MOTION PRIMITIVE PATH PLANNING FOR PARAMETRICALLY UNCERTAIN SYSTEMS VIA THE KOOPMAN OPERATOR

A Dissertation
Presented to
The Academic Faculty

By

Geordan Mihalick Gutow

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
George W. Woodruff School of Mechanical Engineering
Robotics Program

Georgia Institute of Technology

May 2022

© Geordan Mihalick Gutow 2022

MOTION PRIMITIVE PATH PLANNING FOR PARAMETRICALLY UNCERTAIN SYSTEMS VIA THE KOOPMAN OPERATOR

Thesis committee:

Dr. Jonathan Rogers
Aerospace Engineering
Georgia Institute of Technology

Dr. Seth Hutchinson
Interactive Computing
Georgia Institute of Technology

Dr. Anirban Mazumdar
Mechanical Engineering
Georgia Institute of Technology

Dr. Charles Pippen
Senior Research Scientist
Georgia Tech Research Institute

Dr. Panagiotis Tsiotras
Aerospace Engineering
Georgia Institute of Technology

Date approved: April 26, 2022

Far better to light the candle than curse the darkness.

William Watkinson

For my grandparents. Always ready to help.

ACKNOWLEDGMENTS

It takes a village to raise a child; similarly, no thesis is the work of one person alone. First and foremost, I need to thank my parents, for far too many things to list here. For their guidance (both technical and otherwise), for their advice, for patiently listening to me babble, for telepresence kitchen troubleshooting. For instilling in me a love of learning and the belief that the pursuit of knowledge is a worthy goal.

This thesis would not exist with out the guidance, advice, and example of Jonathan Rogers. If Newton stood atop the shoulders of giants, than I feel as though I had a Hyperion (a name sometimes translated as "he who goes before"), a titan shining a light to guide my way. From day one, he helped me see that there was a path forward and that all I had to do was walk it. Even as the road meandered at times, and occasionally dead-ended, I always knew where I was going and believed that I could get there. Thank you for that gift.

The technical foundations of this work owe much to the efforts of Andrew Leonard and Joey Meyers, senior lab members whose analysis and demonstration of the Koopman operator for uncertainty quantification made everything that follows possible. I am particularly indebted to Andrew, who gave me access to his code when I was a shiny new PhD student and helped me see what all this "operator" nonsense was actually *for*. I also want to acknowledge the contributions of Zachary Goddard, Anirban Mazumdar, and Panagiotis Tsiotras, all of whom lent their time and expertise to aspects of my research.

My labmates, both past (Andrew Leonard, Evan Davies, Jared Elinger, Kevin Webb, Brian Eberle, Umberto Saetti, Katherine Skinner, Dakota Musso) and present (Joey Meyers, Adam Garlow, Sam Kemp, Anika Kansky) were invaluable as sounding boards, homework helpers, editors, role models, and advice givers (even if some of it was...questionable). Shoutout in particular to Brian, who taught me a lot about how to stay happy and productive over the long haul through his sheer...Brian-ness. To the rest of you: if I had to pick one thing I regret, it would be the way the pandemic scattered us all to the four corners and the

loss of cross-pollination of ideas that caused. So naturally, just as we start to come back together, I up and graduate.

I would never have made it to Georgia Tech and a PhD program were it not for the encouragement, advice, and backing of Tamer Zaki, Kaliat Ramesh, Noah Cowan, Dan Stoianovici, Vignesh Kannan, Meng Zhao, Christopher Ratto, Kara Shipley, Jonathan Ligo, and Erez Krimsky.

I have the incredible good fortune to have my grandparents, aunt, and uncle in Atlanta. Since the moment I accepted the position at Georgia Tech, they have gone out of *their way* at every turn to smooth *my way*. If I were to list everything they have done to help, every way they have made my time here happier or healthier, it would be longer than the thesis itself. So, thank you. A thousand times, thank you.[1]

[1] But no, I don't need a ride to the grocery store.

TABLE OF CONTENTS

Acknowledgments v

List of Tables x

List of Figures xi

List of Acronyms xiii

Summary xiv

Chapter 1: Introduction 1

Chapter 2: Motion Primitives via the Koopman Operator 12

2.1 Mathematical Background 12

2.1.1 Uncertain Maneuvers 13

2.1.2 Expected Values and the Koopman Operator 14

2.2 The Chance-Constrained Uncertain Planning Problem 16

2.3 Efficiency of Koopman versus Monte Carlo Methods 18

2.3.1 Expected State Planner 19

2.3.2 Path Through a Gap 21

2.3.3 Mobile Obstacles 23

2.3.4 Scylla and Charybdis 24

2.4 Discussion 26

Chapter 3: Optimal Search Algorithms 30

3.1 Algorithmic Contributions 30

3.1.1 Induced Heuristic for AO* 30

3.1.2 State Merging 36

3.1.3 Decaying Accuracy 37

3.1.4 Presorting Realizations 38

3.2 Algorithm Efficiency 42

3.2.1 Dubins Car Examples 42

3.2.2 F-16 Example 48

3.3 Discussion 52

Chapter 4: Sparse Integration Schemes for Motion-Primitive Path Planning . . 54

4.1 Sparse Schemes via Mixed Integer-Linear Programming 55

4.2 Technical Considerations 59

4.2.1 Required Properties of Cubature Rules 60

4.2.2 Problem Modifications to Improve Numerical Performance 64

4.3 Results 70

4.3.1 Effectiveness of MILP 70

4.3.2 Path Planning Example 71

4.4 Discussion 76

Chapter 5: Monte Carlo Algorithms 77

5.1 Monte Carlo Tree Search 77

5.2 Vulcan 79

5.3 Comparison with AO* 82

5.3.1 Strengths of AO* 82

5.3.2 Strengths of Monte Carlo algorithms 85

5.3.3 Supersonic Glide Vehicle Scenario 87

5.4 Discussion 92

Chapter 6: Lazy Collision Checking 94

6.1 Generalized Lazy Hypergraph Search 94

6.1.1 Lazy R/AO* 98

6.1.2 Lazy Vulcan 100

6.1.3 Lazy MCTS 102

6.2 Discussion 105

Chapter 7: Conclusion 106

7.1 Contributions 106

7.2 Future Directions 109

Appendices 111

Appendix A: Invariance of Dubins Car Model to action of SE{2} 112

References 114

Vita 124

LIST OF TABLES

2.1 Accuracy of Koopman operator versus Monte Carlo method 21

3.1 Planning Statistics for R/AO* on Dubins Car and F16 scenarios 47

4.1 Comparison of rule size for van den Bos and MILP. 71

4.2 Number of Nodes in Dense and Optimal Sparse Cubature Rules and Solution Time. 72

4.3 AO* Planning Statistics for Environment Shown in Figure 4.2. 75

4.4 Planning Statistics using Sparse Integration scheme 75

5.1 Performance of Vulcan at varying risk tolerance when allowed the same planning time as AO*. 84

5.2 Vulcan with fixed planning time at varying risk tolerance 85

5.3 Performance of planners with various cost structures on the glide vehicle task 91

6.1 Effects of three events (SP=shortest path, HP=heuristic progress, CD=constant depth) on performance of R/AO* on a two obstacle search task. 99

6.2 Performance information for Lazy Vulcan 100

6.3 Performance information for Lazy MCTS 103

LIST OF FIGURES

2.1 Maneuver Library for Dubins Car-like system 20

2.2 Sample executions of the Expected State Planner on the Path thru a Gap task 22

2.3 Plans chosen by Expected State Planner on the Mobile Obstacles task at different risk tolerances 25

2.4 Plans chosen by Expected State Planner on the Scylla and Charybdis task at different risk tolerances 26

3.1 Visualization of AO* with the Induced Heuristic 34

3.2 Visualization of RAO* 35

3.3 Schematic view of cell-based state merging 37

3.4 Diagram of presorting for fast constraint enforcement 41

3.5 Nominal trajectories for a 51 maneuver library for the Dubins car-like vehicle 43

3.6 Environment of the first planning task for R/AO* 44

3.7 Environment of the second planning task for R/AO* 45

3.8 Realizations Collision-Checked by AO* Algorithm for First Planning Task. 45

3.9 Realizations Collision-Checked by RAO* Algorithm for First Planning Task. 46

3.10 Nominal trajectories for F-16 maneuver library. 50

3.11 Side view of the environment for the F-16, with nominal realization of AO*'s solution. 51

4.1 An And/Or graph with parameter sequences marked 61
4.2 Test environment for the Dubins Car, with goal position marked. 73
5.1 A field of Poisson distributed quadrilaterals 82
5.2 Obstacle field with three walls 84
5.3 Deep search in a grid environment 86
5.4 Deeper search in a grid environment 87
5.5 Maneuvers for Supersonic Glide Vehicle 89
5.6 A three maneuver task for the supersonic glide vehicle 90
6.1 The Generalized Lazy Search architecture 94
6.2 Task in which Lazy Vulcan fails to respect risk tolerance. 101
6.3 Task in which Lazy Vulcan takes a riskier path than regular Vulcan. 101
6.4 Task in which Lazy Vulcan incurs much greater risk than regular Vulcan. . . 102
6.5 Comparison of MCTS and Lazy MCTS searches on Poisson distributed fields 103
6.6 Change in cost by introducing laziness versus change in rate of collision. . . 104

LIST OF ACRONYMS

CC-POMDP Chance-Constrained Partially-Observable Markov Decision Process

CCMDP Chance-Constrained Markov Decision Process

DOF Degrees of Freedom

FP Frobenius-Perron

GLS Generalized Lazy Search

GPU Graphics Processing Unit

MCTS Monte Carlo Tree Search

MDP Markov Decision Process

MILP Mixed Integer-Linear Program

PAO* AO* with Presorting

PDE Partial Differential Equation

PDF Probability Density Function

POMDP Partially-Observable Markov Decision Process

PRAO* RAO* with Presorting

RAO* Risk-bounded AO*

UQ Uncertainty Quantification

SUMMARY

This work considers the application of motion primitives to path planning and obstacle avoidance problems in which the system is subject to significant parametric and/or initial condition uncertainty. In problems involving parametric uncertainty, optimal path planning is achieved by minimizing the expected value of a cost function subject to probabilistic (chance) constraints on vehicle-obstacle collisions. The Koopman operator provides an efficient means to compute expected values for systems under parametric uncertainty. In the context of motion planning, these include both the expected cost function and chance constraints. A maneuver-based planning method is developed that leverages the Koopman operator to minimize an expected cost while satisfying user-imposed risk tolerances. The method is illustrated in three separate examples using a Dubins car model subject to parametric uncertainty in its dynamics or environment. Prediction of constraint violation probability is compared with a Monte Carlo method to demonstrate the advantages of the Koopman-based calculation.

Motion primitive planning under parametric uncertainty may be modeled as a chance-constrained Markov Decision Process (CCMDP). One way to obtain single-query solutions to CCMDPs is by searching the And/Or hypergraph representing the state-action space of the system. The Risk-bounded AO* (RAO*) algorithm has been proposed as a solution method for this problem, but it scales poorly to MDPs resulting from a motion primitive discretization because it has no mechanism to prioritize expansion of AND nodes. An induced heuristic for state-action pairs is described that can be rapidly computed by leveraging the properties of motion primitives; its value can be used to prioritize AND nodes for more efficient search. The resulting algorithm is referred to as AO* with induced heuristic. The hypergraph search is further accelerated by leveraging shared symmetry in constraints and dynamics to move almost all computation necessary to enforce convex polytope constraints offline. The performance improvements are demonstrated with path planning prob-

lems involving a Dubins Car and a nonlinear aircraft model.

The key bottleneck for the path planning under uncertainty algorithms described herein are the expected value computations. Computing an expected value requires integrating over the uncertainty domain, which may be high dimensional. One way to accelerate such calculations is through the use of a "sparse" numerical integration scheme. A method is described for obtaining maximally sparse numerical integration schemes for use with hypergraph search algorithms for path planning problems under parametric uncertainty. The approach formulates a mixed-integer linear program that is tailored to the specific structure of the hypergraph on which chance-constrained motion primitive planning problems are solved. The optimization is solved offline, yielding a sparse integration scheme that can then be used for a variety of planning tasks. Results demonstrate that the sparse schemes maintain the estimation accuracy of the original formulation while requiring dramatically less computation time.

AO* with the induced heuristic is compared and contrasted with sampling based Monte Carlo Tree Search and a variant for chance constrained planning called Vulcan. It is shown that AO* produces distinctly better solutions than the sampling based algorithms in many situations, but that for some problems in which it is impractical the sampling based algorithms may still be able to return a solution. Finally, the potential for the Generalized Lazy Search approach to accelerate planning in conjunction with RAO*, AO*, MCTS, and Vulcan is explored. Laziness proves strongly beneficial for RAO*, but in the tested scenario remains inferior to AO* with the induced heuristic. Moreover, AO* with the induced heuristic is found to not benefit from lazy collision checking; this is traced to the fact that the induced heuristic is itself a lazy collision checking approach that could be described as a novel specialization of the Generalized Lazy Search approach. MCTS and Vulcan are found to experience small to moderate convergence improvements from the Generalized Lazy Search approach in some settings, but in other settings laziness can lead to extreme increases in the frequency of constraint violations.

CHAPTER 1

INTRODUCTION

Path planning is one of the most fundamental problems in robotics. It is also a very broad problem, encompassing in principle everything from playing chess to flying through a forest at speed. Kinodynamic motion planning is of particular practical importance, as it is the specialization concerned with finding the control inputs necessary to steer to a goal while respecting dynamics and satisfying imposed constraints. This is the level of detail necessary to steer a robot (finding control inputs) through a crowded room without hitting anyone (satisfying constraints) or assuming the robot can teleport (respecting dynamics). The motion primitive framework, described in [1], is a popular approach to this problem because it provides a way to guarantee the resulting trajectory respects the system dynamics without needing to do online simulation [2, 3, 4, 5, 6, 7, 8]. It does this by leveraging invariances in the dynamics of the system: If the trajectory followed by an automobile as it turns left when it started facing north is rotated 90 degrees clockwise, this new trajectory will exactly overlap the trajectory for a left turn that started out facing east . In other words, the dynamics of the automobile are invariant to planar rotations (they are also invariant to planar translations). Many dynamical systems of interest exhibit similar properties, which can be codified mathematically as identifying a particular Lie group whose group action commutes with the state flow of the differential equation describing the dynamics. This equivalence of trajectories creates an opportunity for reuse of simulations. Offline, one can compute a number of trajectories (called "maneuvers") corresponding to particular control laws. Planning is then reduced to selecting a concatenation of trajectories that connect the start and goal.

Motion primitive planning becomes significantly more complicated when considering systems with parametric uncertainty. A key assumption in the formulation of motion prim-

itive planning is that the state evolution from the beginning to the end of each maneuver is, up to an offset in the cyclic coordinates, the same every time the maneuver is executed. Under parametric uncertainty in the system dynamics, this may no longer be true; the state after executing a maneuver will generally depend on the unknown parameters. Despite this difficulty, several authors have proposed extensions to motion primitive planning to address uncertainty. Schouwenaars *et al.* [9] proposed a robust motion primitive algorithm that used Monte Carlo simulation to estimate the maximum deviation in state values that could result from each maneuver. These deviations are then factored into a modified cost-function to provide a "crude approximation of the expected value." A more recent robust generalization by Majumdar and Tedrake [10] replaces the notion of a primitive with that of a funnel. A funnel is a set combined with a controller where, if the vehicle begins inside the entry region, it is guaranteed to remain in the funnel while the controller is executing. The authors present a way to compute funnels that will exhibit the Lie group invariance property by leveraging Sum-of-Squares optimization techniques.

While robust approaches are useful in many cases, they offer no mechanism to trade probability of constraint violation for better system performance. In many cases, it is possible to obtain a better path (in terms of overall cost) if constraint violation is allowed with non-zero probability. As an example, Thakur *et al.* [11] use motion primitives as the actions in a Markov Decision Process, with massively parallel Graphics Processing Unit (GPU) computation used to estimate transition probabilities. This approach computes a probabilistically optimal path under uncertainty, but constraint violations are penalized with additional cost. Despite this limited work, motion primitive planning under parametric uncertainty remains a highly under-explored area. Interestingly, no prior work (to the authors' knowledge) has investigated the possibility of chance-constrained planning with motion primitives. In chance-constrained planning, the probability of constraint violation is computed and constrained to be less than some risk tolerance. The advantages of the chance-constrained approach over robust techniques or cost-penalized approaches are

well-known [12, 13, 14, 15, 16]; including chance constraints permits preferences that cannot be produced by any purely cost-penalized construction. This is a double-edged sword as shown by the extensive exchange of letters on the utility and correctness of the chance constrained formulation in the 1970s and 1980s [17, 18, 19, 20, 21, 22, 23, 24, 25]; one must remain aware that the introduction of a chance constraint may well eliminate the cost optimal solution *even in a setting* in which failure (or, more generally, constraint violation whether terminal or not) is penalized "correctly." The chance-constrained formulation is nevertheless strictly more expressive than the pure-penalty formulation, so provided the computational penalties are not too onerous (indeed, some results in chapter 5 suggest chance constraints can provide computational benefits) algorithms capable of handling this formulation have great potential benefits.

Path planning under parametric uncertainty (equivalently, initial condition uncertainty) requires some means of Uncertainty Quantification (UQ) to evolve the state Probability Density Function (PDF) along the path, so that the expected value of the cost and constraint violation probabilities may be calculated. A standard technique is Monte Carlo sampling, used in [11, 16, 26]. It has the distinct advantage of applying to nonlinear systems, arbitrary probability distributions, and unlimited simulation horizons, giving it wider applicability than analytical or parametric techniques like those used in [13, 27, 28, 29]. From an implementation perspective, Monte Carlo methods also benefit greatly from massively parallel computation on GPUs [11, 30]. However, Monte Carlo scales poorly to higher dimensions and the underlying PDF must be estimated from the evolved samples. As discussed in [31], a variety of techniques exist to alleviate this in special cases.

The Frobenius-Perron (FP) operator and its adjoint the Koopman operator form the basis of alternative "explicit Uncertainty Quantification (UQ)" techniques for parametric uncertainty that have been shown in [31, 32, 33, 34, 35] to scale better than Monte Carlo while preserving much of the generality and parallelizability of the technique. In particular, Meyers *et al.* [34] showed that the Koopman operator can be used to compute constraint

violation probabilities and expected costs more accurately than Monte Carlo at a given number of samples, while providing tractable integration domains. Leonard *et al.* [35] used GPUs and Lobachevsky spline integration to apply the Koopman approach in a probabilistic airdrop optimization problem involving five dimensions of parameter uncertainty.

The Koopman operator provides an infinite dimensional linear encoding of the state evolution of nonlinear dynamical systems. Existing work has shown that truncations of basis function expansions of the operator provide an effective means of system linearization that is amenable to discovery from system state data [36, 37, 38, 39, 40, 41]. This provides an approximate continuous representation of the evolution of an observable of the system state. When used for explicit uncertainty quantification, the Koopman operator is instead obtained "exactly" (up to integration error) at a discrete set of selected points in the state space via the method of characteristics [34]. Basis function expansion and data-driven learning are avoided by relying on an existing (potentially black-box) model and by sacrificing information about the evolution of the observable of the state of the system at unaddressed points.

A framework for motion primitive planning under parametric uncertainty using the Koopman operator is presented in chapter 2. The uncertainty space is discretized offline and the state evolution along each primitive is computed at each point and stored. Online, the probability of constraint violation for each stored trajectory is obtained and pulled back to the parameter uncertainty domain (this is the action of the Koopman operator). A simple expected value calculation provides the total constraint violation probability. Three path planning examples are shown for a Dubins car model under parametric uncertainty, involving various types of chance constraints. The accuracy of the constraint violation probability predictions is compared with Monte Carlo methods to demonstrate the computational advantages of the proposed approach.

Combining the Koopman operator with motion primitives permits construction of a Chance-Constrained Markov Decision Process (CCMDP) with a discrete action set. The

state of this CCMDP can be viewed as evolving on a graph, with a path obtained via graph search. Each edge leaving a state in this graph (that is, each maneuver available at that state) is associated with multiple exit states due to the parametric uncertainty affecting the primitive evolution. Any of those states can result if the maneuver is executed, so a complete plan must include paths to the goal from each of them. This type of search graph is recognized as an AND/OR hypergraph, a graph-like structure where some edges connect one vertex to many vertices (instead of all edges being one to one). AND/OR hypergraphs have previously been used to study game tree evaluation [42, 43, 44], planning problems with non-deterministic actions [45], partially-observable planning [46], and for representing deformable objects in image segmentation learning [47].

A classic informed search algorithm for AND/OR hypergraphs is called AO* [48]. AO* uses a heuristic to prioritize its search, which results in a solution tree (instead of the path that results from search on a regular graph). Bagchi and Mahanti prove in [49] that AO* returns the lowest cost tree if the heuristic underestimates the true cost to go (i.e., the heuristic is admissible). As an alternative, Mahanti and Bagchi present in [50] the CF and CS algorithms which together produce optimal solutions for some inadmissible heuristics. Nau *et al.* demonstrate that A* and AO* can be unified as special cases of a generalized Branch and Bound formulation [51]. Chakrabarti *et al.* prove that AO* will yield the optimal solution tree under a variety of additive rules for computing the tree cost even if the heuristic "occasionally" overestimates, and present bounds on solution suboptimality when it does occur [52]. Finally, Chakrabarti *et al.* prove that if one heuristic dominates another, the worst-case set of nodes expanded by AO* using the dominating heuristic is not larger than the worst-case set using the inferior heuristic. This is a weaker version of the A* property that a dominating heuristic does not increase the actual set of expanded nodes.

One major limitation of AO* is that the solution graph must be acyclic, as it conducts a back-propagation step that will not terminate if there are cycles [48]. The actual hypergraph can in principle include cycles so long as the search never encounters them (in particular,

the solution must be acyclic). Jimenez and Torras present an algorithm based on Mahanti and Bagchi's CF algorithm to handle graphs where the cycles are encountered during search [53]. To handle the case where the solution itself is cyclic, Hansen and Zilberstein replace the simple back-propagation step in AO* with either value or policy iteration to construct the LAO* algorithm [54]. They demonstrate that many of the results of [52] hold for LAO* as well as AO*, including the solution optimality result and the effects of heuristic accuracy on node expansion.

Santana *et al.* introduced Risk-bounded AO* (RAO*) as a generalization of AO* for Chance-Constrained Partially-Observable Markov Decision Process (CC-POMDP). In CC-POMDPs, the belief state evolves on an AND/OR hypergraph if the action set is discrete [55]. RAO* modifies AO* by pruning possible states that exhibit a risk of constraint violation in excess of the risk bound. Additional pruning is achieved in the partially observable context by utilizing an admissible heuristic estimate of the risk in addition to the cost heuristic. In [56] and [57] the RAO* algorithm was modified for use in a receding horizon framework by limiting the maximum search depth. While originally proposed as a solution technique for a Partially-Observable Markov Decision Process (POMDP), RAO* can of course be applied to a fully observable Markov Decision Process (MDP).

The framework for chance-constrained planning from chapter 2 can be cast as a discrete action (fully-observable) Markov Decision Process under uncertainty. And/Or graph search techniques such as RAO* may be used to solve for single-query solutions to such problems. However, the enormous branching factor resulting from the discretization of the uncertainty space means that straightforward application of RAO* is impractical for problems of realistic complexity. This weakness is addressed in chapter 3 and a solution proposed. RAO*'s difficulty lies in the fact that AND nodes have no physical existence in the CCMDP formulation; they represent the different actions available at a particular state in the domain (itself represented by a particular OR node). As a result, defining an (admissible) heuristic to meaningfully prioritize between AND nodes is non-trivial. RAO*

sidesteps this issue by following the expansion of an OR node with the expansion of all its AND node children. This is equivalent to implicitly assigning each AND node (each action) the same heuristic value as the parent OR node (the state the action is executed from). Generating the children of every action at a node (and thus, checking the path to each child for constraint violations) will, in general, involve unnecessary work. For example, if the goal is ahead of the current location, examining paths involved in turning around is likely wasted effort. A more efficient search could be achieved by prioritizing the exploration of promising actions. When all actions are motion primitives, the exit state of an action can be computed in a single operation (skipping over the states that occur during execution). Thus, one can compute the heuristic function on the exit states of a particular action in far less time than would be needed to do full collision checking on that action. An induced AND node heuristic can then be defined as the expected value of the cost to come to the exit states plus the original heuristic at those states. The induced heuristic allows prioritizing AND node expansions at the cost of some additional overhead (as this induced heuristic is more expensive to compute than the original heuristic). Applying AO* using this heuristic returns precisely the same solution as RAO* while (in practice) performing fewer checks for constraint violations. In problems where constraint checking is costly the additional overhead is greatly outweighed by reducing the number of constraint checks.

In addition, three modifications applicable to both variants are proposed to further reduce computational effort. The first (nearly trivial) modification, state merging, leverages RAO*'s ability to handle solution graphs by combining similar states via state space discretization. This merging may produce a dramatic reduction in the number of nodes expanded, but requires care in the presence of loops. The second modification – decaying accuracy – applies the logic behind Variable Level-of-Detail planning [58] to sampling of the uncertainty domain. Fewer samples are used when considering far future actions, resulting in reduced accuracy in cost and risk evaluation. When periodic replanning will be performed, a new plan (that uses higher resolution) will be made before poorly-resolved

future actions are taken. The third modification, called presorting, leverages simultaneous symmetry in both vehicle dynamics and constraints to presort realizations (the trajectory of a maneuver for a particular value of the uncertain parameter) by how much they move towards planar constraints. With this presorted database of maneuvers, the number of full collision checks that must be performed online is reduced by checking single realizations in a binary search pattern; once the realization that transitions from not colliding to colliding is found, no further checks need be performed on the remaining realizations. This effectively moves much of the computation needed to check violations of convex polytopic constraints offline to a preprocessing step. Numerical results are presented showcasing the effectiveness of the induced heuristic and the algorithm modifications on a pair of Dubins car examples and a scenario involving a 6DOF nonlinear aircraft model.

The key calculation underpinning the uncertain path planning problem, with or without the presence of chance constraints, is the expected value. Preferences between policies are expressed in terms of the expected cost or reward, while chance constraints (if present) involve the expected value of indicator functions for the relevant constraints. Mathematically, expected values are integrals over the uncertainty domain; this is why the dimension of the uncertainty domain contributes to the "curse of dimensionality" in the uncertain setting. In deterministic path planning problems the curse of dimensionality appears primarily due to the dimension of the state space, which can be mitigated by careful discretization [59, 60] or by seeking only single query solutions rather than a full policy [61, 62, 63]. Uncertainty compounds the curse by introducing its counterpart from numerical integration: expectations are integrals over the uncertainty domain, so the dimension of the uncertainty domain affects the difficulty of evaluating the expected costs and the risk. This can be mitigated by using analytical bounds [27, 64] or parametric representations of the uncertainty [63, 65]. Approaches for addressing the curse of dimensionality in numerical integration are also applicable (see Bungartz and Griebel for a brief exposition [66]). One approach, widely used in the path planning community, is to use a Monte Carlo integration scheme [11, 16, 67,

26, 27, 68, 69, 70]. Monte Carlo methods for numerical integration have numerous advantages including broad applicability, ease of implementation, and an $O(\sqrt{n})$ convergence rate that is independent of the dimension of the problem. Variants of the Monte Carlo Tree Search (MCTS) algorithm leverage this property. The effectiveness of classic MCTS, as well as a variant capable of enforcing chance constraints called Vulcan [69], is studied for the motion primitive path planning problem in chapter 5.

Another option for mitigating the dimensional effects in the expected value are numerical integration schemes based on "sparse grids". Such schemes use "carefully" placed sample points so that the dependence on dimension of the total number needed is reduced. An early approach was devised by Smolyak, who developed a recursive procedure for building a sparse high dimensional rule from univariate rules [71]. This approach has been used to address problems from economics [72, 73] to uncertainty quantification [74, 75], PDEs (Partial Differential Equations) [76], and stochastic differential equations [77]; modifications for purposes such as adaptive integration have also been developed [78]. However, the Smolyak scheme produces non-positive weights in some cases (whereas with positive weights convergence is guaranteed for continuous integrands [79]), so other approaches have been proposed. Van den Bos et al. [75] develop a method to construct a sparse integration scheme from a dense scheme while preserving positivity and symmetry, though they cannot guarantee that the resulting rule contains the fewest possible samples. This suboptimality could be avoided by using optimization in the rule construction, as was done in Xiao and Gimbutas [80], Ryu and Boyd [81], and Keshavarzzadeh et al. [82]. In chapter 4, the problem of designing an integration scheme suitable for use in an AO*-based path planner is addressed from the perspective taken in [75]: given a dense integration scheme that exactly integrates a set of polynomials, remove nodes from the scheme and recompute the weights such that the polynomials are still integrated exactly, the weights remain positive, and symmetry is preserved. The goal of the resulting rule is to significantly accelerate the online computation of expected values in the AO*-based algorithm, as well as analo-

gous algorithms that plan paths on hypergraphs. This is achieved by casting the problem of removing nodes from the dense scheme as a Mixed Integer-Linear Program (MILP). Numerical results show that the MILP approach produces smaller rules than the baseline van den Bos approach, and that rules for hypergraph search obtained for systems with sufficient symmetry provide dramatic benefits to the computational cost of path planning.

Monte Carlo approaches are also quite suitable to path planning under uncertainty. MDPs under uncertainty are structurally similar to adversarial games. In the game perspective, instead of an environment that chooses randomly to make the outcome of actions uncertain, there is an adversary which chooses according to some policy. In the zero-sum case, the adversary acts to minimize your reward and so will always choose the worst possible outcome (for you). This is equivalent to constructing a robust plan from the MDP point of view. Monte Carlo Tree Search (MCTS) is an algorithm introduced by Kocsis and Szepesvari [83] that has had great success in the field of games, achieving state of the art performance in Go [68, 84], Solitaire [85], and chess [84] among others tasks. It is thus natural to apply it to MDPs under uncertainty. Particularly relevant is [69], which proposes a variant of MCTS called Vulcan, which can enforce chance constraints. MCTS for path planning under uncertainty and Vulcan are described in chapter 5, which then conducts a comparison of these algorithms with AO* on tasks for the Dubins car model and a simplified supersonic glide vehicle. It is shown that the solutions returned by AO*, which is an optimal and complete algorithm, are higher quality than those obtained by the sampling algorithms. In particular, the performance of Vulcan is found to degrade for tight chance constraints. The strength of the sampling algorithms is in their anytime nature; they can return good but not perfect quality solutions in relatively short amounts of time, even on some tasks which prove intractable for AO*.

Of the two types of expected value path planning under uncertainty contends with, it is the constraint indicator functions that are the most problematic. Cost functions are generally cheap to evaluate and so make up a smaller fraction of the computational burden.

