MACHINE LEARNING AND DYNAMIC PROGRAMMING ALGORITHMS FOR MOTION PLANNING AND CONTROL

A Dissertation
Presented to
The Academic Faculty

by

Oktay Arslan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Robotics

School of Aerospace Engineering
Georgia Institute of Technology
December 2015

MACHINE LEARNING AND DYNAMIC PROGRAMMING ALGORITHMS FOR MOTION PLANNING AND CONTROL

Approved by:

Professor Panagiotis Tsiotras, Advisor
School of Aerospace Engineering
Georgia Institute of Technology

Professor Frank Dellaert
School of Interactive Computing
Georgia Institute of Technology

Professor Eric Feron
School of Aerospace Engineering
Georgia Institute of Technology

Professor Le Song
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Evangelos Theodorou
School of Aerospace Engineering
Georgia Institute of Technology

Professor Sertac Karaman
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology

Date Approved: 25 September 2015

Ji bo bav û dayikên min Muhyeddin Arslan û Perîxan Arslan, xwişk û brayên min Emine, Nuriye, Leyla, Dilan, Yunus, Ferhad, Newroz, Evîn, Bahar, apênmin Abdulmecid Arslan û Şemseddin Arslan û ji bo bîranîna dapîra min Duri Arslan.

ACKNOWLEDGEMENTS

I would like to thank to several people who contributed and extended their valuable support to the successful completion of this work. This dissertation would not have been possible without their unconditional guidance and support. First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Panagiotis Tsiotras, for his invaluable guidance throughout the research and encouraging me for being fearless in the face of adversity. His passion for understanding the engineering, pursuit of excellence in technical writing, and rigor mathematics, have deeply inspired me and impacted my scientific education. I am particularly thankful to him for introducing me to the fields of aerospace engineering and robotics and giving me the freedom to pursue my research interests. It was a great pleasure to work with him as his doctoral student, which not only unlocked many opportunities but also inspired my enthusiasm towards the pursuit to excellence and elegance in engineering.

I wish to thank to the members of my committee Prof. Eric Feron, Prof. Evangelos A. Theodorou, Prof. Frank Dellaert, Prof. Le Song and Prof. Sertac Karaman for devoting their time to serve on my dissertation reading committee and giving insightful comments to improve this dissertation. Prof. Eric Feron always inspired me to follow my passion for control and robotics and helped me to explore the opportunities at Aurora Flight Sciences Corporation. I am grateful for his guidance and motivation throughout my graduate life. I would like to thank Prof. Sertac Karaman for agreeing to serve on my committee as an external member. He possesses an immense range of knowledge and expertise in robotics and motion planning. Therefore, he has been not only a great reviewer but also a great friend since the first day we met during our undergraduate years. Also, I would like to thank to my former advisor Prof. Gokhan Inalhan for his endless support and guidance

which eventually led me coming to the United States for my doctoral studies.

During my studies at Georgia Tech, I had the privilege to take courses of high quality from leading experts in the field. In particular, I would like to express my sincere gratitude towards Prof. Panagiotis Tsiotras (for his courses on optimal guidance and control and advanced nonlinear control), Prof. Wassim Michael Haddad (for his courses on robust and nonlinear control), Prof. Olivier Bauchau (for his course on advanced topics in classical dynamics), Prof. Eric Johnson (for his courses on advanced flight dynamics and Kalman filtering), late Prof. Mike Stilman (for his courses on robot intelligence and humanoid robotics), Prof. Magnus Egerstedt (for his course on networked control systems), Prof. Jeff Shamma (for his course on game theory and multiagent systems), Prof. Aaron Bobick (for his courses on computer vision and pattern recognition), and Prof. Arkadi Nemirovski (for his courses on linear and nonlinear optimization).

I wish to thank all labmates, Ioannis Exarchos, Dr. Nuno Filipe, Dr. Efstathios Bakolas, Dr. Yiming Zhao, Spyridon Zafeiropoulos, Wei Sun, Dr. Raghvendra V. Cowlagi, Dr. Giuseppe Di Mauro, Daniel Kuehme, Florian Hauer, Daemin Cho, and Imon Chakraborty, who have been great and considerate friends with me during the years at Georgia Tech. Also, I have had the chance to meet with many more smart people: Kocher Arslan, Dr. Teymur Sadikhov, Timothée Cazenave, Dr. Mehrdad Pakmer, Yunpeng Pan, Kaivalya Bakshi, Vivek Vittaldev, Dr. Hancao Li, Dr. Behnood Gholami, Dr. Zohaib Mian and Dr. Timothy Wang, to name a few. The intellectual discussions with them have been just as enriching and fun activities we have had made my life more enjoyable. In particular, I was fortunate to live together with my cousin Kocher Arslan during all years at Georgia Tech. We had so many adventures and help each other against hardship of being far away from our families. Also, I have enjoyed the good fortune of working out with Ioannis Exarchos and exploring the hidden gems in Atlanta with him. And, special thanks to Dr. Teymur Sadikhov for his ample and kind support in overcoming the hurdles of my life, he has always been ready to help whenever I need; his girlfriend Sabina Karimova has always been the perfect hostess.

I would like to thank to Prof. Emilio Frazzoli for hosting me in his laboratory during my stay at MIT, Dr. James Paduano for giving me the opportunity to work on Autonomous Aerial Cargo/Utility System (AACUS) project at Aurora Flight Sciences, and Dr. Stefano Di Cairano and Dr. Karl Berntorp for giving me the opportunity to work at Mitsubishi Electric Research Laboratories, and Dr. Navid Dadkhah for his invaluable support, being a great co-worker, a friend and a roommate.

I would also like to take this opportunity to thank to Prof. Jechiel Jagoda, Anita M. Carter and Daurette L. Joseph and all other academic staff of School of Aerospace Engineering for their invaluable support and patience.

I would like to express my most sincere gratitude to my parents Muhyettin Arslan and Perihan Arslan, and to my siblings Emine, Nuriye, Leyla, Dilan, Yunus, Ferhat, Nevruz, Evin, Bahar for their unconditional love, patience, and support. Without their sacrifice and inspiration, I could not have achieved this educational pinnacle. Lastly, but by no means least, I am extremely grateful and indebted to my uncles Abdulmecit Arslan and Semsettin Arslan for their support and continuous motivation to follow my passion for science and engineering. With tremendous pride and appreciation I dedicate this work to my family, my uncles and my late grandmother Duri Arslan.

And finally, I am deeply grateful to my zeytin gözlüm Janet James for being considerate, providing encouragement and support whenever I need during the cold winter days in Boston.

Contents

List of Tables

List of Figures

SUMMARY

Robot motion planning is one of the central problems in robotics, and has received considerable amount of attention not only from roboticists but also from the control and artificial intelligence (AI) communities. Despite the different types of applications and physical properties of robotic systems, many high-level tasks of autonomous systems can be decomposed into subtasks which require point-to-point navigation while avoiding infeasible regions due to the obstacles in the workspace. This dissertation aims at developing a new class of sampling-based motion planning algorithms that are fast, efficient and asymptotically optimal by employing ideas from Machine Learning (ML) and Dynamic Programming (DP). First, we interpret the robot motion planning problem as a form of a machine learning problem since the underlying search space is not known a priori, and utilize random geometric graphs to compute consistent discretizations of the underlying continuous search space. Then, we integrate existing DP algorithms and ML algorithms to the framework of sampling-based algorithms for better exploitation and exploration, respectively. We introduce a novel sampling-based algorithm, called $\mathrm{RRT}^{\#}$, that improves upon the well-known RRT^{*} algorithm by leveraging value and policy iteration methods as new information is collected. The proposed algorithms yield provable guarantees on correctness, completeness and asymptotic optimality. We also develop an adaptive sampling strategy by considering exploration as a classification (or regression) problem, and use online machine learning algorithms to learn the relevant region of a query, i.e., the region that contains the optimal solution, without significant computational overhead. We then extend the application of sampling-based algorithms to a class of stochastic optimal control problems and problems with differential constraints. Specifically, we introduce the Path Integral

- RRT algorithm, for solving optimal control of stochastic systems and the CL-RRT$^{\#}$ algorithm that uses closed-loop prediction for trajectory generation for differential systems. One of the key benefits of CL-RRT$^{\#}$ is that for many systems, given a low-level tracking controller, it is easier to handle differential constraints, so complex steering procedures are not needed, unlike most existing kinodynamic sampling-based algorithms. Implementation results of sampling-based planners for route planning of a full-scale autonomous helicopter under the Autonomous Aerial Cargo/Utility System Program (AACUS) program are provided.

Chapter I

INTRODUCTION

1.1 Motivation

Robot motion planning is one of the canonical problems in the field of robotics, and it has received an extensive amount of attention not only from roboticists but also from the control and artificial intelligence (AI) communities [39, 79, 81]. Despite the different types of applications and physical properties of robotic systems, many high-level tasks of autonomous systems can be decomposed into subtasks that require point-to-point navigation of the robot while avoiding some infeasible regions due to the obstacles in the workspace. Furthermore, robot motion planning has found many successful applications beyond robotics such as computer animation, computational biology, virtual prototyping, and game AI [32, 41, 45, 79, 88].

Although there are variations of the problem depending on the applications and constraints, loosely speaking, the robot motion planning can be stated as follows. Given a complete description of the geometry of a robot and of the workspace, an initial state, and a goal region, the motion planning problem is to find a control input that can move the robot from its initial state to the goal region, while obeying its differential constraints, i.e., the dynamics of the system, and not colliding with obstacles in the environment. This problem goes by different names in the community, depending on the constraints that the robot is subject to. For instance, a simpler variant of the problem is called the *mover's problem* [104], or the *piano movers' problem* [107, 108]. In this widely-studied problem, the robot is assumed to be a rigid body whose configuration has finite parameterization and is able to move freely, i.e., there are no differential constraints. The goal is to find a collision-free path for the robot to move from an initial configuration to a goal configuration. The

mover's problem is then generalized and extended for the motion planning of the articulated robots, e.g., the robotic manipulators in manufacturing assembly lines. This problem is called the *generalized mover's problem* and the robot is modeled as the union of multiple rigid bodies that are mounted with freely-moving joints [104, 105]. In this thesis, motion planning problems without differential constraints are called *(geometric) path planning problems*. Path planning problems are ubiquitous and arise in many practical applications beyond robotics. Many algorithmic approaches are developed and the complexity of these problems have been well analyzed (see [89]).

An important milestone was achieved in a landmark paper by Lozano-Prez, where planning problems were abstracted into a class of problems, called *spatial planning*, in an unified way by using the notion of the *configuration space* [89]. The main benefit of the abstraction is that a robot with a complex geometric shape is represented with a single point in the configuration space. The dimension of the configuration space is the number of degrees of freedom of a robot, or the minimum number of parameters needed to specify its configuration.

1.2 Thesis Statement and Contributions

The contributions of this thesis span several fields. First, we develop a novel sampling-based algorithm, called the $\mathrm{RRT}^{\#}$ algorithm along with several variants, that leverage dynamic programming for better exploitation and machine learning algorithms for better exploration. Next, we proposed an algorithm that uses policy iteration for exploitation which can be massively parallelized. The proposed algorithms yield provable guarantees on correctness, completeness and asymptotic optimality. We also extend the applications of sampling-based algorithms to a class of stochastic optimal control problems. We introduce a novel algorithm, called the path integral RRT (PI-RRT) algorithm, for optimal control of stochastic systems. We finally introduce a sampling-based algorithm that uses closed-loop prediction for trajectory generation. One of the key benefits of the latter algorithm is that it

can be applied to many systems having complex dynamics and does not require a complex steering procedure unlike the most of the existing kinodynamic sampling-based algorithms. Last but not least, in order to encourage knowledge transition from academia to industry, we apply a variant of the proposed algorithms for high-level route planning of a full-scale autonomous rotorcraft. The developed route planner is tested extensively in simulations and actual flight tests.

1.2.1 Exploitation using Dynamic Programming Algorithms

We analyze the existing asymptotically optimal algorithms, e.g., RRG, RRT^*, and a massively version of RRT^* [27], and reveal the connection between their asymptotic optimality properties and dynamic programing principles. It is shown that the rewiring step introduced in the RRT^* algorithm is essentially a form of policy iteration algorithm. We then propose a novel algorithm, called the $\mathrm{RRT}^{\#}$, that is shown to be asymptotically optimal, while achieving faster convergence. We develop a more elaborate replanning procedure that implements asynchronous value iteration algorithm in order to propagate new information better [8, 9, 11]. Next, we consider different forms of DP, e.g., policy iteration, to develop highly parallelizable motion planners [15].

1.2.2 Exploration using Machine Learning Algorithms

Most of the sampling-based algorithms implement a form rejection sampling owing to their simplicity. We first show that the samples that are in collision can also provide useful information about the topology of the underlying search space, and hence they can be leveraged to guide the future samples towards the free space. We then propose a machine learning (ML)-inspired approach to estimate the relevant region of a motion planning problem during the exploration phase of sampling-based path-planners. The algorithm guides the exploration, so that it draws more samples from the relevant region as the number of iterations increases. The approach works in two steps: first, it predicts if a given sample is collision-free (classification phase) without calling the collision-checker, and it then estimates if it

is a promising sample, i.e., if it has the potential to improve the current best solution (regression phase), without solving the local steering problem. We show in Chapter 6 that the proposed exploration strategy can be seamlessly integrated to the $\mathrm{RRT}^{\#}$ algorithm. Numerical simulations demonstrate the efficiency of the proposed approach [10, 11, 14].

1.2.3 Stochastic Motion Planning via Sampling-based Algorithms

We consider optimal control of dynamical systems which are represented by nonlinear stochastic differential equations. It is well-known that the optimal control policy for this problem can be obtained as a function of a value function that satisfies a nonlinear partial differential equation, namely, the Hamilton-Jacobi-Bellman equation. This nonlinear PDE must be solved backwards in time, but this computation is intractable for large scale systems. Under certain assumptions, and after applying a logarithmic transformation, an alternative characterization of the optimal policy can be given in terms of a path integral. Path Integral (PI) based control methods have recently been shown to provide an elegant solution to a broad class of stochastic optimal control problems. One of the implementation challenges with this formalism is the computation of the expectation of a cost functional over the trajectories of the unforced dynamics. Computing such expectation over trajectories that are sampled uniformly may induce numerical instabilities due to the exponentiation of the cost. Therefore, sampling of low-cost trajectories is essential for the practical implementation of PI-based methods. In this thesis, we use incremental sampling-based algorithms to sample *useful* trajectories from the unforced system dynamics, and make a novel connection between the RRT algorithm and information-theoretic stochastic optimal control. We show in Chapter 7 the results from the numerical implementation of the proposed approach to dynamical systems that are injected with different intensity levels of noise. [7]

1.2.4 Motion Planning using Closed-Loop Prediction

We analyze variants of the RRT algorithm that were successfully implemented in real robotic applications, e.g., the CL-RRT algorithm that used by Team MIT at DARPA Urban Challenge. Unlike the kinodynamic RRT algorithm, the CL-RRT algorithm does not sample randomly control inputs but instead samples reference trajectories that are inputted to a low-level tracking controller designed beforehand. Each sampled reference trajectory is then associated with a state trajectory which is computed via *closed-loop prediction*, that is, the reference trajectory is sent to the tracking controller, and the closed-loop system is simulated forward in time. We leverage the idea of closed-loop prediction, and propose the first asymptotically optimal algorithm, called the CL-RRT$^{\#}$, that uses closed-loop prediction for trajectory generation. The proposed algorithm incrementally grows a graph in the space of reference trajectories and search over alternative reference trajectories that yields lower cost state trajectories. Unlike the kinodynamic variants of asymptotically optimal algorithms, the CL-RRT$^{\#}$ does not require a complex steering procedure, and therefore it is trivial to extend it to systems having complex dynamics given a low-level tracking controller. We show in Chapter 8 that the proposed algorithm can be easily applied to a wide range of dynamical systems yielding satisfactory performance [3].

1.2.5 High-level Route Planning of an Autonomous Rotorcraft

Unmanned Aerial Systems (UAS) have become ubiquitous in many civilian and military applications due to advances in computation and sensing technologies. Recently, the Office of Naval Research (ONR) has announced a five-year program, called Autonomous Aerial Cargo Utility System (AACUS), with primary focus on the development of unmanned Vertical Take Off and Landing (VTOL) air systems that are capable of performing cargo and delivery operations [42]. As a part of this thesis, we develop an RRT$^{\#}$-based algorithm for high-level route planning of a full-scale autonomous helicopter. The goal is to develop a fast route planner that is capable of finding paths that leverage the capabilities of the

autonomous helicopter, e.g., the path have to be smooth, not have sharp turns, and are compliant with the restrictions enforced by the aviation administration, e.g., staying within predefined safe-flight spaces, and avoid no-fly zones. The $\mathrm{RRT}^{\#}$ algorithm together with an efficient collision checker is integrated to the route planner of the helicopter and is tested in several missions during flight tests. Extensive numerical simulations and flight test results demonstrate that the implemented route planner is able to find paths for missions of large operating environments (50km $\times$ 50km $\times$ 10000 ft) within the order of seconds without requiring extensive memory usage [4].

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 provides a survey of the different approaches for the solution of the robotic motion planning problem. Chapter 3 introduces a novel class of sampling-based motion planing algorithms and presents how existing dynamic programming algorithms can be used on incrementally growing random geometric graphs. Chapter 6 explains novel adaptive sampling strategies that build upon machine learning algorithms and discusses their integration to sampling-based motion planning algorithms. Chapter 7 extends applications of the RRT algorithm within path integral framework for solution of a class of stochastic optimal control problems. Chapter 8 presents the first asymptotically optimal sampling-based algorithm that uses closed-loop prediction for trajectory generation. Chapter 9 demonstrates the application of the proposed methodology on a real-world planning problem, specifically explains the implementation of the $\mathrm{RRT}^{\#}$ algorithm for high-level route planning of a helicopter. Finally, in Chapter 10 we conclude the thesis with some remarks and give some potential research avenues for future work.

Chapter II

RELATED WORK

2.1 Exact Methods

These methods are most extensively applied to the motion planning problem during 1980s. The main idea of this approach is to partition the robot's free space into a collection of non-overlapping cells and to construct a connectivity graph representing cell adjacency. One of the first exact cell decomposition methods for solving the general motion planning problem was given in [108]. These methods are nice and come with completeness guarantees, since they solve the problem without resorting to an approximation of the search space. However, exact methods are not scalable and need to perform on an exponentially growing number of cells as the dimension of the search space and the number of obstacles increase.

2.2 Graph Seach-based Planners

Graph search-based planners first generate a graph representation of the robot motion planning problem, and te constructed graph encodes the collision-free configurations (or states) of the robot as its nodes and the feasible robot motions and transitions between each pair of configurations as its edges. Once a reasonable graph is computed, then a graph search algorithm is employed to compute the optimal path (e.g., Dijkstra's graph search algorithm [44]). Unfortunately, the main limitation of these planners is poor scalability to planning problems with high-dimensional search space. The memory and computation time requirements of the Dijkstra's algorithm become very expensive on high-resolution graphs due to its uninformed node expansion strategy. To remedy these drawbacks, several heuristic-based graph search algorithms have been proposed to guide the computations towards the relevant region of the planning problem, e.g., the region that is a smaller subset

of the search space and contains the shortest path.

For example, the A* algorithm uses an admissible and consistent cost-to-go function approximation and expands the nodes from frontier set based on their f-value (summation of the cost-to-come and heuristic value of a node). This helps the A* algorithm to restrict the search effort to a provably minimum number of nodes [52]. Similarly, the Anytime Repairing A* (ARA*) algorithm computes a suboptimal solution with a known bound very quickly by using highly inadmissible heuristics at the beginning of the search [86]. Then, by using better heuristics, the quality of initial suboptimal solution is iteratively improved during subsequent iterations given additional computation time The anytime capability of ARA* allows planning engineers to make a trade-off between solution quality and search effort on different planning problems, that is, we can terminate the algorithm at any stage and retrieve the current best solution. Several extensions of the A* algorithm with re-planning capabilities were developed in order to solve the planning problems in dynamics environments, i.e., edge connections and weights in the graph changes. Notable examples include the LPA* algorithm [71], the D* algorithm [112], and the D* algorithm [70]. These algorithms leverage the previously computed solution and attempt to repair it when the graph is updated due to some changes in the environment.

2.3 Sampling-based Motion Planners

Sampling-based motion planners avoid construction of an explicit representation of the configuration space due to the burden of high computation and memory requirements. Instead, these methods incrementally build discrete data structures, e.g., a tree or a graph, that approximately represent the configuration space and then use some search techniques to extract the solution encoded in the discrete representation [68, 69, 82]. Sampling-based planners are proven to be very successful in many practical applications [87]. Therefore, they have become popular since the early 1990s and are considered the state-of-the-art results for motion planning in high-dimensional space up to date [22, 101, 113, 115, 123].

Sampling-based methods are very efficient in terms of memory requirements since they avoid explicit construction of the collision-free space, as opposed to most exact cell decomposition algorithms. However, these methods sacrifice completeness guarantees and are not able to distinguish infeasible queries. Instead, they come with some relaxed notion of completeness and provide *probabilistic completeness* guarantees, that is, for sampling-based planners, the probability that the planner fails to compute a solution, if one exists, decays to zero as the number of samples approaches infinity [19, 57, 67, 77]. Furthermore, the probability of failure diminishes exponentially, under the assumption that environment has good "visibility" properties [19, 57].

To date, there are two most popular classes of sampling-based motion planning algorithms, notably, Probabilistic Roadmaps (PRMs) [67, 69] and Rapidly-exploring Random Trees (RRTs) [80, 82]. These algorithms compute a discrete representation of the collision-free space by randomly generating collision-free samples and connecting them to their neighbors. Despite their similarities, the algorithms differ in the way that they build a graph on the collision-free configuration space. Also, the PRM algorithm and its variants are multiple-query methods, whereas the RRT algorithm and its variants are single-query methods.

The PRM algorithm works in two phases: a learning phase and a query phase. In the learning phase, a probabilistic roadmap is constructed and stored as a graph whose nodes correspond to collision-free configurations (the milestones) and edges correspond to collision-free paths between these configurations. These paths are computed using a simple and fast local planner. In the query phase, for any given start and goal configurations of the robot, the algorithm connects these configurations to the corresponding nearest nodes of the roadmap; the roadmap is then searched for a path joining the initial and goal nodes by a graph search algorithm. The PRM algorithm has been proven to be very successful for motion planning problems in high-dimensional spaces [69] and is probabilistically complete [67]. There has been considerable amount of research effort devoted to improving the

PRM algorithm and its variants [32, 56, 77].

The PRM algorithm is better-suited for solving motion planning problems that have highly structured and static workspace, e.g., robotic manipulators in factory assembly lines. The learning phase, i.e., the construction of a roadmap, usually takes considerable amount of time since it requires extensive number of collision checking. It is worth to building a roadmap only if the same configuration space is expected to be used repeatedly for different queries. However, there are many practical applications that do not require using the same configuration space several times, for instance, a mobile robot that navigates in an unknown environment. Due to the nature of these applications, researchers have developed online incremental sampling-based planning algorithms such as the RRT algorithm, which is a single-query counterpart to the PRM algorithm [55, 82]). Instead of constructing a roadmap, tree-based planners incrementally build a tree and quickly explore the configuration space with a set of rich trajectories. These planners do not require connecting two states exactly and more easily handle systems with differential constraints. The RRT algorithm has been proven to be probabilistically complete [82]. Owing to their success in effectively handling systems with differential constraints, researchers have developed many variants of the basic RRT algorithm and have applied them to applications beyond robotics [26, 30, 31, 41, 48, 124]. Finally, researchers have implemented motion planners that use the RRT algorithm as their core planning algorithm for robotic competition experimental platforms [33, 72, 75, 116].

2.4 Asymptotically Optimal Motion Planners

Recently, popular sampling-based algorithms such as RRT and PRM are proven to be sub-optimal almost surely. Asymptotically optimal variants, such as RRG, PRM* and RRT* have been proposed [65], in order to remedy this undesirable behavior. The seminal work of [65] has sparked a renewed interest to asymptotically optimal probabilistic, sampling-based motion planners. Several variants have been proposed that utilize the original ideas

of [65]; a partial list includes FMT* [61], RRT^X [95] among others. All of these algorithms compute a sparse yet consistent discretization of the underlying continuous search space and extract the solution encoded in the discrete representation. They randomly generate collision-free samples from the search space and connect the points that are near-by to each other, i.e., within in a ball of some radius. The key step is the connection radius which shrinks as the number of points increases, thus avoiding a large number of connections. This improves computational and memory efficiency. Last but not least, it is recently shown that this type of planners can quickly find time-optimal trajectories for manipulators with acceleration limits by using moderate simplification of the underlying dynamics [74].

2.5 Trajectory Optimization

Robot motion planning can be formulated as a trajectory optimization problem within the optimal control framework by defining proper actions, dynamics and cost functional. The solution of such trajectory optimization problem can be characterized by using techniques built upon dynamic programming of Bellman [21] or the maximum principle by Pontryagin [100]. However, solving a planning problem in high-dimensional search space is still an open problem since both approaches have some limitations when solving practical motion planning problems and yield different benefits compare to each other. First, the techniques that leverage dynamic programming need to compute the optimal value function over the entire search space in order to recover the optimal actions. Representing such value function in high-dimensional, continuous state spaces is a tedious task and suffers from the *curse of dimensionality*, that is, any functional representation that uses uniform discretization schemes requires a number of discrete values that is exponential in the dimensionality of the search space [21]. Nonetheless, these technique have some merits on small scale problems since they provide a feedback policy. On the other hand, the techniques based on the maximum principle do not suffer from the curse of dimensionality since they focus on individual trajectories rather than a value function or a policy defined on the entire search

space. Therefore, these techniques can be practically applied to problems in large, continuous state and action spaces. However, the main drawback of such approaches is that the maximum principle provides necessary conditions for the optimal solution and yields open loop policies. Also, in order to recover the optimal actions, these techniques need to solve a two-point boundary value problem by forming the co-state equations, which is not easy to solve in general.

Some practical dynamic programming methods were developed in order to mitigate the curse of dimensionality by concerning only a single trajectory. Popular examples include local trajectory optimizers such as differential dynamic programming (DDP) technique [60]. In DDP, a trajectory of interest is optimized in its local neighborhood in a differential sense by means of a Taylor expansion, and the algorithms runs in a backward-forward scheme to solve several ordinary differential equations which are formed using a second order expansion of the dynamics and cost function, in a manner analogous to Newton's method. Also, some variants of DDP have been proposed using only first order dynamics for efficiency at the expense of losing quadratic convergence rate [120]. There are some other trajectory optimization methods such as CHOMP [103] and STOMP [62] that have been successfully applied to many practical problems. These optimizers can solve problems that have complex differential and task constraints, and are capable of generating smooth trajectories. However, they do have some drawbacks, such as getting stuck in local optima. Also, since hard constraints such as avoiding obstacles are represented as high costs in the cost functional, a locally optimal trajectory is not necessarily feasible, and it may violate hard constraints. Therefore, the performance of trajectory optimizers is highly dependent on a good initial guess and proper tuning of the parameters for the specific problem. CHOMP and STOMP use random restarts to avoid getting stuck at local optima. In practice, sampling-based planners are used to compute good initial guesses and run as complimentary to trajectory optimizers. A recent survey of local trajectory methods can be found in the thesis by Levine [85].

Chapter III

DYNAMIC PROGRAMMING FOR MOTION PLANNING

3.1 Overview of Dynamic Programming

Dynamic programming solves sequential decision-making problems having a finite number of stages. A sequential decision making problem with finite stages is a is a tuple $\langle \mathcal{X}, U, f, g, G, x_{\text{init}} \rangle$ where

- $\mathcal{X}$ is the set of all possible states of the system, i.e., *the state space*;
- U is the set of controls, i.e., *the control space*;
- $f : \mathcal{X} \times U \mapsto \mathcal{X}$ is a transition function, a mapping specifying the next state $f(x, u)$ if control u is executed when the system is in state x;
- $g : \mathcal{X} \times U \mapsto \mathbb{R}$ is a cost function that gives a real value $g(x, u)$ obtained if control u is executed when the system is in state x;
- $G : \mathcal{X} \mapsto \mathbb{R}$ is a cost function that gives a real value $G(x)$ obtained if the final state is x;
- $x_{\text{init}} \in \mathcal{X}$ is the initial state of the system.

We are interested in minimizing the total cost associated with a policy

$$\Pi = \big\{ \{\mu_0, \mu_1, \ldots, \mu_{N-1}\} : \mu_k \in \mathcal{M}, k = 0, 1, \ldots, N-1 \big\},$$

where $\mathcal{M}$ is the set of functions $\mu \colon \mathcal{X} \mapsto U$ defined by

$$\mathcal{M} = \{\mu : \mu(x) \in U(x), \, \forall x \in \mathcal{X}\}.$$

The total cost of a policy $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\} \in \Pi$ over a finite number of stages and starting at the initial state $v_0 = x_{\text{init}}$ is

$$J_\pi(v_0) = G(v_N) + \sum_{k=0}^{N-1} g(v_k, \mu_k(v_k)), \tag{1}$$

where the state sequence $\{v_k\}_{k=1}^N$ is generated by the transition function f under the policy π:

$$v_{k+1} = f(v_k, \mu_k(v_k)), \quad k = 0, 1, \ldots, N-1.$$

The optimal cost function is

$$J^*(x) = \inf_{\pi \in \Pi} J_\pi(x), \quad x \in \mathcal{X}.$$

Given a sequential decision problem of the form (25)-(27), it is well known that the optimal cost function satisfies the following *Bellman equation*:

$$J^*(x) = \inf_{u \in U(x)} \Big\{ g(x,u) + J^*(f(x,u)) \Big\}, \quad \forall x \in \mathcal{X}. \tag{2}$$

The previous optimization results in an optimal policy $\mu^* \in \mathcal{M}$, that is,

$$\mu^*(x) \in \operatorname*{argmin}_{u \in U(x)} \Big\{ g(x,u) + J^*(f(x,u)) \Big\}, \quad \forall x \in \mathcal{X}. \tag{3}$$

Note that if we are given a policy $\mu \in \mathcal{M}$ (not necessarily optimal) we can compute its cost from

$$J_\mu(x) = g(x, \mu(x)) + J_\mu(f(x, \mu(x))), \quad \forall x \in \mathcal{X}. \tag{4}$$

It follows that $J^*(x) = \inf_{\mu \in \mathcal{M}} J_\mu(x), \quad x \in \mathcal{X}$. By introducing the expression

$$H(x, u, J) = g(x,u) + J(f(x,u)), \quad x \in \mathcal{X}, u \in U(x).$$

and letting the operator T_μ for a given policy $\mu \in \mathcal{M}$,

$$(T_\mu J)(x) = H(x, \mu(x), J), \quad x \in \mathcal{X}, \tag{5}$$

we can define the Bellman operator T

$$(TJ)(x) = \inf_{u \in U(x)} H(x, u, J) = \inf_{\mu \in \mathcal{M}} (T_\mu J)(x), \quad x \in \mathcal{X}, \tag{6}$$

which allows us to write the Bellman equation (2) succinctly as follows

$$J^* = TJ^*, \tag{7}$$

and the optimality condition (3) as

$$T_{\mu^*}J^* = TJ^*. \tag{8}$$

This interpretation of the Bellman equation states that J^* is the fixed point of the Bellman operator T, viewed as a mapping from the set of real-valued functions on $\mathcal{X}$ into itself. Also, in a similar way, J_μ, the cost function of the stationary policy μ, is a fixed point of T_μ (see (4)).

There are three different classes of DP algorithms to compute the optimal policy μ^* and the optimal cost function J^*.

Value Iteration (VI)

This algorithm computes J^* by relaxing Eq. (2), starting with some J^0, and generating a sequence $\left\{T^k J\right\}_{k=0}^{\infty}$ via the iteration

$$J^{k+1} = TJ^k \tag{9}$$

The generated sequence converges to the optimal cost function due to the contraction property of the Bellman operator T. This method is an indirect way of computing the optimal policy μ^*, using the information of the optimal cost function J^*.

Policy Iteration (PI)

This algorithm starts with an initial policy stationary μ^0 and generates a sequence of stationary policies μ^k by performing Bellman updates. Given the current policy μ^k, the typical iteration is performed in two steps:

i) **Policy evaluation**: compute J_{μ^k} as the unique solution of the equation

$$J_{\mu^k} = T_{\mu^k} J_{\mu^k}.$$

ii) **Policy improvement**: compute a policy μ^{k+1} that satisfies

$$T_{\mu^{k+1}} J_{\mu^k} = T J_{\mu^k}.$$

Optimistic Policy Iteration (O-PI)

This algorithm works the same as PI, but differs in the policy evaluation step. Instead of solving the system of linear equations exactly in the policy evaluation step, it performs an approximate evaluation of the current policy and uses this information in the subsequent policy improvement step.

3.2 Random Geometric Graphs

The main difference between standard shortest path problems on graphs and sampling-based methods for solving motion planning problems is the fact that in the former case the graph is given a priori, whereas in the latter case the path is constructed on-the-fly by sampling randomly allowable configuration points from $\mathcal{X}_{\text{free}}$ and by constructing the graph $\mathcal{G}$ incrementally, adding one or more vertices at a time. Of course, such an iterative construction raises several questions, such as: is the resulting graph connected? under what conditions one can expect that $\mathcal{G}$ is an accurate representation of $\mathcal{X}_{\text{free}}$? how does discretized the actions/control inputs to move between sampled successor vertices, etc. All these questions have been addressed in a series of recent papers [64, 65] so we will not elaborate further on the graph construction. Suffice it to say that such random geometric graphs (RGGs) can be constructed easily and such graphs have been the cornerstone of the recent emergence of asymptotically optimal sampling based motion planners.

For completeness, and in order to establish the necessary connections between DP algorithms and RRGs, we provide a brief overview of random graphs as they are used in this chapter. For more details, the interested reader can peruse [23] or [98].

In graph theory, a random geometric graph (RGG) is a mathematical object that is usually used to represent spatial networks. RGGs are constructed by placing a collection of

vertices drawn randomly according to a specified probability distribution. These random points constitute the node set of the graph in some topological space. Its edge set is formed via pairwise connections between these nodes if certain conditions (e.g., if their distance according to some metric is in a given range) are satisfied. Different probability distributions and connection criteria yield random graphs of different properties.

An important class of random geometric graphs is the *random r-disc graphs*. Given the number of points n and a nonnegative radius value r, a random r-disc graph in $\mathbb{R}^d$ is constructed as follows: first, n points are independently drawn from a uniform distribution. These points are pairwise connected if and only if the distance between them is less than r. Depending on the radius, this simple model of random geometric graphs possesses different properties as the number of nodes n increases. A natural question to ask is how the connectivity of the graph changes for different values of the connection radius as the number of samples goes to infinity. In the literature, it is shown that the connectivity of the random graph exhibits a phase transition, and a connected random geometric graph is constructed almost surely when the connection radius r is strictly greater than a critical value $r^* = \left\{ \log(n)/(n\zeta_d) \right\}^d$, where ζ_d is volume of the unit ball in $\mathbb{R}^d$. If the connection radius is chosen less than the critical value r^*, then, multiple disconnected clusters occur almost surely as n goes to infinity [98].

Recently, novel connections have been made between motion planning algorithms and the theory of random geometric graphs [65]. These key insights have led to the development of a new class of algorithms which are asymptotically optimal (e.g., RRG, RRT^*, PRM^*). For example, in the RRG algorithm, a random geometric r-disc graph is first constructed incrementally for a fixed number of iterations. Then, a post-search is performed on this graph to extract the encoded solution. The key step is that the connection radius is shrunk as a function of vertices, while still being strictly greater than the critical radius value. By doing so, it is guaranteed to obtain a connected and sparse graph, yet the graph is rich enough to provide asymptotic optimality guarantees. The authors in [65] showed that the

RRG algorithm yields a consisted discretization of the underlying continuous configuration space, i.e., as the number of points goes to infinity, the lowest-cost solution encoded in the random geometric graph converges to the optimal solution embedded in the continuous configuration space with probability one. In the next chapters, we leverage this nice feature of random geometric graphs to get a consistent discretization of the continuous domain of the robot motion planning problem. With the help of random geometric graphs, the robot motion planning problem boils down to a shortest path problem on a discrete graph.

3.3 From RRGs to DP

Let $\mathcal{G} = (V, E)$ denote the graph constructed by the RRG algorithm at some iteration, where V and $E \subseteq V \times V$ are finite sets of vertices and edges, respectively. Based on the previous discussion, $\mathcal{G}$ is connected and all edge costs are positive, which implies that the cost of all the cycles in $\mathcal{G}$ are positive. Using the previous notation, we can define on this graph the sequential decision system (25) where $x' \in \mathtt{succ}(\mathcal{G}, x)$ and transition cost as in (26). Each parent assignment for each node in $\mathcal{G}$ defines a policy. Convergence of DP algorithms is guaranteed and the resulted optimal policy μ is proper.

