Accelerated Motion Planning Through Hardware/Software Co-Design

by

Sean Michael Murray

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

_____

Daniel J. Sorin, Supervisor

_____

George Konidaris

_____

Derek Hower

_____

Hai Li

_____

John Board

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2019

ABSTRACT

Accelerated Motion Planning Through Hardware/Software Co-Design

by

Sean Michael Murray

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

_____

Daniel J. Sorin, Supervisor

_____

George Konidaris

_____

Derek Hower

_____

Hai Li

_____

John Board

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University

2019

Abstract

Robotics has the potential to dramatically change society over the next decade. Technology has matured such that modern robots can execute complex motions with sub-millimeter precision. Advances in sensing technology have driven down the price of depth cameras and increased their performance. However, the planning algorithms used in currently-deployed systems are too slow to react to changing environments; this has restricted the use of high degree-of-freedom (DOF) robots to tightly-controlled environments where planning in real time is not necessary.

Our work focuses on overcoming this challenge through careful hardware/software co-design. We leverage aggressive precomputation and parallelism to design accelerators for several components of the motion planning problem. We present architectures for accelerating collision detection as well as path search. We show how we can maintain flexibility even with custom hardware, and describe microarchitectures that we have implemented at the register-transfer level. We also show how to generate effective planning roadmaps for use with our designs.

Our accelerators bring the total planning latency to less than 3 microseconds, several orders of magnitude faster than the state of the art. This capability makes it possible to deploy systems that plan under uncertainty, use complex decision making algorithms, or plan for multiple robots in a workspace. We hope this technology will push robotics into domains and applications that were previously infeasible.

Acknowledgements

The five-year process leading to this dissertation has been full of ups and downs. There have been days filled with endless enthusiasm and energy, weeks where it seemed nothing would ever work, and months questioning whether robots would ever advance out of the stone age. I have been incredibly lucky to have had the chance to learn so much about the two distinct fields of robotics and computer architecture. I have grown as a scientist, engineer, and human being. It has been an incredible journey to take and nurture this work from a conversation at office hours, to the lab, to building real systems that push the boundaries of robotics. None of this would have been possible without the support of many wonderful people in my life.

I would like to start by thanking my family. My parents supported my decision to quit a stable job and leave that path behind to chase the dream of making a career out of my hobbies. They also made sure I did my homework as a wee lad, and I'm sure I would not have amounted to anything without their constant support and encouragement.

My two siblings have also been a source of inspiration and friendship throughout my life. During our tumultuous childhood, they were my only constant companions. My older brother has a Ph.D of his own, and served as a role model of scientific passion. My younger brother is an expert in doing exactly what he wants, while somehow maintaining a remarkable work ethic.

My community in Durham made the graduate school experience a joy. The incoming Pratt cohort of 2014 was an amazing collection of individuals excited about science, fellowship, and exploration. They are too many to list, but they made it fun to come to work, to commiserate, and to unwind. The yoga family helped me stay balanced, mindful, and focused. A special thanks to James, Aaron, and Wes for

running away to the mountains. These trips kept me sane.

A huge thanks must go to my two amazing advisers. Professor Daniel Sorin took a chance on an unconventional recruit and is the reason I was given an opportunity to go to graduate school. He has been supportive from Day 1, helping me grow as a computer architect and scientist. Dr. Sorin is an adviser that cares deeply about his students' success and future. I know that I couldn't have asked for a more reasonable, organized, and thoughtful mentor.

Professor George Konidaris has been an unbelievable co-adviser since my second semester at Duke. He has a boundless optimism and enthusiasm; I can't count how many times I walked into a meeting with George in low spirits, and walked out convinced that the keys to the world were just a couple hacking sessions away. Dr. Konidaris is another rare adviser that makes decisions with students' best interests at heart, for which I know many people are grateful. Dr. Konidaris' patience and generosity with his time, combined with his vision for a robotics industry unencumbered by slow motion planning, have made this work possible.

I must also thank Dr. Drew Hilton, for running a fantastic set of courses at Duke, and teaching me a great deal about programming, digital design, and compilers. I would also like to thank Dr. Derek Hower, Dr. John Board, and Dr. Helen Li for serving on my thesis committee. Each member of my committee brings a very unique perspective to bear, and I am very grateful for their time and support.

Perhaps the luckiest break I caught during this whole process was getting to work with Will Floyd-Jones. For more than four years now we have toiled together, designed together, struggled together, despaired together, and occasionally triumphed together. Will's passion, drive, and creativity have been a continual inspiration. The speed at which he can identify a new subject of interest, achieve proficiency, and then become an expert, is unparalleled, and more than slightly terrifying.

Lastly, I thank the four-legged monster, Metta. For stealing bagels, hiding shoes throughout the apartment, coating my car with fur, and letting the entire neighborhood know when a UPS driver dares to even consider turning onto our street. Worth it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The field of robotics faces many challenges in the push to deploy intelligent automation solutions that go beyond the conventional "teach-and-repeat" paradigm. Motion planning is one of the fundamental problems that this effort must overcome. The motion planning task is to compute a path that allows a robot to move from its starting configuration to a goal state. The motion plan must take into consideration the obstacles in the environment in order to guarantee the path is collision-free. These obstacles may be inanimate objects, other mechanical agents, or human beings. Besides being collision-free, other desirable traits might be minimizing time spent, energy expended, or maintaining certain kinematic invariants (such as maintaining the vertical orientation of a coffee mug during motion). Conventional solutions may provide algorithmic guarantees such as probabilistic completeness or even asymptotic optimality, but these algorithms are too slow to enable high-speed robotic systems to react in real time. Indeed, we have reached a point where the robots being built are capable of extremely complex, precise, and dexterous movements, but we lack the ability to efficiently utilize them. This disconnect between the mechanical capabilities of robots and our ability to make use of them is a significant barrier to expanding the influence of robotics to new spaces. Currently, almost all industrial robots work in tightly controlled and fenced-off environments that depend on work parts being in exactly the same place and orientation every cycle, eliminating the need to plan motions at runtime.

1.1 The Need for Accelerated Motion Planning

Rapid motion planning is essential to introduce robots into settings where they must work in unstructured environments or in proximity with humans. In order to be safe, these robots must react to the environment in real time to avoid collisions. These requirements

are formalized in ISO technical specification 15066 [1]. This standard provides guidance for the deployment of robots to be co-located with humans. For example, it stipulates that such "speed and separation monitoring" systems must assume that a human operator may at any point begin moving towards it at a speed of 1.6 meters/second, and maintain an appropriate distance accordingly [1].

Even without humans in the system, rapid motion planning is essential in manufacturing and logistics environments, where the time spent to execute each step in the process is crucial to profitability. As will be discussed in later chapters, conventional planning algorithms require on the order of seconds to produce paths. Given that the motions made by high-speed industrial robots are also on the order of seconds in duration, this has an unacceptable effect on cycle time. Online planning can be avoided by sticking to teach-and-repeat tasks, in which a robot is programmed to follow an exact sequence of trajectories that have been calculated to be collision-free (validated in simulation or often even manually). At runtime, the sensor-less robot blindly repeats its motion each work cycle. This strategy results in brittle robotic systems that are expensive to maintain, as the robots must be reprogrammed to accommodate even small changes in the workspace, but it is the established status quo in almost all deployed industrial robotics systems.

High-speed motion planning is also crucial to make use of complex decision-making algorithms (such as high-level task-space planning) that invoke motion planning hundreds or thousands of times as a subroutine [2, 3, 4]. For example, a robot developing a strategy to assemble a product involving 15 components may need to evaluate many thousands of motions before actually performing the first one. Due to the long latency of motion planning, these applications have previously never been feasible in dynamic environments, where the task plan may need to change at runtime. Our work focuses on developing hardware accelerators for motion planning in the hopes of enabling these higher-level algorithms by transforming motion planning into a for-free primitive.

It is well established that application specific hardware can accelerate critical tasks in addition to improving energy-efficiency. This has been demonstrated in situations such as

using an entirely custom supercomputer architecture for molecular dynamics simulation [5], offloading deep neural network inference onto an ASIC [6], and web search augmented with a reconfigurable fabric of FPGAs [7]. As robots and automation begin to enter unstructured environments, having effective compute capability on these devices is critical for them to navigate through changing environments in real time. Application specific compute solutions are also attractive in the field of robotics because power is an important factor when working with mobile (unconnected) robots, or a facility with many robots.

1.2 Contributions and Outline of this Thesis

The work we present in this thesis is focused on enabling real-time motion planning through careful hardware/software co-design. We have developed custom architectures that accelerate various components of motion planning and have implemented these architectures to the RTL level. These accelerators are not simple hardware implementations of known algorithms; their design necessitated developing hardware-friendly algorithms and workflows. We leverage aggressive precomputation, as well as the parallelism inherent in the motion planning problem, to design architectures that achieve orders of magnitude speedup over conventional solutions. The rest of this thesis is structured as follows:

- In Chapter 2, we provide a brief background in motion planning. We discuss the main components and challenges involved in motion planning, as well as introduce the conventional motion planning solutions. We also consider related work that has attempted to accelerate motion planning.
- In Chapter 2.4 we introduce an architecture for accelerating collision detection. Our first contribution utilized combinatorial circuits to encode the swept volume of robotic motion. We present the underlying microarchitecture, and evaluate its performance. This solution is very high performance with a low area footprint, but has a procedurally-generated implementation specific to a single robotic application.

This property makes it appropriate for an FPGA-based platform, but unsuitable for ASIC consideration.

– In Chapter 4 we overcome some of the limitations inherent in our first work by designing a programmable architecture that enables the accelerator to be targeted to any robot and motion planning roadmap. In this second contribution we sacrifice area and performance for flexibility. The microarchitecture is much more complex, so we describe steps we take to mitigate this complexity. The resulting flexible design is better suited to ASIC production.

– In Chapter 5 we show that path search becomes the bottleneck in motion planning once collision detection has been accelerated. We present an architecture for accelerating path search that reduces the latency for this component of planning by several orders of magnitude. Our implementation is flexible to different robots and roadmaps, and leverages aspects of the path search problem that are specific to robot motion planning.

– All of our contributions in Chapters 2.4 to 5 require the *a priori* selection of a static planning roadmap. In order to meet real-time demands, the static roadmap must be robust to expected obstacles in the scenario. Moreover, because of limited hardware resources, the roadmap must be relatively small, especially compared to those generated by conventional planning algorithms. This final contribution in Chapter 6 details the problem of generating roadmaps that are robust to dynamic obstacles, contain high-quality paths, and are compact enough to fit on specialized hardware.

Chapter 2

Motion Planning Background and Related Work

This chapter gives an overview of the motion planning problem to provide the computer architect with the context and domain knowledge relevant to this thesis.

2.1 Configuration Space

Motion plans must be specified in a format that completely defines the system of interest. Such a format is known as a configuration space, and each point in configuration space represents a unique pose of the particle, rigid body, or robot being studied. In a 1983, Lozano-Perez introduced the usefulness of configuration space (cspace) in the field of robotic motion planning [8]. This is a convenient and powerful framework in which to create paths, in part simply because it is such a compact way of containing all necessary information. To briefly illustrate, consider Figure 2.1. In a), it can be seen that the configuration space for a particle in a 2-dimensional world consists of two translation coordinates. In this simple example, that is all that is necessary to completely describe the system. In b), the particle is now a 2D rigid body, and three parameters are thus needed. A point on the body is defined as the *reference vertex*, and a reference initial configuration is defined. A 2D translation vector places the reference vertex at the specified position, a single rotation value gives the rigid body its correct orientation, and the system becomes completely defined. In c), it can be seen that in three-dimensional space, a rigid body's configuration space has six parameters. The first three are a translational vector that places the reference vertex in space, and three independent rotation angles are necessary to completely define the body's orientation. In general, a rigid body in $\mathbf{n}$-dimensional space requires $\mathbf{n} + \binom{\mathbf{n}}{2}$ parameters to define the system: an $\mathbf{n}$-tuple translating the reference vertex, and $\binom{\mathbf{n}}{2}$ independent

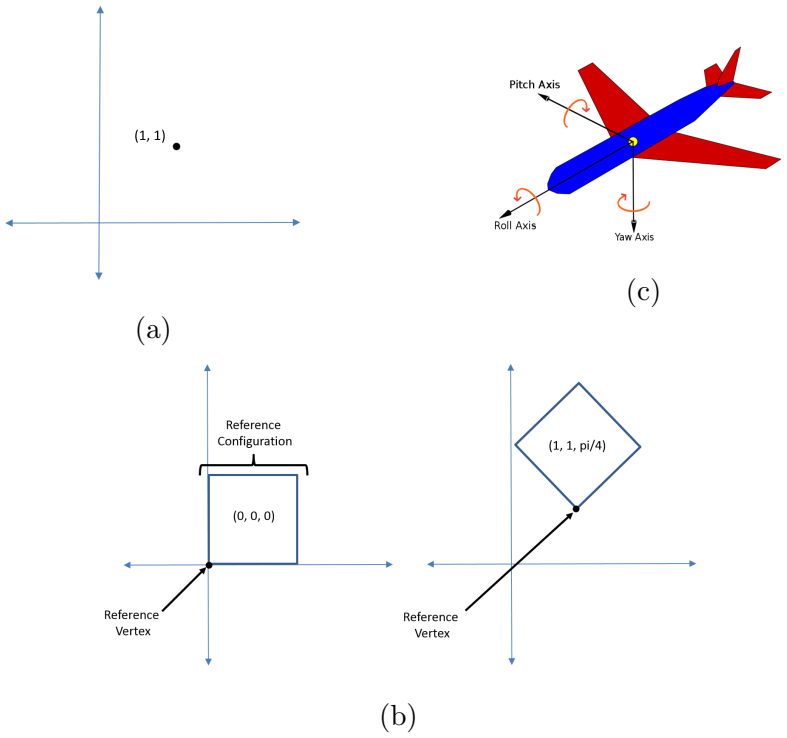


Figure 2.1: a) A particle in 2D space is defined by just 2 parameters. b) A rigid body in 2D space requires both a 2D translation vector, and a single rotation value, defined in relation to a reference vertex and reference configuration. c) A rigid body in 3D space requires a 3D translation vector, as well as roll, pitch, and yaw values[1].

rotation angles. If a joint is added to the rigid body, an additional parameter is required in the configuration space for each degree of freedom the joint introduces. For example, consider a robotic arm with six joints. Such a system would require motion planning to occur in a 12 dimensional space if each joint is a simple revolute joint that only introduces one addition degree of freedom each. If the arm has a fixed base then we can eliminate the translational and rotational components and reduce our system to the 6 degrees of freedom contained in the joints.

[1]Yaw_Axis.svg: Auawise derivative work: (https://creativecommons.org/licenses/by-sa/3.0)

2.2 Components of Motion Planning

A motion plan completely specifies a path a robot can follow from a starting configuration to a goal state without colliding with any obstacles. There are many approaches and algorithms to create motion plans, but there are several fundamental tasks that must be performed and are common to all approaches. We divide these tasks into four categories.

Perception is the use of a combination of sensors and processing to produce a model of the environment. A common strategy is to construct a polygonal hull around environmental obstacles made out of triangles, for example. In our work we assume sensors that produce an occupancy grid. An occupancy grid is a data structure representing which regions of space contain obstacles in a discretized view of the environment. Each discretized region of space is termed a "voxel", a 3D (volumetric) pixel. We leave the problem of constructing the occupancy grid to the vast body of literature concerned with computer vision and sensing.

Roadmap construction is the creation of a graph-based discretization of a robot's configuration space. The most popular motion planning algorithms all use these constructs. The typical graph theory abstractions are used to describe navigation in this space. Each vertex in the graph is a point in the robot's configuration space, and therefore completely defines a specific pose of the robot, and each edge in configuration space represents a movement between two poses. A graph of robot poses and movements is termed a "roadmap." Motion planning in this paradigm thus involves constructing and finding a path through a roadmap that does not collide with any obstacles. If planning in a two dimensional space, dense graphs with high coverage can be quickly constructed. The problem becomes quite difficult, however, when working with robots with many degrees of freedom (many-DOF), as it suffers from the same state space explosion problem present in many other fields. An interesting robotic platform may have six to ten degrees of freedom, so the space that must be explored is far too large to build a dense roadmap. Autonomous vehicles also make use of roadmap-based planning techniques [9], and construction of roadmaps in this domain simply involves different degrees of freedom and constraints. An example toy roadmap in a two-dimensional space is shown in Figure 2.2.

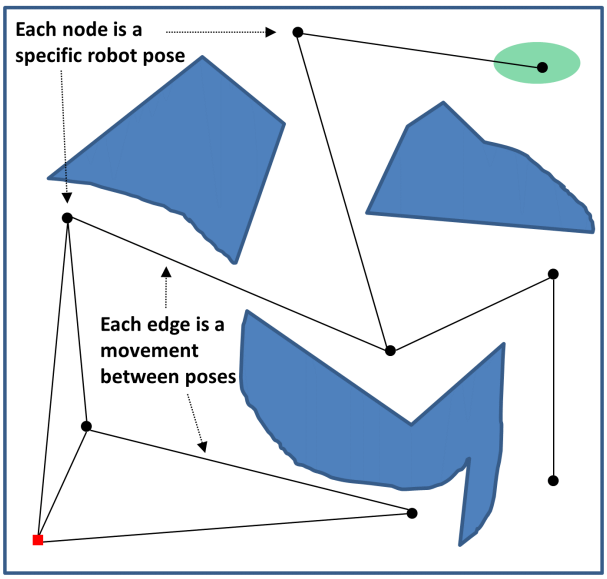


Figure 2.2: Roadmap showing how a path could be found from a starting configuration (red square node) to the goal region (green oval) by sampling in configuration space and avoiding obstacles (amorphous blue regions). This example illustrates planning in 2D, while planning for most robotic arms takes place in a higher dimensional space.

Collision detection is determining if a motion or configuration of a robot is in collision with itself or the environment. There are several ways to perform collision detection which will be discussed in later sections. Collision detection can be interleaved with roadmap construction, or performed after roadmap construction has finished. This step is quite computationally expensive, and is the bottleneck in conventional planning algorithms [10].

The *path search* phase involves traversing the roadmap to check if a path from the starting position to the goal exists, and to identify optimal paths using a cost function to weight each edge. Search is often done using variants of A* or Dijkstra's algorithm [11].

2.3 Planning with Probabilistic Roadmaps

The difficulty of creating plans for robots with many degrees of freedom has been extensively studied. The challenge of planning in very large spaces is such that even the most widely used algorithms are only probabilistically complete, with no guarantees on running time or

memory required to solve a problem of fixed size. In their 1996 work, Overmars and Kavraki [12] detail a process their two labs developed independently aiming to construct roadmaps from which queries could produce motion plans. They focus on many-DOF robots and keep their method general enough that researchers can adjust the algorithm to enable greater performance by taking advantage of knowledge specific to a given scenario/application. This paper is a seminal work in motion planning; we briefly discuss their algorithm here, and how it relates to the strategies we've taken in our work.

The authors break up the Probabilistic Roadmap (PRM) workflow into two phases. The computationally expensive *learning* phase involves the creation of a roadmap consisting of possibly several unconnected graphs where all edges have been determined to be safe, and a fast, inexpensive *query* phase where a path is (hopefully) found through the map to a specified goal configuration. As long as the environment is unchanged, several of these lightweight query calls can be made on the same roadmap. The query and learning phases can also be interleaved if a roadmap must be grown after the initial learning period. The learning phase is itself divided into two steps. The *construction* step creates a base roadmap with a minimal number of cycles, and the *expansion* step enhances the connectivity of the graph to deal with more difficult areas of configuration space.

The construction step of the learning phase follows an iterative process. In each iteration, a random configuration $\mathbf{C_{new}}$ is chosen by sampling values for all the independent degrees of freedom in the robot's configuration space. The first test done is to check whether the obtained configuration is itself collision-free; if so, the node is added to the graph, otherwise it is discarded and the next iteration begins. Next, a list of potential neighbor nodes is assembled by choosing some distance function **D(a, b)** and associated threshold **T**; all nodes **n** with $\mathbf{D}(\mathbf{n}, \mathbf{C_{new}}) < \mathbf{T}$ are added to the list. Working from the closest node in the list to the furthest, each node **n** is tested with a local planner to see if the path from $\mathbf{C_{new}}$ to **n** is collision-free. The properties of the local planner are simply that it must be deterministic, because only this path will be verified to be collision-free. During the query phase the local planner must reproduce the exact same path as it did during the learning

phase. Ideally, the local planner is also fast, because it is used at run time. A common choice of local planner, which we use in our experiments, is simple linear interpolation along all joint angles. This method results in straight line paths (in c-space), is deterministic, and is extremely fast. Depending on the desired connectivity of the graph, one can add edges from $\mathbf{C_{new}}$ to a variable number of nodes from the potential neighbor list that are determined to have collision-free connections; one approach is to add connections for all the nodes that connect $\mathbf{C_{new}}$ to a distinct connected component of the graph, thus merging the two connected components. Ending conditions for the construction step can also be tailored to fit specific application needs and could be a desired number of configurations within a goal region or something as simple as a maximum number of total iterations.

The goal of the expansion step is to improve connectivity in hard regions of configuration space. It is optional, and intended to improve the success rate of the query phase. The basic idea is to keep track during the construction phase of how many times the local planner found collisions between each node n and its potential neighbors. If $\mathbf{C_j}$ had many potential neighbors but was in collision with almost all of them, this indicates that $\mathbf{C_j}$ is in a difficult region of c-space. So one such strategy could be to define a hardness metric, and for every $\mathbf{C_j}$ with a metric above a certain threshold, attempt to sample more configurations in its neighborhood. In this manner, a higher degree of success may be obtained in the query phase by improving the connectivity of the difficult regions. We mention this optional step because in Chapter 6 we find that a human-guided expansion step can greatly increase the performance of fixed roadmap.

The collision checking involved in both steps of the learning phase is the most expensive part of the PRM process. Collision checking is normally done by representing both the obstacles and robot with polygon meshes. Triangles are normally used to take advantage of their (relatively) simple properties [13]. Collision checking then becomes checking whether any of the triangles in the robot's representation intersect with any of the triangles in the obstacles representation (or if the robot collides with itself). Each representation may consist of hundreds or even thousands of triangles, so many thousands or millions of triangle

intersection tests may be necessary to check a *single* configuration of the robot. Collision checking an edge is usually done by breaking the motion up into a sequence of configuration milestones, and ensuring each step of the motion is collision-free. If a motion is broken up into a hundred steps, then a hundred configurations must be checked (each one involving thousands or millions of triangle-tests) to verify that the single edge is safe. Even in the absence of obstacles, ensuring a motion does not result in a self-collision is non-trivial.

The query phase is a much simpler process. It simply involves finding paths between given start and end configurations in the graph. Any graph search or shortest path algorithm suffices. The same local planner is used to create the paths for motion between configurations as was used during the learning phase. It can be run much faster this time, however, because collision checking is not required during the query phase, as the paths generated by the local planner are already guaranteed to be collision-free. In this manner, as long as the environment has not changed since the learning phase, several queries can be run at very low cost.

2.3.1 Optimal Path Planning Algorithms

PRM and the related RRT (rapid exploring random trees) are both only concerned with path feasibility. The algorithms are sound in that they will never return invalid paths, and are probabilistically complete in that while for a given number of samples neither is guaranteed to return a path if one exists, as the number of samples approaches infinity, the likelihood of not finding an existing path decays to 0 [14]. What these algorithms do not do, however, is make any guarantee on the quality of the path found. In fact, they can lead to highly suboptimal, unnecessarily long paths, and this tendency is not attenuated by increasing the number of samples [12]. The second generation of probabilistic planning strategies aimed to remove this deficiency by adapting the algorithms to constantly improve path quality as the sampling process continues. In 2011, Karaman and Frazzoli [14] proposed modifications to several probabilistic algorithms and proved that these new algorithms, PRM* and RRT*, are asymptotically optimal.

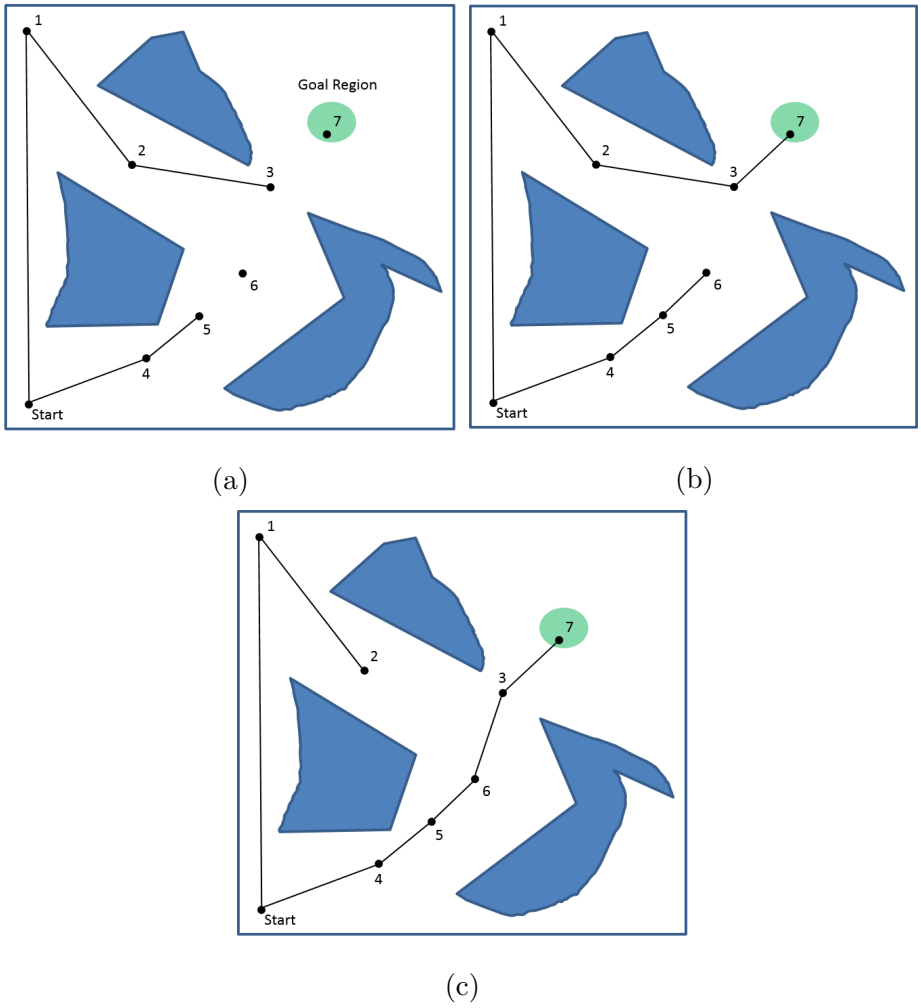


Figure 2.3: a) State of an RRT planner after 5 nodes have been sampled, showing where the next two samples will be. b) Roadmap after these two nodes are sampled in conventional RRT yields a highly non-optimal path. c) When node 6 is sampled, the RRT* algorithm rewires the connection (2,3) to (6,3), yielding a much better path to node 7 when it is sampled.