In special cases constraint evaluation can be accelerated by techniques like the presorting described in chapter 3, but fully general constraints are a serious burden. It is thus not ideal that AO*, like A*, is an algorithm designed to minimize the number of vertex expansions during a search for the shortest path. In practice it is evaluating an edge (i.e. checking a possible realization of an action for collision) that is most expensive. In the graph search setting this can be addressed via lazy collision checking approaches such as [86, 87, 88, 89, 90, 91], which avoid checking edges for collision until the algorithm is "confident" the edge is in the true best path (what exactly "confident" means depends on the algorithm). Work by Mandalika et. al [92] and Lim et al [93] has shown the effectiveness of a Generalized Lazy Search (GLS) architecture that toggles between exploration and edge evaluation based on a user-defined event function. In chapter 6, GLS is extended to hypergraphs (i.e. path planning under uncertainty). Additionally, it is described how Vulcan can be modified in order to work with GLS. Numerical experiments on the Dubins Car task show strong benefits of GLS for RAO* using three different event functions, but the performance is inferior to AO* with the induced heuristic on the same task. Furthermore, AO* is shown to run slower if GLS is applied. The ineffectiveness of GLS with respect to AO* is shown to be due to the fact that the induced heuristic is itself a lazy algorithm that could be expressed as a particular event and selection rule for GLS. Application of GLS to MCTS and Vulcan proves similarly disappointing. For extended planning times numerical results show small improvements in solution quality, but the primary use case of these algorithms is for short planning times and GLS proves unreliable in those cases. Scenarios are shown in which short-planning time lazy MCTS and Vulcan outperform their non-lazy counterparts, but other scenarios are identified in which the laziness causes egregious planning errors that manifest as dramatic increases in collision rate. This unreliability makes it impossible to recommend the use of GLS with short planning time MCTS or Vulcan.

Finally, chapter 7 summarizes the contributions of this work and describes a few possible future directions of research.

CHAPTER 2
MOTION PRIMITIVES VIA THE KOOPMAN OPERATOR

In this chapter, the motion primitive formulation is extended to parametrically uncertain systems. The Koopman operator is used to convert expected values to integrals over the uncertainty domain, which can be readily approximated using numerical integration schemes. The formal chance constrained path planning problem is described and its connection to And/Or hypergraphs elucidated. Focusing on the Koopman vs Monte Carlo comparison, a simple path planning algorithm and a double integrator "Dubins car"-like vehicle are introduced. Two scenarios show the strength of the Koopman operator based approach for computing expected values compared to Monte Carlo sampling. A third scenario is presented that demonstrates the utility of the Koopman operator approach for systems subject to environmental, rather than dynamic, uncertainty.

2.1 Mathematical Background

Let $\mathbb{D}$ be a controlled dynamical system. Let the state be $x \in \mathbb{X}$, and suppose that $\mathbb{D}$ is subject to (vector valued) parametric uncertainty taking values in $s \in S \subseteq \mathbb{R}^m$. Let the control be $u \in \mathbb{U}$ and let μ be a (possibly closed loop) control law. Define $\rho_\mu(t, t_0, x_0, s_i) \in \mathbb{X}$ as the state at time t on the trajectory originating at time t_0 and state x_0 generated by the control law μ when the uncertain parameter is s_i. The Lie Group $\mathbb{G}$ is a symmetry group of $\mathbb{D}$ if $\mathbb{D}$ is invariant to the action of $\mathbb{G}$ on $\mathbb{X}$: $\forall g \in \mathbb{G}$, $\rho_\mu(t, t_0, g \circ x_0, s_i) = g \circ \rho_\mu(t, t_0, x_0, s_i)$. This is equivalent to requiring that $\mathbb{G}$ be a symmetry group in the sense of [1] for every member of the family of deterministic dynamical systems formed by $\mathbb{D}$ at different values of the uncertain parameter. Note that this is only possible if the control law itself respects the symmetry of the system ($\mu(g \circ x)$ not necessarily equal to $\mu(x)$, but $g \circ x + \delta t f(g \circ x, \mu(g \circ x), s)$ must equal $g \circ (x + \delta t f(x, \mu(x), s)$ for f the differential

equation for $\mathbb{D}$).

$\mathbb{G}$ divides $\mathbb{X}$ into equivalence classes with the relation $x_1 \cong x_2 \iff \exists g \in \mathbb{G}$ s.t. $x_1 = g \circ x_2$. These equivalence classes are related to the concept of a trim trajectory from [1]. For a deterministic system, a trim trajectory α is one where the control input is constant and the state evolution is given by $x(t) = \exp(\xi_\alpha(t - t_0)) \circ x_0$. ξ_α is an element of the Lie algebra $\mathfrak{g}$ of $\mathbb{G}$, and $\xi_\alpha \Delta t \in \mathbb{G}$. Since every state in a trim is offset by an element of $\mathbb{G}$, any trim trajectory will remain within a single equivalence class. In an abuse of terminology, these equivalence classes will also be called trims. The invariance property of the system means that every state in a trim is "the same." An open loop control applied at x_1 in the trim will produce the same trajectory as produced at x_2, offset by the action of $g_{12} \in \mathbb{G} : x_1 = g_{12} \circ x_2$.

The generalized coordinates of $\mathbb{X}$ can be separated into cyclic and non-cyclic coordinates. Cyclic coordinates are those on which the system Lagrangian does not depend; their value thus has no effect on dynamics. Non-cyclic coordinates form the remaining set. A symmetry group of $\mathbb{D}$ must act only on the cyclic coordinates. Thus, states in the same trim differ only in their cyclic coordinates. Because symmetry must hold for every parameter value, the introduction of uncertainty to a system may result in a $\mathbb{G}$ acting on fewer coordinates than if the system was deterministic. In this case, additional trims would be required to capture the full behavior of the system. For example, if wind uncertainty with non-zero mean is present, heading changes are not part of the symmetry and so trims would be needed at each heading of interest.

2.1.1 Uncertain Maneuvers

An uncertain maneuver π is defined to be a particular (possibly closed-loop) control law, a termination condition, and a "predecessor" trim. This control law can be applied to states in the predecessor trim to produce state trajectories that will, in general, vary with the uncertain parameter. A "realization" refers to an instance of the state trajectory for a particular

parameter value. The cyclic coordinate part of these state trajectories can be computed for *any* initial condition in the predecessor trim by recording the history of group displacements for a single initial condition in the trim (different parameter values lead to different histories):

$$g_\pi(t - t_0, s) : \rho_\pi(t, t_0, x_0, s) = g_\pi(t - t_0, s) \circ x_0 + n(t, s) \tag{2.1}$$

where $n(t, s)$ is the offset in the non-cyclic coordinates. Thus, g_π for a specific parameter value can be extracted from forward simulation of $\mathbb{D}$ under π's control law. While the cyclic coordinate history for a realization will depend on the initial state x_0, the evolution of the non-cyclic coordinates must be independent of x_0. All x_0's in the trim have the same non-cyclic values, and by definition cyclic values cannot affect the dynamics. As a result, the same single simulation used to obtain the group displacement history for a particular parameter value also provides the non-cyclic coordinate evolution $n(t, s)$ for all initial conditions in the trim. In the absence of this symmetry, a recorded trajectory would be valid only from that single initial condition and so be of little use for path planning.

2.1.2 Expected Values and the Koopman Operator

In path planning, important quantities such as the expected cost and the probability of constraint violation can be viewed as the expected value of a function of the system state. However, constraint indicator functions and costs with an integrated term are in fact functions of the entire state trajectory rather than a particular time step. This is resolved by augmenting the state vector with the integrated part, as shown below for a cost term. Constraint indicator functions can be handled similarly.

Let $H(x_0, t_0, s)$ be a state trajectory, with

$$J(H(x_0, t_0, s)) = \int_0^{\tau_f} d(t, H(x_0, t_0, s)(t))dt \tag{2.2}$$

the integrated cost. τ_f is the duration of the trajectory. The state is augmented with the

accumulated value of the integrand (symbol: D) and its dynamics defined:

$$\dot{D} = d(t, x) \tag{2.3}$$

Then, define a function $z : \mathbb{A} \to \mathbb{R}$ (where $\mathbb{A} = \mathbb{X} \times \mathbb{R}$ is the set of augmented states) that extracts D. The expected cost is now the expected value of z, which is a function only of the (augmented) state at a particular instant.

The expected value of a function of the state of a dynamical system subject to parameter (equivalently, initial condition) uncertainty can be computed using the Koopman operator. Continuing with generic z a function of the (possibly augmented) state, let $P : \mathbb{S} \to \mathbb{R}^+$ be the probability distribution over the uncertain parameter s.

$$E[z(\rho_\mu(t, t_0, x_0, s))] = \int_{\mathbb{S}} P(s)\mathbb{K}_{\mu,t} z(x_f) ds \tag{2.4}$$

where $E[\cdot]$ denotes the expected value and $\mathbb{K}_{\mu,t}$ is the Koopman operator for time evolution t of the uncertain dynamical system obtained by applying the control law μ to $\mathbb{D}$ for a duration t. The Koopman operator acts to map a function of the system state at some future time to a function of the initial conditions and uncertain parameters, which allows computing expected values using the known probability distribution over those same initial conditions and uncertain parameters *without* needing to actually evolve the probability distribution through the dynamics. For a careful derivation of the Koopman operator pull-back and analysis of its computational advantages over alternative techniques, see [34].

For the purposes of this work, it is enough to realize that the Koopman operator converts an expectation over a function of the terminal state to an integral over the uncertainty domain $\mathbb{S}$. The integral in (Equation 2.4) is approximated by gridding $\mathbb{S}$ with n sample points s_j and computing $z(x_f)$ for the trajectories corresponding to those samples:

$$E[z(\rho_\mu(t, t_0, x_0, s))] \approx \sum_{j=1}^{n} P(s_j) z(\rho_\mu(t, t_0, x_0, s_j) \Delta s \tag{2.5}$$

where Δs is the integration voxel size. When μ is the control law of a maneuver, ρ can be obtained for any initial condition using records of offline simulations as discussed in subsection 2.1.1. As the locations of s_j may be chosen freely (though they must be chosen when the offline simulations are done), more sophisticated numerical integration schemes may be used if appropriate. This ability may be particularly useful in higher-dimensional problems; for further discussion of possible integration methodologies, see [34], [35], chapter 4, and chapter 5. In particular, if Monte Carlo integration is used this is equivalent to evaluating the expectation via the Monte Carlo method with uniform proposal distribution. Note the importance of offline access to the system dynamics (for computing motion primitives, either via simulation or via hardware observations), offline access to the support of the probability distribution of the parameters (for selecting parameter samples and recording their realizations), and *online* access to the actual probability distribution of the parameters (for expectation evaluations at planning time).

2.2 The Chance-Constrained Uncertain Planning Problem

The single-query chance constrained planning problem under parameter uncertainty seeks a control law that will drive the vehicle from an initial condition x_0 to a goal state g while violating independent constraints C_i with probability less than a fixed risk tolerance r and minimizing the expectation of a cost function J. This optimization problem may be written as,

$$\begin{gathered} \underset{\mu \in \mathbb{M}}{\operatorname{argmin}}\, E[J(\rho_\mu(t, t_0, x_0, s))] \text{ s.t.} \\ 1 - \prod_i \left(1 - E\left[C_i\left(\rho_\mu(t, t_0, x_0, s)\right)\right]\right) \leq r \end{gathered} \tag{2.6}$$

where $\mathbb{M}$ is the set of all control laws for $\mathbb{D}$. Generalizations of this problem are possible, such as permitting r to vary with time, imposing different risk tolerances for different constraints rather than a single overall tolerance, or allowing tolerance to depend on the cost as in [69].

Given a library of maneuvers, a piecewise continuous control law that drives the system to the goal can be constructed by assigning maneuvers to the initial condition, to every state reached by executing that maneuver, and to every state reached by those maneuvers, etc., until all sequences of states eventually reach a terminal state. This assignment yields a piecewise continuous control law made up of the control laws corresponding to the assigned maneuvers. By ensuring the maneuvers chosen are compatible (in the sense that the terminal state of the preceding maneuver is on the same trim as the initial condition of the following), the trajectory is guaranteed dynamically feasible! The assignment is also a partial policy for the CCMDP whose actions are the maneuvers in the library. If every maneuver leads to a finite set of states, the assignment with lowest expected cost can be obtained via search on an AND/OR graph. Here, an OR node represents a system state while AND nodes represent particular actions (in this case, maneuvers) that are available at the state represented by their parent OR node. The use of a finite set of parameter samples in obtaining the Koopman operator, as discussed in subsection 2.1.1, inherently results in a finite set of exit states for every maneuver. Thus, choosing an edge from an OR node is equivalent to assigning a maneuver, while the multiple edges leaving the AND node represent realizations of the maneuver for the sampled parameter values (and thus lead to OR nodes representing different states the system could be in when the maneuver finishes). A "path" on an AND/OR graph is actually itself a graph, consisting of a selection of an AND node at the root OR node, followed by every OR child of the selected AND node, repeated on down until every leaf node is a terminal state. The resulting piecewise continuous control law is a candidate solution to the optimization in (Equation 2.6). It will not in general be the *optimal* solution, but it will be the best solution out of all control laws constructed

from the maneuvers.

The cost-optimal path on an AND/OR graph can be obtained via the AO* algorithm [48], which can in many ways be thought of as an extension of the classic A* algorithm to AND/OR graphs. Like A*, it maintains a priority queue of vertices based on an admissible estimate of the cost and searches in a best-vertex-first fashion in an effort to minimize the number of expanded vertices. As derived by Chakrabarti et al, the theoretical efficiency guarantees of AO* are weaker than those for A*, but provided the heuristic is admissible AO* is still complete, sound, and optimal [52]. Pseudocode for AO* when specialized to path planning (but without chance constraints) is presented in Procedure 1. Note that AO* requires admissible estimates of the cost to go both from OR nodes (regular graph vertices, representing points in the state space in our context) and AND nodes (representing particular maneuvers executed at particular states). The requirement that the heuristic be defined for actions is non-trivial, as will be discussed in greater detail in chapter 3.

2.3 Efficiency of Koopman versus Monte Carlo Methods

To study the effectiveness of the Koopman operator approach to uncertainty quantification for motion primitives, a maneuver library was constructed for a Dubins car-like vehicle subject to different forms of model or environmental parametric uncertainty. In the following examples, the cost function J being minimized is the distance to the goal at the end of a maneuver. The vehicle travels at constant speed v in the x, y plane, with state $[x, y, \theta, \omega]^T$ and motion governed by the following differential equation:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} v\cos\theta + w_x \\ v\sin\theta + w_y \\ \omega \\ u \end{bmatrix} \tag{2.7}$$

Here, u is the control input, θ is the current heading, and ω is the angular velocity. The parameters w_x, w_y represent the effects of wind. They are i.i.d. Gaussian random variables with mean 0 and standard deviation 0.2 m/s. This dynamical system is invariant to the action of $SE(2)$, the planar rotation and translation group, which is shown in [94] to be a Lie group. A proof of this invariance under wind uncertainty is presented in Appendix A. A nominal velocity of $v = 10$ m/s is selected, and the control is bounded as $u \in [-1000 \text{ rad}/s^2, 1000 \text{ rad}/s^2]$ as in [10]. With winds set to 0, this system is identical to the nominal system from that work, but different forms of uncertainty will be imposed here. The states x, y, θ are the cyclic coordinates and ω is the only non-cyclic coordinate.

Maneuvers are constructed using direct collocation [95] to generate nominal trajectories satisfying the system dynamics while satisfying control saturation limits. These trajectories are shown in Figure 2.1 (top, in green). Maneuvers move the vehicle forward 2.25 meters and left or right by up to 1 meter. Their predecessor trim contains all states with $\omega = 0$. Maneuvers will be referred to by the distance they move the vehicle left; the top maneuver in Figure 2.1 is +1m. Figure 2.1 does not show the negative nominal maneuvers, nor the short "recovery" maneuvers used to regulate $\omega \to 0$ if tracking errors cause rotational velocity to accumulate. In the bottom of Figure 2.1, 121 realizations of the -1m maneuver are shown. The nominal trajectories are tracked by a time-varying LQR controller with identity cost matrices.

2.3.1 Expected State Planner

This portion of the work is focused on the effectiveness of the Koopman operator approach for handling uncertainty, so a simple planning algorithm is used. A recursive depth-limited search is proposed to select the next maneuver according to (Equation 2.6), while accounting for the possibility of a poorly chosen maneuver placing the vehicle in an inescapable collision. The approach is outlined in Procedure 2 and Procedure 3. It declines to search the And/Or graph that represents the true evolution of the states of the system; instead it

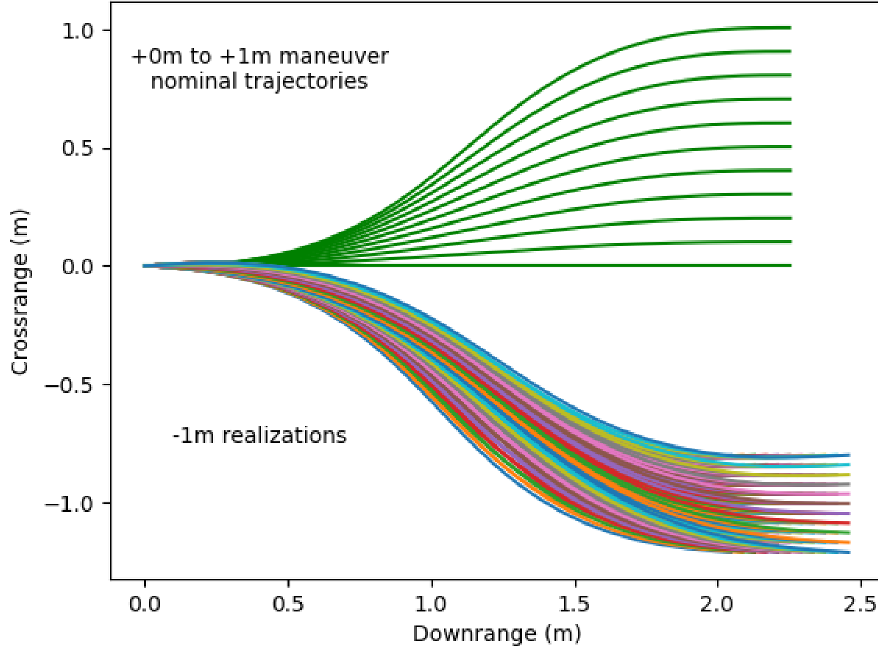

Figure 2.1: Top: Nominal trajectories for 11 of the 21 maneuvers (negative maneuvers not shown). Bottom: 121 realizations of the -1m maneuver under wind uncertainty, presented from the initial condition $x_0 = [0.0, 0.0, 0.0, 0.0]^T$.

collapses the multiple exit states of each maneuver into the single expected exit state, which evolves on a regular graph rather than an And/Or graph.

In Procedure 2, past_cp is the collision probability accumulated along the current sequence of primitives; thus, the initial call to *choose* sets this to 0. x_{obs} is the state from which the environment is observed while t_{obs} is the time at which that observation takes place; in the initial call these are equal to the current state x_0 and time t_0.

First, the set of maneuvers available in the current state is obtained. Procedure 3 is called to compute the collision probability for each maneuver. The probability of a collision during the current maneuver is combined with the collision probability accumulated over past maneuvers (past_cp) assuming that collision in any maneuver is independent of collision in any other. Procedure 3 computes the expected state and time following the execution of the maneuver and, if the new state is a goal state or outside the planning horizon, the accumulated collision probability is returned to *choose*. Otherwise, Procedure 2

Table 2.1: Absolute error (AE) in collision probability predicted by Koopman compared to the mean absolute prediction error (MAE) across 20 executions of various Monte Carlo approaches.

Task	Path Thru Gap			Mobile Obstacles		
Maneuver	(# samples)	**-0.1**	**+0.6**	(# samples)	**+0.1**	**+0.0**
AE from Koopman	121	1.328%	0.628%	360	0.212%	0.486%
MAE from Fair	121	2.849%	1.076%	360	0.728%	0.994%
	200	1.876%	1.100%	2000	0.464%	0.418%
	400	1.150%	0.501%	10,000	0.206%	0.235%
MAE from Uniform	121	2.617%	1.000%	360	1.605%	2.184%
	200	2.325%	0.629%	2000	0.731%	0.585%
	400	1.795%	0.543%	10,000	0.358%	0.386%

is called with the "current" state and time set to the expected state and time following the maneuver under consideration. This recursion continues until the expected state leaves the planning horizon or reaches the goal, at which point the accumulated collision probability is returned. By including collision probability from maneuvers planned from the expected exit state, the procedure generates a heuristic that discourages choosing maneuvers that are safe now but leave the vehicle with no safe maneuvers later. Procedure 2 receives from its call to Procedure 3 a heuristic value that describes how risky each maneuver is. From the maneuvers with risk less than the tolerance, the maneuver with the lowest cost is selected; if no maneuvers have acceptable risk, the least risky maneuver is chosen.

2.3.2 Path Through a Gap

In the first example, the vehicle chooses whether to travel around a wall or through a narrow gap, while its motion is perturbed by a random but uniform wind field (see Figure 2.2). The winds are randomized at each maneuver, but are held constant (at this randomized value) throughout the execution of a maneuver. Winds for one maneuver are conditionally independent of the winds during past or future maneuvers.

In this example, the obstacle is encountered during the first maneuver – therefore, the planning horizon is only a single maneuver. Note, however, that when simulating vehicle

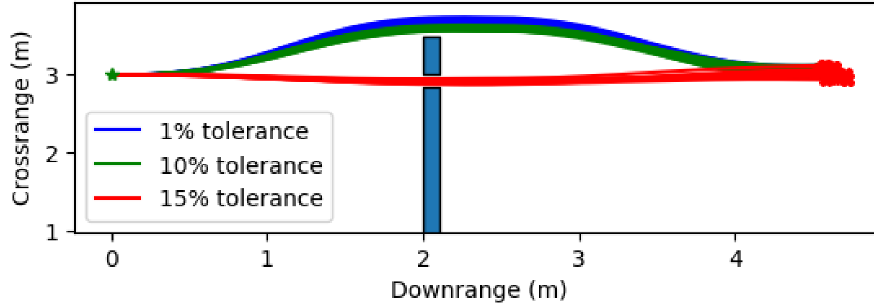

Figure 2.2: The vehicle begins at $x_0 = [0.0, 3.0, 0.0, 0.0]^T$ and seeks to navigate to $[4.5, 3.0, 0.0, 0.0]^T$. Ten runs at each tolerance are pictured for illustrative purposes. Red Xs mark locations where a run terminated.

motion, a new path is planned after each maneuver is executed. Since the paths to the goal tend to be two maneuvers long, this results in the possibility of different paths being generated in each simulation, since a different realization of the wind occurs. This feedback path planning has a greater effect in the second example shown in the next section.

Figure 2.2 shows 10 simulation runs with the risk tolerance r set to 1%, 10%, and 15%. In order to achieve a 1% collision probability, the planner elects to go completely around the wall. Note that, due to a limited number of maneuvers, the planner cannot achieve exactly 1% collision probability; rather, it picks the best maneuver with <=1% chance of collision. In this case, the selected maneuver is the +0.7m maneuver, which carries a predicted collision probability of 0%. In 100,000 runs with randomized winds, 5 collisions occur.

The appeal of explicitly chance-constrained planners is that, by changing a single physically meaningful tuning parameter (the risk tolerance), fundamentally different trajectories can be obtained. Increasing the risk tolerance to 10% causes the planner to select the +0.6m maneuver, which passes closer to the wall and has a predicted 2% chance of collision. From 100,000 runs, the actual risk is 2.6%. Increasing the risk tolerance further to 15% causes the planner to navigate through the gap with the -0.1m maneuver. This path is shorter but more dangerous than going around the wall. The predicted risk for this selected primitive is 12% and the actual risk from 100,000 randomized simulations is 11.5%.

The probability of collision during a maneuver can also be evaluated using a Monte Carlo technique, similar to the methods used in in [11] and [16]. Trajectories are generated offline for randomly sampled wind conditions and stored; at the planning step, these trajectories are shifted to start at the current state and checked for collision. Two proposal distributions are considered: "fair sampling" in which values are drawn directly from the wind distributions, and "uniform sampling" in which a uniform proposal distribution is used (for further discussion of proposal distributions, see [16] and [26]). The top portion of Table 2.1 compares the absolute error in collision probability predicted for the -0.1m and +0.6m maneuvers using the Koopman operator with 121 samples to the average absolute error from 20 executions of fair and uniform Monte Carlo sampling with 121, 200, and 400 samples. All random numbers are generated using the numpy 1.17.4 PCG64 implementation. As shown by the results in Table 1, on average both fair and uniform sampling are less accurate than Koopman with 121 samples. Furthermore, 400 samples are needed to reduce the average Monte Carlo error below that of Koopman with only 121 samples. While the computational effort incurred by running the additional 379 samples required by Monte Carlo to achieve similar accuracy is minimal for this simple system, for systems in which the dynamics are expensive to simulate this difference can be significant.

2.3.3 Mobile Obstacles

The second example replaces wind uncertainty with obstacles that move randomly. During each maneuver each obstacle moves in a random direction at a random speed between 0 and 2 m/s. Directions are independently and uniformly distributed and speeds are independently drawn from N(1,0.2). Speeds and directions are constant during the maneuver, and winds are set to 0.

The planner must predict the future locations of the obstacles. The uncertainty space is sampled using 10 points in velocity and 36 points in direction, for a total of 360 samples per obstacle. The planner heuristic assumes that a particular obstacle motion sample continues

at the same direction and speed during all primitives in a given path.

Figure 2.3 shows example planner solutions with three obstacles for $r = 10\%$ and $r = 20\%$. The algorithm replans after each maneuver, so each run can result in a different path. The blue circles show the obstacle locations at the final time while the black show the locations at each time step (the upper left obstacle interfered with the 20% path only after the vehicle was already past). At $r = 10\%$, the path moves widely around the obstacle field. One thousand runs result in 12 collisions. Increasing the planner's tolerance to 20% causes it to accept narrower clearance with respect to the first obstacle and to pass between the second pair, leading to a shorter overall path. In 1,000 runs at 20% risk, there are 91 collisions. The plotted 10% and 20% runs use the same seed for comparison purposes.

The bottom part of Table 2.1 compares collision probability predictions via Koopman and Monte Carlo for two maneuvers available at the initial state of the mobile obstacle task. From one million runs, the true collision probability for the +0.1 maneuver is 5.239%; for the +0.0 maneuver it is 6.913%. On the +0.1 maneuver, fair Monte Carlo needs 10,000 samples to match the accuracy of Koopman with only 360 samples. The +0.0 maneuver is more forgiving; fair Monte Carlo needs only 2,000 samples to outperform Koopman with 360. Uniform sampling is less accurate than fair sampling at all tested sample counts.

2.3.4 Scylla and Charybdis

A third example showcases the flexibility of the Koopman operator approach by removing model uncertainty and replacing it with environmental uncertainty. In this example, the wind components are set to 0 and uncertainty takes the form of two defined regions of space in which there is a periodic chance, but not guarantee, of the vehicle being destroyed. The larger region, dubbed "Charybdis," imposes a 40% chance of destruction every 0.25 seconds that the vehicle is inside the region. The smaller region, dubbed "Scylla," imposes only a 2.5% chance of destruction, but this happens every 0.025 seconds. As additional complexity, the first maneuver the vehicle makes will not enter either of the dangerous re-

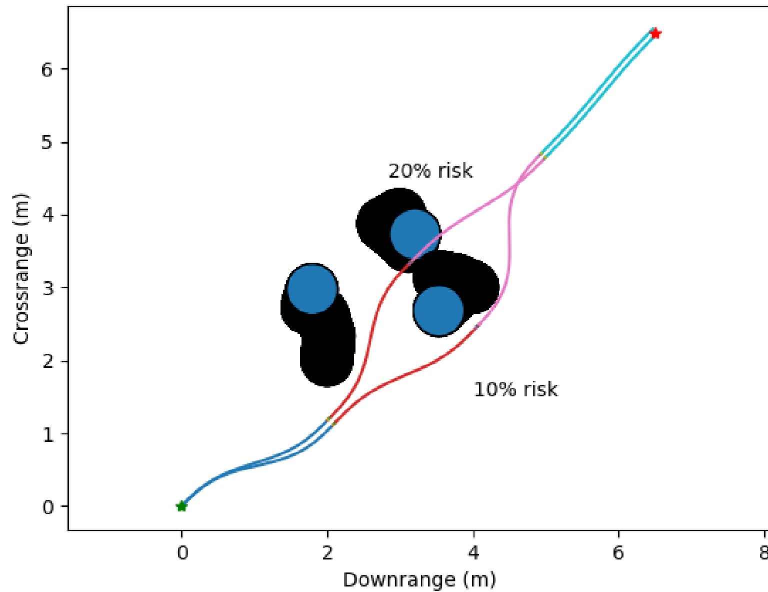

Figure 2.3: To achieve a $< 10\%$ risk of collision, the planner elects to swing wide around the obstacle field. At 20% risk tolerance, it can pass between the obstacles and achieve a shorter path.

gions. Note that, because the model is deterministic for this example, the planner's heuristic risk returned from Procedure 3 is exact.

Chance-constrained planning is a natural fit for tasks that are difficult to express in a robust or deterministic context. The application of the expected state planner to this task is straightforward, but robust and deterministic planning techniques cannot really tackle this challenge. A "robust" solution to this sort of probabilistic obstacle would be to path around the obstacle, completely ignoring its probabilistic nature. A deterministic planner could either treat the probabilistic obstacles as hard obstacles and path around them, or ignore the non-zero chance of destruction if the path goes through the obstacle and path through them. None of these approaches make full use of the available probabilistic information, while chance-constrained planning does.

Figure 2.4 shows 10 simulations of the expected state planner navigating with risk tolerances of $r = 1\%$, $r = 10\%$ and $r = 50\%$. The vehicle must travel completely around the dangerous regions in order to achieve a destruction probability of less than 1%. Observe

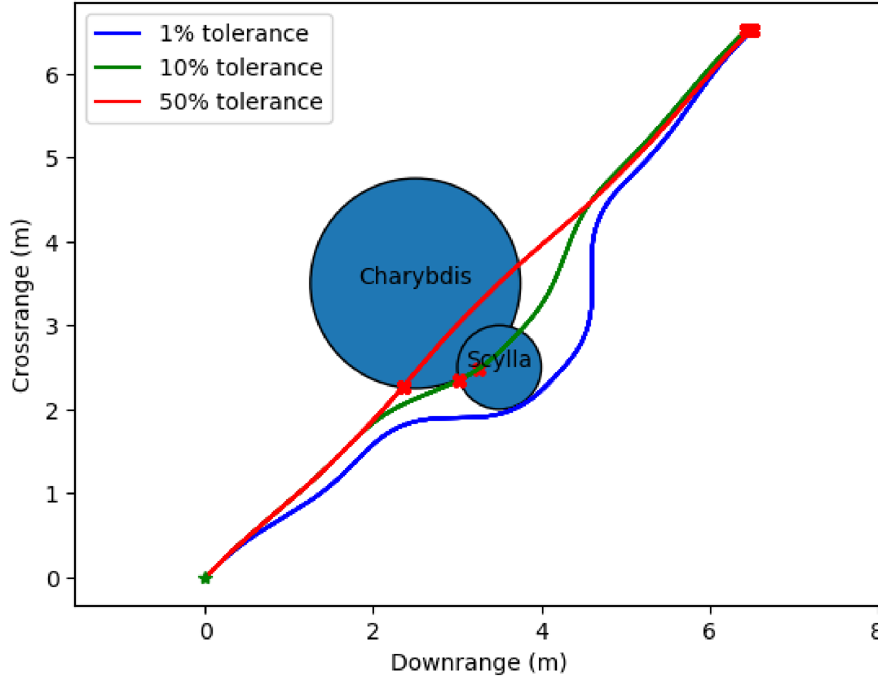

Figure 2.4: The vehicle begins at $x_0 = [0.0, 0.0, \pi/4, 0.0]^T$ and seeks to navigate to $[6.5, 6.5, \pi/4, 0.0]^T$. Only the first ten runs at each tolerance are pictured. Red Xs mark locations where a run terminated.

that the 0m maneuver is cost-optimal for the first maneuver, but would force the vehicle to path through Scylla on its next maneuver. The future collision probability calculation detects this, so the planner elects to divert right so that it can path completely around Scylla with its next maneuver. Increasing the risk tolerance to 10% means a passage through Scylla is acceptable. In 1000 runs, there are 93 deaths for a 9.3% destruction rate. Finally, if the planner is permitted a 50% risk tolerance, passing straight through Charybdis is allowed. Forty percent of the runs are destroyed as soon as they enter Charybdis' reach, consistent with Charybdis' 40% probability of kill.

