoothed path be less than the current shortest path between the two vertices. This is accomplished with an additional call to A*.

5.1.6 Modification of Quality Criterion for L^1 Space

Here we partially relax the SPARS requirement in L^1 spaces that all interfaces for pairs of vertices $v_1, v_2 \in V_S$ are eventually connected by an edge as the number of samples approaches infinity. The result is that superfluous edges are avoided from being added. This is implemented by running A* to find the shortest path between two vertices during the quality criterion checks.

This relaxation applies to sets of three vertices whose values in each dimension are monotonic. Formally: for all sets of vertices $v_1, v_2, v_3 \in V_G$ where edge $e_1 = L(v_1, v_2) \in E$ and $e_2 = L(v_2, v_3) \in E$, for every dimension x in C the set of values $[x_1, x_2, x_3]$ all increase or decrease in the same direction. When this geometry is present, the original SPARS algorithm will eventually add a third edge e_3 as part of its *quality* criterion. However, it easily can be shown that, in an L^1 space, e_3 has the same length, as the two other edges e_1, e_2 added together, as shown in Figure 5.5. Therefore, in terms of path length this third edge does not improve the quality of paths being generated in the graph. This relaxation breaks the SPARS proof *asymptotic near-optimality w/additive cost*, which we revise in Section 5.2.

This modification is implemented within the quality criterion tests. First, within the `Add_Shortcut` function of SPARS [43], we add an additional condition that requires the length of the candidate edge $L(v_1, v_2)$ be *greater than* the length of the shortest path π_G in G_s between v_1 and v_2. With a L^2-norm distance function, this would be impossible since no edge currently exists between v_1 and v_2 in the `Add_Shortcut` function. However, in an L^1-norm space, this is guaranteed to happen whenever a discretized lattice structure of vertices is used to cover the space. This shortest path π_G is found by running A* for every candidate quality edge.

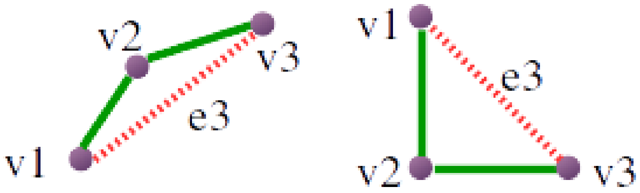


Figure 5.5: Two examples of sets of 3 vertices with monotonic values in multiple dimensions. In both examples the addition of edge e_3 does not improve the path length between v_1 and v_3 with a L^1 metric function.

5.2 Properties of Bolt

Bolt is based on SPARS2 and has the same asymptotically near-optimal path quality guarantees. However, in Section 5.1.6 we relaxed the requirement that all vertices that share an interface be connected by an edge in special circumstances. This breaks the *Connected Interfaces* and *Spanner Property* proofs in the SPARS2 work. Next we modify those proofs from the original:

Theorem 5.2.1 (Connected Interfaces—Modified)**.** *Using the L^1-norm, $\forall v_1, v_2 \in V_S$ which share an interface, either $\exists L(v_1, v_2) \in E_S$ or $\exists \pi(v_1, v_2) \in G_S$ where $|\pi(v_1, v_2)| = |L(v_1, v_2)|$ with probability 1 as M goes to infinity.*

The base proof is found in the original SPARS2 *Connected Interfaces theorem* and says that if there exists an interface between two vertices, there is a non-empty set of sampled states that will eventually be generated that will bridge the interface with a new edge. This is guaranteed by the SPARS2 interface criterion.

Our modification of this theorem allows interface edges to *not* be added when there exists a path through G_s with the same length. Because there is already a path in G_s with the same length as the candidate edge, not adding the edge will not effect the path length of any returned solution in G_s. However, it still must be proved the theoretical guarantees have not been violated in the next lemma.

Lemma 5.2.2 (Coverage of Optimal Paths by G_s Modified)**.** *Consider an optimal path π^* in C_{free}. The probability of having a sequence of vertices in S, $V_\pi = (v_1, v_2, ..., v_n)$ with the following*

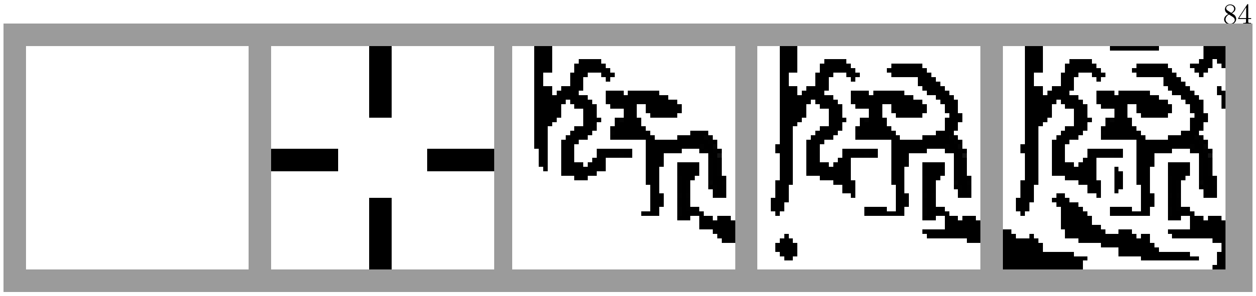


Figure 5.6: The five 2D collision environments of sequentially more complex difficulty used for testing Bolt: Map 1 through Map 5. For the 3D case the obstacles were simply extruded into the third dimension for simplicity of debugging.

properties approaches 1 as M goes to infinity:

- $\forall q \in \pi^*(q_0, q_m)$, $\exists v \in V_\pi : L(q, v) \in C_{free}$
- $L(q_0, v_1) \in C_{free}$ *and* $L(q_m, v_n) \in C_{free}$
- $\forall v_i, v_{i+1} \in V_\pi$, $L(v_i, v_{i+1}) \in E$ *or* $\exists \pi_G(v_i, v_{i+1})$ *where* $|\pi_G(v_i, v_{i+1})| = |L(v_i, v_{i+1})|$

Here we have modified the third bullet point in Lemma 5.2.2 to allow for paths π_G that are not direct edges to provide coverage for an optimal path π^* so long as this π_G has the same length. Again, the resulting solution path must be of the same path quality because the substitute path proposed here has the same length as the direct edge it replaces.

The remaining proofs of the original SPARS hold given our modified lemmas above because any use of the removed edges can be substituted with the π_G of same length.

5.3 Results

We compare the size of the graph generated by SPARS2 and Bolt across multiple environments of various complexities using a point robot. The five environments we tested against in two and three dimensions are pictured in Figure 5.6, ranging from a obstacle-free environment to a highly-cluttered map with many narrow passages. For each map we ran 10 trials for both SPARS2 and Bolt. Runs were tested with the stretch factor calculated as described in Section 5.1.2 and shown in Table 5.1. The termination condition M was set very conservatively to ensure with high

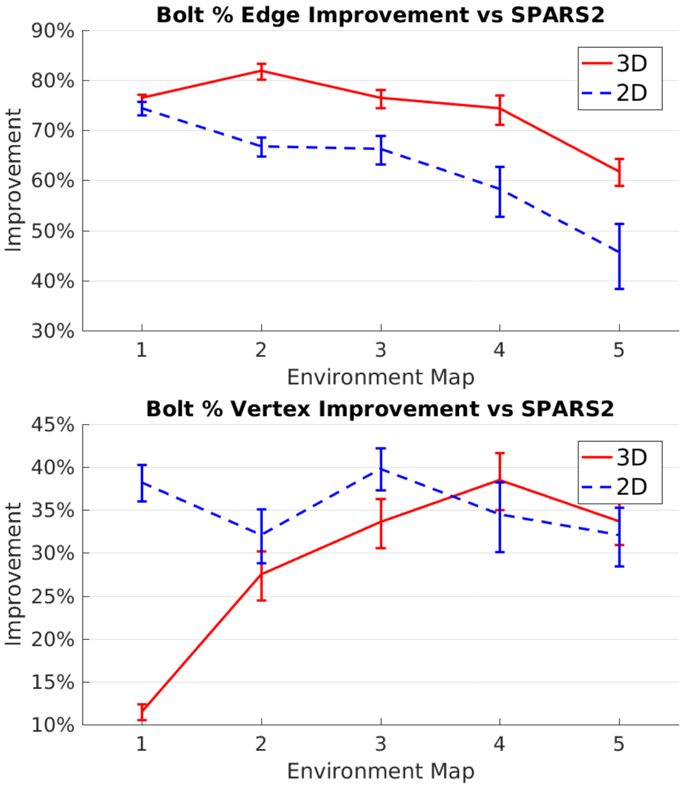


Figure 5.7: Percent of edges (top) and vertices (bottom) improvement between SPARS2 vs Bolt roadmaps

probability that every possible edge and vertex was added to the graph for full coverage, thereby requiring no new edge or vertex be added for $M = 15,000$ random samples before considering the graph *complete*.

Table 5.1: Parameters of Bolt

d	Δ	δ	t	Ψ
2	6.93	0.693	3.36	0.01
3	8.49	0.849	7.68	0.01

We used the SPARS2 implementation provided by the original authors in Open Motion Planning Library (OMPL) [157] with two minor modifications. We used an obstacle clearance as suggested in the original SPARS work, set to 1 unit, and we fixed a bug in the quality path smoothing implementation so that the graph did not increase in size infinitely. Whenever possible,

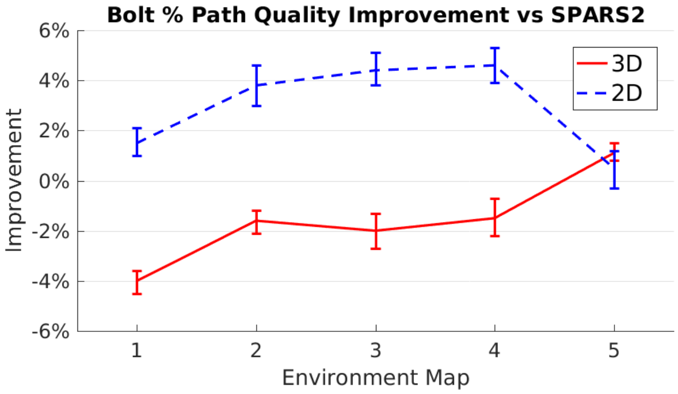


Figure 5.8: Path quality improvement between SPARS2 vs Bolt roadmaps

we used the exact same parameters for both SPARS2 and Bolt.

For every generated roadmap, we planned 1000 random paths through the space and verified the result was with the t-stretch factor of the optimal path. This also ensured we had sufficient coverage and connectivity between all possible states—we had no failed plans in our tests. Each random path was then smoothed and the difference in path quality recorded.

The resulting space optimizations are shown in Figure 5.7. For a 2D c-space without obstacles Bolt adds 74% fewer edges and 38% fewer vertices. In the most cluttered environment Bolt still generates 46% fewer edges and 32% fewer vertices. Similar results are shown for the 3D case. Notably, for collision-free spaces in 3D, there is only a small improvement (11%) with Bolt in the number of vertices necessary to cover the space. This suggests our vertex count improvements work best in cluttered environments and that random sampling performs similarly in free space for both algorithms.

The resulting path quality was compared between Bolt and SPARS in Figure 5.8. Given the standard deviations of errors, the path quality of both methods was essentially the same. However, in the 2D data set, Bolt returned paths with around 3% better path quality with is impressive given the corresponding 62% average reduction in edges. In the 3D data set, Bolt had around 2% *worse* path quality, which meets expectations that the reduction in edges and vertices would result in slightly worse path quality.

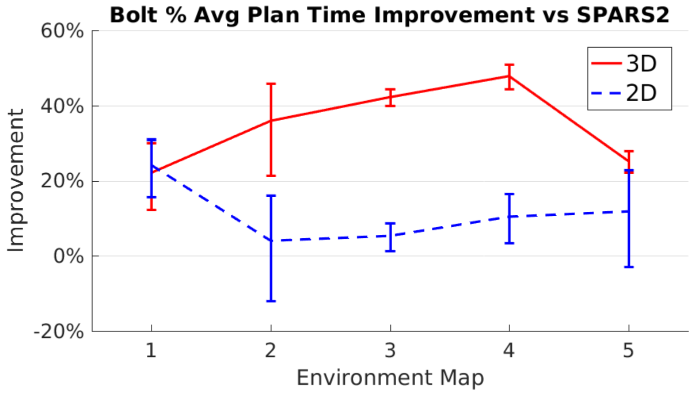


Figure 5.9: Planning time improvement between SPARS2 vs Bolt roadmaps

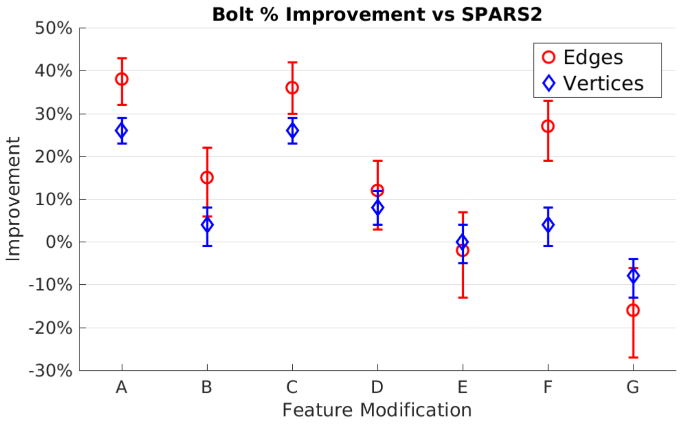


Figure 5.10: Contribution of each improvement to improvement between SPARS2 vs Bolt roadmaps. A) Delay in quality criterion, B) t-stretch factor formula, C) Connectivity criterion check, D) Smoothed path quality criterion, E) Edge removal after vertex addition, F) Edge improvement rule, G) Discretization pre-sampling

The pure A* planning time over the precomputed graph, without changing environments or collision checking, is compared in Figure 5.9. Bolt is faster than SPARS2 on average 11% in 2D and 34% in 3D, indicating that higher dimensions could continue to improve online query resolution time.

Contributions of each individual improvement to the reduction of edges in SPARS2 is shown in Figure 5.10 and are as follows for the 2D space using map 3: the delay in quality criterion 38%, t-stretch factor formula 15%, connectivity criterion check 36%, smoothed path quality criterion 12%, edge removal after vertex addition -2%, edge improvement rule 27%, discretization pre-sampling

-16%. Some of the features depend on each other for improvement, so there is actually an increase in graph size for two of the features. Notably—the removal of edges around new vertices is highly dependent on the connectivity criterion check, and upon further investigation it was found that this feature still contributes to a 2% improvement in the number of edges in the Bolt algorithm. Similarly, the use of a discretized lattice does not improve the SPARS2 algorithm if used alone, but its overall improvement to the Bolt algorithm's average edge count was measured to be 12.45%.

5.4 Discussion

Our improved sparse roadmaps in 3D have demonstrated a reduction in the average number of edges and vertices 74% and 29%, respectively. This resulted in a consistent speedup in solution time (average 35%) and almost no loss in path quality (-2%).

It is difficult to compare the size savings of our method to traditional sampling-based roadmap planners such as PRM or PRM* because they lack an appropriate termination condition; therefore, any results on the graph size of those planners would be arbitrary. For both Bolt and SPARS2, we are able to stop adding samples to the graph when we know with high probability that no further samples can be added to the graph.

5.5 Chapter Summary

We have shown many techniques that further improve the advantages to using SPARS for motion planning, in particular demonstrating improvements of up to 77% reduction in graph size. Pre-computing a full sparse roadmap for motion planning allows for more deterministic solutions to be solved faster. Expensive invariant constraints such as self-collision checking are built into the roadmap ahead of time.

In this chapter we have focused on 2D and 3D problems because systematically studying the properties of higher degrees of freedom (DOF) systems proved too computationally intensive with current hardware—given our termination criterion that all edges and vertices are added. Although we have been successful in pre-computing sparse roadmaps for larger c-spaces, as we will show in the following chapter, the size of the roadmap becomes problematic during experimental validation.

Chapter 6

Preprocessing High-DOF Experience Roadmaps with Invariant Constraints

In this chapter we apply and adapt the graph theory results of *Bolt* presented in the previous chapter to a unified experience-based motion planning (EBMP) framework and benchmark its performance using a dual-arm high-dimensional robot in a challenging environment containing many narrow passageways. We compare and contrast the two approaches presented in Chapter 4 and 5. The primary question investigated in this chapter is, in large configuration spaces (c-spaces), is it 1) more efficient to use a preprocessing step to generate a complete roadmap encoding invariant constraints or 2) to only save experiences previously used in the experience roadmap? The conclusion reached in this chapter is that preprocessing is, in fact, slower than the Thunder approach in many circumstances, but the work laid out here is useful for certain applications, which will be presented in Chapter 7 on multi-modal planning.