As an example of how these modifications work, consider RRT. RRT is very similar to PRM, except that each time a node is sampled, it is only connected to the nearest feasible neighbor instead of to all the neighbors within a certain radius or the **k** closest neighbors. This leads to a tree structure with the root at the starting configuration. The downside of this algorithm is that it is greedy, in that it makes the best choice at sample time, which may not remain optimal or even close to optimal later on. An example is shown in Figure 2.3a. The nodes in this figure are labeled in the order sampling occurred to show why connections were established as shown up to node 5. For the sake of the example, assume the next two nodes to be sampled are known. If nodes 6 and 7 are sampled while following a basic RRT algorithm, the resulting path to node 7 will be highly sub-optimal, as only the closest neighbor is considered for attachment, seen in Figure 2.3b. What RRT* changes is each time a node is added to the graph, all pre-existing neighbors within a certain radius are considered for rewiring if the current path to that neighbor is more costly than it would be to go through the newly added node. In Figure 2.3c, pre-existing node 3 is seen to have a shorter path if rewired to have newly sampled node 6 as its parent, and this results in a shorter path to node 7 (the goal). This optimization does not change the coverage of the tree, but creates much better paths at the cost of additional collision checking. The authors prove that these enhancements do not change the complexity of the algorithms and produce solutions that do in fact converge to optimality as the number of samples grows.

2.4 Parallelizing Planning Algorithms

There have been previous efforts to parallelize robotic planning algorithms. In one paper, the authors investigate the use of the many cores of a GPU to perform collision checks in parallel [15]. In particular, they look into the parallelization of RRT and RRT*. The authors use CUDA to achieve parallelization in three dimensions in the collision checking procedure.

The first dimension has to do with the way the swept volume is approximated by taking

snapshots of the robot at several points along its motion between the two configurations. For example, in collision checking the path between configurations $\mathbf{C_1}$ and $\mathbf{C_2}$ with a discretization level of 100, the algorithm would calculate 100 intermediate configurations along the (deterministic) path between $\mathbf{C_1}$ and $\mathbf{C_2}$, and collision check all these intermediate configurations. If any of these are in collision with any obstacle, then the edge is unsafe. The first dimension of parallelization in the author's implementation is that each thread block works on a different discretization point. So in the previous example, each pair-wise collision check would spawn 100 thread blocks to examine different intermediate configurations. Within each thread block, the authors designed their kernel such that a different thread would collision check the robot against a single obstacle. If there were 16 obstacles in the scene, each thread block would have 16 threads, each doing the collision detection between a single obstacle and the robot.

The highest dimension of parallelization in this study was specific to RRT*, and is not applicable to RRT. In RRT*, the algorithm strives for optimality by collision checking each new sample with all neighbors within a certain radius in c-space before adding the edge with the minimum total cost to the graph. The authors assign a different grid of thread blocks to do the work for each of these pair-wise checks. To extend the previous running example, assume a scene with 16 obstacles and a collision checking strategy with a discretization level of 100. Now on a given iteration of RRT*, $\mathbf{C_{new}}$ is sampled randomly. The radius is chosen such that there are four potential neighbors for $\mathbf{C_{new}}$, consisting of the set $\{\mathbf{C_1}, \mathbf{C_2}, \mathbf{C_3}, \mathbf{C_4}\}$. Each grid would handle a unique pairwise collision between $(\mathbf{C_{new}}, \mathbf{C_j})$. Then for this iteration, the kernel would launch four grids of thread blocks, where each grid had 100 blocks (one for each discrete configuration in the movement), and each block had 16 threads (one for each obstacle in the scene).

The authors report a speedup of their collision checking routine of around 10x; collision detection, which had been taking up almost 99% of the computation time of the algorithm, was now only taking up 80%. The authors do not directly explain why this speedup is not nearly as high as the theoretical speedup from the available parallelism should have been,

but they hint that costly memory transfer times may be to blame. The sequential nature of the sampling process also probably reduced the available work to do at any given point in time. This work brings planning latency down to hundreds of milliseconds, which is an improvement over planning on general purpose CPUs, but still insufficient for real-time planning.

There is one previous work on speeding up motion planning in custom hardware, but it tries to directly accelerate existing algorithms, and has significant limitations. Atay and Bayazit [16] recognize the high degree of parallelism inherent in the PRM algorithm [17] and investigate using an FPGA to take advantage. The authors implement the learning phase of the PRM algorithm in hardware and perform collision detection with triangle-triangle intersection circuits. The design involved storing reference triangles for the robot on the FPGA, and applying transforms to bring them to the correct location for a given query. The authors quickly ran into hardware limitation issues since each intersection circuit is complex and many thousands are needed. This limited them to consider only impractically simple scenarios, where the robot was a single rectangular prism, since this can be represented with only 12 triangles.

After publication of our designs presented in Chapter 2.4, an additional work investigated custom hardware for collision detection. Dadu-P [18] draws on our work and takes a similar approach., but stores edge data in memory rather than in circuits, enabling reconfigurability. Dadu-P also uses an octree of voxels instead of an occupancy grid. The reliance on external memory transfer to bring in swept volume data causes a 25X latency increase in collision detection compared to what we present in Chapter 2.4, and Dadu-P did not attempt to accelerate path search.

Chapter 3

Combinatorial Circuit-Based Collision Detection Acceleration

From the previous section the PRM algorithm may seem like a sufficient solution. Computation can be done up front in a slow learning phase, and then lightweight calls can be made at runtime in the query phase. However, the roadmap is safe only as long as the environment remains unchanged. Any change requires the connectivity of the roadmap to be recomputed. There are variants of the PRM algorithm that try to minimize this recomputation cost, but it is still quite expensive. Since collision detection consumes 99% of the time of building the roadmap, re-verifying safeness requires almost as much computation as building the roadmap from scratch. This limitation is acceptable in tightly controlled, precisely engineered, and caged-off workcells, but is unreasonable for robots that must quickly plan in dynamic environments. Our initial work focused on precisely this problem: designing a solution to enable planning to occur in real time for dynamic environments.

3.1 Direct Acceleration of Existing Algorithm

Our first strategy was to design a triangle-triangle intersection test accelerator. This was the logical place to start; collision detection had been proven to be the bottleneck in motion planning, collision detection involves performing possibly millions of triangle-triangle intersection tests, and there is a huge amount of parallelism in these tests to exploit [17].

Although there are several clever ways to reduce the amount of computation involved, the simplest method of determining if two triangles **ABC** and **DEF** intersect in 3D space is to test if any of the line segments (**AB**, **AC**, or **BC**) intersect with the triangle **DEF** and the same for the segments of **DEF** against triangle **ABC**. Thus, a single triangle-triangle intersection test can be decomposed into the logical disjunction of six line segment-triangle intersection tests (which can conveniently be performed in parallel) [13].

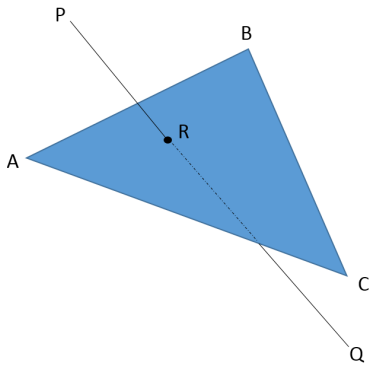


Figure 3.1: A line segment vs triangle test involves determining whether the point **R** at which line **PQ** intersects the plane defined by **ABC** lies within the triangle **ABC**.

A line segment-triangle test is performed by taking the line segment and checking at what point the *line* that extends the segment intersects with the plane of the triangle. For the example shown in Figure 3.1, the equation to check at what point the line extending **PQ** intersects with the plane specified by **ABC** is:

$$\mathbf{A + x(B\text{-}A) + y(C\text{-}A) = P + t(Q\text{-}P).}$$

The variables **t**, **x**, and **y** can be calculated through the following equations (terms in bold typeface can be precomputed, as will be explained later):

$$t = \frac{(P-A) \bullet \mathbf{[(B-A) \times (C-A)]}}{(P-Q) \bullet \mathbf{[(B-A) \times (C-A)]}},$$

$$x = \frac{\mathbf{(C-A)} \bullet [(P-Q) \times (P-A)]}{\mathbf{[(B-A) \times (C-A)]} \bullet (P-Q)},$$

$$y = \frac{\mathbf{(A-B)} \bullet [(P-Q) \times (P-A)]}{\mathbf{[(B-A) \times (C-A)]} \bullet (P-Q)}.$$

If **t** is less than zero or greater than one (the division can be avoided by simply comparing the size of the numerator to the size of the denominator), then the line segment **PQ** does not even intersect the plane of the triangle. If **t** falls between 0 and 1 and the following inequalities hold:

$$0 \leq x, y \leq 1,$$

$$x + y \leq 1,$$

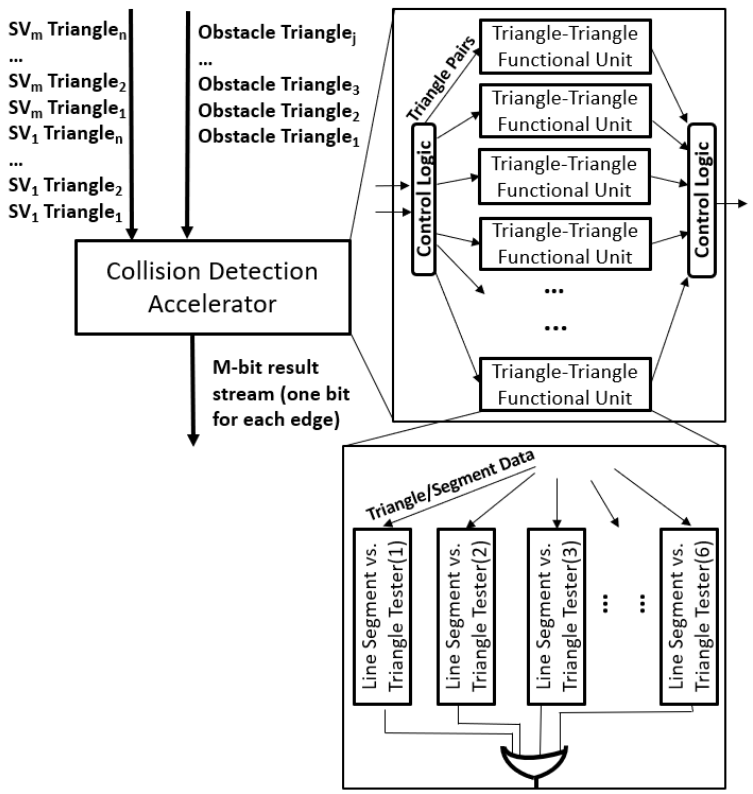


Figure 3.2: Interface for a triangle-triangle based collision detection accelerator (top left). Accelerator takes as input all the triangles for each swept volume (SV) and obstacle. This could be constructed by designing triangle-triangle functional units to efficiently perform pairwise triangle tests (top right). Each functional unit would consist of six line segment vs. triangle testers in parallel (bottom right).

then the line segment-triangle test returns **true**. A potential architecture to accelerate this process is given in Figure 5. The interface accepts a stream of triangles representing swept volumes and a stream of triangles representing obstacles. A single bit for each swept volume is output, representing whether or not that swept volume is in collision. Internally, the accelerator could contain many triangle-triangle functional units to conduct pairwise checks in parallel, with each functional unit performing the six required line segment versus triangle tests in parallel.

Hardware space limitations quickly became apparent when pursuing this strategy, so we made several assumptions to reduce the complexity of the specialized functional units. The first was to use fixed point arithmetic to avoid expensive floating point operations, as

well as to use 9-bit numbers for the coordinates instead of 32. Second, we decided that in order to reduce the amount of online computation needed, we would create a roadmap ahead of time, which would allow the precomputation of many of the terms for the robot triangles. In the equations for $\mathbf{t}$, $\mathbf{x}$, and $\mathbf{y}$ above, the clauses in bold could be calculated ahead of time, assuming **ABC** represents a robot triangle and **PQ** is a line segment from an obstacle.

Even with both these simplifications, the complete parallelization of the triangle-triangle test requires 24 dot products and 7 cross products (the odd number of cross products arises from a corner case that must be checked). This is equivalent to 114 multiplications, 48 additions, and 21 subtractions. The high hardware cost to parallelize a single test was concerning. Indeed, we implemented our design and found that only a few tens (at most) of triangle-triangle functional units would fit on our prototype FPGA board, which was unacceptable for the throughput we desired. From this effort we learned that direct acceleration of algorithms that depend on triangle-triangle intersection tests cannot provide the performance and scalability we require.

3.2 Acceleration of Hardware-Friendly Algorithm

From the first design, we learned that what was needed was a co-designed algorithm to go with custom hardware. Existing algorithms simply did not match well to hardware acceleration. The next route we considered was to aggressively precompute not just some amount of robot geometry, but a whole suite of collision data, and to memoize this data on hardware for fast later access. We accomplish this by discretizing the stream of sensor data. 3D vision sensors typically output a "point cloud" identifying which points in continuous space the sensor perceives as occupied. From this point cloud we create an occupancy grid at some resolution, indicating the presence or absence of an obstacle point(s) within a given region of 3D space. Instead of creating triangle meshes from the point cloud data at runtime, we leverage the fact that a given occupancy grid resolution implies a finite number

of possible obstacles. Expensive collision detection can be done for all edges *in advance* to calculate which regions in space each movement collides with. The precomputed data can be used to create a data structure for each edge that can be queried for membership of a given obstacle voxel. This strategy represents a fundamental change to the PRM algorithm, which typically builds a roadmap for the specific environment at hand.

Our algorithm is similar to an approach by Leven and Hutchinson [19], except they went the opposite direction, creating data structures for each voxel that represented the edges that should be invalidated if present. In effect we are trading a much larger amount of up-front computation for a smaller amount of runtime work. Instead of having to build/reconnect a roadmap each time the environment changes, we build a roadmap in an obstacle-free environment and perform exhaustive collision detection once at design time. Then at runtime we simply access the precomputed data to see how the obstacles in a given environment affect the edges in the roadmap.

Because our goal is to achieve the highest degree of parallelism possible, we avoid storing and accessing the precomputed data in memory elements in software. Instead, we encode a binary representation for each discretized voxel and create logical circuits representing the collision data for each edge. The actual circuit generated for a single edge can be seen in Figure 3.3. Having a logical circuit representation made for an intuitive mapping to hardware descriptive languages. The binary representation for any voxel is streamed over this logic, and if that point is in collision, the circuit outputs **true**.

We augment the collision logic for each edge with additional circuitry to maintain a limited amount of state regarding the task at hand. A given environment contains many obstacle voxels. If even a single voxel is in collision with a swept volume, the edge represented by that swept volume must be removed from consideration. To achieve this, the output of the logic function can be stored in a flip-flop after being OR'ed with the flop's currently stored value, thus allowing many voxels to be streamed through, saving any positive result since the last RESET. The basic structure of this collision detection circuit (CDC) can be seen in Figure 3.4 along with the interface presented to the system. To interface

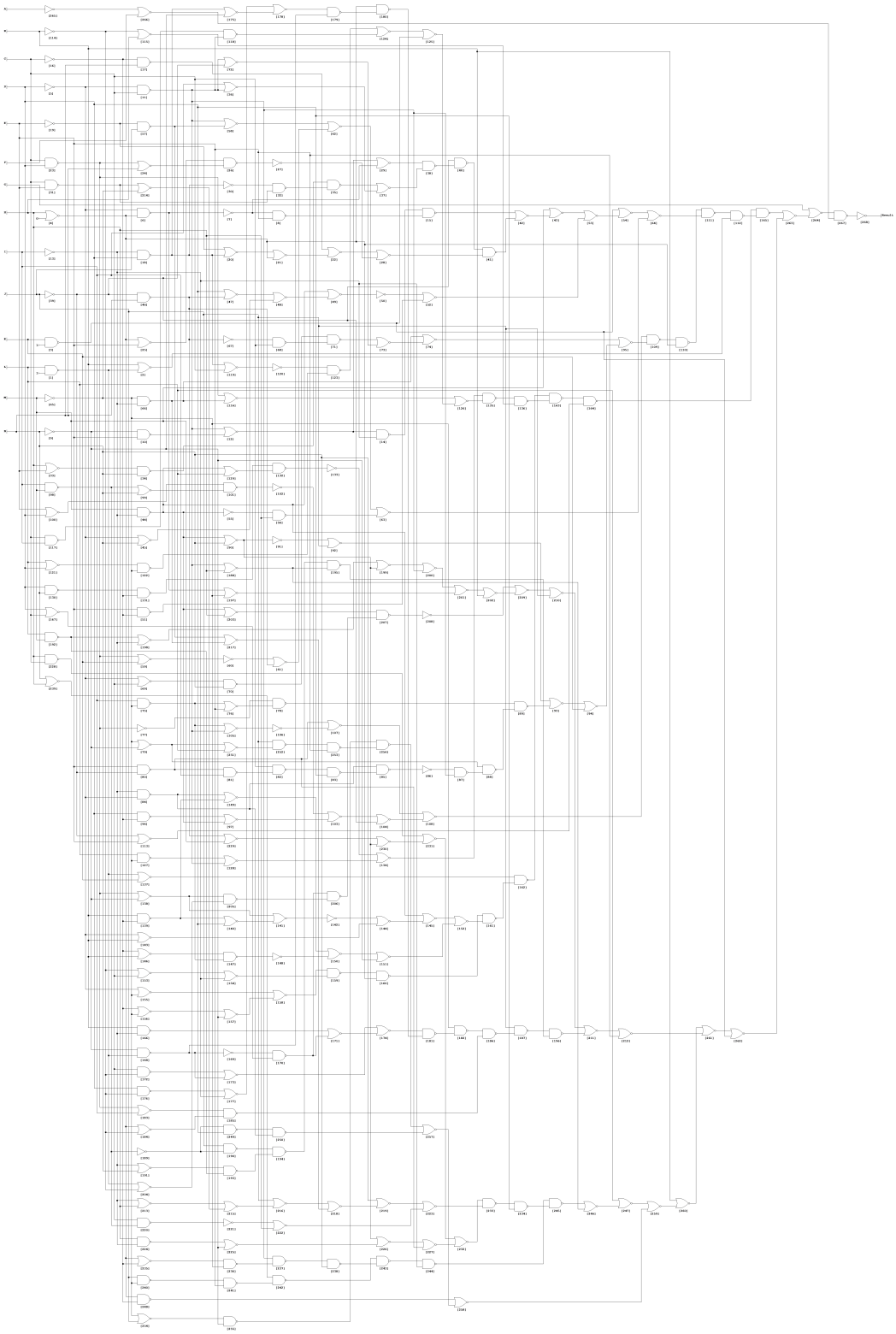


Figure 3.3: The collision circuit generated for a single edge.

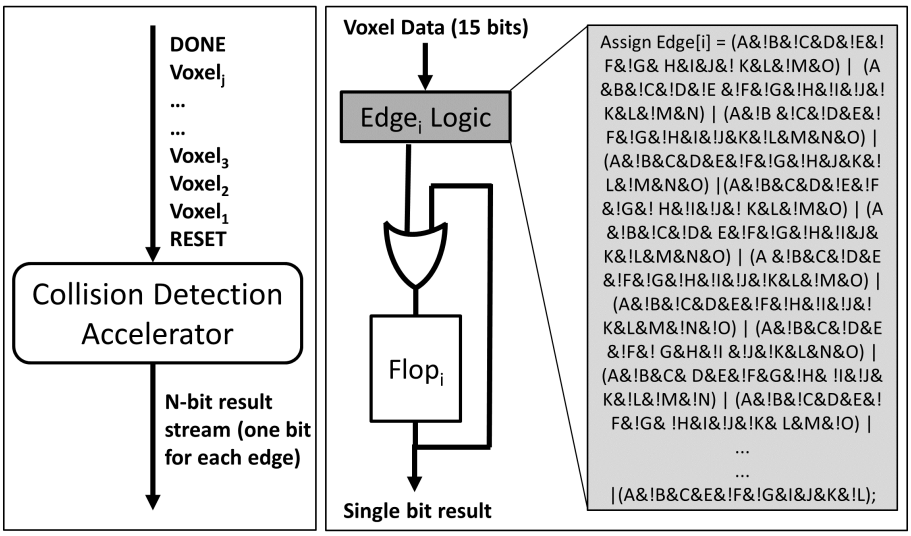


Figure 3.4: The interface of a collision detection accelerator using precomputed data (left). Each collision detection circuit (CDC) contains the logic for an edge, an OR gate, and a flop (right). The variables A through O in the expanded Edge Logic block represent binary voxel ID representations, as will be explained in Section 3.3.

with the accelerator, the host processor can send a RESET signal to instruct the accelerator to clear all flops, followed by a stream of obstacle voxel data (encoded in a fixed-length format), and finally a DONE signal, which initiates transfer of flop data back to the host.

Within our strategy of precomputing collision data there are several additional orthogonal design choices. For example, this strategy is agnostic to the configuration sampling method. Precomputed roadmaps by definition sacrifice asymptotic completeness for the sake of speed, so there is no need for sampling to be probabilistic. Indeed, any *a priori* knowledge about probable obstacle or goal regions can be leveraged to select more useful edges. In addition, the method of discretizing space can be adjusted to the case at hand to create the most useful representation in the most compact form possible.

We now summarize the steps in our workflow:

Preprocessing Steps (done once, at design time):

1. Build the roadmap. For this stage, no collision checking is done except for self-collisions and collisions with permanent features in the environment (e.g., the floor,

ceiling, or a table that is always present). The goal of this step is to create a roadmap with sufficient coverage for solving the expected motion planning problems at an acceptable rate.

2. Discretize the working space of interest and collision check all edges of the roadmap, creating sets of the voxels that each edge collides with.

3. Encode the voxels in a binary representation and formulate logic functions for each edge. Use these logic functions to create RTL-synthesizable descriptions of the circuits in Verilog/VHDL.

Online Query Steps (done each time a plan is needed):

1. Use data from perception sensors to populate an occupancy grid at the same level of discretization at which the roadmap was collision checked.

2. Send all the voxels present in the occupancy grid to the accelerator and collect the results (a bitmask representing which edges are in collision). Use the results to modify the roadmap, setting the edges in collision to a cost of infinity.

3. Perform a graph search through the modified roadmap. A shortest path algorithm such as Dijkstra's can be used to find the shortest path through the map.

4. If a path is found, use the same local planner as in Design Step Two to guide the robot along a safe path to the goal.

3.3 Implementation

We implemented our accelerator to solve problems for a high-DOF robotic arm. We used an Altera Stratix V FPGA on Terasic's DE5 prototyping board. The FPGA interfaces over PCIe to an Intel Xeon 3.5 GHz 4-core processor with 16 GB of RAM.

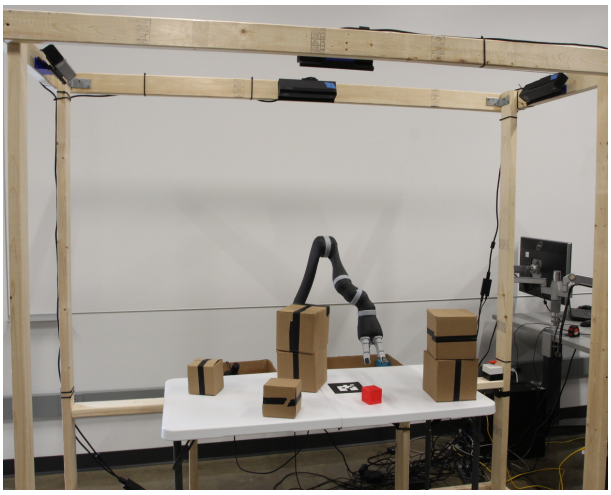


Figure 3.5: Physical layout of the pick-and-place experiment. The work table, four Microsoft Kinects, and the base of the arm are all attached to the wooden frame. The arm picks up colored blocks off the table and places them in bins on the side. The arm must avoid the cardboard boxes, which serve as dynamic obstacles.

The arm we use is the Kinova Jaco2, chosen for its many degrees of freedom. The Jaco arm has one shoulder, two elbows, three wrists, and three fingers. The shoulder and wrists have an infinite range of rotation, while the first and second elbows have ranges of 275 and 325 degrees, respectively. The numerous wrists give the arm great dexterity.

We demonstrated on the pick-and-place use case since this problem is ubiquitous in robotics applications. We placed the robotic arm in front of a table, and each scenario challenged the robot to reach out to grab a toy block while avoiding obstacles and return it to one of two bins on either side of its base, seen in Figure 3.5. Cardboard boxes, ranging from 5-30 cm in each dimension, acted as obstacles.

In the remainder of this section, we discuss the implementation of each stage of the workflow and the implications for the microarchitecture.

Design Step One: Building the Roadmap

As we knew hardware constraints would be a limitation, great care had to be taken to create useful roadmaps of small size. To accomplish this, we made extensive use of the Klampt

Table 3.1: Effect of changing resolution on logic element usage. The logic element usage correlates roughly to the number of voxels with which each edge collides.

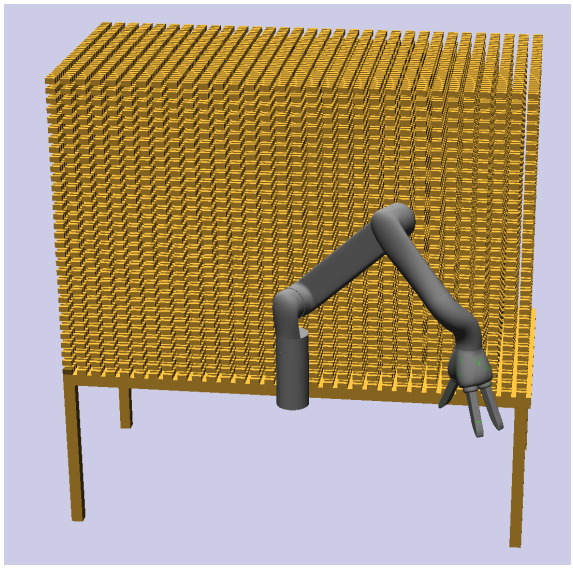
Bits Per Dimension	**4**	**5**	**6**
Logic Elements/Edge	45	160	700
Voxels Colliding/Edge	75	278	1100

robot modeling package [20]. We wanted a high rate of success for working environments, so we followed a heuristic approach combined with probabilistic sampling. First, we created a very large (100,000 edge) roadmap using the PRM* algorithm. Planning was done in an environment with only permanent obstacles present.