2.4 Discussion

These computational results demonstrate the utility of the Koopman based maneuvers approach in terms of sample efficiency. On the tested problems, the Koopman approach was more accurate with fewer samples than not merely "fair" Monte Carlo, but also Monte Carlo

using a uniform proposal distribution. This proposal distribution is chosen because it results in the same weighted sample behavior Koopman exploits. In fact, the samples/weights used by the Koopman operator are possible, though naturally highly improbable, under uniform importance sampling. An additional example showcases the flexibility of the Koopman approach by studying a scenario with purely environmental uncertainty. However, the planner used for these problems is extremely simple and so addressing more interesting problems will require a more sophisticated algorithm. Developing an efficient such algorithm is the focus of the remainder of this thesis.

Procedure 1 AO*(root)

```
1:  Q ← {}; insert root into Q with priority 0
2:  while not complete(root) and Q is not empty do
3:      N ←  pop lowest priority from Q s.t. N is on current best path
4:      if N is OR then
5:          for all ACTION in ACTIONS(N) do
6:              H←heuristic(ACTION)
7:              cost_to_go(ACTION)←H
8:              insert ACTION into Q with priority H+cost_to_come(N)
9:      else
10:         for all CHILD in CHILDREN(N) do
11:             H←heuristic(CHILD)
12:             cost_to_go(CHILD)←H
13:             if CHILD in goal region then
14:                 complete(CHILD)← TRUE
15:             else
16:                 insert CHILD into Q with priority H+cost_to_come(CHILD)
17:     U_Q←{N}
18:     while U_Q is not empty do
19:         N←pop from U_Q
20:         if N is OR then
21:             A'← argmin_{A∈ACTIONS(N)} cost_to_go(A)
22:             best_action(N)←A'
23:             cost_to_go(N)←cost_to_go(A')
24:             if all CHILDREN(N,A') are complete then
25:                 complete(N)←True
26:         else
27:             cost_to_go(N)← Σ_{C∈CHILDREN(N)} P(N,C)[action_cost(N,C) + cost_to_go(C)]
28:         for all P in PARENTS(N) do
29:             if N in CHILDREN(P,best_action(P)) then
30:                 append P to U_Q
31: return root
```

Procedure 2 choose(past_cp, $x_{\text{obs}}, t_{\text{obs}}, x_0, t_0$)

1: $\Pi \leftarrow$ set of maneuvers compatible with x_0
2: cp $\leftarrow$ {evaluate(past_cp, π, x_{obs}, t_{obs}, x_0, t_0) : $\pi \in \Pi$}
3: safe_maneuvers $\leftarrow$ {$\Pi[i]$: cp[i] $< r$ }
4: **if** safe_maneuvers is empty **then**
5: safest_maneuver_id $\leftarrow \operatorname{argmin}$ cp
6: best_maneuver $\leftarrow \Pi$[safest_maneuver_id]
7: best_cp $\leftarrow$ cp[safest_maneuver_id]
8: **else**
9: costs $\leftarrow \{\sum_{j=1}^{n} P(s_j) J(H_\pi(x_0, t_0, s_j))\Delta s$: $\pi \in$ safe_maneuvers}
10: bmi $\leftarrow \operatorname{argmin}$ costs
11: best_maneuver $\leftarrow$ safe_maneuvers[bmi]
12: best_cp $\leftarrow$ cp[index of best_maneuver in Π]
13: **return** best_cp, best_maneuver

Procedure 3 evaluate(past_cp, π, x_{obs}, t_{obs}, x_0, t_0)

1: cp $\leftarrow 1 - \prod_i (1 - \sum_{j=1}^{n} P(s_j) C_i(H_\pi(x_0, t_0, s_j))\Delta s)$
2: total_cp $\leftarrow 1 - (1 -$ past_cp$) * (1 -$ cp$)$
3: $x_e \leftarrow \bar{g}_\pi \circ x_0 + \bar{y}_\pi$
4: $t_e \leftarrow t_0 + \bar{\tau}_\pi$
5: **if** x_e is a goal **or** $x_e \notin$horizon(x_{obs}) **then**
6: **return** total_cp
7: **else**
8: cp,_ $\leftarrow$choose(total_cp,$x_{\text{obs}}, t_{\text{obs}}, x_e, t_e$)
9: **return** cp

CHAPTER 3
OPTIMAL SEARCH ALGORITHMS

In chapter 2 the problem of obtaining a policy for a chance constrained path planning problem was linked to And/Or graph search. This chapter describes an existing algorithm for conducting this search, known as RAO*. RAO* is found to handle problems with large numbers of maneuvers poorly, so an action heuristic that leverages the properties of motion primitives is devised. The search is further accelerated by converting the tree structure to a graph (by rapidly detecting states that are close to each other), by using reduced numbers of Koopman samples at higher search depth, and by exploiting shared symmetry in constraints and dynamics (when available). A much larger library of maneuvers for the Dubins Car model from section 2.3 is presented and numerical results show AO* dramatically outperforming RAO* on problems involving this library. For further evidence, a maneuver library for the F-16 model in JSBSim [96] is introduced and RAO* is compared to AO* on an F-16 task too.

3.1 Algorithmic Contributions

3.1.1 Induced Heuristic for AO*

RAO* is an informed forward search algorithm on AND/OR graphs developed for finding partial policies for CC-POMDPs, first published in [55]. The primary advantage RAO* offers over the generic "path-planning AO*" described in Procedure 1 is that it can enforce chance-constraints. As emphasized previously, though, in conventional AO*, AND and OR nodes are BOTH placed in the priority queue and are expanded in whatever order their heuristic values indicate. When dealing with a state-space model where AND nodes are actions, though, AND nodes effectively have the same heuristic value as their parent OR

nodes. As a result, in RAO*, the algorithm expands a single OR node based on the order maintained in a priority queue, then immediately expands all AND nodes descending from the expanded OR node. This leads to the same order of expansions that "proper" AO* would perform. A heuristic function that can distinguish between AND nodes, however, offers potential performance improvements (the simulation results in section 3.2 demonstrate an order of magnitude improvement in runtime). Uncertain motion primitives provide a way to construct an "induced" heuristic for AND nodes from a generic heuristic function defined on points in the state space:

$$h(n_i) = \sum_{j=1}^{n} P(s_j)\Delta s(\text{action_cost}(n_i, n_j) + \text{cost_to_go}(n_j)) \tag{3.1}$$

where $h(n_i)$ is the expected cost to go from AND node n_i, action_cost(n_i, n_j) is the cost to travel from the state at n_i (which is the state corresponding to the OR node parent of n_i) to the state at OR node n_j by following the action corresponding to node n_i, and cost_to_go(n_j) is the estimated cost to go from OR node n_j. The state space representation of n_j is obtained via the group displacement history recorded for the realization: state$(n_j) = g_{\pi_i}(t_f - t_0, s_j) \circ$ state(n_i). If n_j is a goal, cost_to_go(n_j) is assigned the value of the penalty function (if any); if it is not a goal and has not yet been expanded, cost_to_go(n_j) is initialized with the problem specific heuristic evaluated at the state corresponding to n_j. If state merging is active, n_j may be an existing OR node; cost_to_go(n_j) would then have a value that was computed by a previous policy update (see Procedure 4, line 26).

With the induced heuristic in equation (Equation 3.1) available, AND nodes can be added to the priority queue as in AO* and the children of an expanded OR node will not necessarily be immediately expanded. The algorithm, given in Procedure 4, is a straightforward modification of RAO* to use a proper AO*-like expansion order instead of the special case of OR node followed by all child AND nodes.

Starting with the root node (an OR node), nodes are expanded in a best-first order. The

current best node is the node with the lowest cost to come plus estimated cost to go, chosen from nodes that are reachable from the root by following actions that are currently marked as the best action. Once the next node has been selected, it is expanded.

An OR node has all its AND children generated, and these AND children have their induced heuristic computed per equation (Equation 3.1). The AND children are initialized with 0 risk of constraint violation and placed into the queue; the priority is the cost to come to the parent OR node along the current best path plus the induced heuristic of the action.

An AND node has its OR children generated and their estimated cost to go is as described for equation (Equation 3.1): if the OR node already exists, it has an estimate from a previous policy update. If the OR node is in the goal region, the cost to go is the value of the penalty function. Otherwise, it is the value of the problem heuristic. AND expansions are where collision checking occurs; if a constraint is violated on the path to an OR child, the risk of that OR child is set to 1 and the node is ALSO marked complete. Non goal children with 0 risk are then inserted into the queue.

Expansion is followed by the RAO* policy update, presented here with the simplifications that result from restriction to full observability and the use of motion primitives to represent actions. This is a backpropagating process (as discussed in [54], it is in fact a very simple dynamic programming algorithm). Beginning with the node N just removed from the queue, the expected cost to go to the goal is updated based on the children of N.

If N is an OR node, the best action at the node is the action with lowest cost to go (obtained either from the induced heuristic or a previous policy update) that also satisfies the risk tolerance. For completeness, if no such action is available the algorithm sets the best action to the lowest risk action. The cost to go from the OR node is then the cost to go through this best action, and the risk is similarly the risk through the best action. The termination condition for AO* is that the root node is marked complete; this is tracked by marking an OR node complete when all OR children of its best action have been marked complete.

If N is an AND node, the cost to go is computed as the expected value of the transition cost to each child plus the cost to go at the child; this is yet another application of equation (Equation 3.1). In a similar fashion, the Koopman operator may be used to estimate the risk of constraint violation as the expected value of the risk at the child OR nodes. AND nodes do not need to track completion and their outgoing edges are always all part of the best path.

The policy update loop continues by collecting all the nodes which have edges leading into N. Of the parents, nodes whose currently-marked best path lead to N are placed in a first-in-first-out queue. The first entry in the queue is removed and updated as previously discussed; this loop continues until the queue is empty (which happens when the root node is updated). To minimize redundant calculations, if a node is added to this queue when it is already present, the node is not replicated but is pushed to the end of the queue. This repetition cannot happen if the search graph is a tree, but state merging (subsection 3.1.2) results in a graph where this can occur.

Once the policy update has finished, the algorithm checks if the root node was marked complete by the update step or if the priority queue is empty. In the first situation, an optimal partial policy is recorded as the best action markings in the current tree descending from the root. In the second situation, no assignment of maneuvers leads to the goal without violating the risk tolerance and so no solution is returned.

A visualization of the algorithm is provided in steps 1-6 of Figure 3.1. In step 1, the queue contains only the start state A. The start state is selected to be OR expanded. In step 2, the two actions at A have been added to the queue; the dotted lines show that the connections have not been checked for collision and the hollow circles denote that the child states have not been added to the graph. The queue contains only the actions A1 and A2. In step 3, action A1 is selected to be expanded, resulting in two new OR nodes B and C after checking their connections for collision. The queue now contains action A2 and states B,C. In step 4, state C is OR expanded. There are now three actions A2,C1,C2 and a single state

B in the OR queue. In step 5, B is OR expanded, leaving a queue containing only actions A2,C1,C2,B1,B2. In step 6 action B2 is expanded. This process would continue until the root node is marked complete per the rules of the policy update step.

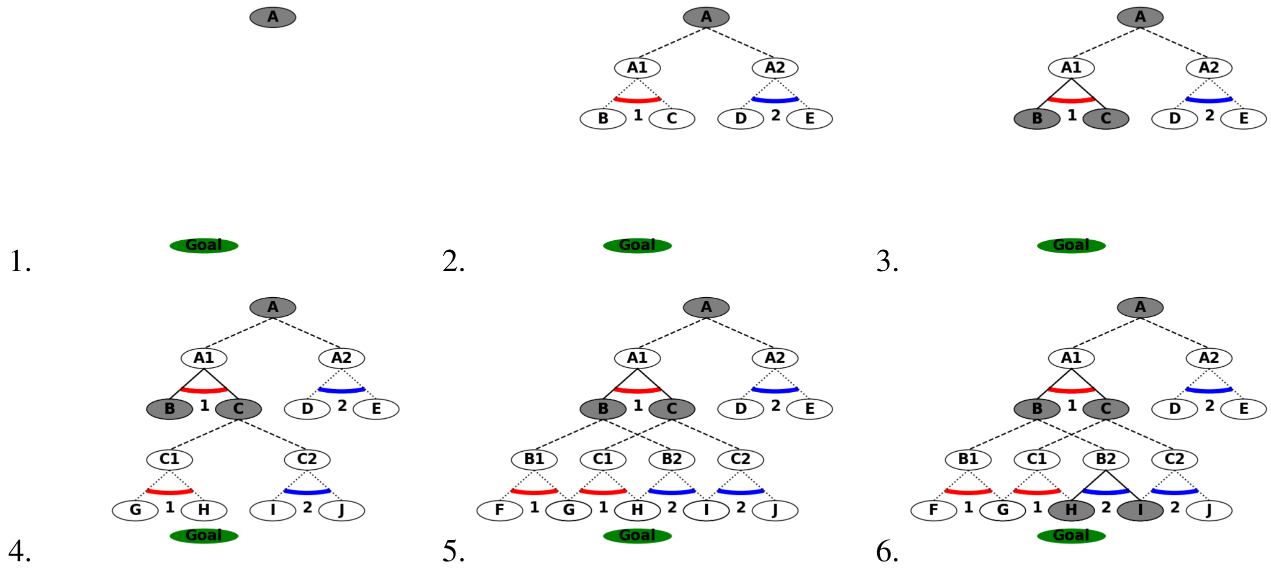

Figure 3.1: Visualization of AO* with the induced heuristic. OR nodes are labeled with letters; AND nodes have the letter of the parent OR node followed by a number. A shaded OR node has had the path to it fully collision checked. In step 1, the OR queue is initialized with the root node (A). In step 2, A is OR expanded. In step 3, A's action 1 is AND expanded. In step 4, C is OR expanded. In step 5, B is OR expanded. In step 6, B has action 2 AND expanded.

Contrast this with what would happen if RAO* were used. Figure 3.2 shows RAO* search. In step 2, RAO* has already collision checked the paths to D and E; AO* never checks those paths for collision.

Compared to RAO*, AO* with the induced heuristic reduces the total number of trajectories that need to be checked for constraint violations (by reducing AND expansions), at the cost of increased overhead in the form of the induced heuristic and (potentially) additional OR expansions. The major cost of both of these forms of overhead is in computing the exit state of a maneuver. Thus, the induced heuristic technique is particularly well-suited to motion primitive problems, where the exit state can be computed without needing to compute the entire trajectory. If it were necessary to integrate the dynamics every time

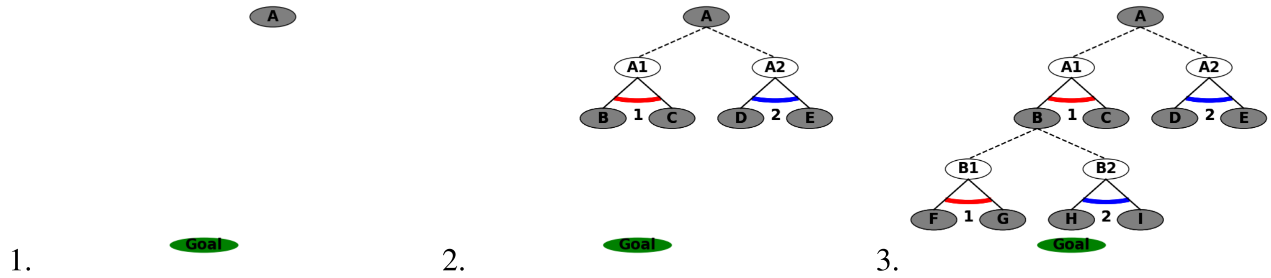

Figure 3.2: Visualization of the RAO* algorithm. OR nodes are labeled with letters; AND nodes have the letter of the parent OR node followed by a number. A shaded OR node has had the path to it fully collision checked. In step 1, the queue is initialized with the root node (A). In step 2, A has been OR expanded and had all its actions AND expanded. In step 3, B has been OR expanded and had all its actions AND expanded.

a new exit state were desired, computing the exit state would not be much cheaper than performing the constraint violation check. Similarly, if checking for constraint violations can be performed quickly (such as through the use of the presorting technique described in Figure 3.4), the benefit of the induced heuristic is reduced.

The ability of the induced heuristic to actually save collision checks compared to RAO* depends on the characteristics of the maneuver library. In the extreme case, a library containing only one maneuver at every state would offer no room for improvement on RAO*'s performance. Under strong assumptions about the accuracy of the problem heuristic, however, AO* with the induced heuristic is guaranteed to outperform RAO*:

Theorem 1 (Efficiency with perfect heuristic) *If the problem's heuristic function always returns the true cost to go from a state, AO* and RAO* both expand only OR nodes on the true best path. With the induced heuristic, AO* expands only AND nodes on the true best path, while RAO* expands all AND children of best path OR nodes.*

Proof: With a perfect heuristic, RAO* always selects OR nodes from the best path for expansion. RAO* will necessarily expand all AND children of the expanded OR nodes. AO* will also only select best-path OR nodes for expansion.

However, the induced heuristic is constructed as the expected value of the problem heuristic. For a perfect problem heuristic the induced heuristic returns the correct expected

cost to go for a particular action. Thus, AO* with the induced heuristic will only expand the best action at each OR node. ■

Simulation results in section 3.2 show that, in practice, the induced heuristic allows AO* to conduct fewer AND expansions than RAO* and that this results in (significantly) reduced runtime.

3.1.2 State Merging

Several modifications can be made to R/AO* to improve computational performance. The most basic modification is to conduct the search on an AND/OR graph instead of an AND/OR tree. The basic expand step does not consider the possibility of there being multiple paths to a given state. Every child is assumed to be a unique state. However, both RAO* and AO* are perfectly capable of handling a search hypergraph that is not a tree (as long as there are no cycles). Failing to combine states that are actually the same results in redundant expand and update calls.

State redundancy in the search process is not caused by a flaw in the search algorithms themselves; rather, using motion primitives to construct the states lacks an inherent notion of two states being the "same." The solution is to define a discretization interval in each state dimension and record a unique identity for each state generated during the search. When a state is generated, modular arithmetic can be used to determine if it falls within the same hyper-rectangle as an existing state; if it does, instead of adding a new state, the existing state is reused in the search algorithm. This eliminates the need to compute and store an actual state discretization in memory, or to compute discretized state displacements for every realization. The result is a decomposition of the state space into cells as shown in Figure 3.3, but each occupied cell is associated with a continuous state that can be located anywhere inside the cell. This is the approach taken by Hybrid A* [97]. Naturally, using a coarser discretization reduces the size of the hypergraph and reduces algorithm runtime. This is at the expense of accuracy as non-identical states get treated as identical.

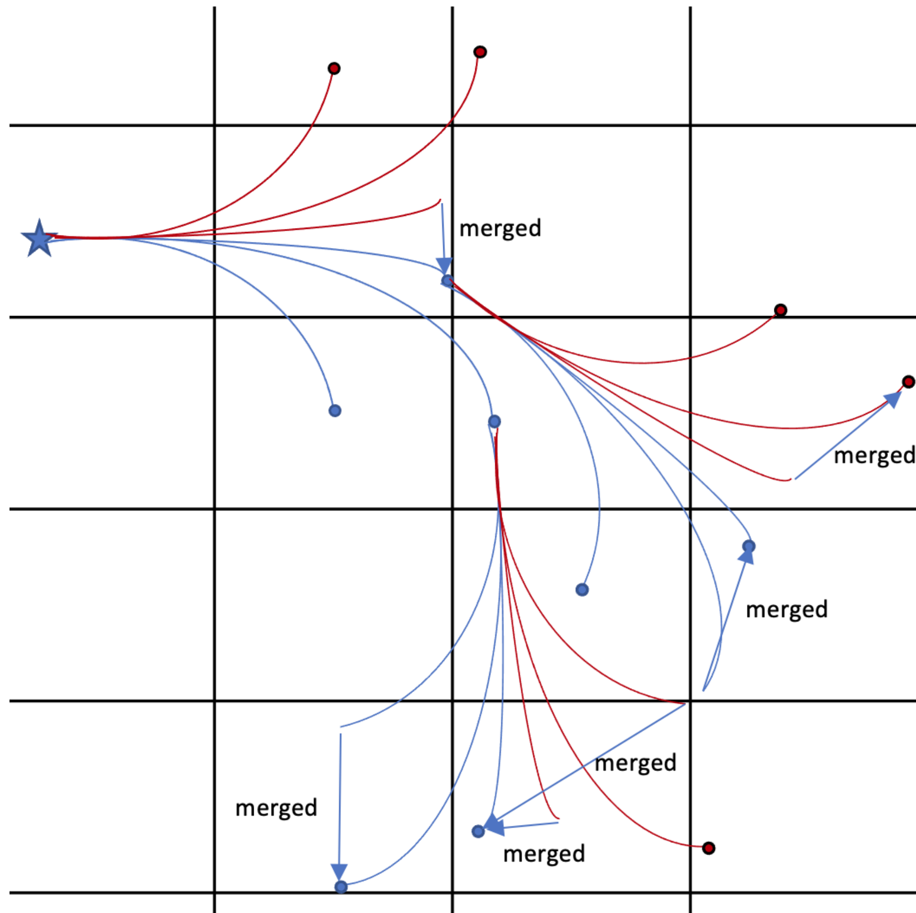

Figure 3.3: Schematic view of cell-based state merging. Two maneuvers (red and blue) are shown; each maneuver has three realizations. Observe that the location states that are merged need not be in the center of the cell.

3.1.3 Decaying Accuracy

The second algorithm modification, decaying accuracy, is applicable in online planning scenarios. Online execution places stringent performance demands on planning algorithms but offers a critical advantage over offline planning: the ability to change the plan based on new information. The far-future components of a plan may never actually be executed; instead, a new plan will be formed and followed. Yet, the number of OR nodes that must be examined by the planner is between $\sum_i \prod_{j\leq i} n_j$ (examine only nodes on the best path) and $\sum_i \prod_{j\leq i} m_j n_j$ (examine every node), for m_j the number of maneuvers available at a state at depth j and n_j the number of samples at depth j. Both sums are dominated by nodes

at high depth, yet the vehicle will not actually execute the portion of the plan made in this region.

Significant runtime savings can be achieved by using fewer parameter samples when j is large. This reduced sampling resolution means that the risk of collision with distant obstacles will be computed with less accuracy; the accuracy of the cost estimate will also be reduced. The argument for this technique is that neither of these effects is likely to create major changes in the actions that should be taken in the near future, and by the time the obstacles are no longer distant a new plan will be computed with higher resolution in the regions those obstacles occupy. Varying the level of detail used in the planning algorithm based on the search depth is inspired by the approach taken in [58], where after a user-specified time horizon the planner stops checking for collisions between the vehicle and certain hard-to-predict obstacles.

3.1.4 Presorting Realizations

The third algorithm modification applies in the specific case of planar inequality constraints. Planar inequality constraints are an important class of path planning constraints representing objects like the ground and impassable walls. Other types of common planar inequality constraints include structural or heating limits that cannot be exceeded. Planar inequality constraints are also closely related to convex polytope obstacles, which can be represented as conjunctions or disjunctions of planar inequalities. Planar inequality constraints can possess symmetry properties similar to those necessary for motion primitive planning, offering an opportunity to accelerate collision checking when the relevant symmetry groups are related.

Consider a plane with normal vector $\hat{n}$ passing through point $\vec{p}$. If the state $\vec{x}$ (which may be a partial state if the plane only occupies a subset of the dimensions of the state space) is required to stay on one side of the plane, the constraint can be written as $\langle \hat{n}, \vec{x} -$

$\vec{p}\rangle \geq 0$ (inner product). With the introduction of a slack variable l, the inequality becomes:

$$\langle \hat{n}, \vec{x} - \vec{p} \rangle = l \tag{3.2}$$

$$l \geq 0 \tag{3.3}$$

For a trajectory, l is of course a function of time.

Theorem 2 *Let $\mathbb{G}_p$ be a Lie group to which the dynamics are invariant. If $\forall g \in \mathbb{G}_p, \langle \hat{n}, g \circ \vec{x} - \vec{p} \rangle = \langle \hat{n}, \vec{x} - g^{-1} \circ \vec{p} \rangle$, the time in a trajectory at which l is smallest will be unchanged by a group action. If this point is determined once, the minimum slack of any trajectory offset from the original by a group action can be computed by a group action. Furthermore, the ordering of system trajectories by their maximum decrease in slack is itself invariant to the action of $\mathbb{G}_p$.*

Proof: Considering only cyclic coordinates, for a particular maneuver realization beginning at initial condition $\vec{x}_0$, the trajectory is given by $H_\pi(t, t_0, \vec{x}_0, s_j)$ and the slack can be computed as:

$$l_0(t) = \langle \hat{n}, H_\pi(t, t_0, \vec{x}_0, s_j) - \vec{p} \rangle \tag{3.4}$$

$$= \langle \hat{n}, H_\pi(t, t_0 \vec{x}_0, s_j) \rangle - \langle \hat{n}, \vec{p} \rangle \tag{3.5}$$

If the initial condition is changed to $g \circ \vec{x}_0$, then by group invariance the slack becomes:

$$l_g(t) = \langle \hat{n}, g \circ H_\pi(t, t_0, \vec{x}_0, s_j) - \vec{p} \rangle \tag{3.6}$$

$$= \langle \hat{n}, H_\pi(t, t_0, \vec{x}_0, s_j) - g^{-1} \circ \vec{p} \rangle \tag{3.7}$$

$$= \langle \hat{n}, H_\pi(t, t_0, \vec{x}_0, s_j) \rangle - \langle \hat{n}, g^{-1} \circ \vec{p} \rangle \tag{3.8}$$

$$= l_0(t) + \langle \hat{n}, \vec{p} \rangle - \langle \hat{n}, g^{-1} \circ \vec{p} \rangle \tag{3.9}$$

$$= l_0(t) + \langle \hat{n}, \vec{p} - g^{-1} \circ \vec{p} \rangle \tag{3.10}$$

Thus, the group action causes a constant offset in the slack. Then, the t which minimizes l_g is the same as the t that minimizes l_0. Additionally, it means that the change in slack over the course of a realization is constant under group action. The ordering of realizations by their maximum decrease in slack is thus invariant to the group action. ■

Using these properties, most of the computational effort of checking a maneuver realization against a planar inequality can be done offline. To illustrate this, consider Figure 3.4. For a particular inequality constraint, such as a line parallel to the one making up the diagonal side of the triangle in Figure 3.4, the maximum decrease in slack is computed for every realization of every maneuver in the library and the realizations are sorted by this value (the decrease is invariant to group action, so it can be computed from an arbitrary initial condition). When the inequality needs to be checked, a binary search is done comparing the decrease in slack to the slack at the state the maneuver is to be executed from. The binary search returns the realization with the smallest decrease in slack that causes it to violate the constraint. All realizations whose slack decreases more than the result of the binary search also violate the constraint, while all realizations whose slack decreases less than the result are guaranteed not to violate it. Not only does this avoid checking every realization for constraint violation, but it also simplifies the process since each realization that is checked requires comparison of a single value instead of comparisons at (presumably numerous) timesteps along the realization.

In practice, $\mathbb{G}_p$ will only be a subgroup of $\mathbb{G}$. For example, dynamics are often invariant to both translation and rotation, but a planar inequality will only be invariant to translations. Dimensions of $\mathbb{G}$ that are not included in $\mathbb{G}_p$ will need to be sampled and a separate offline sort done for each sample. This discretization creates the opportunity for error in collision checking if the discretization is too coarse. Consider a 2D planar state with heading. Planar inequalities in this domain are lines that should not be crossed (as in Figure 3.4). Shifting the initial position in the plane is invariant as discussed, but rotating the initial heading is not. A different sort order holds for every possible angle between the heading and the line.

If a line with a relative angle not in the sort needs to be checked, the nearest matching angle can be used, but this will be slightly inaccurate; some realizations may appear safe while not actually being safe, or vice-versa.

The ability to rapidly check planar inequalities can be used to accelerate checking convex polytopes too, since such obstacles are AND (stay outside) or OR (stay inside) combinations of planar inequalities. In the case of requiring that the vehicle stay outside a convex polytope, it is only possible to prove the constraint is not violated (by showing that the realization stays outside at least one member plane). Proving violation would require showing that all planes are violated at the same time instant, but the sorting-based planar inequality check described here does not include any tracking of the time of closest approach for the slack variable(s). Normal time-sliced collision checking must be conducted if the sorting-based check indicates a collision could occur. Computational effort is still saved, though, in situations where the sorting check proves a collision cannot occur. In Figure 3.4, realizations A and B are proven safe and do not need further checking, but C and D would need to be checked to confirm that they actually enter the triangle.

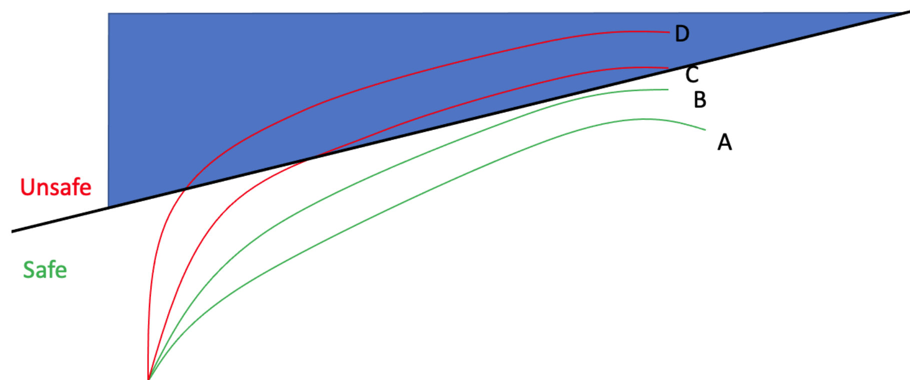

Figure 3.4: Obstacle region in blue and four realizations of a maneuver. Realization A is known from presorting to have the largest slack versus a line parallel to the black edge (while D has the least slack). Once it is known that realization C is unsafe while B is safe, realizations A and D do not need to be checked.

3.2 Algorithm Efficiency

To demonstrate the effectiveness of the induced heuristic and to study the effects of the three algorithm modifications, two dynamical systems are considered in several obstacle environments. The first system is identical to the Dubins Car from section 2.3, though a larger maneuver library and different tasks are considered. The second system is a 6-degree-of-freedom autonomous aircraft and is chosen to demonstrate the scalability of the maneuver-based planning approach to more complex systems. In particular, attention is drawn to the fact that planning difficulty is only loosely related to the complexity of the dynamics, as the equations of motion are not directly involved in planning.

All calculations were done using a 6 core Intel® Xeon® CPU E5-1650 v2 @ 3.50GHz, though no parallelism was leveraged internal to any planner. The planners were implemented in Python 3.6 using shared code to ensure comparable implementations. The numpy 1.17.4 library with OpenBLAS (not Intel's MKL) was used and, where possible, accelerated by numba 0.46. In particular, the group action, the L2 norm, and the goal test function were numba-compiled. Code is available on request.