Chapter IV

MOTION PLANNING USING VALUE ITERATION METHODS

4.1 Problem Formulation

Let $\mathcal{X}$ denote the state space, which is assumed to be an open subset of $\mathbb{R}^d$, where $d \in \mathbb{N}$ with $d \geq 2$. Let the *obstacle region* and the *goal region* be denoted by $\mathcal{X}_{\text{obs}}$ and $\mathcal{X}_{\text{goal}}$, respectively. The obstacle-free space is defined by $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}$. Let the *initial state* be denoted by $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. The neighborhood of a state $x \in \mathcal{X}$ is defined as the open ball of radius $r > 0$ centered at x, that is, $B_r(x) = \{x' \in \mathcal{X} : \|x - x'\| < r\}$. The straight-line segment between given two points $x, x' \in \mathbb{R}^d$ is denoted by $\texttt{Line}(x, x') = \{\theta \in \mathbb{R}, 0 \leq \theta \leq 1 : \theta x + (1 - \theta)x'\}$. Let $\mathcal{G} = (V, E)$ denote a graph, where V and $E \subseteq V \times V$ are finite sets of vertices and edges, respectively. In the sequel, we will use graphs to represent the connections between a (finite) set of points selected randomly from $\mathcal{X}_{\text{free}}$. With a slight abuse of notation, we will use x to denote both a point in the space $\mathcal{X}$ and the corresponding vertex in the graph.

Geometric r-disc graph: Let $V \subset \mathbb{R}^d$ be a finite set, and $r > 0$. A geometric r-disc graph $\mathcal{G}(V; r) = (V, E)$ in d dimensions is an undirected graph with vertex set V and edge set $E = \{(u, v) : \|u - v\| < r\}$.

Successor vertices: Given a vertex $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{succ} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the vertices in V that can be reached from vertex v,

$$\texttt{succ}(\mathcal{G}, v) := \{u \in V : (v, u) \in E\}.$$

Predecessor vertices: Given a vertex $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{pred} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the vertices in V that are the tails of the

edges going into v,

$$\texttt{pred}(\mathcal{G}, v) := \{u \in V : (u, v) \in E\}.$$

Parent vertex: Given a directed graph $\mathcal{G} = (V, E)$ and a vertex $v \in V$, the function $\texttt{parent} : v \mapsto u$ returns a unique vertex $u \in V$ such that $(u, v) \in E$ and $u \in \texttt{pred}(\mathcal{G}, v)$.

Spanning tree: Given a directed graph $\mathcal{G} = (V, E)$, a spanning tree of $\mathcal{G}$ can be defined such that $\mathcal{T} = (V_s, E_s)$, where $V_s = V$ and $E_s = \{(u, v) : (u, v) \in E \text{ and } \texttt{parent}(v) = u\}$.

Edge cost value: Given an edge $e = (u, v) \in E$, the function $\texttt{c} : e \mapsto r$ returns a non-negative real number. Then $\texttt{c}(u, v)$, where $v \in \texttt{succ}(\mathcal{G}, u)$, is the cost incurred by moving from u to v.

Cost-to-come value: Given a vertex $v \in V$, the function $\texttt{g} : v \mapsto r$ returns a non-negative real number r, which is the cost of the path to v from a given initial state $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. We will use $\texttt{g}^*(v)$ to denote the optimal cost-to-come value of the vertex v which can be achieved in $\mathcal{X}_{\text{free}}$.

Heuristic value: Given a vertex $v \in V$, and a goal region $\mathcal{X}_{\text{goal}}$, the function $\texttt{h} : (v, \mathcal{X}_{\text{goal}}) \mapsto r$ returns an estimate r of the optimal cost from v to $\mathcal{X}_{\text{goal}}$; we set $\texttt{h}(v) = 0$ if $v \in \mathcal{X}_{\text{goal}}$. It is an admissible heuristic if it never overestimates the actual cost of reaching $\mathcal{X}_{\text{goal}}$. In this chapter, we always assume that $\texttt{h}$ is an admissible heuristic. It is well known that inadmissible heuristics can be used to speed-up the search, but they may lead to suboptimal paths [97].

We wish to solve the following motion planning problem: Given a bounded and connected open set $\mathcal{X} \subset \mathbb{R}^d$, and the sets $\mathcal{X}_{\text{free}}$ and $\mathcal{X}_{\text{obs}} = \mathcal{X} \backslash \mathcal{X}_{\text{free}}$, and given an initial point $x_{\text{init}} \in \mathcal{X}_{\text{free}}$ and a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$, find the minimum-cost path connecting x_{init} to the goal region $\mathcal{X}_{\text{goal}}$. If no such path exists, then report that no solution is possible.

Next, we present an iterative algorithm that finds the optimal path connecting a sequence of points sampled randomly from $\mathcal{X}_{\text{free}}$. The algorithm is based on ideas similar to those found in the RRT^* algorithm [65], with one important distinction. While the RRT^*

algorithm is based on a local rewiring of the tree after the addition of a new vertex, the proposed algorithm incorporates, at each iteration, a replanning step similar to what is implemented in the LPA* and D* algorithms [70, 71] to efficiently propagate changes in the relevant part of the graph owing to the inclusion of the new vertex. As a result, the proposed algorithm ensures that at each iteration, the optimal path in the current graph is computed.

On the contrary, in the RRT^* algorithm there is no guarantee that the interim path at any intermediate iteration is optimal. Path optimality in RRT^* is ensured only at the limit, as the number of sampled points tends to infinity. Since any algorithm will be terminated after a finite number of iterations, it is important to ensure that at termination, the returned path is optimal, given the available data up to that point. Furthermore, it is important that the computation of the optimal path at each iteration is done efficiently. This means that any prior information from previous iterations is taken into consideration during the next replanning step. In our implementation, this is done by keeping track of the most "promising" vertices in the graph (these are the vertices that can be part of the optimal path) and by updating the cost-to-come values of these vertices as new information becomes available. It is shown that this amounts to implementing a dynamic-type programming step at each iteration, after a suitable reordering of the variables (in order to encode the updated information), similarly to what is done in Gauss-Seidel relaxation methods for the solution of fixed point problems. The details of the proposed algorithm are given in Sections 4.3 and 4.4. First, it is important to review the main ideas behind relaxation methods to iteratively solve a system of equations.

4.2 Overview of Relaxation Methods

Relaxation methods are *iterative* methods for solving systems of equations, including algebraic systems of nonlinear equations and optimization problems in numerical mathematics [25]. These methods can be written informally as

$$x(k+1) = f(x(k)), \qquad k = 0, 1, \ldots, \tag{10}$$

where each $x(k) \in \mathbb{R}^n$ is an n-dimensional vector, and $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ is a vector-valued function. Equation (10) generates a sequence of improving approximate solutions for the equation $x = f(x)$ for any given initial guess $x(0)$. The key issue with iterative methods is whether iteration (10) converges. If the sequence $\{x(k)\}_{k=1}^{\infty}$ converges, i.e., if $x(k) \to x^*$ as $k \to \infty$, and the function f is continuous, then x^* is a *fixed point* of f, that is, it satisfies the equation $x^* = f(x^*)$.

An important issue with relaxation methods is speed of convergence. One way to improve the convergence rate is to carry out the calculations for all components of $x(k)$ in (10) simultaneously. To this end, let $x_i(k)$ denote the ith component of $x(k)$ and let f_i denote the ith component of the function f. Then, equation (10) can be rewritten as

$$x_i(k) = f_i(x_1(k-1), \ldots, x_n(k-1)), \qquad i = 1, \ldots, n. \tag{11}$$

A simple way to parallelize iteration (10) is to create n processes (or threads) and employ each one of them to update a different component of x according to (11). At the kth stage of the iteration, $x(k-1)$ is readily computed and stored in the previous stage for usage in future stages. Therefore, each process knows the values of all components of $x(k-1)$ that are needed for the computation of f. The new values of the components of $x(k)$ are computed in parallel and exchanged between the processes in order to start the next stage. This approach of parallelization speeds up an iterative method such as (10) most of the times, but it may not be practical if x has too many components. For example, there may be few processes available, or the creation of a new process may be an expensive task in terms of time and memory. Therefore, a coarse-grained parallelization of iteration (10) may be desirable in some applications. In particular, assume that the domain of f can be decomposed as the Cartesian product of lower dimensional subspaces according to $\mathbb{R}^n = \prod_{j=1}^{p} \mathbb{R}^{n_j}$, where $\sum_{j=1}^{p} n_j = n$. Accordingly, the vector $x \in \mathbb{R}^n$ can be decomposed conformally as $x = (x^1, \ldots, x^j, \ldots, x^p)$, where $x^j \in \mathbb{R}^{n_j}$ is the *jth*

block-component of x. Subsequently, equation (10) can be written as

$$x^j(k) = f^j(x^1(k-1), \ldots, x^j(k-1), \ldots, x^p(k-1)), \qquad j = 1, \ldots, p, \qquad (12)$$

where each $f^j : \mathbb{R}^n \mapsto \mathbb{R}^{n_j}$ is the jth block component of f. Each one of the p processes is assigned the computation of a different block-component f^j, according to equation (12). This approach is called *block-parallelization* [25].

One of the primary differences in iterative methods is how the previously computed information is incorporated to update the new value of x at each iteration. In equation (11), all of the components of $x(k)$ are updated simultaneously. This type of iteration is called *Jacobi type* iteration. Alternatively, the iteration step can be modified such that the components of x are updated one at a time with respect to the most recent information available. That is, (11) can be modified as

$$x_i(k) = f_i(x_1(k), \ldots, x_{i-1}(k), x_i(k-1), \ldots, x_n(k-1)), \qquad i = 1, \ldots, n. \qquad (13)$$

This is called *Gauss-Seidel type* iteration. Since Gauss-Seidel type methods incorporate the most recent information, they typically converge to the solution of the problem faster than their Jacobi type counterparts. Therefore, Gauss-Seidel type methods are more desirable than Jacobi type ones. On the other hand, a Gauss-Seidel iteration (sometimes called a sweep) may have more interdependencies when computing the different components of x. Hence, a Gauss-Seidel iteration is implemented sequentially. On the other hand, Jacobi type methods have greater potential for massively parallel implementation.

4.2.1 Relaxation Methods for Solving Shortest Path Problems

Let $\mathcal{G} = (V, E)$ be a directed graph where V and $E \subseteq V \times V$ are the vertex and edge sets, respectively, and let $V_{\text{goal}} = V \cap \mathcal{X}_{\text{goal}}$ be the goal set of vertices. Let us assume that there exist a path from every vertex $v_i \in V$ to V_{goal}. Let $\mathtt{J}^*(v_i)$ be the optimal cost-to-go value of the vertex $v_i \in V$, which is computed as the unique solution of the following Bellman

equation

$$\mathtt{J}^*(v_i) = \begin{cases} \min_{v_j \in \mathtt{succ}(\mathcal{G}, v_i)} \left(\mathtt{c}(v_i, v_j) + \mathtt{J}^*(v_j) \right), & v_i \in V \setminus V_{\text{goal}}, \\ 0, & v_i \in V_{\text{goal}}. \end{cases} \tag{14}$$

Once Bellman's equation is solved, i.e., all optimal cost-to-go values $\mathtt{J}^*(v_i)$ are computed, the optimal path from any given vertex v_i to V_{goal} can be easily constructed by starting at v_i and traversing iteratively from the current vertex $v_j \in V$ to any of its successors $v_k \in \mathtt{succ}(\mathcal{G}, v_j)$ that yields the minimum cost-to-go value $\mathtt{J}^*(v_j)$ (ties can be broken arbitrarily), until V_{goal} is reached. Iterative methods can be developed for computing the cost-to-come values efficiently by simply relaxing Bellman's equation [25]. The Bellman-Ford algorithm is an efficient algorithm to solve (14). It is used to solve the single-source shortest-path problem where the edge costs may be negative [20,47]. Two different versions of the Bellman-Ford algorithm are given below, and both of these algorithms converge to $\mathtt{J}^*(v_i)$ for all $v_i \in V$, for arbitrary initial conditions. The nice property of converging to the optimal solution from any initial condition makes the Bellman-Ford algorithm a good choice to use to implement the exploitation step of incremental sampling-based algorithms, where the shortest path problem must be solved repeatedly as new information is obtained during previous exploration steps.

The Jacobi version of the Bellman-Ford algorithm works as follows. The initial conditions are set as $\mathtt{J}^0(v_i) = 0$ for all $v_i \in V_{\text{goal}}$ and $\mathtt{J}^0(v_i) = \infty$ for all $v_i \in V \setminus V_{\text{goal}}$. It is assumed that the algorithm *terminates after N iterations* meaning that for all $N \leq k$, $\mathtt{J}^k(v_i) = \mathtt{J}^{k-1}(v_i)$ for all $v_i \in V$. Then, the kth iteration of the Jacobi-type Bellman-Ford algorithm is written as

$$\mathtt{J}^k(v_i) = \begin{cases} \min_{v_j \in \mathtt{succ}(\mathcal{G}, v_i)} \left(\mathtt{c}(v_i, v_j) + \mathtt{J}^{k-1}(v_j) \right), & v_i \in V \setminus V_{\text{goal}}, \\ 0, & v_i \in V_{\text{goal}}. \end{cases} \tag{15}$$

This algorithm is well suited for massively parallel implementation since the cost-to-go values of all vertices v_i can be updated in parallel. This form is also known as value

iteration.

The Gauss-Seidel version of the Bellman-Ford algorithm uses the same initial conditions and termination criterion as the Jacobi version. Let $o : \{1, \ldots, n\} \mapsto V$ be an ordering function that determines the cost-to-come value update order of the vertices of a Gauss-Seidel iteration. Then the kth iteration of the algorithm is written as

$$\mathtt{J}^k(v_{o(\ell)}) = \begin{cases} \min_{v_j \in \mathtt{succ}(\mathcal{G}, v_{o(\ell)})} \left(\mathtt{c}(v_{o(\ell)}, v_j) + \mathtt{J}^{k'}(v_j) \right), & v_{o(\ell)} \in V \setminus V_{\text{goal}}, \\ 0, & v_{o(\ell)} \in V_{\text{goal}}, \end{cases} \tag{16}$$

where $k' = k$ if $v_j \prec v_{o(\ell)}$ and $k' = k - 1$ if $v_{o(\ell)} \prec v_j$. Here the notation $v \prec u$ means that vertex v is updated before vertex u. Since cost-to-come values of vertices are updated with respect to the given order at each iteration, different updating orders typically generate different results. An ordering which increases parallelism at each iteration, while converging to the correct solution, is favored.

4.3 The $\mathrm{RRT}^{\#}$ *Algorithm - Overview*

The solution of Bellman's Equation gives the optimal path from any vertex to the goal vertex set. In our problem, the initial point x_{init} is given a priori. Therefore, Bellman's Equation can be rewritten in terms of the cost-to-come values as follows:

$$\mathtt{g}^*(v_i) = \begin{cases} 0, & v_i = x_{\text{init}}, \\ \min_{v_j \in \mathtt{pred}(\mathcal{G}, v_i)} \left(\mathtt{g}^*(v_j) + \mathtt{c}(v_j, v_i) \right), & v_i \in V \setminus \{x_{\text{init}}\}. \end{cases} \tag{17}$$

We can rewrite both Jacobi and Gauss-Seidel versions of the Bellman-Ford algorithm in terms of cost-to-come values by relaxing equation (17). To this end, let n_k be the number of vertices (i.e., states) and let $g^k \in \mathbb{R}^{n_k}$ be the n_k-dimensional vector whose components are the cost-to-come values of the vertices during the kth iteration of the Bellman-Ford algorithm, i.e., let $g_i^k = \mathtt{g}^k(v_i)$. Let now f_J be a vector-valued function defined as follows

$$f_{J_i}(g^{k-1}) = \begin{cases} 0, & v_i = x_{\text{init}}, \\ \min_{v_j \in \mathtt{pred}(\mathcal{G}, v_i)} (g_j^{k-1} + \mathtt{c}(v_j, v_i)), & v_i \in V \setminus \{x_{\text{init}}\}. \end{cases} \tag{18}$$

Then, the Jacobi iteration can be succinctly written as $g^k = f_J(g^{k-1})$. The initial conditions are given by $g_i^0 = 0$ for $v_i = x_{\text{init}}$ and $g_i^0 = \infty$ for all $v_i \in V \setminus \{x_{\text{init}}\}$.

The value computed by f_{J_i} at the kth iteration is the one-step lookahead cost-to-come estimate of the vertex v_i, called the locally minimum cost-to-come estimate, or lmc-value of the vertex v_i for short (also called rhs-value in [71]). We therefore write $\texttt{lmc}^k(v_i) = f_{J_i}(g^k)$. The vertex v_i is called *stationary* (or consistent, see [71]) if its g-value is equal to its lmc-value, that is, if $\texttt{g}^k(v_i) = \texttt{lmc}^k(v_i)$ which, in turn, implies that $g_i^k = f_{J_i}(g^k)$. Otherwise, the vertex v_i it is called nonstationary.

Similarly, for the Gauss-Seidel version of the Bellman-Ford algorithm, let $g^{k,\ell} \in \mathbb{R}^{n_k}$ be the n_k-dimensional vector whose components are the cost-to-come values of the vertices after the cost-to-come value of ℓth order vertex is updated during the kth iteration of the algorithm, that is, $g_i^{k,\ell} = g_i^{k+1}$ if $v_i \preceq v_{o(\ell)}$ and $g_i^{k,\ell} = g_i^k$ if $v_{o(\ell)} \prec v_i$. The initial conditions are set similarly to the previous case, namely, $g_i^{0,0} = 0$ for $v_i = x_{\text{init}}$ and $g_i^{0,0} = \infty$ for all $v_i \in V \setminus \{x_{\text{init}}\}$. Then, a Gauss-Seidel iteration can be written succinctly as $g^{k,0} = f_G(g^{k-1,0})$, where

$$f_{G_{o(\ell)}}(g^{k-1,\ell-1}) = \begin{cases} 0, & v_{o(\ell)} = x_{\text{init}}, \\ \min_{v_j \in \texttt{pred}(\mathcal{G}, v_{o(\ell)})}(g_j^{k-1,\ell-1} + \texttt{c}(v_j, v_{o(\ell)})), & v_{o(\ell)} \in V \setminus \{x_{\text{init}}\}. \end{cases} \tag{19}$$

During each iteration, the components of $g^{k-1,0}$ are updated one at a time by $g_{o(\ell)}^{k-1,\ell} = f_{G_{o(\ell)}}(g^{k-1,\ell-1})$ where $g^{k,0} = g^{k-1,n_k}$. The lmc-value of the vertex v_i at stage (k,ℓ) is defined as $\texttt{lmc}^{k,\ell}(v_i) = f_{G_i}(g^{k,\ell})$ for Gauss-Seidel type iterations and the vertex v_i is called stationary if $\texttt{g}^{k,\ell}(v_i) = \texttt{lmc}^{k,\ell}(v_i)$, which implies $g_i^{k,\ell} = f_{G_i}(g^{k,\ell})$.

Based on the previous reformulation of the Bellman-Ford algorithm in terms of Jacobi and Gauss-Seidel iterations, we can now give the details of the $\text{RRT}^{\#}$ algorithm. The $\text{RRT}^{\#}$ algorithm performs two tasks, namely exploration and exploitation, during each iteration. The exploration task implements the extension procedure of the RRG algorithm, and is subsequently followed by the exploitation task which implements the Gauss-Seidel

version of the Bellman-Ford algorithm as in equation (19). A brief description of each of these steps is given below.

Exploration: This step samples randomly a point in $\mathcal{X}_{\text{free}}$ and then extends the underlying graph toward the sampled point by including it as a new vertex in the current graph by connecting the missing edges. The following procedures are part of the exploration step.

Sampling: `Sample` : $\mathbb{N} \rightarrow \mathcal{X}_{\text{free}}$ is a function that returns independent, identically distributed (i.i.d) samples from $\mathcal{X}_{\text{free}}$.

Nearest neighbor: `Nearest` is a function that returns a point from a given finite set V, which is the closest to a given point x in terms of a given distance function.

Near vertices: `Near` is a function that returns the n closest points in a given finite set V to a given point x in terms of a given distance function.

Steering: `Steer` is a function that returns the point in a ball centered around a given state x that is closest, with respect to the given distance function, to another given point x_{new}.

Collision checking: Given two points $x_1, x_2 \in \mathcal{X}_{\text{free}}$, the Boolean function $\texttt{ObstacleFree}(x_1, x_2)$ checks whether the line segment connecting these two points belongs to $\mathcal{X}_{\text{free}}$. It returns `True` if the line segment is a subset of $\mathcal{X}_{\text{free}}$, i.e., $\texttt{Line}(x_1, x_2) \subset \mathcal{X}_{\text{free}}$, and `False` otherwise.

Graph extension: `Extend` is a function that extends the nearest vertex of the graph $\mathcal{G}$ toward the randomly sampled point x_{rand}.

Exploitation: This step implements the task of improving the cost-to-come values of the current vertices as new information becomes available. It also encodes the lowest-cost path information for the promising vertices (see equation (20) below) and v^*_{goal} (the goal vertex that has the lowest cost-to-come value in the goal set) as a spanning tree rooted at the initial vertex. Cost-to-come values of nonstationary vertices are updated in an order based on their f-values, i.e., an underestimate of the cost of the optimal path from the initial

vertex to the goal set passing through the vertex of interest, and ties are broken in favor of vertices which have smaller g-values at each iteration of the Gauss-Seidel version of the Bellman-Ford algorithm. Note that the algorithm works so that stationarity of just a subset of vertices (rather than of all vertices) suffices to compute the optimal path from x_{init} to $\mathcal{X}_{\text{goal}}$. Details of the procedures used in the exploitation step are given below.

Ordering: Given a vertex $v_i \in V$, the function $\texttt{Key} : v_i \mapsto k_i$ returns a real vector $k_i \in \mathbb{R}^2$, whose components are $k_{i1}(v_i) = \texttt{lmc}(v_i) + \texttt{h}(v_i)$ and $k_{i2}(v_i) = \texttt{lmc}(v_i)$. The components of the key correspond to the f- and g-values in the A* algorithm, respectively [93]. The precedence relation between keys is determined according to lexicographical ordering. Given two keys $k_1, k_2 \in \mathbb{R}^2$, the Boolean function $\preccurlyeq : (k_1, k_2) \mapsto \{\texttt{False}, \texttt{True}\}$ returns $\texttt{True}$ if and only if either $k_{11} < k_{21}$ or $(k_{11} = k_{21} \text{ and } k_{12} \leq k_{22})$, and returns $\texttt{False}$ otherwise.

Promising vertices: Given a graph $\mathcal{G} = (V, E)$ with $x_{\text{init}} \in V$, let $\texttt{g}^*(v_i)$ be the optimal cost-to-come value of the vertex v_i that can be achieved on the given graph $\mathcal{G}$, and let $v^*_{\text{goal}} = \text{argmin}_{v_i \in V \cap \mathcal{X}_{\text{goal}}} \texttt{g}^*(v)$. The *promising vertices* $V_{\text{prom}} \subset V$ is the set of vertices that satisfy the following relation

$$V_{\text{prom}} = \{v_i \in V : [\texttt{f}(v_i), \texttt{g}^*(v_i)] \prec [\texttt{f}(v^*_{\text{goal}}), \texttt{g}^*(v^*_{\text{goal}})]\}, \tag{20}$$

where $\texttt{f}(v_i) = \texttt{g}^*(v_i) + \texttt{h}(v_i)$. Only promising vertices have the potential to be part of the optimal path from x_{init} to $\mathcal{X}_{\text{goal}}$. Therefore, all promising vertices must be stationary at the end of each iteration.

Relevant region: Let $x^*_{\text{goal}} \in \mathcal{X}_{\text{goal}}$ be the point in the goal region that has the lowest optimal cost-to-come value in $\mathcal{X}_{\text{goal}}$, i.e., $x^*_{\text{goal}} = \text{argmin}_{x \in \mathcal{X}_{\text{goal}}} \texttt{g}^*(x)$. The *relevant region* of $\mathcal{X}_{\text{free}}$ is the set of points x for which the optimal cost-to-come value of x, plus the estimate of the optimal cost moving from x to $\mathcal{X}_{\text{goal}}$ is less than the optimal cost-to-come value of x^*_{goal}, that is,

$$\mathcal{X}_{\text{rel}} = \{x \in \mathcal{X}_{\text{free}} : \texttt{g}^*(x) + \texttt{h}(x) < \texttt{g}^*(x^*_{\text{goal}})\}. \tag{21}$$

Points that lie in the $\mathcal{X}_{\text{rel}}$ have the potential to be part of the optimal path starting at x_{init} and reaching $\mathcal{X}_{\text{goal}}$.

Replanning: Given a graph $\mathcal{G}^k = (V^k, E^k)$ at the kth iteration, a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$ and an arbitrary vector $g^{k-1,0} \in \mathbb{R}^{n_k}$ of cost-to-come values of all $v_i \in V^k$, where $g_i^{k-1,0} = 0$ for $v_i = x_{\text{init}}$, the function `Replan` : $(\mathcal{G}^k, \mathcal{X}_{\text{goal}}, g^{k-1,0}) \mapsto (\mathcal{G}^k, \mathcal{X}_{\text{goal}}, g^{k,0})$ operates on the *nonstationary* vertices iteratively until all promising vertices become stationary. The `Replan` function is used to propagate the effects of the topological changes in the graph as new vertices are added with each iteration.

Priority of vertices: The priority of vertices is the same as the priority of their associated keys, and a priority queue is used to sort all of the nonstationary vertices of the graph based on their respective key values. The following functions are defined to manage the priority queue.

Updating queue: Given a vertex $v_i \in V$, the function `UpdateQueue` changes the queue based on the g- and lmc-values of the vertex v_i. If the vertex v_i is nonstationary, then it is either inserted into the queue or its priority in the queue is updated based on its up-to-date key value if it is already inside the queue. Otherwise, the vertex is removed from the queue if it is a stationary vertex. The order of expanded vertices is determined by selecting the vertex of minimum key value in the queue for expansion at each step.

Finding minimum: The function $findmin()$ returns the vertex with the highest priority of all vertices in the queue. This is the vertex of minimum key value.

Removing a vertex: Given a vertex $v_i \in V$, the function $remove()$ deletes the vertex v_i from the queue.

Updating priority: Given a vertex $v_i \in V$, and a key value, the function $update()$ changes the priority of the vertex v_i in the priority queue by reassigning the key value of the vertex v_i with the new given key value.

Inserting a vertex: Given a vertex $v_i \in V$, and a key value, the function $insert()$ adds the vertex v_i with the given key value into the queue.

4.4 *The* $\mathrm{RRT}^{\#}$ *Algorithm - Details*

The main body of the $\mathrm{RRT}^{\#}$ algorithm is given in Algorithm 1 and it is similar to the other RRT-variants (RRT, RRG, RRT^{*}, etc.) with the notable exception that it keeps track of vertex stationarity using the key values of all current vertices in the graph. One of the important differences between the RRT^{*} and $\mathrm{RRT}^{\#}$ algorithms is that all vertices in the tree computed by the RRT^{*} algorithm have a uniform type based on their finite cost-to-come value, whereas in the $\mathrm{RRT}^{\#}$ algorithm the vertices have four different types based on their one-step lookahead estimates of the cost-to-come value. In the $\mathrm{RRT}^{\#}$ algorithm, each vertex v is classified into one of the following four categories, based on the values of its $(\mathtt{g}(v), \mathtt{lmc}(v))$ pair.

- Stationary with finite key value: $\mathtt{g}(v) < \infty, \mathtt{lmc}(v) < \infty$ and $\mathtt{g}(v) = \mathtt{lmc}(v)$
- Stationary with infinite key value: $\mathtt{g}(v) = \infty, \mathtt{lmc}(v) = \infty$
- Nonstationary with finite key value: $\mathtt{g}(v) < \infty, \mathtt{lmc}(v) < \infty$ and $\mathtt{g}(v) \neq \mathtt{lmc}(v)$
- Nonstationary with infinite g-value and finite lmc-value: $\mathtt{g}(v) = \infty, \mathtt{lmc}(v) < \infty$

Stationary vertices with infinite key value are always non-promising, whereas for the rest of the cases the vertices can be either promising or non-promising.

Algorithm 1: Body of the $\mathrm{RRT}^{\#}$ Algorithm

```
1  RRT#(x_init, X_goal, X)
2      V ← {x_init}; E ← ∅;
3      G ← (V, E);
4      for k = 1 to N do
5          x_rand ← Sample(k);
6          G ← Extend(G, x_rand);
7          Replan(G, X_goal);
8      (V, E) ← G; E' ← ∅;
9      foreach x ∈ V do
10         E' ← E' ∪ {(parent(x), x)}
11     return T = (V, E')
```

The algorithm starts by adding the initial point x_{init} into the vertex set of the underlying graph. Then, it incrementally grows the graph in $\mathcal{X}_{\text{free}}$ by sampling randomly a point x_{rand} from $\mathcal{X}_{\text{free}}$ and extending the graph toward x_{rand}. The `Replan` procedure, which is provided in Algorithm 3, then propagates the new information due to the extension across the whole graph in order to improve the cost-to-come values of the promising vertices in the graph. All computations due to the sampling and extension steps, followed by exploitation (Lines 4-6 of Algorithm 1), form a single *iteration* of the algorithm. The process is repeated for a given fixed number of iterations. The spanning tree of the final graph which is rooted at the initial vertex, and which contains the lowest-cost path information for the promising vertices and v^*_{goal}, is returned at the end.

This spanning tree (Line 7 in Algorithm 1) contains information about the lowest-cost path for each promising vertex and v^*_{goal}, which can be achieved on the current graph. It utilizes the information provided by the exploration step to the highest degree, and is one of the key difference between the $\text{RRT}^{\#}$ algorithm and other RRT-variants, including the RRT^* algorithm. In addition, in the $\text{RRT}^{\#}$ algorithm the g-values of the promising vertices are equal to their respective optimal cost-to-come values that can be achieved through the edges of the graph. This allows us to initialize the g-value of a new vertex with a smaller estimate value during extension if it has any promising neighbor vertex. This estimate keeps improving to the best possible value whenever new information becomes available on any part of the graph. Hence, the g-value of each promising vertex in the graph converges to its optimal cost-to-come value very quickly.

The `Extend` procedure used in the $\text{RRT}^{\#}$ algorithm is given in Algorithm 2. During each iteration, the `Extend` procedure tries to extend the graph toward the randomly sampled point $x_{\text{rand}} \in \mathcal{X}_{\text{free}}$. First, the closest vertex in the graph x_{nearest} is found in Line 3, then x_{nearest} is steered toward the randomly sampled point x_{rand} in the next line. If the line segment connecting the steered point x_{new} and x_{nearest} is feasible, then the new point x_{new} is prepared for inclusion to the vertex set of the graph. Then, a local search is performed in

Algorithm 2: `Extend` Procedure for RRT$^{\#}$ Algorithm

```
1  Extend(G,x)
2      (V, E) ← G; E' ← ∅;
3      x_nearest ← Nearest(G, x);
4      x_new ← Steer(x_nearest, x);
5      if ObstacleFree(x_nearest, x_new) then
6          Initialize(x_new, x_nearest);
7          X_near ← Near(G, x_new, |V|);
8          foreach x_near ∈ X_near do
9              if ObstacleFree(x_near, x_new) then
10                 if lmc(x_new) > g(x_near) + c(x_near, x_new) then
11                     lmc(x_new) = g(x_near) + c(x_near, x_new);
12                     parent(x_new) = x_near;
13                 E' ← E' ∪ {(x_near, x_new), (x_new, x_near)};
14         V ← V ∪ {x_new};
15         E ← E ∪ E';
16         UpdateQueue(x_new);
17     return G' ← (V, E)
```

some neighborhood of x_{new} (i.e., inside the set of vertices returned by the `Near` procedure) in order to find the local minimum cost-to-come estimate value and the corresponding parent vertex. This is done in Lines 8-13 of Algorithm 2. The new vertex x_{new} and all extensions resulting in feasible paths are added to the vertex and edge sets of the graph in Lines 14-15. In the end, the new vertex is decided to be inserted in the priority queue or not based on its stationarity in the `UpdateQueue` procedure.

Algorithm 3: `Replan` Procedure

```
1  Replan(G,X_goal)
2      while q.findmin() ≺ Key(v*_goal) do
3          x = q.findmin();
4          g(x) = lmc(x);
5          q.delete(x);
6          foreach s ∈ succ(G, x) do
7              if lmc(s) > g(x) + c(x, s) then
8                  lmc(s) = g(x) + c(x, s);
9                  parent(s) = x;
10                 UpdateQueue(s);
```

Algorithm 4: Auxiliary Procedures

```
1  Initialize(x, x')
2      g(x) ← ∞;
3      lmc(x) ← ∞;
4      parent(x) ← x';
5      if x' ≠ ∅ then
6          lmc(x) ← g(x') + c(x', x);
7  UpdateQueue(x)
8      if g(x) ≠ lmc(x) and x ∈ q then
9          q.update(x, Key(x));
10     else if g(x) ≠ lmc(x) and x ∉ q then
11         q.insert(x, Key(x));
12     else if g(x) = lmc(x) and x ∈ q then
13         q.delete(x);
14 Key(s)
15     return k = (lmc(x) + h(x), lmc(x));
```

A newly inserted vertex may be nonstationary if it has a finite lmc-value. Therefore, the spanning tree needs to be checked, and appropriate operations must be performed in order to update lowest-cost path information, if necessary. The `Replan` procedure, which is provided in Algorithm 3, is called to update the spanning tree by operating on the nonstationary and *promising* vertices of the graph, iteratively. It simply pops the most promising nonstationary vertex from the priority queue, if there are any, and this nonstationary vertex is made stationary by assigning its lmc-value to its g-value. Then, its new g-value is propagated among its neighbors in order to improve their lmc-values in Lines 6-10 of Algorithm 3. However, this information propagation may also cause some vertices to become nonstationary; therefore, all resulting nonstationary vertices are inserted in the priority queue as well. This process continues until there are no nonstationary promising vertices left in the priority queue.

Note that the termination condition in Line 2 of Algorithm 3 ensures that when the `Replan` procedure terminates, all promising and nonstationary vertices are expanded. This would be clearly true if the `Replan`$(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure were allowed to operate on all

nonstationary vertices of the graph, that is, if the termination condition in Line 2 of Algorithm 3 were replaced by the condition "*queue is not empty*." In such a case, the `Replan` procedure would expand all vertices until they all became stationary before the procedure terminated. However, the termination condition in Line 2 of Algorithm 3 actually ensures much more, namely, that all *promising* vertices (and only those) are made stationary, so there is no need to expand the non-promising vertices. This is important for efficiency since a non-promising vertex cannot be part of the optimal path. Therefore, there is no need to expand non-promising vertices, thus speeding up the whole algorithm. The details of how this is achieved are given in Theorem 1 in the following section.

Note that without loss of generality, we will assume that the goal region consists of a single point (i.e., $\mathcal{X}_{\text{goal}} = x_{\text{goal}}$) and the graph is grown from x_{goal} to x_{init}. Therefore, the proofs are presented based on the cost-to-go (J-value) and one-step lookahead cost ($\bar{J}$-value) estimates of the nodes of the graph.

4.5 Key Results and Proofs

Lemma 1

The cost-to-go and one-step lookahead cost values of each vertex $v \in V^k$ are nonincreasing with each step of each iteration of the `Replan`$(\mathcal{G}^k, x_{\text{init}})$ *procedure.*

Proof. Let us consider a vertex $v \in V^k$ at the kth iteration of the $\text{RRT}^{\#}$ algorithm. The one-step lookahead cost v is either infinite ($\bar{\text{J}}^{k-1,0}(v) = \infty$) or has a finite value ($\bar{\text{J}}^{k-1,0}(v) < \infty$) at the beginning of the kth iteration. The former occurs when the cost-to-go value of each successor vertex of v is infinite, whereas the latter occurs when there is at least one successor vertex of v with finite cost-to-go value. As shown in Line 11 of Algorithm 2 and Line 8 of Algorithm 3 , the one-step lookahead cost value of vertex v is updated only if Bellman updates are performed for its successors. This situation occurs when one of the successor vertices of v is expanded, and this expansion results in an improvement in the current one-step lookahead cost value of v. Therefore, assuming that the vertex u is

expanded at the ℓth step, we have that $\bar{\mathtt{J}}^{k-1,\ell}(v) = \mathtt{c}(v,u) + \mathtt{J}^{k-1,\ell}(u) < \bar{\mathtt{J}}^{k-1,\ell-1}(v)$ where $u \in \mathtt{succ}(\mathcal{G}^k, v)$, otherwise $\bar{\mathtt{J}}^{k-1,\ell}(v) = \bar{\mathtt{J}}^{k-1,\ell-1}(v)$. Finally, we have that $\bar{\mathtt{J}}^{k-1,\ell}(v) \leq \bar{\mathtt{J}}^{k-1,\ell-1}(v)$ for all $0 < \ell \leq n_k$, where n_k is the last step of the $\mathtt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure.