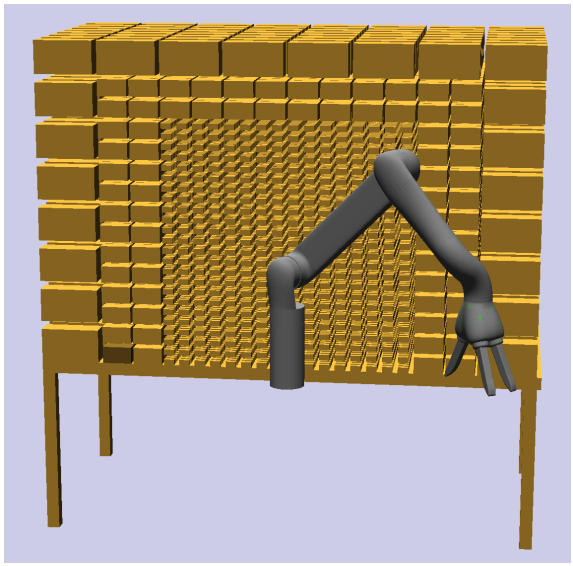
Because the goals (toy blocks) would always be sitting on the table, we needed thorough coverage of the space 4-8 inches above the table. To achieve this, we augmented the resultant graph with a set of configurations just above the surface of the table. We also added shortcut edges from the starting configurations to various spots over the table. These shortcut edges both add useful cycles to the roadmap and also often provide very direct paths in the absence of certain obstacles [21]. This large roadmap must then be processed to a more manageable size. We used a heuristic procedure to iteratively prune the roadmap. In Chapter 6 we detail how this procedure works, the requirements for an improved algorithm, and discuss where this method falls short.

Design Step Two: Discretizing the Workspace and Collision Checking

For the pick-and-place scenarios, the workspace was defined to be the area directly over the table, extending 80 cm high. Once the area of interest is established, the next decision is how exactly to discretize. For simplicity, we employed a simple uniform grid. We examined a few different resolutions ranging from 4 to 10 bits in each dimension. The two competing concerns are the desire to have enough resolution such that the occupancy grid is an accurate



(a)



(b)

Figure 3.6: a) The workspace shown as uniformly discretized. b) A more sophisticated approach can achieve space savings by selectively choosing critical regions to have high resolution and allowing less important regions to be more coarse.

representation of the actual environment, and the fact that the logic function for each edge consumes more hardware as resolution increases. The logic element usage for several levels of resolution is shown in Table 3.1. We chose five bits for each dimension as a good middle ground that provided sufficient precision and also took up a reasonable amount of FPGA resources. The table used in our experiments is 121 centimeters long and 60 centimeters deep, so with a 5 bit resolution/dimension, each block in the occupancy grid is $3.75 \times 1.875 \times 2.5$ cm. Figure 3.6a shows the discretized workspace. Figure 3.6b shows a non-uniform, more sophisticated way that one could discretize. By using knowledge about expected hard or easy/less-important regions, one can selectively increase or decrease resolution to maintain performance while saving space in the logic functions. For the same roadmap, the discretization strategy in Figure 3.6b takes up 27% less space on the FPGA, at the cost of slightly higher design effort.

Each edge in the roadmap constructed in Design Step One is then collision checked in Klampt in the environment containing the full occupancy grid of discretized space. For each edge we create a set of the voxels in collision with that movement. If an edge intersects any part of a voxel, that voxel is included in the set.

Design Step Three: Implementing Logic Functions on FPGA

At the end of Design Step Two, there is a set for each edge containing all the voxels with which that edge collides. A binary representation for each voxel is easily derived since we used a uniform grid in the discretization. We used these binary representations to create a logic function for each edge. For example, if edge 147 collides with the voxels (1,3,5) and (1,3,6), then the voxels are represented by:

00001 00011 00101

00001 00011 00110

The logic functions are then created by simply assigning a variable to each voxel bit (A through O) and combining the two in disjunctive normal form. For this example, the result

would be:

$$Edge147 = (!A\&!B\&!C\&!D\&E\&!F\&!G\&!H\&I\&J\&!K\&!L\&M\&!N\&O)||$$
$$(!A\&!B\&!C\&!D\&E\&!F\&!G\&!H\&I\&J\&!K\&!L\&M\&N\&!O).$$

In this simplistic example, the edge collides with only two obstacle voxels; each edge in the actual roadmap may collide with hundreds or thousands of voxels, so the logic functions can become quite large and take up the bulk of the area of the hardware design. Luckily, there is significant redundancy in these equations, which Leven and Hutchinson [19] refer to as "spatial coherence". For example, the above equation can be simplified to:

$$Edge147 = (!A\&!B\&!C\&!D\&E\&!F\&!G\&!H\&I\&J\&!K\&!L\&M)\&[(!N\&O)||(N\&!O)].$$

We took several actions to reduce the amount of hardware each edge consumes. Logic minimization is a well-studied problem due to its usage in EDA tools. We used a version of the popular *espresso* heuristic logic minimizer [22][23]. *Espresso* can accept as input a set of truth tables, so the sets created in the previous step were converted to truth tables by using the binary encoding of each voxel. We ran these truth tables through *espresso* in groups of 16 to allow the tool to minimize the logic across edges as well. We then converted the output to equivalent Verilog expressions; using *espresso* before converting to Verilog enabled greater than 25% savings in logic utilization on the FPGA, even though all CAD tools (Quartus in this case) do logic minimization of their own. It is likely even more benefit could be realized by "smartly" grouping together edges that shared more in common with each other. Selecting the best edges to group together is a challenging problem ($\binom{\mathbf{N}}{16}$ is quite large when $\mathbf{N}$ is in the thousands) that will be the subject of future work.

Another important design issue is how to distribute the voxels to the CDCs. The board communicates with the host computer over PCIe; as voxels are streamed over the bus to the FPGA they are put into a dual-clocked FIFO, filling up at the bus frequency and draining at the logic frequency. The initial design is in Figure 3.7a. The fifteen bits of each voxel (five bits for each dimension, as discussed above) fan out from the FIFO to the logic for each edge. Each edge's logic function has an associated flip-flop. The input to the flop

Table 3.2: Effect of changing hardware design on logic element usage, normalized to the unbuffered design. All designs were uniformly discretized except the one mentioned.

Design Choice	Logic Element Usage
Unbuffered	1.00
Unbuffered/Non-uniform	0.73
Individual Buffers	1.83
Buffers Shared(16)	1.31
Buffers Shared(32)	1.26
Buffers Shared(64)	1.23

is the OR of the edge logic output with the current value (seen in Figure 3.4). After the input FIFO is drained, the results are fed into an output FIFO which transmits the collision results back to the host computer.

This design is simple but difficult for the FPGA to route in time due to the high fan-out of the input signals. Each input bit must fan out to thousands of CDCs, each of which has several hundred clauses in its logic function. Even clocking the FPGA at 31.25 MHz (1/4 the frequency of the PCIe bus), only 1024 edges could fit on the FPGA. To alleviate this, we investigated a slightly different design. Instead of fanning out the 15 bits of voxel data to all CDCs, only five wires fan out. These five wires are multiplexed over three cycles to send the full 15 bits of voxel data, accumulating the data in a shift register at each CDC. The flops latching the results of the edge logic now need a signal to enable storage only every 3 cycles when input data is valid. This design is in Figure 3.7b. Routing significantly improved with this design, but at the cost of greatly increased logic utilization (an 83% increase). FPGA CAD tools are somewhat opaque, but we believe the increase comes primarily from fewer opportunities for the CAD tools to optimize/re-use logic clauses now that the CDCs compute on unique sets of inputs, rather than from the additional structures introduced.

We found a middle ground between these design points that balances routing difficulty

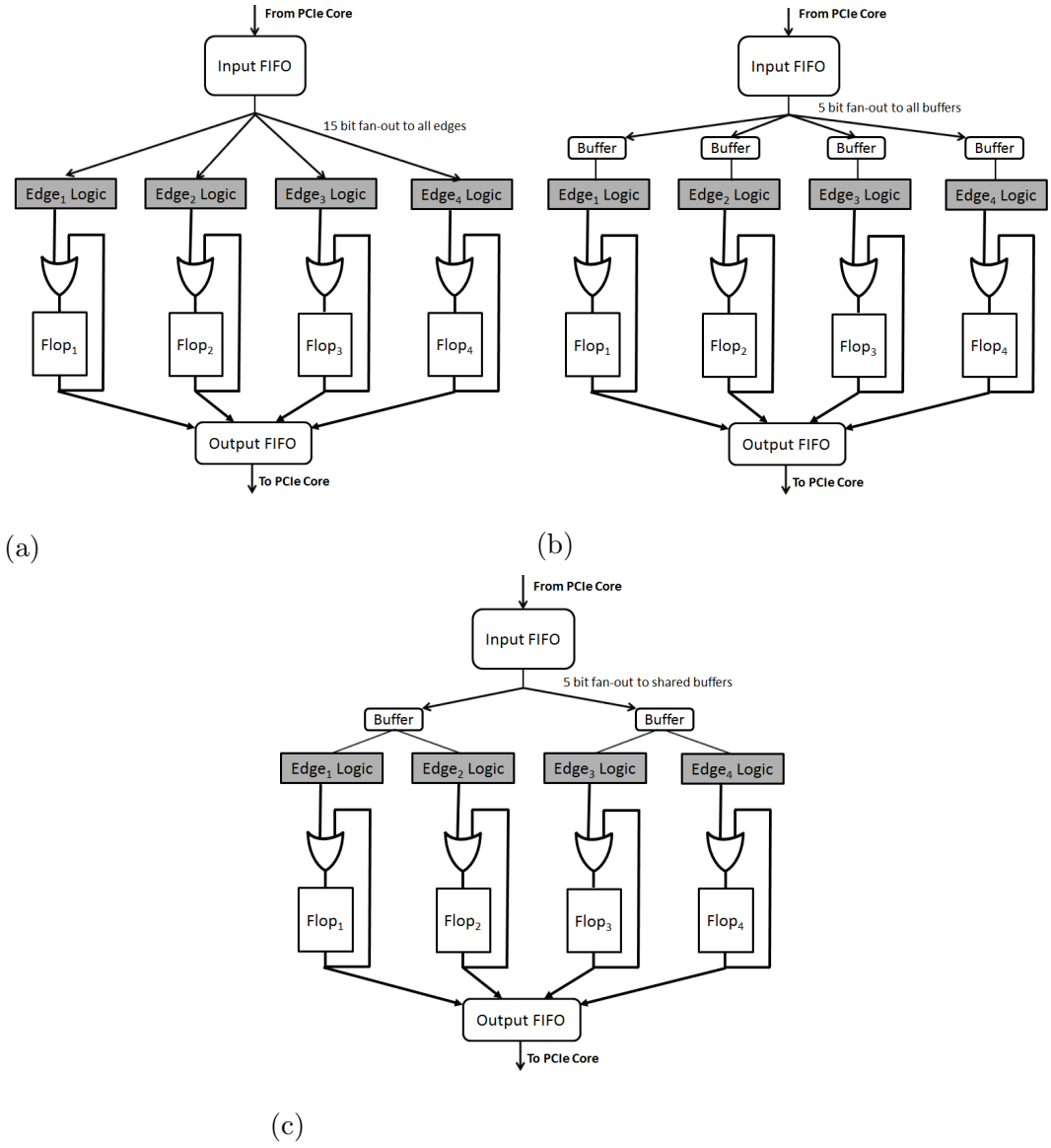


Figure 3.7: a) The design unbuffered with high fan-out. b) Individual buffers for each CDC. c) Multiple CDCs sharing a buffer.

and logic utilization. Instead of allocating a buffer for each CDC, buffers are shared in a hierarchical fashion between groups of CDCs. Figure 3.7c shows an example of this design. Table 3.2 shows the difference in logic utilization/edge for the original case, the case of each CDC having its own buffer, and of 16, 32, and 64 CDCs sharing a single buffer. These adjusted strategies did not recover all of the logic savings of the unbuffered design, but they were able to be compiled and routed much easier. The 32 edge/buffer design allowed 2500 edges to fit on the FPGA, which is more than sufficient for this prototype.

One potential concern about using this buffering technique is that it now takes three cycles to process a point, instead of a single cycle. However, the decreased routing demands of the strategy in Figure 3.7c allow it to be clocked at the same clock frequency as the PCIe bus (125 MHz), compared to the unbuffered design at 1/4 the frequency. The total effect of the more complex microarchitecture is thus both larger logic utilization and modestly higher throughput at the same roadmap size.

Runtime Steps

Perception was done in our experiment with several Microsoft Kinects. These supply sufficient accuracy for our purposes, are relatively cheap, and produce data in a convenient format. An example of a discretized occupancy grid is shown in Figure 3.8. The occupancy grid is then sent over PCIe to the FPGA and the resulting collision bitmask is collected. The data coming back from the FPGA is a vector of which edges are in collision. For each edge in collision, the cost in the roadmap is set to infinity to ensure this path will not be taken during a query. The location of the goal is used to select suitable goal configurations in the roadmap. We accelerated the selection process by precomputing forward-kinematic transforms for all n configurations in the roadmap. This data is stored in a k-d tree that can be very quickly accessed to find which configurations (if any) can be used as a suitable destination for this problem scenario. This structure scales well, with searching the tree taking only *log(n)* time.

A path can now be found using any graph search algorithm on the modified roadmap,

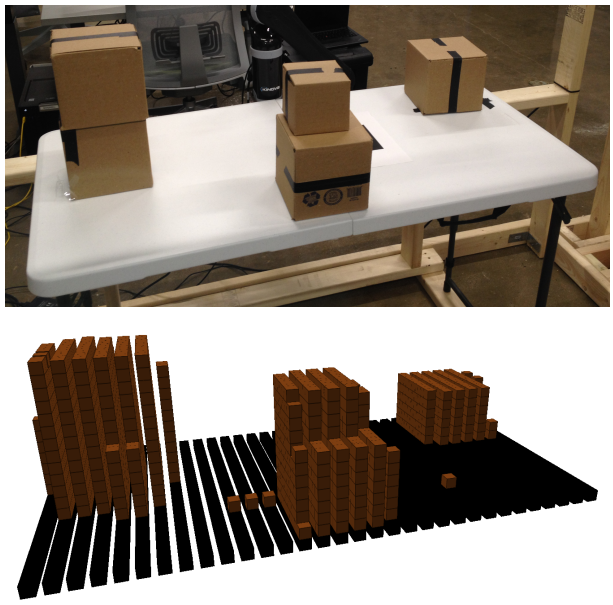


Figure 3.8: A real example scenario (top), and the discretized occupancy grid (bottom).

using the arm's current configuration as the starting node. We used an unoptimized Dijkstra's shortest path algorithm with a heap implementation; faster techniques certainly exist. Edge costs were calculated back in Design Step One and stored for rapid access. If no non-infinite cost path is found to the goal configuration, then the graph has been bisected by obstacles and there is no collision-free path through the precomputed roadmap. In this (unlikely) case, one could fall back onto a conventional software planning routine that has asymptotic completeness. In this way, the common case could be made fast, and the uncommon case would still find a solution (if one exists).

3.4 Results

To determine the source of the speedup from our design, we wrote and evaluated a software version of the same strategy. This implementation used the collision data collected in Design Step Two above to create hashsets of the voxels in collision with each edge. Collision detection in this implementation simply consists of querying obstacle voxels for membership in all of the hashsets. The results of this test can be used to eliminate edges from the

roadmap in exactly the same fashion as described above. The code was instrumented with OpenMP directives to enable the CPU to take advantage of the inherent parallelism in the strategy.

Given the simple and highly parallel nature of this algorithm, it also warranted testing on a GPU. We implemented the same hash set strategy in CUDA and tested on an NVIDIA Tesla K20 (which contains 2496 CUDA cores). Both the CPU and GPU implementations were highly tuned for performance to enable fair comparisons.

To reliably time the speed of the various components of the now-heterogeneous system controlling the robot, we fed 10,000 occupancy grids into the planner and took measurements. With our processor, the total time between obstacle data being sent, and having a motion plan to execute, is less than 650 microseconds (of which less than 25 is collision detection). Previously, collision detection has always been the bottleneck in planning algorithms, but in this approach it is actually one of the fastest components. The vast majority of the 650 microseconds to produce a plan is spent on operating system delays and in the graph search phase. Most occupancy grids for the example scenarios contain $<$750 voxels and, at the clock speed of 125 MHz, are processed by the FPGA in less than 25 microseconds. Traversing the k-d tree to find suitable destination configurations takes 10-20 microseconds. Modifying the roadmap with collision data to assign infinite cost to colliding edges requires 50 microseconds. This leaves the latency for communication and graph search. Subtracting the computation time from the round trip latency across the FPGA yields 150 microseconds for communication. Unsophisticated drivers were used to interface with the FPGA over PCIe, and this latency could be reduced. Communicating the same data over PCIe with mature GPU drivers takes around 15 microseconds each way, so the I/O is a feasible target for optimization. The longest step by far is Dijkstra's shortest path algorithm, which requires 425 microseconds in our implementation. It is likely that at a relatively small roadmap size there are faster ways than Dijkstra's to find a short path, but that is not the focus of our work here.

The design-time steps are very slow compared to what is executed at runtime. The

Table 3.3: The average time (in microseconds) required to perform collision detection using the FPGA as a collision detection accelerator compared to using the same aggressive precomputation to create hashsets for fast query on a CPU or GPU. Results here are for roadmaps containing 2,500 edges.

Custom Hardware	**Precomputed Hashsets**	
FPGA	**CPU**	**GPU**
16	9,550	1,100

computationally expensive collision detection required to build the logic functions takes on the order of 45 minutes, and heuristic roadmap pruning can take several hours. Both these steps happen only once at design time, however, and thus are not a concern.

Table 3.3 shows data comparing the different methods of accelerating collision detection. When running with 16 threads, the software hashset implementation takes less than 10 milliseconds to produce the collision data on the same roadmap used in the experiments on the FPGA. When fully parallelized, the GPU hashset kernel spawns more than 500,000 threads and completes queries in less than 1.1 milliseconds for the same roadmap. NVIDIA's profiling tool *nvprof* was used to evaluate the memory transfer times. For the runtime query transfers (transferring obstacle voxel data to GPU, and result data back to host), communication happened in less than 15 microseconds each way. Transferring the actual hashsets takes much longer, at 8 milliseconds, but this only needs to happen a single time, after which many queries can be made.

These results demonstrate that significant benefit is gained by attacking the problem from both sides. Our speedup comes from both the improved algorithm to reduce problem complexity and a hardware implementation that can exploit more parallelism. Even though the software hashset implementations achieve a large speedup compared to conventional solutions, they do not scale as well as the custom hardware solution. Any given CPU/GPU with a fixed number of hardware threads will experience a linear increase in compute time as the number of edges grows in the roadmap. This effect can be seen in Figure 3.9 showing the

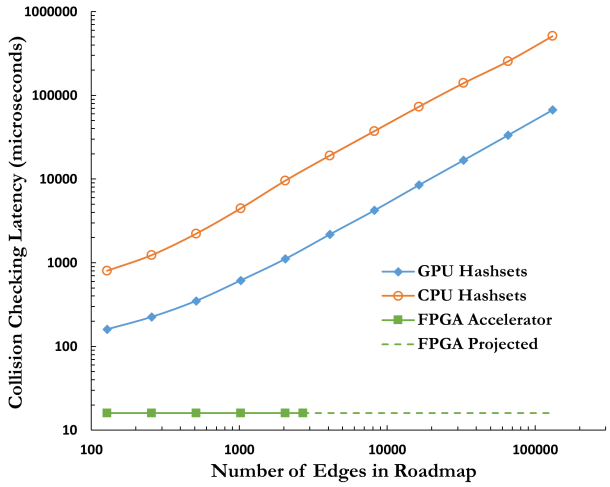


Figure 3.9: Both CPU and GPU solutions scale linearly with respect to the number of edges in the roadmap. The FPGA-architecture described in this paper, however, executes completely in parallel, and thus has a latency independent of roadmap size. All results reflect the latency to process occupancy grids of the same size.

performance of these solutions on roadmaps of increasing size. By contrast, the hardware design is completely parallel and takes constant time to do the computation regardless of the number of edges. The only obstacle in achieving this parallelism for huge roadmaps is the fixed hardware budget.

Collision detection is only one part of motion planning, so we next consider the timing of the motion planning process as a whole. Table 3.4 shows the time to generate a complete motion plan using the FPGA, precomputed hashsets, and conventional solutions. Running the PRM algorithm on the high performance workstation described above on the same sample environments took 0.8 seconds to produce a solution. Rapidly Exploring Random Trees (RRT) is a single-query probabilistic method that is slightly faster at 0.75 seconds. Our approach produces motion plans three orders of magnitude faster than conventional solutions running on CPUs and two orders of magnitude faster than current GPU methods.

Another advantage enjoyed by specialized accelerators is power efficiency. In order to quantify power consumption, we wired a high-wattage current shunt resistor in series with the DE5 board's power supply to determine the peak power consumption of our design while computing collisions for a 1024-edge roadmap. We used an oscilloscope to measure the

Table 3.4: The time (in microseconds) required to produce motion plans using the FPGA as a collision detection accelerator compared to using hashsets on either a CPU or GPU, or conventional software approaches (run on a CPU). RRT is a single-query probabilistic algorithm very similar to PRM, used more commonly when speed is a concern rather than data-reuse. These numbers include time to perform a path search and the latency to transmit data to and from the remote devices for the FPGA/GPU solutions. Although the CPU and GPU can achieve impressive speed-ups using hashsets, these are on relatively small graphs (2,500 edges), and these solutions scale linearly with roadmap size.

Custom Hardware	**Precomputed Hashsets**		**Conventional Approaches**	
FPGA	**CPU**	**GPU**	**PRM**	**RRT**
650	10,000	1,600	815,000	756,000

voltage drop across the resistor during a collision detection query, which caused the board power to increase from a nominal 12W to a peak 15W. The board has other components contributing to this power consumption, so the FPGA would be some fraction of this. GPU solutions, in contrast, are often much higher power. The Tesla K20 is rated for 225 W, and it takes a longer time to execute than the FPGA, thus consuming far more energy.

3.5 Conclusions

This work is the first of our knowledge in the body of literature to focus on the use of software/hardware co-design to accelerate motion planning algorithms. We showed that it is possible to use custom hardware to bring motion planning as a whole to sub-millisecond latency. Our solution achieved several orders of magnitude speedup over the current state of the art. This opens up a whole new range of applications for real-time motion planning where it was previously infeasible. In addition to quickly creating a plan, it also allows you to rapidly verify that a currently executing path is still collision-free, maintaining safety when working with high-speed robots.

There were several limitations to this work. The first is that while the strategy can be applied to any robot, the actual hardware design is specific to a single robot/roadmap pair.

This may be acceptable when using an FPGA-based platform, but it may be desirable to use an ASIC to take advantage of lower power and lower unit cost. In many situations, it would not be viable to make a custom chip specific to a single robot/roadmap. A more useful design would allow some programmability, so the same hardware can accelerate motion planning for a variety of problems. We address this limitation in Chapter 4. The second limitation is that this approach only accelerates collision detection. We show that if you bring collision detection latency to the microsecond range then shortest path search becomes the bottleneck. This problem will only get worse as roadmap size increases. We develop hardware accelerators for path search in Chapter 5.

A third limitation is the fixed roadmap. Unfortunately, having a fixed graph is unavoidable in order to leverage as much precomputation as we do. However, it can lead to the system failing to report a path even when one exists. This situation could occur either due to obstacles appearing too large due to the discretization, or simply if the roadmap created at design-time does not contain enough edges or appropriate edges for the given scenario. With the help of very simple heuristic pruning, we managed to generate compact roadmaps with a success rate of greater than 98%. However, our test environment was relatively simple compared to other possible robotics applications. Even though we had a 98% success rate, we made no guarantees on the quality of the path compared to any baseline. In more challenging environments, it is likely that we will need a more sophisticated strategy to create useful roadmaps, and industry adoption will likely require some quality standards. This problem will be further discussed in Chapter 6.

Chapter 4

A Programmable Architecture for Accelerating Collision Detection

The contribution we present in this chapter is a novel programmable architecture to accelerate collision detection that can be targeted to any robotic application. First, we discuss the need for flexibility, and why our first approach is not sufficient. We then introduce the improved architecture and the microarchitecture, which we implemented down to Verilog. We simulate the design for functional correctness as well as synthesize it in Synopsys to obtain power/area/timing estimates.

4.1 The Need for a Programmable Architecture

The previous chapter work created a custom hardware microarchitecture [24, 25] and accompanying algorithm to accelerate collision detection. We performed exhaustive collision detection ahead of time and used the data to create a specialized circuit for each motion of the roadmap. The circuits are a way to memoize the collision detection results. The microarchitecture is completely parallel, since there is a dedicated circuit for each edge of the roadmap.

This design achieved significant improvements in latency over the previous state of the art. However, a fundamental limitation of this microarchitecture is that there is no way to adapt it to different robots or scenarios since the specialized circuits are generated for specific motions and a specific robot; new Verilog must be generated for each change of the robot or roadmap.