3.2.1 Dubins Car Examples

Return to the Dubins Car model of (Equation 2.7). An expanded maneuver library is considered, with direct collocation [95] used to generate 51 maneuvers at each of 11 trim conditions at evenly-spaced initial angular velocities in the interval $[-.24, +.24]$ rad/s. The uncertainty space is discretized using 7 points in each dimension, spanning ± 3 standard deviations. For each maneuver, the "nominal" realization (with $w_x = w_y = 0$) has approximately zero angular velocity at the end of the realization, and all other realizations (with different values of w_x and w_y) have angular velocities at the end of the realization that are contained within the ± 0.24 rad/s interval. This results in a closed library without resorting to the recovery maneuvers needed in section 2.3. The nominal trajectories of the

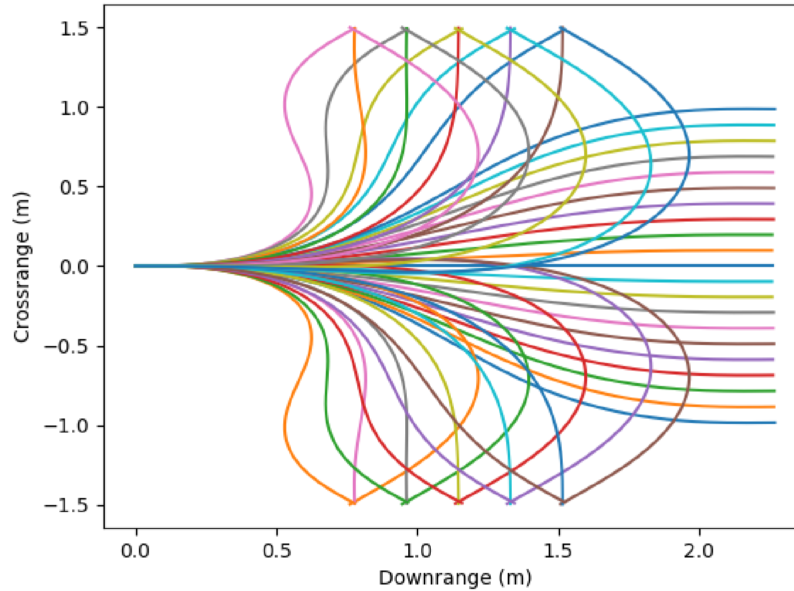

Figure 3.5: Nominal trajectories for the Dubins Car maneuver library, shown for the zero angular velocity trim.

zero angular velocity trim are shown in Figure 3.5.

Two planning tasks are considered. The first planning task, similar to the Path Through Gap task considered earlier, asks for the vehicle to travel from the state $[0.1, 3.0, 0.0, 0.0]^T$ to within 0.2 meters of the position $(x, y) = (4.5, 3.0)$ m or to get farther than 4.5 m downrange. It must do this while colliding with the walls in the environment no more than 20% of the time ($r = 0.2$), and should seek to minimize travel distance. The environment for this planning task is depicted in Figure 3.6. All the obstacles in this environment are convex polygons, and so presorting can be applied to accelerate collision checking. The $\mathbb{R} \times \mathbb{R} \times \mathbb{S}^1 \times \mathbb{R}$ state space is discretized using 0.05 m $\times$ 0.05 m $\times$ 0.05 rad $\times$ 0.05 rad/s cubes for state merging, and decaying accuracy is not employed.

The second planning task is designed to require deeper search. The initial condition and wall are unchanged from the first planning task, but the goal is moved to $(x, y) = (7.0, 3.0)$ m and another vertical wall (with a gap) is added at the $x = 4$ m position, with the gap at a $y = 2.5$ m. In addition to the fixed resolution planners, AO* is run with presorting and a

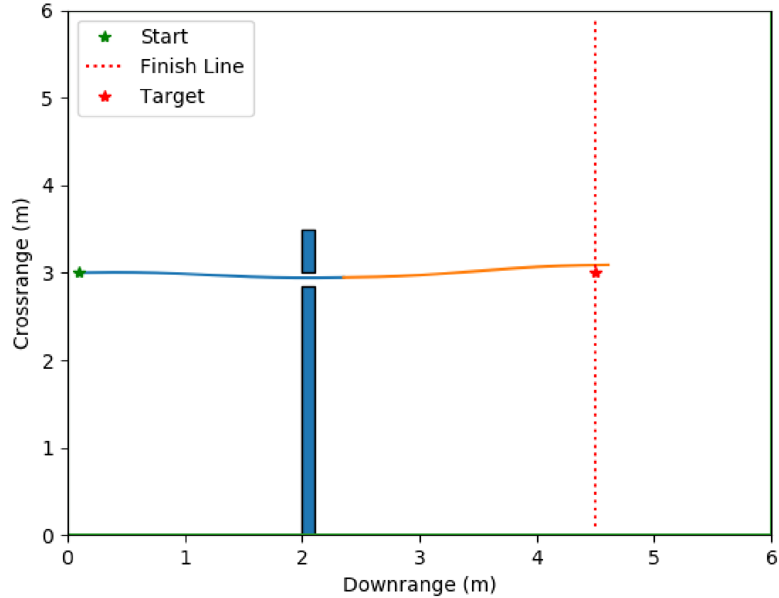

Figure 3.6: Environment for the first planning task with nominal realization of the probabilistically optimal solution shown.

decreasing number of samples: 81 at depth 1, 49 at depth 2, and 25 subsequently.

Figure 3.6 shows the nominal realization of the optimal path for the first planning task (both algorithms find the same solution). Furthermore, Figs. Figure 3.8 and Figure 3.9 show all the maneuver realizations checked for collision by AO* and RAO*, respectively, during the search process. In Figure 3.8, it can be seen that the search is highly focused, with little effort expended on suboptimal maneuvers. This is in contrast to the results in Figure 3.9, which show that RAO* has to consider every action at states chosen for expansion. This results in effort spent on maneuvers that do not even travel towards the goal.

Metrics describing the planning performance of RAO* and AO* (with and without presorting) are presented in Table 3.1 for both Car tasks and the F16 task. PAO* is AO* with the induced heuristic and presorting, and PRAO* is the RAO* algorithm with presorting. The Nodes column shows the number of OR nodes in the explicit AND/OR graph at the end of the search. OR and AND are, respectively, the number of OR expansions and AND

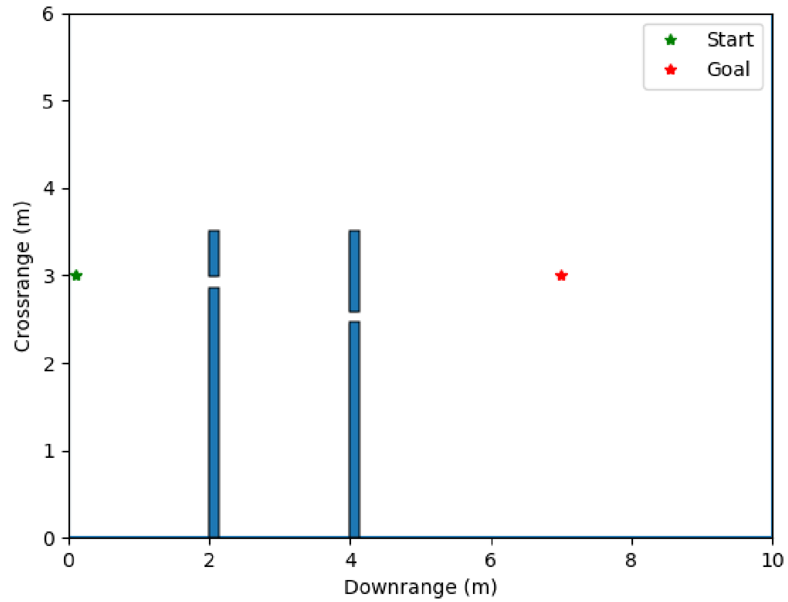

Figure 3.7: Environment for the second planning task, with a second vertical wall added to the first environment.

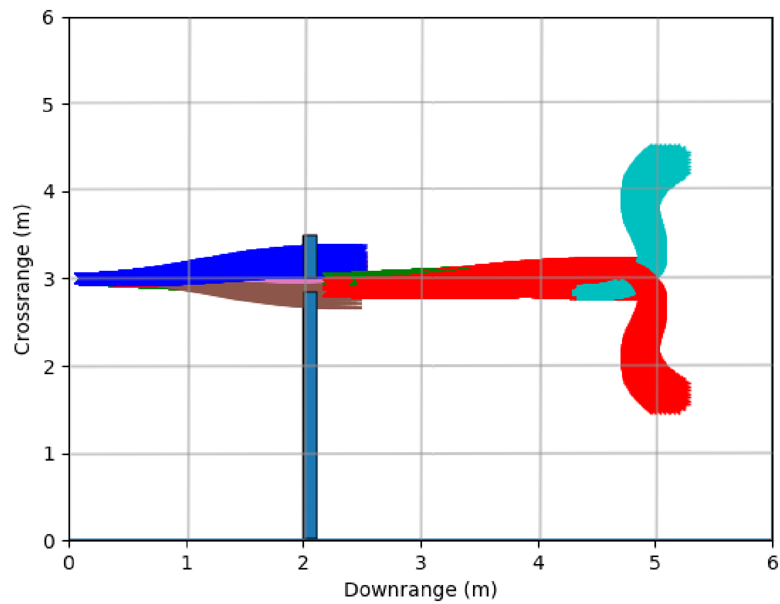

Figure 3.8: Realizations Collision-Checked by AO* Algorithm for First Planning Task.

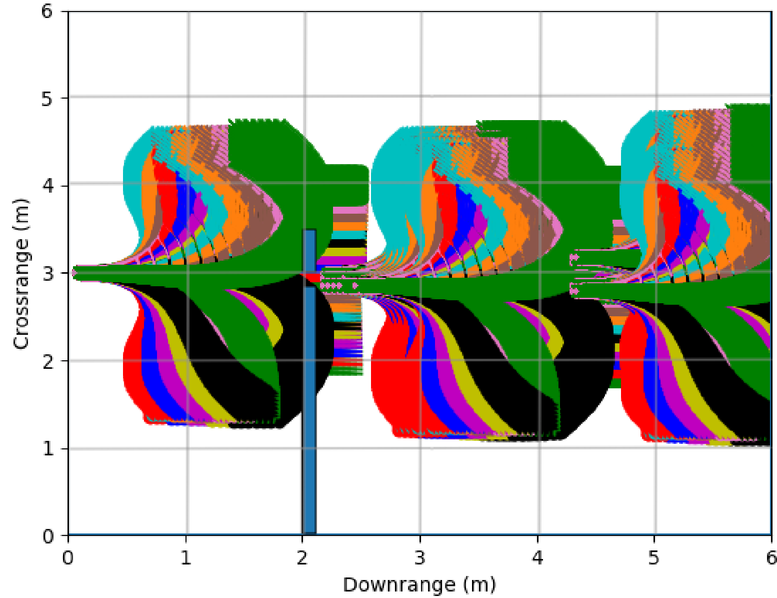

Figure 3.9: Realizations Collision-Checked by RAO* Algorithm for First Planning Task.

expansions. Checks reports the number of edges checked for collisions.

As seen in Table 3.1, the induced heuristic makes AO* dramatically faster than RAO* for all example tasks by reducing the number of AND expansions. Because the speed up comes by reducing the number of AND expansions, presorting is much less helpful for AO* than it is for RAO*. The effect of presorting is to make some AND expansions cheap (since the primary cost of an AND expansion is the collision check), but AO* spends a smaller fraction of its runtime doing AND expansions. There is simply less room for presorting to help. The effect is particularly stark in the first Car task, where presorting reduces the RAO* runtime by a factor of nearly 15 while having a negligible effect on the AO* runtime. However, presorting is available only if obstacles are convex polytopes, while the induced heuristic is defined for any scenario.

The Task 2 Decaying Accuracy row in Table 3.1 presents the results for AO* (with presorting) using decaying accuracy with 81 samples for the first maneuver, 49 samples for the second, and 25 for subsequent maneuvers for the second planning task. This is in contrast to the non-decaying algorithms, which use 49 samples at all depths. Decaying

Table 3.1: Planning Statistics for Each Example.

	Runtime (s)	Nodes	Merged	OR	AND	Checks
Car						
Task 1						
RAO*	129	35,354	27,122	25	1,275	62,475
PRAO*	8.88	35,352	27,124	25	1,275	1,077
AO*	6.41	1,381	41	25	29	1,421
PAO*	5.13	1,381	41	25	29	200
Task 2						
RAO*	8,030	3,827,484	1,635,331	2,186	111,486	5,462,814
PRAO*	1,580	3,826,783	1,636,032	2,186	111,486	84,266
AO*	1,130	90,866	194,707	2,132	5,828	285,572
PAO*	857	90,866	194,707	2,132	5,828	43,577
Decaying	209	33,750	14,459	1,632	1,632	48,208
F16						
Task 1						
RAO*	43.3	60,342	4,999	60	540	65,340
PRAO*	11.8	61,377	5,053	61	549	10,110
AO*	21.3	24,094	833	206	206	24,926
PAO*	12.9	24,094	833	206	206	5,377
Decaying	11.0	10,792	618	185	185	11,409

accuracy AO* returns the same best action as regular AO*, but the estimated risk increases from 12.5% to 18.3% and the estimated cost decreases from 7.3 m to 5.6 m.

These estimates were validated by simulating a closed loop planning process. The planner was executed to select the maneuver for the current state, then the controller for that maneuver was executed. A pseudo-random number generator produced the actual value of the wind parameters. Then, the planner was run again at the new location. This continued until the goal was reached. This process was conducted 100 times with the fixed and decaying accuracy AO* planners. The fixed accuracy planner experienced 17 collisions (slightly off from the predicted risk of 12.5%), while the decaying accuracy planner resulted in 18 collisions (closely matching the predicted risk of 18.3%). By using higher resolution sampling for the next maneuver taken (81 samples vs 49 for the fixed accuracy), the decaying accuracy planner produced a more accurate risk estimate than the fixed accuracy planner could while requiring less total computation.

It is important to note that in the second Car task, AO* and RAO* actually produce different optimal solutions to the problem (despite their theoretical equivalence). This is because state merging as described in subsection 3.1.2 means that the actual graph searched depends on the order of AND expansions. If two nodes are close enough to be merged, the state that will be used after merging to represent both nodes depends on which of the nodes was encountered first. Since AO* and RAO* do not in general perform expansions in the same order, they may search slightly different graphs when merging is active. In Car task 2, this effect is enough that RAO* and AO* select different actions at the root node. Similarly, note that RAO* and PRAO* exhibit slightly different merging behavior due to the finite accuracy of presorting. This results in the different number of merged nodes in Table 3.1.

3.2.2 F-16 Example

The second example studies a system that is higher-dimensional and is designed to illustrate application of the induced heuristic to a practical system. For the purposes of this work, the F-16 simulation model implemented in JSBSim is used [96]. This model is a nonlinear 6Degrees of Freedom (DOF) flight dynamic model consisting of six kinematic states (three for position, three for orientation) and six dynamic states (three body-frame velocity components, three body-frame angular velocity components). The control vector for this system is defined through the four aircraft control inputs: aileron, rudder, elevator, and thrust. Note that the 6DOF dynamics are invariant to rotation and horizontal translations, although a standard atmosphere model is employed which means that atmospheric pressure depends on altitude and thus the dynamics are not invariant to vertical translation. A motion primitive framework treating kinematic states x, y, and heading as cyclic coordinates is appropriate, with Lie Group $SE\{2\}$.

The maneuver library for the F-16 is created using a closed-loop PID controller that tracks time-based reference trajectories for altitude, pitch angle, roll angle, yaw angle, and

speed. The library is created with nine kinds of maneuvers: change yaw rate to -2.9 deg/s, change yaw rate to 0, change yaw rate to 2.9 deg/s, change pitch to -45 deg, change pitch to 0, change pitch to 45 deg, change speed to 200 m/s, change speed to 250 m/s, and change speed to 300 m/s. Each maneuver is created starting from each of ten trim conditions that together cover the pitch angles, yaw rates, and speeds that the F-16 can reach using the available maneuvers. For example, the "change yaw rate to 2.9 deg/s" is created at ten different trim conditions, each of which results from executing one of the other maneuvers in the library. Therefore, the library contains a total of 90 maneuvers. Figure 3.10 shows a visualization of all the maneuvers, where the different initial conditions for each maneuver are the ten trim conditions.

To model uncertainty in the execution of each primitive, a set of realizations for each maneuver is created by adding a constant perturbation to the controller's pitch and yaw reference trajectories. These pitch and yaw perturbations are i.i.d. Gaussian random variables with zero mean and standard deviation of 0.11 deg. In creating the uncertain realizations for each maneuver, the uncertainty domain is sampled to ± 5 standard deviations using 11 samples each for the pitch and yaw dimensions (121 samples). This type of uncertainty in the realization of each maneuver may represent perturbed performance in execution of the maneuver caused by winds, aerodynamic trims, and control biases that cannot be eliminated through closed-loop control.

The planning task for the F-16 involves traveling 5 km downrange in minimum time while avoiding two altitude bands: 1,200m-1,375m from 3 km downrange and 1,800m-2,200m from 2 km downrange. The risk tolerance is 10% ($r = 0.1$). The vehicle starts at an altitude of 2,000m, necessitating a dive or climb to avoid the prohibited bands. The initial forward speed is 250 m/s. The heuristic estimate used is the $x-y$ distance to the goal, divided by 320 m/s. This speed is chosen to be larger than any speed the vehicle will achieve using the available maneuvers, and so the heuristic is guaranteed to be an underestimate of the true time-to-goal. A state space discretization with 5 meters, 1 m/s, 0.01 rad, and 0.01

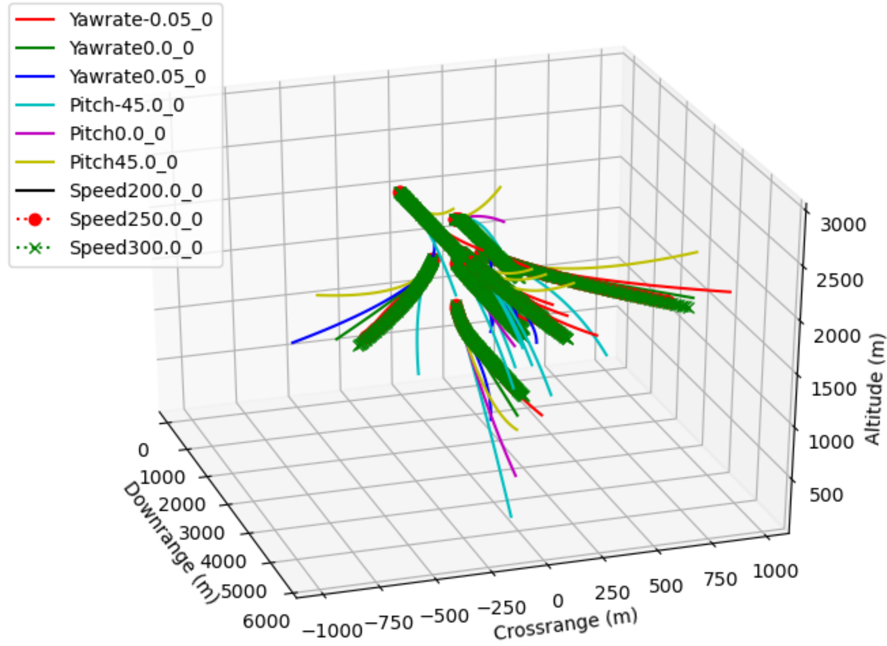

Figure 3.10: Nominal trajectories for F-16 maneuver library.

rad/s resolution is used for the position, velocity, orientation, and angular velocity states, respectively. Planning is conducted using both RAO* and AO* with and without presorting. In addition, non-presorted AO* is employed with 121, 81, then 49 samples at subsequent maneuver steps. A side view of the environment is provided in Figure 3.11, along with the nominal realization of the optimal maneuver sequence produced by AO*.

Results for the five planners are presented in Table Table 3.1. The results are broadly similar to the Dubins Car tasks – AO* does fewer AND expansions than RAO* and is thus faster. However, in this case one of the limitations of AO* is evident. For a given planning task it can perform (many) more OR expansions than RAO*. If OR expansions are sufficiently faster than AND expansions, AO* is still faster than RAO*. On this particular task, though, the *presorted* version of RAO* is actually slightly faster than the presorted version of AO*. Since collision checking is a bigger fraction of the total runtime of RAO*, presorting helps it more than it helps AO*. Of course, the induced heuristic is always available, while presorting can only be used if the obstacles are convex polytopes.

The Decaying Accuracy row reports the behavior of the AO* planner (without presort)

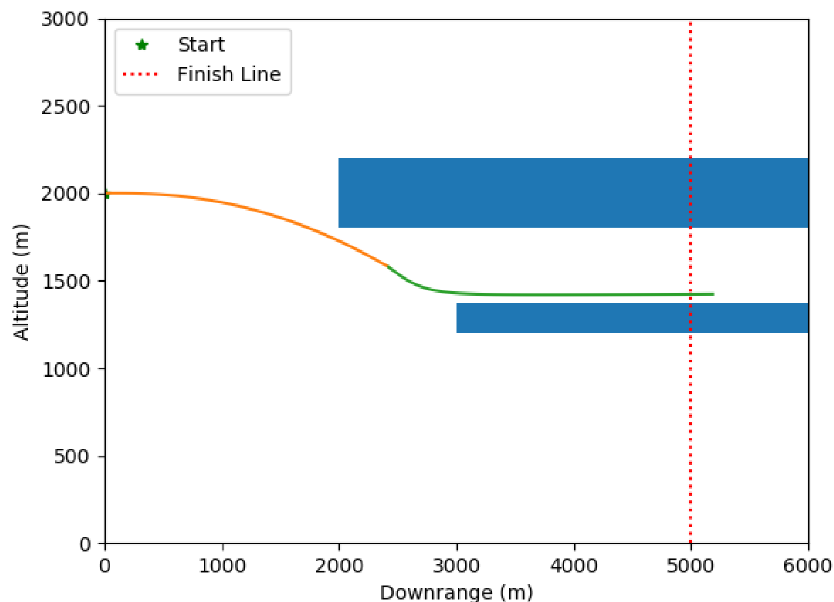

Figure 3.11: Side view of the environment for the F-16, with nominal realization of AO*'s solution.

when it is allowed to use only 81 samples for the second maneuver and 49 samples for the third (and subsequent) maneuvers (note that only depth 3 is actually reached during planning for this particular task). It returns the same initial maneuver (pitch 45 deg down) as the fixed-resolution planners, but does so with fewer expansions due to the reduced branching factor. The estimated cost-to-go and risk do differ from the values computed with the fixed resolution AO* variant – with fixed resolution the risk is estimated at 5.5% and the cost at 20.085 sec, while with decaying accuracy the estimates are 8.4% and 20.102 sec.

As with the Dubins Car task, a closed loop planning process was run 100 times with both decaying and fixed accuracy AO* planners. Under the fixed accuracy planner there were 4 collisions, while the decaying accuracy planner experienced only 3. Both planners were able to achieve the required level of safety (<10 collisions out of 100 trials).

3.3 Discussion

The induced heuristic devised in subsection 3.1.1 generally provides dramatic speed-ups over pure RAO*. Its sole shortcoming appears to be an increase in the necessary number of OR expansions, which in cases of very cheap collision checking can result in slight increases in runtime. Very cheap collision checking makes a setting "easy," though, so this seems a small price to pay. More concerning is that, even with decaying accuracy reducing the number of samples used for deep search, problems with a long planning horizon still require large amounts of planning time. This is the so-called "curse of dimensionality" in action; note that the worst-case complexity of R/AO* is $O(m^d n^d)$ for m the number of actions, n the number of Koopman samples, and d the search depth. The next two chapters discuss techniques (sparse integration and Monte Carlo Tree Search, respectively) that attempt to mitigate this problem.

Procedure 4 AO*(root)

```
1:  Q ← {}; insert root into Q with priority 0
2:  while not complete(root) and Q is not empty do
3:     N ← pop lowest priority from Q s.t. N is on current best path
4:     if N is OR then
5:        for all ACTION in ACTIONS(N) do
6:           H←induced_heuristic(ACTION)
7:           cost_to_go(ACTION)←H
8:           risk(ACTION)←0
9:           insert ACTION into Q with priority H+cost_to_come(N)
10:    else
11:       for all CHILD in CHILDREN(N) do
12:          H←estimate(CHILD)
13:          cost_to_go(CHILD)←H
14:          risk(CHILD)←collision_check(N,CHILD)
15:          if CHILD in goal region or risk(CHILD) is 1 then
16:             complete(CHILD)← TRUE
17:          else
18:             insert CHILD into Q with priority H+cost_to_come(CHILD)
19:    U_Q←{N}
20:    while U_Q is not empty do
21:       N←pop from U_Q
22:       if N is OR then
23:          A'← argmin_{A∈ACTIONS(N)} cost_to_go(A) s.t. risk(A)<risk_tol
24:          if A' is not NULL then
25:             best_action(N)←A'
26:             cost_to_go(N)←cost_to_go(A')
27:             risk(N)←risk(A')
28:             if all CHILDREN(N,A') are complete then
29:                complete(N)←True
30:          else
31:             best_action(N)← argmin_{A∈ACTIONS(N)} risk(A)
32:             cost_to_go(N)←cost_to_go(best_action(N))
33:             risk(N)←risk(best_action(N))
34:             complete(N)←True
35:       else
36:          cost_to_go(N)← ∑_{C∈CHILDREN(N)} P(N,C)[action_cost(N,C) +
             cost_to_go(C)]
37:          risk(N)← ∑_{C∈CHILDREN(N)} P(N,C)risk(C)]
38:       for all P in PARENTS(N) do
39:          if N in CHILDREN(P,best_action(P)) then
40:             append P to U_Q
41: return root
```

CHAPTER 4

SPARSE INTEGRATION SCHEMES FOR MOTION-PRIMITIVE PATH PLANNING

The expected value is the key calculation for path planning under uncertainty. The expected cost or reward of a policy determines if it is a good option or not; chance constraints (if present) are computed as the expected value of the indicator function for the relevant constraint.An expected value is computed as an integral over the uncertainty domain; this is why the dimension of the uncertainty domain contributes to the "curse of dimensionality" in the uncertain setting. This chapter makes the connection between high dimensional integration and hypergraph search based path planning precise and describes a way to construct an integration scheme (effectively, the placement and weighting of Koopman sample points) that efficiently computes the high dimensional integrals that arise during hypergraph search. The mixed integer linear programming approach to constructing such "sparse" integration schemes is applicable beyond this single use case, so it is compared directly with an existing approach for constructing equivalent sparse schemes and shown to generate rules with fewer sample points. The hypergraph specialized rules obtained from MILP are applied to several Dubins car tasks and shown to provide dramatic speed-ups with minor reductions in accuracy.

4.1 Sparse Schemes via Mixed Integer-Linear Programming

Recall the single-query chance constrained planning problem under parametric uncertainty (Equation 2.6):

$$\begin{aligned} \underset{\mu \in \mathbb{M}}{\operatorname{argmin}}\, E[J(\rho_\mu(t, t_0, x_0, s))] \text{ s.t.} \\ 1 - \prod_i \left(1 - E\left[C_i\left(\rho_\mu(t, t_0, x_0, s)\right)\right]\right) \leq r \end{aligned} \tag{4.1}$$

Restricted to (partial) policies constructed from motion primitives, this can be written as

$$\begin{aligned} \underset{\pi \in \Pi}{\operatorname{argmin}}\, Q^\pi(x_0, \pi(x_0)) \text{ s.t.} \\ R^\pi(x_0, \pi(x_0)) \leq r \end{aligned} \tag{4.2}$$

In (Equation 4.2), π is a (partial) policy, Π is the set of all (partial) policies, x_0 is the initial state, and r is the risk tolerance. $\mathbb{X}$ is the set of states and $\mathbb{A}$ the set of actions, so $Q^\pi : \mathbb{X} \times \mathbb{A} \to \mathbb{R}$ is the expected cost to go to the goal if a particular action is taken at the state and policy π is followed thereafter. R^π is the risk of constraint violation under the same circumstances. Let $\mathbb{S}$ be a set of unknown parameters on which the state transition function depends. Both Q^π and R^π therefore involve expected values and can in fact be

rewritten as iterated integrals over the uncertainty set $\mathbb{S}$:

$$
\begin{aligned}
Q^{\pi}(x,a) = & \qquad\qquad (4.3a)\\
&= \int_{\mathbb{S}} P(s_1)\left[c(x,a,s_1)\right. \\
&+\left. Q^{\pi}(x_1^{\pi}(s_1), a_1^{\pi}(s_1))\right] ds_1 \\
&= \int_{\mathbb{S}} P(s_1)\left\{c(x,a,s_1)\right. \qquad\qquad (4.3b)\\
&+ \int_{\mathbb{S}} P(s_2)\left[c(x_1^{\pi}(s_1), a_1^{\pi}(s_1), s_2)\right. \\
&+\left.\left. Q^{\pi}(x_2^{\pi}(s_{1:2}), \pi(x_2^{\pi}(s_{1:2})))\right] ds_2\right\} ds_1 \\
&= \int_{\mathbb{S}^d} \prod_{i=1}^{d} P(s_i)\left[\sum_{j=0}^{d-1} c(x_j^{\pi}(s_{1:j}), a_j^{\pi}(s_{1:j}), s_{j+1})\right. \\
&+\left. p(x_d^{\pi}(s_{1:d}))\right] ds_d \ldots ds_1
\end{aligned}
$$

where $c(x,a,s)$ is the cost of executing action a from state x when the uncertain parameter takes on value s. s_i is the value the uncertain parameter takes during execution of the ith action (above we assumed this is drawn independently from the parameter probability distribution $P(s)$ for each action). x_j^{π} is the state after j actions from policy π (and so depends on the sequence of uncertain parameter values $s_{1:j}$). Similarly, a_j^{π} is the $j+1^{th}$ action taken under the policy (depending on $s_{1:j}$). Finally, d is the maximum number of actions taken and $p(x)$ is the cost for terminating in state x. The nested integrals can be pulled all the way to the front because everything outside the kth nested integral is determined before the kth action is taken (and so does not depend on the kth parameter value). In particular, in (Equation 4.3b) $c(x,a,s_1)$ and $P(s_1)$ do not depend on s_2 and so the integral over ds_2 can be pulled to the front.

Similarly, for the risk:

$$
\begin{aligned}
R^{\pi}(x,a) = & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (4.4)\\
&= \int_{\mathbb{S}} P(s_1)[C(x,a,s_1) + R^{\pi}(x_1^{\pi}(s_1), a_1^{\pi}(s_1))]ds_1 \\
&= \int_{\mathbb{S}^d} \prod_{i=1}^{d} P(s_i) \sum_{j=0}^{d-1} C(x_j^{\pi}(s_{1:j}), a_j^{\pi}(s_{1:j}), s_j) ds_d ... ds_1
\end{aligned}
$$

Here, $C(x,a,s)$ is an indicator function which is 1 if executing action a from state x with parameter value s results in a constraint violation and 0 otherwise. The summation should be thought of as ending early if a constraint violation is encountered, so that the sum is always exactly 0 (if the full trajectory is safe) or 1 (if it is not).