Let ℓ_i $(i = 0, 1, \dots)$ denote the step when vertex v is expanded and is made stationary, that is, $\mathtt{J}^{k-1,\ell_i}(v) = \bar{\mathtt{J}}^{k-1,\ell_i}(v) = \bar{\mathtt{J}}^{k-1,\ell_i-1}(v)$. The cost-to-go value of each vertex, other than the expanded one, remains unchanged during a step of the $\mathtt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure. Therefore, $\mathtt{J}^{k-1,\ell}(v) = \bar{\mathtt{J}}^{k-1,\ell_i}(v)$ for all $\ell_i \leq \ell < \ell_{i+1}$. This implies that $\mathtt{J}^{k-1,\ell}(v) = \mathtt{J}^{k-1,\ell-1}(v)$ for all $\ell_i < \ell < \ell_{i+1}$. For the cost-to-go value of v at the ℓ_{i+1}th step, we may therefore write $\mathtt{J}^{k-1,\ell_{i+1}}(v) = \bar{\mathtt{J}}^{k-1,\ell_{i+1}}(v) \leq \bar{\mathtt{J}}^{k-1,\ell_i}(v) = \mathtt{J}^{k-1,\ell_{i+1}-1}(v)$. It follows that $\mathtt{J}^{k-1,\ell}(v) \leq \mathtt{J}^{k-1,\ell-1}(v)$ for all $0 < \ell \leq n_k$. □

Lemma 2

If a vertex $v \in V$ is nonstationary at stage (k, ℓ), then its cost-to-go value is greater than its one-step lookahead cost value, i.e., $\mathtt{J}^{k-1,\ell}(v) > \bar{\mathtt{J}}^{k-1,\ell}(v)$. Consequently, the J- and $\bar{J}$-values of a vertex v satisfy the relation $\mathtt{J}^{k-1,\ell}(v) \geq \bar{\mathtt{J}}^{k-1,\ell}(v)$ at any given time.

Proof. Let us assume that vertex v is stationary at the beginning of the ℓth step of the kth iteration of the RRT$^{\#}$ algorithm, that is, $\mathtt{J}^{k-1,\ell-1}(v) = \bar{\mathtt{J}}^{k-1,\ell-1}(v)$, and it becomes nonstationary due to the expansion of vertex u at the ℓth step. As discussed in Lemma 1, the J-value of each vertex remains unchanged except for the J-value of the expanded vertex, and hence, $\mathtt{J}^{k-1,\ell}(v) = \mathtt{J}^{k-1,\ell-1}(v)$. Furthermore, the $\bar{J}$-value of v is updated only if one of its successors is expanded, and this expansion yields an improvement in its current $\bar{J}$-value, that is, $\bar{\mathtt{J}}^{k-1,\ell}(v) = \mathtt{c}(v,u) + \mathtt{J}^{k-1,\ell}(u) < \bar{\mathtt{J}}^{k-1,\ell-1}(v)$, where $u \in \mathtt{succ}(\mathcal{G}^k, v)$. Therefore, whenever v is nonstationary, we have $\mathtt{J}^{k-1,\ell}(v) = \mathtt{J}^{k-1,\ell-1}(v) = \bar{\mathtt{J}}^{k-1,\ell-1}(v) > \bar{\mathtt{J}}^{k-1,\ell}(v)$. Also, it holds $\mathtt{J}^{k-1,\ell}(v) = \bar{\mathtt{J}}^{k-1,\ell}(v)$ whenever vertex v is stationary. Hence, it follows that $\mathtt{J}^{k-1,\ell}(v) \geq \bar{\mathtt{J}}^{k-1,\ell}(v)$ for at any given step. □

Lemma 3

Let $u \in V^k$ be the vertex selected for expansion at the ℓth step of the kth iteration, that

is, let u be the nonstationary vertex of highest priority in the queue. Then, the following relations hold for any vertex $v \in V^k$ due to the expansion of the vertex u:

i) If v is nonstationary, then $\texttt{Key}^{k-1,\ell}(v) \succeq \texttt{Key}^{k-1,\ell-1}(u)$.

ii) If v is stationary and becomes nonstationary at the next step, then $\texttt{Key}^{k-1,\ell}(v) \succ \texttt{Key}^{k-1,\ell-1}(u)$.

Proof. First, we need to state some general observations. Without loss of generality, we assume that the vertices v and u are distinct. When vertex $u \in V^k$ is expanded at the ℓth step, it becomes stationary at the next step, and thus $\texttt{J}^{k-1,\ell}(u) = \bar{\texttt{J}}^{k-1,\ell}(u) = \bar{\texttt{J}}^{k-1,\ell-1}(u)$. Therefore, we have $\texttt{Key}^{k-1,\ell}(u) = \texttt{Key}^{k-1,\ell-1}(u)$. Also, the J-values of all other vertices remain unchanged, that is, $\texttt{J}^{k-1,\ell}(v) = \texttt{J}^{k-1,\ell-1}(v)$ for all $v \in V^k$; only their $\bar{J}$-values may be updated. The latter occurs if $v \in \texttt{pred}(\mathcal{G}^k, u)$. Therefore, a vertex $v \in V^k$ can be made nonstationary only if $v \in \texttt{pred}(\mathcal{G}^k, u)$ and its $\bar{J}$-value is also updated. This situation happens when the J-value of one of the successors of v (in this case u) has been decreased, thus resulting in a decrease of the $\bar{J}$-value of v.

Case i): There are three possibilities in this case. First, the key value of the vertex v is not updated and therefore v remains nonstationary. This implies that $\texttt{Key}^{k-1,\ell}(v) = \texttt{Key}^{k-1,\ell-1}(v)$. Since the vertex u is selected for expansion before vertex v, it has higher priority than vertex v, and thus, $\texttt{Key}^{k-1,\ell-1}(u) \preceq \texttt{Key}^{k-1,\ell-1}(v)$. It follows that $\texttt{Key}^{k-1,\ell}(v) = \texttt{Key}^{k-1,\ell-1}(v) \succeq \texttt{Key}^{k-1,\ell-1}(u)$.

Next, assume that the key of vertex v is updated but v still remains nonstationary during the expansion of vertex u. It follows that, necessarily, $u \in \texttt{succ}(\mathcal{G}^k, v)$. In this case, the $\bar{J}$-value of v is decreased after the expansion of vertex u, and we have that $\bar{\texttt{J}}^{k-1,\ell}(v) <$

$\bar{\mathtt{J}}^{k-1,\ell-1}(v)$. It follows that

$$\begin{aligned}
\mathtt{Key}^{k-1,\ell}(v) &= [\mathtt{h}(x_{\text{init}}, v) + \bar{\mathtt{J}}^{k-1,\ell}(v), \bar{\mathtt{J}}^{k-1,\ell}(v)] \\
&= [\mathtt{h}(x_{\text{init}}, v) + \mathtt{c}(v, u) + \mathtt{J}^{k-1,\ell}(u), \mathtt{c}(v, u) + \mathtt{J}^{k-1,\ell}(u)] \\
&\succ [\mathtt{h}(x_{\text{init}}, u) + \mathtt{J}^{k-1,\ell}(u), \mathtt{J}^{k-1,\ell}(u)] \\
&= [\mathtt{h}(x_{\text{init}}, u) + \bar{\mathtt{J}}^{k-1,\ell-1}(u), \bar{\mathtt{J}}^{k-1,\ell-1}(u)] \\
&= \mathtt{Key}^{k-1,\ell-1}(u).
\end{aligned}$$

Finally, assume that vertex v is nonstationary and it becomes stationary during the expansion of vertex u. Since vertex v is nonstationary at stage $(k-1, \ell-1)$, it follows that $\mathtt{J}^{k-1,\ell-1}(v) > \bar{\mathtt{J}}^{k-1,\ell-1}(v)$ according to Lemma 2. Its J-value remains unchanged and it becomes stationary at the next step due to the expansion of the vertex u. It follows that $\bar{\mathtt{J}}^{k-1,\ell}(v) = \mathtt{J}^{k-1,\ell}(v) = \mathtt{J}^{k-1,\ell-1}(v) > \bar{\mathtt{J}}^{k-1,\ell-1}(v)$ which contradicts Lemma 1. Hence, due to the expansion of a vertex at the ℓth step, no vertex other than the expanded one becomes stationary at the next step.

Case ii): Since vertex v becomes nonstationary at the next step due to the expansion of vertex u, it follows that u must be one of the successors of v. That is, $u \in \mathtt{succ}(\mathcal{G}^k, v)$ and $\bar{\mathtt{J}}^{k-1,\ell}(v) = \mathtt{c}(v, u) + \mathtt{J}^{k-1,\ell}(u)$. Furthermore, the $\bar{J}$-value of the vertex v is decreased during the expansion of vertex u, and thus, $\mathtt{J}^{k-1,\ell}(u) = \bar{\mathtt{J}}^{k-1,\ell}(u) = \bar{\mathtt{J}}^{k-1,\ell-1}(u)$. It follows that $\mathtt{J}^{k-1,\ell}(v) = \mathtt{J}^{k-1,\ell-1}(v) = \bar{\mathtt{J}}^{k-1,\ell-1}(v) > \bar{\mathtt{J}}^{k-1,\ell}(v)$ and thus $\mathtt{J}^{k-1,\ell}(v) > \bar{\mathtt{J}}^{k-1,\ell}(v)$. Similarly to Case i), it can be shown that $\mathtt{Key}^{k-1,\ell}(v) \succ \mathtt{Key}^{k-1,\ell-1}(u)$. □

Note that if v is stationary and remains stationary after the expansion of u, nothing can be said about their key values.

Proposition 1

During the kth iteration, the key value of the vertex with the highest priority in the queue is nondecreasing during the execution of the $\mathtt{Replan}(\mathcal{G}^k, x_{\text{init}})$ *procedure.*

Proof. Let the vertex u be the nonstationary vertex of the highest priority in the queue at stage $(k-1, \ell-1)$ of the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure and let $k_{\min}^{k-1,\ell-1}$ denote the corresponding minimum key value in the queue at this time, i.e., let $k_{\min}^{k-1,\ell-1} = \texttt{Key}^{k-1,\ell-1}(u)$. We claim that $k_{\min}^{k-1,\ell-1} \preceq k_{\min}^{k-1,\ell}$ for $0 < \ell \leq n_k$, where n_k is the last step of the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure.

To see this, let u be the vertex selected for expansion and made stationary at the ℓth step during the kth iteration. The contents of the queue may change in two different cases: a stationary vertex v may become nonstationary, in which case $\texttt{J}^{k-1,\ell-1}(v) = \bar{\texttt{J}}^{k-1,\ell-1}(v)$, $\texttt{J}^{k-1,\ell}(v) \neq \bar{\texttt{J}}^{k-1,\ell}(v)$ and $\texttt{Key}^{k-1,\ell-1}(u) \prec \texttt{Key}^{k-1,\ell}(v)$ according to Lemma 3, Case ii); and/or a nonstationary vertex, say v, may remain nonstationary, in which case $\texttt{J}^{k-1,\ell-1}(v) \neq \bar{\texttt{J}}^{k-1,\ell-1}(v)$, $\texttt{J}^{k-1,\ell}(v) \neq \bar{\texttt{J}}^{k-1,\ell}(v)$ and hence $\texttt{Key}^{k-1,\ell-1}(u) \preceq \texttt{Key}^{k-1,\ell}(v)$, according to Lemma 3 Case i). Hence, after the inclusion of these potentially nonstationary vertices into the queue, the key value of the next nonstationary vertex of the highest priority will be greater than or equal to $\texttt{Key}^{k-1,\ell-1}(u)$, and thus $k_{\min}^{k-1,\ell-1} \preceq k_{\min}^{k-1,\ell}$, for $0 < \ell \leq n_k$, thus completing the proof. □

Proposition 2

If a vertex $v_i \in V^k$ is stationary with $\texttt{Key}^{k-1,\ell-1}(v) \preceq k_{\min}^{k-1,\ell-1}$ where $k_{min}^{k-1,\ell-1}$ is the key of highest priority vertex in the queue at stage $(k-1, \ell-1)$ in the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure, then it remains stationary until the procedure terminates.

Proof. Let us assume that vertex $v_i \in V^k$ is stationary with $\texttt{Key}^{k-1,\ell_i-1}(v_i) \preceq k_{\min}^{k-1,\ell_i-1}$, where $k_{\min}^{k-1,\ell_i-1}$ is the key value of the nonstationary vertex of the highest priority in the queue at the ℓ_ith step of the kth iteration of the algorithm. Suppose, on the contrary, that there exists a vertex $v_j \in V^k$ at some future step such that the vertex v_i becomes nonstationary after expansion of the vertex v_j at the ℓ_jth step. It holds that $\texttt{Key}^{k-1,\ell_j-1}(v_j) \prec \texttt{Key}^{k-1,\ell_j}(v_i)$ according to Lemma 3 Case ii) and $k_{\min}^{k-1,\ell_i-1} \preceq k_{\min}^{k-1,\ell_j-1} = \texttt{Key}^{k-1,\ell_j-1}(v_j)$ since $\ell_i \leq \ell_j$ according to Theorem 1. It follows that $\texttt{Key}^{k-1,\ell_i-1}(v_i) \preceq k_{\min}^{k-1,\ell_i-1} \preceq$

$\texttt{Key}^{k-1,\ell_j-1}(v_j) \prec \texttt{Key}^{k-1,\ell_j}(v_i)$, which implies that

$$\begin{aligned}\texttt{Key}^{k-1,\ell_i-1}(v_i) &= [\texttt{h}(x_{\text{init}}, v_i) + \bar{\texttt{J}}^{k-1,\ell_i-1}(v_i), \bar{\texttt{J}}^{k-1,\ell_i-1}(v_i)] \\ &\prec \texttt{Key}^{k-1,\ell_j}(v_i) = [\texttt{h}(x_{\text{init}}, v_i) + \bar{\texttt{J}}^{k-1,\ell_j}(v_i), \bar{\texttt{J}}^{k-1,\ell_j}(v_i)].\end{aligned}$$

This results in $\bar{\texttt{J}}^{k-1,\ell_i-1}(v_i) < \bar{\texttt{J}}^{k-1,\ell_j}(v_i)$ which contradicts Lemma 1. Hence, vertex v_i remains stationary until the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates. □

Theorem 1

Given the graph $\mathcal{G}^k = (V^k, E^k)$, a goal set $\mathcal{X}_{\text{goal}}$, and an initial state x_{init}, the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure incrementally operates on all nonstationary and promising vertices, and only those. Thus, the $\bar{J}$-values of the promising vertices are equal to their respective optimal cost-to-go values when the procedure terminates at the end of the kth iteration.

Proof. Let V^k_{exp} denote the set of all vertices that are expanded during the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure. When a vertex is expanded at some step in the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure it becomes stationary at the next step and remains stationary until the procedure terminates (i.e., its key value does not change). We need to show that all nonstationary and promising vertices are expanded before the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates, that is, we need to show that $V^k_{\text{prom}} \subseteq V^k_{\text{exp}}$. To this end, let ℓ_t be the step at which the termination condition in Algorithm 3 is satisfied for the first time, i.e., let $\texttt{Key}^{k-1,\ell_t-1}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,\ell_t-1}(v_t)$ where v_t is the nonstationary vertex that is selected for expansion at the ℓ_tth step, and assume that there exist a nonstationary and promising vertex $v_i \in V^k_{\text{prom}}$, satisfying $[\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)] \prec [\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})]$, which is not expanded before the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates at the end of the kth iteration.

Should Algorithm 3 were allowed to expand all nonstationary vertices, then vertex v_i would be selected for expansion at the ℓ_ith step after the vertex v_t (i.e., $\ell_t \leq \ell_i$) and $\texttt{Key}^{k-1,\ell_i}(v_i) = [\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)]$ according to Theorem 6 of [71]. Since the vertex v_i is expanded after the vertex v_t, its key value has lower priority than the key value of v_t, so

that $\texttt{Key}^{k-1,\ell_t}(v_t) \preceq \texttt{Key}^{k-1,\ell_i}(v_i)$ as shown in Proposition 1. It follows that

$$\begin{aligned}[\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})] &= \texttt{Key}^{k-1,\ell_t-1}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,\ell_t-1}(v_t) \\ &= \texttt{Key}^{k-1,\ell_t}(v_t) \preceq \texttt{Key}^{k-1,\ell_i}(v_i) \\ &= [\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)],\end{aligned}$$

which yields that $[\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})] \preceq [\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)]$. This implies that the vertex v_i is a non-promising vertex, leading to a contradiction. Hence, all promising vertices are expanded before the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates at the beginning of the ℓ_tth step, and $V^k_{\text{prom}} \subseteq V^k_{\text{exp}}$.

To show that $V^k_{\text{exp}} \subseteq V^k_{\text{prom}}$ let us assume that a nonstationary vertex $v_i \in V^k$ is expanded at the ℓ_ith step in the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure, i.e., $v_i \in V^k_{\text{exp}}$. It then follows that $\texttt{Key}^{k-1,\ell_i}(v_i) = [\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)]$ where $\texttt{f}^k(v_i) = \texttt{h}(x_{\text{init}}, v_i) + \texttt{J}^{*k}(v_i)$ as shown in Theorem 6 of [71]. Also, as shown in Lemma 9 of [71], $\texttt{Key}^{k-1,\ell_i}(v_i) \prec [\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})]$. It follows that $[\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)] \prec [\texttt{value}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})]$, which implies that the vertex v_i is a promising vertex, and hence $V^k_{\text{exp}} \subseteq V^k_{\text{prom}}$. This implies that the set of expanded vertices is equal to the set of nonstationary and promising vertices. Therefore, we have that $\texttt{Key}^{k-1,\ell_i}(v_i) = [\texttt{f}^k(v_i), \texttt{J}^{*k}(v_i)] = [\texttt{h}(x_{\text{init}}, v_i) + \bar{\texttt{J}}^{k-1,\ell_i}(v_i), \bar{\texttt{J}}^{k-1,\ell_i}(v_i)]$ for any promising vertex v_i where $\ell_i < \ell_t$ is the step when the vertex v_i is expanded. Finally, it follows that $\texttt{J}^{k-1,\ell_i}(v_i) = \bar{\texttt{J}}^{k-1,\ell_i}(v_i) = \texttt{J}^{*k}(v_i)$ for any promising vertex $v_i \in V^k$ when the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates at the beginning of the ℓ_tth step. □

Theorem 2

Given the graph $\mathcal{G}^k = (V^k, E^k)$ and an initial state x_{init}, the $\bar{J}$-value of x_{init} is equal to the optimal cost-to-go value when the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ *procedure terminates at the end of the kth iteration.*

Proof. First, we claim that x_{init} is never expanded before the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates. Hence, it remains nonstationary (i.e., it has an infinite J-value and finite $\bar{J}$-value) all the time. This follows from the fact that whenever x_{init} is selected for

expansion, the condition at Line 2 in Algorithm 3 is satisfied. Therefore, the procedure terminates immediately without expanding x_{init}. We therefore only need to show that, upon termination, the optimal cost-to-go value for x_{init} in the current graph has been computed by the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure even though x_{init} is never expanded. To this end, let ℓ_t be the step at which the termination condition in Algorithm 3 is satisfied for the first time, i.e., let $\texttt{Key}^{k-1,\ell_t-1}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,\ell_t-1}(v_t)$ where v_t is the nonstationary vertex that is selected for expansion at the ℓ_tth step. If the assertion were not true, and the Algorithm 3 were allowed to continue expanding all vertices, then necessarily x_{init} would be selected for expansion after vertex v_t. Let ℓ_ith be the step when x_{init} is selected for expansion. According to the assumption we have that $\ell_t \leq \ell_i$. We then have that $\texttt{Key}^{k-1,\ell_i}(x_{\text{init}}) = [\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})]$ where $\texttt{f}^k(x_{\text{init}}) = \texttt{h}(x_{\text{init}}, x_{\text{init}}) + \texttt{J}^{*k}(x_{\text{init}})$ as shown in Theorem 6 of [71]. Since v_t is selected for expansion before x_{init}, its key value has higher priority, i.e., $\texttt{Key}^{k-1,\ell_t}(v_t) \preceq \texttt{Key}^{k-1,\ell_i}(x_{\text{init}})$ as shown in Proposition 1. Also, since $\texttt{J}^{*k}(x_{\text{init}})$ is the optimal cost-to-go value of x_{init} that can be achieved in the current graph $\mathcal{G}^k$, it follows that $[\texttt{f}^k(x_{\text{init}}), \texttt{g}^{*k}(x_{\text{init}})] \preceq \texttt{Key}^{k-1,\ell}(x_{\text{init}})$ for all $\ell \geq 0$. One then obtains

$$\begin{aligned}
[\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})] &\preceq \texttt{Key}^{k-1,\ell_t-1}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,\ell_t-1}(v_t) \\
&= \texttt{Key}^{k-1,\ell_t}(v_t) \preceq \texttt{Key}^{k-1,\ell_i}(x_{\text{init}}) \\
&= [\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})],
\end{aligned}$$

which yields

$$\begin{aligned}
\texttt{Key}^{k-1,\ell_t-1}(x_{\text{init}}) &= [\texttt{f}^k(x_{\text{init}}), \texttt{J}^{*k}(x_{\text{init}})] \\
&= [\texttt{h}(x_{\text{init}}, x_{\text{init}}) + \bar{\texttt{J}}^{k-1,\ell_t-1}(x_{\text{init}}), \bar{\texttt{J}}^{k-1,\ell_t-1}(x_{\text{init}})].
\end{aligned}$$

Hence, $\bar{\texttt{J}}^{k-1,\ell_t-1}(x_{\text{init}}) = \texttt{J}^{*k}(x_{\text{init}})$ when the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates at the beginning of the ℓ_tth step, and the correct cost-to-go value of x_{init} is computed at that step. Running the algorithm after the ℓ_tth step will therefore not improve the cost-to-go value of x_{init}. □

Theorem 3

Let $\mathcal{G}^k = (V^k, E^k)$ and $\mathcal{T}^k = (V^k, E_s^k)$ denote the graph and the spanning tree that is rooted at the vertex $\mathcal{X}_{\text{goal}}$ and constructed by the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ *procedure, respectively, at the kth iteration, and let σ be the corresponding unique path from any head vertex $v_i \in V^k$ to x_{goal} encoded by the tree $\mathcal{T}^k$. Then, this path is the lowest-cost path with respect to the current graph $\mathcal{G}^k$ if the head vertex v_i is a promising vertex or if $v_i = x_{\text{init}}$.*

Proof. Let σ denote the unique path from the head vertex v_i to x_{goal} encoded in the tree $\mathcal{T}^k$ such that $\sigma(\tau_j) = v_{p_j}$ for $0 \leq \tau_j \leq 1$ and $j = 0, 1, \ldots, n_i$ where $\tau_0 = 0, \tau_{n_i} = 1$ and v_{p_j} are vertices along the path. We have $\sigma(\tau_0) = v_{p_0} = v_i$ and $\sigma(\tau_{n_i}) = v_{p_{n_i}} = x_{\text{goal}}$. Also, the parent vertex of each vertex is given by $\mu(v_{p_j}) = v_{p_{j+1}}$. Let us consider the paths in $\mathcal{G}^k$ from any promising vertex or x_{init} to $\mathcal{X}_{\text{goal}}$. First, note that if the head vertex of the path σ is a promising vertex or x_{init}, then all of the intermediate vertices along the path σ are promising. Second, when the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates at the end of the kth iteration, the $\bar{J}$-values of all promising vertices and of x_{init} are equal to their corresponding optimal cost-to-go values as shown in Theorem 3. Hence, $\bar{\texttt{J}}^{k-1,n_k}(v) = \texttt{J}^{*k}(v)$ for all $v \in V_{\text{prom}}^k \cup \{x_{\text{init}}\}$. We can thus write

$$\begin{aligned}
\texttt{J}^{*k}(v_{p_j}) &= \bar{\texttt{J}}^{k-1,n_k}(v_{p_j}) = \texttt{J}^{k-1,n_k}(v_{p_j}) \\
&= \texttt{c}(v_{p_j}, v_{p_{j+1}}) + \texttt{J}^{k-1,n_k}(v_{p_{j+1}}) \\
&= \texttt{c}(v_{p_j}, v_{p_{j+1}}) + \bar{\texttt{J}}^{k-1,n_k}(v_{p_{j+1}}) \\
&= \texttt{c}(v_{p_j}, v_{p_{j+1}}) + \texttt{J}^{*k}(v_{p_{j+1}}),
\end{aligned}$$

for any vertex v_{p_j} along the path σ, which shows that each vertex on σ has achieved the optimal cost-to-go value. □

Theorem 4

Let $\mathcal{G}^k = (V^k, E^k)$ denote the graph at the kth iteration. Then, the relationship $\texttt{J}^{*k+1}(x_{\text{init}}) \leq \texttt{J}^{*k}(x_{\text{init}})$ holds for all k, and the estimated optimal cost decreases with each iteration.

Proof. Without loss of generality, let $k \geq N$, where N is the iteration such that $x_{\text{init}} \in V^N$, i.e., the cost of the lowest-cost path from x_{init} to $\mathcal{X}_{\text{goal}}$ encoded by the graph $\mathcal{G}^k$ is finite. Otherwise, the claim holds trivially. Let now n_k denote the index of the new sampled vertex, which is created in the $\texttt{Extend}(\mathcal{G}^{k-1}, x_{\text{rand}})$ procedure at the beginning of the kth iteration, that is, let $v_{n_k} = x_{\text{new}}$. The J-value of v_{n_k} is initialized with infinity, i.e., $\texttt{J}^{k-1,0}(v_{n_k}) = \infty$ and its $\bar{J}$-value can be a finite value or infinite (the latter occurs if all neighbor vertices have infinite J-values). Therefore, the new vertex can be either stationary with infinite key value (in which case $\texttt{J}^{k-1,0}(v_{n_k}) = \bar{\texttt{J}}^{k-1,0}(v_{n_k}) = \infty$) or nonstationary with finite key value (in which case $\texttt{J}^{k-1,0}(v_{n_k}) = \infty, \bar{\texttt{J}}^{k-1,0}(v_{n_k}) < \infty$). If the new vertex is stationary, then it is not inserted into the queue. The $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure thus terminates without updating the cost-to-go value of any vertex when it is subsequently called after the $\texttt{Extend}(\mathcal{G}^{k-1}, x_{\text{rand}})$ procedure. Therefore, the lowest-cost path computed in the previous iteration will not be modified, and thus $\texttt{J}^{*k}(x_{\text{init}}) = \texttt{J}^{*k-1}(x_{\text{init}})$. When the new vertex is nonstationary, it is inserted into the queue. There are two different cases to consider depending on the type of the new vertex:

Case 1: Let us consider the case when v_{n_k} is a non-promising vertex. Let us also assume that $\texttt{Key}^{k-1,0}(v_{n_k}) \prec \texttt{Key}^{k-1,0}(x_{\text{init}})$. Then the cost-to-go value of v_{n_k} is updated at Line 4 in the Algorithm 3, since all other nonstationary vertices in the queue have lower priority than x_{init}, i.e., $\texttt{Key}^{k-1,0}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,0}(v_i)$ for all nonstationary $v_i \in V^k$. This implies that v_{n_k} is a promising vertex, which leads to a contradiction. Therefore, necessarily $\texttt{Key}^{k-1,0}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,0}(v_{n_k})$ and the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure will terminate immediately without updating the cost-to-go value of any vertex. Thus, the lowest-cost path computed during the previous iteration will not be modified, and hence, $\texttt{J}^{*k}(x_{\text{init}}) = \texttt{J}^{*k-1}(x_{\text{init}})$.

Case 2: Let us consider the case when v_{n_k} is a promising vertex. First, let us assume that $\texttt{Key}^{k-1,0}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,0}(v_{n_k})$ in the very beginning of the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure.

We then have $\texttt{Key}^{k-1,0}(x_{\text{init}}) \preceq \texttt{Key}^{k-1,0}(v_i)$ for all nonstationary $v_i \in V^k$. Therefore, the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure terminates without updating the cost-to-go value of any vertex. This implies that v_{n_k} is a non-promising vertex, leading to a contradiction. Hence, we have that $\texttt{Key}^{k-1,0}(v_{n_k}) \prec \texttt{Key}^{k-1,0}(x_{\text{init}})$ and the cost-to-go value of the new vertex is updated at the first step in the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure. Since the sequence computed by the Asynchronous Value Iteration algorithm converges to the optimal cost-to-go values from any initial values, the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure will compute the lowest-cost path encoded by $\mathcal{G}^k$ by expanding all nonstationary and promising vertices. If the cost of the lowest-cost path from x_{init} to $\mathcal{X}_{\text{goal}}$ passing through the new vertex is better than that of the lowest-cost path computed at the previous iteration, we then have $\texttt{J}^{*k}(x_{\text{init}}) < \texttt{J}^{*k-1}(x_{\text{init}})$; otherwise $\texttt{J}^{*k}(x_{\text{init}}) = \texttt{J}^{*k-1}(x_{\text{init}})$.

To summarize, we have a monotonic decrease in the cost of the lowest-cost path encoded by $\mathcal{G}^k$, and thus we have that $\texttt{J}^{*k+1}(x_{\text{init}}) \leq \texttt{J}^{*k}(x_{\text{init}})$ for all $N \leq k$, where N is the first iteration when a finite cost path from x_{init} to $\mathcal{X}_{\text{goal}}$ is achieved. □

Theorem 5

Let $Y_k^{\text{RRT}^\#}$ denote the cost of the path from x_{init} to x_{goal} computed by the $\text{RRT}^\#$ algorithm at the kth iteration. The $\text{RRT}^\#$ algorithm is asymptotically optimal, that is, $Y_k^{\text{RRT}^\#} \to c^*$ as $k \to \infty$ with probability one.

Proof. Since the $\text{RRT}^\#$ algorithm adopts the `Extend` procedure of the RRG algorithm, they both create the same graph $\mathcal{G}^k$ at the end of the kth iteration. The $\text{RRT}^\#$ algorithm, in addition, keeps a spanning tree $\mathcal{T}^k$ that is rooted at the vertex x_{goal} and contains lowest-cost path information for a subset of vertices (namely, all promising vertices, along with x_{init}) as shown in Theorem 4. Let Y_k^{RRG} denote the cost of the lowest-cost path from x_{init} to x_{goal} in $\mathcal{G}^k$, which is computed by the RRG algorithm, at the kth iteration. In the $\text{RRT}^\#$ algorithm, this path is incrementally computed and stored in a spanning tree $\mathcal{T}^k$ by the $\texttt{Replan}(\mathcal{G}^k, x_{\text{init}})$ procedure at the end of each iteration, that is,

$Y_k^{\mathrm{RRT}^\#} = \bar{\mathrm{J}}^k(x_{\mathrm{init}}) = Y_k^{\mathrm{RRG}}$. In addition, since the RRG algorithm is asymptotically optimal, we have that $Y_k^{\mathrm{RRG}} \to c^*$ as $k \to \infty$ with probability one, where c^* is the cost of the optimal path from x_{init} to x_{goal} in $\mathcal{X}_{\mathrm{free}}$. Hence, it also follows that $Y_k^{\mathrm{RRT}^\#} = Y_k^{\mathrm{RRG}} \to c^*$ as $k \to \infty$ with probability one, which completes the proof. □

4.6 Numerical Simulations

The $\mathrm{RRT}^\#$ algorithm was developed in C++ and run on a computer with a 2.40 GHz processor and 12GB RAM running the Ubuntu 11.10 Linux operating system. A Fibonacci heap was implemented as priority queue to store inconsistent vertices during the search [49]. Extensive simulations were run to compare the performance of the $\mathrm{RRT}^\#$ algorithm with the RRT^* algorithm, whose C implementation is available to download from the RRT^* author's website (`http://sertac.scripts.mit.edu/rrtstar/`).

Both $\mathrm{RRT}^\#$ and RRT^* algorithms were run on three different problem types with the same sample sequence in order to demonstrate the difference in their behavior while growing the tree. All problems tested require finding an optimal path in a square environment minimizing the Euclidean path length. The heuristic value of a vertex is the Euclidean distance from the vertex to the goal. In the first problem type, there are no obstacles in the environment, whereas there are some box-like obstacles in the second and third problem types. In the third problem type, the environment is more cluttered than the one in the second problem type, containing many widely distributed small obstacles.

For the first problem type, the trees computed by both algorithms at different stages are shown in Figure 1. The initial state is plotted as a yellow square and the goal region is shown in blue with magenta border (upper right). The minimal-length path is shown in red. As shown in Figure 1, the best path computed by the $\mathrm{RRT}^\#$ algorithm converges to the optimal path. As mentioned earlier, one of the important differences between the RRT^* and $\mathrm{RRT}^\#$ algorithms is that the latter classifies the vertices in one of the following four categories based on the values of its $(\mathtt{g}(v), \mathtt{lmc}(v))$ pair: Consistent with finite key value

(shown in green), consistent with infinite key value (shown in black), inconsistent with finite key value (shown in blue), and inconsistent with infinite g-value and finite lmc-value (shown in red).

Since only the points in the relevant region $\mathcal{X}_{\text{rel}}$ have the potential to be part of the optimal path, the RRT$^{\#}$ algorithm tries to approximate $\mathcal{X}_{\text{rel}}$ with the set of promising vertices V_{prom} and tends to stop rewiring the parts of the tree which lie outside of the $\mathcal{X}_{\text{rel}}$ as iterations go to infinity. As seen in Figure 1, for this particular scenario, $\mathcal{X}_{\text{rel}}$ is an elliptic region, which is much smaller than the whole $\mathcal{X}_{\text{free}}$. Therefore, uniform random sampling on $\mathcal{X}_{\text{free}}$ results in too many vertices of different types (green, black, red, and blue vertices) outside of the relevant region during the search. The estimate of $\mathcal{X}_{\text{rel}}$ can be used to implement more intelligent sampling strategies, if needed.

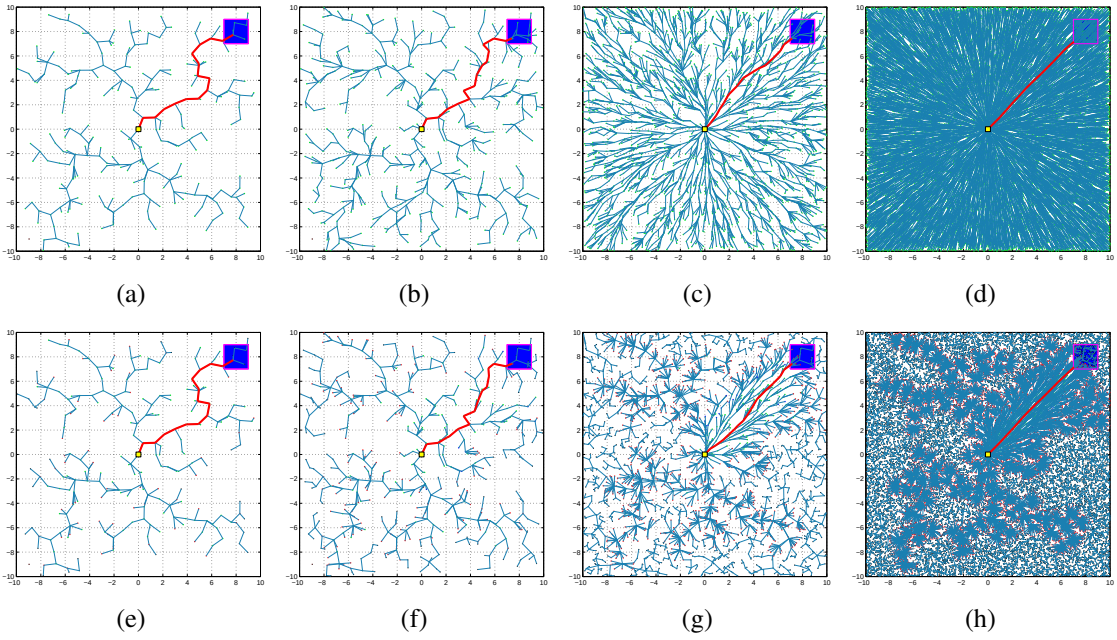


Figure 1: The evolution of the tree computed by RRT* and RRT$^{\#}$ algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2,500 iterations, and (d), (h) is at 25,000 iterations.

In the second problem type, the same experiment was carried out and both algorithms were run in an environment with several obstacles. The configuration of the trees for both

the RRT* and RRT# algorithms at different stages are shown in Figure 2.