6.1 Overview

In our Thunder algorithm presented in Chapter 4, the Planning from Scratch (PFS) component repeatedly searches a space that is mostly the same for each query, neglecting to utilize past knowledge to speed up graph growth. The Retrieve Repair (RR) component was able to utilize past knowledge but only over time as more environments and queries were solved from scratch.

An alternative approach proposed in Chapter 5 and explored in this chapter is to eliminate the PFS component by generating a complete roadmap offline that is optimized for a particular robot when free from variant constraints. This would allow all invariant constraints to already be accounted for—only changing environments would need to be later validated and repaired against. This further reduces or eliminates the amount of growth required of the roadmap during planning. Preprocessing is especially useful for highly constrained applications like humanoids with stability constraints, fixed-base robots that are surrounded by tight workcell walls, and dual-arm robots whose arms can easily collide. The goal of preprocessing is to make the smallest base roadmap possible that is able to connect every possible free space configuration of the robot in order to reduce the amount of unnecessary roadmap rebuilding for every motion plan.

To utilize this preprocessed roadmap for changing environments, additional sampling is required during runtime when parts of the roadmap become invalidated due to obstacles. This can be accomplished either by randomly sampling within the pre-generated roadmap or by using a second component to plan from scratch similar to Thunder. When sampling within the pre-generated roadmap, the algorithm essentially becomes a pre-seeded Probabilistic Roadmap (PRM) approach that it then prunes and grows as necessary for changing environments while still maintaining certain sparse roadmap guarantees. After a solution is found, it is added to the experience roadmap at a finer discretization level than the preprocessing step to improve the path quality of solutions and account for variant constraints.

One difficulty of full preprocessing, and EBMP in general, is the need to solve simple problems fast despite large amounts of previous experiences to search through, repair, and validate. PFS can sometimes perform faster simply due to smaller data structures that are able to solve things such as nearest neighbor (NN) queries efficiently. Yet for high-dimensional problems with expensive constraints and narrow passageways, having a prebuilt roadmap can often outperform PFS.

We will continue to call this preprocessing experienced-based planning approach *Bolt* as partially described in Chapter 5. However, as will be explained later in this chapter, it is necessary to relax some of the sparse graph criteria to allow the problem to become computationally tractable for large c-spaces. In addition, the hybrid discretization and sampling method used in the previous chapter proved to no longer apply due to the increased size of the sparse delta visibility radius (Δ_{sparse}) as will be presented. We will also describe two methods for using a pre-generated roadmap of invariant constraints: Bolt Dual Roadmap (BoltDR) and Bolt Single Roadmap (BoltSR).

6.2 Roadmap Preprocessing

We fully preprocess the motion roadmap by running an extensive offline random sampling phase to generate a course-grained set of experiences. Because of the curse of dimensionality—the exponential growth of the size of a c-space as dimensions is added—the precomputed graph must be very sparse to ensure the roadmap does not become too large for our high-degrees of freedom (DOF) applications on modern computers. In addition, the criteria used to accept or reject vertices and edges must be computationally cheap so that a fully computed roadmap is tractable.

To generate a sparse roadmap in high-dimensional spaces, the Δ_{sparse} must be reasonably large. In this chapter, after parameter sweeping we chose to create a sparse roadmap at the coarsest level possible: with the Δ_{sparse} for each vertex set to *maximum extent*, or maximum distance, possible between any two configurations of the robot. Formally: we set $\Delta_{sparse} = \sum_{n=1}^{N} Jn_{high} - Jn_{low}$ where N is the number of dimensions and J is each joint's low and high limits. This visibility range is, in fact, the same used in the original Visibility-based PRM (V-PRM) method that SPARS was partially inspired from: in V-PRM every sampled state is checked against every other vertex in the graph for visibility regardless of distance.

In a space void of constraints, this would result in a roadmap with only one vertex that provided complete visibility coverage of the entire c-space. In a highly constrained dual-arm robot operating in a tight work cell with no clearance, a roadmap of 200 thousand vertices and 2 million edges is still generated. One downside of using the maximum visibility Δ_{sparse} is that when using the original Sparse Roadmap Spanners 2 (SPARS2) criteria, for every sample that is attempted to be added to the roadmap, it must be collision checked against every other vertex (n^2 checks) in the graph. This is prohibitively expensive. The alternative of using a smaller Δ_{sparse}, however, results in far larger roadmaps.

Another downside of using the maximum visibility Δ_{sparse} is that the discretization findings presented in Chapter 5 no longer apply, as the discretization size is based on Δ_{sparse} being a fraction of the maximum extent. When testing on a full 14 DOF robot, the discretization approach proved too exponentially large for modern computers.

In the following two subsections, we present findings in how to relax both the *quality* and *connectivity* criterion to more quickly build an even sparser roadmap using less memory. We call the result the *relaxed sparse roadmap criterion*, which is a sparse roadmap that is no longer provably asymptotically near optimal but in practice has shown desirable performance well suited for EBMP.

6.2.1 Relaxing Quality Criterion

The SPARS2 quality criterion uses random sampling within a δ radius of a candidate sample to approximate the interface between nearby samples, which was explained in Section 2.3.5.2. However, tracking the interfaces between vertices in high dimensions for a fully generated roadmap and sampling nearby neighbors for quality proved too slow and memory-intensive for this work.

In the following we motivate the difficulty of using the SPARS2 *quality* criterion for a hypothetical 14-DOF dual-arm robot with no constraints on the space (no joint limits, self-collision, etc). The quality criteria requires two additional states be saved for every neighboring interface to define the approximate location of the interface within a hyper-sphere of radius δ (see Figure 2.5). In an ideal lattice structure in the form of a hypercube graph for N dimensions in an L^1 metric space, a fully connected state will have $2N$ edges and therefore $2N$ interfaces to nearby vertices. For $N = 14$ DOF, 28 interfaces would require 56 extra states be saved in memory per vertex. This means an ideal discretized roadmap generated to cover a c-space will require 56 times the memory than one that does not use the SPARS2 quality criterion.

Using the discretization method presented in Bolt in Chapter 5 with a discretization of $\pi/4$ radian for 14 joints with ranges of $[-\pi, \pi]$, our hypothetical roadmap including the quality criterion would include 15,032,385,536 states. Using 8 bytes per float, the total memory requirements for the states in this hypothetical graph using quality criterion would be 1.5 TB. Without the quality criterion, it would be 28 GB.

There are two criteria that require every other vertex in the graph to be collision checked

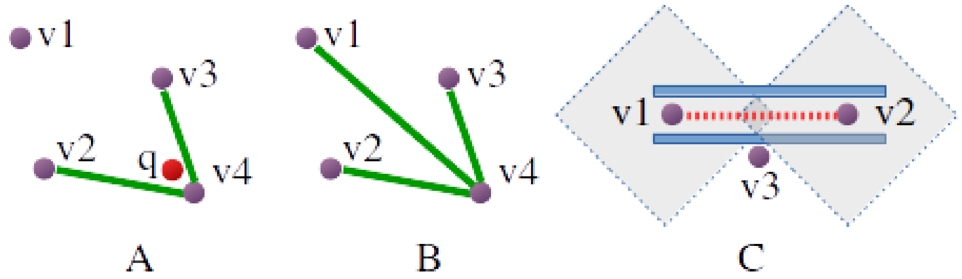


Figure 6.1: Demonstration of A) Effect of connectivity criterion when random sample q is nearest vertex $v4$. All vertices within Δ_{sparse} are in the same connected component except $v1$, so an edge is added between $[v1, v4]$. This is suboptimal as shown in B)—fewer edges would have been added if the connectivity criterion had not been used and the interface criterion eventually added edges between $[v1, v2]$ and $[v1, v3]$. C) An adversarial configuration of a sparse roadmap via the connectivity criterion. This diamond represented Δ_{sparse} range.

against (n^2 checks): the now relaxed *quality* criterion and the *connectivity* criterion. Presented next we simplify the connectivity criterion so we can fully eliminate the need for n^2 collision checking.

6.2.2 Improved Connectivity Criterion

In addition to relaxing the quality criterion, we observe that the SPARS2 *connectivity* criterion is also unnecessary for all but a few adversarial cases that in practice are rare. The original connectivity criterion checks if, for all visible neighbors within Δ_{sparse}, the new candidate sample q_{new} can connect two disconnected components within the sparse roadmap. If it can, either a direct edge between the disconnected components is created if possible (per Chapter 5) or q_{new} is added as a bridge with two additional edges.

One advantage of the connectivity criterion is that it speeds the growth of the roadmap such that necessary edges are more likely to be added quicker. This speedup is not as necessary when pre-processing the roadmap; most all edges added via the connectivity criterion would eventually be added by the interface criterion anyway.

One downside of the connectivity criterion is that extra edges are often added that are unnecessary because the two vertices do not actually share an interface. This can be seen in Figure 6.1A, where a new sample q is being tested. It fails the *coverage* criterion because there are vertices within Δ_{sparse}. However, it finds that $[v_1, v_4]$ are in different connected components and so it adds an edge between that pair. This creates a larger than necessary roadmap; without the connectivity criterion, the *interface* criterion eventually would have sampled vertex on the interface and add direct edges between $[v_1, v_2]$ and $[v_1, v_3]$.

However, eliminating the connectivity criterion is not ideal because of the adversarial case shown on the right in Figure 6.1. Here two vertices $[v_1, v_2]$ are in a narrow passageway and need to be connected to the dotted line and shown in the figure. No vertex can be added by the coverage criterion between $[v_1, v_2]$ because their visibility region Δ_{sparse} completely covers the passageway. The *interface* criterion would normally connect these two vertices, but a non-visible vertex v_3 is the nearest or second nearest neighbor to any sampled vertex within the narrow passageway. Because the interface criterion only considers the two closest nearest neighbors, no sample within the narrow passageway will ever trigger an edge to be added between $[v_1, v_2]$.

The improved version we utilize in our method is to only check connectivity between the *two closest* visible vertices in the roadmap—in contrast to checking against all visible vertices within Δ_{sparse}. This allows the n^2 collision checks to be skipped, but still maintains the important *probabilistically complete* property for our roadmap given the condition that all vertices in the roadmap are some clearance $cl > 0$ of all neighboring vertices. The probability of connecting any two disconnected components is 1, as the value of M goes to infinity, and therefore our sparse roadmap approach is probabilistically complete, as shown in the following proofs:

Lemma 6.2.1 (Existence of Sampling Region)**.** *For a graph G_S with the property that for all $v, v' \in V_S$, $d(v, v') > cl$, it must be true that $\forall v, v' \in V_S$ in different connected components, $\exists S \in C_{free}, ||S|| > 0$ where the closest two vertices are v, v'*

Proof: Conversely to Lemma 6.2.1 using Figure 6.2, if the clearance requirement is violated and $d(v, v') = 0$ such that two vertices v, v' in the graph represent the same state, there is a zero volume sized region S shared by a third vertex v'' in a different connected component whose only nearest neighbor is v'. No edge will be able to be added between v' and v'' because there is 0

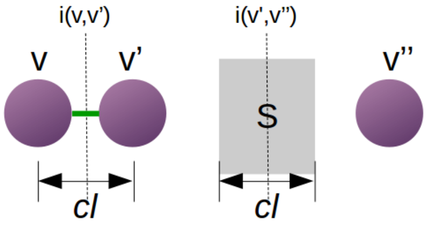


Figure 6.2: Example of how requiring a clearance *cl* between any two vertices allows for disconnected component v'' to become connected by sampling within region S.

probability a sample will be found that has its two nearest neighbors be both v' and v''. However, when the clearance requirement is met, there must be a region S whose nearest neighbors are in different connected components.

Lemma 6.2.2 (Sampling from S)**.** *Following Lemma 6.2.1, the probability we sample from S is 1 as the number of samples M goes to infinity for a finite c-space C.*

Proof: The probability that we do not sample from the region S after M samples is

$$\left(1 - \frac{||S||}{||C_{free}||}\right)^M$$

so the probability we do sample from S is

$$P(sampleS) = 1 - \left(1 - \frac{||S||}{||C_{free}||}\right)^M$$

and for any finite $||S|| > 0$, we know the limit as $M \to \infty$ is 1.

Theorem 6.2.3 (Connectivity)**.** *For all $v, v' \in V_S$ that are connected with a collision-free path in C_{free}, $\exists \pi_S(v, v')$ which connects them on G_S with probability 1 as M goes to infinity.*

Proof: Lemmas 6.2.1 and 6.2.2 have shown that even though we only check for connectivity between the two nearest neighbors that are also visible, we still maintain the ability to connect all disconnected subgraphs so long as a clearance $cl > 0$ is maintained between vertices. Therefore the sparse graph original proof for connectivity still holds, as presented in [43].

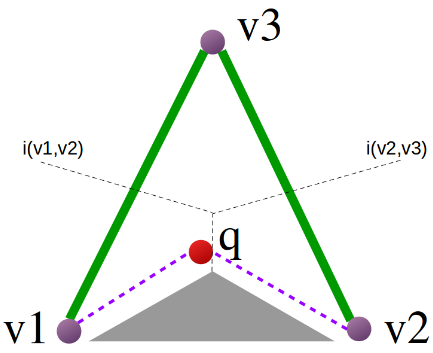


Figure 6.3: An example of a candidate sampled state q being considered to be added as an *interface vertex* between $[v1, v2]$. The length of the potential new edges $L(v1, q), L(q, v2)$ is first tested against the existing path $[v1, v3, v2]$ to check if they improve the path length more than a t-stretch factor.

6.2.3 Improved Interface Criterion

Another optimization required to apply the sparse roadmap criteria to our large c-space was reducing the number of unnecessary vertices and edges added to the roadmap via the *interface* criterion. An example for the following explanation is shown in Figure 6.3. In the original implementation, whenever an interface is detected between two vertices $[v_1, v_2]$ during random sampling of q, a connection is made—even if the vertices are not visible to each other. When the vertices are not visible, a *bridge* is formed by adding q to the roadmap and then adding edges $L(v_1, q), L(q, v_2)$. This behavior led to an observed huge growth in the number of states in the roadmap for complex, non-intuitive c-spaces. Borrowing the t-stretch factor from Sparse Roadmap Spanners (SPARS)'s quality criterion, we only accept interfaces bridges where the following is true:

$$d_{current} > t \cdot d_{new} \tag{6.1}$$

where $d_{current}$ is the shortest path currently on G_S and d_{new} is the length of the bridge. This logic is additionally shown in Algorithm 2. This modification greatly reduced the growth of the roadmap.

6.2.4 Simplified Sparse Roadmap Criteria

In the previous two sections, we described three simplifications to the asymptotically near-optimal criteria that distinguish our new approach from SPARS2. Together these changes make the sparse roadmap smaller in memory and faster to compute, and they are presented in Algorithm 1. We eliminated the *quality* criterion, leaving three criteria remaining: the *connectivity* criterion, the improved *interface* criterion, and the simplified *connectivity* criterion. By only using these three criteria, the frequency is greatly reduced in which a new sampled vertex must be collision checked against every other vertex in the graph. Once two nearby vertices are found to be visible to the newly sampled vertex, checking can be terminated. As the graph coverage increases, the probability is high that a visible vertex is nearby, so few collision checks are required.

This new simplified algorithm is comparable to V-PRM in how it rejects vertices based on visibility to all other graph vertices, but it is significantly different in how it adds edges. In V-PRM, a vertex is only added if 1) no other vertex is visible to it, or 2) the vertex is visible to at least two other vertices that belong to two distinct connected components on the roadmap. In our *relaxed sparse roadmap criteria*, we use a hybrid set of criteria borrowing from both V-PRM and SPARS2. A vertex is only added if no other vertex in the entire roadmap can locally connect to it, or if the vertex is required as a bridge to connect an interface. An edge is only added if the two closest vertices are also visible or if the two closest visible vertices are in different connected components. This has an unfortunate trade-off—this relaxed sparse roadmap criteria approach is no longer asymptotically near-optimal. While the remainder of the work in this chapter can still create sparse roadmaps with SPARS2, for the computational reasons just explained, we used the *relaxed sparse roadmap criteria* with positive results.

6.3 Bolt Experience-Based Planning Algorithms

We now explain two variants of Bolt experience planning, BoltDR and BoltSR, to motivate the answer to the question of whether full preprocessing is a viable approach for large c-spaces.