A programmable architecture is required to address these limitations. One obvious application where the previous design falls short is pick-and-place. If the object being manipulated is large enough that it expands the geometry of the robot (which is the case

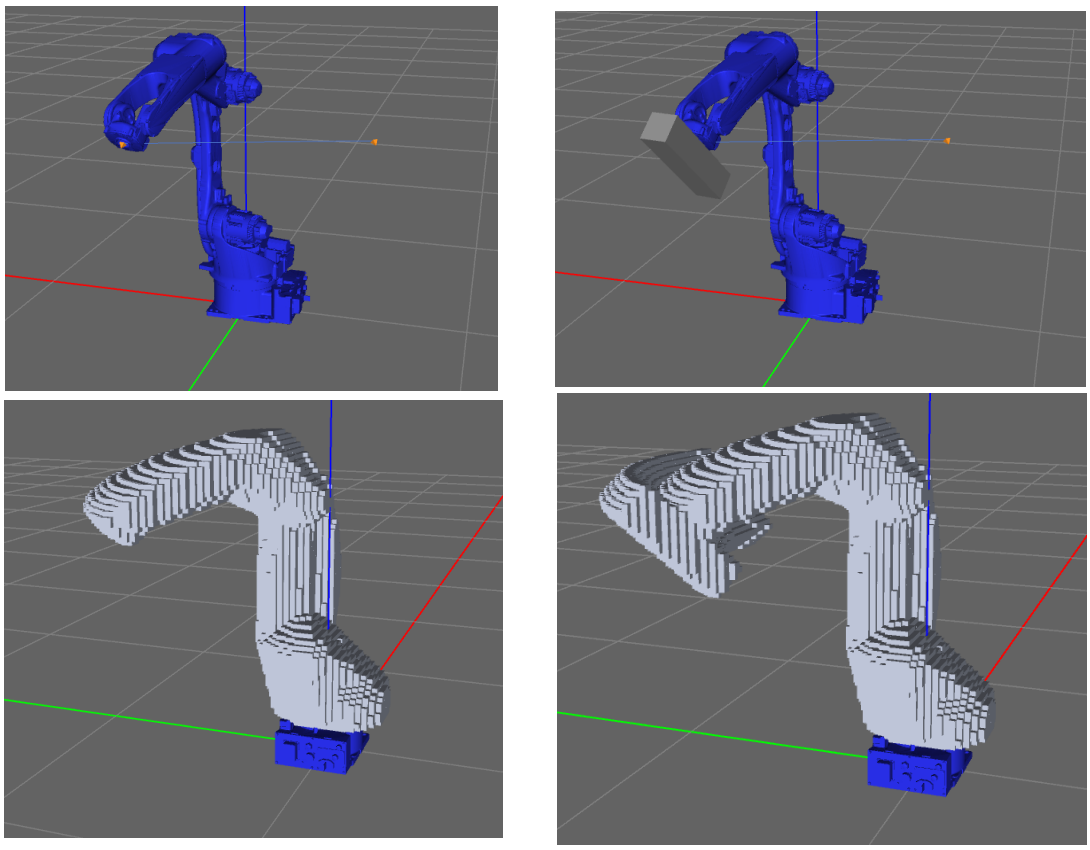


Figure 4.1: The effect of grasping an object on swept volume. On the top is a robot with and without a grasped object. On the bottom are swept volumes for each when the robot executes the same motion.

for most industrial objects), then making a grasp fundamentally changes the configuration space of the robot, and a different roadmap must be employed. The volume swept by each motion is different, and some motions that were previously in the roadmap may now result in a self-collision. This change is illustrated in Figure 4.1. Other robotic applications involve actually changing the end-of-arm-tooling (EOAT) periodically, to complete different parts of a process (perhaps changing from a suction gripper to a parallel-jaw gripper). This limitation could be bypassed by using multiple accelerators or by splitting the hardware resources of a single accelerator over two, three, or **n** roadmaps, but neither of these are satisfactory solutions.

A flexible design also makes an accelerator better suited to developing an ASIC implementation. This would allow larger roadmaps to be employed, due to more effective hardware utilization. We could also take advantage of the higher frequency, lower power, and lower unit-cost benefits of ASIC-hardening.

4.2 A Programmable Architecture

The design we presented in Chapter 2.4 uses precomputed collision sets to build a collision detection circuit for each motion. The input to this circuit is a voxel's binary ID number, and the output is a single bit representing whether that motion is in collision with that voxel. The circuit is logically the sum of products of each voxel ID number in the collision set, and its size can be reduced using logic minimizers. A simplified diagram of this approach is shown in Figure 4.2.

This microarchitecture is very simple and area-efficient, but not at all flexible. The goal of this work is to accelerate collision detection in a programmable fashion. In order for an architecture to be practical, it must be general enough for use in many scenarios by any type of robot. In order to maintain performance, we maintain many aspects of the previous work. In particular:

- We still rely on the construction of a fixed roadmap ahead of time. The roadmap is

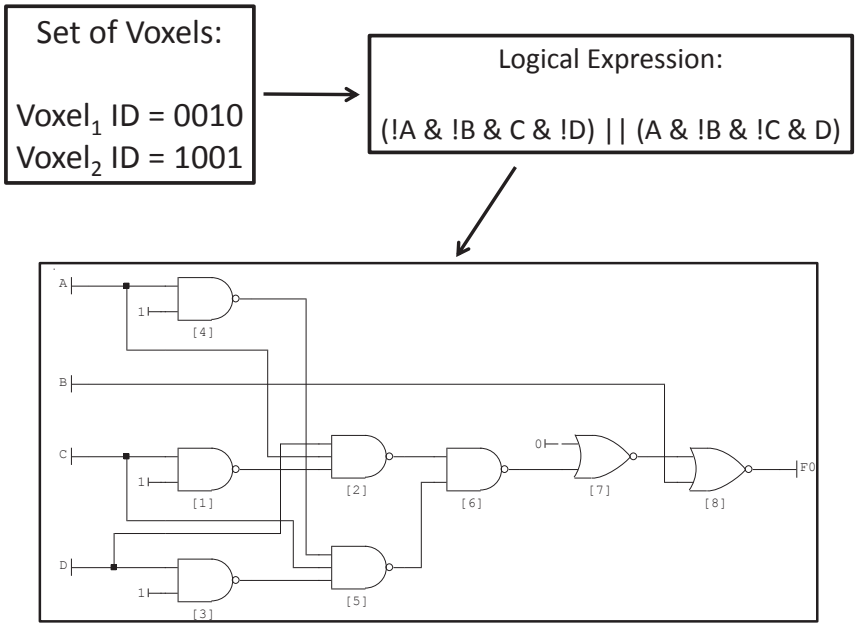


Figure 4.2: The architecture proposed in previous work creates specialized circuits to perform collision detection [25, 24]. Each circuit is fundamentally the hardware sum of products of the ID numbers of the voxels that collide with the given motion. The expression is compressed with a logic minimizer to reduce its footprint. This figure contains a toy example; in a real implementation each voxel would have many more bits in its ID number, and each motion would collide with hundreds or thousands of voxels

made large enough and redundant enough to be robust to obstacles. This allows successive queries to be done rapidly in dynamic environments without reprogramming the accelerator.

- We still leverage extensive preprocessing to avoid runtime computation. The roadmap is exhaustively collision-checked in a discretized view of the environment. However, instead of using the precomputed data to create application-specific circuits as in Figure 4.2, we use it to create configuration files which are used to program the accelerator at runtime.

- At runtime the perception system must still produce the occupancy grid that represents which regions of space in the discretized environment are occupied by obstacles. We stream this data structure to the accelerator, and it outputs a vector indicating which edges are in collision. For all motions in collision, the corresponding edges must be temporarily removed from the roadmap until the environment changes.

4.2.1 Voxel-Based Microarchitecture

Our first attempt at a programmable microarchitecture was very simple. During the programming phase, the user sends the precomputed collision data to the accelerator. Each motion in the roadmap is assigned to an edge module that is essentially a "templated" sum of products. Instead of hardcoded logic functions, the individual terms contain latches to enable any set of voxels to be checked. To achieve parallelism and make the collision checking latency independent of the number of edges, there is a sea of these programmable collision detection circuits, one for each motion in the roadmap.

During this programming phase, each voxel that each motion collides with is sent to the collision detection accelerator, and the voxels are latched in the appropriate edge circuits. The circuit for a single edge can be seen in Figure 4.3. During the runtime phase, the collision detection accelerator streams in voxel IDs and distributes them to all the individual edge circuits. Each edge circuit compares that voxel ID against all of its latches, and

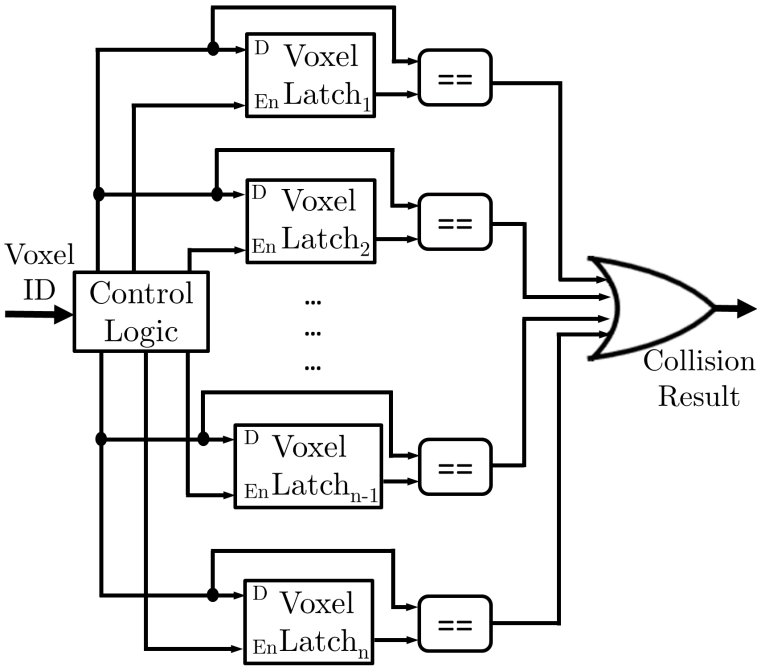


Figure 4.3: A programmable collision detection circuit for a single motion that collides with up to n voxels. During the programming phase, precomputed collision data are sent to the circuit and stored in the latches, orchestrated by the control logic appropriately setting the enable wires. At runtime, obstacle voxels are streamed across and compared with the latched voxels. If any voxels match, the circuit outputs **true.**

outputs **true** if there is any match, which means that specific edge is in collision. The collision detection accelerator aggregates the edge circuit results and outputs a bit vector representing which motions in the currently programmed roadmap are in collision.

This microarchitecture is conceptually simple, easy to validate, and achieves our goal of programmability. However, several challenges are introduced with this feature. Each motion collides with a different volume of space and thus a different number of voxels. This is a problem when implementing a programmable circuit, since we must make the circuits large enough to accommodate edges of various sizes. We can achieve hardware efficiency by creating a sea of variable-sized programmable collision detection circuits. The programming phase then involves a strategic mapping of edges in the roadmap to appropriate collision detection circuits based on the number of voxels with which they collide.

A more important difficulty with this design is the large amount of hardware required. Two main factors cause this strategy to require more hardware resources than a hardcoded design. The state required to store the collision data (and achieve programmability) is significant. If a motion collides with 1,000 voxels, and each voxel is represented with 15 bits, then that single edge consumes 15,000 bits of state, which is quite costly even when using simple latches. The second factor is the inability to use logic minimizers to reduce the size of the circuits, as can be done when working with hard-coded logical expressions. There is a significant amount of redundancy in the expressions both within individual edges and between edges that have overlapping swept volumes. If building a specialized circuit, this redundancy can be exploited to drastically reduce the complexity of the logical expression and reduce the hardware resources its implementation requires. These savings are lost in the programmable case, even though the redundancy is still present.

4.2.2 Boxified Collision Detection

The voxel-latch strategy allows arbitrary combinations of voxels to form the swept volume of an edge. In practice, the voxels of a swept volume are connected and form continuous volumes. The fact that there is order present in the description of a swept volume implies

that there should be ways to compress the data. An alternative to explicitly storing a representation of each individual voxel included in a swept volume could be to store representations of continuous voxels present. In particular, rectangular prisms (for convenience referred to as "boxes") can be represented with only the coordinates of a pair of opposite corners of the box. This can achieve large savings because the state required to store a box is equivalent to the state needed to store two voxels. So each box must only cover more than two of the original voxels to reduce state. This strategy is similar to the common technique in computer graphics of using axis-aligned bounding boxes (AABBs) in that it takes advantage of the simple geometric properties of rectangular prisms [13]. However, AABBs cover a large amount of superfluous volume, and are insufficient if dexterous movements are required in cluttered spaces. We need a more precise solution.

In order to make use of a box-based representation instead of individual voxels, a collection of boxes must be found that includes all of the original voxels. This is analogous to the set cover optimization problem. In the set cover problem, there is a set of elements called the "universe" and a collection of subsets of the universe. The optimization version of the problem involves finding the minimum number of subsets whose union equals the universe. In our case, the swept volume is the universe, and the collection of subsets includes all possible rectangular prisms that overlap with this swept volume. We want to find the minimum number of boxes needed to cover the swept volume in order to reduce the state required by the design. The set cover optimization problem is NP-hard, so we must use an approximate solution [26].

We take a greedy approach to this problem. We first select the box which covers the largest number of voxels in a discretized version of the swept volume without including any extraneous voxels. We remove the covered voxels from the universe (covered voxels become 'don't cares' in subsequent iterations, since boxes can overlap), and recurse until all of the voxels from the original set are covered. We implemented this strategy on a roadmap with 50,000 edges. We found that for a 15-bit resolution each swept volume collides with a mean of 2,000 voxels. Our approximate solution to this version of the set-cover problem required

a mean of 127 boxes to perfectly cover the swept volume (no extra volumes included). The problem becomes more interesting when considering another tradeoff. If we relax the perfect cover requirement and it is acceptable for the box representation to include some number of voxels that were not present in the original swept volume, the number of boxes needed can decrease significantly.

The problem now has multiple dimensions to consider. As a hard constraint, we must cover at least the original swept volume. Under that constraint, we attempt to both minimize the number of boxes required to achieve this cover while balancing the number of extra voxels included. We implement a second greedy process to the algorithm to incorporate this optimization. After the greedy set-cover described above, we run the result through a series of greedy merges. At each iteration the algorithm chooses the two boxes which, when merged, results in the smallest additional volume added to the set. The efficacy of this algorithm is discussed in more detail in Section 4.3.

Being able to store representations of boxes as opposed to individual voxels is only useful if the hardware to check for membership in a box is sufficiently simple to justify this change. Logically, checking if a voxel lies within a box consists of six comparisons. If the vertex of the box closest to the origin is (x_1, y_1, z_1) and the vertex of the box farthest from the origin is (x_2, y_2, z_2), then checking if the voxel located at (v_x, v_y, v_z) is within the box is checking that the following inequalities hold:

$$x_1 \leq v_x \leq x_2, \qquad y_1 \leq v_y \leq y_2, \qquad z_1 \leq v_z \leq z_2.$$

This task can be accomplished in one cycle with six comparators in parallel, or fewer comparators can be multiplexed to use less area. In this work we use six comparators to maximize performance. Figure 4.4 shows the simplicity of this circuit. During the programming phase, the corners of the box are stored in the latches shown. At runtime, incoming voxels are compared against the saved values, and the results ANDed.

Figure 4.5 illustrates how the representation of the swept volume for a single motion in a roadmap changes with these different strategies. Figure 4.5a shows the actual swept volume. Figure 4.5b includes all the individual voxels for the motion. If a hard-coded cir-

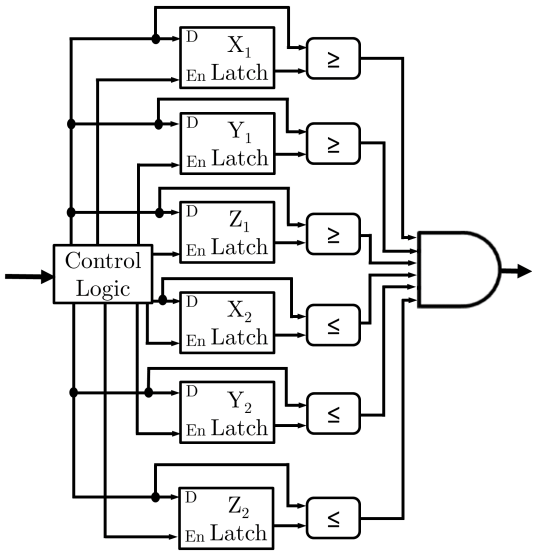


Figure 4.4: Collision detection circuitry for a single box. During programming, the coordinates of opposite corners are latched. At runtime, incoming voxels are compared against the corners to see if they fall within the box. The circuit output for each motion is the logical OR of many such boxes.

cuit is constructed specifically for this swept volume, the motion consumes around 10,000 transistors. If designing a programmable circuit that represents each voxel explicitly as described above, it requires over 660,000 transistors, an increase of over 65x. This same swept volume can be described perfectly using 87 boxes instead of individual voxels, shown in Figure 4.5c. The necessary structures for this strategy consume just under 85,000 transistors. If we allow a 10% increase in volume and represent the edge with only 38 boxes (Figure 4.5d), we can bring the cost down to 36,000 transistors, less than a 4x increase over the hardcoded circuit.

Because we are boxifying the pre-computed swept-volume collision data and not the dynamic obstacles present in any given runtime query, a large number of small obstacles does not degrade the quality of paths. The volume a robot sweeps through space as it moves is by definition contiguous, and so it lends itself well to the boxification during the precomputation phase.

A potential limit of the aggressive boxification strategy would be in applications that cannot tolerate any loss in resolution, perhaps in robotic surgery. In these cases, boxification

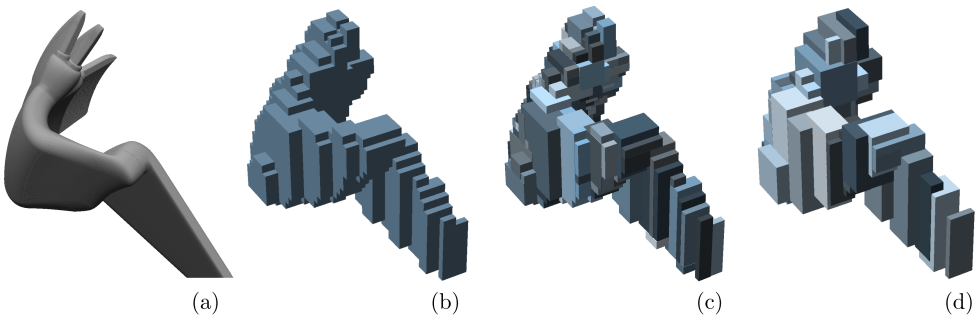


Figure 4.5: (a) The swept volume created by the motion of a robotic arm. (b) This same swept volume represented by voxels. (c) The volume as covered by 87 boxes without any additional volume inclusion. (d) The volume covered by only 38 boxes if a 10% volume increase is allowed.

can still be used, but the parameter for acceptable added volume can be set to 0%. Even at this setting, boxification provides an 85% footprint savings over an uncompressed swept volume.

4.3 Results

We used the Synopsys toolchain and the NanGate 15 nm Open Cell Library [27] to synthesize our design and obtain performance, area, and power estimates. The following numbers are for an implementation with 32,000 edges. We target an ASIC implementation for a number of reasons. With programmability now inherent in the microarchitecture, using a reconfigurable platform such as an FPGA incurs high overhead without benefit. By targeting an ASIC we can achieve larger roadmap sizes at the same area. An ASIC implementation means we can also take advantage of better power efficiency, higher frequency, and lower unit-cost.

Since the collision detection microarchitecture is completely parallel with respect to the edges in the roadmap, its latency in terms of cycles is identical with our previous contribution, and depends solely on the number of obstacle voxels it must process. For the random pick-and-place environments we sampled, there was an average of 750 obstacle voxels, which means collision checking takes an average of 750 cycles, since each voxel

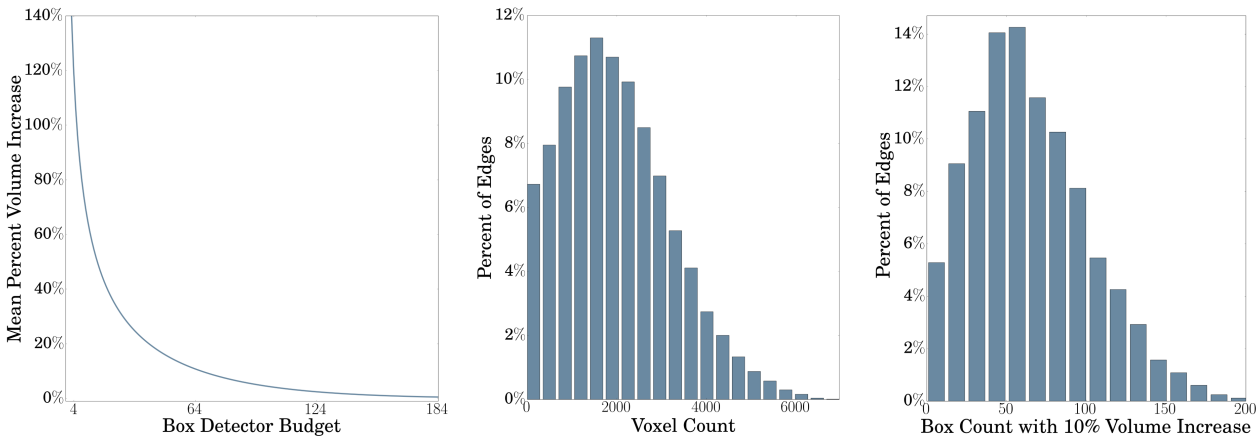


Figure 4.6: As the box detector budget decreases, the volume added increases. At a box budget of 64, the average increase in swept volume is just 11% (left). Histograms of the number voxels composing each edge (middle), and the number of boxes in a representation that allows a 10% volume increase (right).

requires only a single cycle to process. The Synopsys toolchain indicates the design can be clocked at 1 GHz, which means collision detection on average consumes less than a microsecond.

The area consumed by the 32k edge design is just under $400mm^2$, with an estimate of just under 2 billion transistors. As a coarse contrast, the Stratix V used for the implementation of our first contribution consists of 3.8 billion transistors. Using the reconfigurable FPGA fabric to store the hard-coded circuit design, we were constrained by both routing availability and lookup tables to roadmaps of less than 3k edges. This illustrates the area efficiency benefits of ASIC-hardening. Synopsys estimates the power consumption of the design to be 30 Watts.

There are also several aspects of our boxification strategy for performing collision detection that must be evaluated. The first is the technique of aggressively reducing the number of boxes in the swept volume representation in the pursuit of reduced hardware cost. Using our algorithm described in Section 4.2.2, we performed a sweep of the parameter space on a 50,000 edge roadmap to evaluate the trade-off between the hardware budget and the increase in volume of the representation. This tradeoff is visualized in Figure 4.6. We found the average number of boxes to exactly represent the original voxelized volume is

127. However, with a box budget of only 64, the average volume of each representation increases by just 11%. It is important to note that this optimization is safe, since volume is only added, never removed, so a colliding motion will never erroneously be flagged as available. Given that the size of each box representation is equivalent to two voxels, it is clear that representing volumes with boxes is vastly more efficient than with voxels.

Another question is how often this more coarse representation causes a degradation in the ability of the microarchitecture to find paths. We tested a 50,000-edge roadmap against 5,000 sampled pick-and-place environments using both an individual voxel representation and a boxification strategy that allowed a 10% increase in volume. Allowing the 10% increase in volume caused an increase in failure rate of only 0.13%. Of the remaining paths, the average path length increased by 0.59%. This small penalty is more than offset by the 50% decrease in hardware cost compared to a perfect cover using boxes. These results show that our strategy of performing collision detection with box representations effectively reduces the hardware resources required as well as maintains path quality in our example scenario; future work will have to test this strategy on different robots, discretization resolutions, and applications.

4.4 Conclusion

In the previous chapter we developed a specialized architecture to accelerate collision detection. It achieved the desired latency, but was limited to small roadmap sizes, and its implementation was completely specific to a single robot/roadmap pair. This prevented the architecture from being targeted to an ASIC, and made it difficult to apply to applications with changing configuration spaces. In this chapter we presented a contribution that overcomes both of these limitations. This architecture can achieve larger roadmap sizes, and through a programming phase can be targeted to different robots, roadmaps, and changing configuration spaces.

Chapter 5

Hardware-Accelerated Shortest Path Search

The contributions we present in this chapter are a pair of architectures to accelerate graph search in the context of motion planning. We discuss the importance of path search in motion planning and why we should accelerate it. We then introduce an architecture to accelerate path search for a specific planning roadmap. This design could be used in conjunction with the collision detection framework described in Chapter 2.4. We also present a programmable path search accelerator that can be used with the programmable collision detection accelerator from Chapter 4. We simulate the designs for functional correctness as well as synthesize them in Synopsys to obtain power/area/timing estimates.

5.1 The Need for Hardware-Accelerated Path Search

Although path search is not the bottleneck in conventional motion planning algorithms, it can become the slowest phase if collision detection is sufficiently accelerated. In Chapter 2.4 we showed that after implementing our accelerator, graph search became the slowest component by several orders of magnitude. This problem only gets worse as the planning roadmaps get larger.

Having the entire motion planning pipeline operate in the microsecond range is essential in several applications. When planning for an autonomous vehicle in uncertain environments with unpredictable agents it may be difficult or impossible to know exactly where all the obstacles are going. In these cases, you may create a model of possible agent behaviors and run motion planning hundreds or thousands of times while sampling possible outcomes. If the planning latency is in the millisecond time frame, this risk-aware planning

would have an infeasible latency in the seconds. If we can achieve microsecond latency for motion planning, then this becomes a feasible strategy to accommodate uncertainty while being able to maintain a millisecond-level reactivity.

In one recent work that focused on perception latency, the authors investigate the benefits of event cameras to enable drones to avoid collisions with oncoming objects [28]. They show that the difference between millisecond latency and microsecond latency makes a difference for high performance drones making evasive maneuvers [28]. This latency is also required for more complex decision making algorithms [2, 3, 4] that may invoke motion planning thousands of times as a subroutine.

Because graph search is such a broadly applicable problem, previous work has sought to accelerate it, but no prior work is fast enough for our purposes. Several papers have used GPUs to speed up the processing of large graphs but do not achieve real-time performance and are high power [29, 30, 31]. As will be shown in Section 4.3, GPU solutions are not sufficient to bring the shortest path latency in line with accelerated collision detection. There has also been work on specialized processors for accelerating shortest path algorithms. Bondhugula [32] accelerates a block-variant of the Floyd-Warshall algorithm. Sridharan [33] designed a parallelized version of Dijkstra's algorithm, and Takei [34] extended this with large scale graph processing in mind. These approaches achieve insufficient performance for our goals because they rely on slow memory accesses and do not exploit properties of the path search problem that are specific to robot motion planning.

5.2 Accelerating a Specific Roadmap

Our first effort to accelerate shortest path was for a fixed robot roadmap, similar to our first work on collision detection. The approach we took is a dataflow architecture designed to perform the Bellman-Ford algorithm. The architecture consists of a topology of nodes we refer to as Bellman-Ford Compute Units connected to form the same structure as the fixed roadmap.

Algorithm 1 Bellman-Ford Pseudocode

Input: G = (V, E), src
Initialize: $dist[] \leftarrow \{inf\}$, $next\_hop[] \leftarrow \{null\}$
1: $dist[src] := 0$
2: $not\_complete :=$ **true**
3: **while** $not\_complete$ **do**
4: $not\_complete :=$ **false**
5: **for** $< v_1, v_2, weight >$ **in** E **do**
6: **if** $dist[v_1] + weight < dist[v_2]$ **then**
7: $dist[v_2] := dist[v_1] + weight$
8: $next\_hop[v_2] := v_1$
9: $not\_complete :=$ **true**
10: **end if**
11: **end for**
12: **end while**
13: **return** dist, next_hop

The Bellman-Ford algorithm is a single source shortest path algorithm. First, all nodes except the source are initialized to a cost of infinity. At each iteration, every node checks each neighbor for a better shortest path, by adding the neighbor's current cost to the source to the neighbor's edge weight. Pseudocode is shown in Algorithm 1. This algorithm is commonly used in a distributed manner to implement the Routing Information Protocol, with the difference that whole routing tables are communicated, instead of just neighbor costs [35]. Previous work on improving the average-case complexity of SSSP search has resulted in newer algorithms such as delta-stepping [36], but these algorithms do not lend themselves as intuitively to an efficient hardware implementation, and the bounds are not as promising on general graphs [37].