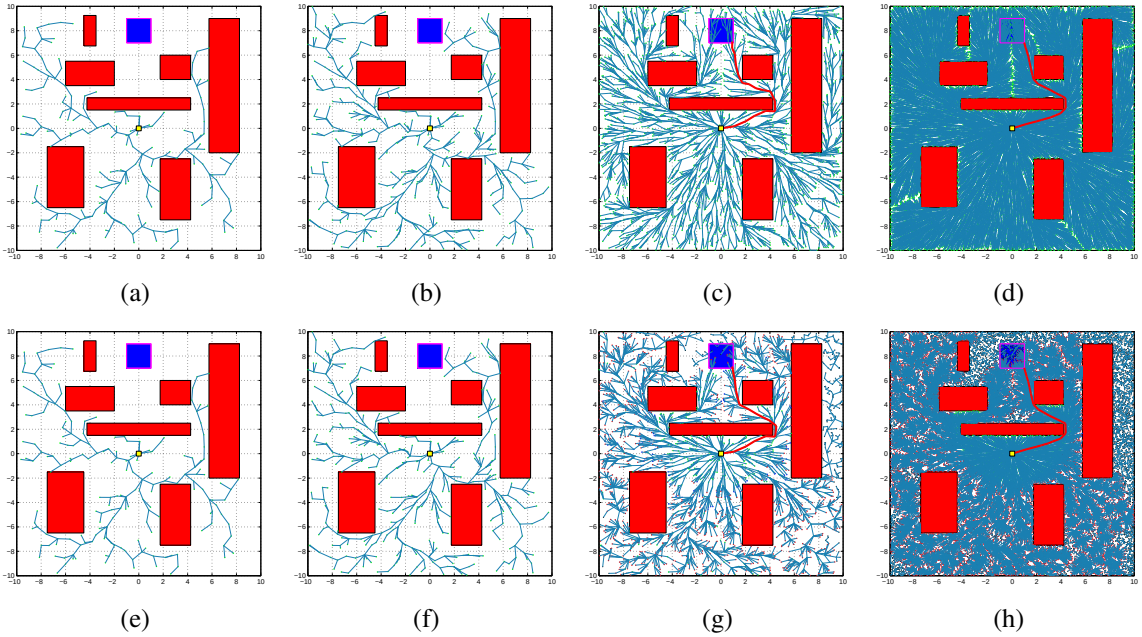


Figure 2: The evolution of the tree computed by RRT* and RRT# algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2,500 iterations, and (d), (h) is at 25,000 iterations.

In the third problem type, both algorithms were run in a more cluttered environment, where there are many different homotopy classes containing the local minimum solution for the problem. As shown in Figure 3, both algorithms switch between paths which have locally best cost, eventually converging to the optimal solution.

Finally, in the fourth problem type, both algorithms were run in a obstacle-free environment where there are different cost zones. The cost coefficient of each zone from top to bottom is 1.5, 0.75, 2.5, 0.75, and 1.5, respectively and 1 elsewhere. As seen in Figure 4, both algorithms compute the optimal path which has longer segments in low-cost zones.

4.7 Variants of the RRT# Algorithm

Too many non-promising vertices are included in the tree computed by the RRT# algorithm as observed in the previous simulations. This is owing to the fact that the RRT#

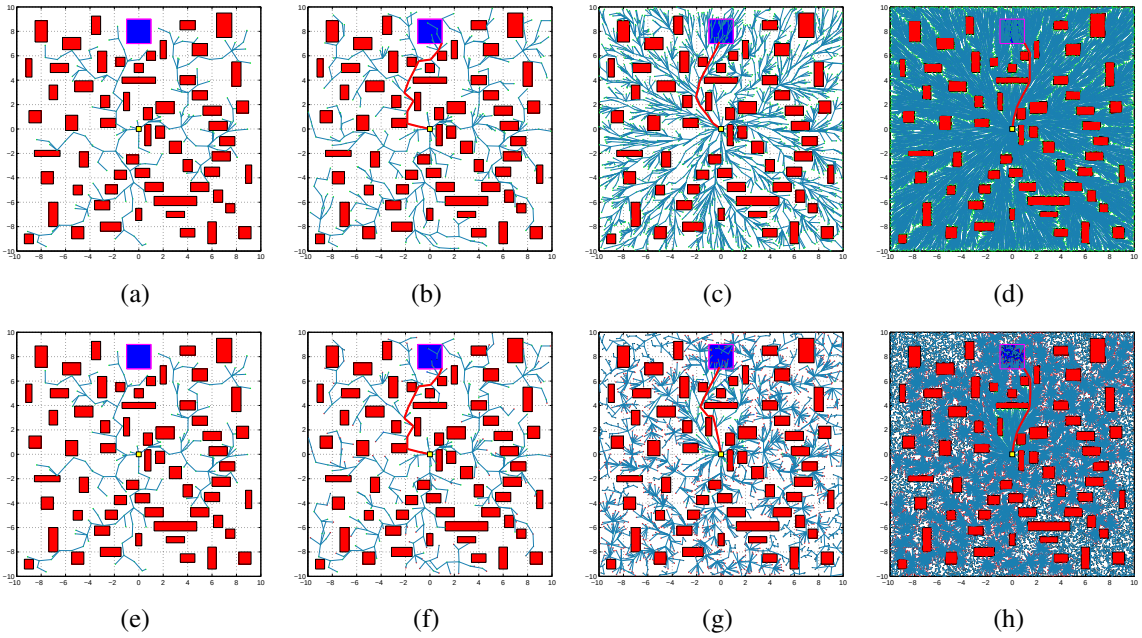


Figure 3: The evolution of the tree computed by RRT^* and $\mathrm{RRT}^{\#}$ algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2,500 iterations, and (d), (h) is at 25,000 iterations.

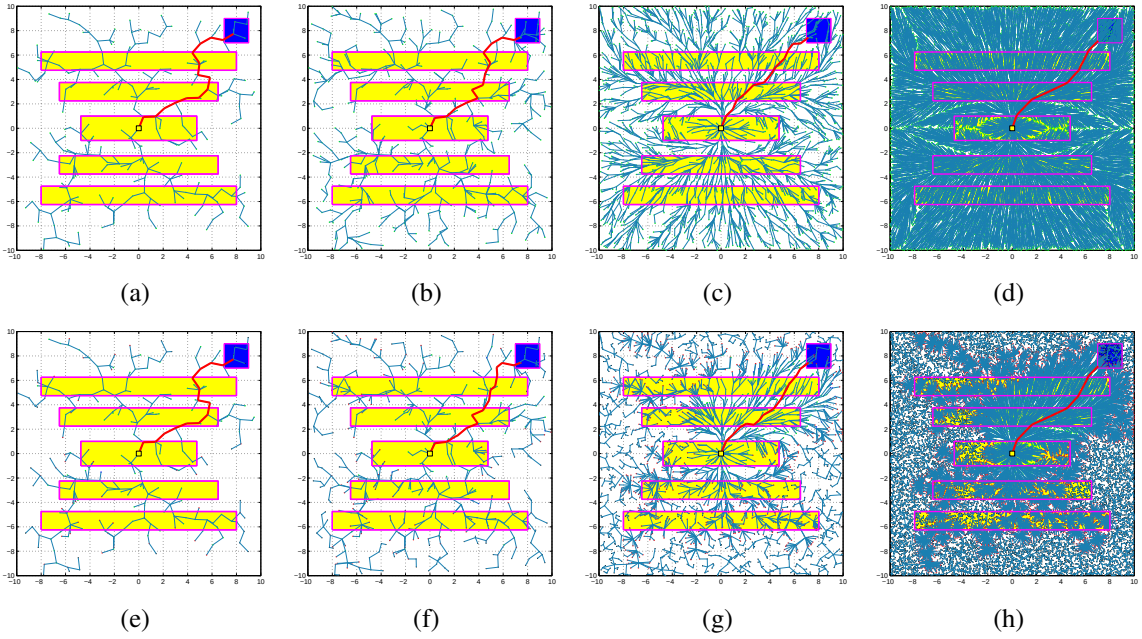


Figure 4: The evolution of the tree computed by RRT^* and $\mathrm{RRT}^{\#}$ algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2,500 iterations, and (d), (h) is at 25,000 iterations.

algorithm includes all new vertices in the graph regardless of their type. A simple vertex selection criterion can be used in the `Extend` procedure in order to prevent the algorithm from growing the tree towards the region outside $\mathcal{X}_{\text{rel}}$. However, being over-selective on vertex inclusion may degrade the performance of the algorithm – and thus lead to a sub-optimal solution – since the cost-to-come value of all vertices, which is used to decide if a new vertex is promising or not, is an estimate of the optimal one. In this section, we propose three variants of the baseline $\text{RRT}^{\#}$ algorithm.

Algorithm 5: `Extend` Procedure for $\text{RRT}_1^{\#}$ Algorithm

```
1  Extend(G,x)
2      (V, E) ← G; E' ← ∅;
3      x_nearest ← Nearest(G, x);
4      x_new ← Steer(x_nearest, x);
5      if ObstacleFree(x_nearest, x_new) then
6          Initialize(x_new);
7          X_near ← Near(G, x_new, |V|);
8          foreach x_near ∈ X_near do
9              if ObstacleFree(x_near, x_new) then
10                 if lmc(x_new) > g(x_near) + c(x_near, x_new) then
11                     lmc(x_new) = g(x_near) + c(x_near, x_new);
12                     parent(x_new) = x_near;
13                 E' ← E' ∪ {(x_near, x_new), (x_new, x_near)};
14         if parent(x_new) ≠ ∅ then
15             V ← V ∪ {x_new};
16             E ← E ∪ E';
17             UpdateQueue(x_new);
18     return G' ← (V, E)
```

$\text{RRT}_1^{\#}$: In the first variant, which is given in Algorithm 5, if a new vertex happens to be consistent with infinite key value (black vertex), it is not included in the graph. This situation can happen if all of the neighbor vertices of the new vertex are inconsistent with infinite g-value and finite lmc-value (red vertices). First, the estimates of the cost-to-come-value of the new vertex x_{new} are initialized with infinite cost, and its parent vertex is set to 'null' in Line 6. Then, a better value for the lmc-value of the

new vertex is searched among its neighbor vertices. During this search, the parent of the new vertex remains unassigned only if there are no any neighboring vertices with finite g-value.

$\mathrm{RRT}_2^{\#}$: In the second variant, the algorithm becomes more selective on vertices to be added to the graph and the "$\texttt{parent}(x_{\mathrm{new}}) \neq \emptyset \wedge \texttt{Key}(\texttt{parent}(x_{\mathrm{new}})) \prec \texttt{Key}(x^*_{\mathrm{goal}})$" condition is checked in Line 14. Simply, a new vertex is included to the graph only if its parent is a promising vertex.

$\mathrm{RRT}_3^{\#}$: Lastly, the third variant is most selective on vertex for inclusion and $\texttt{Key}(x_{\mathrm{new}}) \prec \texttt{Key}(x^*_{\mathrm{goal}})$ condition is checked, that is, only promising new vertices are included in the graph.

4.8 Numerical Simulation

The same experiments as before were carried out for the three variants of the $\mathrm{RRT}^{\#}$ algorithm. As seen in the figures below, all variants successfully prevent the inclusion of vertices which lie in the unfavorable regions of the search space. As seen in Figures 5(e), 7(e), 9(e), and 11(e), the $\mathrm{RRT}_1^{\#}$ algorithm does not include any black vertices in the tree (these are the vertices that are consistent with infinite key value, hence non-promising), but still computes a solution for the problem, which is as good as the one computed by the RRT^* and $\mathrm{RRT}^{\#}$ algorithms. However, there are still many red (i.e., non-promising and inconsistent with infinite g-value and finite lmc-value) vertices included in the tree. This is owing to the fact that these vertices are never made consistent until the last iteration, since they mostly lie outside of $\mathcal{X}_{\mathrm{rel}}$. Therefore, they remain in the priority queue and need to be sorted during each iteration. This makes the `Replan` procedure slower. In the $\mathrm{RRT}_2^{\#}$ algorithm, the number of red vertices included into the tree is reduced by simply enforcing them to have a promising parent vertex for the new vertex that is considered for extension. Red vertices are mostly included into the branches of the tree that are formed outside of the $\mathcal{X}_{\mathrm{rel}}$ during exploration phase. As seen in Figures 5(v), 7(v), 9(v), and 11(v), the $\mathrm{RRT}_2^{\#}$

algorithm tends not to include vertices into the branches of the tree which are very far away from the optimal solution. Lastly, the $\mathrm{RRT}_3^{\#}$ algorithm includes a new vertex into the tree only if it is a promising one. Therefore, all vertices in the tree, other than the goal vertices, are either green or blue, which are located around the boundary of $\mathcal{X}_{\mathrm{rel}}$.

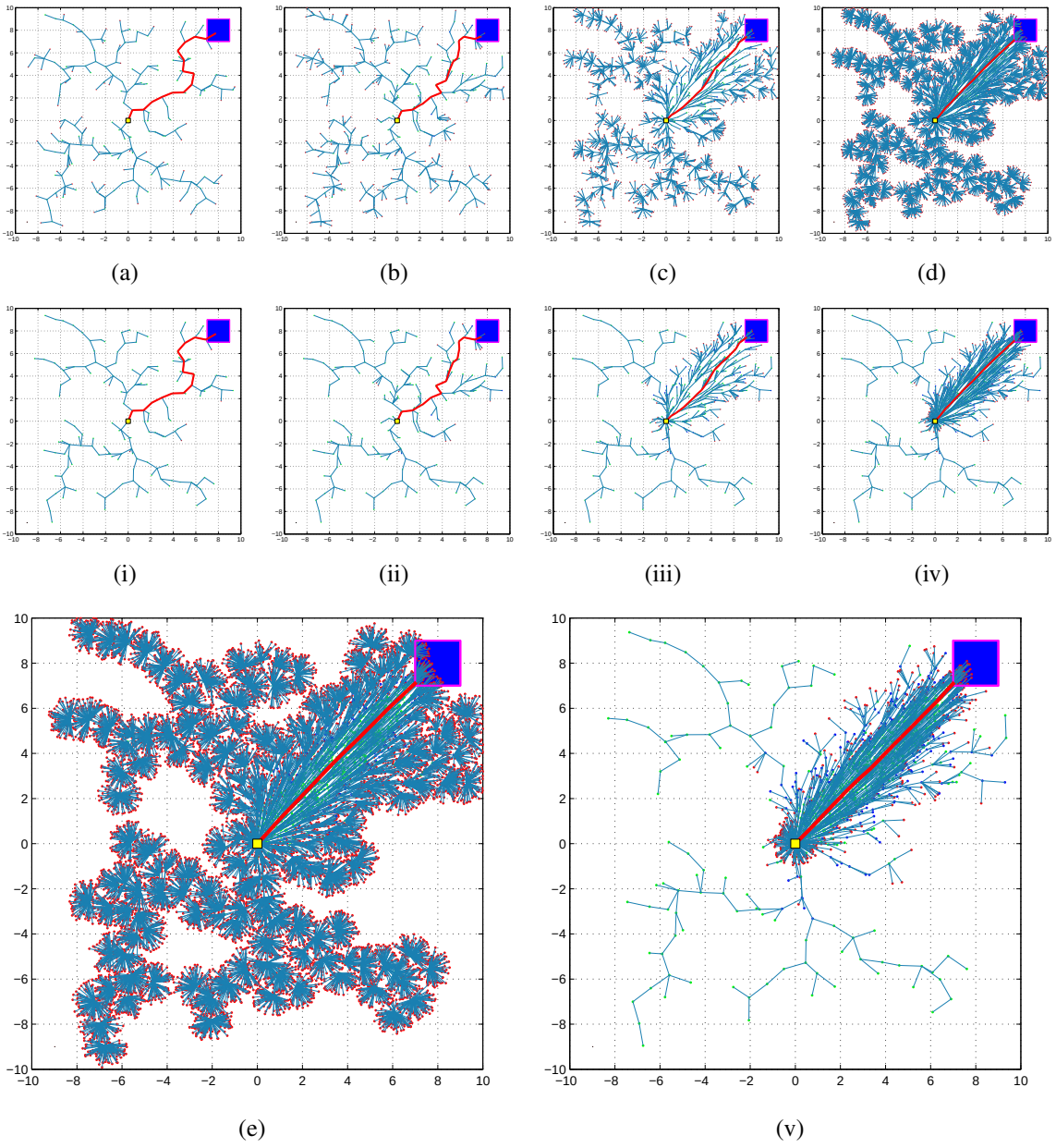


Figure 5: The evolution of the tree computed by $\mathrm{RRT}_1^{\#}$ and $\mathrm{RRT}_2^{\#}$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2,500 iterations, (d), (iv) is at 10,000 iterations, and (e), (v) is at 25,000 iterations.

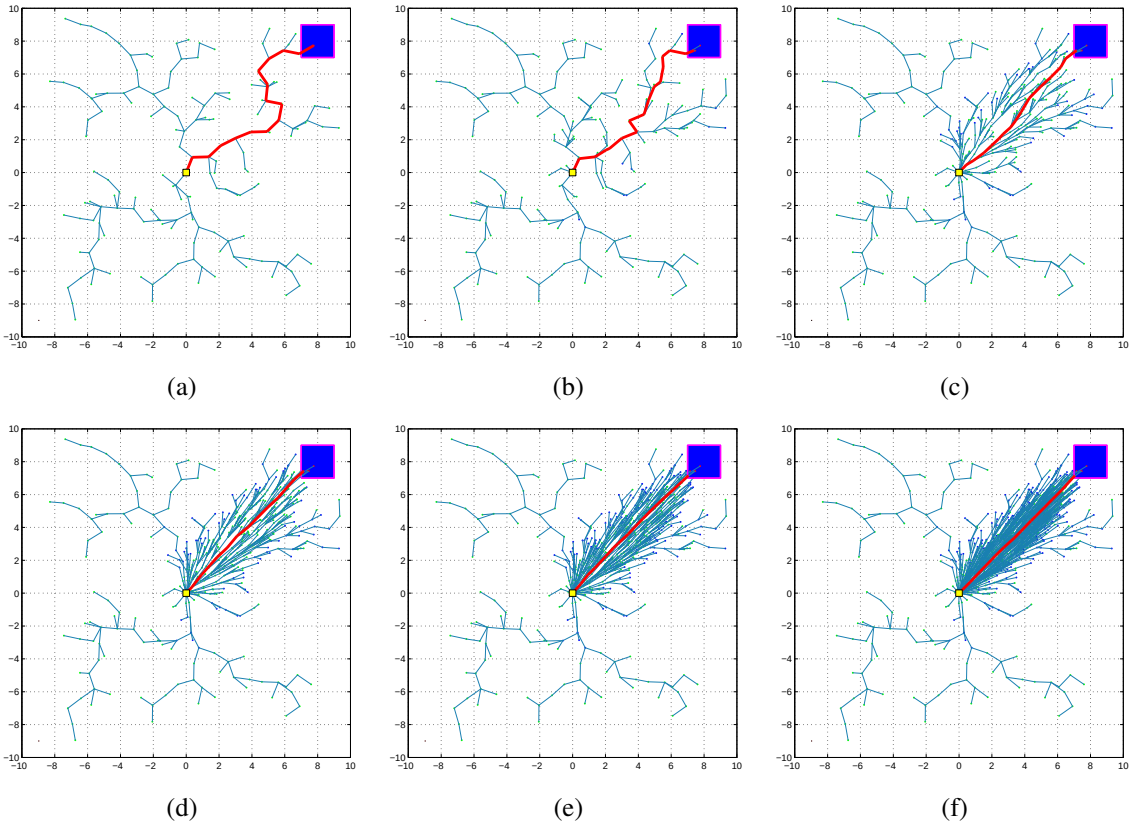


Figure 6: The evolution of the tree computed by $\mathrm{RRT}_3^{\#}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2,500 iterations, in (d) is at 5,000 iterations, in (e) is at 10,000 iterations, and in (f) is at 25,000 iterations.

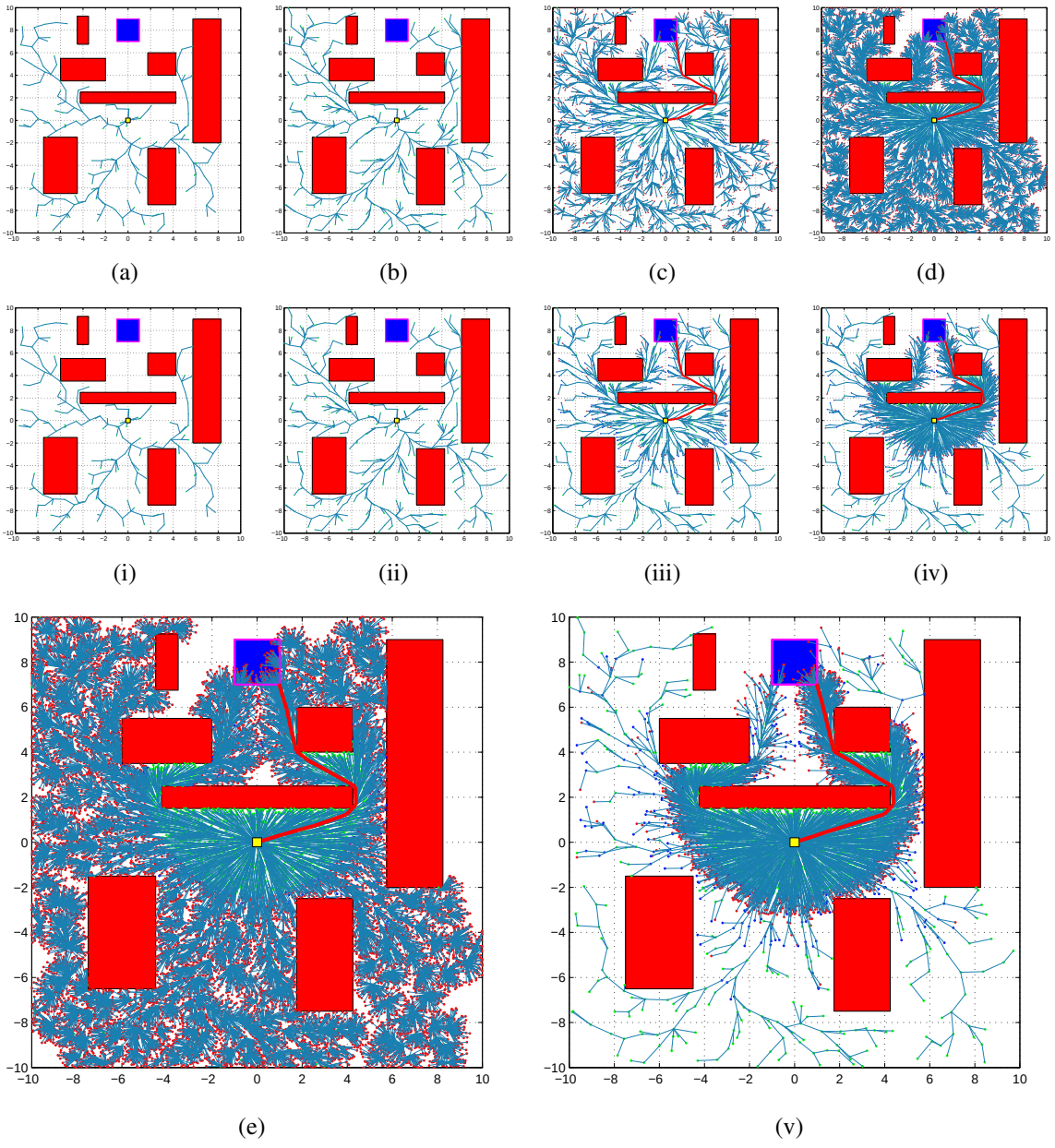


Figure 7: The evolution of the tree computed by $\mathrm{RRT}_1^{\#}$ and $\mathrm{RRT}_2^{\#}$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2,500 iterations, (d), (iv) is at 10,000 iterations, and (e), (v) is at 25,000 iterations.

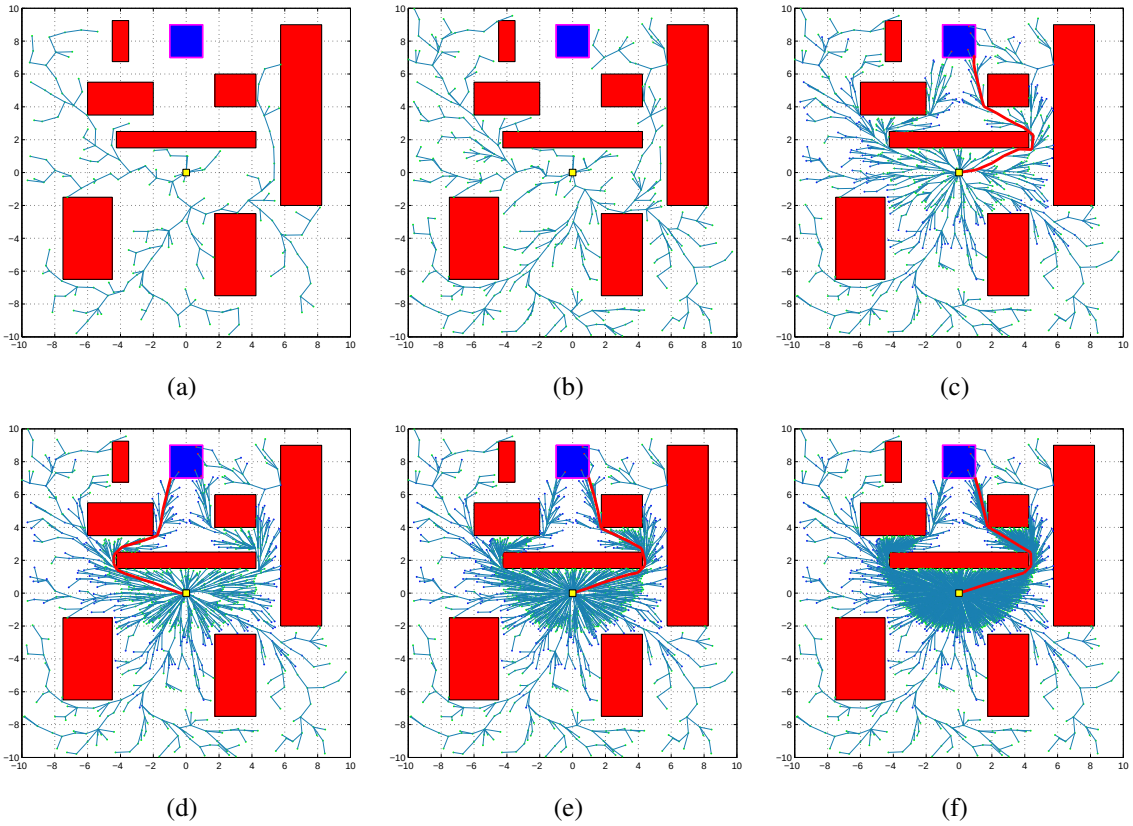


Figure 8: The evolution of the tree computed by $\mathrm{RRT}_3^{\#}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2,500 iterations, in (d) is at 5,000 iterations, in (e) is at 10,000 iterations, and in (f) is at 25,000 iterations.

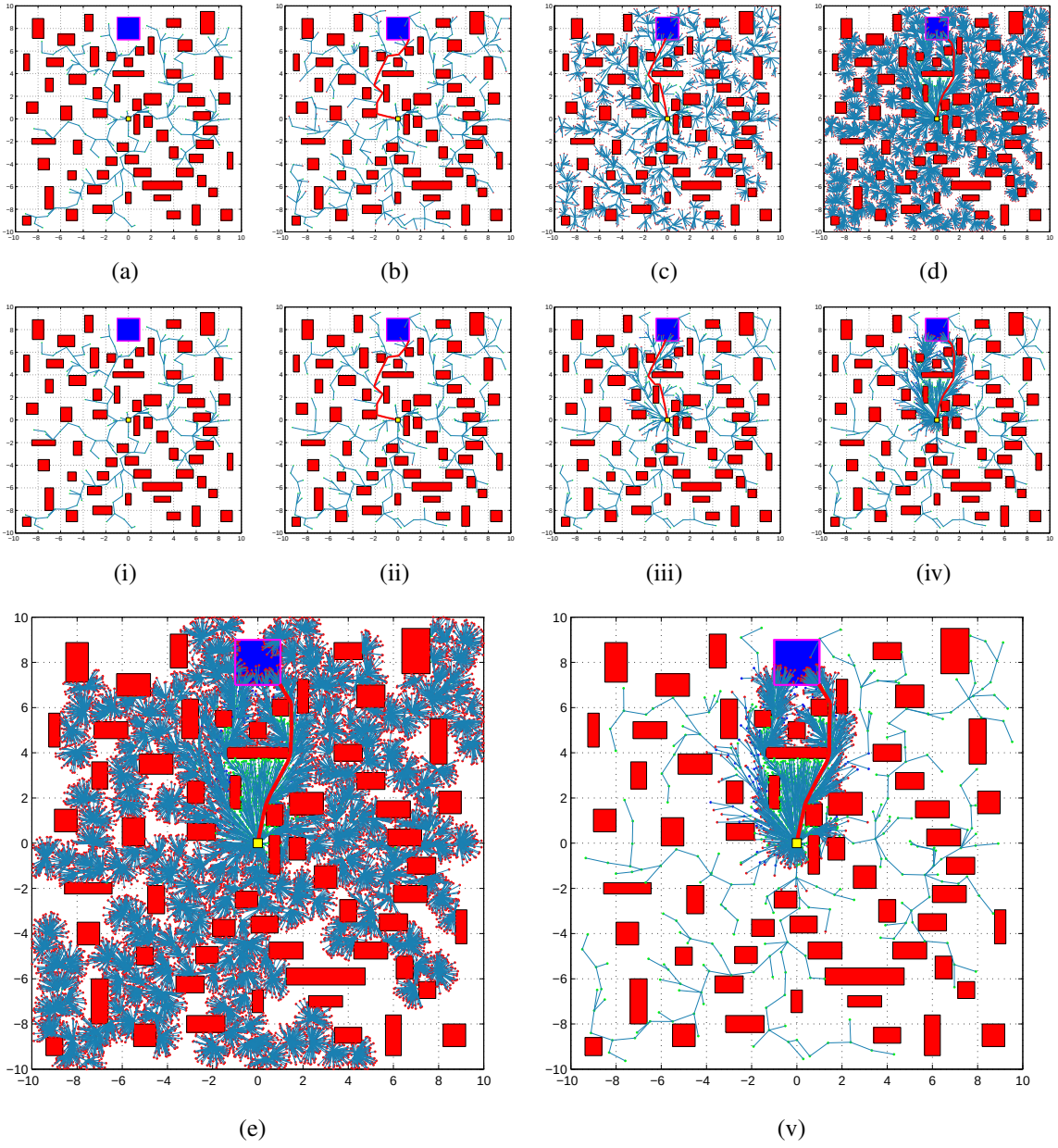


Figure 9: The evolution of the tree computed by $\mathrm{RRT}_1^{\#}$ and $\mathrm{RRT}_2^{\#}$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2,500 iterations, (d), (iv) is at 10,000 iterations, and (e), (v) is at 25,000 iterations.

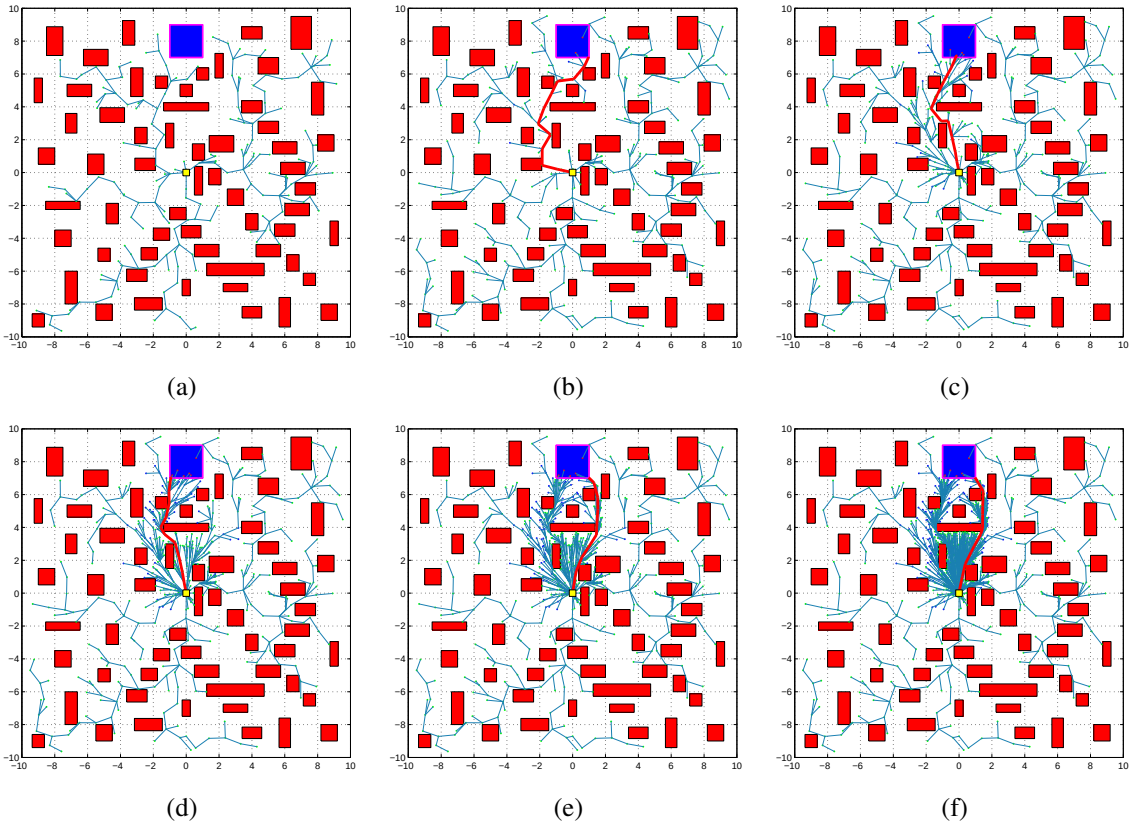


Figure 10: The evolution of the tree computed by $\mathrm{RRT}_3^{\#}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2,500 iterations, in (d) is at 5,000 iterations, in (e) is at 10,000 iterations, and in (f) is at 25,000 iterations.

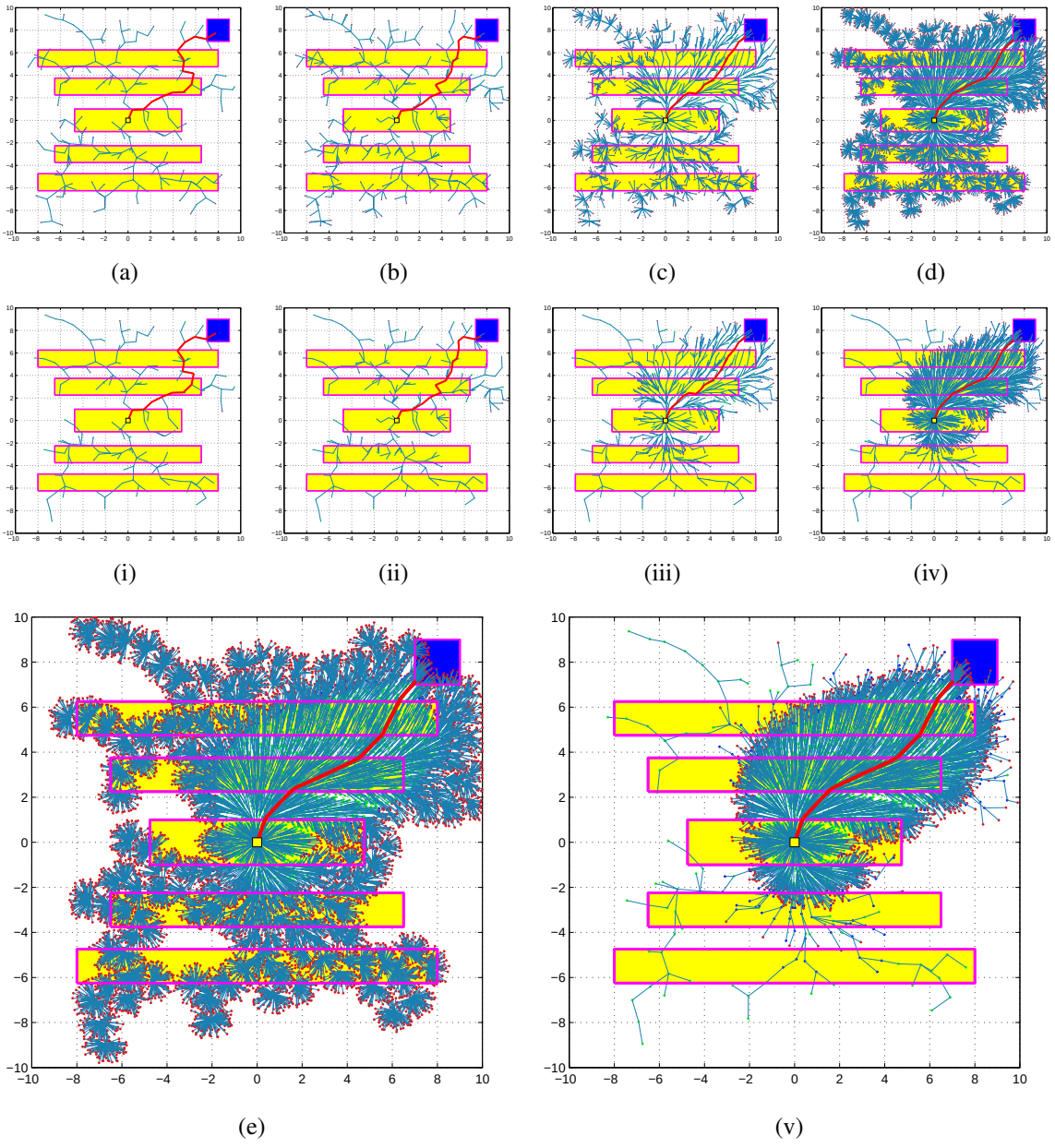


Figure 11: The evolution of the tree computed by $\mathrm{RRT}_1^{\#}$ and $\mathrm{RRT}_2^{\#}$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2,500 iterations, (d), (iv) is at 10,000 iterations, and (e), (v) is at 25,000 iterations.