Algorithm 1 SparseRoadmapSpanner(M, t, Δ_{sparse})

```
$G_S$ ←InitializeGraph()
while $failures < M$ do                                   ▷ Termination condition
    $q$ ←SampleConfiguration()
    $W$ ←R-NearestNeighbors($\Delta_{sparse}$)
    $V$ ←FirstTwoVisible($W$)                             ▷ Visibility check
    if $V == \emptyset$ then                              ▷ Coverage Criterion
        $V_S \leftarrow V_S \cup \{q\}$                   ▷ AddVertex($q, G_S$)
    else if $|V| == 2 \wedge L(V[1], V[2]) \notin E_S$ then   ▷ Edge does not already exist
        $v_1 \leftarrow V[1]$
        $v_2 \leftarrow V[2]$
        if NotConnected($v_1, v_2$) then                  ▷ Connectivity Criterion
            if L($v_1, v_2$) $\in C_{free}$ then
                $E_S \leftarrow E_S \cup L(v_1, v_2)$     ▷ AddEdge($v_1, v_2, G_S$)
            else
                $V_S \leftarrow V_S \cup \{q\}$           ▷ AddVertex($q, G_S$)
                $E_S \leftarrow E_S \cup \{L(v_1, q), L(q, v_2)\}$   ▷ AddEdges($v_1, q, v_2, G_S$)
        else if $v_1 == W[1] \wedge v_2 == W[2]$ then     ▷ Interface Criterion
            AddInterface($q, v_1, v_2, t, G_S$)
    if *no change in* $G_S$ then
        $failures++$
return $G_S$
```

Algorithm 2 AddInterface(q, v_1, v_2, t, G_S)

```
$d_{curr} = distL^1$(AStar($G_S, v_1, v_2$))              ▷ Current length of path on $G_S$
$d_{new} = distL^1(v_1, v_2)$                             ▷ Direct distance between vertices
if $(d_{curr} == \emptyset \vee d_{curr} > t \cdot d_{new}) \wedge L(v_1, v_2) \in C_{free}$ then
    $E_S \leftarrow E_S \cup L(v_1, v_2)$                 ▷ AddEdge($v_1, v_2, G_S$)
else
    $d_{new} = distL^1(v_1, q) + distL^1(q, v_2)$         ▷ Bridge distance
    if $d_{curr} == \emptyset \vee d_{curr} > t \cdot d_{new}$ then
        $V_S \leftarrow V_S \cup \{q\}$                   ▷ AddVertex($q, G_S$)
        $E_S \leftarrow E_S \cup \{L(v_1, q), L(q, v_2)\}$   ▷ AddEdges($v_1, q, v_2, G_S$)
```

6.3.1 Bolt Dual Roadmap

The Bolt Dual Roadmap (BoltDR) algorithm altogether eliminates the need for the PFS component with the rationale that any building of a graph from scratch is inefficient when most environments are similar, and a robot with many constraints wastes a lot of time rejecting invalid samples. Instead, a copy of the experience roadmap is made for each planning instance—which we call the *task roadmap*—and further problem-specific modifications are applied directly to the copied roadmap. This separation of roadmaps allows the parent experience roadmap to only grow as new *trajectory solutions* are added back as specific experiences.

The task roadmap is searched for a solution to the planning query, checking if $[q_{start}, q_{goal}]$ has local paths onto the preprocessed experience roadmap. If no solution is found, it must be the case that changing environments have disabled parts of the graph. To repair the connectivity of the task roadmap, new samples are added to improve connectivity. The new samples are put through the same sparse roadmap criteria used for generating the experience roadmap with the same Δ_{sparse}. This prevents the graph from growing too large and further slowing down the search algorithm. In the absence of variable obstacles, for a fully preprocessed roadmap, no random samples should be possible to add. In reality, changing environments results in many parts of the preprocessed roadmap being disabled, at which point the random sampler is able to add additional vertices and edges to repair the missing coverage.

The random sampler used for BoltDR uses the simple heuristic for improving the planning time that, for manipulation problems, often the narrow passageways are near the start and goal states. Therefore, our sampler alternates between uniform sampling and sampling with a bias near those states.

The BoltDR algorithm alternates between checking for a valid path through the task roadmap and adding new samples, or alternatively two threads can parallelize these components as we did in our implementation. It should be noted that with proper bookkeeping, no actual copy of the graphs is required.

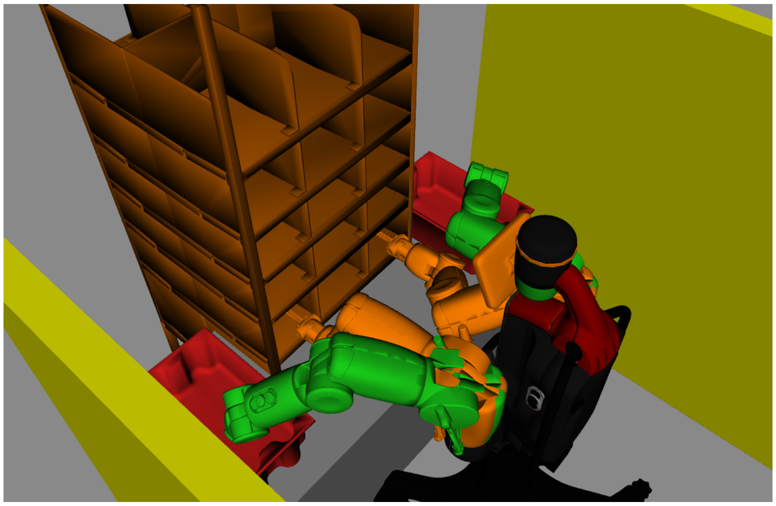


Figure 6.4: Experimental setup of dual-arm shelf picking where the green robot is the start state and the orange robot is one of the goal states. Three walls define the workcell of the robot, and a table and warehouse shelf are in front of the robot.

The task roadmap presented in this BoltDR algorithm is also useful for other applications: in Chapter 7 we use the task roadmap for more complex multi-modal planning problems that contain multiple goals. In these problems, multiple copies of the experience roadmap are added to the task roadmap, representing discrete mode changes.

6.3.2 Bolt Single Roadmap

The Bolt Single Roadmap (BoltSR) is an alternative to BoltDR that is essentially the Thunder algorithm presented in Chapter 4 in that it uses both a PFS and a RR component. The difference from Thunder is that instead of beginning with an empty roadmap and only saving the experiences that are actually used by the robot, BoltSR uses a preprocessed roadmap of all invariant constraints using the *simplified sparse roadmap criteria*. This allows the RR component to draw from an extensive prebuilt roadmap even for queries it has never seen before. No task roadmap is used and no modifications are done to the experience roadmap except when post-processing solution paths. For more details on the implementation of BoltSR, refer to the Thunder description.

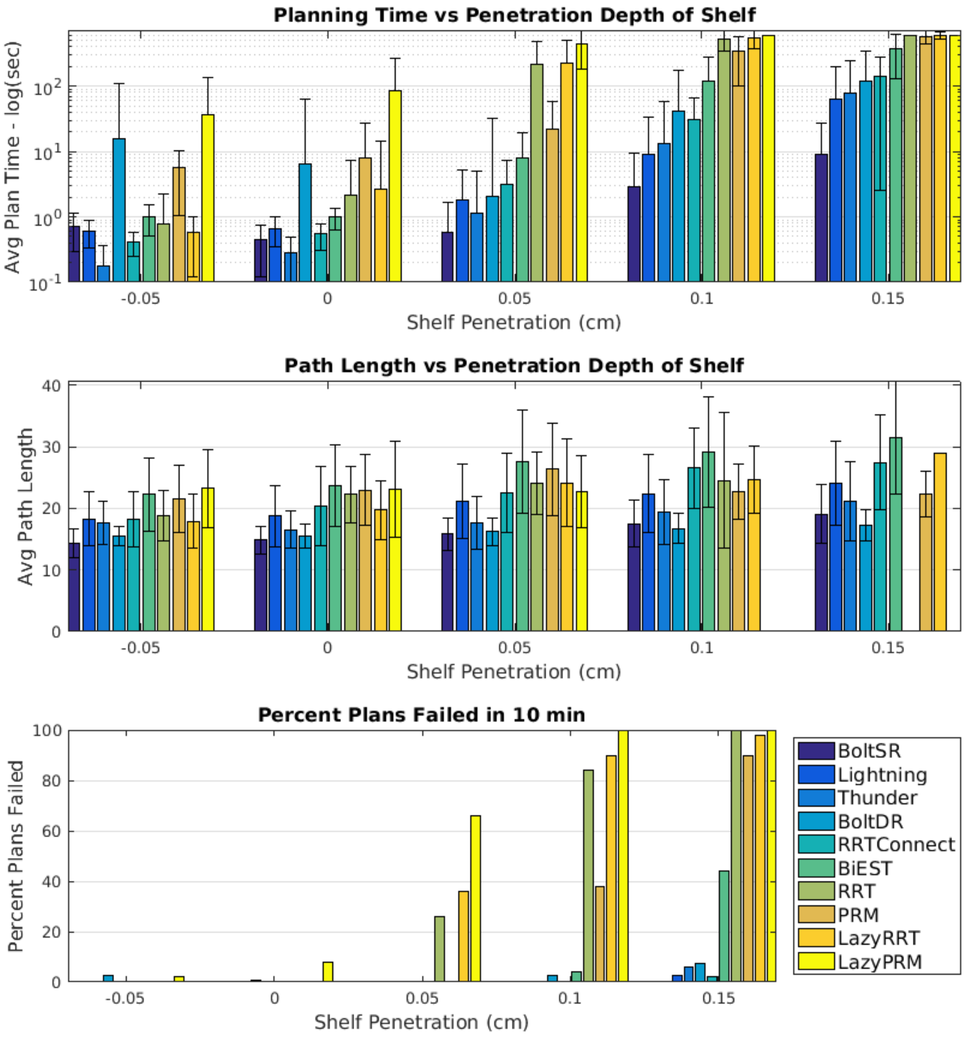


Figure 6.5: Average plan times, path lengths, and frequency of *no solution* result of various planners grouped by penetration depths into shelf (difficulty of narrow passageway)

6.4 Results

Our test environment, shown in Figure 6.4, is a small workcell with three surrounding walls, a table in front of the robot, two bins for placing objects, and a warehouse shelf as used in the Amazon Picking Challenge. This environment is highly constrained with many collision objects. We use the Rethink Robotics Baxter robot [140] and use the full 14 DOF dual-arms to simultaneously pick two

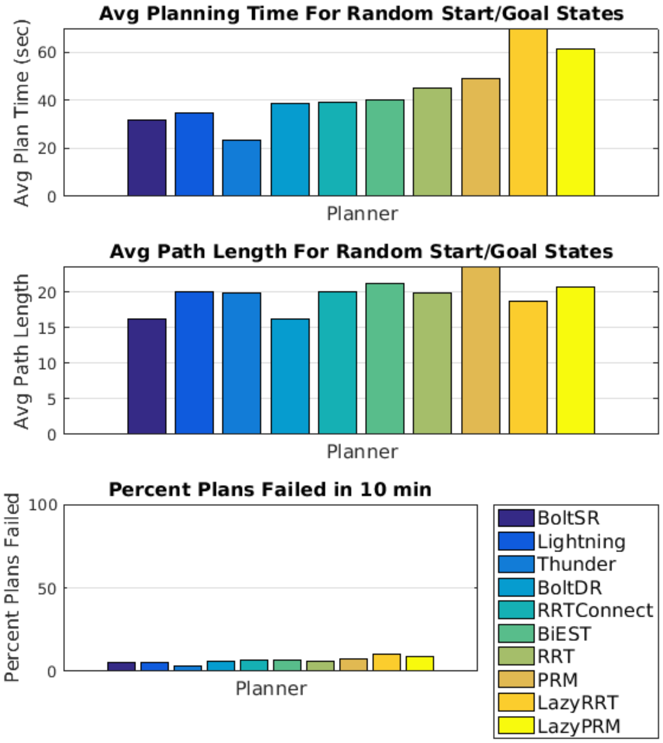


Figure 6.6: Average plan times, path lengths, and frequency of *no solution* result of various planners when planning between random start/goals. This benchmark demonstrated behavior when there are no common tasks/motions and narrow passageways are rare.

objects from a shelf. We simulate a real Amazon warehouse by adding noise to the exact location of the randomly placed shelf in front of Baxter, replicating dramatic calibration errors that might occur when Amazon Kiva robots drive the shelf to the picking robot. We use 3 cm of random noise in the X and Y positions of the shelf, and +/- 5 degrees of Z-axis rotational noise to the orientation of the shelf. This demonstrates our algorithm's ability to adapt and repair previous experiences to changing environments. MoveIt! was used as the framework for testing this setup.

The preprocessed roadmap is allowed 24 hours to generate a roadmap that encodes the invariant constraints of the described environment minus the shelf. All samples are required to have a 1.5 cm clearance. The roadmap is terminated after 2,473,000 sample attempts, and the resulting roadmap contains 4,882 vertices, 56,628 edges.

We generate unique goal locations by setting the pose of both end effectors aligned to the center of various compartments of the shelf using a randomly seeded inverse kinematic (IK) solver. For each goal pose, multiple redundant joint positions are added to the set of possible goal states. The *shelf penetration depth* sets how far into the shelf the end effectors reach, where a value of 0 places the tips of the parallel fingers vertically aligned with the lip of the shelf. This experimental variable allows us to approximate the difficulty of a narrow passageway problem: the deeper the two arms reach into the shelf the harder the problem becomes for a probabilistic sampling-based planner. We incremented this penetration depth in intervals of 5 cm, from -5 cm to 15 cm.

For 4 unique goal locations, we run 400 trials for each of the 5 shelf penetration depths, totaling 8000 trials per planner. Planners are given 10 minutes to solve each trial before being aborted. Post-processing smoothing was not included in the planning time, nor was the time required to re-integrate solutions into the experience roadmap, as this learning phase can be done offline, in the cloud, or when the robot is idle.

This benchmark for EBMP demonstrates both *free space* planning (henceforth our label for penetration depths of -5 cm, 0 cm) and varying levels of difficulty for *narrow passageway* planning (penetration depths of 5 cm, 10 cm, 15 cm). Traditional sampling-based methods have difficulty with the dimensionality of a dual-arm robot and the synchronized arm movements in confined spaces between Baxter's chest and the shelves. The required states to perform this motion have a low probability of being randomly sampled.

In Figure 6.5 we compare the BoltDR and BoltSR variants we have presented in this chapter to our first experienced-based planer Thunder and the original planner that inspired this work, Lightning. We also compare our planner to many popular sampling-based planners available in the Open Motion Planning Library (OMPL) including SPARS2, RRT-Connect, PRM, RRT, Lazy PRM, Lazy RRT, and Bi-directional Expansive Trees (BiEST) [66]. We include the lazy versions of PRM and RRT for a more fair comparison of those algorithms to ours since our planners also use lazy collision checking. All planners are allocated two threads for equal computational comparison; where necessary, two instances of the same planner were run in different threads during benchmarking. In this two instance case, when one of the threads finds a solution the other thread

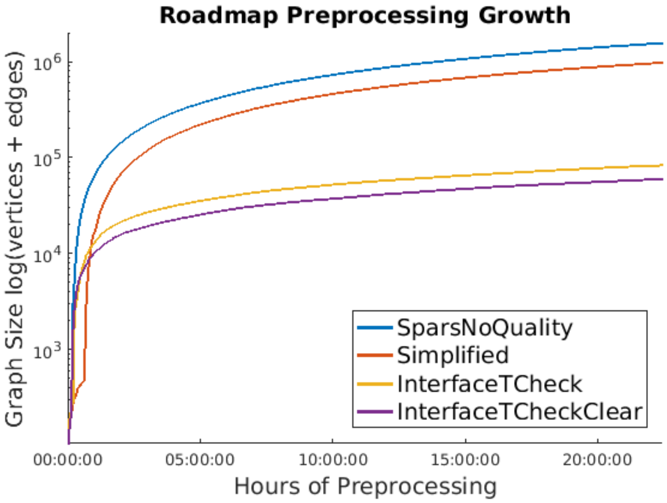


Figure 6.7: Logarithmic plot of growth of size (vertices + edges) of roadmap over time. *SparsNoQuality* is the original SPARS2 algorithm minus the quality criterion. *Simplified* is Algorithm 1 without the additional t-stretch factor test for the *interface* criterion. *InterfaceTCheck* is with the t-stretch factor test. *InterfaceTCheckClear* includes a 1.5 cm clearance.

is immediately terminated without any hybridization of solutions. Notably—the SPARS2 implementation available in OMPL was unable to solve any of our problems due to the size of the c-space and difficulty of the constraints.