In our approach, each node in our spatial architecture is a Bellman-Ford Compute Unit (BFCU). In this hardcoded implementation, the behavior of each BFCU is quite simple. Each BFCU has a register storing its current best cost and is physically connected to its logical neighbors. When the valid bit on a neighbor's connection is set, the node will add that neighbor's edge weight to the incoming cost, and see if this results in a lower cost than is currently stored. If so, the new cost is written to the register, and the information is propagated by setting its own valid bits and sending the new cost to its neighbors. If the

new cost is not an improvement, the message is discarded. There is also a register to store each BFCU's next hop, and some control logic to enable extraction of the path.

An advantage of this design is that there is very little data to transfer, since the network topology and edge weights are baked into the architecture. This makes transferring cost updates very simple. Communication between the collision detection modules is also straightforward and involves a single bit for each edge, indicating whether that edge is in collision. For an edge that is in collision, each associated BFCU simply sets a bit indicating they should not use that edge until the next reset. Upon reset, collision bits and best cost/next hop registers are re-initialized and another query can be made.

This design has a very small footprint. We wrote Verilog for the BFCUs and Verilog-generating scripts that parse a roadmap description file, instantiate the correct number of BFCUs, and create the connections between them to match the topology of the roadmap. We synthesized the design using the NanGate 15 nm Open Cell Library in the Synopsys toolchain to obtain power, area, and timing estimates. Since the amount of work being done each cycle is so small, the design met timing all the way up to 2.5 GHz, but we will report data from 1 GHz since it is likely this would share a clock domain with the collision detection accelerator. For roadmaps of various sizes this design shows a linear relationship between the area required and the number of nodes. This is shown in Figure 5.1.

The same linear trend holds for power, which at 1 GHz was approximately 0.02 mW for each BFCU. This means that for a roadmap of 16,384 (2^{14}) nodes, a hardcoded shortest path accelerator would be around 1.6 mm^2 and consume less than 400 mW. The total execution time of the module scales with the number of hops across the graph. In the worst case, this grows linearly with the number of nodes in the graph. The common case shows much slower growth. For graphs with around 16k nodes, experiments show our module would complete in less than 1 microsecond.

This work brought the latency of shortest path down to the same order of magnitude as hardware accelerated collision detection. This would allow the two phases of motion planning to be pipelined, generating motion plans at a very high throughput.

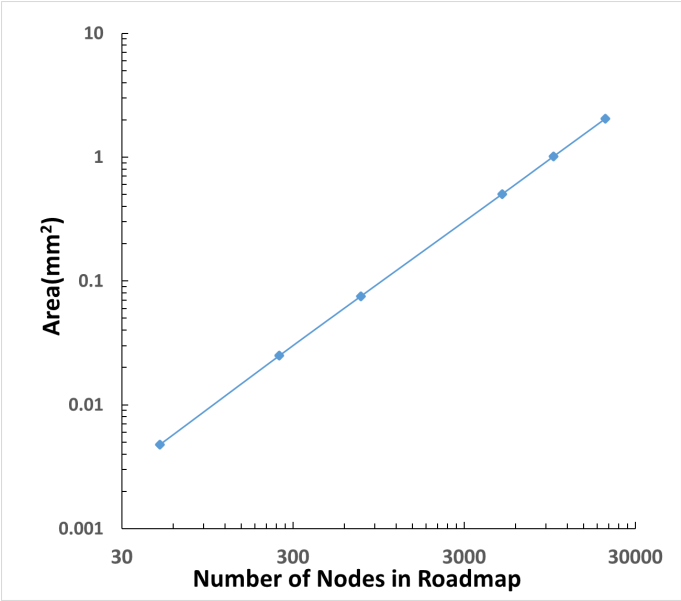


Figure 5.1: The hardware resources needed to accelerate specific roadmaps scales linearly with the number of nodes.

5.3 Programmable Shortest Path Acceleration

The main challenge in designing a programmable architecture to accelerate graph processing is to achieve a design that can handle any expected graph topology, avoid costly memory accesses, and has reasonable resource requirements. Previous work in accelerating graph search has been for more general applications where very few assumptions can be made about the expected input. Our key insight is that since our strategy involves a precomputed static roadmap, we can guarantee certain properties such as its maximum degree, maximum edge weight, and maximum path cost.

Bounding these quantities enables us to design much more compact and efficient storage structures and datapaths than if we allowed for arbitrary graphs. The approach we take in this work is to extend the dataflow architecture to perform the Bellman-Ford algorithm described previously. We augment the BFCUs with additional programmable state, and implement a sea of physical nodes connected via a low-cost interconnection network used to achieve reconfigurable logical topologies.

Operation of the path search microarchitecture falls into three phases. A preprocess-

ing phase involves creating a mapping from the logical graph to the physical hardware resources. The programming phase configures the data structures of the architecture with the necessary information about the graph at hand. During the runtime phase, the distributed Bellman-Ford algorithm runs to calculate the shortest path. We first describe the design and behavior of our programmable Bellman-Ford Compute Units (BFCUs). We then explain how we handle changing roadmap topology with an application-specific interconnection network. Next, we discuss the interface between the path search accelerator and the rest of the system (both collision detection accelerator and host), and the preprocessing necessary to find a good mapping of logical graph to physical resources.

5.3.1 Programmable BFCU Microarchitecture

The functionality of the BFCU discussed here is very similar to what is presented in Section 5.2. The primary difference is the addition of programmable tables describing the roadmap, instead of these properties being baked into the circuitry. Now each compute unit must store the physical addresses of its logical neighbors, so that update messages can be sent to the correct location. The edge weights to each of these neighbors are stored in another table. These tables are filled once during the programming phase, and can be reused many times until switching to a different graph.

Each BFCU also has a register storing its current best cost to the destination. (We treat Bellman-Ford as a single-destination rather than a single-source algorithm.) These registers are all initialized to a maximum value which represents infinity. To start the shortest path search, the BFCU to which the destination node was mapped is updated to a cost of zero. The destination vertex then iterates over its neighbor table, and sends to each neighbor a message with its cost (zero) added to that neighbor's edge weight. When the neighbor receives this message, it compares this new cost with its current cost. If the new cost is less than its current cost, then several things happen. First, the vertex updates its best cost register as well as its next hop pointer. Second, it begins iterating over its own neighbor table to find the physical addresses of its neighbors, and sends each of them

a message with its cost added to that neighbor's edge weight. Figure 5.2 shows the basic microarchitectural layout of the Bellman Ford Compute Circuit.

The decision to send values post-addition instead of pre-addition may seem incidental, but it is important. Receiving a post-addition value allows the BFCU to evaluate whether to process or discard the message in a single cycle, rather than waiting a cycle to access the table with neighbor weights, doing an addition, and then comparing (this would require additional state to save messages arriving during the table lookup). Even though processing a new best cost and iterating over the neighbor table takes several cycles, each BFCU can maintain a throughput of one message each cycle. If the BFCU is in the process of walking through the table to send updates when it receives a new cost update message, there are two possibilities. If the new cost is an improvement, then the as-of-yet unsent messages on this table walk are stale, so the iteration can be aborted and restarted with the new cost. If the new cost is not an improvement, the message is discarded. The uninterrupted ability to handle a message in each cycle eliminates the need for input buffers in the BFCU and means the network can count on being able to eject a message at each BFCU every clock cycle, which will be important when discussing the interconnection network.

Aside from cost update messages, the BFCUs handle two other types of messages as well. If the BFCU receives a next hop query, it responds with the address of the neighbor from which it received its best cost. This class of message allows the path itself to be extracted. The BFCU can also receive a best cost query, to which it responds with the current value of its best cost register.

Several design choices must be made to keep the size of the BFCU small enough that the reconfigurable architecture can scale to large graph sizes. We can leverage properties of our specific application to inform the design. If each node is allowed to have an unbounded number of logical neighbors, for example, the neighbor address table would need to be large. The strategy we use involves precomputing a roadmap, so we can guarantee that each node will have at most four neighbors without affecting the quality of the roadmap. This limitation can be overcome if necessary by logically splitting a node with too many

neighbors into multiple nodes connected with an edge weight of zero. Similar decisions must be made with maximum path cost and edge weight, to bound the size of adders, interconnect width, and registers. Graph edge costs can be scaled to respect these constraints. If an edge is truly needed with a cost greater than the register size allows, it can be represented as two (or more) edges in serial with costs such that the sum is the desired value. The architecture can also accommodate both directed and undirected graphs.

Distributed Bellman-Ford algorithms often implement a split-horizon advertisement policy to allow the network to more effectively deal with failed links [35]. In our architecture, collisions (which are effectively failed links in the roadmap) are communicated before graph search begins, removing the need to keep a split horizon for that reason. Advertising a more costly route back to a node's next hop still unnecessarily occupies network bandwidth. However, our simulations show these redundant messages have a negligible impact on performance (less than 1% increase in path search completion time). This allows us to keep complexity down by not implementing a split horizon.

These decisions all help keep the increase of the area footprint of each BFCU to a reasonable level even when adding reconfigurability. Each node requires around 6,000 transistors. Of this, 70% is comprised of programmable state, while the rest is combinational logic. This modest size makes it feasible to implement the number of compute nodes needed to solve challenging problems. Figure 5.2 shows the basic microarchitectural layout of the reconfigurable Bellman Ford Compute Unit, and Algorithm 2 summarizes the behavior it implements.

5.3.2 Interconnection Network Microarchitecture

In order to execute the Bellman-Ford algorithm, each BFCU needs to be able to communicate with its logical neighbors. However, because the microarchitecture must be reconfigurable, this communication must happen over a network so that the sea of physical BFCUs can emulate the behavior of the desired graph topology. The network enables the BFCU to abstract away communication issues and behave as if it is actually connected to its logical

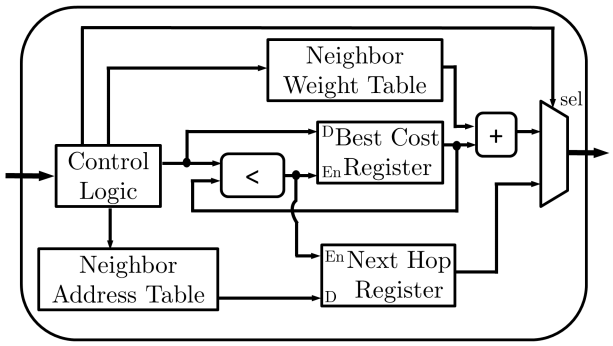


Figure 5.2: Each Bellman Ford Compute Unit (BFCU) has a table of the physical addresses of its logical neighbors, and a table of their edge weights. Behavior described below.

Algorithm 2 Behavior of Bellman-Ford Compute Node

```
1: for each msg received do
2:   if msg.type == COST_UPDATE then
3:     if msg.cost < best_cost then
4:       best_cost := msg.cost
5:       next_hop := msg.source
6:       for each < neighbor, edge_weight > do
7:         send(best_cost + edge_weight)
8:       end for
9:     end if
10:  else if msg.type == COST_QUERY then
11:    send(best_cost)
12:  else
13:    send(next_hop)
14:  end if
15: end for
```

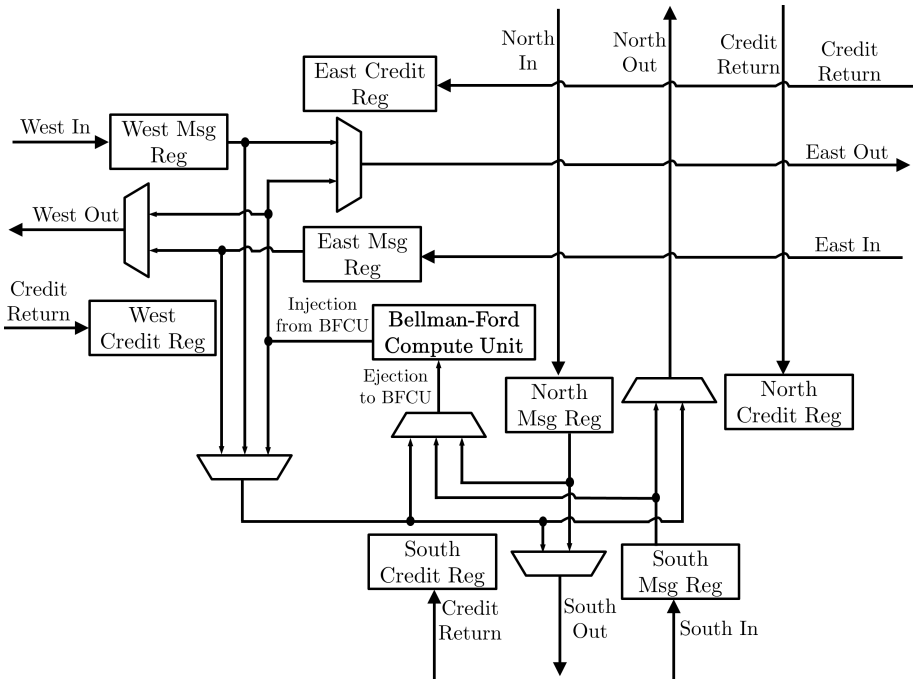


Figure 5.3: Router microarchitecture for our on-chip network. Direction ordered routing along with static in-flight priority effectively decouples the X and Y directions. Arbitration logic is minimal, and only four total buffer entries in the router are required.

neighbors, even though they may not be physically adjacent.

This network must also be efficient both in space and power since a useful design must accommodate roadmaps with thousands of nodes. We based our network on the low-cost router microarchitecture proposed by Kim [38]. The microarchitecture emphasizes simplicity as a first-class constraint to enable scaling to large network sizes. We first explain how this router works, and then present how we make a number of application-specific optimizations to take advantage of properties of our traffic patterns and message sizes.

The crossbar switch is where most of the area is consumed in a typical router. The complexity of a crossbar is quadratic both in terms of ports and link width. The benefits from this design stem from partitioning the router into separate components for the X and Y directions and implementing a direction-ordered routing protocol. In each router, priority is given to in-flight messages that are continuing in the same direction. This allows messages that arrive from and are continuing to travel on the X direction to pass through

the X partition of the router without interacting with any of the components of the Y partition (and vice-versa).

Aside from the crossbar, router input buffers also account for a significant portion of on-chip network power and area. The static priority given to in-flight messages makes arbitration logic trivial, enabling the router to operate in a single cycle. Kim uses this to bring down the number of input buffer entries to two at each port. Maintaining two entries allows back-to-back flits to flow uninterrupted, since the two entries cover the credit return latency [38]. The only other buffer that Kim proposes is a "turning" buffer that fully decouples the two routing directions. When a message arrives at the correct column, it is placed in the buffer to open up resources in the X direction. This necessarily incurs a minimum one cycle turning latency.

When implementing this network for our application, we noted several opportunities for optimization. The first is that because inter-BFCU messages are very small, each message can be composed as a single flit. Since each message is a single flit, multiple flits rarely travel back-to-back. This allows us to maintain performance with only a single buffer entry at each port, since we do not need to cover credit return latency. Each output direction of the router has a single associated credit. The credit starts at 1 and is decremented when sending a message in that direction. The credit is returned out of band directly from the neighboring router once that message has vacated the buffer entry in the neighbor. Implementing this single buffer entry scheme reduces the area by 40%.

Upon further examination, the intermediate turning buffer also appeared to be under-utilized. Instead of helping throughput by freeing resources in the East/West direction, it was simply introducing a turning penalty without benefit. We found that performance was virtually unchanged when removing the turning buffer entirely. This is partly because of the low traffic the distributed Bellman-Ford algorithm creates, and also because the BFCUs can service a message every cycle, so a message very rarely stalls in an input buffer waiting to be ejected (only if two messages arrive for the same vertex simultaneously). This optimization enabled us to further reduce the size of the router by directly connecting the

East and West input buffers to the North/South/Ejection muxes.

Additionally, the properties of our distributed Bellman Ford algorithm allow us to avoid implementing any starvation avoidance protocol, as is typically required with this router [38]. Because new messages in this network are only generated as the algorithm progresses (when better costs are received), a stall at any location in the network is algorithmically guaranteed to eventually cause the rest of the network to quiesce. The quiescence frees up whatever resources are needed for the messages to continue. Our simulations show that these stalls are extremely rare. The microarchitecture for our router is seen in Figure 5.3. The design consumes 7,500 transistors. About 65% of this is the four single-entry input buffers, which shows the importance of reducing the buffer demands of each individual router.

5.3.3 Preprocessing Phase

Key to the strategy of our programmable network is the ability to achieve good performance with a wide variety of logical roadmaps. In order to accomplish this, we must smartly map our logical roadmap configurations to physical BFCUs. To minimize communication latency, logical neighbors are ideally mapped to physically neighboring BFCUs, but this is not always possible due to the roadmap's topology. The physical network of BFCUs is a 2D mesh, while it is unlikely the logical roadmap is planar given that it is created in the high-dimensional configuration space of the robot.

We use a simulated annealing approach to obtain an acceptable solution to this mapping problem during a preprocessing phase. Simulated annealing is a classic technique to search for the optimum of a complicated state space. First, the logical vertices are randomly assigned to locations on the physical mesh. The system is initialized with a certain "temperature" and cooling rate. At each iteration, the system attempts to transition into a neighbor state. We generate neighbor states by randomly selecting a vertex (vertex A). We pick one of this vertex's logical neighbors (vertex B), and randomly select a physical location in the neighboring vicinity of vertex B. The neighbor state is constructed by swap-

ping vertex A with whatever logical vertex is currently mapped to this location. If this new state decreases the system's energy (in our problem defined as the mean physical distance between logical neighbors), it is accepted. If it increases the system's energy, it is accepted with a probability that depends on the current temperature of the system. The higher the temperature, the more likely the system will accept higher energy neighbor states. Accepting higher energy states allows the algorithm to find its way out of local minima. Each iteration, the temperature decreases exponentially at the cooling rate. Annealing took on the order of seconds to minutes, depending on the parameters used.

5.3.4 Programming and Runtime Interface

The usage model we envision for this architecture involves three phases. During the preprocessing phase described previously the user generates and stores various configuration files relevant to different robot/roadmap combinations. Upon switching to a new combination, the accelerator goes through a programming phase, during which control messages flow through the network that send necessary address and edge cost information to each BFCU. Reprogramming is **not** necessary in between queries due to changes in the environment. For these changes (such as the obstacles or goal having moved) a soft-reset restores edges that were flagged as in-collision and the next query can begin immediately.

During the runtime phase the host computer sends perception data to the accelerator, along with source and destination node IDs. The perception data is a stream of which voxels are present in the current environment. The collision detection accelerator calculates which motions are safe and upon completion sends the results to the path search accelerator (without further host-accelerator communication). The path search accelerator modifies the roadmap accordingly by eliminating edges in collision. The path search accelerator then runs and returns a path to the host. The dataflow of the overall architecture including the collision detection accelerator is seen in Figure 5.4.

The interface between the path search and the collision detection modules occurs at points on the interconnection network that we call "control nodes". The control nodes are

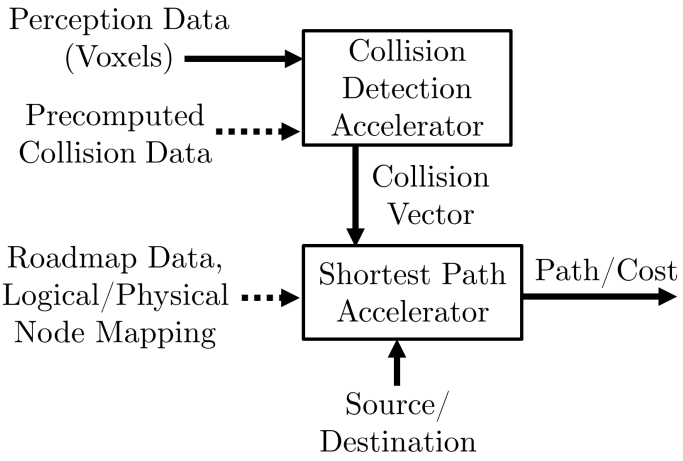


Figure 5.4: The overall dataflow of our architecture. Dotted arrows indicate communication that happens during the programming phase, and solid arrows indicate runtime communication.

located on the perimeter of the interconnection network, as seen in Figure 5.5. The collision detection circuits send a bit vector representing which motions (graph edges) are in collision to the control nodes. For each motion in collision, the control nodes send messages to the BFCUs assigned to the vertices on either side of the edge, indicating that the BFCU should not use that edge for the next query.

The control nodes are also responsible for collecting the path itself upon completion. To this end, parameters are set (during the programming phase) to direct the control nodes how to assess the status of the shortest path search. These include the number of cycles to wait before starting to probe the source vertex's best cost, as well as the conditions that indicate completion. These conditions can be determined with static analysis, as will be discussed more in Section 4.3.

The size of the control nodes is dominated by the storage required to hold the mapping of edge ID to physical addresses of the involved configurations. This mapping enables the control nodes to forward the collision detection results to the appropriate places on the network. If a 128 x 128 mesh is implemented, then each control node consumes almost 190,000 transistors, almost all of which is the mapping table.

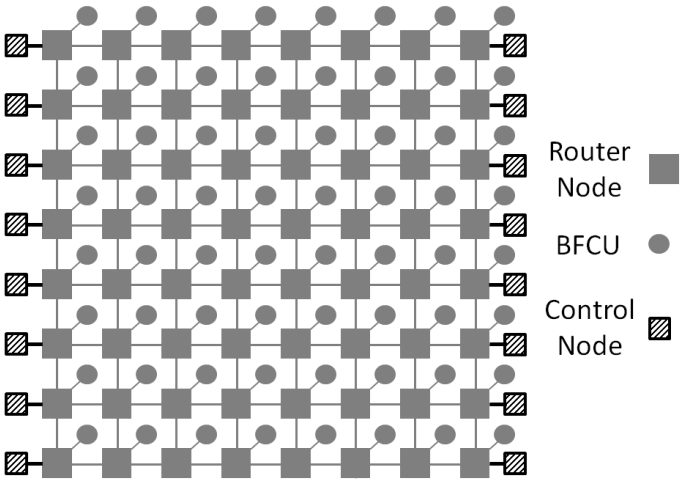


Figure 5.5: The path search architecture consists of a sea of BFCUs on a custom interconnection network. Control nodes sit on the side of the network and are responsible for interfacing to the rest of the chip.

5.4 Results

For our evaluation we generate roadmaps of various sizes for the six degree-of-freedom Jaco II robot arm made by Kinova. We run experiments on sampled environments consisting of randomly placed and sized obstacles and different source/destination pairs. We tested the behavior of the systems solving problems for roadmaps ranging from 4k to 256k edges, but our area and timing numbers will focus on a 128 x 128 implementation solving problems for a 16k-vertex, 32k-edge roadmap. Previous work has shown this size is sufficient to solve challenging problems in the robotics space [11]. We used the Synopsys toolchain and the NanGate 15 nm Open Cell Library [27] to synthesize our design and obtain performance, area, and power estimates.

It should be noted that our results are not unique to the specific application or robot used. Our design can be used in any roadmap-based planning task; this type of planning is used in a wide range of robotic applications including autonomous driving [9], automated inspection [39], and automated machine-tending [40]. We have tested and used different iterations of our accelerator with four different robots, a range of end-of-arm-tooling, and in a variety of scenarios with consistent results. We present results for the Jaco because it

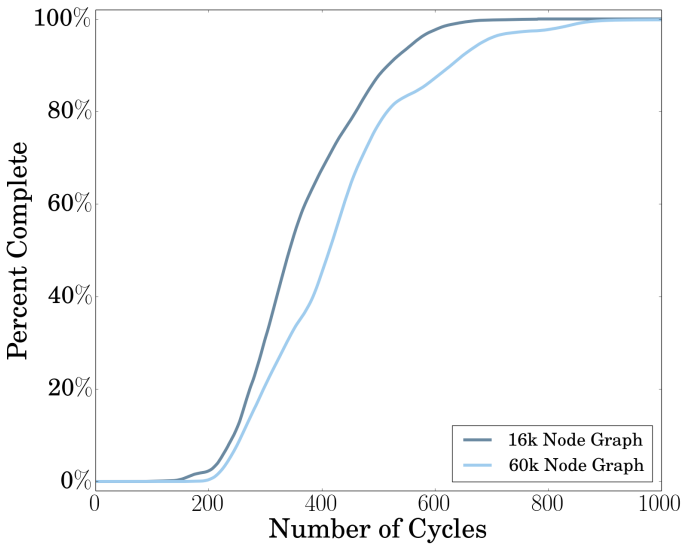


Figure 5.6: Results from simulation showing the distribution of path search completion times for 5,000 trials.

is one of the most widely-used robots in research labs.

To test the effectiveness of our path search microarchitecture, we wrote a cycle-accurate simulator for our interconnection network and the associated BFCUs. Although we have Verilog for the design and tested functional correctness on smaller implementations with an RTL simulator, running thousands of timing experiments for a 16,384-vertex implementation in an RTL simulator is time-prohibitive. The simulator allowed us to quickly explore the design space while developing the microarchitecture as well as efficiently profile the final result.

For the 16k-vertex graph, the mean time to completion is 360 cycles. In addition to the speed of the graph search itself, one aspect of our microarchitecture that must be evaluated is our method of detecting search completion. Figure 5.6 shows the probability of completion at various times for two sizes of roadmap, simulated over 5,000 sampled environments. Using a static analysis of the data to select parameters, we configure the path extraction module during the programming phase to identify completion. For the 16k-vertex graph, the strategy correctly identifies completion over 99% of the time. This comes at an average overhead of 270 extra cycles.

Table 5.1: Breakdown of design size by component for a 128x128 implementation of our architecture.

Component	**Area** (mm^2)	**Transistor Estimate (M)**
Collision Detection	397	1990
Network Routers	24	122
BFCUs	19	97
Control Nodes	10	48
Total	**450**	**2260**

This method is acceptable because while it is not strictly guaranteed that the path search will have completed running when the path is retrieved, it will never return an invalid path. In the less-than 1% of cases where completion was not correctly identified in the example above it simply returns a slightly sub-optimal path. If this is not appropriate for certain applications, the algorithm is guaranteed to quiesce at a rate bounded by the number of vertices in the graph, and a more conservative parameter setting can be used.