(Equation 4.3a) and (Equation 4.4) show that the expected values necessary to evaluate a candidate partial policy can be viewed as high dimensional integrals of the form $\int_{\mathbb{S}^d} P(\vec{s})f(\vec{s})d\vec{s}$, where $P(\vec{s})$ is known ahead of time (since it is the joint distribution describing the sequence of uncertain parameters). Such an integral can be approximated by choosing sample points in the full product domain and taking an appropriately weighted sum:

$$
\int_{\mathbb{S}^d} P(\vec{s})f(\vec{s})d\vec{s} \approx \sum_{i=1}^{n} w_i f(\vec{s}_i) \qquad (4.5)
$$

A rule for selecting the sample points $\vec{s}_i$ and assigning the weights w_i is called a cubature rule and the sample points are the cubature nodes or cubature points.

A "dense" cubature rule can be obtained by applying a single-dimensional quadrature rule individually to the dimensions in the domain, then taking the Cartesian product of the points as the cubature nodes. The cubature weights would simply be the product of the quadrature weights. This is sometimes called the "product" or "tensor product" cubature rule [79]; while it is simple to conceive and implement, it is not an efficient rule. One measure of the accuracy of a cubature rule is the highest degree of monomials it can exactly

integrate [81]. From this perspective, the tensor product rule is highly redundant – it can exactly integrate some, but not all, monomials of higher degree than the univariate rules from which it was built [72]. Van Den Bos *et al.* identify this weakness of the tensor product rule in [75] and propose a scheme to improve efficiency by effectively removing certain cubature nodes. The scheme in [75] sets certain weights to 0 and appropriately adjusts the remaining weights to maintain the ability to exactly integrate all monomials of a specified degree (while reducing the total number of cubature nodes). However, the approach is suboptimal in that it does not guarantee that the resulting rule has the minimum possible number of retained nodes. Obtaining the minimal rule is a non-convex optimization problem that may be defined as:

$$\begin{aligned} \min_{\vec{w}} \|\vec{w}\|_0 \text{ s.t.} \\ G\vec{w} &= \vec{m} \\ \vec{w} &\geq 0 \end{aligned} \tag{4.6}$$

where $\vec{w}$ is the vector of weights of the cubature nodes and $\vec{m}$ is a vector of the exact integrals of all monomials that should be integrated exactly. G is a special Vandermonde matrix constructed from the locations of the cubature nodes and the desired set of monomials. Each column of G corresponds to one of the cubature nodes, while each row corresponds to one of the monomials. The (i, j) entry of G is the ith monomial evaluated at the jth cubature node. Thus, the dot product of a row of G and $\vec{w}$ is the estimate of the integral of that monomial. This problem is almost a Linear Program, but the objective function (the 0-norm of the weight vector) is non-convex. The positivity of weights is technically optional, but cubature rules with positive weights have desirable properties and so this is a common restriction [98, 80, 75, 82].

Introducing binary indicator variables a_i that are 1 if node n_i is to be retained (and so

weight $w_i \neq 0$) converts the problem into a Mixed Integer-Linear Program (MILP):

$$\begin{aligned}
\min_{\vec{w},\vec{a}} \sum_i a_i \text{ s.t.} & \\
G\vec{w} &= \vec{m} \\
\vec{w} &\geq 0 \\
a_i &\geq w_i \\
a_i &\in \{0, 1\}
\end{aligned} \tag{4.7}$$

4.2 Technical Considerations

(Equation 4.7) can be used to "sparsify" a generic integration rule while maintaining a prescribed level of polynomial exactness. However, the use case of chance-constrained path planning via forward search creates complexities that impose additional requirements on a sparse integration scheme.

In the planning problem considered here, the state space of the system is restricted to evolving on an AND/OR hypergraph where OR choices correspond to choosing an action to take and the AND outcomes are the outcomes of the action for sampled parameter values of known probability (for additional details, see [99]). A state corresponds to a particular cubature node based on the sequence of parameter values that led to it, and the child states for an AND expansion are those whose parameter values are branches of the parent state's parameter sequence. Figure 4.1 shows a sample AND/OR search. Lettered vertices are states (the OR nodes of the graph); vertices with both letters and numbers are the actions (the AND nodes of the graph) available at the lettered state. The sequence in chevrons above each state is the sequence of parameter values that led to that state (i.e., the point in $\mathbb{S}^d$, which specifies the relevant weight in the cubature scheme). Thus, the initial state A is equivalent to a point in $\mathbb{S}^0$, while state H corresponds to a point in $\mathbb{S}^2$. If a tensor product rule were used, the weight for H would be the probability of transitioning from A to B

under action A1 multiplied by the probability of transitioning from B to H under action B2. In a sparse rule, the weight may be different.

Forward search is incremental. Therefore, it is necessary to be able to compute values such as the heuristic estimate of the cost or reward of action A1, which requires integrating over only its immediate children (i.e., states B and C). This requires a cubature rule over $\mathbb{S}^1$. However, to support a forward search involving up to d actions in sequence, a series of cubature rules are required for $\mathbb{S}^1$, $\mathbb{S}^2$,...,$\mathbb{S}^d$. To keep track of this, nodes, weights, etc. will be sub-scripted both with the index of the rule they belong to (i.e. how many times the parameter is sampled, which is smaller than the dimension of the rule if $\mathbb{S}$ is multi-dimensional) and their index in that particular rule. The rule for $\mathbb{S}^d$ is called the rule at depth d. It is not enough to simply have a series of rules of appropriate dimension, though. These cubature rules must have special structure to allow the rules of different dimension to operate together and to preserve the optimality guarantees of the search algorithm. These structural requirements (and mechanisms for imposing them as MILP constraints) are described in the following.

4.2.1 Required Properties of Cubature Rules

The first property required for the series of integration rules derived in this work is that the higher dimensional rules be "branches" of the lower dimensional rules. This is similar to the nesting property discussed in the numerical integration literature [100, 66, 75], but it is not equivalent. Nesting refers to pairs of rules over the same integration domain, and requires that the rule with more sample points (i.e., the more accurate rule) include all the sample points of the smaller rule. This is useful for error estimation, as one can compare the results of the two rules without requiring any additional function evaluations. Branching instead refers to pairs of rules over different integration domains, and requires that the higher dimensional rule only include points whose truncation is in the lower dimensional rule(s). Let $n_{d,i} = \langle s_1, s_2, ..., s_{d-1}, s_d \rangle$ be the ith cubature node in the rule at depth $d > 1$, where

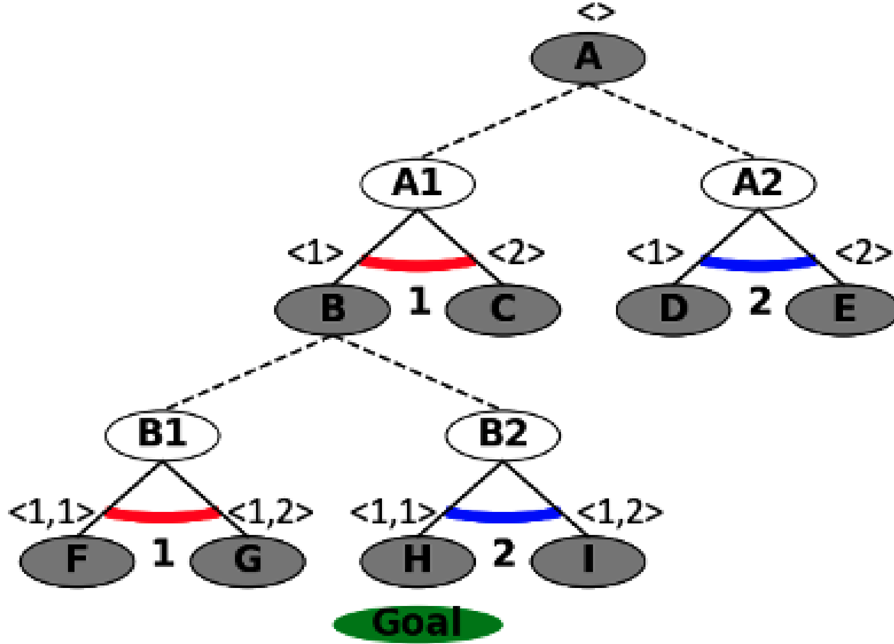

Figure 4.1:]
And/Or graph with parameter sequences (i.e. which cubature nodes correspond to the state) showing in chevrons.

$s_j \in \mathbb{S}$ is the parameter setting for that node for action # j. $n_{d,i}$ satisfies branching if and only if $\exists n_{d-1,k} = \langle s_1, s_2, ..., s_{d-1} \rangle$ in the depth $d-1$ rule. Then, $n_{d,i}$ is a branch of $n_{d-1,k}$. Let nodes(d) be the set of indices for all the cubature nodes in the rule at depth d. Given a cubature node $n_{d,i}$, branches$(n_{d,i}) = \{l \in$ nodes$(d+1) : n_{d+1,l}$ is a branch of $n_{d,i}\}$.

As an example, consider a rule at depth 3 and let the uncertain parameter $s \in \mathbb{S} = \{1, 2\}$ (so in this case dimension and depth are equal). The point $\langle 1, 1, 2 \rangle$ is only permitted in the depth 3 rule if the point $\langle 1, 1 \rangle$ is included in the depth 2 rule. Including $\langle 1, 1 \rangle$ in the depth 2 rule means that the integrand is evaluated for the case where the parameter takes on 1 for the first action and 1 for the second action. If $\langle 1, 1, 2 \rangle$ is included, the integrand will need to be evaluated for the case where the parameter then takes on the value 2 at depth 3. Requiring that $\langle 1, 1 \rangle$ be included in the depth 2 rule ensures that the distribution of vehicle states under the current policy after $\langle 1, 1 \rangle$ is already available, so that computing the evolution under $\langle 1, 1, 2 \rangle$ requires simulating only one layer of actions (with parameter value 2). If branching were not required, the depth 3 rule might require a state distribution that can

only be obtained by going back to the root and recomputing for a whole new sequence of parameter values. In Figure 4.1, a failure to enforce branching might result in having to evaluate state H without having evaluated state B. The branching constraint, which requires that if a branch of node $n_{d,i}$ has non-zero weight so too must $n_{d,i}$, can be imposed on the indicator variables:

$$\forall i \in \text{nodes}(d) \forall j \in \text{branches}(n_{d,i}) a_{d,i} \geq a_{d+1,j} \tag{4.8}$$

where $a_{m,n}$ is the indicator for node n at depth m.

The second condition that must be enforced is that the sum of the weights of all the branches of node $n_{d,k}$ must exceed the weight of that node:

$$w_{d,k} \leq \sum_{j \in \text{branches}(n_{d,k})} w_{d+1,j} \tag{4.9}$$

where $w_{d,k}$ is the weight corresponding to node k in the depth d rule. If the heuristic h used in the search algorithm is consistent (i.e. the heuristic evaluated at state A is never larger than the heuristic evaluated at a neighboring state B plus the cost to go from A to B), then $\forall j \in$branches$(n_{d,k})$

$$g(n_{d,k}) + h(n_{d,k}) \leq g(n_{d+1,j}) + h(n_{d+1,j}) \tag{4.10}$$

where $g(n)$ is the cost to come to node n and $h(n)$ is the heuristic evaluated at node n. Combined, (Equation 4.9) and (Equation 4.10) imply that the contribution of a node to the total cost is no larger than that of all its branches:

$$\begin{aligned} & w_{d,k}(g(n_{d,k}) + h(n_{d,k})) \\ & \leq \sum_{j \in \text{branches}(n_{d,k})} w_{d+1,j}(g(n_{d+1,j}) + h(n_{d+1,j})) \end{aligned} \tag{4.11}$$

Thus, since expanding a node during the search is equivalent to replacing the contribution of a node with the estimated expected value of the contributions of all its children, this will not decrease the estimated cost:

$$\begin{aligned} E_d = & \sum_{i \in \text{nodes}(d)} w_{d,i}(g(n_{d,i}) + h(n_{d,i})) \\ \leq & \sum_{i \neq k} w_{d,i}(g(n_{d,i}) + h(n_{d,i})) \\ + & \sum_{j \in \text{branches}(n_{d,k})} w_{d+1,j}(g(n_{d+1,j}) + h(n_{d+1,j})) \end{aligned} \tag{4.12}$$

(Equation 4.12) implies that the final estimate of the cost or risk of a policy is no smaller than any of the estimates obtained while some paths haven't been expanded all the way to a terminal state. Forward search algorithms use this property to eliminate suboptimal policies without full evaluation: once the current policy is completely evaluated, any policy with a cost estimate larger than that of the current policy is definitely inferior and can be discarded. Policy risk estimation can be handled as a cost estimate where the heuristic in use is simply 0; this is consistent (which implies it is also admissible) and thus the previous reasoning applies. Then, one can simply discard a policy as soon as the risk estimate exceeds the risk tolerance.

A third requirement must be enforced on the cubature rule for the optimization to actually return a set of rules with a minimal total number of nodes. A weight's indicator variable must only be 1 if the weight is non-zero, since this indicator is used to enforce the branching constraint. In the single rule case of (Equation 4.7), this is guaranteed simply by minimizing the sum of the indicator variables. The optimal solution will never include a non-zero indicator for a zero weight. With multiple rules and constraints imposed that involve weights and indicators belonging to different rules, however, it is possible to choose to set an indicator to 1 even when the weight is zero. Doing so can "permit" zeroing out weights (and thus indicators) in other rules, resulting in an overall decrease in the objective. To prevent this, it is necessary to impose constraints equivalent to $a_{i,j} = 0 \iff w_{i,j} = 0$,

where $a_{i,j}$ is the indicator for weight j belonging to the rule at depth i. In the context of mixed-integer linear programming, such a constraint may be approximated as:

$$a_{i,j} \leq w_{i,j} + 1 - \epsilon \tag{4.13}$$

where ϵ is a small constant. This means the indicator can only be non-zero if the weight is larger than ϵ and so, in addition to the intended purpose, this constraint also prevents solutions where any weight is between 0 and ϵ.

Finally, as noted above the weights are required to be positive. While this is common practice when deriving cubature rules, it is particularly important here as the weights represent a probability distribution (which is necessarily positive). Moreover, positivity of the weights ensures that an estimate that uses an admissible heuristic in place of the true cost-to-go is still an underestimate. This preserves consistency and allows the search algorithm to perform the pruning behavior described earlier.

4.2.2 Problem Modifications to Improve Numerical Performance

Imposing the conditions of Section subsection 4.2.1 for rules of depth $1, ..., d$ yields the following mixed-integer linear program:

$$\begin{aligned}
\min_{\vec{w}_i, \vec{a}_i} \quad & \sum_{1 \leq i \leq d} \sum_{a_{i,j} \in \vec{a}_i} a_{i,j} \text{ s.t. } \forall i \\
& G_i \vec{w}_i = \vec{m}_i \\
\forall j \in \text{nodes}(i) : \quad & \sum_{k \in \text{branches}(n_{i,j})} w_{i+1,k} \geq w_{i,j} \\
& \vec{w}_i \geq 0 \\
& \vec{a}_i \geq \vec{w}_i \\
& \vec{w}_i + 1 - \epsilon \geq \vec{a}_i \\
\forall j \in \text{nodes}(i) \forall k \in \text{branches}(n_{i,j}) : \quad & a_{i,j} \geq a_{i+1,k} \\
& a_{i,j} \in \{0, 1\}
\end{aligned} \tag{4.14}$$

where for the rule at depth i $\vec{w}_i$ is the vector of weights, $\vec{a}_i$ is the vector of indicators, G_i is the Vandermonde matrix, and $\vec{m}_i$ is the vector of exact monomial integrals. This is a challenging optimization problem for several reasons. First, it involves a very large number of variables. Suppose a 10 dimension rule with compatible 2-dimensional (2D), 4D, 6D, and 8D rules is desired. This would be a depth 5 set of rules with a 2 dimensional $\mathbb{S}$. If the initial dense rules are obtained as the Cartesian product of a Gaussian quadrature rule for degree 13 polynomials (7 points per dimension), the optimization problem involves $\sum_{i=1}^{5} 7^{2i} = 288,360,149$ weights and the same number of indicators. Second, if the polynomial degree is large the range of values in the G matrices will be very large. This leads to poor conditioning of the constraint matrix. Third, for high dimensional rules the optimal weight values will be very small. This results in issues with numerical precision and is particularly challenging when combined with the need to approximate a strict inequality with an ϵ tolerance as in Equation 4.13.

The first challenge (the number of variables) can be mitigated by requiring that the integration scheme exhibit symmetry. As pointed out in [75], the weights in an integration scheme are generally not unique. Instead, when integrating against some underlying probability distribution the weights should match the symmetries of the distribution. Thus, the optimization can address each symmetric set of nodes as one unit, with a single weight, single indicator, and a count specifying how many nodes are eliminated if this weight is set to zero.

If sufficient forms of symmetry are available, this leads to a dramatic reduction in the number of variables in the optimization. Consider once more a 10 dimensional scheme with compatible 2D, 4D, 6D, and 8D rules built from a seven point 1D rule. If the underlying 1D distribution is symmetric about its mean, the weights can be symmetric about that value in every single dimension. In effect, only 4 points are used and the total number of unique weights is only $\sum_{i=1}^{5} 4^{2i} = 1,118,480$. Additional symmetry can reduce this further. If the order of the dimensions does not matter, then the weights for the points

$\langle 1, 2 \rangle$ and $\langle 2, 1 \rangle$ should be the same. This permutation symmetry further reduces the number of unique weights to only 20,348. Additionally, some of the monomials in the original problem can exhibit symmetry and so their rows can be eliminated from the G matrix. If mirror symmetry about the axes is imposed, then monomials containing odd powers in the mirror symmetric axes will automatically integrate to 0 and can be dropped. If permutation symmetry is applied, only a single permutation of the exponents is necessary – e.g., $x^a y^b$ is redundant with $x^b y^a$. Applying symmetry to (Equation 4.14) yields the following optimization problem:

$$
\begin{aligned}
\min_{\widetilde{w}_i, \widetilde{a}_i} \sum_{1 \le i \le d} \vec{c}_i^T \widetilde{a}_i \text{ s.t. } \forall i & \\
\widetilde{G}_i \widetilde{w}_i &= \vec{m}_i \\
\widetilde{w}_i &\geq 0 \\
\widetilde{a}_i &\geq \widetilde{w}_i \\
\widetilde{w}_i + 1 - \epsilon &\geq \widetilde{a}_i \\
\forall j \in \text{symmetric}(i) : & \\
\sum_{k \in \text{branches}(n_{i,j})} \widetilde{w}_{i+1,k} &\geq \widetilde{w}_{i,j} \\
\widetilde{a}_{i,j} &\in \{0, 1\} \\
\forall k \in \text{branches}(n_{i,j}) \quad \widetilde{a}_{i,j} &\geq \widetilde{a}_{i+1,k}
\end{aligned}
\tag{4.15}
$$

where $\widetilde{w}_i, \widetilde{a}_i$ contain only the unique entries and $\vec{c}_i$ is a vector counting how many nodes share that weight and indicator. $\widetilde{w}_{i,j}$ and $\widetilde{a}_{i,j}$ are the weight and indicator respectively for node j at depth i, so multiple j could point to the same value. symmetric(i) is the set of indices of nodes at depth i containing only one index for each such set of nodes with the same weight. $\widetilde{G}_i$ is the modified Vandermonde matrix where rows for redundant monomials have been removed and every column corresponding to a given unique weight has been summed together. Column j of $\widetilde{G}_i$, $\widetilde{g}_{i,j}$, is computed from the columns $g_{i,k}$ of the

original Vandermonde matrix G_i as:

$$\widetilde{g}_{i,j} = \sum_{k \in SS(i,j)} g_{i,k} \tag{4.16}$$

where $SS(i,j)$ is the jth symmetric set of nodes at depth i. Thus, $\forall k \in SS(i,j)$, $\widetilde{w}_{i,k}$ is the jth entry in $\widetilde{w}_i$.

The condition number of the constraint matrix can be improved by independently scaling each row of the G matrices (equivalent to premultiplying by a diagonal matrix M). In particular, dividing by the geometric mean of the maximum entry of the row and the smallest positive entry proved effective in the tests described below. Recall that each column corresponds to a different node, while each row is a different monomial. The first row of G is generally all ones as it corresponds to the zero monomial, while later rows are the result of very high dimensional monomials and so have very large values (or very small if the node has entries <1). If magnitudes of the node points are all similar, much of the variation in the entries occurs between rows rather than within them and can be canceled out by this scaling.

The numerical difficulties associated with very small optimal weights can also be mitigated by scaling. The "true" weights can be multiplied by some (large) constant factor for optimization purposes (this factor can be different for each depth) and the true values recovered in post-processing. Dividing by the arithmetic mean of the minimum and maximum weight in the original Cartesian product rule for a given depth proved effective in practice. Applying both matrix and weight scalings affects the definition of several constraints as

shown in (Equation 4.17).

$$
\begin{aligned}
\min_{\widetilde{x}_i,\widetilde{a}_i} \sum_{1\le i\le d} \vec{c}_i^T \widetilde{a}_i \text{ s.t. } \forall i \qquad\qquad & \qquad (4.17)\\
M_i \widetilde{G}_i \widetilde{x}_i &= M_i \vec{m}_i / s_i \\
\widetilde{x}_i &\ge 0 \\
\widetilde{a}_i / s_i &\ge \widetilde{x}_i \\
\widetilde{x}_i + 1 - \epsilon &\ge \widetilde{a}_i \\
\forall j \in \text{symmetric}(i): & \\
\sum_{k\in\text{branches}(n_{i,j})} s_{i+1}\widetilde{x}_{i+1,k} &\ge s_i \widetilde{x}_{i,j} \\
\widetilde{a}_{i,j} &\in \{0,1\} \\
\forall k \in \text{branches}(n_{i,j}) \qquad \widetilde{a}_{i,j} &\ge \widetilde{a}_{i+1,k}
\end{aligned}
$$

Here, w has been replaced by x to indicate that it is not actually a weight but instead just an optimization variable. M_i is the matrix applying scaling to the G matrices to help with conditioning, while s_i is the weight scale factor for depth i ($\widetilde{w}_i = s_i \widetilde{x}_i$). Note that the scaling is not applied to the "strict" inequality – now w_i may be between 0 and ϵ provided that x_i is not.

Finally, the optimization may be simplified due to the redundancy of some of the exact integration constraints (certain rows of the G matrix) when combined with the branch sum constraint (from (Equation 4.9)). Let the subscript l denote weights and nodes belonging to a lower dimensional rule and h those of a higher dimensional rule. Let $m(x)$ be a monomial that evaluates the same on a node n_l and any branch of that node n_h (i.e., a monomial with zero exponents in the dimensions appearing in h but not l, such as $x^2 y^0$ for $l = 1$ and

$h = 2$). Then, the exact integration and sum constraints are collectively

$$\sum_{j \in \text{nodes}(l)} w_{l,j} m(n_{l,j}) = e \tag{4.18}$$

$$\sum_{k \in \text{nodes}(h)} w_{h,k} m(n_{h,k}) = \frac{V_h}{V_l} e \tag{4.19}$$

$$w_{l,j} \leq \sum_{p \in \text{branches}(n_{l,j})} w_{h,p} \tag{4.20}$$

where e is the exact integral of m over the domain of the l rule, V_l is the sum of the weights of the l rule, and V_h is the sum of the weights of the h rule. When dealing with probabilities, V_l and V_h are both 1. Now, if all the nodes in the h rule are branches of nodes in the l rule, $\forall n_{h,k} \exists n_{l,j} : m(n_{h,k}) = m(n_{l,j})$. So for expected value integrals, (Equation 4.19) can be rewritten as:

$$\begin{aligned} \sum_{k \in \text{nodes}(h)} w_{h,k} m(n_{h,k}) &= \\ &= \sum_{j \in \text{nodes}(l)} \sum_{k \in \text{branches}(n_{l,j})} w_{h,k} m(n_{l,j}) \\ &= \sum_{j \in \text{nodes}(l)} w_{l,j} m(n_{l,j}) \end{aligned} \tag{4.21}$$

This is satisfied if (Equation 4.20) is replaced with equality. Next, suppose $\exists j : w_{l,j} < \sum_{k \in \text{branches}(n_{l,j})} w_{h,k}$; then for that j $\sum_{k \in \text{branches}(n_{l,j})} w_{h,k} m(n_{l,j}) > w_{l,j} m(n_{l,j})$. However, no j can satisfy $\sum_{k \in \text{branches}(n_{l,j})} w_{h,k} m(n_{l,j}) < w_{l,j} m(n_{l,j})$, so if the branched weight sum is larger than the parent weight for some j (i.e., if (Equation 4.20) is not an equality), (Equation 4.21) cannot be satisfied. Thus, if (Equation 4.18), (Equation 4.19), and the branching constraint (Equation 4.8) are imposed, the constraint in (Equation 4.20) is redundant. Alternatively, (Equation 4.20) (originally presented as (Equation 4.9)) can be made an equality and (Equation 4.19) can be removed. The latter approach, which provides a means to completely drop many low-dimensional monomials from the exact integration constraint in the high depth rules, resulted in better numerical behavior in the tests described below

and so is recommended.

4.3 Results

4.3.1 Effectiveness of MILP

van den Bos et al.'s approach cannot impose the properties of subsection 4.2.1, so it cannot directly construct sparse schemes for chance constrained path planning. However, van den Bos et al. provide a table of the number of nodes retained by their implementation for a variety of dimensions D and degrees of polynomial exactness K (this is Table 1 in [75]). It is not directly stated in the reference what one dimensional scheme was used to generate this table; the present authors were able to replicate the reported numbers of retained nodes for the Smolyak schemes by using a Gauss-Legendre rule of degree K (i.e. with $\frac{K+1}{2}$ points). The MILP of (Equation 4.7) can construct comparable sparse rules (for positive weights).

The number of nodes retained by the van den Bos approach depends on the order in which null vectors are processed, so the numbers presented in the reference were not exactly replicated in general. Table 4.1 reports the smallest schemes obtained from three ordering strategies: 1. prioritize the null vector with the smallest α (see van den Bos et al. Algorithm 2) 2. prioritize the null vector that eliminates the most nodes (the greedy strategy) 3. ordered based on singular value. For option 3, since the null vectors were obtained as the complex conjugate of the right singular vectors corresponding to the smallest singular values, it was possible to sort by corresponding singular value. Both increasing and decreasing orders were tested. The selection criteria used (see step 9 of Algorithm 2 in van den Bos et al.) was to choose the option with the most symmetric nodes. In the cases of $(D = 7, K = 7)$, $(D = 7, K = 11)$, and $(D = 10, K = 7)$ the present authors were unable to replicate the results of [75]. In the remaining cases, the author's implementation yielded similar (often better) rules to those reported. The MILP approach resulted in universally smaller rules, with the advantage increasing for higher dimensions.

Table 4.1: Comparison of rule size for van den Bos and MILP.

D	K	van den Bos	replication	MILP
5	5	113	113	113
	7	544	544	544
	9	1313	1115	963
	11	4096	4096	3776
	13	6005	6133	4325
7	5	689	689	689
	9	19717	18915	11105
	13	158709	172217	85867
10	5	13461	13185	9088
	9	1368449	1276193	767533
	11	8284617	12165120	7351296
	13	26598325	26441353	8809877

4.3.2 Path Planning Example

Recall the Dubins Car-like vehicle subject to wind uncertainty from (Equation 2.7):

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} v\cos\theta + w_x \\ v\sin\theta + w_y \\ \omega \\ u \end{bmatrix} \tag{4.22}$$

v is the constant (10 m/s in the examples below) airspeed, and u is the control input. x, y are the position, θ is the heading, and ω the angular velocity. The winds $w_x, w_y \sim N(0, 0.2) \times N(0, 0.2)$ m/s are assumed to hold constant for the duration of a maneuver, and then are redrawn from the parameter distribution for the next maneuver. The expected value of a function of the policy of length n (i.e., cost or risk) is thus an integral over $\mathbb{R}^{2n}$ with a Gaussian weighting function. Cubature schemes for dimension $2, 4, ..., 2n$ are needed. This problem exhibits mirror symmetry across the axes and permutation symmetry between any included dimensions. As the underlying PDF is Gaussian, the initial rule is constructed from a 7 point Gauss-Hermite rule across each dimension. Table 4.2 shows, for a polynomial degree of 10, the sizes of the dense and optimal rules, as well as the run-

time needed to solve the MILP on a 6 core Intel® Xeon® CPU E5-1650 v2 @ 3.50GHz using Gurobi 9.1.2. FeasibilityTol, OptimalityTol, and IntegralityTol were all set to 1e-9, NumericFocus was set to 3, and IntegralityFocus was set to 1. An ϵ of 1e-9 was used. Gurobi was accessed through the PICOS 2.2 interface. As shown in Table Table 4.2, the number of nodes in the optimal sparse rule is significantly less than the dense rule at the same dimension, and the degree of sparsity increases substantially as the dimension of the problem increases. The presented times are for solving the MILP using Gurobi and do not include setup or post-processing, which can be significant. Solution times are linear in the number of nodes in the dense scheme.

For efficiency reasons, state merging as described in [99] is applied without modification. This means that sometimes, a state will have a child whose realization sequence is not an extension of the parent's. This sacrifices exact polynomial integration in favor of the significant performance benefits of state merging. Since it is rare for the cost or constraint functions to truly be polynomial functions of the uncertain parameter, this is not considered to be a major sacrifice.

Table 4.2: Number of Nodes in Dense and Optimal Sparse Cubature Rules and Solution Time.

# actions	Max dim	Dense	Optimal	Time (s)
1	2	49	40	0.052
2	4	$2,450$	505	0.10
3	6	1.201×10^5	$3,182$	0.53
4	8	5.885×10^6	3.291×10^4	4.0
5	10	2.884×10^8	2.757×10^5	4.4
6	12	1.413×10^{10}	3.119×10^6	12
7	14	6.924×10^{11}	3.264×10^7	270

Table 4.3 shows the solution times and search complexity measurements for a path planning example with a goal position of 7m downrange, 3m crossrange and a risk tolerance of 20%. The AND and OR columns are the numbers of AND/OR vertices expanded. This task requires 4 actions to span the path from start to goal, necessitating an 8 dimensional integral. The environment in shown in Figure 4.2. The planning problem was solved

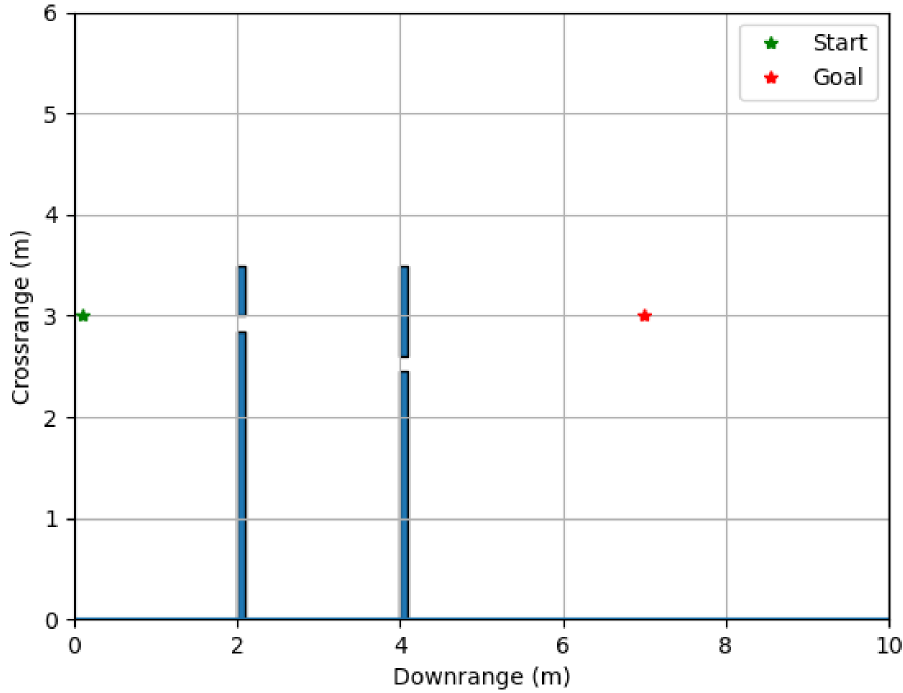

Figure 4.2: Test environment for the Dubins Car, with goal position marked.

using the AO* with induced heuristic from chapter 3, where either the dense or optimal sparse rule was used by the planner for integration. The actual cost (distance traveled) and risk of the returned policy is estimated by averaging the results of 1,000 Monte Carlo simulations of the policy. As shown in Table 4.3, all the tested sparse rules provide dramatic runtime improvements compared to the dense rule, but it is advantageous to use a sparse rule designed for the maximum search depth that will occur in the actual problem. While a rule obtained for maximum dimension 14 does involve constructing rules for 2D-12D, those lower-dimensional rules are *less sparse* than would be obtained if the maximum dimension were set lower. An unnecessarily deep rule thus results in increased runtime as shown in Table 4.3. It is possible that using an excessively deep rule can improve accuracy, as shown by the better risk estimates in Table Table 4.3. However, this is not guaranteed – for instance, the deeper rules had slightly less accurate cost estimates in this scenario. The risk was overall estimated less accurately than the cost because the constraint indicator functions that appear in the risk integral are not smooth and so need very high degree

polynomials to approximate accurately.