In the top of Figure 6.5, we show the logarithmic average planning time of the planners against the various penetration depths, along with their corresponding standard deviation error bars. The middle of Figure 6.5 shows average path length of those plans, and the bottom shows the percent the planners fail within the allocated 10 minutes. Failed plans are excluded from the path length averages but included in the planning time averages.

An important takeaway from the bottom of Figure 6.5 is that when the problem is hard, PFS approaches cannot solve the presented problems, failing completely! This is a huge win for EBMP planners.

The secondary experiment in Figure 6.6 shows the average performance of the same planners and environment when planning between 250 random start/goal states. This benchmark demonstrates an adversarial problem for EBMP: efficient recall when no common task or motion is being performed. Only the workcell walls remain constant across the trials. This can be a challenge because the EBMP planner is unable to specialize in learning particular parts of the c-space, and datastructures have the risk of getting too large.

The comparison of roadmap preprocessing growth for variations of the sparse roadmap criteria is shown in Figure 6.7. It shows an order of magnitude improvement in roadmap size when using the t-stretch factor test for the *interface criterion*. An addition minor improvement is utilizing a small clearance criterion, which we set to 1.5 cm to nearest obstacle.

6.5 Discussion

Bolt Single Roadmap: In our results BoltSR—the approach of using a separate thread for a PFS planner—outperformed our previous EBMP planners and PFS planners in all narrow passageway problems showing, for example, a speedup of 8.6x against Thunder and a speedup of 15.8x against RRT-Connect. This is due to BoltSR's use of the best of both worlds: recall from a precomputed graph of invariant constraints and scratch planning. However, this same large roadmap actually hurts BoltSR in easy problems that other planners can solve quickly: in free space Thunder actually outperforms BoltSR with a speedup of 4x because Thunder has a much smaller experience roadmap to search through and collision check (only those experiences in which it has seen before). Still, Figure 6.5 shows that BoltSR performs similarly to other PFS planners, being outperformed only by RRT-Connect with a 1.7x times speedup in free space problems.

Additionally, the second benchmark (Figure 6.6) indicates that BoltSR still performs faster than all PFS planners even when there is no pattern or commonality in the motion plans being solved. BoltSR solves the random problem benchmark 18.8% faster than RRT-Connect. Preprocessing the roadmap makes BoltSR a faster planner in general motion planning, not just for specific tasks like reaching into a shelf.

Bolt Dual Roadmap: In all problem types of Figure 6.5, our BoltDR method—planning *without* a PFS component—performs slower than all other EBMP planners. In free space BoltDR is 95.4% slower than BoltSR, and for the most difficult narrow passageway problem BoltDR is 92.4% slower compared to BoltSR. This indicates that there are actually strong advantages of having a secondary thread running a planner like RRT-Connect that grows dual trees to bias exploration of the c-space. Still, for narrow passageway problems BoltDR outperformed all PFS planners, which demonstrates that pre-computing an entire roadmap is a viable improvement that has other advantages as we will see in Chapter 7.

The results in the second benchmark (Figure 6.6) show that for the general motion planning problem, BoltDR also outperforms all PFS planners but does worse than BoltSR, Thunder, and Lightning EBMP. Overall the advantages of avoiding a secondary PFS thread are not strong.

Path Length All of our EBMP planners—BoltSR, BoltDR, and Thunder—demonstrated an improved average path length vs PFS because of the underlying sparse graph datastructure that improves over time. Variance in the path length was also greatly improved for BoltSR and BoltDR, with roughly half the standard deviation in narrow passageways compared to RRT-Connect. This is an important property of our EBMP planners because it allows more predictable and reliable paths to be generated than traditional probabilistic methods.

Timeouts For the general motion planning problem in Figure 6.6, and also for most of our other benchmarks, a small number of unsolved planning queries are always present. It is possible the randomly generated start/goal state pairs are impossible to plan for; *probabilistic completeness* can only guarantee that a solution will be found if one exists as time goes to infinity. Likely some of the problems would have been solved if given more than the 10 minute timeout, but this limit is typical for probabilistic planning benchmarks.

Preprocessing The *simplified sparse graph* criteria has demonstrated the ability to grow a feasible sparse roadmap using the max visibility range that can be successfully used to solve general motion planning problems. The generated roadmap had good connectivity, at termination the roadmap had only 13 disjoint sets out of 4163 vertices (0.31 % of the vertices) and with even more time this number would decrease.

6.6 Chapter Summary

We have applied the *Bolt* approach presented in Chapter 5 to high DOF problems by simplifying and speeding up the criteria used for roadmap graph. We presented a new *relaxed sparse roadmap criteria* approach that, despite not being asymptotically near-optimal, in practice has attractive properties and uses. This relaxed approach is ideal for problems where roadmaps can get intractably large and computationally infeasible even when using the SPARS2 theoretical guarantees.

We applied our preprocessed roadmap to changing environments using two new variants, BoltDR and BoltSR, that avoid redundant roadmap growth around invariant constraints and learn from experience. We benchmarked these variants and compared to many state of the art alternative motion planning algorithms. BoltDR, lacking a PFS component, performed far worse than BoltSR and only had mediocre improvements over PFS planners—indicating it is not the best approach except in certain applications as discussed in Chapter 7.

BoltSR proved superior to all PFS planners and outperformed all EBMP planners when solving hard narrow-passageway problems and showed a 93% improvement over RRT-Connect. BoltSR is a very capable EBMP, and a viable approach, though we will show even better results by not preprocessing the roadmap in Chapter 8.

In the next chapter we apply our sparsely preprocessed experience roadmaps as a building block to a complex multi-modal underconstrained Cartesian planner.

Chapter 7

Multi-Modal Planning for Dual-Arm Underconstrained Cartesian Trajectories

In this chapter we apply our experience-based Bolt Dual Roadmap (BoltDR) algorithm from Chapter 6 to a multi-modal dual-arm underconstrained Cartesian path planning problem that is applicable to many complex manipulation and manufacturing applications. This work is a unified hybrid approach to the traditionally disparate tasks of planning along an underconstrained Cartesian path with infinite solutions and planning its corresponding free space approach and retreat motions. This is accomplished by adding an additional task dimension to the configuration space (c-space), duplicating the precomputed experience roadmap for the free space motion phases and generating the middle phase using a *underconstrained Cartesian graph generator*. Our approach avoids the commonly used naive method of guessing and checking different start and goal configurations for the Cartesian path that risks reaching invalid robot configurations. Our work in this chapter is demonstrated on a dual-arm robot in a difficult and overlapping whiteboard drawing task.

The three phase approach used in this chapter can be extended to any number of phases, within computation limits. The multi-modal aspect of this work is applicable to many classes of task planning problems. For example, grasping an object is typically accomplished with five phases: 1) free space planning 2) Cartesian approach planning 3) a grasp action 4) Cartesian retreat planning and 5) free space planning. Humanoid walking is another multi-modal example where the environmental contacts (left, right, or both feet) define each mode [61].

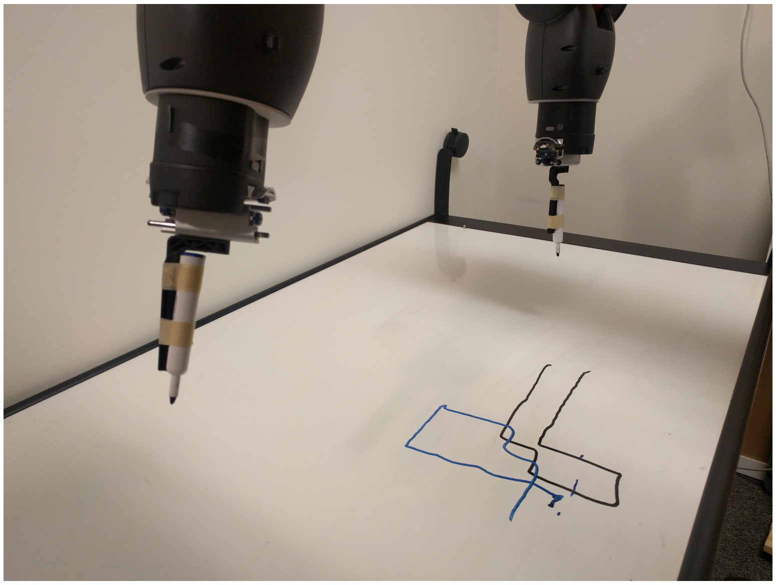


Figure 7.1: The Baxter Research Robot holding two whiteboard markers above an overlapping path it drew simultaneously with both arms.

7.1 Overview

The problem we solve in this chapter consists of three phases: two free space phases and one Cartesian phase. The phases sometimes are referred to as the "free transfer interval" and "contour following interval". During the free space phase, there are few constraints on the generated trajectory: it must simply avoid collision with obstacles and itself. During the underconstrained Cartesian phase, the trajectory has many constraints on it; at every step we define the desired location of the tip of the end effector with some tolerance for the roll, pitch, and yaw angles.

The objective of the first phase is to move from the robot's current state to the start state of the Cartesian trajectory. Because the Cartesian trajectory is underconstrained, there are actually many candidate goal states in the first phase. In the second phase the end effector searches its redundant inverse kinematic (IK) solutions to find the shortest distance collision-free path through the specified Cartesian waypoints. In the final phase the robot moves from the last state of the Cartesian trajectory to a resting pose, by planning a second free space plan.

With infinite solutions to an underconstrained Cartesian trajectory there are infinite potential start and goal states. However, due to obstacles in the environment, the kinematic limits of the robot, and singularities, some start and goal states can be unreachable. The naive approach to solving this problem is to randomly retry different start states for the Cartesian path. This is time-consuming and usually produces sub-optimal paths. In our approach we combine the three phases into one hybrid planning problem using a discrete *task* dimension, ensuring singularity avoidance for a given discretization.

Allowing *underconstrained* Cartesian planning is beneficial in that it greatly increases the size of a robot's reachable workspace. A fixed-base robot following a fully constrained trajectory limits the possible length of the path due to the physical constraints of the robot. However, when tolerances are allowed on the exact orientation of the end effector with respect to the workpiece being manipulated, the workspace of the robot greatly increases.

The focus of this chapter—similar to the entirety of this thesis—is kinematic planning. While real-world robotic operations typically require non-positional modes of control such as force or impedance control, a nominal kinematic plan is always required to ensure that the robot is capable of spatially performing the desired operation. Furthermore, a nominal plan ensures that other control modes remain stable by only allowing small deviations.

Additionally, we assume in this chapter the robot has 6 or 7 degrees of freedom (DOF) in each arm, such that there exists several (in the 6 DOF case) or infinite (in the 7 DOF case) solutions to any end effector pose within the workspace. We plan in the joint space which allows the algorithm to search utilize the null-space of pose constraints.

7.2 Method

7.2.1 Generation of Redundant Candidate Poses

In this chapter the input to our problem is a nominal 3D path π_{in}, specified as two sets of discretized waypoints—one for each arm—as shown in Figure 7.2a. Additionally specified are tolerances δ_{rpy} for each orientation axis (roll, pitch, and yaw) that can vary or remain constant

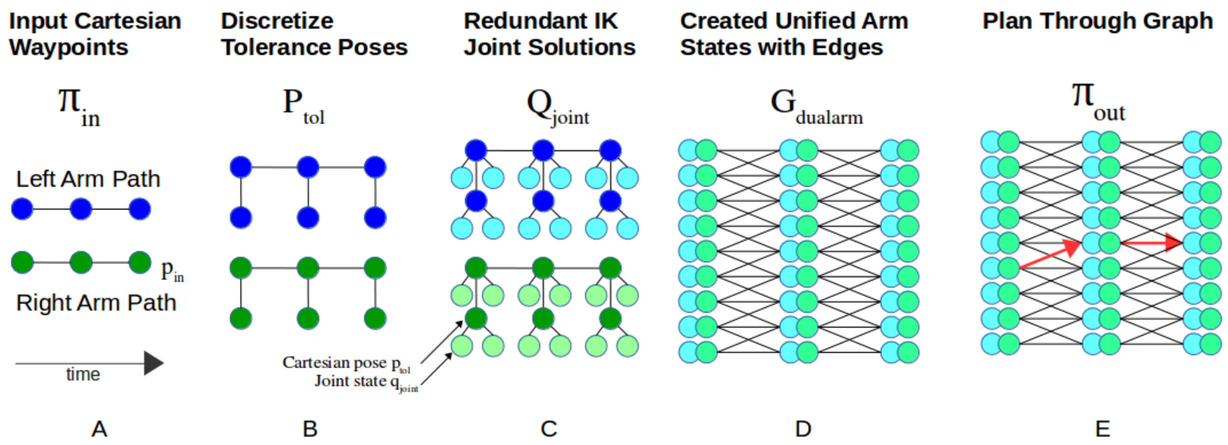


Figure 7.2: Steps for converting underconstrained Cartesian waypoints to a graph of redundant IK solutions for dual arms. This graph can then be searched through for the optimal motion given the discretization factor.

throughout the trajectory. In this section we generate candidate states independently for each arm.

First a set of candidate Cartesian *tolerance poses* P_{tol} is generated for each Cartesian pose p_{in} along the waypoint path by generating alternative poses within tolerance for a specified discretization factor d_{tol}. This is shown in Figure 7.2b. A conventional example is a robot holding a pointed tool with the z axis pointing to the work surface: the orientation of z does not effect the tip contact point, so many discretized tolerance poses can be generated along this axis.

Next, for each pose $p_{tol} \in P_{tol}$ an IK solver is used to find redundant joint solutions at a specified discretization factor d_{ik}. This is achieved by iteratively changing the joint value of the free joint. For every point in our nominal path π_{in} we should have many candidate joint solutions $Q_{monoarm}$ to achieve that point as shown in Figure 7.2c.

Every state $q_{joint} \in Q_{monoarm}$ will be used to populate vertices in a graph over which a shortest path search algorithm like A* runs. Although delaying collision checking using a technique such as lazy collision checking [14] is typically time-saving, in our case the cost of the number of potential edges that will be generated in our underconstrained Cartesian graph becomes the bottleneck, and it is important to prune as many vertices and edges ahead of time as possible. Therefore, the next step of our approach is to collision check the set of candidate joint solutions along the entire trajectory, for each arm.

7.2.2 Combining Dual-Arm Trajectories

Here we combine the set of candidate joint states $Q_{monoarm}$ of size N for each arm into a unified dual-arm set of joint states $Q_{dualarm}$. There are many approaches to doing this. The most complex could involve allowing the two arms to proceed along their arm trajectories at varying rates to allow avoidance of arm collisions by pausing one arm while the other passes. In this chapter, however, we assume both arms move in synchronization along the input nominal path π_{in}. This allows for the possibility that a poorly designed input path will have no solutions if it requires both arms be at overlapping spacial locations at the same time.

With the assumption that time variability is not allowed between the two arms, the most complete method for combining both arms is the quadratic with two arms method: for every candidate q_{joint} in one arm, combine with every candidate q_{joint} in the other arm. This results in N^2 candidate states for each nominal point along the input trajectory. For long paths, this exponential growth can be computationally intractable, so next we describe an alternative method for combining the two arms.

Rather than pair every candidate state from one arm with every candidate state from another, we develop a faster method that combines every state in one arm with a random state from the other arm. We repeat this random pairing a constant number of times $r << N$ where N is the number of $q_{joint} \in Q_{monoarm}$. The larger the value r, the smoother the generated trajectory will likely be at the cost of computation time. This method for combining two arms allows for significant reduction in graph size with the trade-off of more arm motion noise while executing the trajectory.

Finally, we run a secondary collision checking step with the unified dual-arm robot states to eliminate configurations that are in self-collision.

7.2.3 Generating an Underconstrained Cartesian Graph

The set of unified dual-arm Cartesian points $Q_{dualarm}$ is converted into a graph $G_{dualarm}$ to allow the shortest path to be searched as shown in Figure 7.2 D and E. For each point along the nominal path $p_i \in \pi_{in}$, every possible dual-arm solution is inserted into the graph $G_{dualarm}$ and the vertex set v_{p_i} for point p_i is saved. Next, edges are inserted between every pair of vertices between sets $[v_{p_i}, v_{p_{i-1}}]$. This is again an exponential operation, and the number of edges is the determining factor in the speed of A* search, so we present another pruning method to keep the size of the graph manageable.