Summing the time to both complete path search and detect completion with high accuracy yields a mean of 630 cycles. However, as is common in accelerator design, moving data around takes just as much time as the computation. There is additional overhead of 950 cycles to communicate collisions to the BFCUs and actually extract the path. If we include from Chapter 4 the mean time to perform collision detection, the total average latency is 2,330 cycles from the time the first obstacle voxels arrive, to the time a path is ready for output. Synthesis in Synopsys indicates the accelerator could easily be clocked at 1 GHz, so this equates to a 2.3 microsecond latency. This latency is roughly five orders of magnitude faster than conventional sampling-based planners.

The breakdown of area on the chip is given in Table 5.1. For completeness we include numbers for both a programmable collision detection accelerator (as described in Chapter 4), as well as a programmable Bellman Ford accelerator. The table gives numbers both in terms of area and transistor estimate, in an attempt to make comparisons agnostic to technology size.

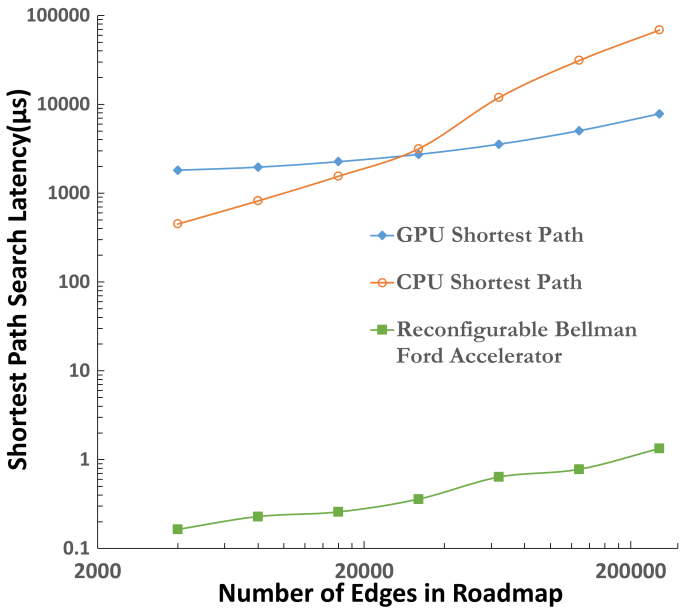


Figure 5.7: Comparison of scaling behavior for shortest path using our proposed accelerator, Nvidia's graph analytics API, and a CPU-based shortest path library.

In total, a 16k-vertex design is 450 mm^2 and requires around 2.3 billion transistors. The majority of space is taken up by the collision detection circuits. The next largest components are the network routers, which are dominated by the four single-entry buffers. Synopsys estimates the power consumption of the accelerator to be 35 watts. Similar to the area results, the majority of the power is consumed by the collision detection circuits, and the programmable network accounts for the rest.

Figure 5.7 shows the scaling behavior of our Bellman-Ford accelerator at different roadmap sizes and illustrates the need for this custom hardware solution. Having dedicated hardware for each node allows the performance to scale linearly with the average number of hops through the graph. Along with our custom hardware solution, we show the performance of shortest path search on a CPU and GPU. The CPU is a 4-core Haswell i7 (16 GB RAM) running the shortest path implementation in Klampt [20], a well-optimized robotics software package. The GPU is running shortest path using the nvGraph graph analytics API [41] on a Tesla K80. Because our microarchitecture involves tightly coupling the shortest path with collision detection, while the GPU involves communication

over PCI-e, no data movement overhead was included for either to be fair (so this figure is strictly concerned with compute time). Even so, the compute time for the GPU is actually slower than the CPU for small graph sizes, crosses over around 32,000 edges, and remains several orders of magnitude slower than our accelerator. This demonstrates that in order to bring the latency of shortest path to the same order of magnitude as accelerated collision detection, a custom hardware solution is needed.

5.5 Conclusions

In this chapter we presented several novel architectures for accelerating path search. One is a very compact and efficient design to enable acceleration of a specific roadmap. This is applicable in domains with very high volume of robots doing the same task, such as autonomous driving. A second design enabled programmability. This flexible microarchitecture consists of a programmable fabric of computing elements that allows fast calculation of shortest paths using a distributed Bellman-Ford strategy. To our knowledge, these are the first accelerators for path search that focus on the needs and characteristics of the motion planning application. Tightly coupling the collision detection accelerator into either fabric brings the total motion planning latency down an additional two orders of magnitude over our previous work that only accelerates collision detection [25, 24].

Being able to perform collision detection and shortest path in under 3 microseconds combined makes it possible to plan under uncertainty or to use complex decision making algorithms. Either of these may invoke motion planning thousands of times as a subroutine [2, 3, 4]. Our architecture allows robots to adjust to dynamic environments in real time, and does not constrain users to adhere to a single use-case. Both collision detection and shortest path search are critical in a myriad of fields such as molecular dynamics simulation, gaming, autonomous vehicles, and augmented/virtual reality. Our architecture provides a general framework for accelerating these tasks that could be used in many such latency-sensitive applications.

The main limitation that remains is that the strategy still depends on generating effective fixed roadmaps ahead of time. Widely used algorithms such as PRM are only effective at handling the current set of obstacles, and do not provide any way to reason about their robustness if the environment changes. This is the open question that will be addressed in the next chapter.

Chapter 6

Roadmap Generation in Dynamic, Semi-Structured Environments

In this chapter we introduce a new method for roadmap generation that can effectively be used with our accelerators. We first discuss the need for intelligent roadmap generation techniques. We then review existing approaches to this problem, and discuss why they are insufficient. We show how we can apply work done in the graph theory and data-mining communities to the robotic motion planning problem. We present a new framework for treating motion planning roadmaps as unreliable networks, with links that can fail at any point in time. We leverage the work done in other fields, adapting data-mining algorithms to generate compact roadmaps from these unreliable networks. We also introduce a set of benchmarks that can be used not only to evaluate our roadmap generation techniques, but any motion planning task.

6.1 The Need for Intelligent Roadmap Generation

The previous three chapters have described how to accelerate various components of motion planning by designing novel hardware architectures. These architectures create highly parallel and efficient dataflow paths that enable computation of collision detection and shortest paths without costly memory accesses and branches. However, the designs we have presented all leverage extensive precomputation. They minimize the size of the problem that must be solved at runtime by assuming that the user can construct a roadmap during configuration of the system.

Construction of a roadmap ahead of time avoids the need to explore high dimensional configuration spaces for each query, when trying to minimize latency. Moreover, it allows the precomputation of expensive collision checks. This effectively turns runtime collision

detection into set lookups, as described in Chapters 2.4 and 4, instead of costly geometric intersection tests. This has the effect of significantly reducing the amount of hardware needed to perform collision detection. Having a known roadmap also allows for an efficient dataflow architecture for path search, as seen in Chapter 5.

The downside of relying on this precomputation is that at runtime the accelerator is limited to a fixed roadmap. Conventional software planners rely on the ability to sample additional nodes *ad infinitum* to attain probabilistic completeness and probabilistic optimality [42, 14]. Using a fixed roadmap in the manner we have discussed does not have this option. Our method is conservative in that it will never return a path that is in collision, but it is possible that it will report failure even when a path exists. This can happen for a number of reasons. Obstacles present in the environment may bisect the precomputed roadmap so that no path can be found from the start to the goal, even though paths may exist if considering the entire free configuration space. The discretization of the environment causes obstacles to appear slightly larger than they are, so it is also possible that a path through the roadmap is safe, but is falsely thought to be in collision. Finally, the precomputed roadmap may not contain good coverage in the areas that the current query demands; even if no obstacles are present, the roadmap may not have any nodes close enough to consider an acceptable goal. If any of these situations occur, our strategy could always fall back on a software planner to regain probabilistic completeness. However, this would negate much of the performance benefits of specialized hardware, so this must be a highly uncommon case. We require a solution that will generate roadmaps that will find high quality paths with high probability.

Most importantly, our roadmap generation techniques must also generate roadmaps that are compact enough to fit on our specialized hardware. It is easy to obtain robust performance on roadmaps that are allowed to have an unbounded size (such as those generated with PRM*). However, the problem is more challenging when trying to achieve this good performance on a budget. The collision detection and shortest path accelerators we presented in the last two chapters have the capacity for 32k edge, 16k node roadmaps.

Regardless of what the exact number is, any hardware accelerator of a fixed size with a real-time latency requirement is limited to solving problems of a certain maximum complexity.

Even without considering dedicated motion planning hardware, having unnecessarily large roadmaps is undesirable for a number of reasons. Graphs with hundreds of thousands or millions of edges are costly to store and load from memory. The roadmap may need to be communicated to mobile robots over a lossy medium such as a wireless network, where huge data structures are not ideal. Most importantly, large graphs take longer to query in software, so smaller graphs are more desirable to bring down the planning latency.

6.2 Robotics-Specific Related Work

There are motion planning algorithms already employed by the robotics community that produce compact data structures. The tree-based algorithms such as RRT and RRT* are designed to quickly produce small solutions [43, 14]. Unfortunately these are single-query roadmaps that are unsuitable for applications where there are many possible start and ending positions. The trees are also by definition sparse graphs, and would not be robust to the presence of dynamic obstacles, since their acyclic nature makes them easy to bisect.

There has been some interest in how to generate small multi-query roadmaps, and this line of work is the most relevant from the robotics community. Bekris notes [44] that the commonly used planning algorithms such as PRM and PRM* often produce exceedingly large roadmaps. This is a direct side effect of the randomized strategy the algorithms use for exploration of configuration space. Although randomness is necessary to achieve their probabilistic completeness guarantees, it has the effect of adding nodes and edges that may be in uninteresting or unimportant regions of the environment. The effect is especially pronounced with the variants that guarantee asymptotic optimality as well as completeness, since these planners must be run for even longer to find paths that approach optimality [45].

Bekris proposes the use of graph spanner algorithms to extract subgraphs of a more reasonable size from very large roadmaps [44]. A graph spanner is a graph with the same set of nodes as the original graph, but only a subset of edges, while maintaining the connectedness of each connected component [46]. A t-spanner is a special class of spanner with the property that for all pairs of vertices (u,v), the shortest path between u and v in the spanner is no more than t-times the shortest path between u and v in the original graph [47]. There can be many possible t-spanners for a given graph. The smallest t-spanner would obviously result in the fastest query time, but unfortunately the problem of finding the smallest t-spanner of a graph is NP-hard [48].

Bekris and Marble employ a randomized algorithm that approximates a (2k-1)-spanner with a time complexity polynomial with respect to k and the number of nodes in the original graph [44]. They found that actual path degradation was much less than the worst case bound of the algorithm. They later extended this work by modifying the k-PRM* algorithm to add fewer edges during construction of the graph [45]. Bekris also proposed a visibility-based criterion to determine whether or not a node should be included in the subgraph [49]. This makes use of the fact that planning algorithms often create roadmaps that have unnecessary node-density in "easy" regions of configuration space while finding ways through the difficult regions. In this work, coverage must still be maintained throughout all configuration space, but roadmaps can be created with fewer nodes than if using a graph spanner.

The Sparse Roadmap Spanner (SPARS) algorithm simultaneously builds up a light-weight roadmap and a more dense roadmap for comparison [50]. The dense version is built up with a vanilla PRM-style approach where each new sample node is connected to all neighbors within a delta-ball. The sparse roadmap includes visibility checks to ensure coverage of C-space is maintained, as well as ensuring quality does not degrade compared to the dense graph by more than a fixed amount. The authors show that this approach creates roadmaps with a bounded path degradation between arbitrary nodes compared to the optimal paths found in the dense graph. They also show that asymptotically the

algorithm will continue to add new nodes to the sparse graph with probability 0 [50, 51]. This demonstrated it was possible to have a finite data structure obtain path qualities that were a bounded factor from optimal, something even more traditional algorithms like PRM* do not provide [52].

Like our work, the graph spanner line of work is focused on creating a compact data structure for multi-query use. However, the motivating scenarios behind it differ from the challenges that we are facing. Bekris identifies four main properties he would like his smaller roadmaps to have. He wants the small roadmaps to produce paths of high *quality* (the paths should not be much longer than the path in the larger roadmap). The small roadmaps should have good *connectivity*, and remain connected if the larger roadmap is connected. Bekris also wants to maintain equivalent *coverage* of the configuration space as in the larger roadmap. Finally, the smaller roadmap should achieve the desired *size*, to allow for fast query [50].

The difference between our needs and those that Bekris' line of work addresses are that the characteristics of connectivity and coverage in his work are defined in the context of static environments. The scenarios in which roadmaps are currently used for multiple queries are ones in which the obstacles are fixed, and just the start and end robot poses change. In this framework, the user can take advantage of the preprocessing of the PRM and simply run shortest path queries to generate each plan. In the setting where the obstacles can move, collision detection must be performed again. When being done in software, performing collision detection for the roadmap is almost as expensive as rebuilding the roadmap from scratch. The visibility based nature of the SPARS algorithm only takes into account a fixed set of obstacles, and does not take into consideration the possibility of dynamic obstacles that may change position. For this reason, it may believe it has sufficient coverage and connectivity in an area because of a specific instantiation of obstacles, when in fact it is not robust at all to other instantiations. Furthermore, spanner algorithms do not leverage any semantic knowledge about what areas of configuration space are more important than others. They maintain coverage and connectivity throughout configuration

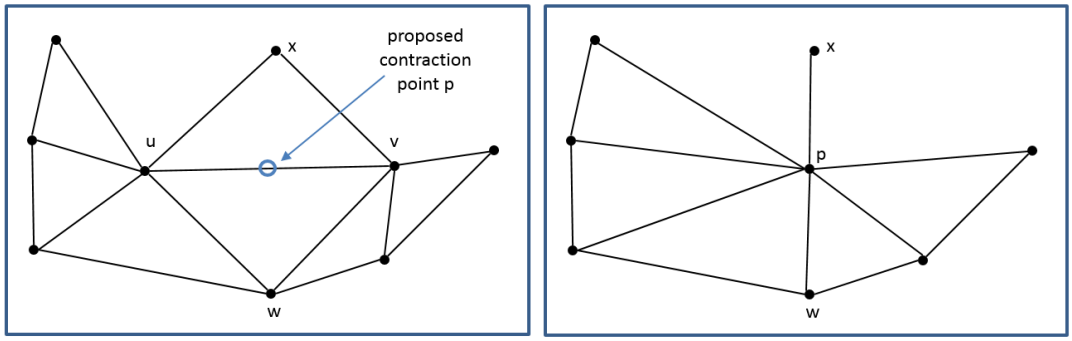


Figure 6.1: A roadmap with edge (**u,v**) being considered for contraction to midpoint **p** (left). The edges resulting from the proposed contraction (right). This contraction would not be legal if there were obstacles causing the new edges to be in collision.

space, when it may be the case that there are large regions of configuration space where it is acceptable to completely lack coverage, allowing a more effective utilization of a fixed-size graph budget.

In another line of work, Shaharabani et al. [53] investigate the feasibility of reducing the size of roadmaps through an algorithm they name "Roadmap Sparsification by Edge Contraction." The idea behind this work is that if a pair of (connected) nodes (**u**, **v**) share multiple neighbors in common, several edges can be removed by contracting the edge between **u** and **v**. Consider the example in Figure 6.1. Edge (**u**,**v**) is being considered for contraction to midpoint **p**. If successful (the new edges are collision free), this contraction reduces the number of edges in the roadmap by three, and the number of nodes by one.

The authors present two metrics for evaluating the sparse graph created by their algorithm. The first is path degradation, defined by how much longer the paths returned by the new roadmap are compared to the old (larger) map. The second is the compression factor, or how much less space is required to store the roadmap. The authors were able to achieve compression factors on the order of 10-30, with path lengths increasing by less than 10%. The ideas discussed in this work are very interesting but have limited application here for several reasons. Like most multi-query roadmaps, their model assumes an unchanging obstacle environment. In this setting it is easier to trim edges and vertices away from a roadmap without affecting quality since you know *a priori* exactly where the obstacles are,

but this becomes a much more difficult problem if the obstacles can move around. This is a similar deficiency to the graph spanner line of work. This technique also relies on vertices having many shared neighbors in common in order to achieve high space reductions, which means the initial roadmap must not only be large but quite dense to realize large compression gains.

6.3 Planning Roadmaps as Unreliable Graphs

With no directly applicable work in the robotics literature, we began looking for analogous problems in other areas. Fortunately, the data structures used in motion planning are not unique. Since motion planning roadmaps are just graphs, we began searching the much larger corpus of research concerning general graph theory.

The main deficiency in the robotics literature was the lack of any consideration of dynamic obstacles and the impact they would have on the ability of the roadmap to successfully find motion plans. In our use-case of a fixed roadmap, we must perform collision detection for each query, invalidate the motions that are unsafe, and hopefully find a path through the remaining roadmap to the goal. If we abstract away from the roadmap the ideas of robot configurations, motions, obstacles, and collision detection then we are simply left with a graph where some of the edges may not be available during any given problem instantiation. Our goal is to take this into account when constructing the roadmap such that it has an acceptably high chance of solving each motion planning challenge.

In graph theory, this construct is known as an uncertain, unreliable, or probabilistic graph (depending on the application domain). To avoid confusion with the probabilistic roadmaps algorithm, we avoid the term probabilistic graph and refer to uncertain or unreliable graphs. The first use of these unreliable graphs was in the context of physical network analysis. The field wished to model the effects of redundant components on overall reliability in complex systems [54]. In these studies, overall system reliability is defined as the chance of there being a path from the input node to the output node given that each

edge (representing a component) is unreliable. From that beginning, the scope of research expanded to study more general problems such as those facing communications networks and power distribution. More recently, these techniques have been used by the data mining community to make sense of the vast amounts of noisy data gathered via social network analysis and gene sequencing. To the best of our knowledge, we are the first in the robotics community to employ this framework to treat the roadmap generation problem as having an unreliable graph with a certain probability of link failure, and trying to leverage knowledge of that probability to create robust, compact roadmaps.

6.4 Unreliable Graph Background/Related Work

Research in this space relies on what is termed the "possible world" semantics of unreliable graphs [55]. This framework treats each edge as a random variable and assumes independence between edges. If edges are undirected, you then have a Markov network that can be sampled from. If a sample is drawn for each unreliable edge, then one "possible world" from this distribution can be constructed. For very small graphs, this means that one could iterate over all possible worlds, see which of these possible worlds maintains a property of interest, and make inferences about the system as a whole. In a graph where the uncertainty lies in the presence or absence of each edge, there are 2^E possible worlds to test, making this brute force approach infeasible for all but the simplest graphs.

The computational complexity of these problems was first rigorously studied by Michael Ball [56, 57, 58]. Ball defines an unreliable network as a stochastic binary system whose state (up or down) is a function of the states of its unreliable components. He noted that exact analysis of the reliability of these general networks was limited to less than 50 nodes [58]. Ball shows that calculating the chance that two nodes are connected in an unreliable network (with arbitrary network topology) is #P-complete, and is in fact also NP-hard [58]. If you severely constrain the network topology, the problem becomes more tractable at the cost of decreased flexibility in network design. The most common class of simplified

graph topologies is the *series-parallel* graph, whose topology makes it very easy to efficiently analyze [58]. A two terminal graph is series parallel if it can be formed by the following rules [59]:

1. A graph with two vertices (s, t) joined by a single edge is series parallel with terminals being s and t.

2. Let G_1 and G_2 be series parallel graphs with terminals (s_1, t_1), (s_2, t_2), respectively. Then the graph $H = G_1 \bigcup G_2$ is series parallel if it is constructed by either:

   (a) Placing the two graphs in series, with t_1 being merged into s_2, and the new terminals being (s_1, t_2).

   (b) Placing the graphs in parallel by merging s_2 into s_1 and t_2 into t_1, with the new terminals being (s_1, t_1).

6.4.1 The Most Reliable Subgraph Problem

The data mining field investigates questions very similar to those we need in robotic motion planning. They deal with very large data sets that may contain many uncertain connections. Effective analysis of these data sets involves pruning away unlikely or unimportant connections, while maintaining critical structures. The resulting smaller data set is easier to store, transmit, and ideally has lost a minimal amount of information during compression. In a 2007 paper the task is formalized as the Most Reliable Subgraph Problem [60]. The authors introduce the problem of identifying which connections are most important for the reliability of the system, and which ones can discarded without a significant impact on reliability. System reliability is defined as the probability that a special set of vertices known as *terminal vertices* remain connected.

Along with minimizing loss in reliability, this problem also has the goal of bringing the size of the graph down by a specific amount. Given an unreliable graph $G = V, E$ and a set of terminals $T \subseteq V$, where each edge in E can fail with some probability, the most reliable subgraph problem is to find the subgraph $H = V', E'$. The subgraph H should

have the properties that $V' \subseteq V$, $E' \subseteq E$, and $|E| - |E'| >= K$, where K is the desired reduction in number of edges. Lastly, it should maintain that $T \subseteq V'$, and it should maximize the probability that the terminals in T are connected [60]. This last property can be formalized by stating that for any other subgraph H' with at most $|E| - K$ edges, $Rel(H) \geq Rel(H')$, where $Rel(G)$ is the reliability of graph G. This version of the task is known as the k-terminal reliability problem. When $k = 2$ it is known as the two-terminal reliability problem, and when $k = |V|$ it is known as the all-terminal reliability problem.

6.4.2 Monte Carlo Pruning Algorithms

Unfortunately, extracting the most reliable subgraph is a challenging problem. Hintsanen [60] showed that the k-terminal most reliable subgraph problem (MRSP) is NP-hard. Just as Ball noted when studying the computational complexity of determining reliability of a fixed graph (a counting problem), Hinstanen notes that the complexity of extracting the most reliable subgraph (an optimization problem) can be simplified by constraining graph topology [60]. In fact, Hinstanen gives a polynomial-time algorithm for the k-terminal MRSP for series-parallel graphs.

For the more general and interesting case of arbitrary graph topology, Hinstanen provides a greedy heuristic that can be performed for the two-terminal case in polynomial time, with no guarantees or bounds on sub-optimality [60]. The heuristic follows a Monte-Carlo-like approach, taking advantage of the possible world semantics. First, the two-terminal reliability of the original graph $G = V, E$ is estimated (since even determining two-terminal reliability is NP-hard). For each edge $e \in E$ there is an associated probability p_e that the edge has failed or not. Flipping a coin for each edge to determine its status comprises one trial. For each trial, the two terminals are checked for connectedness. Performing a number of trials gives an approximation of the reliability of the original graph. This process is then repeated for each edge e on $G' = V, E'$ where $E' = E \setminus e$. Any edge e whose removal does not impact reliability can be removed immediately since this implies that no acyclic path can be found between the terminals that utilizes e. After this step, the edge which affected

reliability the least is removed. The process then begins iterating by calculating the impact on this subgraph of removing each remaining single edge, selecting the one with the least impact, and repeating.

Although this algorithm is straightforward to implement, it has several deficiencies. First is the obvious fact that following the heuristic does not find the most reliable subgraph, and is not even guaranteed to find a good approximation of the optimal solution. Aside from the unbounded sub-optimality, it has the added difficulty of running a very large number of Monte Carlo simulations. The complexity of the algorithm is $O(N|E|^2 + K(|E| + log|E|))$ where N is the number of trials needed during each iteration. It is unclear how large N must be to provide a good approximation of the reliability, which is what guides the pruning procedure. For graphs with hundreds of thousands of nodes it seems N would need to be quite large in order to accurately follow the greedy process.

6.4.3 Incremental Construction Algorithms

Toivonen et al. [59] later presented several improved algorithms for the extraction of reliable subgraphs for the two-terminal and k-terminal versions of the problem. These approaches differ from Monte-Carlo pruning algorithms in that they successively build up larger graphs from paths determined to be useful rather than iteratively removing edges. The first proposed algorithm is named "Best Paths Incremental", or BPI. The premise behind BPI is that it tries to find the most probable/reliable paths between the two terminals. These paths are sorted in terms of reliability, and combined in decreasing order to build up a subgraph. Once the number of edges in the subgraph exceeds the desired edge budget, Monte-Carlo pruning can be used for several iterations to meet the space requirement [59]. This algorithm can be performed relatively efficiently because finding the most reliable paths between terminals is far easier than finding a most reliable subgraph. The probability p_e associated with each edge is simply transformed into weight $w_e = -\log(p_e)$. From here, there are many algorithms that can produce the k-shortest paths with a polynomial time complexity [61]. The BPI algorithm has a complexity of $O((|E| - K)(k^2|V|^2 + k|V||E|)\log(k|V|))$,

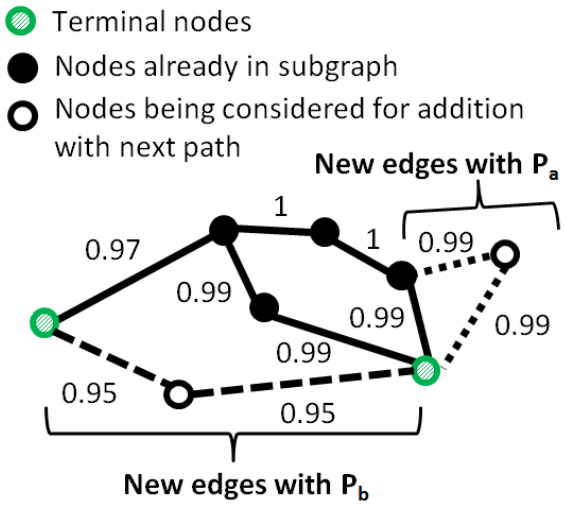


Figure 6.2: Two possible paths that could be added to the subgraph in the next iteration. P_a has a higher path reliability, but does not add as much redundancy to the graph as P_b.

where k is the number of best paths that are needed to create a subgraph of the desired size ($|E| - K$).

Hintsanen et al. propose a solution to the k-terminal version of the most reliable subgraph problem using a framework like Best Paths Incremental (BPI) as the inspiration. They first note that BPI is much easier to implement, faster to run, and provides better paths than simple Monte Carlo pruning, but that its main drawback is that each incremental path is greedily added based solely on its individual reliability, and not the effect adding the edge would have on the whole subgraph's reliability [62]. An example of this distinction is shown in Figure 6.2. This figure shows that even though candidate path P_a has a reliability of 0.95 (simply calculated by $\prod_{e \in P_a} p(e)$) which is higher than the 0.90 reliability of P_b, P_b has a much higher effect on subgraph reliability since it adds more independent links. The problem comes back to the fact that calculating graph reliability is more challenging than calculating the reliability of a single path; as mentioned before, computing graph reliability is NP-hard. There are an exponential number of possible paths we could add, so we have an exponential number of NP-hard problems to solve.