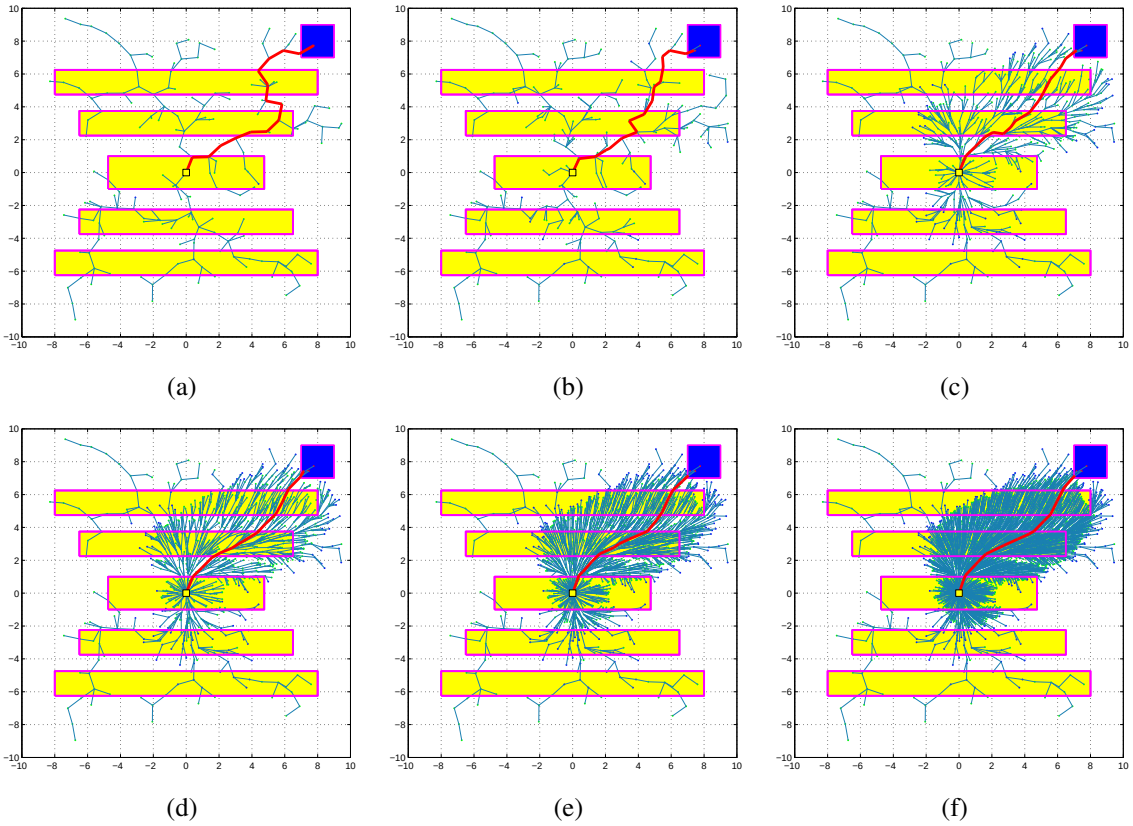


Figure 12: The evolution of the tree computed by $\mathrm{RRT}_3^{\#}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2,500 iterations, in (d) is at 5,000 iterations, in (e) is at 10,000 iterations, and in (f) is at 25,000 iterations.

4.9 Numerical Simulations in High-dimensional Planning Problems

We tested the RRT*, the RRT$^\#$ and its variant algorithms on several scenarios to evaluate their performance.

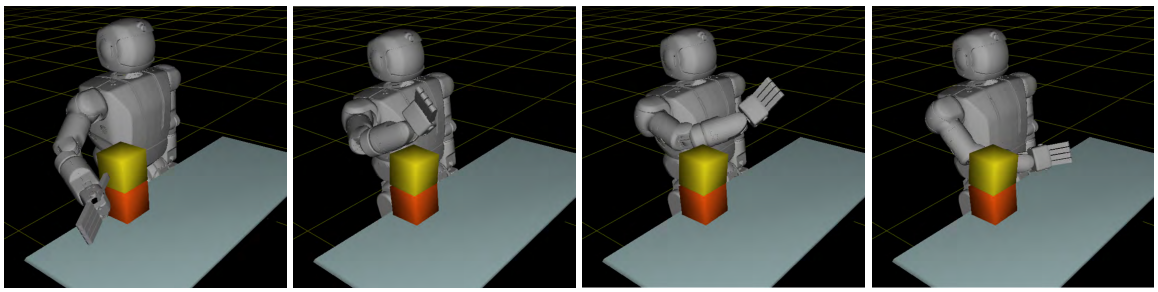


Figure 13: Initial and goal configurations of HUBO (6D)

4.9.1 Motion Planning for Single-Arm Manipulation (6 DoFs)

In this section, we test all algorithms on planning problems in high-dimensional search spaces. First, we simulate a simple workspace in which there are a table and two boxes along with a humanoid robot (HUBO), as shown in Figure 13. At the initial step, the HUBO is at rest with its right arm on one side of the two boxes, and is tasked to move its right arm to the other side of the boxes as, shown in the left-most and right-most subfigures of Figure 13, respectively.

Table 1: Results for Single-Arm Planning Problem

Solution		RRT*	RRT$^\#$	RRT$^\#_1$	RRT$^\#_2$	RRT$^\#_3$
First	Time (s)	15.38 (7.45)	14.26 (7.71)	13.88 (7.48)	13.84 (6.97)	13.81 (6.52)
	Cost (rad)	6.22 (2.20)	4.55 (1.49)	4.56 (1.71)	4.67 (1.69)	4.63 (1.48)
Final	Time (s)	1316.67 (338.89)	1187.92 (270.23)	1195.90 (283.00)	1079.40 (249.38)	675.39 (122.59)
	Cost (rad)	5.43 (2.20)	2.60 (0.36)	2.65 (0.43)	2.65 (0.42)	2.62 (0.35)
# of Vertices		4360.17 (121.83)	4237.25 (84.36)	3394.96 (433.75)	2256.34 (235.21)	1572.20 (292.14)

A Monte-Carlo study was performed in order to compare the convergence rate and variance for all algorithms. All algorithms were run for 5,000 iterations and their results were averaged over 100 trials. The averaged results of all algorithms are summarized in Table 1. As seen below, the RRT$^\#$ algorithm and its three variants outperformed the RRT*

algorithm in all cases at finding the lower-cost path. Among all of them, the $\mathrm{RRT}_3^{\#}$ is the fastest algorithm and computes the best solution in a much shorter amount of time.

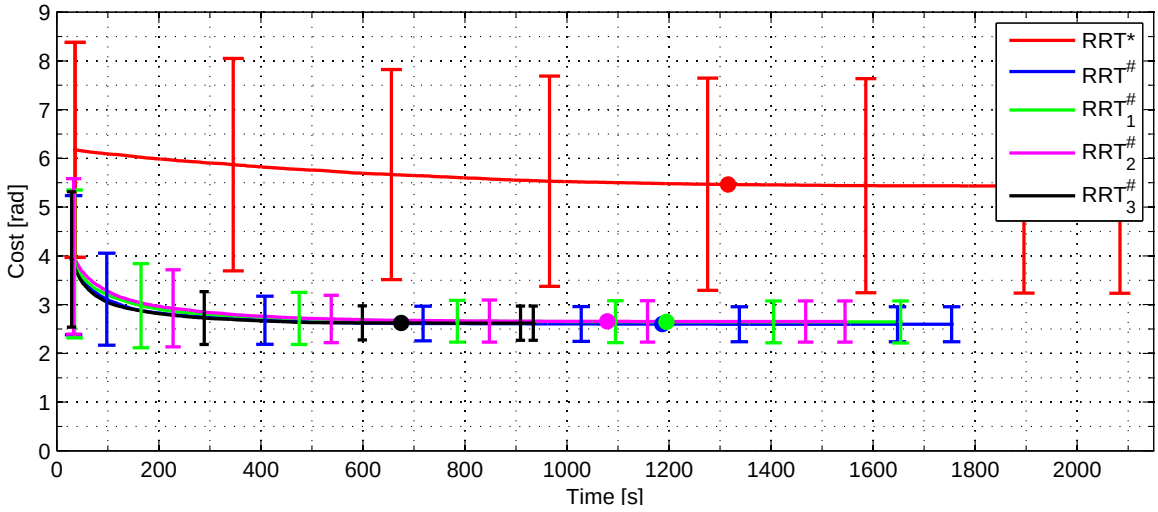


Figure 14: The convergence rate of the algorithms in 6D

The convergence rate, the variance in trials, as vertical bars, and the completion time, as filled circles, of all algorithms are shown in Figure 14. The RRT^* algorithm has the slowest convergence rate and the largest variance, and the $\mathrm{RRT}_3^{\#}$ algorithm has the fastest convergence rate and the smallest variance in the trials. Animations of the 6DOF and 12DOF cases shown in Figures 13 and 15 can be found in `https://goo.gl/FFOUON`.

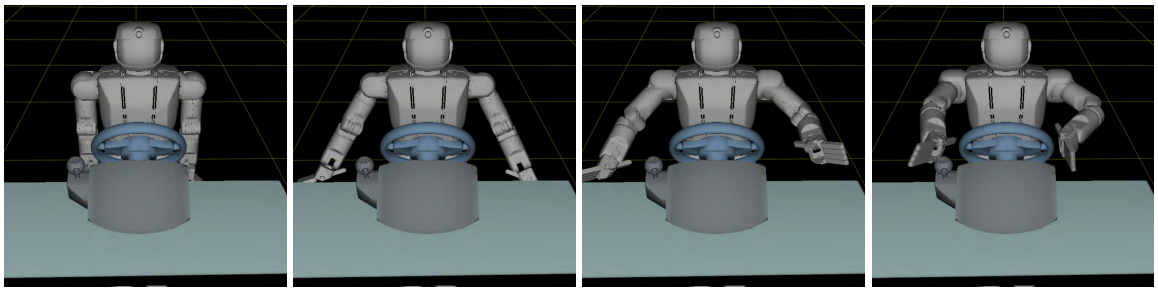


Figure 15: Initial and goal configurations of HUBO (12D)

4.9.2 Motion Planning for Dual-Arm Manipulation (12 DoFs)

In the final set of simulations, we tested all algorithms for a planning problem in a 12D search space. At the initial step, both arms of the HUBO lie on each side while it is

standing, and then the HUBO is commanded to move its right and left arms to pre-grasp poses for the stick and steering wheel, respectively, as shown in Figure 15.

Table 2: Results for Dual-Arm Planning Problem

Solution		RRT^*	$\mathrm{RRT}^\#$	$\mathrm{RRT}^\#_1$	$\mathrm{RRT}^\#_2$	$\mathrm{RRT}^\#_3$
First	Time (s)	111.47 (45.17)	108.73 (48.22)	107.42 (43.14)	103.14 (44.37)	100.07 (40.71)
	Cost (rad)	10.57 (2.12)	6.71 (1.70)	6.65 (1.76)	6.67 (1.62)	6.59 (1.45)
Final	Time (s)	6390.98 (714.09)	5455.71 (610.22)	4811.12 (305.10)	4170.12 (272.15)	2457.38 (221.71)
	Cost (rad)	9.72 (3.71)	4.96 (0.84)	4.93 (0.78)	4.94 (0.72)	4.92 (0.67)
# of Vertices		22470.14 (571.40)	21347.72 (351.74)	16147.81 (315.77)	8451.78 (307.97)	6187.18 (312.15)

A Monte-Carlo study is performed in order to compare the convergence rate and variance in the trials of all algorithms. All algorithms were run for 20,000 iterations, and their results were averaged over 25 trials. The averaged results for all algorithms are summarized in Table 2. The $\mathrm{RRT}^\#$ and its variant algorithms outperform the RRT^* algorithm at finding lower-cost paths at the final iteration on average. Among them, the $\mathrm{RRT}^\#_3$ is the fastest algorithm, and computes the best solution in a much shorter amount of time.

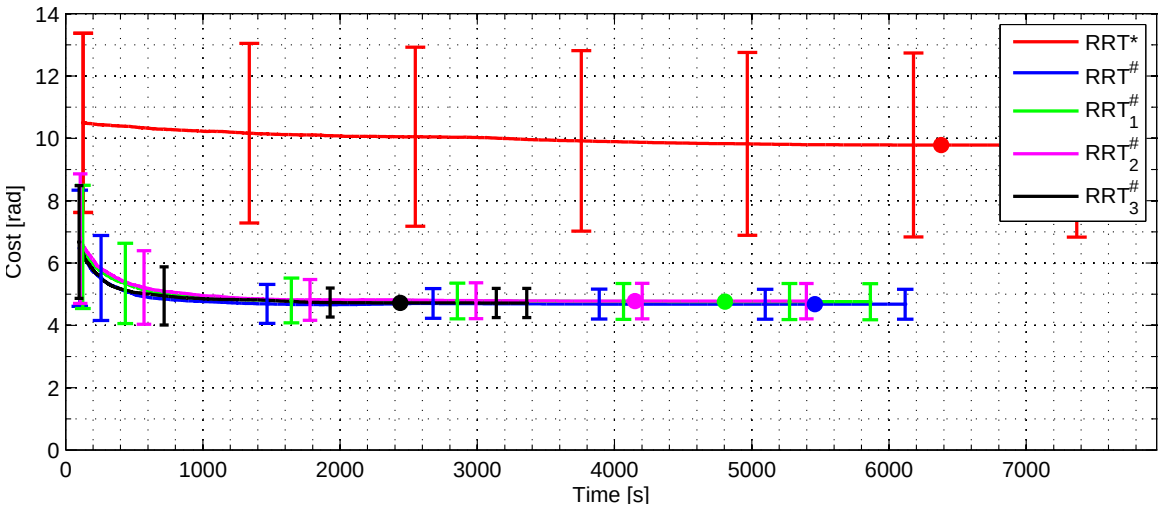


Figure 16: The convergence rate of the algorithms in 12D

The convergence rate, the variance in trials, as vertical bars, and the completion time, as filled circles, of all algorithms are shown in Figure 16. Among them, the RRT^* algorithm has the slowest convergence rate and the largest variance in the trials. The $\mathrm{RRT}^\#_3$ algorithm has the fastest convergence rate and the smallest variance.

4.10 Conclusion

In this chapter, a new incremental sampling-based algorithm, denoted by $\mathrm{RRT}^{\#}$ is presented, which offers asymptotically optimal solutions for motion planning problems. The $\mathrm{RRT}^{\#}$ algorithm relies heavily on the random geometric graph data structure and the RRG algorithm [98], which is known to have asymptotic optimality properties. A bottleneck of optimal sampling-based algorithms is the slow convergence to the optimal solution, although sampling-based algorithms are capable of finding a feasible solution, often almost in real-time. By incorporating stationarity information of all current vertices in the tree (essentially by comparing the current cost-to-come values of the vertices with the cost-to-come values via one of the neighboring vertices) we can have more informed estimates of the optimal values of the potential paths, thus speeding up convergence. Furthermore, once a feasible path has been found, vertex stationarity can be used to estimate the region where the optimal solution should be found. This results in an initial convergence rate that is better than the one achieved by the RRT^{*} algorithm.

Chapter V

MOTION PLANNING USING POLICY ITERATION METHODS

5.1 Overview

Recent progress in randomized motion planners has led to the development of a new class of sampling-based algorithms that provide asymptotic optimality guarantees, notably the RRT* and the PRM* algorithms among others. Careful analysis reveals that the so-called "rewiring" step in these algorithms can be interpreted as a local policy iteration (PI) step (i.e., a local policy evaluation step followed by a local policy improvement step) so that *asymptotically*, as the number of samples tend to infinity, both algorithms converge to the optimal path. Policy iteration, along with value iteration (VI) are common methods for solving dynamic programming (DP) problems. Based on this observation, the RRT$^\#$ algorithm was proposed, which performs, during each iteration, Bellman updates (aka "back-ups") on those vertices of the graph that have the potential of being part of the optimal path (i.e., the "promising" vertices). The RRT$^\#$ algorithm thus utilizes dynamic programming ideas and implements them incrementally on randomly generated graphs to obtain high quality solutions. In this work, and based on this key insight, we explore a different class of dynamic programming algorithms for solving shortest-path problems on random graphs generated by iterative sampling methods. This class of algorithms utilize policy iteration instead of value iteration and are better suited for massive parallelization.

In this chapter, we depart from the previous VI-based algorithms and we propose, instead, a novel class of algorithms based on policy-iteration (PI). Some preliminary results were presented in [13]. Policy iteration is an alternative to value iteration for solving dynamic programming problems and fits naturally into our framework, in the sense that a policy in a graph search amounts to nothing more but an assignment of a (unique) parent to

each vertex. Use of policy iteration has the following benefits: first, no queue is needed to keep track of the cost of each vertex. A subset of vertices is selected for Bellman updates and policy improvement on these vertices can be done in parallel at each iteration. Second, for a given graph, determination of the *optimal* policy is obtained after a *finite* number of iterations since the policy space is finite [23]. The determination of the optimal value for each vertex, on the other hand, requires an infinite number of iterations. More crucially, in order to find the optimal policy only the correct ordering of the vertices is needed, not their exact value. This can be utilized to develop approximation algorithms that speed up convergence. Third, although policy iteration methods are often slower than value iteration methods, they tend to be better amenable for parallelization and are faster if the structure of the problem is taken into consideration during implementation.

5.2 Problem Formulation

Let $\mathcal{X}$ denote the configuration (search) space, which is assumed to be an open subset of $\mathbb{R}^d$, where $d \in \mathbb{N}$ with $d \geq 2$. The *obstacle region* and the *goal region* are denoted by $\mathcal{X}_{\text{obs}}$ and $\mathcal{X}_{\text{goal}}$, respectively, both assumed to be closed sets. The obstacle-free space is defined by $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}$. Elements of $\mathcal{X}$ are the states (or configurations) of the system. Let the *initial configuration* of the robot be denoted by $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. The (open) neighborhood of a state $x \in \mathcal{X}$ is the open ball of radius $r > 0$ centered at x, that is, $B_r(x) = \{x' \in \mathcal{X} : \|x - x'\| < r\}$.

We will approximate $\mathcal{X}_{\text{free}}$ with an increasingly dense sequence of discrete subsets of $\mathcal{X}_{\text{free}}$. That is, $\mathcal{X}_{\text{free}}$ will be approximated by a finite set of configuration points selected randomly from $\mathcal{X}_{\text{free}}$. Each such discrete approximation of $\mathcal{X}_{\text{free}}$ will be encoded in a graph $\mathcal{G} = (V, E)$ with V being the set of vertices (the elements of the discrete approximation of $\mathcal{X}_{\text{free}}$) and with edge set $E \subseteq V \times V$ encoding allowable transitions between elements of V. Hence, $\mathcal{G}$ is a directed graph. Transitions between two vertices x and x' in V are enabled by a control action $u \in U(x)$ such that x' is the successor vertex of x in $\mathcal{G}$ under the action

u. Let $U = \cup_{x \in V} U(x)$. We use the mapping $f : V \times U \to V$ given by

$$x' = f(x, u), \qquad u \in U(x), \tag{22}$$

to formalize the transition from x to x' under the control action u. In this case, we say that x' is the successor of x and that x is the predecessor of x'. The set of predecessors of $x \in V$ will be denoted by $\texttt{pred}(\mathcal{G}, x)$, and the set of successors of x will be denoted by $\texttt{succ}(\mathcal{G}, x)$. Also, we let $\overline{\texttt{pred}}(\mathcal{G}, x) = \texttt{pred}(\mathcal{G}, x) \cup \{x\}$. Note that, using the previous definitions, the set of admissible control actions at x may be equivalently defined as

$$U(x) = \{u : x' = f(x, u),\ x' \in \texttt{succ}(\mathcal{G}, x)\}. \tag{23}$$

Thus, the control set $U(x)$ defines unambiguously the set $\texttt{succ}(\mathcal{G}, x)$ of the successors of x, in the sense that there is one-to-one correspondence between control actions $u \in U(x)$ and elements of $\texttt{succ}(\mathcal{G}, x)$ via (22). Equivalently, once the directed graph $\mathcal{G}$ is given, for each edge $(x, x') \in E$ corresponds a control $u \in U(x)$ enabling this transition. It should be remarked that the latter statement, when dealing with dynamical systems (such as robots, etc) amounts to a controllability condition. Controllability is always satisfied for fully actuated systems, but may not be satisfied for underactuated systems (such as for many case of kinodynamic planning with differential constraints). For sampling-based methods such as RRT^* this controllability condition is equivalent to the existence of a steering function that drives the system between any two given states.

Once we have abstracted $\mathcal{X}_{\text{free}}$ using the graph $\mathcal{G}$, the motion planning problem becomes one of a shortest path problem on the graph $\mathcal{G}$. To this end, we define the path σ of length N in $\mathcal{G}$ to be a sequence of vertices $\sigma = (x_0, x_1, \ldots, x_N)$ such that $x_{k+1} \in \texttt{succ}(\mathcal{G}, x_k)$ for all $k = 0, 1, \ldots, N-1$. When we want to specify explicitly the first node of the path we will use the first node as an argument, i.e., we will write $\sigma(x_0)$. The kth element of σ will be denoted by σ_k. That is, if $\sigma(x_0) = (x_0, x_1, \ldots, x_N)$ then $\sigma_k(x_0) = x_k$ for all $k = 0, 1, \ldots, N$. A path is rooted at x_{init} if $x_0 = x_{\text{init}}$. A path rooted at x_{init} terminates at a given goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$ if $x_N \in \mathcal{X}_{\text{goal}}$.

To each edge (x, x') encoding an allowable transition, we associate a cost $\mathsf{c}(x, x')$. Given a path $\sigma(x_0)$, the cumulative cost along this path is then

$$\sum_{k=0}^{N-1} \mathsf{c}(x_k, x_{k+1}). \tag{24}$$

Given a point $x \in \mathcal{X}$, a mapping $\mu : x \mapsto u \in U(x)$ that assigns a control action to be executed at each point x is called a *policy*. Under some assumptions on the connectivity of the graph $\mathcal{G}$ and the cost of the directed edges, one can use DP algorithms and the corresponding Bellman equation in order to compute optimal policies. Note that a stationary policy μ for this problem defines a graph whose edges are $(x, f(x, \mu(x))) \in E$ for all $x \in V$. The policy μ is proper if and only if this graph is acyclic, i.e., the graph has no cycles. Thus, there exists a proper policy μ if and only if each node is connected to the $\mathcal{X}_{\text{goal}}$ with a directed path. Furthermore, an improper policy has finite cost, starting from every initial state, if and only if all the cycles of the corresponding graph have nonnegative cost [23]. Convergence of the DP algorithms is proven if the graph is connected and the costs of all its cycles are positive [24]. In our case, the graph computed by the RRG algorithm is a connected graph by construction, and all edge cost values are positive, which implies that the costs of all its cycles are positive. Therefore, convergence is guaranteed and the resulting optimal policy is proper.

In terms of DP notation, our system has the following equation

$$x' = f(x, u) \tag{25}$$

where the cost function is defined as

$$g(x, u) = \mathsf{c}(x, f(x, u)). \tag{26}$$

It follows that the corresponding Bellman's equation takes the form

$$J^*(x) = \min_{u \in U(x)} \left\{ \mathsf{c}(x, f(x, u)) + J^*(f(x, u)) \right\}. \tag{27}$$

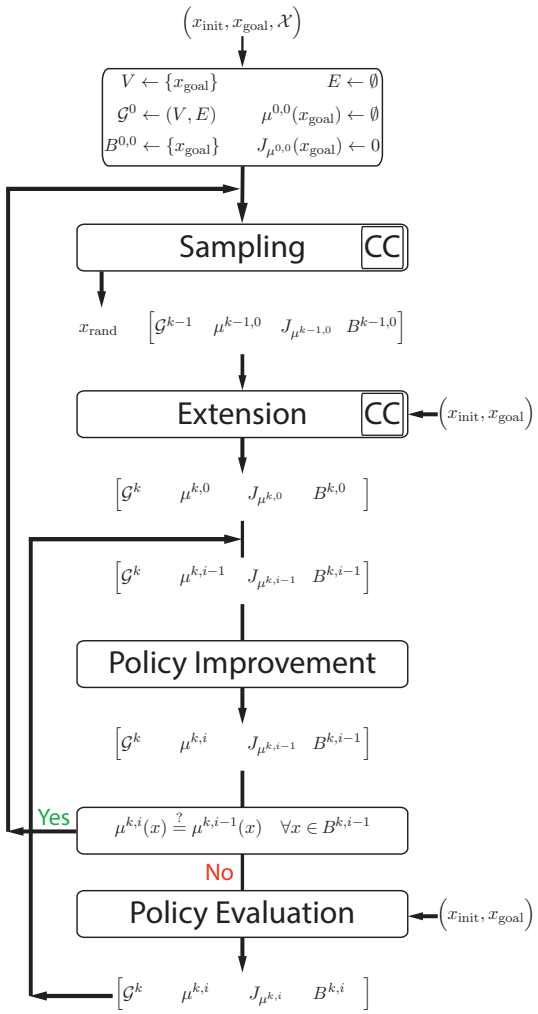


Figure 17: Overview of the PI based $\mathrm{RRT}^{\#}$ Algorithm

5.3 DP Algorithms for Sampling-based Planners

The sampling-based motion planner which utilizes VI, i.e., $\mathrm{RRT}^{\#}$, was presented in [9]. The $\mathrm{RRT}^{\#}$ algorithm implements the Gauss-Seidel version of the VI algorithm and provides a sequential implementation. In this chapter, we follow up on the same idea and propose a sampling-based algorithm which utilizes PI algorithm as shown in Figure 17.

The body of PI-based $\mathrm{RRT}^{\#}$ algorithm is given in Algorithm 6. The algorithm includes a new vertex and a couple new edges into the existing graph at each iteration. If this new information has a potential to improve the existing policy, then, a slightly modified PI algorithm is called subsequently in the `Replan` procedure. For the sake of efficiency, unlike the

standard PI algorithm, policy improvement is performed only for a subset vertices which have the potential to be part of the optimal solution.

Algorithm 6: Body of the $\mathrm{RRT}^{\#}$ Algorithm (Sync. PI)

```
1  RRT#(x_init, x_goal, X)
2      V ← {x_goal}; S ← V; E ← ∅;
3      G ← (V, E);
4      for k = 1 to N do
5          x_rand = Sample(k);
6          (G, S') ← Extend(G, S, x_init, x_rand);
7          if |S'| > |S| then
8              S ← Replan(S', x_init, x_goal);
9      (V, E) ← G; E' ← ∅;
10     foreach x ∈ V do
11         E' ← E' ∪ {(x, μ(x))};
12     return T = (V, E');
```

The `Extend` procedure is given in Algorithm 7. If a new vertex is decided for inclusion, its control is initialized by performing policy improvement in Lines 6-14. Then, it is checked in Line 15 if the new vertex has a potential to improve the existing policy. If so, it is included to the set of vertices which are selected to perform policy improvement.

The `Replan` procedure which implements the PI algorithm is shown in Algorithm 8. The policy improvement step is performed in Lines 2-9 until the existing becomes almost stationary. Note that the for-loop in the `Replan` procedure can run parallel.

The policy evaluation step is implemented in Algorithm 9. Algorithm 9 solves a system of linear equations by exploiting the underlying structure. Simply, the existing policy forms a tree in the current graph and the solution of the system of linear equations corresponds to the cost of each path connecting vertices to the goal region via edges of the tree.

Algorithm 7: `Extend` Procedure for RRT$^{\#}$ Algorithm (Sync. PI)

Extend($\mathcal{G}$, S, x_{init}, x_{rand})
- $(V, E) \leftarrow \mathcal{G}$; $E' \leftarrow \emptyset$;
- $x_{\text{nearest}} = \texttt{Nearest}(\mathcal{G}, x_{\text{rand}})$;
- $x_{\text{new}} = \texttt{Steer}(x_{\text{rand}}, x_{\text{nearest}})$;
- **if** $\texttt{ObstacleFree}(x_{\text{new}}, x_{\text{nearest}})$ **then**
  - $\texttt{J}(x_{\text{new}}) = \texttt{c}(x_{\text{new}}, x_{\text{nearest}}) + \texttt{J}(x_{\text{nearest}})$;
  - $\mu(x_{\text{new}}) = x_{\text{nearest}}$;
  - $\mathcal{X}_{\text{near}} \leftarrow \texttt{Near}(\mathcal{G}, x_{\text{new}}, |V|)$;
  - **foreach** $x_{\text{near}} \in \mathcal{X}_{\text{near}}$ **do**
    - **if** $\texttt{ObstacleFree}(x_{\text{new}}, x_{\text{near}})$ **then**
      - **if** $\texttt{J}(x_{\text{new}}) > \texttt{c}(x_{\text{new}}, x_{\text{near}}) + \texttt{J}(x_{\text{near}})$ **then**
        - $\texttt{J}(x_{\text{new}}) = \texttt{c}(x_{\text{new}}, x_{\text{near}}) + \texttt{J}(x_{\text{near}})$;
        - $\mu(x_{\text{new}}) = x_{\text{near}}$;
      - $E' \leftarrow E' \cup \{(x_{\text{new}}, x_{\text{near}}), (x_{\text{near}}, x_{\text{new}})\}$;
  - **if** $\texttt{h}(x_{\text{init}}, \mu(x_{\text{new}})) + \texttt{J}(\mu(x_{\text{new}})) \leq \texttt{J}(x_{\text{init}})$ **then**
    - $S \leftarrow S \cup \{x_{\text{new}}\}$;
  - $V \leftarrow V \cup \{x_{\text{new}}\}$;
  - $E \leftarrow E \cup E'$;
- $\mathcal{G} \leftarrow (V, E)$;
- **return** $(\mathcal{G}, S)$;

Algorithm 8: `Replan` Procedure (Sync. PI)

Replan($\mathcal{G}$, S, x_{init}, x_{goal})
- **Loop**
  - **foreach** $x \in S$ **do**
    - $J' = \texttt{J}(x)$;
    - **foreach** $v \in \texttt{succ}(\mathcal{G}, x)$ **do**
      - **if** $J' > \texttt{c}(x, v) + \texttt{J}(v)$ **then**
        - $J' = \texttt{c}(x, v) + \texttt{J}(v)$;
        - $\mu(x) = v$;
    - $\Delta J(x) = \texttt{J}(x) - J'$;
  - **if** $\max_{x \in S} \Delta J(x) \leq \epsilon$ **then**
    - **return** S;
  - $S \leftarrow \texttt{Evaluate}(\mathcal{G}, x_{\text{init}}, x_{\text{goal}})$;

Algorithm 9: `Evaluate` Procedure (PI)

```
1  Evaluate(G, x_init, x_goal)
2      (V, E) ← G;
3      if x_init ∈ V then
4          x = x_init;
5          while x ≠ x_goal do
6              q.push_front(x);
7              x = μ(x);
8          J(x_goal) = 0;
9          while q.empty() do
10             x = q.pop_front();
11             J(x) = c(x, μ(x)) + J(μ(x));
12     else
13         J(x_init) = ∞;
14     S ← {x_goal};
15     q.push_back(x_goal);
16     while q.nonempty() do
17         x = q.pop_front();
18         if h(x_init, x) + J(x) ≤ J(x_init) then
19             foreach v ∈ children(x) do
20                 J(v) = c(v, x) + J(x);
21                 S ← S ∪ {v};
22                 q.push_back(v);
23     return S;
```

5.4 Theoretical Analysis

Let $\mathcal{G}^k = (V^k, E^k)$ be the graph at the end of the kth iteration of the PI-RRT$^\#$ algorithm. Given a vertex $x \in V^k$, let the control set $U^k(x)$ and the successor set $\texttt{succ}(\mathcal{G}^k, x)$ be divided into two disjoint sets $\left\{U^{k,*}(x), U^{k,\prime}(x)\right\}$ and $\left\{S^{k,*}(x), S^{k,\prime}(x)\right\}$, respectively, as follows: $U^k(x) = U^{k,*}(x) \cup U^{k,\prime}(x)$ where $U^{k,*}(x) = \arg\min_{u \in U^k(x)} H(x, u, J^{k,*})$ and $U^{k,\prime}(x) = U^k(x) \setminus U^{k,*}(x)$ and $\texttt{succ}(\mathcal{G}^k, x) = S^{k,*}(x) \cup S^{k,\prime}(x)$ where $S^{k,*}(x) = \{x^* \in V^k : \exists u^* \in U^{k,*}(x) \text{ s.t. } x^* = f(x, u^*)\}$ and $S^{k,\prime}(x) = \texttt{succ}(\mathcal{G}^k, x) \setminus S^{k,*}(x)$. Furthermore, let $\mathcal{M}^k$ denote the set of all policies at the end of the kth iteration of the PI-RRT$^\#$ algorithm, that is, $\mathcal{M}^k = \{\mu : \mu(x) \in U^k(x), \forall x \in V^k\}$. Given a policy $\mu \in \mathcal{M}^k$ and an initial state x, $\sigma^\mu(x)$ denotes the path resulting from executing the policy μ starting at x. That is, $\sigma^\mu(x) = (x_0, x_1, \ldots, x_N)$ such that $\sigma^\mu_0(x) = x$ and $\sigma^\mu_N(x) \in \mathcal{X}_{\text{goal}}$ for $N > 0$, where $\sigma^\mu_j(x)$ is the jth element of $\sigma^\mu(x)$. By definition, note that $x_{j+1} = f(x_j, \mu(x_j))$ for $j = 0, 1, \ldots, N-1$. Finally, let $\Sigma^k(x)$ be the set of all paths rooted at x and let $\Sigma^{k,*}(x)$ denote the set of all paths of lowest-cost rooted at x that reach the goal region at the kth iteration of the algorithm, that is, $\Sigma^{k,*}(x) = \{\sigma \in \Sigma^k(x) : \sigma = \sigma^\mu(x)$ such that $\exists \mu \in \mathcal{M}^k,\ J_\mu(x) = J^{k,*}(x)\}$. Note that the set $\Sigma^{k,*}(x)$ may contain more than one paths. Let $N^k(x)$ denote the shortest path length in $\Sigma^{k,*}(x)$, that is, $N^k(x) = \min_{\sigma^\mu(x) \in \Sigma^{k,*}(x)} \texttt{len}(\sigma^\mu(x))$.

Let us define the following sets for a given policy $\mu^{k,i} \in \mathcal{M}^k$ and its corresponding value function $J_{\mu^{k,i}}$ at the end of ith policy improvement step and at the kth iteration of the PI-RRT$^\#$ algorithm:

a) The set of vertices in $\mathcal{G}^k$ whose optimal cost function value is less than that of x_{init}, i.e., the set of promising vertices, i.e., $V^k_{\text{prom}} = \{x \in V^k : J^{k,*}(x) < J^{k,*}(x_{\text{init}})\}$

b) The set of promising vertices in $\mathcal{G}^k$ whose optimal cost function value is achieved by executing the policy $\mu^{k,i}$, i.e., $O^{k,i} = \{x \in V^k : J_{\mu^{k,i}}(x) = J^{k,*}(x) < J^{k,*}(x_{\text{init}})\}$

c) The set of vertices that can be connected to the goal region with an optimal path of length less than or equal to ℓ, i.e., $L^{k,\ell} = \{x \in V^k : N^k(x) \leq \ell\}$

d) The set of promising vertices that are connected to the goal region via optimal paths whose length is less than or equal to ℓ, i.e., $P^{k,\ell} = L^{k,\ell} \cap V^k_{\text{prom}} = \{x \in V^k : N^k(x) \leq \ell \text{ and} J^{k,*}(x) < J^{k,*}(x_{\text{init}})\}$. Note that the set of promising vertices that can be connected to the goal region via optimal paths whose length is exactly $\ell + 1$ is given by $\partial P^{k,\ell} = P^{k,\ell+1} \setminus P^{k,\ell}$.

e) The set of vertices that are selected for Bellman update during the beginning of the ith policy improvement, i.e., $B^{k,i} = \{\overline{\texttt{pred}}(\mathcal{G}^k, x) : x \in V^k,\ J_{\mu^{k,i}}(x) < J_{\mu^{k,i}}(x_{\text{init}})\}$

Lemma 4

The sequence $O^{k,i}$ generated by the policy iteration step of the PI-$\text{RRT}^{\#}$ *algorithm is non-decreasing, that is, $O^{k,i} \subseteq O^{k,i+1}$ for all $i = 0, 1, \ldots$.*

Proof. This lemma is proven by showing that if the optimal cost function and policy of a point are computed at some iteration, then these solution will not be updated as the number of iterations goes.