A final example illustrates the increasing runtime benefits of sparsity at greater search depths. Two additional planning tasks were executed using AO* requiring 5 actions (10D) and 6 actions (12D), respectively. The environment was the same as in Figure 4.2 with the goal moved to 8 and 12 meters. Table 4.4 shows the performance of the dense rule compared with that of the optimal sparse rule. In the 5-action case, the sparse rule enables an 83% reduction in runtime and a 90% reduction in the number of states (vertices) considered by the planner compared to using the dense rule. Importantly, the 6-action task is intractable using the dense rule due to memory requirements that exceeded the 64 GB capacity of the computer, but using the optimal sparse rule a solution is obtained in less than fifteen minutes.

Table 4.3: AO* Planning Statistics for Environment Shown in Figure 4.2.

Scheme	Time (s)	Cost Estimate	True Cost	Risk Estimate	True Risk	Vertices	Merged	OR	AND	Collision Checks
Dense	233	7.107	7.087	8.601%	18.4%	52,097	14,348	850	1,356	265,776
8D Sparse	46.1	7.083	7.084	7.886%	18.5%	6,787	1,130	340	705	31,664
10D Sparse	51.1	7.079	7.084	8.242%	18.5%	7,822	1,231	352	698	36,208
12D Sparse	78.5	7.080	7.084	9.098%	18.5%	9,687	1,866	412	754	46,208

Table 4.4: Planning Statistics for Tasks of Increasing Search Depth.

Task	Time (s)	Cost Estimate	True Cost	Risk Estimate	True Risk	Vertices	Merged	OR	AND	Collision Checks
8m (10D)										
Dense	551	8.081	8.061	8.601%	18.4%	80,137	43,736	1,646	2,528	495,488
Sparse	93	8.054	8.058	8.242%	18.5%	8,049	2,597	584	1,029	42,580
12m (12D)										
Sparse	857	12.02	12.03	9.098%	18.5%	9,649	17,326	2,184	3,676	107,896

4.4 Discussion

The MILP formulation for sparse scheme construction is notably general; in addition to the specialized rules for path planning, it can construct generic sparse rules for particular polynomial exactness and the resulting rules are smaller than with the existing van den Bos approach (indeed, they are theoretically the sparsest possible such rules for a given starting dense rule and required polynomials). The expressive power of MILP constraints, as showcased by the path planning application, offers the potential for specialized rules in other applications. For problems in which such schemes are practical, the numerical results shown the enormous benefits available. For path planning, though, the tractability of sparse integration scheme construction is dependent on the symmetry described in subsection 4.2.2. For even deeper search, or for problems that lack sufficient symmetry, another approach to mitigate the curse of dimensionality is needed. The next chapter turns to the classic Monte Carlo technique and compares Monte Carlo Tree Search against AO* with the induced heuristic.

CHAPTER 5
MONTE CARLO ALGORITHMS

5.1 Monte Carlo Tree Search

MDPs under uncertainty are structurally similar to adversarial games; in the game perspective, instead of an environment that chooses randomly to make the outcome of actions uncertain, there is an adversary which chooses according to some policy. In the zero-sum case, the adversary acts to minimize your reward and so will always choose the worst possible outcome (for you). This is equivalent to constructing a robust plan from the MDP point of view. Monte Carlo Tree Search (MCTS) is an algorithm introduced by Kocsis and Szepesvari [83] that has had great success in the field of games, achieving state of the art performance in Go [68, 84], Solitaire [85], and chess [84] among others tasks. It is thus natural to apply it to MDPs under uncertainty, as was done as early as [83]. A suitable planner for non-chance constrained uncertain path planning with motion primitives can be obtained by calling Procedure 5 repeatedly on the current state until a fixed time has elapsed. MCTS generally takes a reward maximization perspective; to support the cost minimization objectives considered throughout this work, one can simply negate the running cost (typically path length) and terminal penalty. Line 14 is the classical UCT algorithm for choosing an action to sample in MCTS and the constant $c = \sqrt{2}$ unless otherwise noted. Line 6 offers scope for heuristic information to contribute to the planner; in the results that follow, the default policy selects the action whose nominal exit state has the smallest sum of cost to come and heuristic cost to go to the goal. Furthermore, state construction in Line 18 can benefit from the cell-based state merging of subsection 3.1.2.

Procedure 5 MCTS Sample(history)

1: **if** history violates a constraint **or** history is terminal **then**
2: set history's value to the terminal reward
3: add 1 to the history's sample count
4: **return**
5: **if** history's sample count is 0 **then**
6: set next action to that of the default policy for history
7: **else if** any actions available at history have never been sampled **then**
8: set next action to the first unsampled action
9: **else**
10: best_score= $-\infty$
11: **for all** action available at history **do**
12: exploitation=estimated value of action at history
13: exploration= $\sqrt{\frac{\log \text{history's sample count}}{\text{sample count of action at history}}}$
14: score= exploitation $+ c *$ exploration
15: **if** score>best_score **then**
16: set next action to action
17: set best_score to score
18: draw a parameter value from the proposal distribution and use this to get the child of next action, constructing a new one if the parameter value has never been drawn before
19: call MCTS Sample on the newly constructed child
20: **for all** action available at history **do**
21: set sample count of action at history to the total number of children of action at history (if a child was drawn multiple times, each one counts as a sample)
22: **if** action has been sampled at least once **then**
23: set estimated value of action at history to the average value of the children plus the average value of the immediate reward for the action (if any)
24: **else**
25: set estimated value of action at history to $-\infty$
26: set history's sample count to the sum of the sample counts of all the actions
27: set history's best action to the action with maximum estimated value
28: set history's value to the value of the best action
29: **return**

5.2 Vulcan

Procedure 6 Vulcan(start state s_0)

1: set the estimated value of all actions $\tilde{Q}(h, a)$, the sample counts N_h, and the action sample counts $N_{h,a}$ to 0
2: **while** run time $\leq$ limit **do**
3: call Sample on the start state s_0
4: **if** Sample returned **false then**
5: **return** no solution
6: call Cleanup on the start state s_0
7: **return** the estimate of the value of the best action at the start state and the estimated policy

Vulcan is a modification of MCTS that can enforce chance constraints. It was proposed by Ayton and Williams in [69]. Like MCTS, Vulcan requires an exploration-exploitation tradeoff coefficient, which is set to $\sqrt{2}$ in the following. It also needs a default policy, which is again selecting the action whose nominal exit state has the smallest heuristic cost to go to the goal. State merging is impractical for Vulcan since nodes should only be combined if they have the same sequence execution risk. Pseudocode (from [69]) is presented in Procedure 6, Procedure 7, and Procedure 8 for convenience.

Vulcan supports a generalization of chance-constraints in which the upper limit on the risk can be a concave non-decreasing function Δ of the reward of the policy. Ayton and Williams define the "sequence execution risk" of a state history from step t to step n, $h_{t:n}$, via Eqs. 8-10 in [69] as:

$$ser(h_{t:n}) = \frac{1 - \prod_{i=t}^{n-1}(1 - r(s_i, a_i))}{\prod_{i=1}^{n-1}(1 - r_i(s_i, a_i))} \max_j \textbf{NOT } C_j(h_{t:n}) \tag{5.1}$$

where $r(s_i, a_i)$ is the probability of immediate constraint violation while executing action a_i from state s_i and the C_j are the constraint indicator functions. Thus, ser is automatically 0 for a state history that does violate a constraint. This construction is chosen so that the following constraint, which is the one actually enforced by Vulcan in Line 2 of Procedure 7,

Procedure 7 Sample(history $h_{0:t}$)

1: **if** history has reached the search depth **then**
2: **if** history satisfies the risk bound **then**
3: increment history's sample count $N_{h_{0:t}}$
4: **return** SUCCESS
5: **else**
6: **return** **false**
7: **while** history has at least one safe action **do**
8: **if** history's sample count is 0 **then**
9: set next action to that of the default policy for history
10: **else if** any safe actions available at history have never been sampled **then**
11: set next action to the first safe unsampled action
12: **else**
13: best_score= $-\infty$
14: **for all** safe actions available at history **do**
15: exploitation=estimated value of action at history
16: exploration= $\sqrt{\frac{\text{log history's sample count}}{\text{sample count of action at history}}}$
17: score= exploitation $+ c *$ exploration
18: **if** score>best_score **then**
19: set next action to action
20: set best_score to score
21: draw a parameter value from the proposal distribution and use this to get the child of next action, constructing a new one if the parameter value has never been drawn before
22: call Sample on the newly constructed child
23: **if** Sample returned **true then**
24: set sample count of next action at history to the total number of children of next action at history (if a child was drawn multiple times, each one counts as a sample)
25: **if** next action has been sampled at least once **then**
26: set estimated value of next action at history to the average value of the children plus the average value of the immediate reward for next action (if any)
27: **else**
28: set estimated value of next action at history to $-\infty$
29: set history's sample count to the sum of the sample counts of all the safe actions
30: set history's best action to the safe action with maximum estimated value
31: set history's value to the value of the best action
32: **return** **true**
33: **else**
34: mark next action as unsafe
35: set history's sample count to the sum of the sample counts of all the safe actions
36: **return** **false**

Procedure 8 Cleanup(history $h_{0:t}$)

```
1: if history's sample count N_{h_{0:t}} is 0 then
2:    if history satisfies the risk bound then
3:       return true
4:    else
5:       return false
6: if history has reached the search depth then
7:    return true
8: while history has at least one safe action do
9:    call Cleanup on all safe children of history's best action
10:   if all Cleanup calls returned true then
11:      set sample count of history's best action to the sum of the sample counts of its
         child histories
12:      set estimated value of history's best action to the average value of the children
         plus the average value of the immediate reward for next action (if any)
13:      set history's sample count to the sum of the sample counts of all the safe actions
14:      set history's value to the value of the best action
15:      return true
16:   else
17:      mark next action as unsafe
18:      set history's best action to the safe action with maximum estimated value
19: return false
```

is a sufficient condition for satisfying the execution risk bound:

$$ser(h_{0:n}) \leq \Delta(f(h_{0:n})) \tag{5.2}$$

for appropriately chosen f. For all experiments reported here, the risk tolerance is set to be a constant (which is necessarily non-decreasing). The f function chosen is from equation 20 in [69], which is identical for all children of a given action. Then, since all non-colliding children of a state have the same sequence execution risk, (Equation 5.2), which relates sequence execution risk, f, and Δ, will either be satisfied for all non-colliding children of an action or none of them.

5.3 Comparison with AO*

A great strength of MCTS and Vulcan is that they are "anytime" algorithms; on many problems they can run for a short amount of time and achieve decent performance, with solutions approaching the optimal solution as runtime is increased. By contrast, AO* must run for whatever length of time is required for it to find the optimal solution. However, on tasks where it is feasible to run AO* to completion, it generally provides better solutions than MCTS and Vulcan.

5.3.1 Strengths of AO*

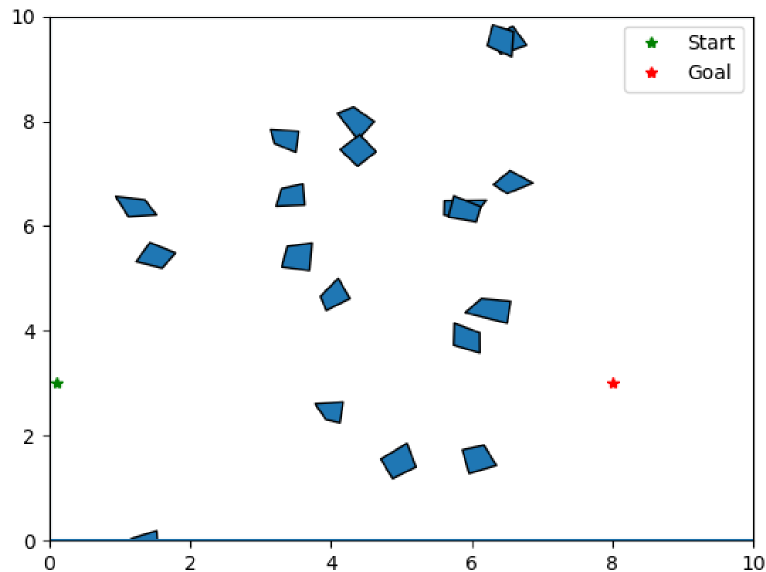

Figure 5.1: A field of Poisson distributed quadrilaterals

Consider the task shown in Figure 5.1. The system is once again the double integrator vehicle of (Equation 2.7), using the maneuver library of section 3.2. The cost function is the distance traveled plus the remaining distance to the goal. The risk tolerance was set to 20%. AO* leverages the 10 dimensional, degree 10 sparse grid from chapter 4 and returns a policy in 54 seconds. In 100 executions of this policy, the average cost incurred is 7.9061

meters (standard devation: 0.0047897 m) and there were no collisions. For the same risk tolerance, Vulcan was allowed 60 seconds to plan and used a 7x7 uniform grid of samples out to ± 3 standard deviations. It incurs an 11.4% higher cost of 8.8086 meters (standard deviation: 1.8983 m), and collides 8 times in 100 executions, despite being allowed to replan for an additional 60 seconds after each maneuver is executed. If a higher resolution set of samples is used for Vulcan (a 13x13 uniform grid), Vulcan performs even worse: the average cost increases to 12.501 meters (standard deviation: 0.47434 m) and it crashes 99 times in 100 trials. The problem here is that, with so many different possible outcomes for each action represented, 60 seconds is not enough time for Vulcan to consistently pass multiple trajectories through any successor states. The total number of possible action outcomes directly affects Vulcan's "early" efficiency because of the need to compute the immediate risk of constraint violation for an action as part of computing the sequence execution risk. This requires checking every realization and so scales linearly with the size of the uncertainty grid. As Vulcan loops, the same actions will get re-encountered and this value does not need to be recomputed, but for short runtimes this amortization is not available as each action occurs only a small number of times. In particular, with the 169 sample library and 60 seconds to plan, Vulcan is not even able to sample every action at the root once. As a result, for excessively short runtimes such as this, Vulcan is forced to plan with very incomplete information and can perform extremely poorly.

MCTS cannot enforce the chance-constraint and so is not, in general, directly comparable to AO* on tasks involving chance constraints. However, in this example the chance constraint is not "tight;" the AO* policy does not incur the maximum allowed risk. As a result, AO* solving without the chance constraint returns exactly the same plan. It is therefore reasonable to compare against MCTS, as well as against Vulcan, in this scenario. Unlike Vulcan, MCTS achieves nearly identical performance to AO* even when using the 13x13 grid: the average cost incurred cost over 100 trials is 7.9059 meters (standard deviation: 0.0047340 m) and no collisions occur. However, as with Vulcan this requires allowing

MCTS to replan after executing each maneuver, while AO* returns a full policy that can simply be executed. Note that if AO* is allowed to replan, it too achieves an average cost of 7.9059 meters (and its standard deviation of 0.0045033 m is actually slightly smaller).

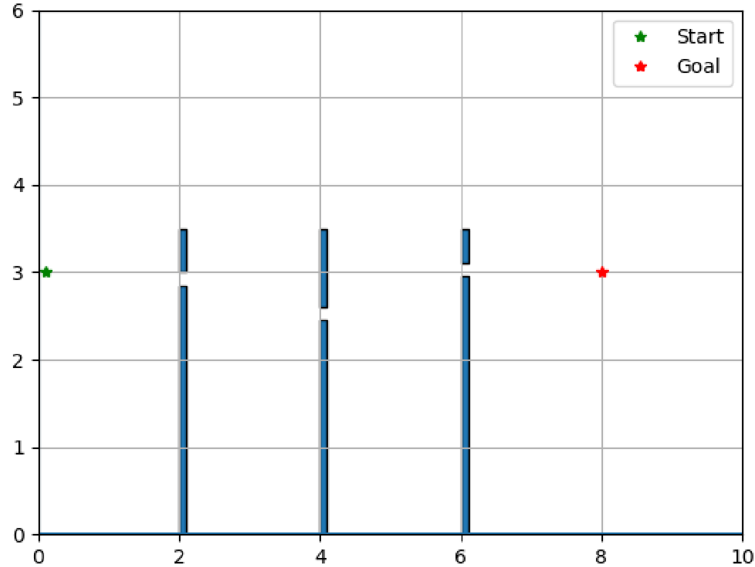

Figure 5.2: Obstacle field with three walls

Risk Tol	AO* Duration	AO* Cost	AO* Risk	Vulcan Cost	Vulcan Risk
1%	178s	8.2449 ± 0.044004	7.3%	10.3308 ± 3.1898	35%
5%	231s	8.1923 ± 0.043442	36%	10.4767 ± 2.5706	51%
10%	254s	8.1843 ± 0.037406	38%	9.93301 ± 1.68356	68%

Table 5.1: Performance of Vulcan at varying risk tolerance when allowed the same planning time as AO*.

AO* is particularly useful in scenarios where the chance constraint is tight. MCTS is not directly applicable, and the performance of Vulcan tends to degrade for demanding risk tolerances. Consider the task in Figure 5.2 and the statistics in Table 5.1. AO* planning was conducted using the 10 dimensional degree 10 grid from chapter 4. As the risk tolerance is decreased from 10% to 1% planning time decreases from 254 seconds to 178 seconds. The reported Cost and Risk are obtained from 100 simulations of the AO* policy, or 100

Risk Tol.	Cost	Std. Dev.	Risk
1%	12.1175	0.11053	100%
5%	11.9634	0.76574	97%
10%	11.9291	0.84951	95%
20%	11.3367	1.5915	79%

Table 5.2: Vulcan with fixed planning time at varying risk tolerance

executions in which Vulcan (with the 13x13 uniform grid) is allowed at each stage the time it took AO* to plan its complete policy (listed in Table 5.1). The average and standard deviation costs are reported. In the case of 1% tolerance, the AO* results are for 1000 simulations for better accuracy on the collision rate (note the violation of the risk tolerance due to 1. no replanning during execution 2. sacrificed accuracy at depth due to sparse integration). Where AO*'s planning time improves for steady solution quality at tighter risk tolerances, Vulcan misses the risk limit by larger and larger margins as the tolerance tightens. Note that if AO* uses the dense scheme on the 1% tolerance task, planning time increases to 670 seconds but only 1 in 100 trials collide (average cost: 8.2715, standard deviation: 0.041842). Moreover, if AO* uses the sparse scheme for 1% tolerance but is allowed to replan after each action, it does not collide at all in 100 trials (average cost: 8.2738, standard deviation: 0.040872).

To demonstrate that the degradation of Vulcan's performance is not (entirely) due to the reduced time it was allowed at tighter tolerances (justified by the fact that AO* needed less time at these tolerances), Table 5.2 records the performance of Vulcan at various risk tolerances, with a fixed per-action planning time of 20 seconds. Results are from 100 trials. Tighter risk tolerances at fixed (short) planning time lead to increasing rates of collision .

5.3.2 Strengths of Monte Carlo algorithms

MCTS and Vulcan are therefore best used when the problem is "too hard" to solve for optimality with AO*. This is most readily achieved by problems where the goal is distant, requiring deep search to complete a policy. While the sparse integration schemes of chap-

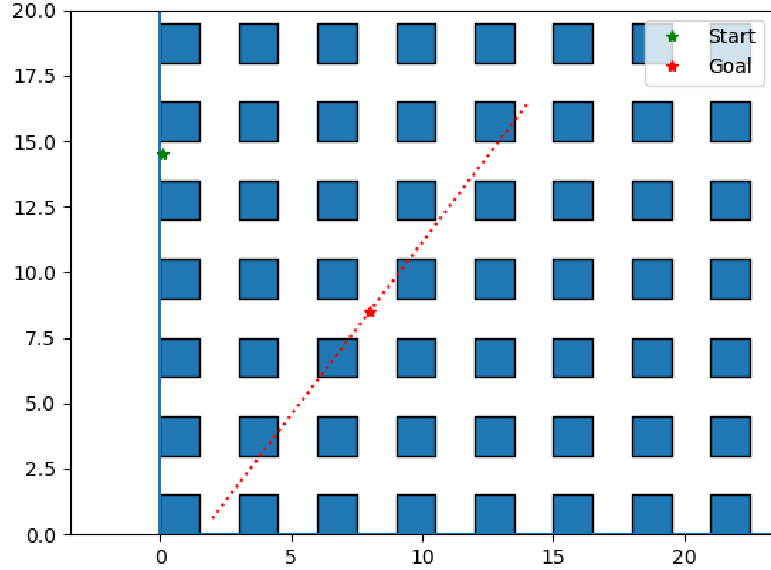

Figure 5.3: Deep search in a grid environment. Crossing the red dotted line counts as reaching the goal, but the cost is smaller if nearer to the goal point.

ter 4 help, for a sufficiently deep search problem the construction of the sparse scheme is itself intractable . On such challenging problems, MCTS may still be able to obtain a "good enough" solution with a relatively short amount of planning time. Figure 5.3 shows a task where the Manhattan distance to reach the goal is 13.9 meters; to encourage MCTS to actually attempt to reach the goal rather than crashing early to avoid incurring running cost, the cost function is set to be the distance traveled plus four times the remaining distance to the goal. With only 20 seconds of planning time to choose each maneuver, and using the 13x13 uniform grid, MCTS achieves an average cost over 100 trials of 12.9516 (standard deviation: 2.26434) and collides only once. Reaching the goal required 5 maneuvers in 49 trials, 6 maneuvers in 49 trials, and 8 maneuvers in one case (the last trial collided after only one maneuver). Such depth would require a 16 dimensional sparse integration scheme, which exceeds anything obtained in chapter 4. However, this task is also nearing the limits of what MCTS can handle. Moving the goal position 3 meters downrange as shown in Figure 5.4 notably degrades the performance of MCTS. Across 100 trials, again with

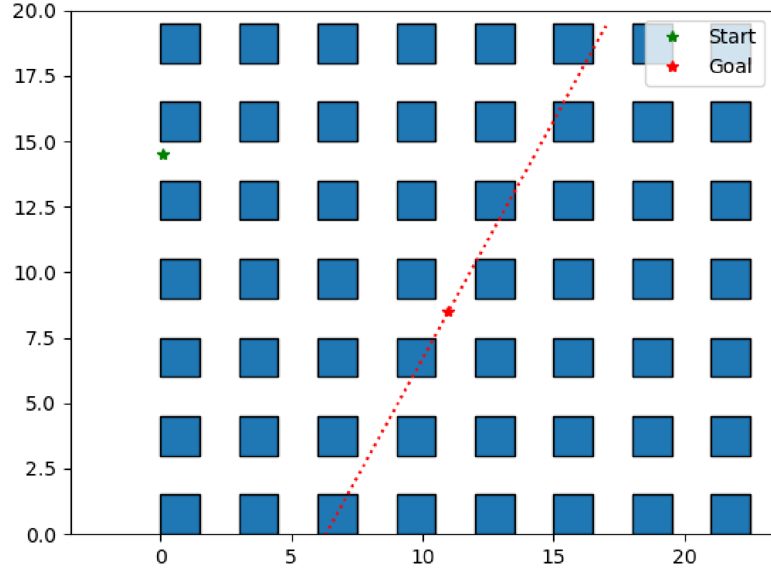

Figure 5.4: This task requires deeper search than Figure 5.3. Crossing the red dotted line counts as reaching the goal, but the cost is smaller if nearer to the goal point.

20 seconds of planning time per decision, it incurs an average cost of 17.5643 (standard deviation: 4.31181) and collides 4 times. The average cost now exceeds the Manhattan distance (16.9) and the number of crashes increased. The new task requires 6 maneuvers only once, while 7 maneuvers were needed 90 times and 8 maneuvers were needed 5 times. The remaining cases crash after 1, 3, or 5 maneuvers. Naturally, additional runtime could be allocated to MCTS to improve performance.

5.3.3 Supersonic Glide Vehicle Scenario

Vulcan can be more effective than AO* on problems that are relatively shallow. To show this, a new dynamical system is introduced in Equation 5.3; this is a very simple model of a supersonic glide vehicle at constant altitude. In addition to a classical quadratic drag model for speed v, the integrated heat load Q provides a (cyclic) state on which constraints could be imposed. Appended to the state is a "load" value l computed as the current drag times the current angular velocity ω, which provides additional interesting constraints. This is a

function of non-cyclic states (velocities) and so is itself non-cyclic.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{v} \\ \dot{\psi} \\ \dot{\omega} \\ \dot{Q} \\ \dot{l} \end{bmatrix} = \begin{bmatrix} v\cos\psi \\ v\sin\psi \\ -\frac{C_d\rho}{m}v^2 \\ \omega \\ c_{\text{eff}}u - D\omega \\ \frac{C_f}{4}C_d\rho v^3 \\ (c_{\text{eff}}u - \frac{2\omega C_d\rho v}{m})C_d\rho v^2 \end{bmatrix} \tag{5.3}$$

x and y are the downrange and crossrange position in meters, ψ is the yaw/angle between the velocity vector and the inertial x-axis. The non-cyclic coordinates are v and ω. Atmosphere density ρ, drag coefficient C_d, vehicle mass m, yaw control effectiveness c_{eff}, yaw damping D, and friction coefficient C_f are all parameters that could be uncertain for this model. Unlike the Dubins car and F16 models previously considered, this system has constantly decreasing speed and so can never return to a trim condition after leaving it. In fact, if C_d, ρ, and m are constant the speed can be obtained in closed form:

$$v(t) = \frac{mv_0}{C_d\rho v_0 t + m} \tag{5.4}$$

for v_0 the initial speed and t the elapsed time.

Going forward, $\rho = 1.225\frac{\text{kg}}{m^3}$, $m = 1000$kg, and $D = 10\frac{1}{s}$, while $C_d \sim N(0.1, 0.01)m^2$, $c_{\text{eff}} \sim N(1, 0.1)$, and $C_f \sim N(0.3, 0.01)s$. A maneuver library using proportional controllers tracking angular velocity was constructed for trims with $\omega = 0$rad/s at $v = 50$m/s to $v = 500$m/s in increments of 50m/s, as well as a trim at $v = 575$m/s and $v = 1000$m/s. Realizations of the 1000m/s maneuvers are shown in Figure 5.5 for five evenly spaced samples in c_{eff} and C_d, three in C_f, out to $\pm 3\sigma$. Integration was done using a RK4 integrator with a timestep of 1e-4 seconds, and states are recorded every hundredth of a second. Each

realization thus contains 602 points.

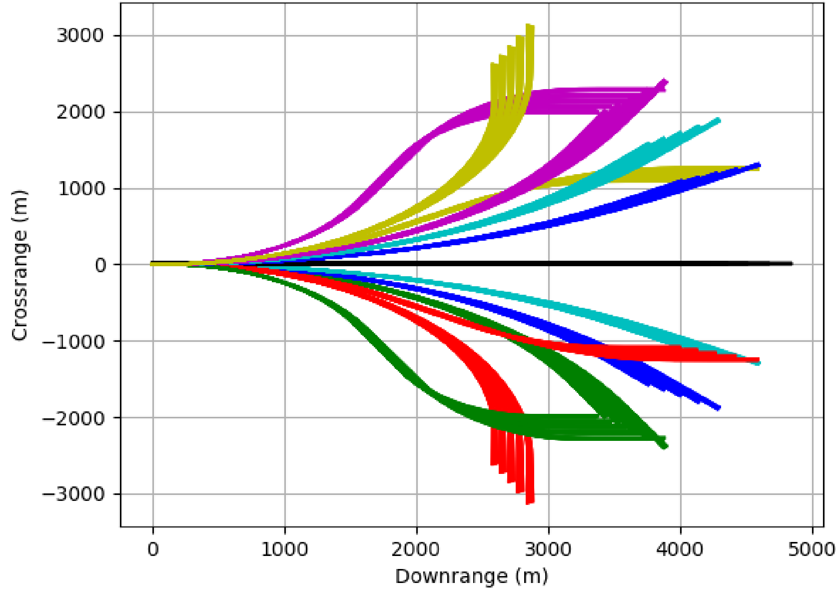

Figure 5.5: Realizations of maneuvers for the supersonic glide model starting with 0 angular velocity and 1000 m/s forward speed.