The authors propose a clever algorithm to approximate the problem. They divide their

solution into a *path sampling* and *subgraph construction* phase [62]. Given a current set of paths C (which form a subgraph), the challenge in the path sampling phase is to find the candidate path P that maximizes $R(C \cup P)$. They do this by iterating over all the paths currently in the subgraph, producing "realizations" of its edges until the path fails. A realization is produced by flipping a coin for each edge, and determining if it is available for the current query. Once all paths have failed, the best path from among the un-failed edges of the original graph is added. The subgraph construction phase involves selecting from among the sampled paths in C a subset of paths C' that will meet the desired edge budget, while maximizing reliability [62]. Testing all possible combinations of paths is infeasible, so the authors adopt some heuristics.

The main difference in the author's solution for the two terminal and k-terminal problem is that instead of the first phase sampling candidate paths, it samples candidate spanning trees that connect all terminals [63]. Producing a spanning tree is more involved than producing a path, especially since the graph is uncertain and the spanning trees should have good reliability. To do this, first $(k^2 - k)/2$ candidate trees are initialized, each with the most reliable pairwise path between two of the query nodes.

This algorithm provides a good starting point for how to think about the k-terminal subgraph extraction problem, but lacks several desirable properties. The most important deficit is that it does not consider any form of path quality other than reliability. This reflects the authors' background in data mining. They are concerned with uncertain connections between pieces of data, and not unreliable links that have an additional cost metric associated with them. We will modify this algorithm to incorporate the additional constraint.

6.5 Application to Motion Planning Roadmaps

When confronted with our fixed hardware budget, we explored and modified several of these techniques to produce roadmaps. For the purpose of our discussion, we divide the roadmap

generation problem into two phases.

1. Generation of the baseline graph is necessary for any subgraph algorithm. We need some initial roadmap to either prune down, or to simply compare against as we build up our compact roadmap. The baseline graph will serve as an ideal benchmark and is expected to be quite large. The exact size is not important, and will depend on the application; in challenging scenarios more edges are needed than in simpler ones. Given infinite storage space or hardware budget, the baseline graph is the structure that would be chosen to plan from. The baseline serves both as the input to the subgraph algorithm, and as a comparison to judge effectiveness.

2. The baseline graph B is likely both too slow to efficiently query in software and too large to create a dedicated hardware implementation. Let W be a generator of sample environments, from which we can draw example environments that include obstacles, goals, and starting position. Creating this generator will require the problem scenario be parameterized in terms of all its properties. Our next task is to generate a compact graph from B using a specified edge budget K and the generator W. Given these inputs, the algorithm should output a graph $G = V, E$ where $|E| \leq K$. For a given environment drawn from W, G should be able to produce a collision-free plan with high probability, if a path would have existed in B. Furthermore, the quality of paths should be maintained such that the cost for a solution in a given environment through G is comparable to the cost through B.

6.5.1 Baseline Graph Generation

Generation of the baseline graph is not our primary focus, so we briefly describe it here. We experimented with two main strategies to generate large roadmaps to seed subgraph algorithms. Both involve first modeling the static portion of the environment. This is done so that computation is not wasted by considering in the large roadmap portions of the workspace that are always in collision. The first strategy is to simply run a conventional

sampling-based planner such as PRM or PRM* until the graph reaches a specified size. This has the advantages of providing uniform coverage in the robot's configuration space, and of being a completely general approach.

There are several downsides of relying on a sampling based planner to generate the initial baseline graphs. Depending on the kinematics of the robot, uniform coverage in configuration space may not equate to uniform coverage in the robot's 3D task-space. More importantly, sampling based planners do not take into account any semantic knowledge about the task of interest. While general, failing to leverage this information means that sampling-based planners create roadmaps that have many poses and motions in areas that are completely irrelevant to the application, and may not have especially dense coverage in critical areas.

An alternative is to augment or entirely create a roadmap using knowledge of the robot's task, anticipated obstacles, and workflow. For example, if a robot is being installed to spot weld a part, but the part is subject to some variability in its presenting location, then you can select an assortment of 6-dof tool poses (xyz, rpy) covering the expected range of locations of the part. Inverse kinematics can be used to convert these tool poses to robot configurations, and they can be added to the roadmap. The number of iterations needed from a sampling-based planner to achieve equivalent density in the same area would be enormous. A similar strategy could be used to add additional grasp poses over a bin for a pick-and-place task, additional scanning poses for an automated inspection task, etc. An example of what this may look like for a pick-and-place task is shown in Figure 6.3. An additional benefit of this strategy as it pertains to subgraph algorithms is that a classification of the terminal nodes happens as a side-effect of this process. The primary downside of this approach is that it is not general. Some amount of knowledge about the application of interest is needed to seed the process. This is acceptable because this entire phase is done offline before installation, so this time is not on the critical path.

One of our first experiments illustrated the difference between these techniques for baseline graph generation. Two baseline graphs were generated. One was generated by

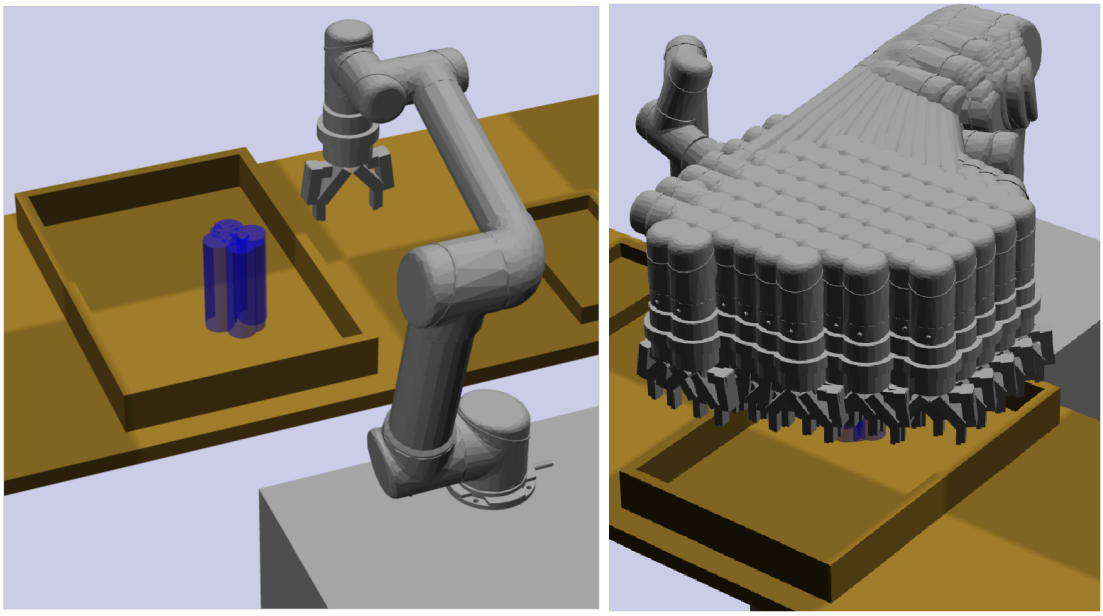


Figure 6.3: In a pick and place scenario there may be a known area of the workspace where the goals are expected to be present (left). We can leverage this knowledge by sampling extra grasp configurations in this region, and augmenting the baseline graph (right).

running the PRM algorithm until the roadmap contained 100,000 edges. The other was also generated using the PRM algorithm, with the exception that the first 250 nodes were not chosen randomly, but were a set of grasp poses chosen above a table for a pick and place task. After these initial 250 nodes, the PRM algorithm was continued until this graph also had 100,000 edges. We implemented a naive Monte-Carlo type approach to prune each baseline graph. We next created a generator of random pick and place tasks. Some parts of the environment would remain constant, such as the table and the fixed-base of the robot, while the locations of the goal and obstacles on the table, the number of obstacles, and their size were all given distributions that could be drawn from.

In order to discover which parts of the roadmap were not contributing to its efficacy, we generated 10,000 pick-and-place environments. For each of these 10,000 environments, the PRM was queried to find the shortest collision-free path to the goal. If a solution was found, each edge in the resulting path was stored. After doing this for the 10,000 environments, we had a structure showing which edges were being used in paths frequently, which were

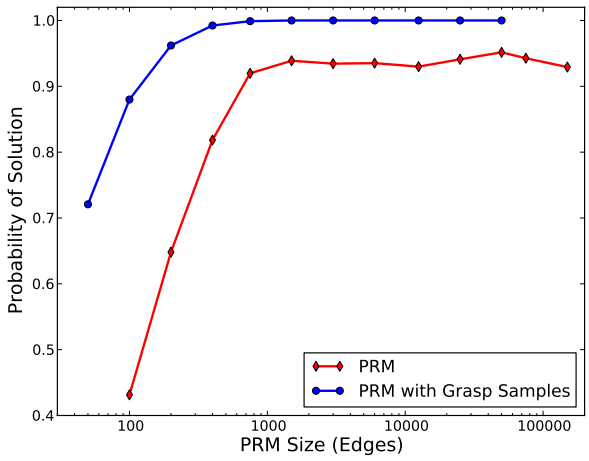


Figure 6.4: The probability of finding a plan when pruning a baseline graph generated solely with PRM (red) or a PRM augmented with extra grasp samples in the picking area (blue).

being used some of the time, and which edges were never used. This enabled us to prune the roadmap by deleting the edges that were never used or used infrequently. This process was done iteratively, profiling and pruning to create subgraphs of a range of sizes, from less than 100 edges to the full 100,000. The resulting subgraphs were then tested by querying each with 1,000 additional random scenarios and calculating a success rate. The results are shown in Figure 6.4. The results show that for both of the baseline graphs, useful subgraphs can be extracted that have less than 1% of the number of edges. However, the roadmap with the additional density of grasp samples can achieve higher success rates at the same edge budget. For the rest of our work, we used baseline graphs that were augmented with increased density in important parts of the workspace.

6.5.2 Extracting Reliable Subgraphs

We drew heavily on the work of Hintsanen et al. [63] on the k-terminal subgraph problem to guide our strategy. We made several modifications to their algorithm to adapt it to meet the needs of the motion planning problem. Next we review the modified algorithm workflow, while outlining the differences in application need and characteristics, and explain

the changes we made.

Estimating Edge Reliability

Let the baseline graph be designated $G = V, E$. All of the reliable subgraph extraction methods found in the literature assume as input that each edge $e \in E$ in the baseline graph has an associated probability $0 \leq p_e \leq 1$. When generating possible worlds, a value v is drawn from a uniform distribution over $[0, 1]$ for each edge; if $p_e \geq v$, then the edge is included in that world. In Hintsanen's work, these probability values represent uncertainties that different genes are connected to each other in metabolic or regulatory pathways. In the motion planning problem they represent each edge's reliability, or likelihood of not being in collision for a given query.

This representation makes estimating p_e logically straightforward, though computationally slow. We take our generator of sample environments, W, from which we can draw example environments, to create some number of sample obstacle sets (we use 10,000). We collision check the set of $e \in E$ for each of these. We sum the number of times that each edge was collision-free, and divide it by the number of trials run. If W is an unbiased generator of obstacles, then this provides an unbiased estimator of p_e for each edge. Because past research has not focused on multi-query motion planning roadmaps in dynamic environments, we are not aware of any previous work that has attempted to estimate edge reliability in this manner.

Defining Source/Sink Terminal Nodes

One fundamental aspect where the motion planning problem has different characteristics than prior work on this problem is in the definition of terminal nodes. Hintsanen's prior work on the k-terminal subgraph problem defines a single class of terminal nodes, and tries to maximize reliability in the subgraph between all terminals. However, we note that in motion planning there are often two or more classes of terminal nodes. We make modifications to the algorithm to accommodate source and sink terminals, and the change

is extensible to a greater number of classifications as well.

This change is helpful because it makes the problem much more tractable. When considering only a single set K of terminal nodes, $K \subset V, \quad k = |K|$, the number of pairwise reliabilities that the algorithm must maximize is $\binom{k}{2}$. In Hintsanen's work, they were considering only a small number of terminals, with k usually less than 10, so this was not a problem. In robotics applications however, we may have a single source terminal but need 500 sink terminals to handle variation in goal location. This would require over 100,000 pairwise combinations if using the algorithm as proposed by Hintsanen. However, in our applications we do not care at all about maximizing reliability between sink terminals, and only need to maintain $source \rightarrow sink$ reliability. This can be achieved with only 500 pairwise combinations. In industrial robotics there are often a set of "teach waypoints" that the robot must go to as part of a work cycle. These positions may move the arm out of the way to allow a 3D camera to perform part localization, for a conveyor belt to move in the next part, or to perform a grasp. These can provide the terminals as input to the algorithm. For this work we assume two sets of terminal nodes S and D, with the following invariants:

- $S \subset V$
- $D \subset V$
- $S \cap D = \varnothing$
- $S \cup D = K$
- In the possible world with all edges in the baseline graph G present, then for each pairwise combination u, v, where $u \in S$ and $v \in D$, there exists a path from u to v through G

Sampling Candidate Trees

One of the largest differences between our application needs and those of Hintsanen is that we must consider not only subgraph reliability, but the quality of the paths the subgraph

contains. Each edge has not only an associated reliability, but also an associated weight that is usually a function of how long it takes the robot to traverse that edge. To accommodate this, at different points in the algorithm we search for shortest paths using different cost functions. Let $Rel()$ indicate a cost function that considers the reliability of each edge, given by $Rel(e) = -log(p_e)$. This is the only metric considered by Hintsanen. We introduce another cost function $Dist()$ which indicates the cost to traverse the edge.

As seen in Algorithm 3 lines 1 to 3, initialization occurs in the same way as Hintsanen [63], except that instead of starting a new tree for every terminal pair, we only include pairs from source terminals to sink terminals. For each pair, the most reliable path is used to initialize a tree. We then enter the main loop of the tree sampling phase, which is terminated once we have built up a specified number of "complete" trees. A tree is considered complete if it contains all the source and sink terminals. Hintsanen et al. note that while the number of candidate trees sampled does have a positive effect on their results, benefits begin to diminish when sampling more than 50 complete trees, and we see similar results in our work.

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

The process of extending a tree is shown in Algorithm 4. If there are source terminals absent in the tree, then a random source terminal out of the missing set is chosen, along with a random destination terminal out of the set already present in the tree; if all the source terminals are already contained in the tree, then one of them is randomly chosen along with one of the destination terminals that is absent (lines 1 to 5). When extending trees, we select best paths among the available set of edges using their distance-based cost function, instead of reliability. This is in contract to Hintsanen et al., who only consider reliability in their study. We find that even though we do not use reliability as a cost

metric for finding best paths, the resultant subgraphs still have high reliability (due to the stochasticity of the algorithm, only edges with high reliability are often in the available set). Before searching for a best path, we also temporarily set the cost of each edge that is *already* in the tree being extended to zero. This biases the algorithm to extend the tree using paths that add a fewer number of edges to the tree.

One major modification we make to the algorithm here is in introducing epochs to the sampling phase. The average reliability of the edges in our application is significantly higher than those investigated by Hintsanen. This has the effect that a subset of edges were being added to many candidate trees, despite there being good alternatives with only slightly higher costs, that could have added useful redundancy in the rare case that the best path experiences a cut event due to obstacles. We counter this effect by allowing edges from the baseline graph to be used in extensions once per epoch. Once used, they are temporarily removed from the baseline graph (line 14). If an extension fails to find a path through the reduced graph in a given realization, then the epoch is reset, and all the original edges are restored to the baseline graph (line 11). We find that this change to the sampling algorithm increases the number of unique edges in the complete trees by 18%.

If no trees are intact, Hintsanen suggests to initialize a new tree with a path that is intact in the current realization. However, we find that since in our application we have many more terminals than considered by Hintsanen, our trees end up much larger (in the thousands of edges), and so there is much less chance that a tree is fully intact in a given realization. We also begin the algorithm with many more trees from the outset, so there is less need to start new trees in the middle of the algorithm. To account for this difference, we make a change. Instead of creating new trees, we "repair" the oldest incomplete tree when we encounter a situation with no intact trees. If the oldest incomplete tree still has paths between all the source and destination terminals it contains, we consider it already repaired and we extend it. Otherwise, we find paths through the available edges from the source to the sink terminals it contains, add the corresponding edges to the tree, and then extend it.

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

Selecting Trees to Construct Subgraph

The next step is to select a subset of the trees sampled. This phase of the algorithm takes as input an edge budget along with the collection of complete trees sampled, and outputs a subgraph. Hintsanen et al. [63] focus only on maximizing reliability, so we present here a modified algorithm that we find maintains reliability while also selecting trees with high-quality paths.

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

After finding the tree that maximizes this quality metric, we remove it from the set of complete trees, and add it to the subgraph. The algorithm terminates when either the edge budget has been exceeded, or the set of complete trees has been exhausted. If the latter

Algorithm 3 SampleTrees

Let $\mathbf{E}(G)$ provide the edges associated with G, and $\mathbf{V}(G)$ the vertices associated with G, whether G is a path, tree, or graph

Let $E_a, E_f \leftarrow$ **RealizeEdges**(E) indicate producing a possible world by drawing from a uniform distribution for each edge, and placing the failed links in E_f and the available links in E_a

Let $P \leftarrow$ **PathSearch**$(E, u, v, func())$ indicate searching for the shortest $u \rightarrow v$ path through edges E using the specified cost function as the metric for the search, and placing the resultant path in P

Let $E \leftarrow$ **ResetEpoch()** indicate restoring all the original edges from the baseline graph to E

Input: Baseline graph $G = (V, E)$, Source terminals S, Sink terminals D, Cost function $Rel()$ that considers the reliability of each edge, Cost function $Dist()$ that considers the cost to traverse each edge, Number of complete trees to sample N

Initialize: $Trees \leftarrow \emptyset$, $CompleteTrees \leftarrow \emptyset$

1: **for** each pair of terminals $< u, v >$ where $u \in S, v \in D$ **do**
2: $P \leftarrow$ **PathSearch**$(E, u, v, Rel())$
3: Add P as a new candidate tree to $Trees$
4: **end for**
5: **while** $|CompleteTrees| < N$ **and** $|Trees| > 0$ **do**
6: $E_a, E_f \leftarrow$ **RealizeEdges**(E)
7: **for** each $T \in Trees$ **do**
8: **if** $\mathbf{E}(T) \subset E_a$ **then**
9: **ExtendTree**(T, S, D, E, E_a)
10: **if** $S \subset \mathbf{V}(T)$ **and** $D \subset \mathbf{V}(T)$ **then**
11: remove T from $Trees$ and place in $CompleteTrees$
12: **end if**
13: continue at line 5
14: **end if**
15: **end for**
16: **RepairTree**$(Trees[0], S, D, E, E_a)$
17: **ExtendTree**$(Trees[0], S, D, E, E_a)$
18: **if** $S \subset V(Trees[0])$ **and** $D \subset V(Trees[0])$ **then**
19: remove $Trees[0]$ from $Trees$ and place in $CompleteTrees$
20: **end if**
21: **end while**
22: **return** $CompleteTrees$

Algorithm 4 ExtendTree

Input: T, S, D, E, E_a

$S' \leftarrow S \setminus \mathbf{V}(T)$
if $|S'| > 0$ **then**
    Randomly select $u \in S'$ and $v \in D \cap \mathbf{V}(T)$
else
    Randomly select $u \in S$ and $v \in D \setminus \mathbf{V}(T)$
end if
Set $Dist(e)$ to 0 for all $e \in \mathbf{E}(T)$
$P \leftarrow$ **PathSearch**$(E_a, u, v, Dist())$
Restore original $Dist(e)$ for all $e \in \mathbf{E}(T)$
if $P == \emptyset$ **then**
    $E \leftarrow$ **ResetEpoch**()
else
    Add the nodes and edges in path P to T
    $E \leftarrow E \setminus \mathbf{E}(P)$
end if

Algorithm 5 RepairTree

Input: T, S, D, E, E_a

for each pair of terminals $< u, v >$ where $u \in S \cap \mathbf{V}(T), v \in D \cap \mathbf{V}(T)$ **do**
    if PathSearch($E_a \cap \mathbf{E}(T)$, u, v, Dist()) == $\emptyset$ **then**
        $P \leftarrow$ **PathSearch**$(E_a, u, v, Dist())$
        if $P == \emptyset$ **then**
            $E \leftarrow$ **ResetEpoch**()
            continue at line 1
        end if
        Add the nodes and edges in path P to T
        $E \leftarrow E \setminus E(P)$
        $E_a \leftarrow E_a \setminus E(P)$
    end if
end for

occurs, the sampling phase can be re-run with tuned inputs intended to provide a larger number of candidate trees and unique edges.

Algorithm 6 SelectTrees

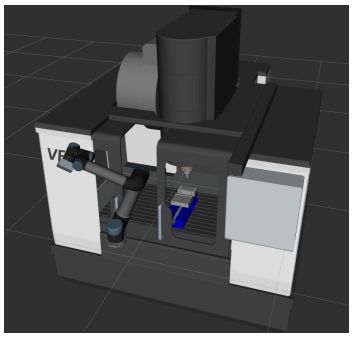
Let $P, C \leftarrow$ **PathSearch**$(E, u, v, func())$ indicate searching for the shortest $u \rightarrow v$ path through edges E using the specified cost function as the metric for the search, and placing the resultant path in P, the cost of the shortest path in C

Input: $CompleteTrees$, E, S, D, $Dist()$, Edge Budget B
Initialize: $SubGraph$ with the tree in $CompleteTrees$ with fewest edges

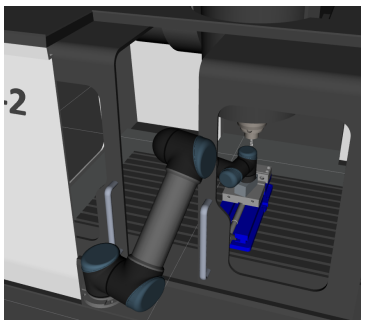
1: **while** $|E(Subgraph)| < B$ **and** $|CompleteTrees| > 0$ **do**
2: $E_a, E_f \leftarrow$ **RealizeEdges**(E)
3: $BestScore \leftarrow 0$, $BestTree \leftarrow \emptyset$
4: **for** each $T \in CompleteTrees$ **do**
5: **if** $E(T) \subset E(Subgraph)$ **then**
6: Remove T from $CompleteTrees$
7: continue on line 4
8: **end if**
9: $sum \leftarrow 0$
10: **for** each pair of terminals $< u, v >$ where $u \in S, v \in D$ **do**
11: $P, C \leftarrow$ **PathSearch**$(E(T) \cap E_a, u, v, Dist())$
12: $P', C' \leftarrow$ **PathSearch**$(E(SubGraph) \cap E_a, u, v, Dist())$
13: **if** $(C' - C) > 0$ **then**
14: $sum \leftarrow sum + (C' - C)$
15: **end if**
16: **end for**
17: $score \leftarrow sum/|E(T) \setminus E(Subgraph)|$
18: **if** $score > BestScore$ **then**
19: $BestScore \leftarrow score$
20: $BestTree \leftarrow T$
21: **end if**
22: **end for**
23: remove $BestTree$ from $CompleteTrees$ and add its edges and nodes to $SubGraph$
24: **end while**
25: **return** $SubGraph$

6.6 Benchmarks for Experimentation

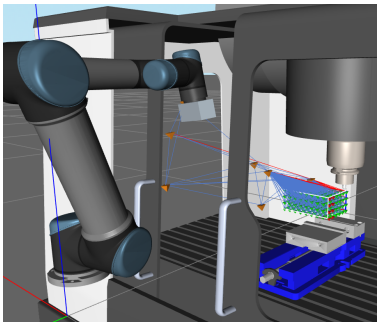
One of the challenges in developing an accelerator for robotics is that the field has a well-known lack of standardization when it comes to system evaluation [64, 65]. In contrast to benchmark-driven sectors such as computer architecture and computer vision, the robotics



(a) An example instance of the machine-tend scenario.



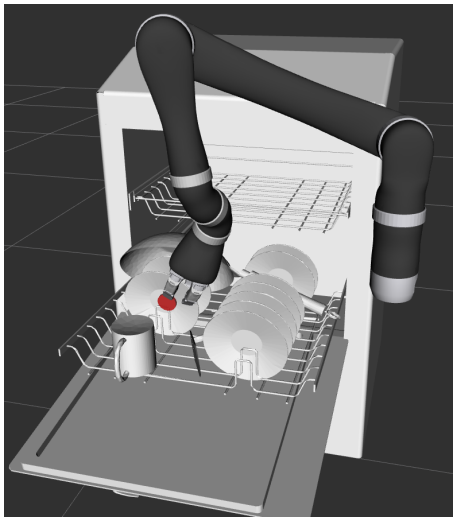
(b) An example solution of the machine-tend scenario.



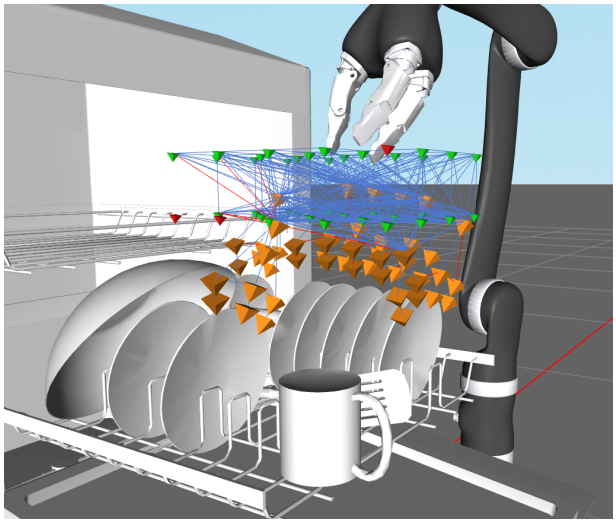
(c) An example roadmap for the machine-tend scenario.

Figure 6.5: The *machine-tend* scenario tries to capture the characteristics common in industrial machine tending tasks. These applications often involve a robotic arm reaching into the enclosed space of a machine to remove machined/processed parts, and then inserting new raw work stock. We include the UR5 robot which is popular for machine tending. In this scenario we define the task as the UR5 reaching into the machine and placing new stock at a randomly defined location. A successful motion for an instantiation is defined as bringing the work part within 1 cm and 5 degrees of the indicated location and orientation.

community has not reached agreement about a set of benchmarks that should be used to measure new ideas and methods. We developed four distinct scenarios that can be used to evaluate robotic planning strategies. Each one uses a different robot to accomplish a different task. We parameterize each scenario and provide a random environment generator so that users can produce unique training and test sets for each scenario. These environment generators have been bundled as a ROS package for distribution to the robotics community as a whole. We also provide sample code so that users can see how to make queries in the environments using already available open-source planning frameworks. Figures 6.5 to 6.8 briefly describe each benchmark. We show some example baseline graphs, but the illustrations are for less dense and/or incomplete baseline graphs, since the high density of vertices and connections makes the baseline graphs used in our experiments hard to visualize.