Each discrete step along π_{in} is associated with a time step t to move the robot's joints. Each edge e connects two joint states of the robot, and the distance between each joint is measured as Δ_e. Given the robot's velocity limits V_{max} and the time step t, the max joint position change is calculated for every joint:

$$\Delta_{max} = t \cdot V_{max} \tag{7.1}$$

to determine if e is dynamically feasible to the first derivative. This velocity limit check removes a significant amount of dynamically infeasible motions between waypoints. We have now generated the *Cartesian graph*, next we combine it with free space planning.

7.2.4 Multi-Modal Planning with Free Space

Here we present the techniques developed to combine two free space roadmaps with the underconstrained Cartesian graph. The typical c-space normally only includes the 14 DOF dual-arm robot, each dimension being a continuous value bounded by the joint limits. In our multi-modal approach, we add an extra dimension that is a discrete integer value which indicates what step (or mode) the vertex is on during graph search.

The free space graphs were generated using the BoltDR experience planner presented in Chapter 5. It uses the precomputed sparse roadmap that makes it easier to duplicate it for the third free space phase, and allows connecting the middle-phase Cartesian graph to the first and

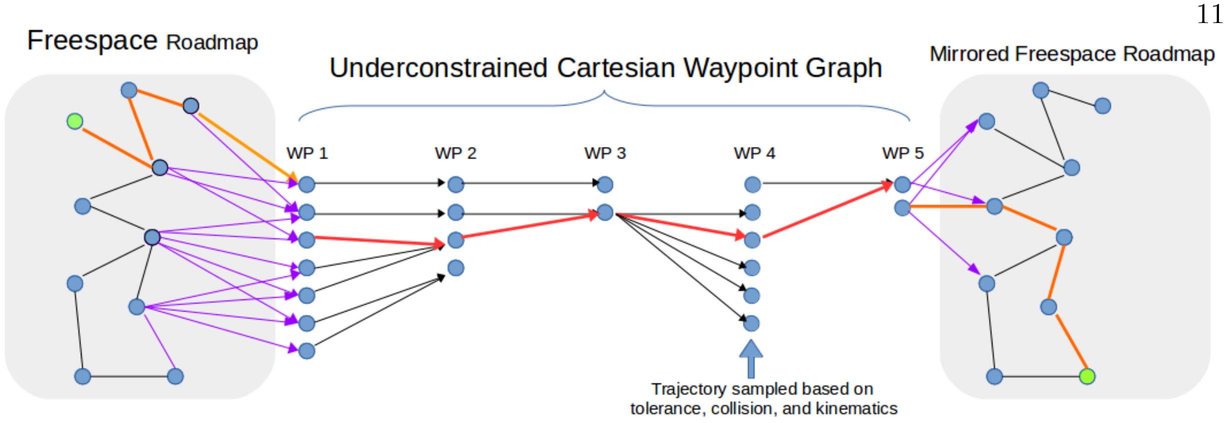

Freespace Roadmap
Underconstrained Cartesian Waypoint Graph
Mirrored Freespace Roadmap
WP 1
WP 2
WP 3
WP 4
WP 5
Trajectory sampled based on tolerance, collision, and kinematics

Figure 7.3: Method for combining a underconstrained Cartesian graph with two precomputed free space roadmaps using BoltDR. This allows the entire multi-modal planning problem to be solved in a unified and complete way.

third phases without sampling.

The *task roadmap* is used to contain the unified 3-mode planning problem. Only certain edges are encoded to be *transition* points in which a mode can change. These *transition edges* are populated by iterating through each start state and goal state of the Cartesian graph and finding the k nearest neighbors on the roadmap. If the motion from the start/goal states of the Cartesian graph to the nearest neighbors in the free space graph are collision-free, the edge is added. An overall demonstration of the unified graph planning problem is shown in Figure 7.3 where two free space roadmaps are shown connected to a Cartesian graph with transition edges.

7.2.5 Multi-Modal Heuristic Search

The unified task roadmap is searched using the classic heuristic-based A* approach. The heuristic must be admissible and requires special attention: in typical joint-based c-spaces, the straight-line Euclidean distance or L^1-distance is a sufficient approximation of the actual distance to goal. This does not work in our unified planning problem because the discrete component of our c-space results in indirect paths through the graph. Instead, we leverage a custom heuristic that sums the distances between each discrete mode, and a constant transition cost for mode changes, of the remaining path to the final goal state. In effect, we direct the A* search to solve multiple subgoals in one search instance.

Our heuristic distance function utilizes information gathered during the construction phase of the Cartesian graph. During construction, the shortest path between the set of start and goal vertices in the Cartesian graph is found, and the corresponding vertices $[S_{cart}, G_{cart}]$ are remembered. Additionally, an arbitrary transition cost $C_{tran} > 0$ is assigned to the transition edges to bias the A* search to make quicker progress across the multiple phases of the graph. A trade-off exists between the solution time (when using a larger C_{tran}) and solution quality (when using a smaller C_{tran}). A recommended value for C_{tran} is the maximum extent of the joint-based c-space, but further investigation of this value is needed.

For a state in the first mode the full heuristic function is:

$$\begin{aligned} dist_{total} &= dist(S_{global}, S_{cart}) + C_{tran} + \\ &= dist(S_{cart}, G_{cart}) + C_{tran} + \\ &= dist(G_{cart}, G_{global}) \end{aligned} \tag{7.2}$$

where S_{global} and G_{global} represent the current and final states of the robot, respectively. As the A* search progresses across the modes, this heuristic function reduces and simplifies removing the cost components that are no longer ahead of the current state.

7.3 Results

The unified planning problem was implemented with Open Motion Planning Library (OMPL) and MoveIt! as previously presented in this thesis, and the problems are tested on a Rethink Robotics Baxter robot [140]. A whiteboard drawing experiment verifies our method, starting with Baxter's arms initially being a significant distance apart holding two pens. Simple shapes are drawn on a whiteboard using both arms. The two drawn shapes are close together and at times overlapping as shown in Figure 7.4, demonstrating the ability to plan with two arms in overlapping work volumes. The orientation of the pens is allowed to change during execution of the two paths.

An example of a generated plan is shown in Figure 7.5, and the resulting drawing is shown

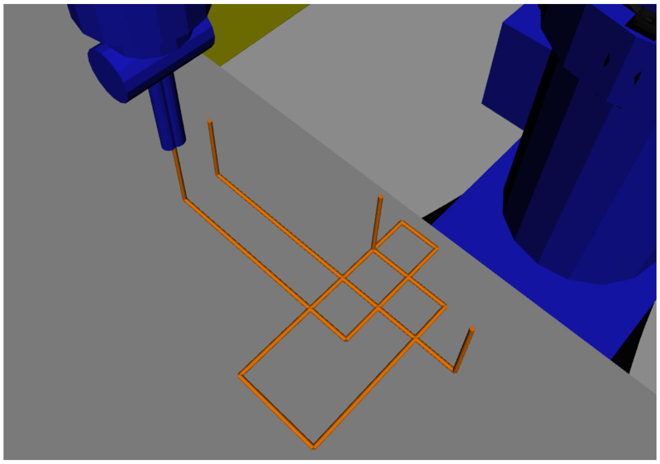


Figure 7.4: Input three-dimensional Cartesian path. Exact orientation of end effector pointing at orange path is underconstrained

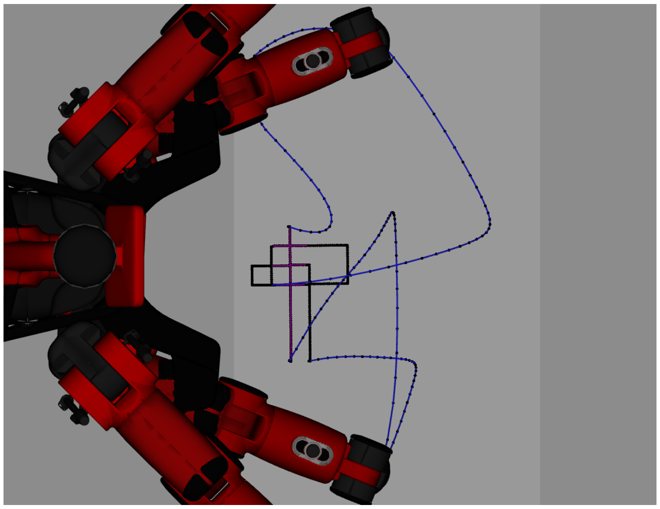


Figure 7.5: Birds-eye view of generated trajectory, transitioning from curved free space plan to geometric Cartesian plan

in Figure 7.1. We executed the complex dual arm drawing task on real hardware as shown in the supplemental video [1] . Planning time took on average 60 seconds.

[1] https://www.youtube.com/watch?v=Tw3qOeOAKlc

7.4 Chapter Summary

In this chapter we have shown a novel method for finding motion plans in complex environments with many constraints. Our constraints included collision checking two arms working in overlapping work volumes, avoiding arbitrary obstacle environments, following two underconstrained Cartesian paths with varying tolerances per rotation axis, and solving for two subgoals during the graph search. We verified our results on a physical robot that was able to draw our input shapes to a reasonable quality given lack of closed loop control.

Chapter 8

Design Considerations for the Thunder Framework

In this final chapter we present an improved version of the Thunder experience-based motion planning (EBMP) framework that demonstrates the best performance compared to all other approaches presented in this thesis and all conventional Planning from Scratch (PFS) planners we benchmarked. Additionally, we present two variants of planners—*Thunder-ERRT* and *BoltSR-ERRT*—that close the loop in the PFS component by feeding past experiences back into the randomly sampling planner. Finally, we demonstrate an implementation improvement to the Thunder planner that increases the speed of learning, which has a significant impact on planning time for our final benchmarking dataset.

8.1 ERRT-Connect

In this section we present an improved variant to the PFS component used in Thunder that has shown a 103x speedup in performance versus our original Thunder approach in narrow passageways. As suggested in Lightning [11], we utilize the RRT-Connect algorithm for its advantageous greedy approach to exploring the configuration space (c-space) by growing trees from both the start and goal states. Yet there are inefficiencies when the PFS component wastes time blindly sampling random states to grow the RRT-Connect tree. In the highly constrained environments we have been exploring, random samples are often invalid and must be rejected, and the probability of sampling useful states through narrow passageways is very low. We therefore present an improved variant of RRT-Connect that involves growing the dual trees using past states that have been saved in our experience roadmap as input samples. These saved states have already been shown to be valid for invariant constraints and have proven useful in past planning problems because otherwise they would not be in the experience roadmap. This new variant we call the Experience-RRT-Connect (ERRT-Connect) planner—not to be confused with Execution-extended RRT (ERRT) [20]. ERRT is similar in that it grows an Rapidly Exploring Random Tree (RRT) using states from the previous plan, but it does not use a full experience roadmap, bi-directional tree growing, nor nearest neighbor (NN) search for useful samples. As we will show in the results section, using past states as input samples increase the speed at which the tree grows for certain problems.

Further optimization to ERRT-Connect can occur when considering in what order to attempt inserting past states from the experience roadmap into the dual trees. We use a heuristic of attempting to insert the closest states, for some metric function, to the start and goal states of the tree. This is implemented by calling a NN search for both the start state and goal states. This heuristic encodes the assumption that the hardest areas of the tree to build are near the start and goal states. This is intuitive if considering pick and place tasks where reaching into a cluttered area (such as a table) and placing into another cluttered area (such as a shelf) is the most difficult part of the motion plan in contrast to free space planning during the translation phase.

A key property of the Thunder Framework is that it is probabilistically complete—this property is derived from the PFS RRT-Connect component which we have now modified. To maintain this important theoretical property, uniform random sampling must still be employed. In planning problems that have never been seen before, ERRT-Connect can actually reduce the solve time compared to using the original RRT-Connect algorithm. This is because the previous NN samples are not always beneficial to tree growth; they are generally only useful when solving problems with narrow passageways. For this reason, we alternate every other sample input into the RRT-Connect algorithm between being uniformly random or taken as a NN on the experience roadmap.

We call this new approach of utilizing ERRT-Connect as the PFS component within the Thunder framework the *Thunder-ERRT* variant. In addition we combine BoltSR with ERRT-Connect and call that method *BoltSR-ERRT*. The performances of these planners will be presented in Section 8.4.

As an aside, this experience-biased sampling approach is general enough to be applicable to most sampling-based motion planning algorithms, not just RRT-Connect. It is particularly applicable to algorithms that are bi-directional.

8.2 Faster Learning

In the benchmarking results presented in Chapter 6 and later in this chapter, we run each planner 8000 times in a complicated workspace to allow the EBMP algorithms to learn from experience and reuse plans. These trials took several days on a modern computer, and a significant amount of time was spent integrating the experiences back into the experience roadmap. A visualization of the reintegration of learned experiences into the roadmap is shown in Figure 8.1.

In our original Thunder implementation in Chapter 4, we presented a heuristic that ordered the insertion of states from a smoothed path in an ideal manner given the sparse roadmap criteria. This ordering facilitated the experience roadmap to have a high probability of creating the vertices and edges required to allow the same problem to quickly be solved again in the future. However, in implementation there was still a chance (about 3%) that the path would not be fully connected in the graph such that similar future problems could not be solved from recall but only from scratch. Eventually, if future similar problems were run, the experience graph would learn the path—but the learning was slow.

To further ensure an experience be fully inserted into the experience roadmap for future recall, we now add an iterative approach with increasing discretization. After every vertex in a solution path has been attempted to be added to the sparse roadmap, a simple connectivity test is employed to determine if the first and last states in the path are in the same connected component. If they are not yet connected, the solution path can be interpolated at a finer discretization amount, and then the path insertion process can be repeated. This iterative approach is shown to further improve start-goal connectivity in Table 8.1, which results in the planner solving problems with recall sooner.

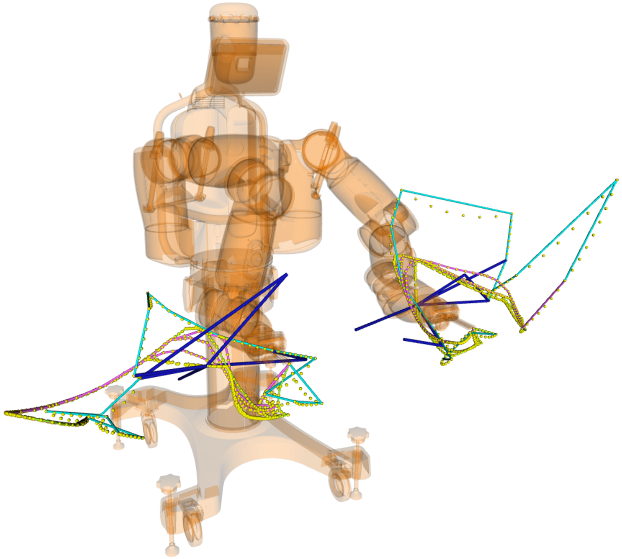


Figure 8.1: Baxter visualization of recalled paths (cyan) for two robot arms overlaid with the smoothed path (yellow) and the vertices that were added to the experience roadmap (dark blue).

In the results section we apply this improvement to *Thunder*, *Thunder-ERRT*, and *BoltSR-ERRT*, then compare it against the original Thunder algorithm.

8.3 Simplified Sparse Roadmap Criteria

Our new simplified sparse roadmap criteria presented in Chapter 6 also are applied to the *Thunder* and *Thunder-ERRT* planners, which result in a far smaller roadmap than the original Thunder planner that used the full Sparse Roadmap Spanners 2 (SPARS2) criteria. This is because the *quality* criterion in SPARS2 required many unnecessary edges and vertices be added in exchange for the asymptotically near-optimal guarantees. In the results section, we show that using this simplified criteria also greatly improved the performance of the algorithm.