First, note that $O^{k,0} = V^k \cap \mathcal{X}_{\text{goal}} \neq \varnothing$. Let now $i > 0$ and assume that $x \in O^{k,i}$. Then, by definition, we have that $J_{\mu^{k,i}} = T_{\mu^{k,i}} J_{\mu^{k,i}}$ where $\mu^{k,i}$ is the policy computed at the end of ith policy improvement step at the kth iteration of the PI-$\text{RRT}^{\#}$ algorithm. The previous expression implies that $(T_{\mu^{k,i}} J_{\mu^{k,i}})(x) = J^{k,*}(x)$, and hence $\mu^{k,i}(x) \subseteq U^{k,*}(x)$. Similarly, the cost function $J_{\mu^{k,i}}$ satisfies $J_{\mu^{k,i}}(x) = J^{k,*}(x) < J^{k,*}(x_{\text{init}}) \leq J_{\mu^{k,i}}(x_{\text{init}})$ which yields $J_{\mu^{k,i}}(x) < J_{\mu^{k,i}}(x_{\text{init}})$. It follows that the vertex x and its predecessors will be selected for Bellman update during the next policy improvement, that is, $\overline{\texttt{pred}}(\mathcal{G}^k, x) \in B^{k,i}$.

After policy improvement, the updated policy and the corresponding cost function are given by $(T_{\mu^{k,i+1}} J_{\mu^{k,i}})(x) = (T J_{\mu^{k,i}})(x) = J^{k,*}(x)$ which implies $(T_{\mu^{k,i+1}} J_{\mu^{k,i}})(x) = J^{k,*}(x)$ and hence $\mu^{k,i+1}(x) \subseteq U^{k,*}(x)$. Similarly, $J^{k,*}(x) \leq J_{\mu^{k,i+1}}(x) \leq J_{\mu^{k,i}}(x) = J^{k,*}(x)$ and hence $J_{\mu^{k,i+1}}(x) = J^{k,*}(x) < J^{k,*}(x_{\text{init}})$. It follows that $x \in O^{k,i+1}$. □

Lemma 5

The sequence $L^{k,\ell}$ is non-decreasing, that is, $L^{k,\ell} \subseteq L^{k,\ell+1}$ for $\ell = 0, 1, \ldots$. Furthermore,

for all $x \in \partial L^{k,\ell} = L^{k,\ell+1} \setminus L^{k,\ell}$*, there exists* $x^* \in L^{k,\ell}$ *such that* $x^* \in S^{k,*}(x)$.

Proof. This lemma is proven by using an induction argument. Once the base case is shown to be held, the induction case is proven by leveraging the principle of optimality for shortest path problems, that is, any subarc of an optimal path is also optimal.

For $\ell = 0$ we have that $L^{k,0} = V^k \cap \mathcal{X}_{\text{free}} \neq \varnothing$. Let now $\ell > 0$ and assume that $x \in L^{k,\ell}$. Then, by definition, there exists a policy $\mu \in \mathcal{M}^k$ such that the vertex x achieves its optimal cost function value, i.e., $J_\mu(x) = J^{k,*}(x)$, and the optimal path connecting x to the goal region has length less than or equal to ℓ, that is, $\texttt{len}(\sigma^\mu(x)) \leq \ell < \ell + 1$, which implies, trivially, that $x \in L^{k,\ell+1}$.

To show the second part of the statement, first notice that, by definition, the vertices in the set $\partial L^{k,\ell}$ are the ones which can be connected to the goal region via an optimal path of length exactly $(\ell + 1)$. Let us now assume that $x \in \partial L^{k,\ell}$ and let $\sigma^\mu(x) \in \Sigma^{k,*}(x)$ be the optimal path of length $(\ell + 1)$ between x and the goal region. Let $\sigma_1^\mu(x) = x^*$ and $\sigma^\mu(x^*)$ be the sub-arc rooted at x^* resulting from applying μ. By construction of the path $\sigma^\mu(x)$, we have that $x \in \texttt{pred}(\mathcal{G}^k, x^*)$. Also, since $\sigma^\mu(x)$ is the optimal path rooted at x, the control action applied at vertex x needs to be optimal , i.e., $\mu(x) \in U^{k,*}(x)$ and $\sigma^\mu(x^*)$ is the optimal path which connects x^* to goal region due to the *principle of optimality*, $\sigma^\mu(x^*) \in \Sigma^{k,*}(x^*)$. This implies that $x^* \in S^{k,*}(x)$ and $x^* \in L^{k,\ell}$ since the length of the path σ' is ℓ, i.e., $\texttt{len}(\sigma^\mu(x^*)) = \ell$. □

Corollary 1

The sequence $P^{k,\ell}$ *is non-decreasing, that is,* $P^{k,\ell} \subseteq P^{k,\ell+1}$ *for* $\ell = 0, 1, \ldots$*. Furthermore, for all* $x \in \partial P^{k,\ell} = P^{k,\ell+1} \setminus P^{k,\ell}$*, there exists* $x^* \in P^{k,\ell}$ *such that* $x^* \in S^{k,*}(x)$.

Lemma 6

For a given vertex $x \in B^{k,i}$*, if* $J_{\mu^{k,i}}(x^*) = J^{k,*}(x^*)$ *and* $x^* \in S^{k,*}(x)$ *hold, then we have* $\mu^{k,i+1}(x) \subseteq U^{k,*}(x)$ *and* $J_{\mu^{k,i+1}}(x) = J^{k,*}(x)$ *at the end of* $(i+1)$*th policy improvement step at the* k*th iteration of the* PI-RRT$^\#$ *algorithm.*

Proof. This lemma is proven by analyzing the cost function value and policy for a point during the Bellman update when its successors that are along the optimal path have already achieved their optimal cost function value and policy.

Let us check how the function $H(x, \cdot, J_{\mu^{k,i}})$ changes with respect to controls from $U^{k,*}(x)$ and $U^{k,\prime}(x)$. At the beginning of $(i+1)$th policy improvement step, the new policy $\mu^{k,i+1}$ is computed as follows. For all $x \in B^{k,i}$

$$\begin{aligned}\mu^{k,i+1}(x) &\in \arg\min_{u \in U^k(x)} H(x, u, J_{\mu^{k,i}}) \\ &= \arg\min_{u \in U^k(x)} \Big\{ g(x,u) + J_{\mu^{k,i}}(f(x,u)) \Big\}.\end{aligned}$$

For $u^* \in U^{k,*}(x)$ and $u' \in U^{k,\prime}(x)$, we have the following:

$$\begin{aligned}H(x, u^*, J_{\mu^{k,i}}) &= g(x,u^*) + J_{\mu^{k,i}}(f(x,u^*)) \\ &= g(x,u^*) + J_{\mu^{k,i}}(x^*) = g(x,u^*) + J^{k,*}(x^*) \\ &< g(x,u') + J^{k,*}(x') \\ &= g(x,u') + J^{k,*}(f(x,u')) \\ &\leq g(x,u') + J_{\mu^{k,i}}(f(x,u')) \\ &= H(x, u', J_{\mu^{k,i}})\end{aligned}$$

which implies that

$$H(x, u^*, J_{\mu^{k,i}}) < H(x, u', J_{\mu^{k,i}}) \quad \forall u^* \in U^*(x), u' \in U'(x).$$

Hence, it follows that

$$\mu^{k,i+1}(x) \in \operatorname*{argmin}_{u \in U^k(x)} H(x, u, J_{\mu^{k,i}}) = \operatorname*{argmin}_{u \in U^{k,*}(x)} H(x, u, J_{\mu^{k,i}}) \quad \Rightarrow \quad \mu^{k,i+1}(x) \subseteq U^{k,*}(x).$$

Let us check the cost function $J_{\mu^{k,i+1}}$ for $x^* \in S^{k,*}$, which is subsequently computed during the policy evaluation step for the new policy $\mu^{k,i+1}$.

$$J^{k,*}(x^*) \leq J_{\mu^{k,i+1}}(x^*) \leq J_{\mu^{k,i}}(x^*) = J^{k,*}(x^*) \quad \Rightarrow \quad J_{\mu^{k,i+1}}(x^*) = J^{k,*}(x^*) \quad \forall x^* \in S^{k,*}(x)$$

We can now write $J_{\mu^{k,i+1}}(x)$ as follows:

$$J_{\mu^{k,i+1}}(x) = g(x,u^*)+J_{\mu^{k,i+1}}(f(x,u^*)) = g(x,u^*)+J_{\mu^{k,i+1}}(x^*) = g(x,u^*)+J^{k,*}(x^*) = J^{k,*}(x)$$

which implies that $J_{\mu^{k,i+1}}(x) = J^{k,*}(x)$.

□

Lemma 7

For a given policy $\mu^{k,i}$ and its corresponding cost function $J_{\mu^{k,i}}$, if $P^{k,\ell} \subseteq O^{k,i}$ holds, then we have that $\partial P^{k,\ell} \subseteq B^{k,i}$, which implies $P^{k,\ell+1} \subseteq B^{k,i}$, before the beginning $(i+1)$th policy improvement step. Then, $P^{k,\ell+1} \subseteq O^{k,i+1}$ holds after $(i+1)$th policy improvement step.

Proof. This lemma is proven by using the fact that tails of optimal paths are extended at least by one segment after each policy improvement step.

We have $P^{k,\ell} \subseteq O^{k,i}$ by our assumption and know that the sequence $P^{k,\ell}$ is non-decreasing due to Lemma 5, i.e., $P^{k,\ell} \subseteq P^{k,\ell+1}$. Since the sequence $O^{k,i}$ is also non-decreasing due to Lemma 4, i.e., $O^{k,i} \subseteq O^{k,i+1}$, this implies that $P^{k,\ell} \subseteq O^{k,i+1}$. Therefore, we only need to show if the boundary set $\partial P^{k,\ell}$ is a subset of $O^{k,i+1}$ by end of $(i+1)$th policy improvement.

As shown in Lemma 5, we have that for all $x \in \partial P^{k,\ell}$, there exists $x' \in P^{k,\ell}$ such that $x' \in S^{k,*}(x)$, which also implies $x \in \texttt{pred}(\mathcal{G}^k, x')$. Since $P^{k,\ell} \subseteq O^{k,i}$, we have the following:

$$J_{\mu^{k,i}}(x') = J^{k,*}(x') < J^{k,*}(x_{\text{init}}) \leq J_{\mu^{k,i}}(x_{\text{init}}) \quad \Rightarrow \quad J_{\mu^{k,i}}(x') < J_{\mu^{k,i}}(x_{\text{init}})$$

Therefore, $\overline{\texttt{pred}}(\mathcal{G}^k, x') \in B^{k,i}$ which implies that all vertices of $\partial P^{k,\ell}$ are selected for Bellman update before $(i+1)$th policy improvement step, i.e., $\partial P^{k,\ell} \subseteq B^{k,i}$ and $P^{k,\ell+1} \subseteq B^{k,i}$.

As shown in Lemma 6, since $J_{\mu^{k,i}}(x') = J^{k,*}(x')$, $x' \in S^{k,*}(x)$ holds, all vertices of

$\partial P^{k,j}$ achieve their optimal policy and cost function value after end of the policy improvement, i.e., $\mu^{k,i+1}(x) \subseteq U^{k,*}(x)$ and $J_{\mu^{k,i+1}}(x) = J^{k,*}(x)$. This implies that $\partial P^{k,\ell} \subseteq O^{k,i+1}$ and $P^{k,\ell+1} \subseteq O^{k,i+1}$ which completes the proof. □

Lemma 8

All vertices whose optimal cost function value is less than that of x_{init} and are along optimal paths whose length is less than or equal to i achieve their optimal cost function value at the end of ith policy improvement step, that is, $P^{k,i} \subseteq O^{k,i}$ holds for $i = 0, 1, \ldots$ when using policy $\mu^{k,i}$ and its corresponding cost function $J_{\mu^{k,i}}$.

Proof. This lemma is proven by using an inductive argument. Once the base case is shown to be held, the induction case is proven by showing that tails of the optimal paths that connect promising vertices to the goal region are extended at least by one segment after a policy improvement step.

Our claim $P^{k,i} \subseteq O^{k,i}$ will be proven by using an induction argument.

Basis $i = 0$**:** First, note that $V^k_{\text{goal}} \neq \varnothing$. Let now assume $x \in V^k_{\text{goal}}$, then, we have that $N^k(x) = 0$ and $J^{k,*}(x) = 0 < J^{k,*}(x_{\text{init}})$. Therefore, $P^{k,0} = V^k_{\text{goal}}$ holds. Also, for all $x \in P^{k,0}$, we have that $J_{\mu^{k,0}}(x) = J^{k,*}(x) = 0 < J^{k,*}(x_{\text{init}})$ which implies $P^{k,0} \subseteq O^{k,0}$.

Basis $i = 1$**:** The set of vertices along optimal paths whose length is less than or equal to one is a subset of goal vertices and their predecessors, that is, $P^{k,1} = P^{k,0} \cup \{x \in V^k : \exists x' \in V^k \cap \mathcal{X}_{\text{goal}} \text{ s.t. } x \in \texttt{pred}(\mathcal{G}^k, x'),\ \texttt{c}(x, x') < J^{k,*}(x_{\text{init}})\}$. For all $x' \in V^k_{\text{goal}}$, we have that $J_{\mu^{k,0}}(x') = 0 < J_{\mu^{k,0}}(x_{\text{init}})$. Therefore all goal vertices and their predecessors are selected for Bellman update at the beginning of the first policy improvement step, i.e., $B^{k,0} = \{\overline{\texttt{pred}}(\mathcal{G}^k, x') : x' \in V^k_{\text{goal}}\}$ which implies $P^{k,1} \subseteq B^{k,0}$. All vertices in $P^{k,1}$ will achieve its optimal cost function value at the end of the first policy improvement step, that is, $J_{\mu^{k,1}}(x) = J^{k,*}(x) = \texttt{c}(x, x')$ where $x \in P^{k,1}$, $x' \in S^{k,*}(x)$ and $x' \in V_{\text{goal}}$, which implies $P^{k,1} \subseteq O^{k,1}$.

Inductive step: Let us assume $P^{k,i} \subseteq O^{k,i}$ holds and we need to show this assumption implies $P^{k,i+1} \subseteq O^{k,i+1}$ at the end of $(i+1)$th policy improvement step. Proof of such property is a direct result of Lemma 7 when a special case $\ell = i$ is considered.

□

Theorem 6 (Optimality of Each Iteration)

The optimal action and the cost function value for the initial vertex is achieved when the `Replan` *procedure of the* PI-RRT$^{\#}$ *algorithm terminates after a finite number of policy improvement steps.*

Proof. This theorem is proven by using the fact that termination of the `Replan` procedure is guaranteed since the policy space is finite. The optimality result can easily be derived for the cases when the `Replan` procedure terminates by performing a number of policy improvement steps that happens to be more than the number of segments of the optimal path connecting x_{init} to the goal region. For earlier termination, it is shown that the optimal cost function value and policy for x_{init} had already been computed.

We will investigate the case in which the algorithm terminates before performing $N^k(x_{\text{init}})$ policy improvement steps. Otherwise, the optimality property is a straightforward result due to Lemma 8 if the `Replan` procedure terminates after performing more than or equal to $N^k(x_{\text{init}})$ policy improvement steps.

Let us assume, ad absurdum, that the `Replan` procedure terminates at the end of ith policy improvement step at kth iteration with a suboptimal cost function value for the initial vertex, that is, $J_{\mu^{k,i-1}}(x_{\text{init}}) > J^{k,*}(x_{\text{init}})$. Since the termination condition holds, we observe no policy update for all vertices in $B^{k,i-1}$ after performing a policy improvement attempt. That is, for all $x \in B^{k,i-1}$, we have $\mu^{k,i}(x) = \mu^{k,i-1}(x)$.

We know that $P^{k,i-1} \subseteq O^{k,i-1}$ holds for $i = 1, 2, \ldots$ as shown in Lemma 8. This implies that $P^{k,i} \in B^{k,i-1}$ holds at the beginning of ith policy improvement step and $P^{k,i} \subseteq O^{k,i}$ holds at the end of ith policy improvement due to Lemma 7. This means that all

vertices in $P^{k,i}$ achieve the optimal action and the cost function value at the end of ith policy improvement step. That is, for all $x \in P^{k,i}$, we have that $\mu^{k,i}(x) = \mu^{k,*}(x)$ and $J_{\mu^{k,i}}(x) = J^{k,*}(x)$.

Since for all vertices in $B^{k,i-1}$ there is no update observed between policies $\mu^{k,i}$ and $\mu^{k,i-1}$, we have that $\mu^{k,i}(x) = \mu^{k,i-1}(x) = \mu^{k,*}(x)$ holds for all $x \in P^{k,i} \subseteq B^{k,i-1}$. Let us check the cost function value of vertices in $P^{k,i}$ at the beginning of ith policy improvement step. We already know that vertices in $P^{k,i-1}$ have achieved their optimal cost function value. Since $P^{k,i} = P^{k,i-1} \cup \partial P^{k,i-1}$ holds, we only need to check cost function values for vertices in the boundary set $\partial P^{k,i-1}$. For all vertices in $\partial P^{k,i-1}$, their cost function value can be expressed as $J_{\mu^{k,i-1}}(x) = g(x, \mu^{k,i-1}(x)) + J_{\mu^{k,i-1}}(f(x, \mu^{k,i-1}(x)))$. We already know that $\mu^{k,i-1}(x) = \mu^{k,*}(x)$ holds for all vertices in $\partial P^{k,i-1} \subseteq B^{k,i-1}$.

Let us define $\mu^{k,*}(x) = u^* \in U^{k,*}(x)$ and $x^* \in S^{k,*}(x)$ such that $x^* = f(x, u^*)$. Since $x^* \in P^{k,i-1}$ holds, the optimal successor achieves its optimal cost function value, that is, $J_{\mu^{k,i-1}}(x^*) = J^{k,*}(x^*)$. Then, for all vertices in $\partial P^{k,i-1}$, we can express their cost function value as $J_{\mu^{k,i-1}}(x) = g(x, u^*) + J^{k,*}(x') = J^{k,*}(x)$. This result shows that all vertices in $P^{k,i}$ already achieve the optimal action and the cost function value at the beginning of ith policy improvement step, that is, $P^{k,i} \subseteq O^{k,i-1}$. We just show that $P^{k,i-1} \subseteq O^{k,i-1}$ implies $P^{k,i} \subseteq O^{k,i-1}$ for all $i = 1, 2, \ldots$. It follows that $P^{k,\ell} \subseteq O^{k,i-1}$ holds for $\ell = 0, 1, \ldots$ by using this inductive argument.

Let us consider the case when $\ell = N^k(x_{\text{init}}) - 1$. We have that $P^{k,N^k(x_{\text{init}})-1} \subseteq O^{k,i-1}$, this implies that all vertices which may be intermediate vertices along optimal paths between x_{init} and goal region achieve their optimal action and cost function value at the beginning of ith policy improvement step. Note that x_{init} is selected for Bellman update at the beginning of ith policy improvement step since its cost function value can be written as $J_{\mu^{k,i-1}}(x_{\text{init}}) = g(x_{\text{init}}, u) + J_{\mu^{k,i-1}}(x')$ where $u \in U^k(x_{\text{init}})$ and $x' \in S^k(x_{\text{init}})$ such that $u = \mu^{k,i-1}(x_{\text{init}})$ and $x' = f(x_{\text{init}}, u)$. This implies that $x_{\text{init}} \in \texttt{pred}(\mathcal{G}^k, x')$ and $J_{\mu^{k,i-1}}(x') < J_{\mu^{k,i-1}}(x_{\text{init}})$, therefore, we have $x_{\text{init}} \in B^{k,i-1}$. However, since the

termination condition holds, Bellman update for x_{init} does not yield any update in its action during ith policy improvement step, that is, $\mu^{k,i}(x_{\text{init}}) = \mu^{k,i-1}(x_{\text{init}})$. We also know that $S^{k,*}(x_{\text{init}}) \subseteq P^{k,N^k(x_{\text{init}})-1}$ holds by definition, and all vertices in $S^{k,*}(x_{\text{init}})$ have the optimal action and the cost function value since $P^{k,N-1} \subseteq O^{k,i-1}$ holds, that is, $\mu^{k,i-1}(x') = \mu^{k,*}(x')$ and $J^{k,i-1}(x') = J^{k,*}(x')$ for all $x' \in S^{k,*}(x_{\text{init}})$. Then, we have $\mu^{k,i}(x_{\text{init}}) \subseteq U^{k,*}(x_{\text{init}})$ and $J_{\mu^{k,i}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}})$ at the end of ith policy improvement step due to Lemma 6. This implies $\mu^{k,i-1}(x_{\text{init}}) = \mu^{k,i}(x_{\text{init}}) = \mu^{k,*}(x_{\text{init}})$. Now, we can check the cost function value of x_{init} at the beginning of ith policy improvement step as follows, that is, $J_{\mu^{k,i-1}}(x_{\text{init}}) = g(x, \mu^{k,i-1}(x_{\text{init}})) + J_{\mu^{k,i-1}}(f(x, \mu^{k,i-1}(x_{\text{init}})))$. We know that $\mu^{k,i-1}(x_{\text{init}}) = \mu^{k,*}(x)$. Let us define $\mu^{k,*}(x_{\text{init}}) = u^* \in U^{k,*}(x_{\text{init}})$ and $x' \in S^{k,*}(x_{\text{init}})$ such that $x' = f(x, u^*)$. Since we have $x' \in P^{k,N^k(x_{\text{init}})-1}$, x' achieves the optimal cost function value, that is., $J_{\mu^{k,i-1}}(x') = J^{k,*}(x')$. Then, we have $J_{\mu^{k,i-1}}(x_{\text{init}}) = g(x, u^*) + J^{k,*}(x') = J^{k,*}(x_{\text{init}})$.

Finally, we prove that $J_{\mu^{k,i-1}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}})$ which leads a contradiction with our initial assumption of terminating with a suboptimal cost function value for initial vertex. Hence, when the `Replan` procedure terminates at the beginning $N^k(x_{\text{init}})$th policy improvement steps, as we have shown, it already computes the optimal action and cost function value for x_{init}. If it terminates more than or equal to $N^k(x_{\text{init}})$ policy improvement steps, then, the optimality result follows from Lemma 8. The `Replan` procedure is guaranteed to terminate after a finite number of policy improvement steps since the policy space is finite. This is owing to properties of policy iteration methods [23].

□

Theorem 7 (Termination at Finite Step)

Let $\mathcal{G}^k = (V^k, E^k)$ be the graph built at the end of kth iteration. Then, the `Replan` *procedure of the* PI-RRT$^{\#}$ *algorithm terminates after at most $(\overline{N}^k + 2)th$ policy improvement steps where $\overline{N}^k = \max_{x \in V^k_{\text{prom}}} N^k(x)$.*

Proof. This theorem is proven by using the fact that the sequence $B^{k,i}$ with respect to i has

a limiting super set, i.e., the set of promising vertices and their predecessors. Then, it is shown that the termination condition of he `Replan` procedure holds when the optimal cost function value and policy for all vertices in the limiting set of $B^{k,i}$ are computed, which takes at most as many policy improvement steps as the number of segments of the longest optimal path connecting vertices in the limiting set of $B^{k,i}$ to the goal region.

Let us assume, ad absurdum, that the `Replan` procedure does not terminate at the end of $(\overline{N}^k + 2)th$ policy improvement step at kth iteration. This implies that there exists a point $x \in B^{k,\overline{N}^k+1}$ such that its cost function value is reduced, and its policy is updated at the end of $(\overline{N}^k + 2)th$ policy improvement step. That is, there exists a control $u \in U^k(x)$ that yields $J_{\mu^{k,\overline{N}^k+1}}(x) > g(x,u) + J_{\mu^{k,\overline{N}^k+1}}(x')$ where $x' \in S^k(x)$ such that $\mu^{k,\overline{N}^k+1} \neq \mu^{k,\overline{N}^k+2}(x) = u$ and $x' = f(x,u)$. We will check if the preceding inequality makes sense by analyzing the value of $J_{\mu^{k,\overline{N}^k+1}}(x)$.

Let us define the following set: $\overline{V}^k_{\text{prom}} = \{\overline{\texttt{pred}}(\mathcal{G}^k, x) : x \in V^k_{\text{prom}}\}$

By definition we have $P^{k,\overline{N}^k} = V^k_{\text{prom}}$, and this implies $V^k_{\text{prom}} \subseteq O^{k,\overline{N}^k} \subseteq O^{k,\overline{N}^k+1}$ due to Lemma 8 and Lemma 4. For all $x \in V^k_{\text{prom}}$, we have $J_{\mu^{k,\overline{N}^k+1}}(x) = J^{k,*}(x) < J^{k,*}(x_{\text{init}}) \leq J_{\mu^{k,\overline{N}^k+1}}(x_{\text{init}})$, and this implies $\overline{\texttt{pred}}(\mathcal{G}, x) \in B^{k,\overline{N}^k+1}$. Therefore, $\overline{V}^k_{\text{prom}} \subseteq B^{k,\overline{N}^k+1}$ by definition. Since $V^k_{\text{prom}} \in O^{k,\overline{N}^k}$ and $\overline{V}^k_{\text{prom}} \subseteq B^{k,\overline{N}^k+1}$, it can also be shown, similar to Lemma 7, that all vertices of $\overline{V}^k_{\text{prom}}$ achieve its optimal cost value and policy after $(\overline{N}^k + 1)th$ policy improvement step, that is, $J_{\mu^{k,\overline{N}^k+1}}(x) = J^{k,*}(x),\ \mu^{k,\overline{N}^k+1}(x) \subseteq U^{k,*}(x) \quad \forall x \in \overline{V}^k_{\text{prom}}$.

Note that for the successor vertex of x_{init} along the optimal path between x_{init} and the goal region we have that $N^k(x') = N^k(x_{\text{init}}) - 1 \leq \overline{N}^k$ since $x' \in V^k_{\text{prom}}$. This implies that $N^k(x_{\text{init}}) \leq \overline{N}^k + 1$, and therefore, we have that $J_{\mu^{k,\overline{N}^k+1}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}})$ due to Lemma 8. By definition, for all $x \in V^k$ with $\overline{\texttt{pred}}(\mathcal{G}^k, x) \in B^{k,\overline{N}^k+1}$, we have that $J_{\mu^{k,\overline{N}^k+1}}(x) < J_{\mu^{k,\overline{N}^k+1}}(x_{\text{init}})$. Since $J_{\mu^{k,\overline{N}^k+1}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}})$, we have the following:

$$J^{k,*}(x) \leq J_{\mu^{k,\overline{N}^k+1}}(x) < J_{\mu^{k,\overline{N}^k+1}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}}) \quad \Rightarrow \quad J^{k,*}(x) < J^{k,*}(x_{\text{init}})$$

Therefore, we also have $x \in V^k_{\text{prom}}$ and $\overline{\texttt{pred}}(\mathcal{G}^k, x) \in \overline{V}^k_{\text{prom}}$ which implies that $B^{k,\overline{N}^k+1} \subseteq$

$\overline{V}^k_{\text{prom}}$. Following the two preceding results, we have $B^{k,\overline{N}^k+1} = \overline{V}^k_{\text{prom}}$.

If these results are used for, by our initial assumption, the point $x \in B^{\overline{N}^k+1} = \overline{V}^k_{\text{prom}}$ whose policy is updated during $(\overline{N}^k + 2)th$ policy improvement step, we have $J^{k,*}(x) = J_{\mu^{k,\overline{N}^k+1}}(x) > g(x,u) + J_{\mu^{k,\overline{N}^k+1}}(x') = g(x,u) + J^{k,*}(x')$. This leads the contradiction $J^{k,*}(x) > g(x,u) + J^{k,*}(x')$ which completes the proof. □

Theorem 8 (Asymptotic Optimality)

Let $\mathcal{G}^k = (V^k, E^k)$ be the graph built at the end of kth iteration. As $k \to \infty$, the policy $\mu^{k,N^k}(x_{\text{init}})$ and its corresponding cost function $J_{\mu^{k,N^k}}(x_{\text{init}})$, where N^k is maximum number of policy improvement steps performed during the kth iteration, converge to the optimal policy $\mu^*(x_{\text{init}})$ and the corresponding optimal cost function $J_{\mu^*}(x_{\text{init}})$ associated with continuous state space.

Proof. Let $\mathcal{G}^k = (V^k, E^k)$ be the graph that is constructed by the RRG algorithm at the beginning of kth iteration. In the PI-based RRT$^\#$ algorithm, the optimal cost function value of x_{init} with respect to $\mathcal{G}^k$ is computed during the `Replan` procedure at the end of kth iteration, that is, $J_{\mu^{k,N^k}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}})$ and $\mu^{k,N^k}(x_{\text{init}}) = \mu^{k,*}(x_{\text{init}})$. Since the RRG algorithm is asymptotically optimal, $\mathcal{G}^k$ will encode the optimal path between x_{init} and goal region as the number of iterations goes to infinity. This implies that $J_{\mu^{k,N^k}}(x_{\text{init}}) = J^{k,*}(x_{\text{init}}) \to J^*(x_{\text{init}})$ and $\mu^{k,N^k}(x_{\text{init}}) = \mu^{k,*}(x_{\text{init}}) \to \mu^*(x_{\text{init}})$. Therefore, the PI-based RRT$^\#$ algorithm is asymptotically optimal. □

5.5 Numerical Simulations

We have implemented both the baseline RRT$^\#$ and PI-RRT$^\#$ algorithms in MATLAB and performed Monte Carlo simulations on shortest path planning problems in two different 2D environments. The initial and goal points are shown in Figure 18 in yellow and dark blue squares, respectively. The obstacles are shown in red and the best path computed during each iteration is shown in yellow.

The results are averaged over 100 trials and each trial is run for 10,000 iterations. There is no vertex rejection rule applied during the extension procedure. We compute the total time required to complete a trial and measure the time spent on non-planning (sampling, extension, etc.) and planning related procedures of the algorithms, separately. The growth of the tree is shown in Figure 18. At each iteration, a subset of promising vertices is determined during the policy evaluation step and policy improvement is performed only for these vertices. The promising vertices are shown in magenta in Figure 18.

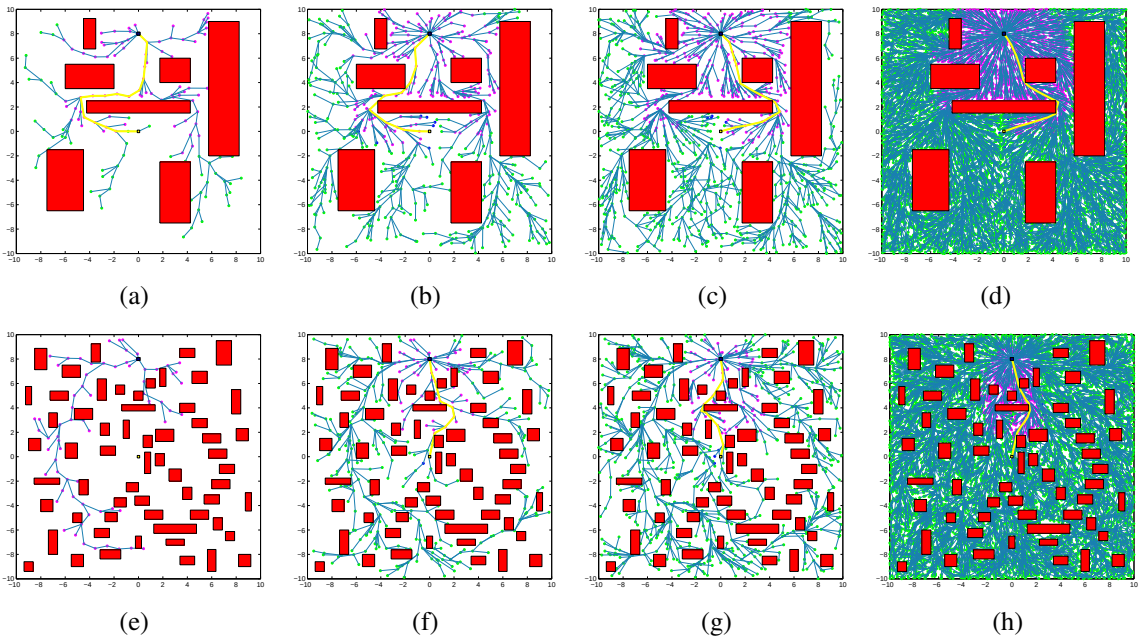


Figure 18: The evolution of the tree computed by PI-$\mathrm{RRT}^{\#}$ algorithm is shown in (a)-(d) for the problem with less cluttered environment, and (e)-(h) for the problem with cluttered environment. The configuration of the trees (a), (e) is at 200 iterations, (b), (f) is at 600 iterations, (c), (g) is at 1,000 iterations, and (d), (h) is at 10,000 iterations.

For the first problem, the average time spent for non-planning (left) and planning (right) related procedures in $\mathrm{RRT}^{\#}$ and PI-$\mathrm{RRT}^{\#}$ algorithms are shown in blue and red colors, respectively in Figure 19. As seen in the left figure, PI-$\mathrm{RRT}^{\#}$ is slightly faster than $\mathrm{RRT}^{\#}$, especially when adding a new vertex to the graph. Since there is no priority queue in the PI-$\mathrm{RRT}^{\#}$ algorithm, it is much cheaper to include a new vertex, and there is no need for ordering. As seen in the right figure, the relation between time and iteration is linear

when the number of iterations becomes large in the log-log scale plot, which implies a polynomial relationship, i.e., $t(n) = cn$, where $t(n)$ is the time required to complete n iterations. The fitted time-iteration lines (dashed) for RRT$^{\#}$ and PI-RRT$^{\#}$ are shown in magenta and green, respectively. In our implementation, we used one processor to perform policy improvement due to simplicity. However, policy improvement step can be done in parallel. One can then divide the set of promising vertices into disjoint sets and assign each of the subsets to a different processor.

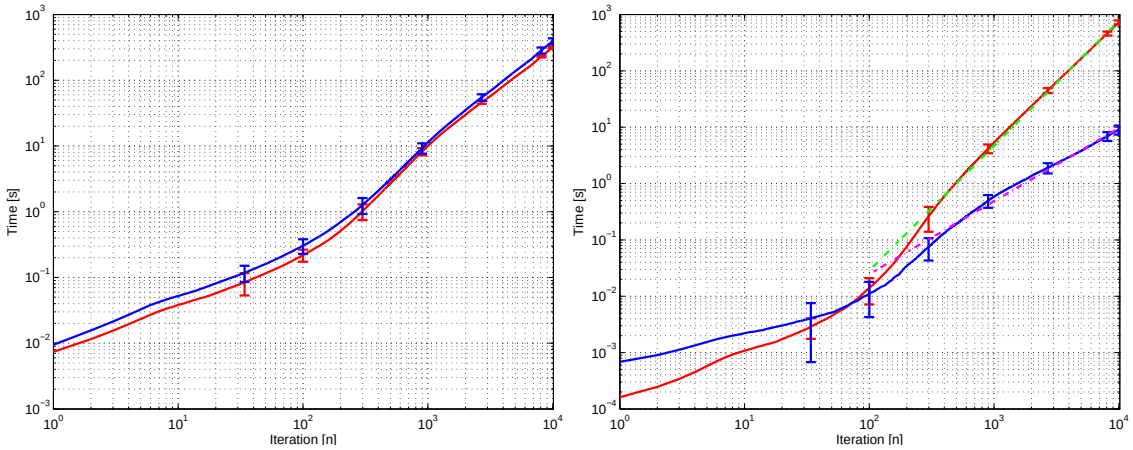


Figure 19: The time required for non-planning (left) and planning (right) procedures to complete a certain number of iterations for the first problem set. The time curve for RRT$^{\#}$ and PI-RRT$^{\#}$ are shown in blue and red, respectively. Vertical bars denote standard deviation averaged over 100 trials.