Consider the task shown in Figure 5.6, with the additional non-visible constraints that $-6.5e4\frac{N}{s} \leq l \leq 6.5e4\frac{N}{s}$ and $Q \leq 3.84e7J$, with an initial speed of $1000\frac{m}{s}$. This task requires up to three maneuvers to complete, and so involves 9-dimensional integrals. Additionally, since the uncertain parameters are not interchangeable (as they are in the case of wind uncertainty), there is less symmetry to use to construct sparse integration schemes. So Vulcan competes directly with a dense integration scheme for this task and proves advantageous. The incurred cost and risk of various planners, with and without chance constraints, are reported in Table 5.3. Each column evaluates the trajectories as (planned using the cost specified in the penalty column) using a different cost function to allow comparing the effects of cost function design on algorithm performance. The considered cost functions are all of the form $J = \text{distance traveled} + g(\text{terminal state})$. 1x, 4x, and 20x set g as the relevant constant times the distance between terminal state and goal state. The "4,1" penalty is more sophisticated: if the terminal state is not in the goal region (i.e. the trajectory violated

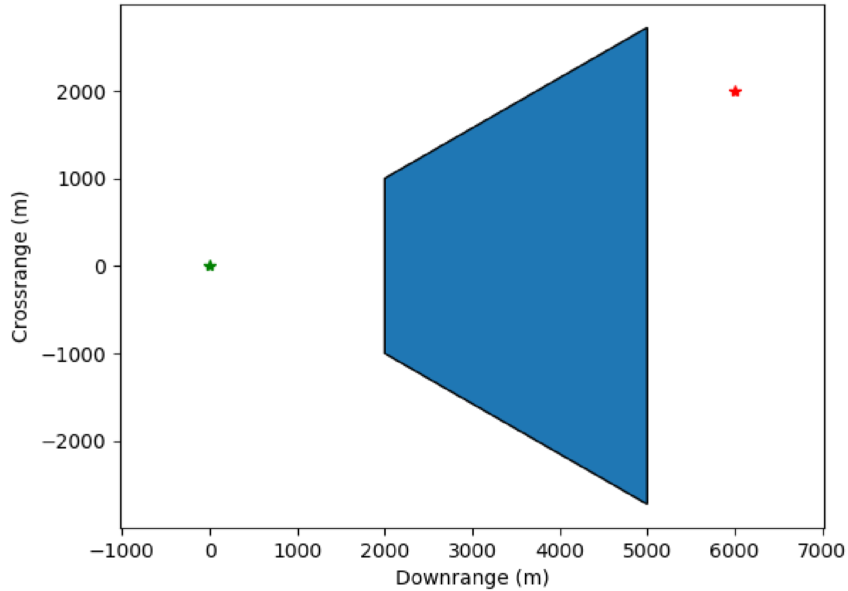

Figure 5.6: A three maneuver task for the supersonic glide vehicle

a constraint) the penalty is 4 times the remaining distance, but if the terminal state is in the goal region the penalty is just the distance to the goal state. This penalty structure increases the distinction between safe and unsafe trajectories. Results are reported for Vulcan and MCTS with 40 seconds to plan each action, while AO*'s policy is computed offline and executed. AO*'s library used 5 samples for c_{eff} and C_d and 3 samples for C_f; Vulcan and MCTS use a higher resolution sampling of the uncertainty (7 samples each for c_{eff} and C_d, 3 for C_f, 147 total). Due to the inflated costs involved in this problem, the exploration coefficient was increased to $c = 1000\sqrt(2)$ for both Vulcan and MCTS. AO* and MCTS employed state merging with a hyper-rectangle size of 1 meter in x and y, 1m/s in v, 0.05 radians in ψ, 0.05 rad/s in ω, $1000J$ in Q, and 100N/s in l.

AO* with the 1.0x penalty and a 20% risk tolerance requires 640 seconds to find a policy; when executed 100 times, this policy achieves an average cost of 8,654 meters and collides 13 times. By contrast, Vulcan can use a higher resolution sampling of the uncertainty and needs to plan for only 40 seconds per maneuver to get acceptable, though distinctly suboptimal, performance. In 100 trials, Vulcan incurred an average cost of 9,240

Table 5.3: Performance of planners with various cost structures on the glide vehicle task. AO* 20% is AO* with a 20% risk tolerance, AO* No CC is AO* without a chance constraint. Costs were computed for all cost structures on the resulting trajectories, with standard deviations over 100 trials in parentheses.

Planner	Penalty	Risk	1x cost	(4,1) cost	4x cost	20x cost
AO* 20%	1x	13%	8654 (816)	10367 (4238)	14215 (3085)	43870 (23019)
Vulcan 20%	1x	13%	9293 (1766)	11020 (4273)	16788 (6189)	56758 (34368)
AO* 20%	4,1	13%	8654 (816)	10367 (4238)	14215 (3085)	43870 (23019)
Vulcan 20%	4,1	13%	9630 (1982)	11526 (4608)	18134 (6963)	63491 (37652)
AO* No CC	4,1	13%	8654 (816)	10367 (4238)	14215 (3085)	43870 (23019)
AO* No CC	4x	13%	8654 (817)	10370 (4239)	14217 (3083)	43891 (23015)
MCTS	4,1	17%	8550 (929)	10781 (4564)	14453 (3224)	45932 (24487)
MCTS	20x	16%	8569 (916)	10746 (4561)	14461 (3214)	45883 (24470)

meters (standard deviation: 1,691 m) and also collided 13 times. Thus, provided cost optimal performance is not critical, one could use Vulcan to get a plan much faster than using AO*.

By contrast, MCTS requires a highly tailored cost function to achieve good performance with only 40 seconds for planning (using $c = 1000\sqrt{2}$). If the terminal penalty is simply four times the distance to the goal, MCTS returns plans with greater than 99% predicted chance of collision (not shown in the table). If the penalty is increased to 20 times the remaining distance, MCTS gets acceptable plans in 40 seconds and achieves over 100 trials an average cost, in the inflated penalty it was asked to minimize, of 45,883 (inflated costs are technically unit-less). 16 trajectories collide. Interestingly, if the same trajectories are scored using the 1x penalty it turns out the score is actually smaller than AO* achieved even seeking to minimize the 1x penalty . It is, however, colliding at a higher rate. Using a 20x weight on the final distance to the goal is both extreme and unnecessary; MCTS benefits in this case from a cost structure that better distinguishes between violating a constraint *while near the goal* and actually reaching said goal. Replacing a terminal penalty that is always some multiple of the remaining distance to the goal with one that uses a smaller multiplier if the terminal state is in the goal region improves the performance of both AO* and MCTS on this task. AO*, with no chance constraint and the 4,1 penalty, achieves an

average cost of 10,367 over 100 trials, 13 of which collide. The performance of this policy is *identical* to that obtained with a 20% chance constraint and the 1x penalty on all tested metrics . This policy was obtained in 668 seconds, which is slightly slower than with the chance constraint. With no chance constraint and a uniformly increased 4x penalty, the policy took 807 seconds to compute *and* was very slightly inferior from a cost perspective. If a 20% chance constraint is imposed on top of the 4,1 penalty, AO*'s results, total number of nodes, merges, collision checks, AND expansions, and OR expansions are unchanged. The primary effect of this goal aware penalty is thus reduce planning time to nearly match that of AO* with the chance constraint . MCTS using the 4,1 incurs an average cost over 100 trials of 10,781 with only 40 seconds of planning time, though it does collide 17 times. This is, as expected, inferior to the performance of the AO* policy. Note, however, that, if these trajectories are evaluated using the 1x penalty, the incurred cost is 8,550 meters. This is roughly the same as achieved with the 20x penalty and is superior to the cost incurred by Vulcan when Vulcan is asked to minimize that very same penalty. Moreover, Vulcan itself does NOT benefit from the 4,1 penalty; using that cost function causes Vulcan's achieved cost to go up in every tested metric .

5.4 Discussion

AO* is an optimal algorithm while MCTS and Vulcan are only asymptotically optimal. Thus for problems on which it is possible to run AO* to completion, the policy it yields will be better than could be obtained from MCTS or Vulcan (sometimes dramatically so). Accordingly, the best use case for these algorithms appears to be situations where letting AO* finish is not practical due to intractability or simply a need for very fast planner response (and a willingness to accept low quality solutions). A standout case for AO* is when chance constraints are present and extremely tight, as Vulcan appears to struggle with such problems and pure MCTS cannot enforce chance constraints (though it may still give a solution that respects them if the cost function is selected appropriately).

Careful selection of the cost function proves to be important for the effectiveness of MCTS, as one scenario shows that a cost function with insufficient importance on arriving at the goal will lead the algorithm to intentionally crash into an obstacle *even though* the optimal solution per AO* with the same cost structure (and no chance constraint) is to plan around it. This scenario serves to highlight the utility of chance constraints despite the concerns raised by Blau, Hogan, LaValle and others: AO* converges much faster when a chance constraint is active than when it is not, and Vulcan proves less sensitive to the precise definition of the cost than MCTS does. The case for chance constraints should not be overstated, however; much of the performance benefit for AO* can be replicated through a cost function that places extra penalty on constraint violation (and results in the same optimal policy as using the chance constraint). In fact, when MCTS uses this cost function it produces a better solution than Vulcan did with a chance constraint and the unmodified cost. This last reinforces the point made in the introduction: chance constraints are useful, but to leverage them it is necessary that the available planning algorithms not be worse than for the pure penalty case. AO* may in fact work better with chance constraints than without, but the same is not true for MCTS style algorithms. On multiple tasks where both MCTS and Vulcan were used, MCTS returned significantly lower cost solutions than Vulcan while still (admittedly coincidentally) respecting the chance constraint.

CHAPTER 6

LAZY COLLISION CHECKING

6.1 Generalized Lazy Hypergraph Search

A large part of the runtime of all four algorithms (AO* with induced heuristic, RAO*, MCTS, and Vulcan) is spent on checking realizations of actions to see if they violate a constraint. The runtime expense of collision checking is well-known in the graph search (i.e. deterministic path planning) setting, where it is commonly addressed via a "lazy" collision checking approach: algorithms that delay enforcing constraints on an edge until they are "confident" that edge is actually in the path. Lazy approaches came to prominence applied to probabilistic roadmap techniques in [86]. Many variations of this idea have been developed since, both for probabilistic roadmap algorithms [87, 88, 90, 91] and other approaches [89, 90]. Mandalika et al. [92] and Lim et al. [93] demonstrate a generalized lazy search architecture that captures numerous lazy approaches as special cases. This architecture conducts search (without constraint enforcement) until an "event" toggle is tripped (ex. "found a path to the goal" or "encountered a vertex with lower heuristic value than any seen before"). It then executes a selection rule to pick some edges in the current best path to enforce the constraints on (ex. first unchecked edge in the path), then toggles

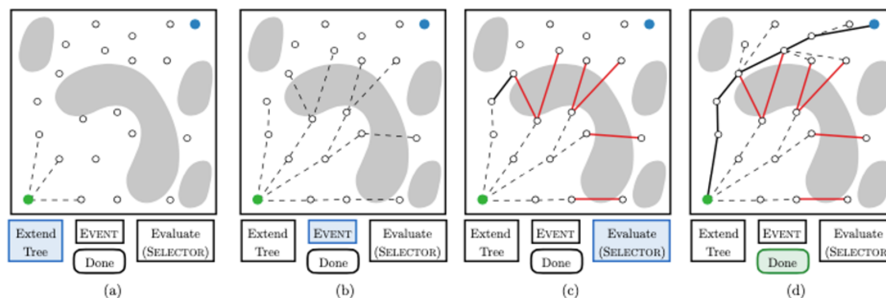

Figure 6.1: The Generalized Lazy Search architecture from Mandalika et al. It was devised for graph search but the idea applies to hypergraphs as well.

back to searching. This architecture is shown in Figure 6.1: in a), the search algorithm executes as if it searched an obstacle free environment. This continues until b), where an event function is suddenly satisfied. In c), a selector function chooses edges to check for collisions. This loops until d), where the search algorithm returns a path all the way to the goal that has been entirely collision checked. The Generalized Lazy Search (GLS) architecture can be extended to hypergraph search in order to improve the computational performance of the previously discussed algorithms. To this end, the "meta-algorithm" of Procedure 9 applies the approach of Generalized Lazy Search to structured hypergraph search (like AO*) and to MCTS.

Procedure 9 Generalized Lazy Hypergraph Search

```
 1: while not done do
 2:    while search not finished and event not true do
 3:       one step of the Hypergraph search
 4:    if search proved there is no solution then
 5:       done=true
 6:    if search finished and policy fully collision checked then
 7:       done=true
 8:    else
 9:       edges=chosen from policy according to selection rule
10:       Check edges for collision
11:       use hypergraph search's policy update to propagate the consequences of new col-
          lision detections
```

Lines 2-3 are simply an interleaving of the execution of the underlying search algorithm with checking the event toggle; this loop exits if the search algorithm says it is done or if the event is tripped. The search may terminate because it proved there can be no solution (Lines 4-5), or because it has found what it believes to be the best solution. Lines 6-7 handle the situation where the search terminated without an impossibility proof. If the search is AO* or RAO*, then the current policy *is* the best possible solution for the modified MDP where no collisions can occur in edges that have not yet been collision checked. Vulcan (or, in general, MCTS) is an asymptotically convergent algorithm and so it only stops based on a runtime constraint. Regardless, if the search algorithm thinks it has a solution and all the

edges included in the candidate have been collision checked, then this candidate is in fact the true solution and should be returned. If this fails, then either the search was interrupted by the event toggle or the candidate is not fully collision checked and so cannot be trusted. So in Line 9, unchecked edges are selected for evaluation (which happens in line 10). Line 11 then uses the policy update process of the underlying search algorithm to propagate the consequences of this collision checking. In the case of Vulcan, Procedure 9 also needs to call Cleanup (Procedure 8) on the starting state before returning its final answer (and it does NOT need to call Cleanup after each step of the Hypergraph search).

While Procedure 9 can be applied directly to AO*, RAO*, and MCTS, the original presentation of Vulcan lacks a policy update process that can serve in Line 11. The difficulty is that Vulcan uses the sequence execution risk to decide if an action is permitted w.r.t. the chance constraint. Recall the definition of ser from section 5.2:

$$ser(h_{t:n}) = \frac{1 - \prod_i^{n-1}(1 - r(s_i, a_i))}{\prod_i^{n-1}(1 - r(s_i, a_i))} \mathbf{1}_C(h_{t:n}) \tag{6.1}$$

$h_{t:n}$ is the sequence of states and actions under consideration, $r(s_i, a_i)$ is the probability of immediate constraint violation when executing the ith action in the history at the ith state in the history, and $\mathbf{1}_C$ is an indicator function that is 1 if the sequence of states doesn't violate the constraint and 0 otherwise. When doing a lazy search, r will initially be thought to be zero; the delayed edge evaluation can change this value and increase ser for all states that are below the updated edge in the search tree. Thus, for lazy search a special forward-backward policy update is needed that first propagates the changes in ser forward to the leaves of the tree, then propagates any changes in legal or best actions back to the root. The forward process is presented in Procedure 11 and the full policy update (which calls the forward process) in Procedure 12. Both algorithms conduct an action update process, which is factored out and presented as Procedure 10.

Three events and one selector for the Generalized Lazy Hypergraph Search are im-

Procedure 10 Action Update for Vulcan (history, actions)

```
1: for all actions passed in to this procedure do
2:    set the sample count of the action to the sum of the child sample counts
3:    set the value of the action to the average of the child values
4:    set history's sample count to the sum of the sample counts for each safe action
5: set history's best action largest valued safe action
6: set history's value to the value of the current best action
```

Procedure 11 Forward Update for Vulcan (history)

```
1: if history violates a constraint then
2:    set history's value to the terminal reward
3:    return true
4: if history is not the root then
5:    set history's cumulative safety to the parent's cumulative safety*(1-risk of action at
      parent)
6:    set ser to (1-history's cumulative safety)/history's cumulative safety
7: if history is terminal then
8:    set history's value to the terminal reward
9:    return history's ser≤ risk bounding function
10: while history has any safe actions do
11:    if all non-colliding children of current best action satisfy the risk bound then
12:       for all children of current best action do
13:          call Forward Update on the child history
14:       if all Forward Updates returned true then
15:          call Action Update on the current best action
16:          return true
17:    mark current best action as unsafe
18:    set history's best action largest valued safe action
19: return false
```

plemented. The first event ("shortest path" from [92]) fires only if the underlying search finishes, and so is only useful for R/AO* (MCTS variants are never "done" searching since they are asymptotic algorithms). The second event is a "heuristic progress" event, which fires if the planner places a state into the policy that has a smaller heuristically estimated cost to go (equivalently, larger reward) than for any state whose incident edge has been collision checked. As one of the strengths of MCTS is that it does not need a heuristic, this event is less useful here. The final implemented event, "constant depth," fires if the current policy has at least a set number of unevaluated edges. The only implemented selector

Procedure 12 Policy Update for Vulcan (history)

```
1: call Forward Update on history
2: if Forward Update returned false then
3:     mark the action leading to history unsafe in every parent
4:     return
5: if history is not the root then
6:     Q=First-In-First-Out queue where re-adding any entry pushes it to the end
7:     put all safe actions at history into Q
8:     while Q is not empty do
9:         get a history h and a set of actions {a_i} from Q
10:        call Action Update on h, {a_i}
11:        if Action Update changed the value of h then
12:            for all parents of h do
13:                Q adds all actions at the parent that lead to h
```

chooses the unchecked edges belonging to the first action in the policy that has unchecked edges (the "forward selector" from [92]).

6.1.1 Lazy R/AO*

Table 6.1 collects performance data for R/AO* on the first Dubins Car task from chapter 3 (see Figure 3.6). RAO* benefits dramatically from lazy collision checking yet remains inferior to regular AO* in terms of runtime. Regular AO* performs the same number of collision checks as lazy RAO* with the constant depth event if the depth is set to the number of uncertainty samples used for a single action: the constant depth event is roughly equivalent to using the induced heuristic to run AO*. RAO* with shortest path or heuristic progress events does fewer collision checks than regular AO*, but is still slower due to the extra search effort from laziness.

In contrast, the data shows that AO* does not benefit from lazy collision checking with any of the three implemented events. The shortest path and heuristic progress events *do* slightly reduce the number of collision checks but force many extra AND and OR node expansions. As a result, the total runtime increases. This suggests that for the current test problem, the induced heuristic strikes an effective balance between minimizing collision checks and search effort (represented by the number of AND and OR expansions). The

underlying heuristic (the Euclidean distance to the goal) is fairly accurate in this environment as there are not terribly many obstacles and the vehicle is initially headed towards the goal; laziness might perform better relative to the induced heuristic on a task where the heuristic is less accurate. When using the induced heuristic, AO* conducts exclusively OR expansions (which do not check constraints) until such time as there is an AND node on the best path with estimated cost (using the induced heuristic) smaller than the estimated cost of any of the OR nodes on the best path. This is a permissible event function for GLS . AO* then conducts an AND expansion (i.e. checks for collisions and updates the policy) of just that AND node before returning to OR expansions. This is a selection rule . The induced heuristic itself could be implemented as an instance of the GLS approach (for hypergraphs). This helps to explain the limited benefits of further lazy collision checking: a (comparatively sophisticated) lazy scheme is already in place to reduce the number of collision checks. While the induced heuristic does not appear to be quite as edge efficient as some of the other possibilities, it is dramatically more vertex efficient. In a scenario with sufficiently computationally expensive collision checks a runtime improvement might be observed, but the effect would be at best incremental. In the observed problems, the total reduction in collision check count via laziness applied on top of the induced heuristic is merely 10%.

Table 6.1: Effects of three events (SP=shortest path, HP=heuristic progress, CD=constant depth) on performance of R/AO* on a two obstacle search task.

	Dur(s)	Nodes	Checks	AND	OR
RAO*	76	35,354	124,950	1,275	25
SP RAO*	31	112,594	2,646	5,814	114
HP RAO*	25	91,209	2,646	4,539	89
CD RAO*	10	38,641	2,842	1,428	28
AO*	6	1,381	2,842	29	25
SP/HP AO*	20	4,614	2,548	96	112

6.1.2 Lazy Vulcan

For Vulcan, the Heuristic Progress and Shortest Path events are not strictly applicable, so the focus is on the constant depth event. For relatively long permitted runtimes, laziness can improve the quality of the solution Vulcan returns. Vulcan was tested with and without laziness on the second task from chapter 3 (see Figure 3.7), using the same 51 maneuver library but with a 9x9 uniform grid across $\pm 3\sigma$ for uncertainty sampling. Accordingly, the constant depth event fired when there were 81 unchecked edges. The risk tolerance was 20%. The predicted cost and risk presented in Table 6.2 were obtained from 36 independent runs of the algorithm with a 180 second allowed runtime, which was sufficient for Vulcan to stabilize its choice of initial action. The actual cost and risk are average and standard deviation values from 1000 "closed loop" trials: the algorithm was run for 180 seconds, the action chosen for the current state was executed, and then the algorithm was run again (repeating until a constraint violation or goal region was reached). Laziness resulted in a plan predicted to be lower cost but higher risk, and in practice the plan was cheaper but risky (using almost the entire risk budget). Interestingly, the predictions were less accurate for lazy Vulcan (despite the improved average performance); this is consistent with the greater variation in achieved cost for the lazy variant.

Table 6.2: Performance information for Vulcan with and without the use of the constant depth event Generalized Lazy Search Architecture.

Lazy	Predicted Cost	Actual Cost	Cost Error	Predicted Risk	Actual Risk
No	7.113 ± 0.006088	7.123 ± 0.04046	0.198%	$7.472\% \pm 5.569\%$	6.7%
Yes	6.941 ± 0.01513	7.057 ± 0.09218	1.65%	$12.62\% \pm 1.905\%$	19.6%

The variability (perhaps "volatility" is a better word) of the actual performance of Lazy Vulcan shown here is consistent with one limitation of laziness for Vulcan: with short runtimes (i.e. when used for online, suboptimal planning), laziness can cause serious planning errors. The issue is that, when the available runtime is small, Lazy Vulcan may not have time to "repair" a plan that has been discovered to have obstacles making it unsafe. Fig-

ure 6.2, Figure 6.3, and Figure 6.4 show three environments in which lazy collision checking with short (in all cases, 20 seconds) allowed planning time leads to severely degraded performance. The plots show the obstacle layout as well as the trajectories considered by (a sample run of) Vulcan while planning the first maneuver. In all cases, the 51 maneuver library with a 13x13 uniform grid across $\pm 3\sigma$ was used; the constant depth event fired when 169 edges were unchecked. In the first scenario, Vulcan is able to achieve a 13/100 collision rate while lazy Vulcan crashes 28/100 times. The second example is more challenging; regular Vulcan violates the constraint 33/100 times while Lazy Vulcan crashes 45/100 times. In the grid environment, Vulcan collides 37/100 times while Lazy Vulcan does so in 77/100 trials .

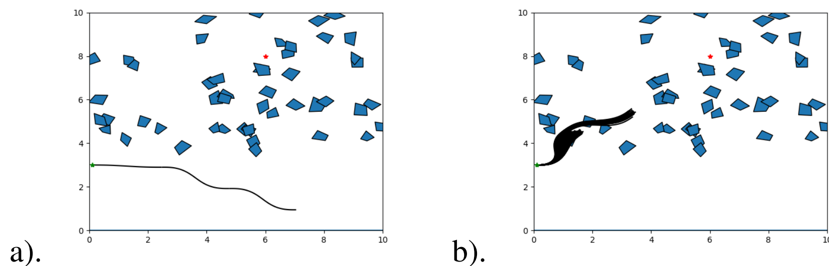

Figure 6.2: This field consists of Poisson distributed quadrilaterals. The goal is to travel 6 meters downrange, with a cost penalty equal to the final distance to the red star. With a risk tolerance of 20% and 20 seconds to plan, regular Vulcan accepts the low risk-low reward option of traveling below the obstacle forest, but Lazy Vulcan tries to fly through the forest and violates the risk tolerance.

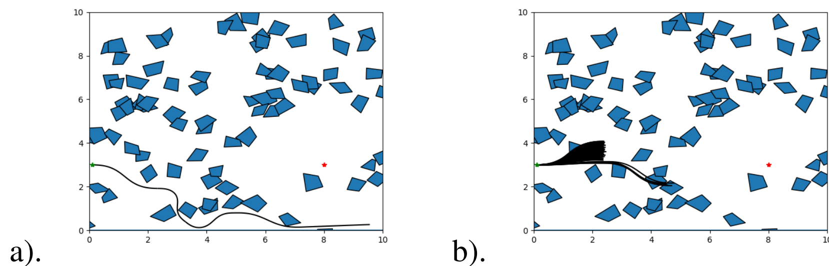

Figure 6.3: This field consists of more densely Poisson distributed quadrilaterals. The goal is to travel 8 meters downrange, with a cost penalty equal to the final distance to the red star. With a risk tolerance of 20% and 20 seconds to plan, regular Vulcan on average finds a safer plan than Lazy Vulcan does.

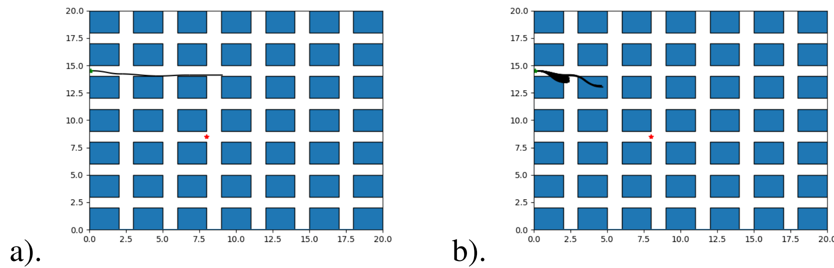

Figure 6.4: In this regular grid, Lazy Vulcan initially thinks it can fly straight towards the goal. When the lazy collision checking finds the box in the way, there is no longer time to plan something safer.

When using Vulcan with short runtimes, the policy will in general be suboptimal (as the algorithm is only asymptotically convergent). So in any situation where this is appropriate, solution cost-optimality is not a major concern. The primary effect of lazy collision checking for Vulcan is to improve solution cost-optimality, which as shown above comes at the cost of unexpectedly high risk plans. Trading a little bit of time or fuel savings for the occasional wildly irresponsible plan seems a poor decision, so laziness is not recommended for use with Vulcan at short runtimes.

6.1.3 Lazy MCTS

As with Vulcan, the Heuristic Progress and Shortest Path events are not applicable to MCTS. The effects of laziness on MCTS broadly mirror the effects on Vulcan (unsurprising, since Vulcan is simply a variant). For extended planning times, laziness can provide an incremental improvement in terms of cost at the expense of reduced predictive accuracy. MCTS was run for 180 seconds with and without laziness on the same task as Vulcan (Figure 3.7, with 51 maneuvers, a 9x9 uniform grid of samples, and a cost function equal to the distance traveled plus 1.3 times the remaining distance to the goal). The performance is recorded in Table 6.3, which shows the same pattern as Table 6.2: improved average case cost on execution, in exchange for increased error in predicting said cost. MCTS also collides less often when laziness is used.

Table 6.3: Performance information for MCTS with and without the use of the constant depth event Generalized Lazy Search Architecture. The event fires when there are 81 unchecked edges in the current best path estimate. Average and standard deviation values were obtained from 100 trials.

Lazy	Predicted Cost	Actual Cost	Cost Error	Predicted Risk	Actual Risk
No	7.234 ± 0.01088	7.227 ± 0.08494	0.0969%	$2.852\% \pm 1.182\%$	1%
Yes	7.198 ± 0.003747	7.184 ± 0.02635	0.195%	$1.576 \pm 0.5468\%$	0%

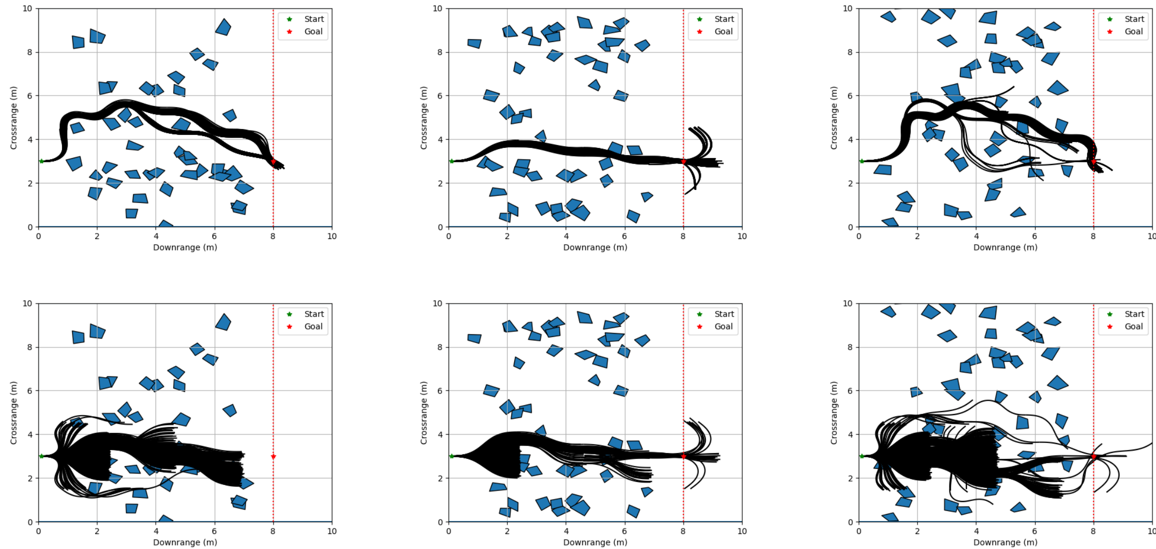

Figure 6.5: The top row shows the edges checked during search by MCTS on three of the thirty random fields; the bottom row shows the edges checked by Lazy MCTS on the same.

However, as previously discussed the primary use case for MCTS is using short planning times to get "good enough" solutions. To study the utility of laziness in this application, a set of thirty Poisson distributed obstacle fields were generated. The collision checks done by a single execution of MCTS and Lazy MCTS for three of those fields are presented in Figure 6.5; the same initial and goal positions are used for every task. The algorithms perform roughly the same on the middle task, while MCTS is better on the left column and Lazy MCTS is better on the right column. MCTS is allotted 20 seconds of planning time per decision and seeks to minimize a cost defined as the distance traveled plus three times the distance to the goal position. The double integrator model is used for dynamics and the library is the same as for the previous set of examples. In 20 of the 30 cases, laziness

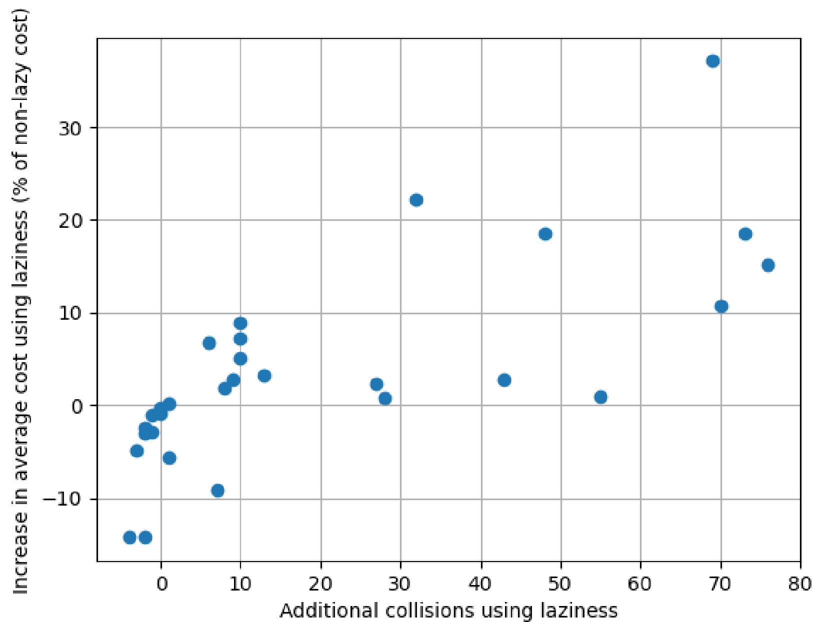

Figure 6.6: Change in cost by introducing laziness versus change in rate of collision.

resulted in increased numbers of collisions; Figure 6.6 plots the change in average cost over 100 trials against the change in the number of collisions. All of the cases where laziness reduced the number of collisions resulted in reduced costs; two cases where collisions increased also reduced costs. In the remaining 18 cases, laziness resulted in more constraint violations and increased average cost incurred. Moreover, the improvements offered by laziness when they did occur were usually small; only 2 of the 12 cases where cost improved yielded an improvement greater than 10% of the non-lazy cost (and only 4 exceed 5%). It appears that laziness will only rarely have a large effect on the quality of the policy obtained for short runtimes, and when it does have a large effect the effect may be quite negative: consider that in 10 of 30 scenarios, the number of collisions increased by 27 or more in 100 trials. From an end-user perspective this is very undesireable behavior; minor improvements in solution cost (when one has already accepted sub-optimal solutions by deciding to use an anytime algorithm) are not worth the risk of extreme underperformance as occurs in the case where laziness increases the average cost from 9.34 to 12.81 and the collision rate from 5% to 74% .