8.4 Results

In this section we use the same experimental setup explained in Chapter 6, testing narrow passageway shelf picking problems and random start/goal free space planning. In this chapter we only compare against one PFS planner—RRT-Connect—since it was previously shown to be the

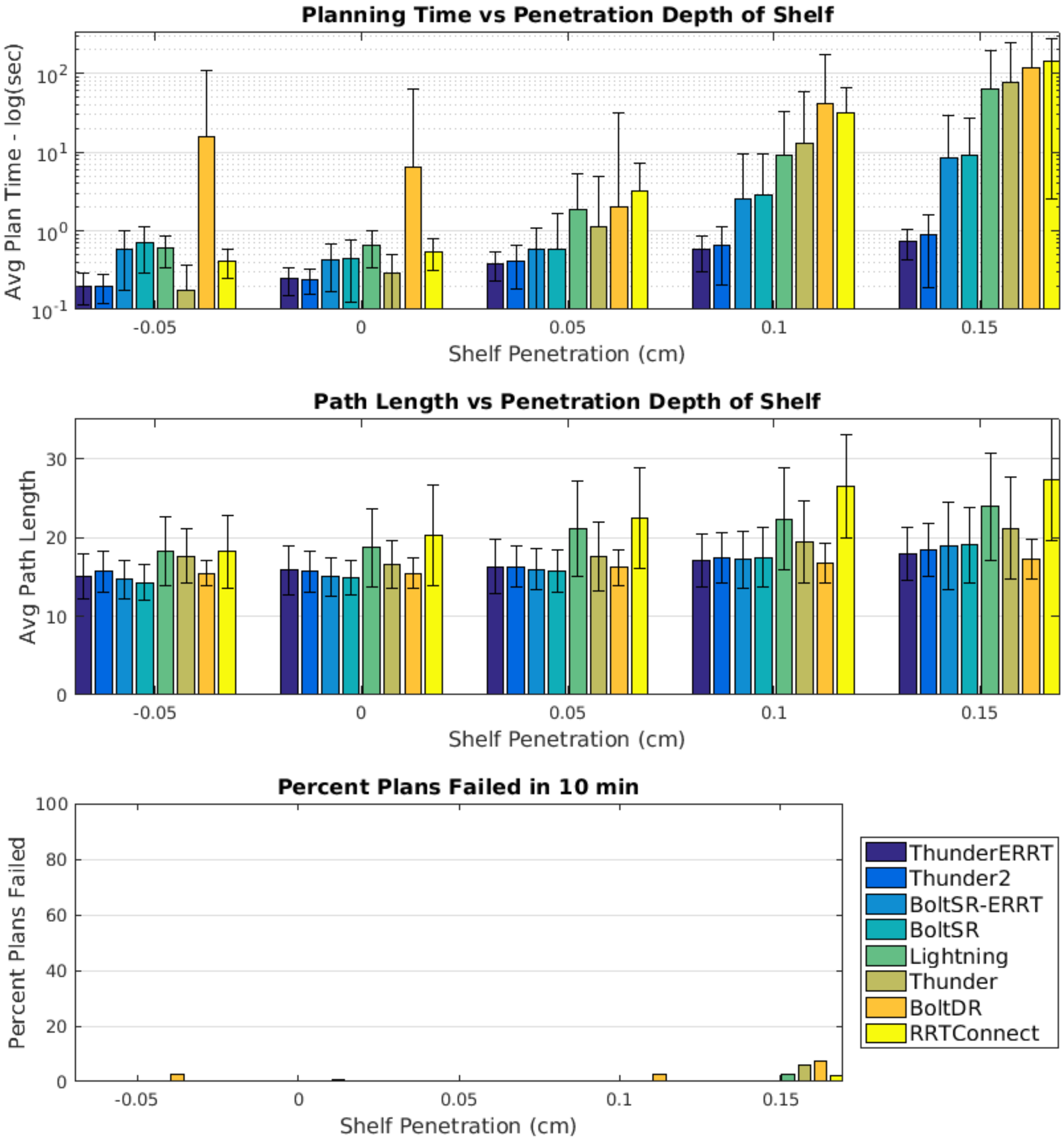


Figure 8.2: Average plan times, path lengths, and frequency of *no solution* result for various planners grouped by penetration depths into shelf (difficulty of narrow passageway)

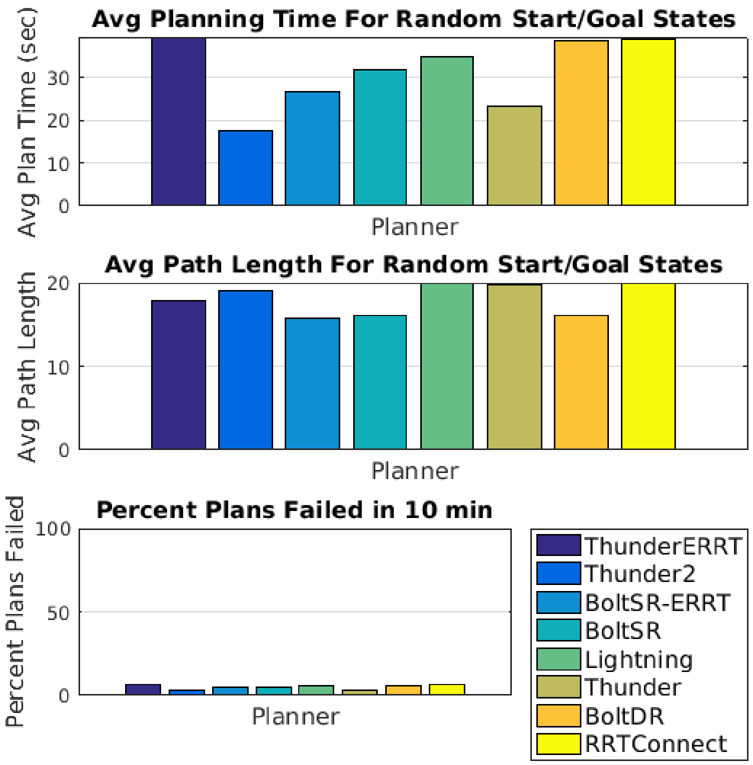


Figure 8.3: Average plan times, path lengths, and frequency of *no solution* result of various planners when planning between random start/goals. This benchmark demonstrated behavior when there are no common tasks/motions, and narrow passageways are rare.

best performing PFS planner we tested. We compare our new variants *Thunder*, *Thunder-ERRT*, and *BoltSR-ERRT* against our previous approaches: Bolt Single Roadmap (BoltSR), Bolt Dual Roadmap (BoltDR), *Thunder-SPARS*, and Lightning.

Thunder-ERRT: In Figure 8.2 the results of planning through narrow passageways in the shelf picking problem show that Thunder-ERRT is by far the best approach with a speedup of 103x from the original Thunder-SPARS. Compared to RRTConnect, Thunder-ERRT has a **189x speedup** (99.5% faster). However, this improvement is due to the ERRT-Connect heuristic presented in Section 8.1 that is based on the assumption that there are narrow passageways near the start and goal states. While true for many manipulation tasks, this is not always the case. The random start/goal state planning problem in Figure 8.3 demonstrates that the *ERRT-Connect* heuristic can actually hurt planning time, performing worse than all planners including RRT-Connect (but only by a small margin: 1.3% slower than RRTConnect).

Thunder: Our updated *Thunder* algorithm using the faster learning method and simplified sparse roadmap criteria is likely the best algorithm in this thesis, performing slightly slower than Thunder-ERRT for narrow passageway problems (21% slower) but much faster than Thunder-ERRT in the random start/goal state problem (55.6% faster). Given that heuristics are generally too difficult to properly apply to the general planning problem, the overall performance of Thunder across different types of planning problems is superior to Thunder-ERRT for most applications. In the narrow passageway problem, Thunder outperformed RRT-Connect with a speed up of **156.6x** and in the random start/goal problems, Thunder outperformed RRT-Connect with a speedup of 2.2x .

BSR-ERRT: For completeness, we also combined our Bolt graph preprocessing work in Chapter 6 with the *ERRT-Connect* variant presented in this chapter. There is a 6.7% improvement from the original BoltSR for narrow passageways and 15% improvement for the random start/goals problem. This improvement remained consistent across all shelf penetration depths, so while minor, the BSR-ERRT variant is superior to BoltSR with standard RRT-Connect and better than all other planning approaches except Thunder-ERRT and Thunder.

Utilization of Recall: In Figure 8.4 the frequency in which the recall planner solves the query faster than the PFS planner is plotted against time—as the planners learn how to reach into the shelves, the need for the PFS planner drops and solutions are solved from recall 95% of the time for the Thunder variants. For the Bolt variants the utilization is less, around 80%, which is likely because of the over head of searching a very large precomputed roadmap for a solution. The Lightning framework performs the worse, averaging 75%.

In Figure 8.5 the same plot is shown for the general planning problem and random start/goal states. This adversarial problem is evident in the experience planner's much lower utilization of recall—nearly half the usage of recall than in the shelf reaching task. It is probable that utilization would increase if more than 250 trials were run, but this was not feasible for workload of this

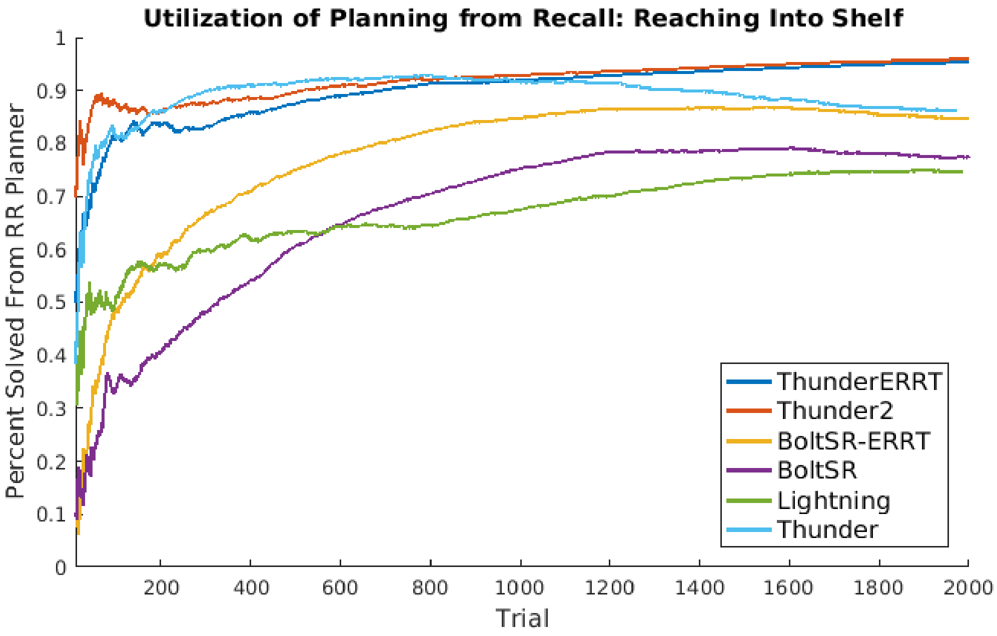


Figure 8.4: Frequency in which the recall planner solves the query faster than the PFS planner for reaching into shelf tasks.

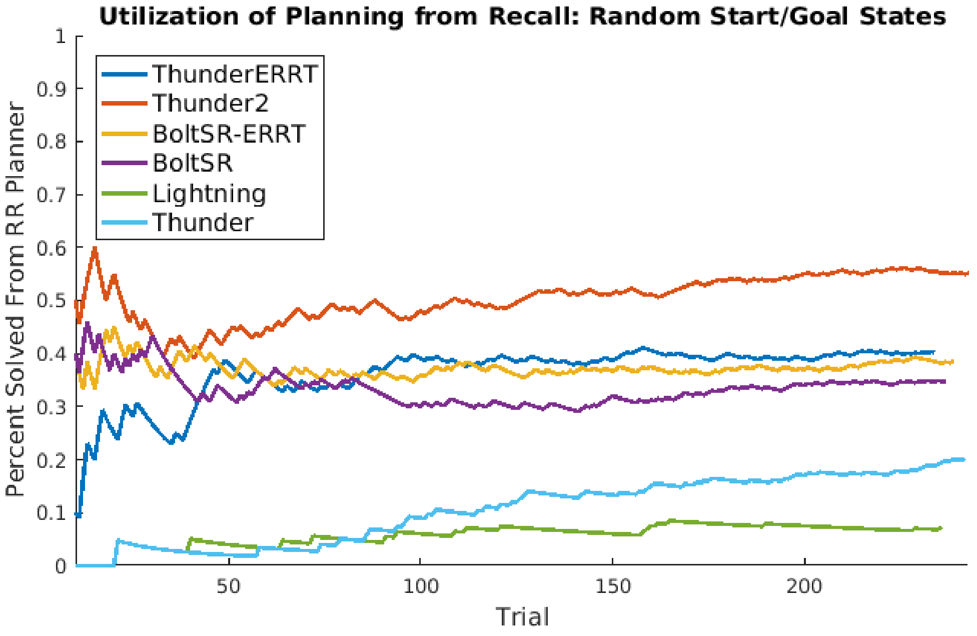


Figure 8.5: Frequency in which the recall planner solves the query faster than the PFS planner for random start/goal state queries.

benchmark. In this general planning problem the Thunder approach was by far the most successful at recall, and again Lightning was the worst.

Table 8.1: Frequency of Failed Path Insertion

ThunderERRT	3	1.28%
Thunder	1	0.41%
BoltSR-ERRT	1	0.42%
BoltSR	0	0.00%

Faster Learning Table 8.1 shows the failure rate of fully inserting solved paths back into the experience roadmap. The simplified sparse roadmap criteria is used to determine which pieces, if any, of the path will be saved into the graph. These results show there is a high probability—worst case 98.72% chance of success—that an experience will be learned on the first instance.

8.5 Chapter Summary

In this chapter we presented three implementation variants to our Bolt and Thunder algorithms that at best resulted in a speedup to 189x compared to RRT-Connect in the narrow passageway problem of reaching into a shelf. For the adversarial benchmark of completely random start/goal queries—where no pattern exists that helps EBMP planners specialize—our best planner still showed a 2.2x speedup compared to the fastest PFS planner RRT-Connect.

From these results we recommend the *Thunder* approach, as presented in this chapter, as the recommended experienced-based motion planner for all problems irrespective of narrow-passageway difficulty due to its ability to perform well in all situations without heuristics.

Chapter 9

Conclusion

This thesis has demonstrated that motion planning without taking advantage of past knowledge is far less computationally efficient than leveraging experiences—especially in highly constrained environments. Our approach of using sparse roadmaps has allowed memory to be efficiently used in saving experiences for large configuration spaces (c-spaces). We have presented two major variants to our approach: only saving experiences into the roadmap that have been seen before (Thunder), or fully preprocessing the invariant constraints into a large roadmap (Bolt). The Thunder approach results in faster query resolution times even for easy problems, while the preprocessing approach is advantageous for certain applications such as multi-modal problems.

To accomplish full preprocessing of high degrees of freedom (DOF) c-spaces, a *simplified sparse roadmap criteria* was presented that eliminated the memory and computation-intensive interface bookkeeping in Sparse Roadmap Spanners 2 (SPARS2), with the trade-off of no longer having asymptotically near-optimal guarantees. This simplified criteria removed the need for collision checking every sample against every vertex in the graph, allowing expedited graph growth. We also presented an improved *interface criterion* that reduced the growth of the graph such that added paths are within a t-stretch factor.

In Chapter 7 we applied our roadmap preprocessing techniques to multi-modal planning were a single graph can represent multiple tasks and be solve in a unified approach. This reduces the chances of planning into invalid configurations or hitting singularities in the multi-step process. Our application was the problem of dual-arm underconstrained Cartesian planning, and we additionally presented our results combining two overlapping arm trajectories into a single graph and exploiting the redundancy of the 7 DOF robotic arms and the tolerance of the nominal waypoint paths to find a valid solution through the graph. Optimizations were presented for keeping this large graph search problem tractable.

We demonstrated our approaches on two platforms: a whole-body 30 DOF humanoid with stability constraints and a dual-arm 14 DOF fixed-based robot reaching into confined shelf compartments. In these benchmarks we demonstrated a speedup of 189x (99.4% faster) in difficult motion planning problems compared to RRT-Connect while still achieving a modest speedup of 2.2x for random free space planning problems.

9.1 Future Work

A future improvement for the work presented in this thesis is allowing multiple task or environment-specific roadmaps to be stored and recalled. Each roadmap would be optimized for the particular task domain, e.g. working in a kitchen, holding a cup of water upright, or fulfilling warehouse orders. The challenge in adding this functionality is how to identify the best partitioning of experience roadmaps across infinite tasks and environments. High level semantic data of the environment could be used, perhaps from a machine learning image processing pipeline. Another approach could involve distributing a set of spheres in strategic locations within the robot's reachable workspace that would each be collision checked and used as a fingerprint for which roadmap to utilize.

For the humanoid planning demonstrations, we would like to improve our setup to allow the fixed foot to change, allowing shuffling feet motions more useful for manipulation. Transitioning between the three modes of environment contact (left foot, right foot, both feet) will require our multi-modal work be applied to the humanoid problem. Full walking, particularly for uneven terrain, may also be possible but will likely require a hybrid approach to address the problem of expansive c-spaces.

In the Thunder Framework we would like to add the ability to handle dynamics in nonholonomic systems and account for more optimization objectives beyond path length.

In our Bolt approach we would like to find a faster approach to generating a SPARS graph that does not require us, in high c-spaces, to relax the asymptotic near-optimality guarantees. More investigation remains on the effects of our relaxed sparse graph criteria.