Let n_{p} denote the number of processors and N denote the computational load per processor, i.e, $L = n/n_{\mathrm{p}}$, where n is number of iteration and can be considered as an upper bound on the number of promising vertices. By simple algebra, one can show that the load per processor needs to satisfy the following relationship for PI-RRT$^{\#}$ algorithm in order to outperform the baseline RRT$^{\#}$ algorithm for faster planning.

$$L = \frac{n}{n_{\mathrm{p}}} > \frac{c_0}{c_{\mathrm{pi}}} n^{1+t_0-t_{\mathrm{pi}}}$$

The PI-RRT$^{\#}$ algorithm can be a better choice than the RRT$^{\#}$ algorithm for problems in high-dimensional search spaces since in those spaces, one needs to run planning

algorithms for a large number of iterations in order to explore the space densely. This requirement induces a bottleneck on the $\mathrm{RRT}^{\#}$ algorithm since the replanning procedure is performed sequentially and requires ordering of vertices. Therefore, this operation may take longer as the number of vertices increases significantly. On the other hand, the PI-$\mathrm{RRT}^{\#}$ algorithm does not require any ordering of vertices and one can keep the replanning tractable by employing more processors (e.g., spawning more threads) as needed in order to keep the desired load per processor requirement. Given the current advancement in parallel computing technologies, e.g., GPUs, a well designed parallel implementation of the PI-$\mathrm{RRT}^{\#}$ may yield real-time execution performance for some problems that are known to be infeasible with the existing algorithms.

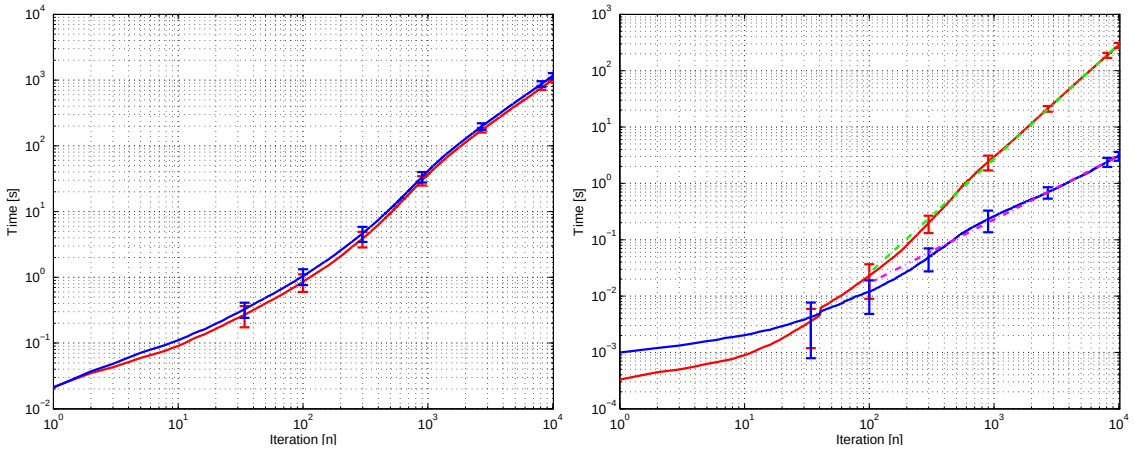


Figure 20: The time required for non-planning (left) and planning (right) procedures to complete a certain number of iterations for the second problem set. The time curve for $\mathrm{RRT}^{\#}$ and PI-$\mathrm{RRT}^{\#}$ are shown in blue and red, respectively. Vertical bars denote standard deviation averaged over 100 trials.

5.6 Conclusion

In this chapter, we show how policy iteration methods can be utilized in the framework of sampling-based algorithms. This novel connection between DP and RRGs yields different types of algorithms which have asymptotic optimality guarantees and execution models (sequential, parallel). The benefit of the proposed algorithm is that the replanning step can be massively parallelized, which can be exploited by computational powers of GPUs.

Chapter VI

MACHINE LEARNING FOR MOTION PLANNING

6.1 Overview

Sampling-based motion planning algorithms need to incorporate efficient exploration strategies in order to gather information about the possibly high-dimensional search space. Most exploration strategies implement a form of rejection sampling in order to collect a large number of collision-free configurations. Rejection sampling is used mainly owing to its implementation simplicity. However, this approach is redundant since not all collision-free configurations have the potential to be part of the optimal solution at a given query. In this chapter, we propose to use machine learning ideas in order to estimate the relevant region of the search space, that is, the region where the optimal path is more likely to be found. The result is that future samples are drawn from the relevant region with increased probability, as the number of iterations increases.

The idea of using machine learning fits naturally within the framework of sampling-based algorithms since an incremental dataset describing the topology of the configuration space is readily available from previous samples. Robotic motion planning can be interpreted as a form of learning problem, since the high-dimensional configuration search space is not known explicitly a priori. Therefore, its solution inherently poses a fundamental problem, which is known as the *exploration versus exploitation dilemma*. Specifically, as the motion planner starts gathering more information about the search space, it may favor exploitation of the current knowledge in order to improve the best solution which has been computed so far. This is mainly due to the fact that the space requirements of an exact representation of the configuration space grows very quickly with the problem dimensionality and the number of obstacles, and hence exhaustive search is impractical (for example, it

has been proven in [104] that the general motion planning problem is PSPACE-complete). Therefore, a motion planner needs to devote some time for exploitation, i.e., to produce a "good enough" solution based on the available information, as exploration progresses. However, concentrating only on improving the current best solution may cause the planner to get stuck in a local minimum since potentially better solutions may have not been explored yet. Therefore, a motion planner should perform the exploration and exploitation tasks in harmony, striking a balance between the two.

Despite the recent advances in the development of motion planners for high-dimensional spaces using sampling-based methods [9, 61, 65, 69, 80, 95], efficient exploration still remains a challenging issue for sampling-based planners [83]. However, for many problems an admissible heuristic and an approximation of the optimal cost-to-come (or cost-to-go) function is available during the search. In such cases, one can characterize the relevant region of the given task, i.e., the subset of the search space which contains the optimal solution. In this context, exploration can be viewed as a problem of learning the relevant region associated with the given task. Interestingly, the authors of this paper have shown that this region can be approximated incrementally as a by-product of the $\mathrm{RRT}^{\#}$ algorithm [9].

In this chapter, we follow up on this insight and use machine learning ideas in order to achieve better exploration of the search space. This is based on the simple fact that since incremental sampling-based algorithms collect a lot of data about the planning problem as iterations progress, one can utilize this information to provide informative labels of the collected samples (obstacle or free) and to associate approximate cost (utility) values with each sample. These labels, collectively, can be used to guide the selection of future samples. By employing active learning and by inferencing based on the collected data, one can guide future samples toward the favorable region of the search space without invoking the computationally expensive collision checking and local steering procedures [40], thus speeding up the algorithm.

The proposed adaptive sampling strategy is integrated in the $\mathrm{RRT}^{\#}$ algorithm to guide the future exploration of the search space. We give a detailed explanation of the proposed adaptive sampling strategy in the subsequent sections. Our numerical simulations demonstrate that the proposed adaptive sampling strategy can reduce the number of vertices in the underlying search graph significantly, yet the path-planner is able to produce high quality paths.

6.2 Related Work

A plethora of approaches have been developed in order to guide the sampling strategy toward a specific part of the configuration space. A comparative study of these approaches can be found in [51]. The majority of these methods try to address the so-called "narrow passage problem," which deals with drawing more samples from difficult and complex parts of the configuration space [53, 54]. More recently, Bialkowski has proposed an approach which guides samples towards the free space [28]. In [35], the authors have proposed an entropy-guided exploration strategy to guide sampling toward the regions that would yield maximal expected information gain. The approach is elegant but owing to the computations involved it appears to be more appropriate for an off-line construction of the graph, which is the case for multi-query algorithms, such as PRM. In [37] the authors keep a history of extension attempts for each state and the results (success or fail), which are used to compute the utility of each state in an information-theoretic sense. This algorithm guides exploration in a way so as to increase the overall utility.

Most closely related to our work is the work in [36] and [96]. In [36] the authors propose an approximate approach based on machine learning ideas for multi-query algorithms that attempts to reduce collision-checking time. However, the current work differs from this approach in several ways. First, the authors of [36] only focus on adapting the sampling strategy to collect more collision-free samples, not for learning the relevant region. Their exploration strategy is tailored to reduce the variance of an approximated model of

the configuration space. Although *active sampling* generates useful samples which yield an accurate model, these configurations are not necessarily related to the solution of the task of interest. This is mainly due to the fact that the approach in [36], similarly to [35], is designed for multi-query algorithms, i.e., the task is not known a priori. Second, our approach is built on fundamentally different assumptions even when only the "learning the configuration space problem" (classification) is considered. In [36] it is assumed that the feature vector and its label (x, y) are jointly Gaussian, and the prediction algorithm is developed by utilizing the nice properties of Gaussian distributions (i.e., conditional distribution of multivariate Gaussian distribution is also a Gaussian). This assumption seems to be reasonable for typical classification problems when there is no information about the underlying structure. However, in the context of sampling-based motion planning, one has some extra information. For example, when rejection sampling is used, we know that the underlying class conditional distribution is a uniform pdf which is defined over a bounded domain and has support which is unknown but can be explored. The use of Gaussian distributions, which do not have compact support, conflicts with the underlying structure of the problem. In our approach, we leverage this key observation and estimate the class conditional distributions directly by estimators on a bounded domain, without resorting to the relaxation of y being continuous.

Finally, in [96], the authors proposed an approach which uses instance-based learning techniques. Their approach stores previous local planning queries and their outcomes (in-collision or collision-free) in a table. Then, a k-NN density estimator computes the probability of a given query point to be in-collision without calling the actual collision checker. The exact collision checking processes is postponed, and is performed only for those points that have a small probability of being in-collision. This idea is also used in our work and can be considered as an alternative way of estimating the posterior distribution. However, it is known that k-NN density estimators have several drawbacks, e.g., they are prone to local noise, yield an estimate distribution with heavy tails, etc; also, the resulting

density estimate is not a true pdf since its integral over the whole configuration space diverges [29]. Our approach, on the other hand, does not have any such problems thanks to the nice properties of kernel density estimators.

6.3 Machine Learning Guided Exploration

6.3.1 Problem Formulation

Let $\mathcal{X}$ denote the configuration space, which is assumed to be an open subset of $\mathbb{R}^d$, where $d \in \mathbb{N}$ with $d \geq 2$. Let the *obstacle region* and the *goal region* be denoted by $\mathcal{X}_{\text{obs}}$ and $\mathcal{X}_{\text{goal}}$, respectively. The obstacle-free space is defined by $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}$. Let the *initial configuration* be denoted by $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. Let $\mathcal{G} = (V, E)$ denote a graph, where V and $E \subseteq V \times V$ are finite sets of vertices and edges, respectively. We will use graphs to represent the connections between a (finite) set of configuration points selected randomly from $\mathcal{X}_{\text{free}}$. Given a vertex $v \in V$, the function $\mathrm{g} : v \mapsto c$ returns a non-negative real number c, which is the cost of the path to v from a given initial state $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. We will use $\mathrm{g}^*(v)$ to denote the optimal cost-to-come value of the vertex v which can be achieved in $\mathcal{X}_{\text{free}}$. Given a vertex $v \in V$, and a goal region $\mathcal{X}_{\text{goal}}$, the heuristic function $\mathtt{h} : (v, \mathcal{X}_{\text{goal}}) \mapsto c$ returns an estimate c of the optimal cost from v to $\mathcal{X}_{\text{goal}}$; we set $\mathtt{h}(v) = 0$ if $v \in \mathcal{X}_{\text{goal}}$. The function $\mathtt{h}$ is an admissible heuristic if it never overestimates the actual cost of reaching $\mathcal{X}_{\text{goal}}$. In this chapter, we always assume that $\mathtt{h}$ is an admissible heuristic. We wish to solve the following problem:

Optimal motion planning problem: Given a bounded and connected open set $\mathcal{X} \subset \mathbb{R}^d$, the sets $\mathcal{X}_{\text{free}}$ and $\mathcal{X}_{\text{obs}} = \mathcal{X} \backslash \mathcal{X}_{\text{free}}$, an initial point $x_{\text{init}} \in \mathcal{X}_{\text{free}}$ and a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$, find the minimum-cost path connecting x_{init} to $\mathcal{X}_{\text{goal}}$.

In sampling-based algorithms, the planning algorithm avoids exhaustive discretization of the search space by randomly drawing configurations which are incorporated into a tree or a graph. The method of random generation of these configurations is called a *sampling strategy*. A good sampling strategy should adapt to the topology of the search space and

provide information that can improve the computed solution.

Learning problem: Let $x^*_{\text{goal}} \in \mathcal{X}_{\text{goal}}$ be the point in the goal region that has the lowest optimal cost-to-come value in $\mathcal{X}_{\text{goal}}$, i.e., let $x^*_{\text{goal}} = \text{argmin}_{x \in \mathcal{X}_{\text{goal}}} \mathtt{g}^*(x)$. The *relevant region* of $\mathcal{X}_{\text{free}}$ is the set of points x for which the optimal cost-to-come value of x, plus the estimate of the optimal cost moving from x to $\mathcal{X}_{\text{goal}}$ is less than the optimal cost-to-come value of x^*_{goal}, that is,

$$\mathcal{X}_{\text{rel}} = \{x \in \mathcal{X}_{\text{free}} : \mathtt{g}^*(x) + \mathtt{h}(x) < \mathtt{g}^*(x^*_{\text{goal}})\}. \tag{28}$$

Points that lie in $\mathcal{X}_{\text{rel}}$ have the potential to be part of the optimal path starting at x_{init} and terminating in $\mathcal{X}_{\text{goal}}$. Our goal is to learn $\mathcal{X}_{\text{rel}}$ and develop a sampling strategy that draws samples only from $\mathcal{X}_{\text{rel}}$. This problem can be formulated as a combination of a classification and a regression problem. First, we need to predict the label of a given arbitrary point $x \in \mathcal{X}$, and then its cost-to-come value needs to be computed approximately, using some regression technique to check if inequality (28) is satisfied.

6.3.2 Approach

Before explaining our approach, some terminology needs to be introduced. Let $\mathcal{X}$ and $\mathcal{Y}$ denote the space of input and output values, respectively. Let $x^{(i)} \in \mathcal{X}$ be the *feature vector* of the *i*th example, also called "input" variables, and let $y^{(i)} \in \mathcal{Y}$ be its label, also called the "output" or *target* variable. A pair $\left(x^{(i)}, y^{(i)}\right)$ is called a *training example*. The *training set* is a list of m training examples of the form $\mathcal{D} = \left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\right\}$. In a supervised learning framework, given the training set $\mathcal{D}$, a learning algorithm seeks a function $\hat{y}_\ell : \mathcal{X} \mapsto \mathcal{Y}$ so that $\hat{y}_\ell(x)$ is a "good" predictor for the corresponding value of y. The function $\hat{y}_\ell$ is called a *hypothesis*, for historical reasons. The target variable that needs to be predicted can be continuous or it may take a finite number of discrete values. The learning problem is a *regression* problem when $\mathcal{Y}$ is continuous, and is a *classification* problem if $\mathcal{Y}$ is a discrete set.

Two problems arise in the context of sampling-based algorithms: a classification problem, i.e., the prediction of the label of an unobserved sample x, and a regression problem, i.e., the prediction of the optimal cost-to-come value of the sample x.

6.3.3 Learning the Configuration Space

Given the training set $\mathcal{D} = \left\{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\right\}$ where the pair $\left(x^{(i)}, y^{(i)}\right)$ denotes a randomly drawn point and its label computed by the collision-checker at the *i*th iteration, we wish to find a function $\hat{y}_{cs} : \mathcal{X} \mapsto \{-1, 1\}$ that gives a good prediction for determining if a given point x is in the obstacle space or the free space.

This problem can be solved efficiently via a Bayesian classifier, which makes decisions based on class conditional distributions and priors [29]. The approach computes two approximate probability density functions (pdf) in order to determine where the obstacle-free and obstacle spaces lie in the search space, based on the available data at any given iteration. Then, the classifier uses the Bayesian rule to predict if a given point x is in $\mathcal{X}_{\text{free}}$ or $\mathcal{X}_{\text{obs}}$. A real collision checking is performed *only* for points which are classified as collision-free. All of the samples, regardless if they are in collision or not, are stored in a list which forms the training set $\mathcal{D}$. A kernel density estimator is used to learn the associated class conditional distributions. The kernel density estimator $\hat{f}_X(x)$ for the estimation of the density value $f_X(x)$ at point x is defined as

$$\hat{f}_X(x) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{K}_{\mathbf{H}} \left(x - x^{(i)}\right), \tag{29}$$

where $\mathbf{H}$ is the bandwidth matrix (nonsingular) and $\mathcal{K}_{\mathbf{H}} : \mathbb{R}^d \mapsto \mathbb{R}$ denotes the multivariate kernel function which is defined as follows:

$$\mathcal{K}_{\mathbf{H}}(x) = \frac{1}{\det(\mathbf{H})} \mathcal{K}(\mathbf{H}^{-1} x). \tag{30}$$

We use a diagonal bandwidth matrix for the sake of simplicity, i.e., $\mathbf{H} = \texttt{diag}(h, \ldots, h)$ where the kernel function $\mathcal{K}$ satisfies the following properties:

i) $\mathcal{K}$ is a density function, that is, $\int_{\mathbb{R}^d} \mathcal{K}(x)\, \mathrm{d}x = 1$ and $\mathcal{K}(x) \geq 0$.

ii) $\mathcal{K}$ is symmetric, that is, $\int_{\mathbb{R}^d} x\mathcal{K}(x)\,\mathrm{d}x = 0$.

Typical kernels involve the Uniform, Gaussian and Epanechnikov kernel functions [110]. In this work, we use the Epanechnikov kernel function

$$\mathcal{K}(x) = \frac{d+2}{2\zeta_d}(1 - x^\intercal x)\mathrm{I}(x^\intercal x \leq 1), \tag{31}$$

where $\mathrm{I}(\cdot)$ is the indicator function and ζ_d is the measure (volume) of the unit sphere in $\mathbb{R}^d$.

6.3.4 Proposed Adaptive Sampling Strategy

The proposed adaptive sampling strategy is given in Algorithm 10. First, the algorithm initializes the lists used to store the collected samples with the empty set and initializes the sampling pdf $\hat{f}_X$ with a pdf which is uniform over $\mathcal{X}$. Then, the algorithm incrementally samples from $\hat{f}_X$ in Line 5. In the subsequent step, a collision-checking operation is performed for the randomly generated sample x_{rand}. The sample x_{rand} is stored in either X_{free} or X_{obs} based on the result of the collision-checking.

In Lines 8 and 11, the `DensityEstimator` procedure implements a nonparametric density estimation method. In this work, we have implemented a kernel density estimator that uses the multivariate Epanechnikov kernel with variable, but uniform in all dimensions, bandwidth. In our implementation, the bandwidth h is updated as a function of the size of the training set $\mathcal{D}$ as follows

$$h \propto (\log(|\mathcal{D}|)/|\mathcal{D}|)^{1/d}. \tag{32}$$

The Epanechnikov kernel in (31) has been used instead of, say, a Gaussian kernel because of its finite support. This property makes querying the density value of a given point tractable. For any kernel of finite support, the summation in equation (29) needs to be performed for only the local neighbors of the query point. This neighbor set can be computed efficiently using specific spatial data structures, such as, kd-trees [106]. Simply, the density value of point x is computed using equation (29) and it predicts how likely is for the point x

Algorithm 10: Adaptive Sampling Algorithm

1 **AdaptiveSampling**($\mathcal{X}$)
2 $X_{\text{obs}} \leftarrow \emptyset$; $X_{\text{free}} \leftarrow \emptyset$;
3 $\hat{f}_X(\cdot) \leftarrow p_{\text{uniform}}(\cdot|\mathcal{X})$;
4 **for** $i = 1$ *to* N **do**
5 $x_{\text{rand}} \leftarrow \texttt{SampleDensity}(\hat{f}_X)$;
6 **if** $\texttt{OnObstacle}(x_{\text{rand}})$ **then**
7 $X_{\text{obs}} \leftarrow X_{\text{obs}} \cup \{x_{\text{rand}}\}$;
8 $b_{\text{obs}} \leftarrow \texttt{DensityEstimator}(X_{\text{obs}})$;
9 **else**
10 $X_{\text{free}} \leftarrow X_{\text{free}} \cup \{x_{\text{rand}}\}$;
11 $b_{\text{free}} \leftarrow \texttt{DensityEstimator}(X_{\text{free}})$;
12 $\hat{f}_X \leftarrow \texttt{Classifier}(X, b_{\text{obs}}, b_{\text{free}})$;
13 $X \leftarrow (X_{\text{obs}}, X_{\text{free}})$;
14 **return** X;

to be in the obstacle-free or obstacle spaces. In Line 12, the `Classifier` procedure implements a Bayesian classifier and the label of a given point x is determined by the following Bayesian decision rule:

$$\hat{y}_{cs}(x) = \begin{cases} 1 & \text{if } q_{\text{free}}(x) \geq q_{\text{obs}}(x), \\ -1 & \text{otherwise}, \end{cases} \tag{33}$$

where $q_{\text{obs}}(x) := \eta P(x|y = -1)P(y = -1)$ and $q_{\text{free}}(x) := \eta P(x|y = 1)P(y = 1)$, where $\eta = 1/P(x)$ is a normalizing coefficient. This classifier separates the configuration space $\mathcal{X}$ into two approximate sets of obstacle $\widehat{\mathcal{X}}_{\text{obs}}$ and obstacle-free $\widehat{\mathcal{X}}_{\text{free}}$ regions, i.e., $\widehat{\mathcal{X}}_{\text{obs}} = \{x \in \mathcal{X} : \hat{y}_{cs}(x) = -1\}$ and $\widehat{\mathcal{X}}_{\text{free}} = \{x \in \mathcal{X} : \hat{y}_{cs}(x) = 1\}$. At each iteration, the class density functions are approximated by the kernel density estimator based on the available data, as follows

$$b_{\text{obs}} = P(x|y = -1) = \frac{1}{|X_{\text{obs}}|} \sum_{x' \in X_{\text{obs}}} \mathcal{K}_{\mathbf{H}_{\text{o}}}(x - x')$$

$$b_{\text{free}} = P(x|y = 1) = \frac{1}{|X_{\text{free}}|} \sum_{x' \in X_{\text{free}}} \mathcal{K}_{\mathbf{H}_{\text{f}}}(x - x')$$

where $\mathbf{H}_{\text{o}} = \texttt{diag}(h_{\text{o}}, \ldots, h_{\text{o}})$ and $\mathbf{H}_{\text{f}} = \texttt{diag}(h_{\text{f}}, \ldots, h_{\text{f}})$ are computed according to equation (32) using the sizes of X_{obs} and X_{free}, respectively.

The class priors are computed as the ratio of the samples in each class according to the expression $P(y = -1) = |X_{\text{obs}}|/|X|$ and $P(y = 1) = |X_{\text{free}}|/|X|$. Finally, having the $\widehat{\mathcal{X}}_{\text{obs}}$ and $\widehat{\mathcal{X}}_{\text{free}}$ sets, the `Classifier` procedure returns the following pdf which is uniform over $\widehat{\mathcal{X}}_{\text{free}}$:

$$\hat{f}_X(x) = \begin{cases} 1/\mu(\widehat{\mathcal{X}}_{\text{free}}) & \text{if } x \in \widehat{\mathcal{X}}_{\text{free}}, \\ 0 & \text{if } x \in \widehat{\mathcal{X}}_{\text{obs}}. \end{cases} \tag{34}$$

6.3.5 Learning the Cost-to-come (or cost-to-go) Value

Given the set of training data $\mathcal{D} = \left\{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\right\}$ where the pair $\left(x^{(i)}, y^{(i)}\right)$ denotes a randomly drawn point along with its lmc-value (see Ref. [9]) computed by the replanning procedure at the *i*th iteration, we wish to find a function $\hat{y}_{cv} : \mathcal{X} \mapsto \mathbb{R}$ that gives a good estimate of the cost-to-come value of a given point x.

Due to the incremental setting of the sampling-based algorithms, we consider *locally weighted learning* based methods [18], which are a form of lazy learning, to solve the aforementioned regression problem owing to their easy training. In this method, the training data is stored in memory and only a small subset is retrieved to answer a specific query. Relevance of the data is measured by using a distance function (e.g., nearby points look alike or have similar features). For regression problems, the method fits a surface to nearby points using a distance weighted regression as follows:

$$\hat{y}_{cv}(x) = \frac{\sum_i y^{(i)} w(x, x^{(i)})}{\sum_i w(x, x^{(i)})}. \tag{35}$$

The weighting function $w(x, x')$ measures the relevance of two points and can be defined by using a kernel function, for example, $w(x, x') = \mathcal{K}_{\mathbf{H}_{\text{v}}}(x - x')$ where $\mathbf{H}_{\text{v}} = \texttt{diag}(h_{\text{v}}, \ldots, h_{\text{v}})$ is computed from equation (32) according to size of vertex set V. In the proposed approach, whenever a new sample is examined for inclusion in the graph, first its cost-to-come value is estimated using the lmc-values of the neighbor vertices according to equation (35). Then, the new sample is included in the graph if its approximate

cost-to-come value satisfies the following inequality, which is a relaxed version of (28)

$$\widehat{\mathcal{X}}_{\text{rel}} = \{x \in \widehat{\mathcal{X}}_{\text{free}} : \hat{\mathtt{g}}(x) + \mathtt{h}(x) < \mathtt{lmc}(x^*_{\text{goal}})\}. \tag{36}$$

6.3.6 Integration to the $\text{RRT}^{\#}$ Algorithm

The proposed approach can be seamlessly integrated to the $\text{RRT}^{\#}$ algorithm [9]. In fact, the proposed approach can be used with any single-query sampling-based motion planning algorithm, as long as it provides information of the cost-to-come (or cost-to-go) values for all the vertices. However, it is essential for the planning algorithm to provide accurate estimates of these cost values to achieve a good performance. In this chapter we have chosen the $\text{RRT}^{\#}$ algorithm to leverage its fast convergence properties, which is the result of using a relaxation step for the local rewiring of the graph. Details of the $\text{RRT}^{\#}$ algorithm and its variants can be found in [9] and [12]. Instead of implementing a uniform sampling strategy, the `Sample` procedure of the $\text{RRT}^{\#}$ algorithm is replaced with the proposed adaptive sampling strategy. The details of the `Sample` procedure is given in Algorithm 11.

The `Extend` procedure of the $\text{RRT}^{\#}$ algorithm is also modified, and relevancy of a new sample is checked by the proposed approach in Algorithm 12 before invoking any collision checker or solving the local steering problem. If the new sample is predicted to be part of the relevant region, then the typical operations of the $\text{RRT}^{\#}$ algorithm are performed for the inclusion of the new sample in the current graph. Lines 2-11 implement the Bayesian classifier and compute the posterior distribution of the new sample. If the new sample is predicted to be in the obstacle-free space, then the locally weighted regression is applied in Lines 13-20 to determine if the new information has the potential to improve the current best solution.

Algorithm 11: `Sample` Procedure

Sample($\hat{f}_X$)
  $(X, b_{\text{obs}}, b_{\text{free}}) \leftarrow \hat{f}_X$; $(X_{\text{obs}}, X_{\text{free}}) \leftarrow X$;
  $x \leftarrow \texttt{SampleDensity}(\hat{f}_X)$;
  while $\texttt{OnObstacle}(x)$ **do**
    $X_{\text{obs}} \leftarrow X_{\text{obs}} \cup \{x\}$; $X \leftarrow (X_{\text{obs}}, X_{\text{free}})$;
    $b_{\text{obs}} \leftarrow \texttt{DensityEstimator}(X_{\text{obs}})$;
    $\hat{f}_X \leftarrow \texttt{Classifier}(X, b_{\text{obs}}, b_{\text{free}})$;
    $x \leftarrow \texttt{SampleDensity}(\hat{f}_X)$;
  $X_{\text{free}} \leftarrow X_{\text{free}} \cup \{x\}$; $X \leftarrow (X_{\text{obs}}, X_{\text{free}})$;
  $b_{\text{free}} \leftarrow \texttt{DensityEstimator}(X_{\text{free}})$;
  $\hat{f}_X \leftarrow \texttt{Classifier}(X, b_{\text{obs}}, b_{\text{free}})$;
  return $(\hat{f}_X, x)$;

SampleDensity($\hat{f}_X$)
  $(X, b_{\text{obs}}, b_{\text{free}}) \leftarrow \hat{f}_X$; $(X_{\text{obs}}, X_{\text{free}}) \leftarrow X$;
  $P_{\text{p,obs}} = |X_{\text{obs}}|/|X|$; $P_{\text{p,free}} = |X_{\text{free}}|/|X|$;
  do
    $x \leftarrow \texttt{Sample}(\mathcal{X})$;
    $P_{\text{c,obs}} = b_{\text{obs}}(x)$; $P_{\text{c,free}} = b_{\text{free}}(x)$;
    $q_{\text{obs}}(x) = P_{\text{c,obs}} \cdot P_{\text{p,obs}}$; $q_{\text{free}}(x) = P_{\text{c,free}} \cdot P_{\text{p,free}}$;
  while $q_{\text{free}}(x) < q_{\text{obs}}(x)$
  return x;

6.4 Simulation Results

We have performed several simulations in order to confirm the efficiency of the proposed approach. Here we present the results for a two-link robot moving in the plane to demonstrate that the proposed adaptive sampling strategy is capable of generating a high-number of collision-free samples. The workspace and configuration space of the two-link robot are shown in Figure 21. Objects are intentionally placed in the workspace to form narrow passages in the configuration space. The sampling strategy has also been integrated to the $\text{RRT}^{\#}$ algorithm to solve a path planning problem in 2D environment in order to visualize the growth of the search tree.

Algorithm 12: Relevancy Check Procedure

IsRelevant($\mathcal{G}$, X, x)
  $(V, E) \leftarrow \mathcal{G}$; $(X_{\text{obs}}, X_{\text{free}}) \leftarrow X$;
  $P_{\text{p,obs}} = |X_{\text{obs}}|/|X|$; $P_{\text{p,free}} = |X_{\text{free}}|/|X|$;
  $\mathcal{S}_{\text{obs}} \leftarrow \texttt{Near}(X_{\text{obs}}, x, |X_{\text{obs}}|)$; $P_{\text{c,obs}} = 0$;
  foreach $x' \in \mathcal{S}_{\text{obs}}$ **do**
    $P_{\text{c,obs}} = P_{\text{c,obs}} + \mathcal{K}_{\mathbf{H}_{\text{o}}}(x - x')$;
  $\mathcal{S}_{\text{free}} \leftarrow \texttt{Near}(X_{\text{free}}, x, |X_{\text{free}}|)$; $P_{\text{c,free}} = 0$;
  foreach $x' \in \mathcal{S}_{\text{free}}$ **do**
    $P_{\text{c,free}} = P_{\text{c,free}} + \mathcal{K}_{\mathbf{H}_{\text{f}}}(x - x')$;
  $P_{\text{c,obs}} = P_{\text{c,obs}}/|\mathcal{S}_{\text{obs}}|$; $P_{\text{c,free}} = P_{\text{c,free}}/|\mathcal{S}_{\text{free}}|$;
  $q_{\text{obs}}(x) = P_{\text{c,obs}} \cdot P_{\text{p,obs}}$; $q_{\text{free}}(x) = P_{\text{c,free}} \cdot P_{\text{p,free}}$;
  if $q_{\text{free}}(x) \geq q_{\text{obs}}(x)$ **then**
    $\hat{g}(x) = 0$; $w_{\text{total}} = 0$;
    $\mathcal{X}_{\text{near}} \leftarrow \texttt{Near}(\mathcal{G}, x, |V|)$;
    foreach $x' \in \mathcal{X}_{\text{near}}$ **do**
      $w(x, x') = \mathcal{K}_{\mathbf{H}_{\text{v}}}(x - x')$;
      $\hat{g}(x) = \hat{g}(x) + \texttt{lmc}(x')w(x, x')$;
      $w_{\text{total}} = w_{\text{total}} + w(x, x')$;
    $\hat{g}(x) = \hat{g}(x)/w_{\text{total}}$;
    $\texttt{Key}(x) = (\hat{g}(x) + \texttt{h}(x), \hat{g}(x))$;
    return $\texttt{Key}(x) \prec \texttt{Key}(v^*_{\text{goal}})$;
  return `False`;

6.4.1 2D-link Robot

We first tested if the proposed adaptive sampling strategy generates a large number of collision-free configurations. Both uniform and adaptive sampling strategies were used to generate 100,000 samples. In order to demonstrate that the proposed approach eventually draws samples from difficult parts of the configuration space, e.g., narrow passages, all points on the boundary of obstacles were sampled offline and used to initialize the obstacle list X_{obs} of the classifier. By doing so, all narrow passages are blocked at the start of the algorithm.

Figure 22 compares the ratio of the collision-free samples over the total number of samples ($r = |X_{\text{free}}|/|X|$) in a trial for uniform and adaptive sampling strategies, respectively. The ratio plots for adaptive and uniform sampling strategies are shown in green and blue

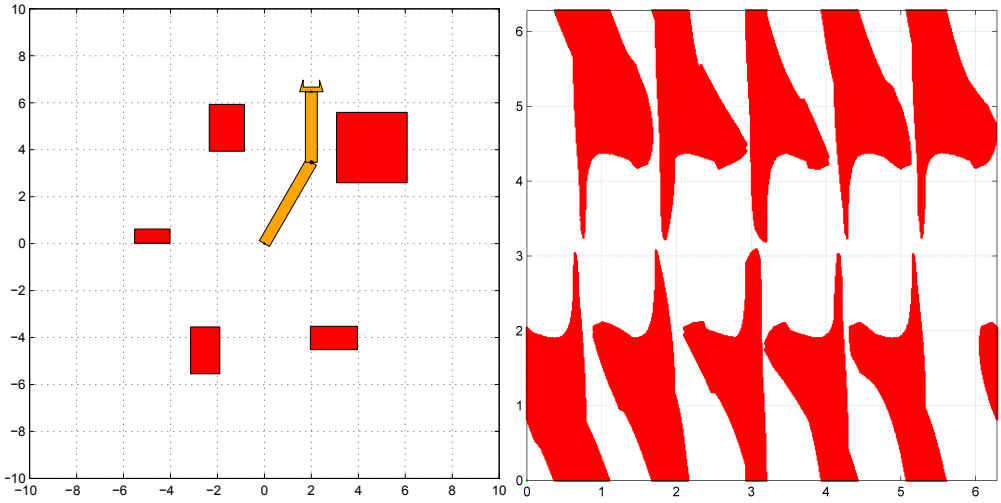


Figure 21: Workspace and configuration space of a 2-link robot.

colors, respectively. It is seen that this ratio converges to one for the proposed approach, whereas it lingers around $r = \mu(\mathcal{X}_{\text{free}})/\mu(X)$ for a uniform sampling strategy, shown in red color.