6.2 Discussion

The GLS architecture proves to be something of a disappointment; it is clearly beneficial to RAO*, but the established events and selectors prove inferior to the specialized induced heuristic. The induced heuristic itself lazily enforces constraints, as the exit states of maneuvers are computed when the OR node is expanded but the collisions are not checked until the AND node is expanded. As a result, AO* with the induced heuristic does not benefit from further lazy collision checking wrapped around it. For the sampling based algorithms, laziness provides minor benefits in terms of solution quality at extended planning time. The primary use case for these algorithms, though, is for short planning times, where laziness proves unreliable. In numerous tested scenarios, using lazy collision checking on MCTS or Vulcan with only a short planning time leads to unacceptably elevated rates of constraint violation, possibly in exchange for improvements in the average incurred cost. Short planning times necessarily mean the user is willing to accept suboptimal solutions from their planner, so using a variant that poses a risk of extreme constraint violation rates in exchange for slightly less suboptimal solutions seems unreasonable.

CHAPTER 7
CONCLUSION

7.1 Contributions

This work has extended the motion primitive framework developed by Frazzoli et al. [1] to systems with parametric uncertainty by leveraging an explicit uncertainty quantification technique based on the Koopman operator [101]. For the problem of path planning under parametric uncertainty with chance constraints, these uncertain motion primitives result in a Chance-Constrained Markov Decision Process. Experimental results demonstrate two key advantages of this approach over previous motion primitive techniques for uncertain systems. First, in contrast to both robust and deterministic approaches, the chance-constrained formulation allows path optimization to be smoothly traded for collision risk. Secondly, compared to Monte Carlo-based schemes, the explicit Koopman operator approach allows the planner to accurately quantify risk with a small number of samples, reducing computational effort to achieve a given level of accuracy.

By restricting the action set of the problem to the set of uncertain maneuvers, the CCMDP obtained via the uncertain motion primitive formulation can be solved as a search problem on an And/Or tree [99]. Chance constraints can be incorporated in this setting by algorithms such as RAO* [55], but results have shown that this algorithm scales poorly to problems with large numbers of actions (as is desireable in the uncertain motion primitive setting). Accordingly, this work presented a heuristic on actions for And/Or search (dubbed the "induced" heuristic) that leverages the computational advantages of motion primitives and the Koopman operator to accelerate the hypergraph search. AO* with the induced heuristic risks performing more OR expansions than RAO* in exchange for generally reducing the number of AND expansions. In the motion primitive context, OR expansions

are cheap as the final state of a given maneuver realization can be computed immediately from the group displacement history. AND expansions remain expensive due to the need for collision checking, though, so the ability of the induced heuristic to reduce the number of those expansions means it is generally faster, often much faster, than RAO*. Additionally, techniques were developed to accelerate search by efficiently converting the search tree into a search graph, by using decaying numbers of parameter samples as the time horizon increases, and by exploiting simultaneous symmetry in dynamics and constraints to offline computation necessary for planar and convex polytopic constraints.

The key calculation involved in chance constrained path planning is the expected value. In addition to the need to compute expected costs for the objective function, the probability of violating a constraint is itself an expected value of an indicator function. Expected values are integrals whose integration domain is the uncertainty set and so such calculations can benefit from techniques from numerical integration. chapter 4 describes a process using mixed integer-linear programming to construct sparse integration schemes tailored to the problem of search on a hypergraph. Prior work has shown that sparse schemes can be constructed by downsampling dense schemes [75]; using mixed integer-linear programming to do so ensures that the resulting schemes are as sparse as possible, while also enforcing technical requirements necessary for compatibility with hypergraph search. The naive approach to constructing a MILP for this results in a poorly conditioned problem involving enormous numbers of variables; leveraging the inherent symmetry of many integration domains as described in [75] helps ensure the MILP is tractable. Additional improvements in tractability are obtained through scaling tricks. Once rendered tractable, the MILP is an improvement on the suboptimal algorithm presented by [75], as it yields maximally sparse rules. Optimal sparse schemes obtained via MILP are shown to provide dramatic runtime reductions for chance constrained path planning.

In addition to deterministic sparse integration schemes, high dimensional integration can be tackled via Monte Carlo methods. Accordingly, chapter 5 provides a comparative

analysis of AO* and two Monte Carlo Tree Search methods: conventional MCTS, as well as Vulcan, which is capable of enforcing chance constraints. AO*, which is both optimal and complete, yields the best possible policy for a given task. Thus, on tasks where it is practical to wait for AO* to finish, it is preferred over MCTS or Vulcan which would yield inferior solutions. AO* is particularly useful in problems with demanding risk tolerances, as Vulcan's performance tends to degrade in these settings while AO*'s may actually benefit. AO*'s planning time can be reduced by making the risk tolerance more stringent. This occurs because tightening the risk tolerance reduces the set of feasible policies; as a result, AO* can discard suboptimal policies sooner. The recommended use case for MCTS or Vulcan is situations where the AO* solution time is excessive. These algorithms can provide a guaranteed update rate and in many situations can offer a moderate quality solution very quickly, even when the optimal solution is difficult to compute (and so AO* is impractical).

Of the two types of expected value calculation, the cost component is in general quite tractable as the integrand is usually cheap to evaluate. The motion primitive formulation is particularly helpful here, as any components of the cost function that respect the symmetry of the dynamics (such as a distance traveled component in many settings) can be computed offline and stored just like the displacements. Thus, the constraint evaluations are a major driver of the computational cost of not only AO*, but also MCTS and Vulcan as well. In specialized settings the presorting technique described in chapter 3 helps, but for general constraints it is not applicable. This work presented a lazy collision checking (or, more generally, lazy *constraint evaluation*) paradigm based on the work of Mandalika et al. [92] and applied it to RAO*, AO*, MCTS, and Vulcan on a variety of tasks. Lazy collision checking proved beneficial to RAO* but was outperformed by AO* using the induced heuristic. This is a result of the fact that the induced heuristic can itself be viewed as a lazy collision checking approach. For extended planning times lazy collision checking was found to offer small benefits for Vulcan and MCTS, but in the recommended use case of problems with short planning times it proved unreliable. While some problems saw

improved performance due to laziness, others saw dramatic increases in collision rates.

7.2 Future Directions

This work has shown that for problems of large scale and (especially) high dimensional uncertainty, AO* is not practical as an online planning algorithm. However, using MCTS or Vulcan directly on such problems can lead to poor quality solutions. Future work should look to marry the quality of AO* solutions with the anytime nature of MCTS-style algorithms. One possible approach would be to use AO* offline to produce a coarse policy using low resolution sampling of the uncertainty and then use this policy as the default policy for an MCTS algorithm. This would tend to improve the quality of the samples and could thus improve solution quality.

In the realm of chance constraints, while Vulcan represents a workable solution to the challenge of enforcing chance constraints in an MCTS style algorithm, it attempts to guarantee constraint satisfaction despite *probabilistically* approximating an optimal solution. There may be benefit in accepting a degree of approximation error in not just cost optimality but also constraint satisfaction. This might alleviate a trend observed in chapter 5, where pure penalty MCTS returns lower cost solutions than Vulcan at fixed planning times; the need to enforce the chance constraint reduces the convergence rate of Vulcan below what MCTS "should" be capable of.

Finally, this work has assumed access to the "true" distribution of the uncertain parameters. It is straightforward to extend the AND/OR search approach to handle partial observability, which would allow the system to update its parameter distribution online (see, for example, RAO* in [55]). However, this is computationally challenging as it increases the dimension of the uncertainty domain to also include the dimension of the observation domain. If actions and observations are sufficiently decoupled, one could potentially dispense with the full partial observability formulation and simply allow an estimator to update the parameter distribution during closed loop planning; provided the support of the param-

eter distribution does not change, the same realizations can be used (just with different weights/probabilities during the sum).

Appendices

APPENDIX A

INVARIANCE OF DUBINS CAR MODEL TO ACTION OF SE{2}

Proof:

$$g \circ \rho_\mu(t, t_0, x_0, s) = \rho_\mu(t, t_0, g \circ x_0, s) \iff g \circ [x_0 + f(x_0, \mu(t), s)dt] = g \circ x_0 + f(g \circ x_0, \mu(t), s)dt \tag{A.1}$$

for f the controlled derivative of the system. In the case of the Dubins Car model given in equation (Equation 2.7) and with $\mathbb{G} = SE\{2\}$:

$$x_0 = \begin{bmatrix} x \\ y \\ \theta \\ \omega \end{bmatrix}, s = \begin{bmatrix} w_x \\ w_y \end{bmatrix}, g = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & p \\ \sin(\phi) & \cos(\phi) & q \\ 0 & 0 & 1 \end{bmatrix}, \mu(t) = u, f(x_0, \mu(t), s) = \begin{bmatrix} v\cos(\theta) + w_x \\ v\sin(\theta) + w_y \\ \omega \\ u \end{bmatrix} \tag{A.2}$$

which yields:

$$g \circ x_0 = \begin{bmatrix} \cos(\phi)x - \sin(\phi)y + p \\ \sin(\phi)x + \cos(\phi)y + q \\ \theta + \phi \\ \omega \end{bmatrix} \Rightarrow f(g \circ x_0, \mu(t), s) = \begin{bmatrix} v\cos(\theta + \phi) + w_x \\ v\sin(\theta + \phi) + w_y \\ \omega \\ u \end{bmatrix} \tag{A.3}$$

Then

$$g \circ [x_0 + f(x_0, \mu(t), s)dt] \tag{A.4}$$

$$= \begin{bmatrix} \cos(\phi)[x + v\cos(\theta)dt + w_x dt] - \sin(\phi)[y + v\sin(\theta)dt + w_y dt] + p \\ \sin(\phi)[x + v\cos(\theta)dt + w_x dt] + \cos(\phi)[y + v\sin(\theta)dt + w_y dt] + q \\ \theta + \phi + \omega dt \\ \omega + udt \end{bmatrix} \tag{A.5}$$

$$= \begin{bmatrix} \cos(\phi)x - \sin(\phi)y + p + [v\cos(\theta + \phi) + \cos(\phi)w_x - \sin(\phi)w_y]dt \\ \sin(\phi)x + \cos(\phi)y + q + [v\sin(\theta + \phi) + \sin(\phi)w_x + \cos(\phi)w_y]dt \\ \theta + \phi + \omega dt \\ \omega + udt \end{bmatrix} \tag{A.6}$$

Whereas the right hand side becomes:

$$g \circ x_0 + f(g \circ x_0, \mu(t), s)dt = \begin{bmatrix} \cos(\phi)x - \sin(\phi)y + p + [v\cos(\theta + \phi) + w_x]dt \\ \sin(\phi)x + \cos(\phi)y + q + [v\sin(\theta + \phi) + w_y]dt \\ \theta + \phi + \omega dt \\ \omega + udt \end{bmatrix} \tag{A.7}$$

The difference is only that the wind has been rotated along with the trajectory on the LHS. So the dynamics are invariant as long as the wind probability distribution is rotationally symmetric. ■

REFERENCES

[1] E. Frazzoli, M. A. Dahleh, and E. Feron, “Maneuver-based motion planning for nonlinear systems with symmetries,” *IEEE Transactions on Robotics*, vol. 21, no. 6, pp. 1077–1091, Dec. 2005.

[2] A. Gray, Y. Gao, T. Lin, J. K. Hedrick, H. E. Tseng, and F. Borrelli, “Predictive control for agile semi-autonomous ground vehicles using motion primitives,” in *2012 American Control Conference (ACC)*, Jun. 2012, pp. 4239–4244.

[3] M. Pivtoraiko, D. Mellinger, and V. Kumar, “Incremental micro-uav motion replanning for exploring unknown environments,” in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 2452–2458.

[4] D. J. Grymin, C. B. Neas, and M. Farhood, “A hierarchical approach for primitive-based motion planning and control of autonomous vehicles,” *Robotics and Autonomous Systems*, vol. 62, no. 2, pp. 214–228, 2014.

[5] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia, “Automated composition of motion primitives for multi-robot systems from safe LTL specifications,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2014, pp. 1525–1532.

[6] A. A. Paranjape, K. C. Meier, X. Shi, S.-J. Chung, and S. Hutchinson, “Motion primitives and 3D path planning for fast flight through a forest,” *The International Journal of Robotics Research*, vol. 34, no. 3, pp. 357–377, 2015.

[7] M. Vukosavljev, Z. Kroeze, M. E. Broucke, and A. P. Schoellig, “A framework for multi-vehicle navigation using feedback-based motion primitives,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 223–229.

[8] Z. C. Goddard *et al.*, “Utilizing reinforcement learning to continuously improve a primitive-based motion planner,” in *AIAA Scitech 2021 Forum*. 2021. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2021-1752.

[9] T. Schouwenaars, B. Mettler, E. Feron, and J. P. How, “Robust motion planning using a maneuver automaton with built-in uncertainties,” in *Proceedings of the 2003 American Control Conference, 2003.*, vol. 3, Jun. 2003, 2211–2216 vol.3.

[10] A. Majumdar and R. Tedrake, “Funnel libraries for real-time robust feedback motion planning,” *The International Journal of Robotics Research*, vol. 36, no. 8, pp. 947–982, Jun. 2017.

[11] A. Thakur, P. Svec, and S. K. Gupta, “Gpu based generation of state transition models using simulations for unmanned surface vehicle trajectory planning,” *Robotics and Autonomous Systems*, vol. 60, no. 12, pp. 1457–1471, 2012.

[12] A. Charnes, W. W. Cooper, and G. H. Symonds, “Cost horizons and certainty equivalents: An approach to stochastic programming of heating oil,” *Management Science*, vol. 4, no. 3, pp. 235–263, 1958.

[13] A. T. Schwarm and M. Nikolaou, “Chance-constrained model predictive control,” *AIChE Journal*, vol. 45, no. 8, pp. 1743–1752, 1999.

[14] L. Blackmore, Hui Li, and B. Williams, “A probabilistic approach to optimal robust path planning with obstacles,” in *2006 American Control Conference*, Jun. 2006, pp. 7–14.

[15] A. Undurti and J. P. How, “An online algorithm for constrained pomdps,” in *2010 IEEE International Conference on Robotics and Automation*, IEEE, 2010, pp. 3966–3973.

[16] L. Blackmore, M. Ono, A. Bektassov, and B. C. Williams, “A probabilistic particle-control approximation of chance-constrained stochastic predictive control,” *IEEE Transactions on Robotics*, vol. 26, no. 3, pp. 502–517, Jun. 2010.

[17] R. A. Blau, “Stochastic programming and decision analysis: An apparent dilemma,” *Management Science*, vol. 21, no. 3, pp. 271–276, 1974.

[18] A. Charnes and W. Cooper, “Note—a comment on blau’s dilemma in “stochastic programming” and bayesian decision analysis,” *Management Science*, vol. 22, no. 4, pp. 498–500, 1975.

[19] A. J. Hogan, J. G. Morris, and H. E. Thompson, “Decision problems under risk and chance constrained programming: Dilemmas in the transition,” *Management Science*, vol. 27, no. 6, pp. 698–716, 1981.

[20] A. Charnes and W. Cooper, “Response to “decision problems under risk and chance constrained programming: Dilemmas in the transition”,” *Management Science*, vol. 29, no. 6, pp. 750–753, 1983.

[21] A. J. Hogan, J. G. Morris, and H. E. Thompson, “Reply to Professors Charnes and Cooper Concerning Their Response to “Decision Problems Under Risk and Chance Constrained Programming”,” *Management Science*, vol. 30, no. 2, pp. 258–259, 1984.

[22] R. Jagannathan, "Use of sample information in stochastic recourse and chance-constrained programming models," *Management science*, vol. 31, no. 1, pp. 96–108, 1985.

[23] I. H. Lavalle, "On information-augmented chance-constrained programs," *Operations research letters*, vol. 4, no. 5, pp. 225–230, 1986.

[24] I. H. LaValle, "Note—Response to "Use of Sample Information in Stochastic Recourse and Chance-Constrained Programming Models": On the "Bayesability" of CCP's," *Management science*, vol. 33, no. 10, pp. 1224–1228, 1987.

[25] R. F. Nau, "Note—blau's dilemma revisited," *Management science*, vol. 33, no. 10, pp. 1232–1237, 1987.

[26] L. Janson, E. Schmerling, and M. Pavone, "Monte carlo motion planning for robot trajectory optimization under uncertainty," in *Robotics Research: Volume 2*, A. Bicchi and W. Burgard, Eds. Cham: Springer International Publishing, 2018, pp. 343–361, ISBN: 978-3-319-60916-4.

[27] N. E. Du Toit and J. W. Burdick, "Probabilistic Collision Checking With Chance Constraints," *IEEE Transactions on Robotics*, vol. 27, no. 4, pp. 809–815, Aug. 2011.

[28] M. P. Vitus, W. Zhang, and C. J. Tomlin, "A hierarchical method for stochastic motion planning in uncertain environments," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2012, pp. 2263–2268.

[29] A. Mesbah, S. Streif, R. Findeisen, and R. D. Braatz, "Stochastic nonlinear model predictive control with probabilistic constraints," in *2014 American Control Conference*, Jun. 2014, pp. 2413–2419.

[30] M. Ilg, J. Rogers, and M. Costello, "Projectile monte-carlo trajectory analysis using a graphics processing unit," in *AIAA Atmospheric Flight Mechanics Conference*, 2011.

[31] P. Dutta and R. Bhattacharya, "Hypersonic State Estimation Using the Frobenius-Perron Operator," *Journal of Guidance, Control, and Dynamics*, vol. 34, no. 2, pp. 325–344, 2011.

[32] A. Halder and R. Bhattacharya, "Dispersion Analysis in Hypersonic Flight During Planetary Entry Using Stochastic Liouville Equation," *Journal of Guidance, Control, and Dynamics*, vol. 34, no. 2, pp. 459–474, 2011.

[33] P. Dutta, A. Halder, and R. Bhattacharya, "Uncertainty quantification for stochastic nonlinear systems using Perron-Frobenius operator and Karhunen-Loève expan-

sion," in *2012 IEEE International Conference on Control Applications*, Oct. 2012, pp. 1449–1454.

[34] J. J. Meyers, A. M. Leonard, J. D. Rogers, and A. R. Gerlach, "Koopman Operator Approach to Optimal Control Selection Under Uncertainty," in *Proceedings of the 2019 American Control Conference, 2019*, 2019.

[35] A. Leonard, J. Rogers, and A. Gerlach, "Koopman operator approach to airdrop mission planning under uncertainty," *Journal of Guidance, Control, and Dynamics*, Jul. 2019.

[36] I. Mezić and A. Banaszuk, "Comparison of systems with complex behavior," *Physica D: Nonlinear Phenomena*, vol. 197, no. 1, pp. 101–133, 2004.

[37] M. O. Williams, I. G. Kevrekidis, and C. W. Rowley, "A data–driven approximation of the koopman operator: Extending dynamic mode decomposition," *Journal of Nonlinear Science*, vol. 25, no. 6, pp. 1307–1346, Dec. 2015.

[38] S. Klus, P. Koltai, and C. Schütte, "On the numerical approximation of the Perron-Frobenius and Koopman operator," *Journal of Computational Dynamics*, vol. 3, no. 1, pp. 51–79, 2016, cited By 31.

[39] Steven L. Brunton and Binbni W. Brunton and Joshua L. Proctor and J. Nathan Kutz, "Koopman Invariant Subspaces and Finite Linear Representations of Nonlinear Dynamical Systems for Control," *PloS One*, vol. 11, no. 2, 2016.

[40] M. Korda and I. Mezic, "Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control," *Automatica*, vol. 93, pp. 149–160, 2018.

[41] I. Abraham and T. D. Murphey, "Active Learning of Dynamics for Data-Driven Control Using Koopman Operators," *IEEE Transactions on Robotics*, vol. 35, no. 5, pp. 1071–1083, Oct. 2019.

[42] G. Stockman, "A minimax algorithm better than alpha-beta?" *Artificial Intelligence*, vol. 12, no. 2, pp. 179–196, 1979.

[43] M. Saks and A. Wigderson, "Probabilistic boolean decision trees and the complexity of evaluating game trees," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, 1986, pp. 29–38.

[44] W. Pijls and A. de Bruin, "Game tree algorithms and solution trees," *Theoretical computer science*, vol. 252, no. 1-2, pp. 197–215, 2001.

[45] R. Washington, "Incremental markov-model planning," in *Proceedings Eighth IEEE International Conference on Tools with Artificial Intelligence*, 1996, pp. 41–47.

[46] B. Bonet and H. Geffner, "Planning with incomplete information as heuristic search in belief space," in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, ser. AIPS'00, Breckenridge, CO, USA: AAAI Press, 2000, pp. 52–61, ISBN: 1577351118.

[47] Y. Chen, L. Zhu, C. Lin, H. Zhang, and A. L. Yuille, "Rapid inference on a novel and/or graph for object detection, segmentation and parsing," in *Advances in neural information processing systems*, 2008, pp. 289–296.

[48] N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

[49] A. Bagchi and A. Mahanti, "Admissible heuristic search in and/or graphs," *Theoretical Computer Science*, vol. 24, no. 2, pp. 207–219, 1983.

[50] A. Mahanti and A. Bagchi, "And/or graph heuristic search methods," *J. ACM*, vol. 32, no. 1, pp. 28–51, Jan. 1985.

[51] D. S. Nau, V. Kumar, and L. Kanal, "General branch and bound, and its relation to a* and ao*," *Artificial Intelligence*, vol. 23, no. 1, pp. 29–58, 1984.

[52] P. Chakrabarti, S. Ghose, and S. DeSarkar, "Admissibility of AO* when heuristics overestimate," *Artificial Intelligence*, vol. 34, no. 1, pp. 97–113, 1987.

[53] P. Jiménez and C. Torras, "An efficient algorithm for searching implicit and/or graphs with cycles," *Artificial Intelligence*, vol. 124, no. 1, pp. 1–30, 2000.

[54] E. A. Hansen and S. Zilberstein, "Lao*: A heuristic search algorithm that finds solutions with loops," *Artificial Intelligence*, vol. 129, no. 1, pp. 35–62, 2001.

[55] P. Santana, S. Thiébaux, and B. Williams, "Rao*: An algorithm for chance-constrained pomdp's," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16, Phoenix, Arizona: AAAI Press, 2016, pp. 3308–3314.

[56] X. Huang, A. Jasour, M. Deyo, A. Hofmann, and B. C. Williams, "Hybrid risk-aware conditional planning with applications in autonomous vehicles," in *2018 IEEE Conference on Decision and Control (CDC)*, 2018, pp. 3608–3614.

[57] X. Huang, S. Hong, A. Hofmann, and B. C. Williams, "Online risk-bounded motion planning for autonomous vehicles in dynamic environments," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 29, no. 1, pp. 214–222, Jul. 2019.

[58] S. Zickler and M. Veloso, "Variable level-of-detail motion planning in environments with poorly predictable bodies," in *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, NLD: IOS Press, 2010, pp. 189–194, ISBN: 9781607506058.

[59] C. Boutilier, R. Dearden, and M. Goldszmidt, "Stochastic dynamic programming with factored representations," *Artificial Intelligence*, vol. 121, no. 1, pp. 49–107, 2000.

[60] Z. Feng, R. Dearden, N. Meuleau, and R. Washington, "Dynamic programming for structured continuous markov decision problems," in *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, ser. UAI '04, Banff, Canada: AUAI Press, 2004, pp. 154–161, ISBN: 0974903906.

[61] J. Pineau, G. Gordon, and S. Thrun, "Point-based value iteration: An anytime algorithm for pomdps," in *Proceedings of the 18th international joint conference on Artificial intelligence*, 2003, pp. 1025–1030.

[62] Mausam, E. Benazera, R. Brafman, N. Meuleau, and E. A. Hansen, "Planning with continuous resources in stochastic domains," in *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, ser. IJCAI'05, Edinburgh, Scotland: Morgan Kaufmann Publishers Inc., 2005, pp. 1244–1251.

[63] B. J. Luders, M. Kothariyand, and J. P. How, "Chance constrained rrt for probabilistic robustness to environmental uncertainty," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, American Institute of Aeronautics and Astronautics, Aug. 2010.

[64] S. D. Bopardikar, B. Englot, A. Speranzon, and J. van den Berg, "Robust belief space planning under intermittent sensing via a maximum eigenvalue-based bound," *The International Journal of Robotics Research*, vol. 35, no. 13, pp. 1609–1626, 2016. eprint: https://doi.org/10.1177/0278364916653816.

[65] V. Indelman, L. Carlone, and F. Dellaert, "Planning in the continuous domain: A generalized belief space approach for autonomous navigation in unknown environments," *The International Journal of Robotics Research*, vol. 34, no. 7, pp. 849–882, 2015. eprint: https://doi.org/10.1177/0278364914561102.

[66] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta Numerica*, vol. 13, pp. 147–269, 2004.

[67] A. Somani, N. Ye, D. Hsu, and W. S. Lee, "Despot: Online pomdp planning with regularization," in *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., vol. 26, Curran Associates, Inc., 2013.

[68] C. B. Browne *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[69] B. J. Ayton and B. C. Williams, *Vulcan: A monte carlo algorithm for large chance constrained mdps with risk bounding functions*, 2018. arXiv: 1809.01220 [cs.AI].

[70] H. Yetkin, J. McMahon, N. Topin, A. Wolek, Z. Waters, and D. J. Stilwell, "Online planning for autonomous underwater vehicles performing information gathering tasks in large subsea environments," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2019, pp. 6354–6361.

[71] S. A. Smolyak, "Quadrature and interpolation formulas for tensor products of certain classes of functions," in *Doklady Akademii Nauk*, Russian Academy of Sciences, vol. 148, 1963, pp. 1042–1045.

[72] F. Heiss and V. Winschel, "Likelihood approximation by numerical integration on sparse grids," *Journal of Econometrics*, vol. 144, no. 1, pp. 62–80, 2008.

[73] K. L. Judd, L. Maliar, S. Maliar, and R. Valero, "Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain," *Journal of Economic Dynamics and Control*, vol. 44, no. C, pp. 92–123, 2014.

[74] *A New Sparse Grid Based Method for Uncertainty Propagation*, vol. Volume 5: 35th Design Automation Conference, Parts A and B, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Aug. 2009, pp. 1205–1215. eprint: https://asmedigitalcollection.asme.org/IDETC-CIE/proceedings-pdf/IDETC-CIE2009/49026/1205/2776692/1205_1.pdf.

[75] L. van den Bos, B. Koren, and R. Dwight, "Non-intrusive uncertainty quantification using reduced cubature rules," *Journal of Computational Physics*, vol. 332, pp. 418–445, 2017.

[76] G. Avila and T. Carrington, "Solving the schroedinger equation using smolyak interpolants," *The Journal of Chemical Physics*, vol. 139, no. 13, p. 134 114, 2013. eprint: https://doi.org/10.1063/1.4821348.

[77] N. Agarwal and N. R. Aluru, "Weighted smolyak algorithm for solution of stochastic differential equations on non-uniform probability measures," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 11, pp. 1365–1389, 2011. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.3019.

[78] C. Schillings and C. Schwab, "Sparse, adaptive smolyak quadratures for bayesian inverse problems," *Inverse Problems*, vol. 29, no. 6, p. 065 011, 2013.

[79] In *Methods of Numerical Integration (Second Edition)*, P. J. Davis and P. Rabinowitz, Eds., Second Edition, Academic Press, 1984, pp. 605–612, ISBN: 978-0-12-206360-2.

[80] H. Xiao and Z. Gimbutas, "A numerical algorithm for the construction of efficient quadrature rules in two and higher dimensions," *Computers and Mathematics with Applications*, vol. 59, no. 2, pp. 663–676, 2010.

[81] E. K. Ryu and S. P. Boyd, "Extensions of gauss quadrature via linear programming," *Foundations of Computational Mathematics*, vol. 15, no. 4, pp. 953–971, 2015.

[82] V. Keshavarzzadeh, R. M. Kirby, and A. Narayan, "Numerical integration in multiple dimensions with designed quadrature," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, A2033–A2061, 2018. eprint: https://doi.org/10.1137/17M1137875.

[83] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European Conference on Machine Learning*, Springer, 2006, pp. 282–293.

[84] D. Silver *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 `[cs.AI]`.

[85] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower bounding klondike solitaire with monte-carlo planning," in *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'09, Thessaloniki, Greece: AAAI Press, 2009, pp. 26–33, ISBN: 9781577354062.

[86] R. Bohlin and L. E. Kavraki, "Path planning using lazy prm," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, IEEE, vol. 1, 2000, pp. 521–528.

[87] G. Sánchez and J.-C. Latombe, "On delaying collision checking in prm planning: Application to multi-robot coordination," *The International Journal of Robotics Research*, vol. 21, no. 1, pp. 5–26, 2002. eprint: https://doi.org/10.1177/027836402320556458.

[88] J. Denny, K. Shi, and N. M. Amato, "Lazy toggle prm: A single-query approach to motion planning," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 2407–2414.

[89] B. Cohen, M. Phillips, and M. Likhachev, “Planning single-arm manipulations with n-arm robots,” in *Proceedings of Robotics: Science and Systems*, Berkeley, USA, Jul. 2014.

[90] K. Hauser, “Lazy collision checking in asymptotically-optimal motion planning,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 2951–2957.

[91] D. Kim, Y. Kwon, and S.-e. Yoon, “Adaptive lazy collision checking for optimal sampling-based motion planning,” in *2018 15th International Conference on Ubiquitous Robots (UR)*, 2018, pp. 320–327.

[92] A. Mandalika, S. Choudhury, O. Salzman, and S. Srinivasa, “Generalized lazy search for robot motion planning:interleaving search and edge evaluation via event-based toggles,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 29, no. 1, pp. 745–753, May 2021.

[93] J. Lim, S. Srinivasa, and P. Tsiotras, *Lazy lifelong planning for efficient replanning in graphs with expensive edge evaluation*, 2021. arXiv: 2105.12076 [cs.RO].

[94] R. M. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, First. CRC Press, Mar. 1994, ISBN: 9780849379819.

[95] J. K. Moore and A. van den Bogert. “opty: Software for trajectory optimization and parameter identification using direct collocation.” (Jun. 2017).

[96] J. Berndt *et al.* “JSBSim.” (2020).

[97] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, “Practical Search Techniques in Path Planning for Autonomous Driving,” in *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics*, AAAI, Chicago, USA, Jun. 2008.

[98] M. W. Wilson, “A general algorithm for nonnegative quadrature formulas,” *Mathematics of Computation*, vol. 23, no. 106, pp. 253–258, 1969.

[99] G. Gutow and J. D. Rogers, “And/or search techniques for chance constrained motion primitive path planning,” *Robotics and Autonomous Systems*, vol. 149, p. 103 991, 2022.

[100] T. N. Patterson, “The optimum addition of points to quadrature formulae,” *Mathematics of Computation*, vol. 22, no. 104, pp. 847–856, 1968.

[101] G. Gutow and J. D. Rogers, “Koopman operator method for chance-constrained motion primitive planning,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 1572–1578, 2020.

VITA

Geordan Mihalick Gutow graduated from Wisconsin's Oshkosh North High School in 2014 before attending Johns Hopkins University. While studying there he participated in the Baja SAE competition team and conducted undergraduate research under Kaliat Ramesh. He graduated in 2018 with a Bachelor's in Mechanical Engineering and enrolled in the Robotics PhD program at the Georgia Institute of Technology. His research interests include artificial decision making, dynamic systems modeling, and applications of robotics to aerospace.