9.2 Concluding Remarks

After years of applying motion planning to real world robots including the Willow Garage PR2, Rethink Robotics Baxter, Kawada Industries HRP2, Kinova Jaco2, and Kuka IIWA, I believe a great deficit in motion planning frameworks like MoveIt!, and the motion planning field in general, is a lack of consideration and integration of closed-loop, reactive control-based approaches. This thesis is no exception to the trend of motion planning researchers ignoring the realities of noisy sensors and actuators and the effects of interaction of a robotic manipulator to physical objects. While it has not been the main focus of my work presented in this thesis (with the exception of the related underconstrained Cartesian efforts presented in Chapter 7), I believe the integration of motion planning with reactive control is a much needed area of further investigation. Additionally, tight integration with perception pipelines is important and a common failure point in competitions like the Amazon Picking Challenge [37].

Outside observers sometimes have considered the field of motion planning to be solved, but many difficult problems still remain in balancing the limitations of today's computing power with the ever-increasing demands of what humans expect from robots. Desires for high DOF humanoid robots to operate in unstructured ever-changing environments in complex multi-task manipulation tasks are still a challenge largely unsolved. Many remaining problems involve integrating motion planning with the multitude of other disparate components of robotics across the domains of computer science, controls, and mechanical systems. Coordinating this complexity into a cohesive reliable system is a monumental task insurmountable by any single human.

One of the goals of the MoveIt! Motion Planning Framework is to bring together many algorithmic components of robotic motion planning and connect them to perception pipelines and controllers of physical hardware. Progressing open source platforms for working together with roboticists around the world is an incredibly important undertaking to advance the state of the art in robotics. While I am not the main author of the MoveIt! framework, I have greatly enjoyed learning from and being a part of the project.

For frameworks like MoveIt! to be successful, they also must focus on the software's ease of use, as presented in Chapter 3. When robotics research is the priority, it is understandable that spending time on tangential aspects of the project, such as GUIs and configuration tools, can be less important to the researcher. Still, I would like to encourage researchers and developers alike, when possible, to spend the additional time making their work reusable by taking into consideration the barriers to entry that other users might encounter. Too often, software is touted as "open source" when its usefulness in actuality is severely limited by the difficulties users encounter in applying it to other robotic projects. By sharing accessible open source robotics software, the progress of robotic technology is accelerated, and the robotics community as a whole benefits.

Bibliography

[1] ISO/PAS 17506:2012 industrial automation systems and integration – collada digital asset schema specification for 3d visualization of industrial data. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=59902`, oct 2013.

[2] P Agarwal. Compact representations for shortest-path queries. In IROS Workshop on Progress and Open Problems in Motion Planning, 2011.

[3] Sandip Aine, Siddharth Swaminathan, Venkatraman Narayanan, Victor Hwang, and Maxim Likhachev. Multi-heuristic A*. The International Journal of Robotics Research, 35(1-3):224–243, 2016.

[4] R. Alami, J. P. Laumond, and T. Siméon. Two manipulation planning algorithms. In Proceedings of the Workshop on Algorithmic Foundations of Robotics, WAFR, pages 109–125, Natick, MA, USA, 1995. A. K. Peters, Ltd.

[5] Nancy M Amato, O Burchan Bayazit, Lucia K Dale, Christopher Jones, and Daniel Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. IEEE Transactions on Robotics and Automation, 16(4):442–447, 2000.

[6] Robert O Ambrose, Hal Aldridge, R Scott Askew, Robert R Burridge, William Bluethmann, Myron Diftler, Chris Lovchik, Darby Magruder, and Fredrik Rehnmark. Robonaut: NASA's space humanoid. Intelligent Systems and their Applications, IEEE, 15(4):57–63, 2000.

[7] Christopher G Atkeson and Jun Morimoto. Nonparametric representation of policies and value functions: A trajectory-based approach. 2002.

[8] Giorgio Ausiello, Giuseppe F Italiano, Alberto Marchetti Spaccamela, and Umberto Nanni. Incremental algorithms for minimal length paths. Journal of Algorithms, 12(4):615–638, 1991.

[9] Nora Ayanian, James Keller, D Cappelleri, and Vijay Kumar. Development of a successful open-ended robotics design course at the high school level. Computer Education J, 1(3):21–31, 2010.

[10] Jerome Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. The International Journal of Robotics Research, 10(6):628–649, 1991.

[11] D. Berenson, P. Abbeel, and K. Goldberg. A robot path planning framework that learns from experience. In Robotics and Automation (ICRA), 2012 IEEE International Conference on, May 2012.

[12] D. Berenson, S. S. Srinivasa, D. Ferguson, and J. J. Kuffner. Manipulation planning on constraint manifolds. In 2009 IEEE International Conference on Robotics and Automation, pages 625–632, May 2009.

[13] Dmitry Berenson, Joel Chestnutt, Siddhartha S Srinivasa, James J Kuffner, and Satoshi Kagami. Pose-constrained whole-body planning using task space region chains. In 2009 9th IEEE-RAS International Conference on Humanoid Robots, pages 181–187. IEEE, 2009.

[14] Robert Bohlin and EE Kavraki. Path planning using lazy prm. In Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on, volume 1, pages 521–528. IEEE, 2000.

[15] Kevin J Boudreau. Let a thousand flowers bloom? An early look at large numbers of software app developers and patterns of innovation. Organization Science, 23(5):1409–1427, 2012.

[16] C. Bowen, G. Ye, and R. Alterovitz. Asymptotically optimal motion planning for learned tasks using time-dependent cost maps. Automation Science and Engineering, IEEE Transactions on, PP(99):1–12, 2014.

[17] Tim Bradshaw. Intel chief raises doubts over moore's law. Financial Times, July 2015.

[18] Michael S Branicky, Ross A Knepper, and James J Kuffner. Path and trajectory diversity: Theory and algorithms. In Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on, pages 1359–1364. IEEE, 2008.

[19] R.A. Brooks and T. Lozano-Perez. A subdivision algorithm in configuration space for findpath with rotation. Systems, Man and Cybernetics, IEEE Transactions on, SMC-15(2):224–233, March 1985.

[20] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on, volume 3, pages 2383–2388. IEEE, 2002.

[21] Davide Brugali, Walter Nowak, Luca Gherardi, Alexey Zakharov, and Erwin Prassler. Component-based refactoring of motion planning libraries. In Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, pages 4042–4049. IEEE, 2010.

[22] Herman Bruyninckx. Open robot control software: the orocos project. In Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on, volume 3, pages 2523–2528. IEEE, 2001.

[23] Tim. Caldwell, David. Coleman, and Nikolaus Correll. Optimal parameter identification for discrete mechanical systems with application to flexible object manipulation. In Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on, pages 898–905, Sept 2014.

[24] Tim Caldwell, David Coleman, and Nikolaus Correll. Robotic manipulation for identification of flexible objects. International Symposium on Experimental Robotics, 2014.

[25] John Canny. The complexity of robot motion planning. MIT press, 1988.

[26] John M Carroll. Interfacing thought: Cognitive aspects of human-computer interaction. The MIT Press, 1987.

[27] Sachin Chitta, E Gil Jones, Matei Ciocarlie, and Kaijen Hsiao. Perception, planning, and execution for mobile manipulation in unstructured environments. IEEE Robotics and Automation Magazine, 19(2):58–71, 2012.

[28] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. Principles of robot motion: theory, algorithms, and implementation. MIT press, 2005.

[29] Benjamin Cohen, Sachin Chitta, and Maxim Likhachev. Search-based planning for dual-arm manipulation with upright orientation constraints. In Robotics and Automation (ICRA), 2012 IEEE International Conference on, pages 3784–3790. IEEE, 2012.

[30] Benjamin Cohen, Ioan A Şucan, and Sachin Chitta. A generic infrastructure for benchmarking motion planners. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 589–595. IEEE, 2012.

[31] Benjamin J Cohen, Sachin Chitta, and Maxim Likhachev. Search-based planning for manipulation with motion primitives. In Robotics and Automation (ICRA), 2010 IEEE International Conference on, pages 2902–2908. IEEE, 2010.

[32] David Coleman and Nikolaus Correll. Sparser sparse roadmaps. arXiv preprint arXiv:1610.07671, 2016.

[33] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a MoveIt! case study. Best Practices in Robot Software Development, Journal of Software Engineering for Robotics, 2014.

[34] David Coleman, Ioan A Şucan, Mark Moll, Kei Okada, and Nikolaus Correll. Experience-based planning with sparse roadmap spanners. In 2015 IEEE International Conference on Robotics and Automation (ICRA), pages 900–905. IEEE, 2015.

[35] Toby HJ Collett, Bruce A MacDonald, and Brian P Gerkey. Player 2.0: Toward a practical robot programming framework. In Australasian Conference on Robotics and Automation, 2005.

[36] N. Correll, R. Wing, and D. Coleman. A one-year introductory robotics curriculum for computer science upperclassmen. Education, IEEE Transactions on, 56(1):54–60, 2013.

[37] Nikolaus Correll, Kostas E Bekris, Dmitry Berenson, Oliver Brock, Albert Causo, Kris Hauser, Kei Okada, Alberto Rodriguez, Joseph M Romano, and Peter R Wurman. Lessons from the amazon picking challenge. Robotics: Science and Systems XII, 2016.

[38] Ioan A. Şucan and Sachin Chitta. Moveit! `http://moveit.ros.org/`, October 2013.

[39] Debadeepta Dey, Tian Yu Liu, Martial Hebert, and J Andrew Bagnell. Contextual sequence prediction with application to control library optimization. Robotics, page 49, 2013.

[40] Rosen Diankov. Ikfast: The robot kinematics compiler. `http://openrave.org/docs/latest_stable/openravepy/ikfast/\#ikfast-the-robot-kinematics-compiler`, October 2013.

[41] Rosen Diankov and James Kuffner. Openrave: A planning architecture for autonomous robotics. Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, page 79, 2008.

[42] A Dobson and K.E. Bekris. Improving sparse roadmap spanners. In Robotics and Automation (ICRA), 2013 IEEE International Conference on, pages 4106–4111, May 2013.

[43] Andrew Dobson and Kostas E. Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. The International Journal of Robotics Research, 33(1):18–47, 2014.

[44] Andrew Dobson and Kostas E. Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. Int. Journal Robotics Research, 33(1):18–47, January 2014.

[45] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. Journal of the ACM (JACM), 40(5):1048–1066, 1993.

[46] Stefan Edelkamp. Updating shortest paths. In Proceedings of the European Conference on Artificial Intelligence, pages 655–559, Brighton, UK, 1998. Citeseer.

[47] S. Edwards, D. Solomon, J. Nicho, C. Lewis, P. Hvass, J. Zoss, and C. Flannigan. Descartes: Cartesian path planner inter-face pipeline. Github, 2014.

[48] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with rrts. In Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on, pages 1243–1248. IEEE, 2006.

[49] Dave Ferguson and Anthony Stentz. Field d*: An interpolation-based path planner and replanner. In Robotics Research, pages 239–253. Springer, 2007.

[50] Esteban Feuerstein and Alberto Marchetti-Spaccamela. Dynamic algorithms for shortest paths in planar graphs. Theoretical Computer Science, 116(2):359–371, 1993.

[51] Stefan Fischer, Manfred Hallschmid, Anna Lisa Elsner, and Jan Born. Sleep forms memory for finger skills. Proceedings of the National Academy of Sciences, 99(18):11987–11991, 2002.

[52] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In Proceedings of the 2002 ACM symposium on Document engineering, pages 26–33. ACM, 2002.

[53] Emilio Frazzoli, Munther A Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. Journal of Guidance, Control, and Dynamics, 25(1):116–129, 2002.

[54] Wilbert O Galitz. The essential guide to user interface design: an introduction to GUI design principles and techniques. Wiley, 2007.

[55] Willow Garage. URDF: Universal Robotic Description Format. `http://wiki.ros.org/urdf`, Oct 2013.

[56] Roland Geraerts and Mark H Overmars. A comparative study of probabilistic roadmap planners. In Algorithmic Foundations of Robotics V, pages 43–58. Springer, 2004.

[57] Michael A Goodrich and Dan R Olsen Jr. Seven principles of efficient human robot interaction. In 2003 IEEE International Conference on Systems, Man and Cybernetics, volume 4, pages 3942–3948. IEEE, 2003.

[58] Z. Y. Guo and T. C. Hsia. Joint trajectory generation for redundant robots in an environment with obstacles. In Proceedings., IEEE International Conference on Robotics and Automation, pages 157–162 vol.1, May 1990.

[59] Luc Guyot, Nicolas Heiniger, Olivier Michel, and Fabien Rohrer. Teaching robotics with an open curriculum based on the e-puck robot, simulations and competitions. In Proceedings of the 2nd International Conference on Robotics in Education (RiE 2011), 2011.

[60] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. Systems Science and Cybernetics, IEEE Transactions on, 4(2):100–107, July 1968.

[61] K. Hauser, T. Bretl, and J. C. Latombe. Non-gaited humanoid locomotion planning. In 5th IEEE-RAS International Conference on Humanoid Robots, 2005., pages 7–12, Dec 2005.

[62] Kris Hauser, Timothy Bretl, Kensuke Harada, and Jean-Claude Latombe. Using motion primitives in probabilistic sample-based planning for humanoid robots. In Algorithmic foundation of robotics VII. Springer, 2008.

[63] Kris Hauser, Victor Ng-Thow-Hing, and Hector Gonzalez-Banos. Robotics Research: The 13th International Symposium ISRR, chapter Multi-modal Motion Planning for a Humanoid Robot Manipulation Task, pages 307–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[64] David Hsu, Lydia E. Kavraki, Jean-Claude Latombe, Rajeev Motwani, and Stephen Sorkin. On finding narrow passages with probabilistic roadmap planners. In Proceedings of the Third Workshop on the Algorithmic Foundations of Robotics on Robotics : The Algorithmic Perspective: The Algorithmic Perspective, WAFR '98, pages 141–153, Natick, MA, USA, 1998. A. K. Peters, Ltd.

[65] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. The International Journal of Robotics Research, 21(3):233–255, 2002.

[66] David Hsu, J-C Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on, volume 3, pages 2719–2726. IEEE, 1997.

[67] S. Hutchinson, G.D. Hager, and P.I. Corke. A tutorial on visual servo control. Robotics and Automation, IEEE Transactions on, 12(5):651–670, 1996.

[68] Yong K Hwang and Narendra Ahuja. Path planning using potential field representation. In 1988 Orlando Technical Symposium, pages 481–489. International Society for Optics and Photonics, 1988.

[69] L. Jaillet and J. M. Porta. Path planning under kinematic constraints by rapidly exploring manifolds. IEEE Transactions on Robotics, 29(1):105–117, Feb 2013.

[70] Nikolay Jetchev and Marc Toussaint. Trajectory prediction: learning to map situations to robot trajectories. In Proceedings of the 26th annual international conference on machine learning. ACM, 2009.

[71] Nikolay Jetchev and Marc Toussaint. Trajectory prediction in cluttered voxel environments. In Robotics and Automation (ICRA), 2010 IEEE International Conference on, pages 2523–2528. IEEE, 2010.

[72] Nikolay Jetchev and Marc Toussaint. Fast motion planning from experience: trajectory prediction for speeding up movement generation. Autonomous Robots, 34(1-2):111–127, 2013.

[73] Xiaoxi Jiang and Marcelo Kallmann. Learning humanoid reaching tasks in dynamic environments. In Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on, Oct 2007.

[74] Rico Jonschkowski and Oliver Brock. Learning task-specific state representations by maximizing slowness and predictability. In ERLARS 2013 - 6th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems, 2013.

[75] M. Waleed Kadous, Raymond Ka-Man Sheh, and Claude Sammut. Effective user interface design for rescue robotics. In Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction, HRI '06, pages 250–257, New York, NY, USA, 2006. ACM.

[76] M Kallman and Maja Mataric. Motion planning using dynamic roadmaps. In Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on, volume 5, pages 4399–4404. IEEE, 2004.

[77] Noriyuki Kanehira, TU Kawasaki, Shigehiko Ohta, T Ismumi, Tadahiro Kawada, Fumio Kanehiro, Shuuji Kajita, and Kenji Kaneko. Design and experiments of advanced leg module (hrp-2l) for humanoid robot (hrp-2) development. In Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on, volume 3, pages 2455–2460. IEEE, 2002.

[78] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research, 30(7):846–894, 2011.

[79] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research, 30(7):846–894, 2011.

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

[81] Lydia E. Kavraki, Jean-Claude Latombe, and E. Latombe. Probabilistic roadmaps for robot path planning, 1998.