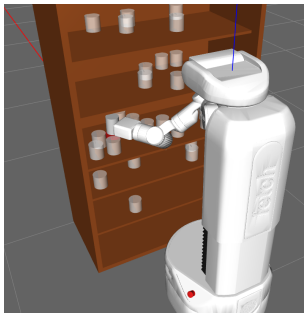
(a) An example instance of the plate-grab scenario.



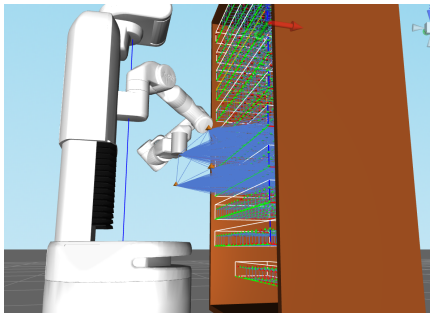
(b) An example baseline graph for the plate-grab scenario.

Figure 6.6: The *plate-grab* scenario is a more domestic task. It involves a JACO robot reaching to grasp a plate in a dishwasher. The robot is randomly placed along any of the three sides of the dishwasher, with one plate being randomly selected as the "goal" plate. We define acceptable goal positions as the palm of the robot facing any part of the rim of the goal plate, with the palm being between 0.5 to 4 cm away from the rim.

(a) An example instance of the shelf-place scenario.

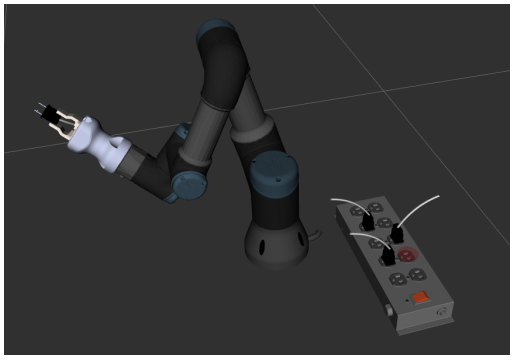


(b) An example solution.

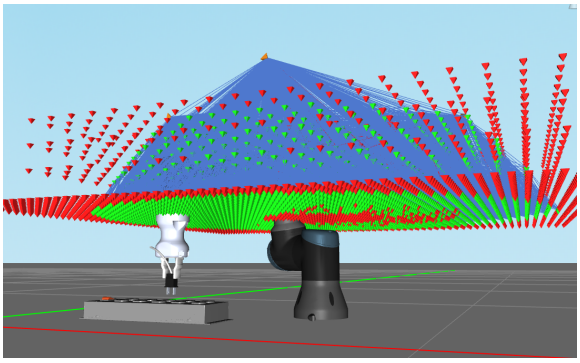


(c) An example baseline graph for the shelf-place scenario.

Figure 6.7: The *shelf-place* scenario models the challenges of picking and placing objects on cluttered shelves, common in both domestic and logistic applications. We use the Fetch robot here, since it contains a prismatic torso joint that allows the robot a large amount of height-flexibility. The scene generator populates a random number of shelves at random heights, and populates them with clutter, along with a specific goal location (indicated as a red disk). Success for this scenario is defined as bringing the can in the Fetch's hand to within 2 cm of the goal position, oriented vertically.



(a) An example instance of the power-strip scenario.



(b) An example baseline graph for the power-strip scenario.

Figure 6.8: The *power-strip* scenario involves a UR3 robot reaching to place a plug into a power strip, without colliding with the wires already present. The power strip is randomly placed around the robot, and is randomly populated with other plugs. One of the free receptacles is labeled the goal (indicated in red), and success is defined as bringing the plug-in-hand to 1-3 cm above the goal receptacle.

6.7 Experiments and Results

We began our experiments by generating training and test samples for each scenario. We use the scenario generators to generate 10,000 environments of each benchmark for training purposes, and a separate 1,000 environments for testing. We then generated a baseline graph for each benchmark, and performed collision detection on the 10,000 training environments to calculate a reliability for each edge, indicating the likelihood the edge will be available for a given query. This serves as the necessary input to the different phases of the subgraph algorithm we describe above. We also chose for all the benchmarks a cost function $Dist()$ that is simply the L2 norm of the vector between the two configurations. We define a subset of the nodes in each baseline graph to act as source and sink terminals. We then ran *SampleTrees* to collect $|S| * |D|$ complete trees. After obtaining the candidate complete trees, we ran the *SelectTree* phase of the algorithm with a range of edge budgets for each benchmark. At this point we have a set of subgraphs of a range of sizes all generated from the same baseline graph. We next use the set of environments designated as the test set and make queries through these environments using the the baseline graph and all the

subgraphs for that benchmark, collecting relevant data. For each benchmark, we present results on two primary metrics. The first is the decrease in path feasibility. This captures the increase in the likelihood that a subgraph could not find a path between a pair of terminals when a path existed in the baseline graph.

One of the first things we noticed when testing our subgraph strategy on these different benchmarks is that in the machine-tend scenario, the UR5 is not robust to dynamic obstacles in between it and the opening to the machine. For this reason, the machine-tend scenario is the only one without dynamic obstacles, having only the goal location vary between queries. This is because we found that with anything obstructing the narrow passageway, our baseline roadmaps were bisected most of the time. Upon seeing this, we tried running conventional sampling-based planners such as RRT and RRTConnect in these scenarios, but they also failed to find a plan within 1 minute of computation time. These results indicate that having dynamic obstacles when picking up stock is likely fine, but having them near a narrow passageway is not a feasible application. It is likely that the kinematics of a 7-DOF arm would allow one to perform better in this situation.

Because this benchmark does not have dynamic obstacles, the effective reliability of all edges in the baseline graph becomes 1. The subgraph problem then reduces to simply finding subgraphs with the fewest number of edges that contain the highest quality paths between terminals. The results for this benchmark can be seen in Figure 6.9. Because this scenario does not have dynamic obstacles, we only present numbers on the effects on path quality. Path quality is unchanged until we decrease the edge budget below the number of unique edges in the union of the shortest paths between each pair of source/sink terminals.

The *plate-grab* scenario was somewhat better suited for our algorithm. The open space above the dishwasher means that there is an area that can afford to be pruned without suffering effects on path feasibility. We only see dramatic increases in failure rates when the edge budget decreases to the point where the "approach edges" that enable direct connections to terminal nodes begin to be excluded from the subgraph. This increases the likelihood that terminal nodes will be bisected by the other clutter in the dishwasher and

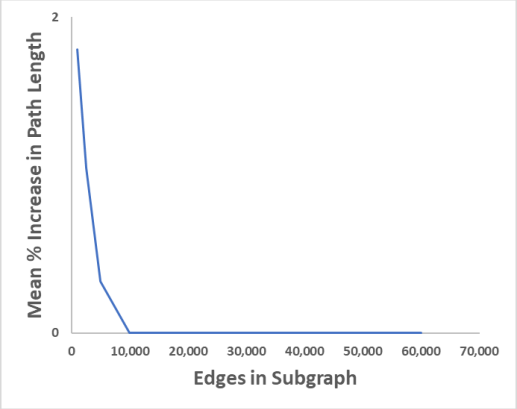


Figure 6.9: Results from the *machine-tend* scenario. The baseline graph had approximately 100,000 edges and 400 terminals.

decreases success rate. Effects on path quality remain modest until reducing the size of the roadmap below 40,000.

The *shelf-place* benchmark was the most challenging of those with dynamic obstacles. It presented difficulties because often the goal location was behind or very close to other objects on the shelves. The baseline graph of 5 million edges had only a 64% success rate. This benchmark also suffered the most from increases in path failure rate. Trying to have a single roadmap cover the entire shelf assembly may be infeasible. In the future we hope to leverage the ability to reprogram our accelerators with different roadmaps. Using this feature, we could divide the shelf assembly into 10-15 regions, and program our accelerator for the specific region relevant for the query.

The *power-strip* scenario was the easiest among those with changing obstacle position. The baseline graph was able to successfully query just over 95% of the test set, and even sampling a subgraph of just 20,000 edges from the baseline graph of 2,000,000 edges had less than a 6% effect on failure rate (an increase to 5.3% from 5%), and an average path length increase of less than 5%.

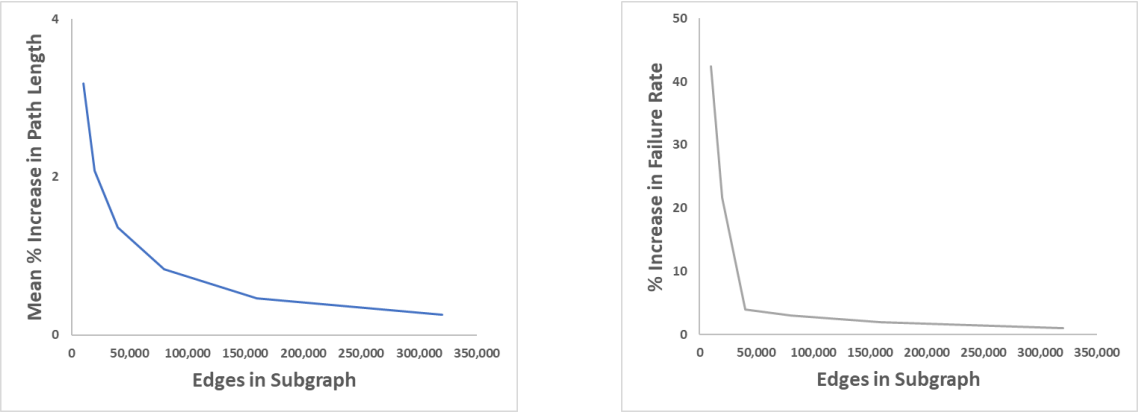


Figure 6.10: Results from the *plate-grab* scenario. The baseline graph had approximately 2 million edges, 5,000 terminals, and a 78% success rate.

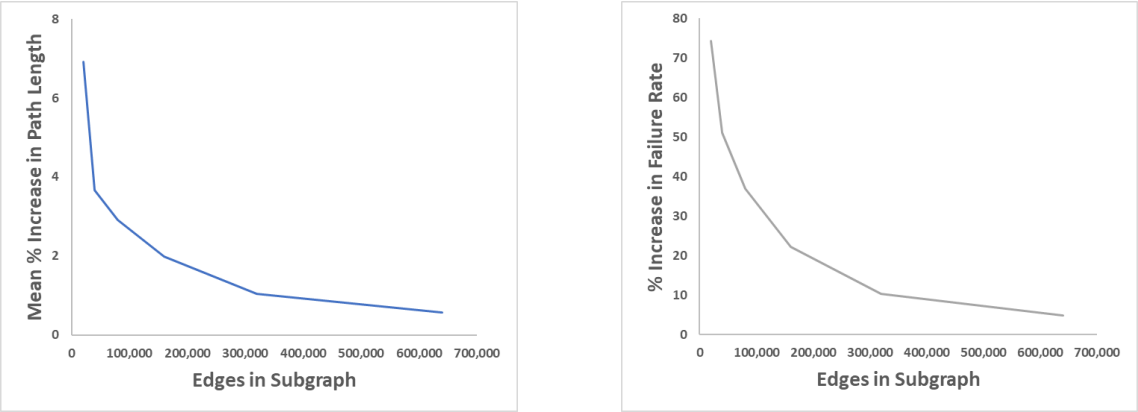


Figure 6.11: Results from the *shelf-place* scenario. The baseline graph had approximately 5 million edges, 11,000 terminals, and a 64% success rate.

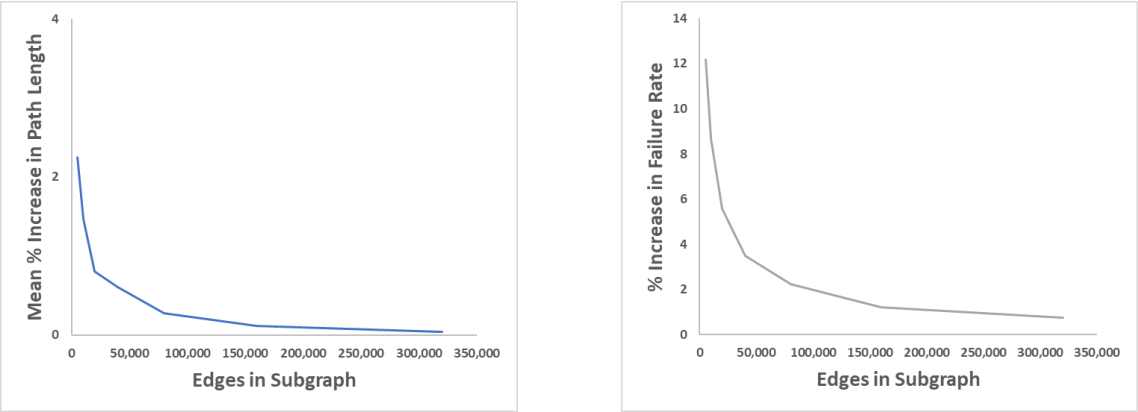


Figure 6.12: Results from the *power-strip* scenario. The baseline graph had approximately 2 million edges, 3,000 terminals, and a 95% success rate.

Chapter 7

Conclusions

Contributions and Insights

The primary concepts guiding our research into the acceleration of robotic motion planning have been parallelism and precomputation. Where possible, we have tried to exploit as much hardware parallelism as possible by designing architectures that fully "unroll" the problems. For collision detection, this involved dedicating a collision detection module for each edge in the roadmap. In Chapter 2.4 this was a combinatorial circuit that represented the collision data for an edge as a complex Boolean expression. In Chapter 4 we added flexibility and scalability to the design by implementing a programmable architecture with novel collision data compression techniques. This allows the design to be targeted to different robots or applications as the situation demands. In both of these designs, the parallelism in the microarchitecture enables an incoming voxel to be collision checked by all edges in parallel, making the time to perform collision detection independent of the number of edges in the roadmap. To the best of our knowledge, no prior work has fully parallelized collision detection in this way.

In both designs we also minimized the size of the run-time problem by leveraging as much precomputation as possible. We exhaustively collision check each edge in our roadmap in a discretized space, constructing a swept volume for each motion. Doing these expensive geometric collision checks ahead of time effectively turns runtime collision detection into set lookups, enabling the simple microarchitectures described in Chapters 2.4 and 4.

For calculating shortest paths, exploiting parallelism involved implementing dataflow architectures to execute the Bellman-Ford algorithm. In Chapter 5 we presented designs tailored to a specific roadmap, as well as a programmable version. In both versions, we achieve parallelism by creating a physical Bellman-Ford compute unit for each node in the

roadmap. Having dedicated compute hardware for each node allows the speed of completion to scale solely with the length of the path to be discovered. We take advantage of precomputation here as well, by performing simulated annealing to find a high-quality mapping of logical nodes onto our programmable architecture, so that logical neighbors are close on the physical network. To the best of our knowledge, ours in the first work to implement a custom accelerator for graph search specifically optimized for the motion planning problem.

In Chapter 6 we explore the precomputation necessary to generate high-quality roadmaps for use on our accelerators. We present a new framework for treating motion planning roadmaps as unreliable networks. We show how by generating representative random environment generators, we can analyze individual edge reliability. Using work from the data-mining community as a guide, we develop a subgraph algorithm that considers not only path reliability, but also maintains path quality. We show that this algorithm can produce roadmaps of appropriate sizes for our accelerator that can solve interesting problems.

Construction of a roadmap ahead of time avoids the need to explore configuration spaces at runtime, when trying to minimize latency. Importantly, it also makes runtime planning in our framework independent of the number of degrees of freedom of the robot. This property is critical when working with robots with more than 7 joints.

In the course of our exploration of motion planning techniques in robotics, we observe that solutions fall onto a spectrum of generality, and the choice of where to fall on the spectrum is usually based on desired performance and a certain tolerance for complexity. On one end of the spectrum is conventional industrial automation. This type of automation involves manually teaching a set of waypoints, and programming a robot to repeat a single trajectory through these waypoints. In these applications the system often has no sensors at all, or a minimum set of sensors to detect catastrophic failure. This type of solution is not general at all, and is extremely brittle. Any robustness to change comes from clever mechanical design of structural jigs and end-effectors. However, these solutions maximize performance in the common case, and keep complexity to an absolute minimum; the number

of components to maintain or that could fail is as low as possible.

On the other end of the spectrum are most solutions proposed by academia. They strive for total generality and completeness. Sampling based motion planners fall into this category. They can solve any solve-able motion planning problem given enough computation time, and do not require any manual teaching of the robot. However, the inputs to this type of training add a significant amount of complexity to the system. Users need accurate CAD models of their robot and environment; this can be a burden because the end-effector may change, the realities of cable management mean there may be flexible tubes hanging off the robot, and other fixtures around the robot may be adjusted periodically. These solutions may provide some asymptotic optimality guarantee, but with realistic computational constraints, will produce solutions of far less quality than what could be accomplished by a human technician. For these reasons, these techniques are almost never adopted by industry.

In our work we try to bridge the gap between these two extremes. We propose strategies that allow the user to leverage *a priori* knowledge of the task at hand to construct useful roadmaps. Our work shows this enables us to make better use of limited hardware budgets. The strategy is not completely general, but can achieve runtime performance that is closer to conventional automation, and also robust to changing environments.

In this body of work we have shown that it is possible to use custom hardware to bring motion planning as a whole to the low-microsecond latency. Our solutions achieve several orders of magnitude speedup over the current state of the art. Being able to perform collision detection and shortest path in under 3 microseconds makes it possible to plan under uncertainty, use complex decision making algorithms, or plan for multiple robots in a workspace. We hope this technology will push robotics into domains and applications that were previously infeasible.

Future Work

The main opportunities for extension of our work lie in improving hardware scalability and the performance of roadmap generation. It seems likely that with clever resource allocation we could handle larger roadmaps on the same size accelerator without incurring much performance degradation. For example, we implemented a completely parallelized microarchitecture for the Bellman-Ford algorithm. However, early simulation we have done indicates we could achieve similar performance while allowing multiple logical roadmap nodes to multiplex a single physical Bellman-Ford Compute Unit. There is also great potential in more sophisticated strategies for both baseline graph generation, and subgraph sampling.

Bibliography

[1] J. A. Marvel and R. Norcross, "Implementing speed and separation monitoring in collaborative robot workcells," *Robotics and Computer-Integrated Manufacturing*, vol. 44, pp. 144 – 155, 2017.

[2] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 639–646, IEEE, 2014.

[3] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Ffrob: An efficient heuristic for task and motion planning," in *Algorithmic Foundations of Robotics XI*, pp. 179–195, Springer, 2015.

[4] J. Wolfe, B. Marthi, and S. J. Russell, "Combined task and motion planning for mobile manipulation.," in *ICAPS*, pp. 254–258, 2010.

[5] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 1–12, 2007.

[6] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[7] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. G. Jeremy Fowers, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[8] T. Lozano-Perez, "Spatial planning: A configuration space approach," *IEEE Transactions on Computers*, vol. C32, no. 2, pp. 108–120, 1983.

[9] B. Paden, M. Cáp, S. Z. Yong, D. S. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *CoRR*, 2016.

[10] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[11] R. Bohlin and L. E. Kavraki, "Path planning using lazy prm," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, 2000.

[12] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[13] C. Ericson, *Real-Time Collision Detection*. Boca Raton, FL, USA: CRC Press, Inc., 2004.

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

[15] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[16] N. Atay and B. Bayazit, "A motion planning processor on reconfigurable hardware," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 125–132, 2006.

[17] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *In Proc. IEEE Int. Conf. Robot. Autom. (ICRA*, pp. 688–694, 1999.

[18] S. Lian, Y. Han, X. Chen, Y. Wang, and H. Xiao, "Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 109:1–109:6, ACM, 2018.

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

[20] K. Hauser, "Robust contact generation for robot simulation with unstructured meshes," in *Proceedings of the International Symposium on Robotics Research*, 2013.

[21] D. Nieuwenhuisen and M. H. Overmars, "Useful cycles in probabilistic roadmap graphs," in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 1, pp. 446–452 Vol.1, 2004.

[22] R. Brayton, G. Hatchel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer Academic Publishers, 1984.

[23] R. Rudell, "Multiple-valued logic minimization for pla synthesis," Tech. Rep. UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.

[24] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, "The microarchitecture of a real-time robot motion planning accelerator," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

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

[26] C. Lund and M. Yannakakis, "On the hardness of approximating minimization problems," *Journal of the ACM*, pp. 960–981, 1994.

[27] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, 2015.

[28] D. Falanga, S. Kim, and D. Scaramuzza, "How fast is too fast? the role of perception latency in high-speed sense and avoid," *IEEE Robotics and Automation Letters*, vol. 4, pp. 1884–1891, April 2019.

[29] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, 2007.

[30] G. J. Katz and J. T. Kider, Jr, "All-pairs shortest-paths for large graphs on the gpu," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2008.

[31] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "Phast: Hardware-accelerated shortest path trees," *Journal of Parallel and Distributed Computing*, vol. 73, 2013.

[32] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan, "Hardware/software integration for fpga-based all-pairs shortest-paths," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[33] K. Sridharan, T. K. Priya, and P. R. Kumar, "Hardware architecture for finding shortest paths," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, 2009.

[34] Y. Takei, M. Hariyama, and M. Kameyama, "Evaluation of an fpga-based shortest-path-search accelerator," in *The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing*, 2015.

[35] Y.-C. Hu, D. B. Johnson, and A. Perrig, "Sead: secure efficient distance vector routing for mobile wireless ad hoc networks," *Ad Hoc Networks*, pp. 175 – 192, 2003.

[36] U. Meyer and P. Sanders, "Delta-stepping: A parallel single source shortest path algorithm," in *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, Springer-Verlag, 1998.

[37] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan, "Parallel shortest-paths using radius stepping," *CoRR*, 2016.

[38] J. Kim, "Low-cost router microarchitecture for on-chip networks," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[39] M. Ulrich, G. Lux, L. Jurgensen, and G. Reinhart, "Automated and cycle time optimized path planning for robot-based inspection systems," *6th CIRP Conference on Assembly Technologies and Systems (CATS)*, pp. 377 – 382, 2016.

[40] R. Bohlin, "Motion planning for industrial robots," *PhD Thesis, Chalmers University of Technology*, 1999.

[41] Nvidia, *nvGraph API Reference*. http://docs.nvidia.com/cuda/nvgraph/: CUDA Toolkit Documentation, 2017.

[42] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

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

[44] J. D. Marble and K. E. Bekris, "Computing spanners of asymptotically optimal probabilistic roadmaps," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4292–4298, 2011.

[45] J. D. Marble and K. E. Bekris, "Asymptotically near-optimal is good enough for motion planning," in *Robotics Research : The 15th International Symposium ISRR* (H. I. Christensen and O. Khatib, eds.), (Cham), pp. 419–436, Springer International Publishing, 2011.

[46] D. Peleg and A. A. Schäffer, "Graph spanners," *Journal of graph theory*, vol. 13, no. 1, pp. 99–116, 1989.

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

[48] M. Elkin and D. Peleg, "Strong inapproximability of the basic k-spanner problem," in *Automata, Languages and Programming: 27th International Colloquium* (U. Montanari, J. D. P. Rolim, and E. Welzl, eds.), pp. 636–648, 2000.

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

[50] A. Dobson, A. Krontiris, and K. E. Bekris, "Sparse roadmap spanners," in *Algorithmic Foundations of Robotics X* (E. Frazzoli, T. Lozano-Perez, N. Roy, and D. Rus, eds.), (Berlin, Heidelberg), pp. 279–296, Springer Berlin Heidelberg, 2013.

[51] A. Dobson and K. E. Bekris, "Improving sparse roadmap spanners," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Karlsruhe, Germany), 2013.

[52] A. Dobson and K. E. Bekris, "Sparse roadmap spanners for asymptotically near-optimal motion planning," *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 18–47, 2014.

[53] O. Salzman, D. Shaharabani, P. K. Agarwal, and D. Halperin, "Sparsification of motion-planning roadmaps by edge contraction," *The International Journal of Robotics Research*, vol. 33, no. 4, pp. 1711–1725, 2014.

[54] E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," *Journal of the Franklin Institute*, vol. 262, no. 3, pp. 191–208, 1956.

[55] C. C. Aggarwal, *Managing and Mining Uncertain Data*. Springer Publishing Company, Incorporated, 2009.

[56] M. O. Ball, "Computing network reliability," *Operations Research*, vol. 27, no. 4, pp. 823–838, 1979.

[57] M. O. Ball, "Complexity of network reliability computations," *Networks*, vol. 10, no. 2, pp. 153–165, 1980.

[58] M. O. Ball, "Computational complexity of network reliability analysis: An overview," *IEEE Transactions on Reliability*, vol. 35, no. 3, pp. 230–239, 1986.

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

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

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

[62] P. Hintsanen, H. Toivonen, and P. Sevon, "Fast discovery of reliable subnetworks," in *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*, pp. 104–111, IEEE, 2010.

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

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

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

[66] T. Wiedemeyer, "IAI Kinect2," 2015. Accessed June 12, 2015.

[67] S. Niekum, "ar_track_alvar," 2011 – 2015.

[68] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using GPUs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2010.

[69] Wikipedia, "Aircraft principal axes — wikipedia, the free encyclopedia," 2016. [Online; accessed 9-March-2016].

[70] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using GPUs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2010.

[71] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," *International Journal of Robotics Research*, Feb. 2012.

[72] A. Yershova and S. M. LaValle, "Improving motion-planning algorithms by efficient nearest-neighbor searching," *IEEE Transactions on Robotics*, 2007.

[73] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation*, 2000.

[74] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with rrts," in *IEEE International Conference on Robotics and Automation*, 2006.

[75] W. Wang, D. Balkcom, and A. Chakrabarti, "A fast online spanner for roadmap construction," *The International Journal of Robotics Research*, 2015.

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

[77] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[78] L. Liu, *Discovering Reliable Communities In Uncertain Graphs*. PhD thesis, Kent State University, 2015.

[79] R. Jin, L. Liu, and C. C. Aggarwal, "Discovering highly reliable subgraphs in uncertain graphs," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, 2011.

[80] R. Jin, L. Liu, B. Ding, and H. Wang, "Distance-constraint reachability computation in uncertain graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 551–562, 2011.

Biography

Sean Murray attended Rice University in Houston. He graduated in 2011 with a B.S. in Chemical Engineering. He worked for several years in industry, before matriculating at Duke University in the Fall of 2014. He earned an M.S. in Electrical and Computer Engineering in 2016, and his Ph.D in the same department in 2019. He was awarded the Kristina M. Johnson Fellowship in 2018. His research focuses on how computer architecture principles can be applied to accelerate robotics applications. In particular, he focuses on designing practical systems that make realtime motion planning feasible.