[82] Selma Kchir, Tewfik Ziadi, Mikal Ziane, and Serge Stinckwich. A top-down approach to managing variability in robotics algorithms. arXiv preprint arXiv:1312.7572, 2013.

[83] Jens Kober and Jan Peters. Reinforcement learning in robotics: A survey. In Reinforcement Learning, pages 579–610. Springer, 2012.

[84] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. Robotics, IEEE Transactions on, 21(3):354–363, June 2005.

[85] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a. Artificial Intelligence, 155(1):93–146, 2004.

[86] George Konidaris and Andrew Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In Proceedings of the 23rd international conference on Machine learning, pages 489–496. ACM, 2006.

[87] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. Autonomous Robots, 22(2):101–132, 2007.

[88] James J Kuffner. Cloud-enabled robots. In IEEE-RAS International Conference on Humanoid Robotics, Nashville, TN, 2010.

[89] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on, volume 2, pages 995–1001 vol.2, 2000.

[90] T. Kunz, U. Reiser, M. Stilman, and A. Verl. Real-time path planning for a robot arm in changing environments. In Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, pages 5906–5911, Oct 2010.

[91] Tobias Kunz, Ulrich Reiser, Mike Stilman, and Alexander Verl. Real-time path planning for a robot arm in changing environments. In Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, pages 5906–5911. IEEE, 2010.

[92] Tobias Kunz and Mike Stilman. Probabilistically complete kinodynamic planning for robot manipulators with acceleration limits. In Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on, pages 3713–3719. IEEE, 2014.

[93] Yuan-Hsin Kuo and B.A. MacDonald. A distributed real-time software framework for robotic applications. In Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on, pages 1964–1969, 2005.

[94] Kavraki Latombe, L Kavraki, Jc Latombe, R Motwani, and P Raghavan. Randomized query processing in robot motion planning. 1995.

[95] SM LaValle, P Cheng, J Kuffner, S Lindemann, A Manohar, B Tovar, L Yang, and A Yershova. Msl: Motion strategy library. `http://msl.cs.uiuc.edu/msl/`, October 2013.

[96] Steven M LaValle. Rapidly-exploring random trees a new tool for path planning. 1998.

[97] Steven M LaValle, Michael S Branicky, and Stephen R Lindemann. On the relationship between classical grid search and probabilistic roadmaps. The International Journal of Robotics Research, 23(7-8):673–692, 2004.

[98] Peter Leven and Seth Hutchinson. Toward real-time path planning in changing environments. 2000.

[99] Peter Leven and Seth Hutchinson. A framework for real-time path planning in changing environments. The International Journal of Robotics Research, 21(12):999–1030, 2002.

[100] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. arXiv preprint arXiv:1603.02199, 2016.

[101] Tsai-Yen Li and Yang-Chuan Shie. An incremental learning approach to motion planning with roadmap management. In Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on, volume 4, pages 3411–3416. IEEE, 2002.

[102] Jyh-Ming Lien and Yanyan Lu. Planning motion in environments with similar obstacles. In Robotics: Science and Systems. Citeseer, 2009.

[103] Maxim Likhachev. Sbpl graph search library. `http://sbpl.net/`, April 2014.

[104] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. In Advances in Neural Information Processing Systems, page None, 2003.

[105] Chih-Chung Lin and Ruei-Chuan Chang. On the dynamic shortest path problem. Journal of Information Processing, 13(4):470–476, 1991.

[106] Chenggang Liu and Christopher G Atkeson. Standing balance control using a trajectory library. In Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on, pages 3031–3036. IEEE, 2009.

[107] Hong Liu, Xuezhi Deng, Hongbin Zha, and Ding Ding. A path planner in changing environments by using wc nodes mapping coupled with lazy edges evaluation. In Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, pages 4078–4083. IEEE, 2006.

[108] Hong Liu, Kai Rao, and Fang Xiao. Obstacle guided rrt path planner with region classification for changing environments. In Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on, pages 164–171, Dec 2013.

[109] C. Lopera, H. Tome, A.R. Tsouroukdissian, and F. Stulp. Comparing motion generation and motion recall for everyday mobile manipulation tasks. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 3045–3046, Oct 2012.

[110] Carmen Lopera, Hilario Tomé, and Freek Stulp. Comparing motion generation and motion recall for everyday robotic tasks. In Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on, pages 146–152. IEEE, 2012.

[111] Yanyan Lu and Jyh-Ming Lien. Finding critical changes in dynamic configuration spaces. In Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, pages 2626–2631. IEEE, 2011.

[112] Lu Ma, Mahsa Ghafarianzadeh, Dave Coleman, Nikolaus Correll, and Gabe Sibley. Simultaneous localization, mapping, and manipulation for unsupervised object discovery. arXiv preprint arXiv:1411.0802, 2014.

[113] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. On the benefits of making robotic software frameworks thin. In Intelligent Robots and Systems (IROS'07), 2007. IEEE International Conference on. IEEE, 2007.

[114] James D Marble and Kostas E Bekris. Asymptotically near-optimal is good enough for motion planning. In International Symposium on Robotics Research, 2011.

[115] S.R. Martin, S.E. Wright, and J.W. Sheppard. Offline and online evolutionary bi-directional rrt algorithms for efficient re-planning in dynamic environments. In Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on, pages 1131–1136, Sept 2007.

[116] A. Menon, B. Cohen, and M. Likhachev. Motion planning for smooth pickup of moving objects. In Robotics and Automation (ICRA), 2014 IEEE International Conference on, pages 453–460, May 2014.

[117] Tekin Meriçli, Manuela Veloso, and H Levent Akın. Experience guided mobile manipulation planning. In 8th International Cognitive Robotics Workshop, AAAI, volume 12, pages 22–23, 2012.

[118] Tekin Meriçli, Manuela Veloso, and H Levent Akın. Push-manipulation of complex passive mobile objects using experimentally acquired motion models. Autonomous Robots, pages 1–13, 2014.

[119] Mark Moll, Ioan A Şucan, Janice Bordeaux, and Lydia E Kavraki. Teaching motion planning concepts to undergraduate students. In Advanced Robotics and its Social Impacts (ARSO), 2011 IEEE Workshop on, pages 27–30. IEEE, 2011.

[120] Mark Moll, Ioan A. Şucan, and Lydia E. Kavraki. Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. IEEE Robotics & Automation Magazine (Special Issue on Replicable and Measurable Robotics Research), 22(3):96–102, September 2015.

[121] Igor Mordatch, Kendall Lowrey, and Emanuel Todorov. Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids. In Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on, pages 5307–5314. IEEE, 2015.

[122] Igor Mordatch, Emanuel Todorov, and Zoran Popović. Discovery of complex behaviors through contact-invariant optimization. ACM Transactions on Graphics (TOG), 31(4):43, 2012.

[123] Ohloh. Ohloh: Moveit! project summary factoids. `https://www.ohloh.net/p/moveit_/factoids\#FactoidTeamSizeVeryLarge`, October 2013.

[124] Jia Pan, S. Chitta, and D. Manocha. Fcl: A general purpose library for collision and proximity queries. In Robotics and Automation (ICRA), 2012 IEEE International Conference on, pages 3859–3866, 2012.

[125] Jia Pan and Dinesh Manocha. Fast and robust motion planning with noisy data using machine learning. In Proceedings of the 30th International Conference on Machine Learning, 2013.

[126] Zengxi Pan, Joseph Polden, Nathan Larkin, Stephen Van Duin, and John Norrish. Recent progress on programming methods for industrial robots. Robotics and Computer-Integrated Manufacturing, 28(2):87 – 94, 2012.

[127] C. Park, F. Rabe, S. Sharma, C. Scheurer, U. E. Zimmermann, and D. Manocha. Parallel cartesian planning in dynamic environments using constrained trajectory planning. In 2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids), pages 983–990, Nov 2015.

[128] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In Robotics and Automation, 2009. ICRA'09. IEEE International Conference on, pages 763–768. IEEE, 2009.

[129] Alexander Perez and Jan Rosell. A roadmap to robot motion planning software development. Computer Applications in Engineering Education, 18(4):651–660, 2010.

[130] Leonid Peshkin and Edwin D De Jong. Context-based policy search: transfer of experience across problems. In ICML-2002 Workshop on Development of Representations, 2002.

[131] Roland Philippsen. A light formulation of the e interpolated path replanner. Technical report, ETH-Zürich, 2006.

[132] Mike Phillips, Benjamin J Cohen, Sachin Chitta, and Maxim Likhachev. E-graphs: Bootstrapping planning with experience graphs. In Robotics: Science and Systems, 2012.

[133] Mike Phillips, Victor Hwang, Sachin Chitta, and Maxim Likhachev. Learning to plan for constrained manipulation from demonstrations. In Robotics: Science and Systems, 2013.

[134] E. Plaku, K.E. Bekris, and E.E. Kavraki. Oops for motion planning: An online, open-source, programming system. In Robotics and Automation, 2007 IEEE International Conference on, pages 3711–3716, 2007.

[135] M. Pomarlan and I.A. Sucan. Motion planning for manipulators in dynamically changing environments using real-time mapping of free workspace. In Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on, pages 483–487, Nov 2013.

[136] Mihai Pomârlan and Ioan A Sucan. Motion planning for manipulators in dynamically changing environments using real-time mapping of free workspace. In IEEE Intl. Symp. on Computational Intelligence and Informatics, pages 483–487, 2013.

[137] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In ICRA Workshop On Open Source Software, volume 3, 2009.

[138] Nathan Ratliff, Matthew Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In Robotics and Automation, 2009. ICRA'09. IEEE International Conference on, pages 489–494. IEEE, 2009.

[139] John H. Reif. Complexity of the mover's problem and generalizations. 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, 0:421–427, 1979.

[140] Rethink Robotics. Baxter research robot. http://cdn-staging.rethinkrobotics.com/wp-content/uploads/2014/ 9:13, 2013.

[141] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs, 25, 1995.

[142] Douglas C Schmidt. Why software reuse has failed and how to make it work for you. C++ Report, 11(1):1999, 1999.

[143] Douglas C Schmidt and Adam Porter. Leveraging open-source communities to improve the quality & performance of open-source software. In Proceedings of the 1st Workshop on Open Source Software Engineering, 2001.

[144] Henrik Schumann-Olsen, Marianne Bakken, Øystein Hov Holhjem, and Petter Risholm. Parallel dynamic roadmaps for real-time motion planning in complex dynamic scenes. 2014.

[145] Fabian Schwarzer, M Saha, and JC Latombe. Software: Motion planning kit. `http://robotics.stanford.edu/~mitul/mpk/`, November 2013.

[146] T. Simon, J.-P. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. Advanced Robotics, 14(6):477–493, 2000.

[147] Thierry Simon, Jean-Paul Laumond, Juan Corts, and Anis Sahbani. Manipulation planning with probabilistic roadmaps. The International Journal of Robotics Research, 23(7-8):729–746, 2004.

[148] William D Smart. Is a common middleware for robotics possible? In Proceedings of the IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. Citeseer, 2007.

[149] Ruben Smits. Kdl: Kinematics and dynamics library. `http://www.orocos.org/kdl`, October 2013.

[150] JF Stein. Representation of egocentric space in the posterior parietal cortex. Experimental Physiology, 74(5):583–606, 1989.

[151] Aaron Steinfeld, Terrence Fong, David Kaber, Michael Lewis, Jean Scholtz, Alan Schultz, and Michael Goodrich. Common metrics for human-robot interaction. In Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction, HRI '06, pages 33–40, New York, NY, USA, 2006. ACM.

[152] Anthony Stentz. Optimal and efficient path planning for unknown and dynamic environments. International Journal of Robotics and Automation, 10:89–100, 1993.

[153] Anthony Stentz. The focussed d* algorithm for real-time replanning. In In Proceedings of the International Joint Conference on Artificial Intelligence, pages 1652–1659, 1995.

[154] Martin Stolle and Christopher G Atkeson. Knowledge transfer using local features. In Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on, pages 26–31. IEEE, 2007.

[155] Martin Stolle, Hanns Tappeiner, Joel Chestnutt, and Chris Atkeson. Transfer of policies based on trajectory libraries. In IEEE/RSJ Intl. Conf. Intelligent Robots and Systems, 2007.

[156] Martin Stolle, Hanns Tappeiner, Joel Chestnutt, and Christopher G Atkeson. Transfer of policies based on trajectory libraries. In Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on, pages 2981–2986. IEEE, 2007.

[157] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. IEEE Robotics & Automation Magazine, 19(4):72–82, December 2012.

[158] Ioan Alexandru Sucan and Sachin Chitta. Motion planning with constraints using configuration space approximations. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 1904–1910. IEEE, 2012.

[159] S. H. Suh, I. K. Woo, and S. K. Noh. Development of an automatic trajectory planning system (atps) for spray painting robots. In Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on, pages 1948–1955 vol.3, Apr 1991.

[160] Sebastian Thrun. Lifelong Learning Algorithms, pages 181–209. Springer US, Boston, MA, 1998.

[161] J. Vannoy and Jing Xiao. Real-time adaptive motion planning (ramp) of mobile manipulators in dynamic environments with unforeseen changes. Robotics, IEEE Transactions on, 24(5):1199–1212, Oct 2008.

[162] XJ Wu, J Tang, Q Li, and KH Heng. Development of a configuration space motion planner for robot in dynamic environment. Robotics and Computer-Integrated Manufacturing, 25(1):13–31, 2009.

[163] Keenan A Wyrobek, Eric H Berger, HF Machiel Van der Loos, and J Kenneth Salisbury. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on, pages 2165–2170. IEEE, 2008.

[164] Holly A Yanco and Jill L Drury. A taxonomy for human-robot interaction. In Proceedings of the AAAI Fall Symposium on Human-Robot Interaction, pages 111–119, 2002.

[165] Matthew Zucker, James Kuffner, and Michael Branicky. Multipartite rrts for rapid replanning in dynamic environments. In Robotics and Automation, 2007 IEEE International Conference on, pages 1603–1609. IEEE, 2007.

Glossary

Δ_{sparse} sparse delta visibility radius. xi, xiii, xiv, 26, 27, 28, 63, 64, 66, 77, 76, 77, 78, 80, 81, 91, 92, 94, 95, 100

BoltDR Bolt Dual Roadmap. xv, 91, 98, 100, 101, 104, 106, 107, 108, 109, 114, 122

BoltSR Bolt Single Roadmap. 91, 98, 101, 104, 106, 107, 108, 122, 125

c-space configuration space. iii, xi, xii, 3, 4, 6, 7, 10, 11, 14, 15, 16, 18, 21, 22, 23, 24, 23, 24, 26, 27, 28, 29, 30, 31, 33, 40, 52, 62, 63, 64, 67, 71, 72, 73, 74, 75, 76, 77, 80, 86, 88, 90, 91, 92, 93, 96, 98, 105, 106, 109, 114, 115, 116, 119, 128, 129, 130

DOF degrees of freedom. 3, 7, 10, 11, 27, 62, 66, 67, 88, 92, 93, 101, 107, 111, 114, 128, 129, 130

EBMP experience-based motion planning. iii, 1, 2, 3, 4, 7, 8, 10, 11, 12, 13, 17, 18, 22, 23, 28, 30, 35, 61, 72, 90, 91, 93, 104, 105, 106, 107, 108, 119, 121, 127

ERRT-Connect Experience-RRT-Connect. 7, 119, 120, 124

IK inverse kinematic. 19, 30, 35, 40, 47, 51, 52, 73, 103, 110, 112

NN nearest neighbor. 74, 91, 119, 120

OMPL Open Motion Planning Library. xii, 33, 41, 50, 51, 54, 58, 66, 85, 104, 116

PFS Planning from Scratch. xvi, 7, 12, 15, 20, 30, 61, 62, 66, 67, 70, 71, 75, 90, 91, 98, 101, 105, 106, 107, 108, 119, 120, 122, 125, 127

PRM Probabilistic Roadmap. 9, 15, 22, 23, 25, 26, 31, 65, 74, 75, 88, 91

RR Retrieve Repair. 20, 61, 62, 90, 101

RRT Rapidly Exploring Random Tree. xi, 6, 9, 16, 17, 18, 20, 21, 119

SPARS Sparse Roadmap Spanners. 26, 27, 28, 62, 63, 64, 65, 71, 74, 75, 79, 80, 81, 82, 84, 85, 86, 88, 96

SPARS2 Sparse Roadmap Spanners 2. xiii, xiv, 27, 62, 66, 72, 73, 74, 82, 83, 84, 85, 86, 85, 86, 87, 86, 87, 88, 92, 93, 94, 97, 98, 104, 107, 122, 128

V-PRM Visibility-based PRM. 28, 92, 98