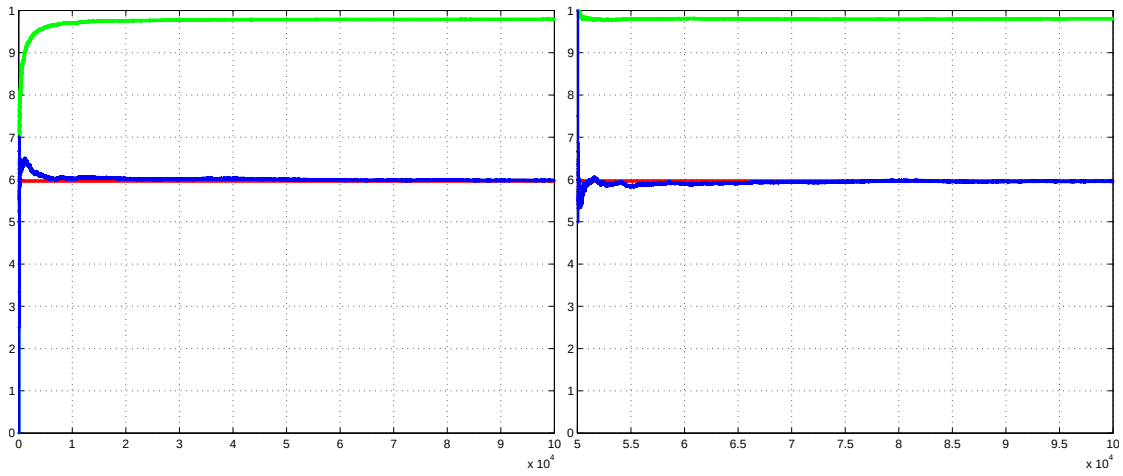
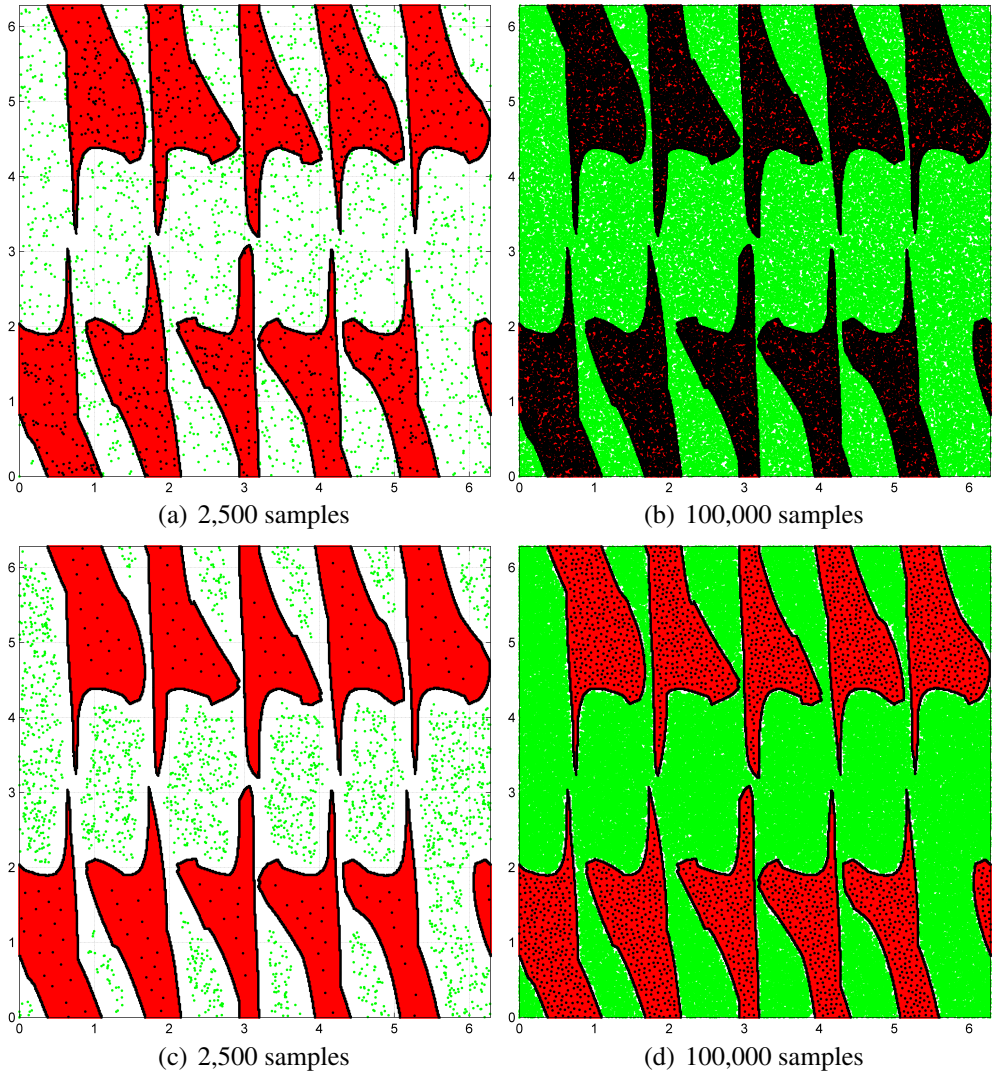


Figure 22: Ratio of the number of collision-free samples over the total number of samples starting from intermediate iterations: left is with $i = 1$, and right is with $i = 50,001$.

The distribution of samples drawn by the uniform and sampling strategies is shown Figure 23. The free space and configuration space obstacles are shown in white and red, respectively. The samples that are free of and on collisions are shown in green and black, respectively. The proposed adaptive sampling strategy significantly reduces the number of points drawn from the obstacle space and generates samples inside the narrow passages

shown, whereas the uniform sampling strategy results in a large number of points on the obstacle space, depending the measure of $\mu(\mathcal{X}_{\text{obs}})$.



(a) 2,500 samples (b) 100,000 samples

(c) 2,500 samples (d) 100,000 samples

Figure 23: The distribution of samples randomly drawn by uniform and adaptive sampling strategies is shown in (a)-(b) and (c)-(d), respectively.

6.4.2 Path Planning in 2D Environment

In this problem, our aim is to find an optimal path that connects a given initial point to the goal region, while minimizing the Euclidean path length in a square environment. The Euclidean distance from a given state to the goal set was used as an admissible heuristic for that state. The growth of the tree at different stages is shown in Figure 24. The initial state

is plotted as a yellow square,the goal region is shown in dark blue with (upper middle), and the obstacles are shown in dark red. The minimal-length path is shown in yellow. As shown in Figure 24, the lowest-cost path computed by the algorithm converges to the optimal solution. Note that in these simulations we have used a slightly different implementation of the algorithms, namely, the tree is rooted to the goal set instead of the initial state and the growth of the tree is reversed.

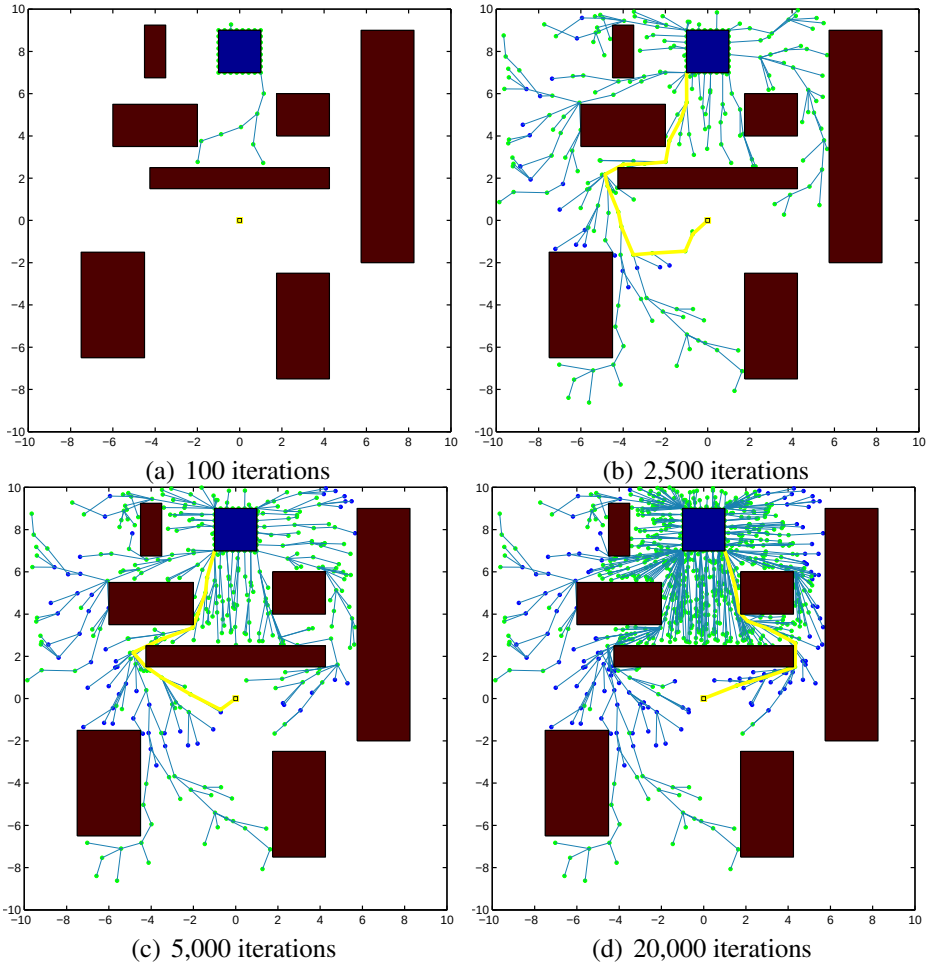
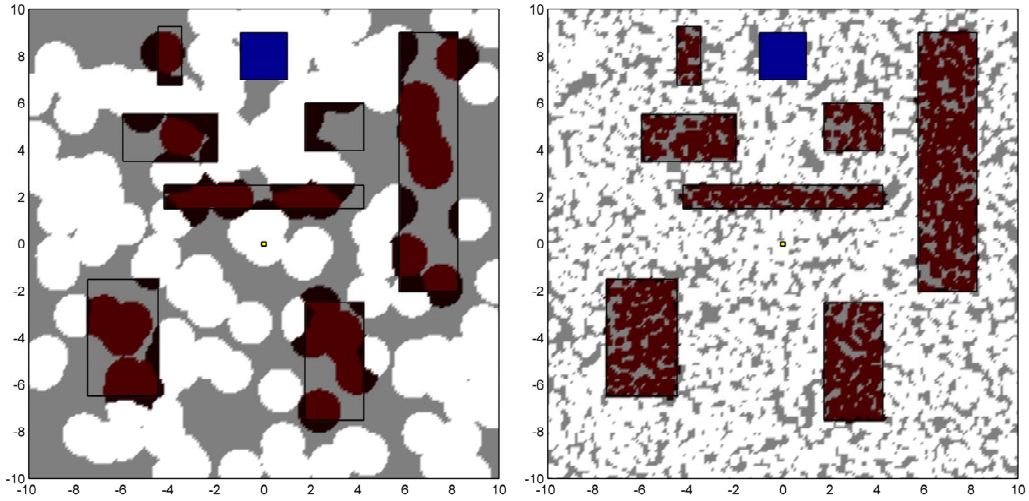


Figure 24: Evolution of the tree shown at different iterations.

It is seen that once an initial solution is computed, exploration is prevented from going toward the unfavorable regions of the configuration space. This helps to greatly reduce the number of vertices kept in the graph, yet it computes high quality solutions.



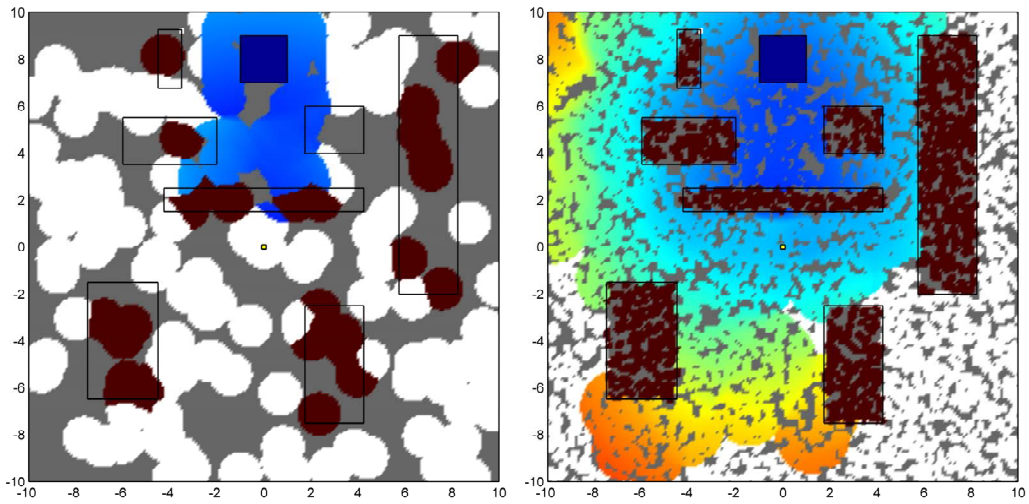
(a) 100 iterations (b) 20,000 iterations

Figure 25: Learned configuration space.

The learned configuration space at different stages of the process are shown in Figure 25. The correctly predicted free space is shown in white color, the correctly predicted obstacle space is shown in dark red, the incorrectly predicted obstacle and free spaces are shown in black color, and the unexplored regions (no prediction is available) are shown in gray color. These plots show how the configuration space looks like from the classifier's perspective. The configuration space is densely gridded and all points are queried to the Bayesian classifier. The results are plotted based on the predicted labels. As seen in Figure 25-(b), the classifier builds an almost exact model of the configuration space, at least in the neighborhood of the relevant region.

The approximate f-value function (heuristic value plus cost-to-go) at different stages is shown in Figure 26. Low and high values of the f-value function are shown in blue and red colors, respectively, and the intermediate values are shown using gradient colors. This function is used to determine if a candidate sample is promising or not.

The approximate relevant region at different stages is shown in Figure 27. The true relevant and approximate regions are shown in blue and purple colors, respectively, and their intersection is shown in green color. In Figure 27-(a), the exact relevant region is plotted based on the true cost-to-go values, which are computed off-line. Since a solution



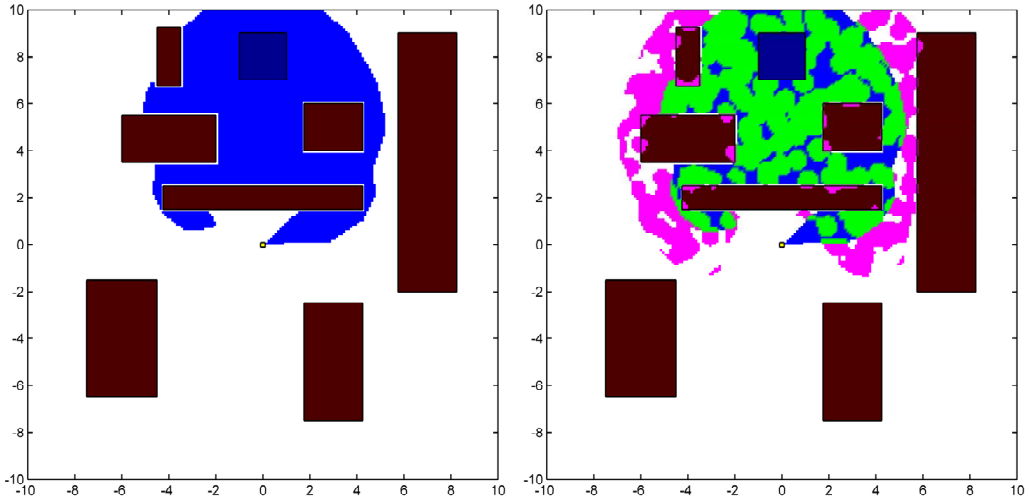
(a) 100 iterations (b) 20,000 iterations

Figure 26: Approximate f-value function (cost-to-go plus heuristic value).

has not been found yet, the algorithm considers initially the whole configuration space as the relevant region. As seen in the next figures, however, the algorithm progressively computes an approximation of the actual relevant region. In these plots the purple and green regions denote the incorrect and correct predictions, respectively.

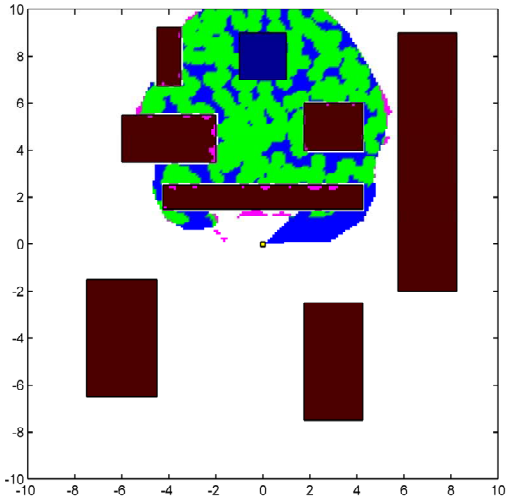
6.5 Conclusion

In this chapter, we propose a novel adaptive sampling strategy and integrate it to the $\mathrm{RRT}^{\#}$ algorithm, an asymptotically optimal sampling-based path-planning algorithm. The proposed adaptive sampling strategy utilizes the history of computed information, specifically, the label of the samples and their cost-to-come (or cost-to-go) values, to guide the exploration towards the region of the search space where samples having a great potential to improve the existing solution are more likely to be found. The approach utilizes ideas from machine learning to make predictions about how likely is for a new sample to be part of the free space and improve the current solution, without calling the computationally expensive collision checking and local steering procedures. Simulations demonstrate the effectiveness of the proposed approach, both in terms of reducing the number of samples lying in the obstacle space, and exploring the relevant region efficiently.

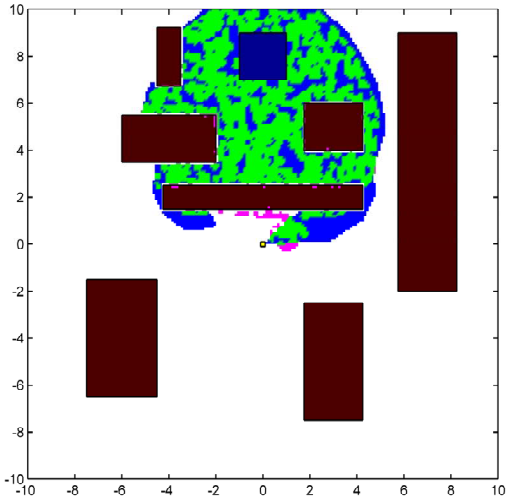


(a) 100 iterations

(b) 2,500 iterations



(c) 5,000 iterations



(d) 20,000 iterations

Figure 27: Approximate relevant region.

Chapter VII

STOCHASTIC MOTION PLANNING

7.1 Overview

This chapter considers optimal control of dynamical systems which are represented by nonlinear stochastic differential equations. It is well-known that the optimal control policy for this problem can be obtained as a function of a value function that satisfies a nonlinear partial differential equation, namely, the Hamilton-Jacobi-Bellman equation. This nonlinear PDE must be solved backwards in time, and this computation is intractable for large scale systems. Under certain assumptions, and after applying a logarithmic transformation, an alternative characterization of the optimal policy can be given in terms of a path integral. Path Integral (PI) based control methods have recently been shown to provide elegant solutions to a broad class of stochastic optimal control problems. One of the implementation challenges with this formalism is the computation of the expectation of a cost functional over the trajectories of the unforced dynamics. Computing such expectation over trajectories that are sampled uniformly may induce numerical instabilities due to the exponentiation of the cost. Therefore, sampling of low-cost trajectories is essential for the practical implementation of PI-based methods. In this chapter, we use incremental sampling-based algorithms to sample useful trajectories from the unforced system dynamics, and make a novel connection between Rapidly-exploring Random Trees (RRTs) and information-theoretic stochastic optimal control. We show the results from the numerical implementation of the proposed approach to several examples.

7.2 Introduction

In [118, 119], the authors showed the connection between Kullback-Leibler (KL) and Path Integral (PI) control with an information-theoretic view of stochastic optimal control. In addition, the same authors derived the iterative path integral optimal control without relying on policy parameterizations, as in [117]. We review the work in [118, 119] starting with the definitions of free energy and relative entropy and their connections to dynamic programming. In addition, we discuss how the iterative scheme developed in [118] and [119] can be modified to incorporate incremental sampling-based methods such as Rapidly-exploring Random Trees (RRT) to guide sampling [39, 83]. An early approach which leverages the RRT algorithm to solve stochastic optimal control problems for linear systems under environmental uncertainty is given in [92].

Within the mathematical framework of path integral control, the Feynman-Kac lemma plays an essential role, since it creates a connection between Stochastic Differential Equations (SDEs) and backward Partial Differential Equations (PDEs). This fundamental connection between SDEs and backward PDEs has inspired new avenues for the development of stochastic control algorithms such as Policy Improvement with Path Integrals (PI^2) [117] that rely on forward sampling. PI^2 has been applied to a plethora of motor control tasks from robotic object manipulation and locomotion to general trajectory optimization and gain scheduling [34, 114, 117], but it relies on a suitable parameterization of the optimal control policy. While policy parameterization such as Dynamic Movement Primitives (DMPs) [59] improves sampling by steering trajectories in high-dimensional state spaces towards areas of interest, it does not exploit the feedback structure provided by the path integral control framework. In PI^2 trajectories are sampled from the initial state of the task, the optimal parameter variations are computed, and the parameters are updated. In the next iteration, trajectories are sampled again from the same initial state and the iterative process continues until convergence. It is clear that in the case of policy parameterization one has to explicitly design the structure of the feedback control policy and then treat the gains as

parameters to be optimized.

With respect to information theoretic formulations of policy search methods, [99], our work here does not depend on policy parameterizations and is grounded on the Relative Entropy - Free Energy Dualities and their connection to Dynamic Programming Principle.

7.3 Notation

A *probability space* is a triple $(\Omega, \mathcal{F}, \mathsf{p})$ where $(\Omega, \mathcal{F})$ is a measurable space with Ω a non-empty set, which is called the *sample space*, $\mathcal{F} \subseteq 2^{\Omega}$ a σ-algebra of subsets of Ω, whose elements are called events, and p is a *probability measure* on $\mathcal{F}$, that is, p is a finite measure on $\mathcal{F}$ with $\mathsf{p}(\Omega) = 1$.

A real random variable is a function $X : \Omega \to \mathbb{R}$ with the property that $\{\omega \in \Omega : X(\omega) \leq x\} \in \mathcal{F}$ for each $x \in \mathbb{R}$. Such a function is said to be $\mathcal{F}$-measurable. An extended (real) random variable can also take the values $\pm\infty$. If X is a random variable on the probability space $(\Omega, \mathcal{F}, \mathsf{p})$, then its *expectation* is defined by

$$\mathbb{E}_{\mathsf{p}}[X] = \int_{\Omega} X(\omega)\, \mathrm{d}\mathsf{p}(\omega), \tag{37}$$

provided that the integral in the right-hand side exists. As usual, and for notational simplicity, in the sequel we will drop the explicit dependence on $\omega \in \Omega$ in (37). In other words, the notation $\mathbb{E}_{\mathsf{p}}[X]$ is another (shorter) notation for the integral $\int X \mathrm{d}\mathsf{p}$.

7.4 Stochastic Control Based on Free Energy and Relative Entropy Dualities

Let $(\Omega, \mathcal{F})$ be a measurable space where Ω is a non-empty set and $\mathcal{F} \subseteq 2^{\Omega}$ is a σ-algebra of subsets of Ω, and let $\mathbf{P}(\Omega)$ be the set of all probability measures defined on $(\Omega, \mathcal{F})$.

Definition 1

Let $\mathsf{p} \in \mathbf{P}(\Omega)$ be a probability measure, $\mathbf{x} = \mathbf{x}(\omega),\ \omega \in \Omega$ be a random variable, $t, \rho \in \mathbb{R}$ be real numbers, and let $\mathcal{J}(\mathbf{x}, t)$ be a measurable function. The *Helmholtz free energy* of

$\mathcal{J}(\mathbf{x}, t)$ with respect to p is defined by

$$\begin{aligned}\mathcal{E}_{\mathsf{p}}\left(\mathcal{J}(\mathbf{x}, t); \rho\right) &= \log\left(\int \exp\left(\rho \mathcal{J}(\mathbf{x}, t)\right) \mathrm{d}\mathsf{p}\right) \\ &= \log \mathbb{E}_{\mathsf{p}}\left[\exp\left(\rho \mathcal{J}(\mathbf{x}, t)\right)\right]. \end{aligned} \tag{38}$$

Definition 2

Let $\mathsf{p}, \mathsf{q} \in \mathbf{P}(\Omega)$ be two probability measures. The *relative entropy* of p with respect to q is defined as[1]:

$$\mathbb{KL}\left(\mathsf{q} \| \mathsf{p}\right) = \begin{cases} \displaystyle\int \log\left(\frac{\mathrm{d}\mathsf{q}}{\mathrm{d}\mathsf{p}}\right) \mathrm{d}\mathsf{q} & \text{if } \mathsf{q} \ll \mathsf{p} \text{ and } \log\left(\frac{\mathrm{d}\mathsf{q}}{\mathrm{d}\mathsf{p}}\right) \in L^1, \\ +\infty & \text{otherwise.} \end{cases} \tag{39}$$

We will also consider the function $\xi(\mathbf{x}, t)$, defined by

$$\xi(\mathbf{x}, t) = \tfrac{1}{\rho} \mathcal{E}_{\mathsf{p}}\left(\mathcal{J}(\mathbf{x}, t); \rho\right) = \tfrac{1}{\rho} \log \mathbb{E}_{\mathsf{p}}\left[\exp\left(\rho \mathcal{J}(\mathbf{x}, t)\right)\right]. \tag{40}$$

To derive the basic relationship between free energy and relative entropy [43], we express the expectation $\mathbb{E}_{\mathsf{p}}$ taken under the probability measure p as a function of the expectation $\mathbb{E}_{\mathsf{q}}$ taken under the probability measure q. More precisely, we have:

$$\mathbb{E}_{\mathsf{p}}\left[\exp\left(\rho \mathcal{J}(\mathbf{x}, t)\right)\right] = \int \exp\left(\rho \mathcal{J}(\mathbf{x}, t)\right) \frac{\mathrm{d}\mathsf{p}}{\mathrm{d}\mathsf{q}} \mathrm{d}\mathsf{q}.$$

By taking the logarithm of both sides of the previous equation and by making use of Jensen's inequality [43], it can be shown that:

$$\log \mathbb{E}_{\mathsf{p}}\left[\exp\left(\rho \mathcal{J}(\mathbf{x}, t)\right)\right] \geq \int \rho \mathcal{J}(\mathbf{x}, t) \, \mathrm{d}\mathsf{q} - \mathbb{KL}\left(\mathsf{q} \| \mathsf{p}\right). \tag{41}$$

Let $\rho < 0$. By multiplying both sides of (41) with $-1/|\rho|$, one obtains:

$$\boxed{\xi(\mathbf{x}, t) = -\frac{1}{|\rho|} \mathcal{E}_{\mathsf{p}}\left(\mathcal{J}(\mathbf{x}, t); \rho\right) \leq \mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x}, t)\right] + \frac{1}{|\rho|} \mathbb{KL}\left(\mathsf{q} \| \mathsf{p}\right)} \tag{42}$$

[1] Given two probability measures p and q, we say that q is *absolutely continuous* with p and write $\mathsf{q} \ll \mathsf{p}$ if $\mathsf{q} = 0 \Rightarrow \mathsf{p} = 0$, see page 161 of [94].

where $\mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)\right] = \int \mathcal{J}(\mathbf{x},t)\,\mathrm{d}\mathsf{q}$. The inequality (42) provides us with a duality relationship between relative entropy and free energy. Essentially, one could define the following minimization problem:

$$-\frac{1}{|\rho|}\mathcal{E}_{\mathsf{p}}\left(\mathcal{J}(\mathbf{x},t);\rho\right) = \inf_{\mathsf{q}\in\mathbf{P}(\Omega)}\left(\mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)\right] + \frac{1}{|\rho|}\mathbb{KL}(\mathsf{q}\|\mathsf{p})\right). \tag{43}$$

It can be shown that the infimum in (43) is attained at q^*, where

$$\mathrm{d}\mathsf{q}^* = \frac{\exp\left(-|\rho|\mathcal{J}(\mathbf{x},t)\right)}{\int \exp\left(-|\rho|\mathcal{J}(\mathbf{x},t)\right)\mathrm{d}\mathsf{p}}\,\mathrm{d}\mathsf{p}. \tag{44}$$

One can put q^* in (43) to verify that the right-hand side of the equation is indeed equal to its lower bound [43].

A rather intuitive way of writing (42) is to express it in the following form:

$$\underbrace{-\frac{1}{|\rho|}\mathcal{E}_{\mathsf{p}}\left(\mathcal{J}(\mathbf{x},t);\rho\right)}_{\textbf{Helmholtz Free Energy}} \leq \underbrace{\text{Mean State Cost} + \frac{1}{|\rho|}\text{Information Cost}}_{\textbf{Non-Equilibrium Free Energy}} \tag{45}$$

where "Mean State Cost" and "Information Cost" are defined as $\mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)\right]$ and $\mathbb{KL}\left(\mathsf{q}\|\mathsf{p}\right)$, respectively.

In the next sections, we derive the form of (43) for the case when $\mathbf{x}$ is the state of a nonlinear stochastic differential equation affine in noise and control.

7.4.1 Application of the Legendre Transformation to Stochastic Differential Equations

We consider the general uncontrolled and controlled stochastic dynamics affine in noise as follows:

$$\mathrm{d}\mathbf{x} = \mathbf{A}(\mathbf{x})\,\mathrm{d}t + \mathbf{C}(\mathbf{x})\,\mathrm{d}\mathbf{w}^{(0)}, \tag{46}$$

$$\mathrm{d}\mathbf{x} = \mathbf{F}(\mathbf{x},\mathbf{u})\,\mathrm{d}t + \mathbf{C}(\mathbf{x})\,\mathrm{d}\mathbf{w}^{(1)}, \tag{47}$$

where $\mathbf{x}\in\mathbb{R}^n$ denotes the state of the system, $\mathbf{u}\in\mathbb{R}^m$ denotes the control input, $\mathbf{C}(\mathbf{x})\in\mathbb{R}^{n\times m}$ is the diffusion matrix, $\mathbf{F}(\mathbf{x},\mathbf{u})\in\mathbb{R}^n$ is the drift dynamics, and $\mathbf{w}^{(0),(1)}\in\mathbb{R}^m$ are

Wiener processes (Brownian motion). The upper-scripts (0) and (1) are used to distinguish the two noise processes in the uncontrolled and controlled dynamics, respectively. The drift term $\mathbf{A}(\mathbf{x}) \in \mathbb{R}^n$ is defined by $\mathbf{A}(\mathbf{x}) = \mathbf{F}(\mathbf{x}, 0)$. The diffusion matrix may be partitioned as $\mathbf{C}(\mathbf{x}) = \begin{bmatrix} \mathbf{0} & \mathbf{C}_c^\intercal(\mathbf{x}) \end{bmatrix}^\intercal$ where $\mathbf{0} \in \mathbb{R}^{(n-m)\times m}$ and $\mathbf{C}_c(\mathbf{x}) \in \mathbb{R}^{m\times m}$ is invertible. Similarly, the drift term in the controlled dynamics may be partitioned as $\mathbf{F}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \mathbf{F}_1^\intercal(\mathbf{x}, \mathbf{u}) & \mathbf{F}_2^\intercal(\mathbf{x}, \mathbf{u}) \end{bmatrix}^\intercal$ where $\mathbf{F}_1(\mathbf{x}, \mathbf{u}) \in \mathbb{R}^{(n-m)}$ and $\mathbf{F}_2(\mathbf{x}, \mathbf{u}) \in \mathbb{R}^m$; and the drift term in the uncontrolled dynamics may be partitioned as $\mathbf{A}(\mathbf{x}) = \begin{bmatrix} \mathbf{A}_1^\intercal(\mathbf{x}) & \mathbf{A}_2^\intercal(\mathbf{x}) \end{bmatrix}^\intercal$ where $\mathbf{A}_1(\mathbf{x}) \in \mathbb{R}^{(n-m)}$ and $\mathbf{A}_2(\mathbf{x}) \in \mathbb{R}^m$. The class of systems whose matrices can be partitioned as such contains rigid body, and multi body dynamics as well as kinematic models such as the ones considered in this work. Henceforth, for simplicity, we will assume that $m = n$. The case when $m < n$ can be treated similarly; see for instance [122]. Let $\mathbf{\Sigma}(\mathbf{x}) = \mathbf{C}(\mathbf{x})\mathbf{C}^\intercal(\mathbf{x}) \in \mathbb{R}^{m\times m}$ and also define the following quantity:

$$\delta\mathbf{F}(\mathbf{x}, \mathbf{u}) = \mathbf{F}(\mathbf{x}, \mathbf{u}) - \mathbf{A}(\mathbf{x}) = \mathbf{F}(\mathbf{x}, \mathbf{u}) - \mathbf{F}(\mathbf{x}, 0), \quad \forall \mathbf{x}, \mathbf{u}.$$

To the system (47) we also associated the state cost

$$\mathcal{J}(\mathbf{x}(\cdot), t) = \Phi(\mathbf{x}(t_\mathrm{f})) + \int_t^{t_\mathrm{f}} q(\mathbf{x}(\tau), \tau)\, \mathrm{d}\tau. \tag{48}$$

where the function $q : (x, t) \mapsto r$ returns a non-negative real number r for a given state x and time t. With a slight abuse of notation we will also use $\mathcal{J}(\mathbf{x}, t)$ to denote the value of $\mathcal{J}(\mathbf{x}(\cdot), t)$ along the trajectory $\mathbf{x}(\cdot)$ starting from $\mathbf{x} = \mathbf{x}(t)$ at time t. Expectations evaluated on trajectories generated by the uncontrolled dynamics and controlled dynamics will be represented by $\mathbb{E}_\mathsf{p}[\,\cdot\,]$ and $\mathbb{E}_\mathsf{q}[\,\cdot\,]$, respectively. The following fact can be found in [122].

Proposition 3

Given the measures p, q induced by the trajectories of (46) and (47), respectively, the *Radon-Nikodym derivative* of q with respect to p is defined by

$$\begin{aligned} \frac{\mathrm{d}\mathsf{q}}{\mathrm{d}\mathsf{p}} = \exp\left(\int_t^{t_\mathrm{f}} \delta\mathbf{F}^\intercal(\mathbf{x}(\tau), \mathbf{u}(\tau))\mathbf{C}^{-1}(\mathbf{x}(\tau))\, \mathrm{d}\mathbf{w}^{(1)}(\tau)\right) + \\ \exp\left(\int_t^{t_\mathrm{f}} \tfrac{1}{2}\delta\mathbf{F}^\intercal(\mathbf{x}(\tau), \mathbf{u}(\tau))\mathbf{\Sigma}^{-1}(\mathbf{x}(\tau))\, \delta\mathbf{F}(\mathbf{x}(\tau), \mathbf{u}(\tau))\, \mathrm{d}\tau\right). \end{aligned} \tag{49}$$

Given equation (49), the relative entropy term in (42) takes the form:

$$\frac{1}{|\rho|}\mathbb{KL}(\mathsf{q}\|\mathsf{p}) = \mathbb{E}_{\mathsf{q}}\left[\frac{1}{2|\rho|}\int_t^{t_\mathrm{f}} \delta\mathbf{F}^\intercal(\mathbf{x}(\tau),\mathbf{u}(\tau))\boldsymbol{\Sigma}^{-1}(\mathbf{x}(\tau))\delta\mathbf{F}(\mathbf{x}(\tau),\mathbf{u}(\tau))\,\mathrm{d}\tau\right],$$

Substituting the previous expression of the Kullback-Leibler divergence into (42) one obtains

$$-\frac{1}{|\rho|}\mathcal{E}_{\mathsf{p}}\left(\mathcal{J}(\mathbf{x},t);\rho\right) \leq \mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)\right] + \mathbb{E}_{\mathsf{q}}\left[\frac{1}{2|\rho|}\int_t^{t_\mathrm{f}} \delta\mathbf{F}^\intercal(\mathbf{x}(\tau),\mathbf{u}(\tau))\boldsymbol{\Sigma}^{-1}(\mathbf{x}(\tau))\delta\mathbf{F}(\mathbf{x}(\tau),\mathbf{u}(\tau))\,\mathrm{d}\tau\right].$$

The previous equation can be written in the form (45) with state cost term defined as $\mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)\right]$ and information cost defined as $\mathbb{E}_{\mathsf{q}}\left[\frac{1}{2|\rho|}\int_t^{t_\mathrm{f}} \delta\mathbf{F}^\intercal(\mathbf{x}(\tau),\mathbf{u}(\tau))\boldsymbol{\Sigma}^{-1}(\mathbf{x}(\tau))\delta\mathbf{F}(\mathbf{x}(\tau),\mathbf{u}(\tau))\,\mathrm{d}\tau\right]$. Next, we further specialize the class of systems where (45) is applied to, and discuss its connections to stochastic optimal control as in [43, 118, 119]. To this end, let us consider the special case of (46) and (47) with uncontrolled and controlled stochastic dynamics of the following form, respectively:

$$\mathrm{d}\mathbf{x} = \mathbf{f}(\mathbf{x})\,\mathrm{d}t + \frac{1}{\sqrt{|\rho|}}\mathbf{B}(\mathbf{x})\,\mathrm{d}\mathbf{w}^{(0)}, \tag{50}$$

$$\mathrm{d}\mathbf{x} = \mathbf{f}(\mathbf{x})\,\mathrm{d}t + \mathbf{B}(\mathbf{x})\left(\mathbf{u}\,\mathrm{d}t + \frac{1}{\sqrt{|\rho|}}\,\mathrm{d}\mathbf{w}^{(1)}\right), \tag{51}$$

where $\mathbf{x} \in \mathbb{R}^n$ denotes the state of the system, $\mathbf{B}(\mathbf{x}) \in \mathbb{R}^{n\times m}$ is the control/diffusion matrix, $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^n$ is the passive dynamics, $\mathbf{u} \in \mathbb{R}^m$ is the control vector and $\mathbf{w}^{(0),(1)}$ are m-dimensional Wiener noise processes.

For the dynamics in (50) and (51) the form of the Radon-Nikodym derivative in (49) can be computed as follows. Noticing that $\delta\mathbf{F}(\mathbf{x},\mathbf{u}) = \mathbf{B}(\mathbf{x})\mathbf{u}$, $\mathbf{C}(\mathbf{x}) = \mathbf{B}(\mathbf{x})/\sqrt{|\rho|}$ and $\boldsymbol{\Sigma}(\mathbf{x}) = \mathbf{B}(\mathbf{x})\mathbf{B}^\intercal(\mathbf{x})/|\rho|$, and substituting these expressions in (49) yields

$$\frac{\mathrm{d}\mathsf{q}}{\mathrm{d}\mathsf{p}} = \exp\left(|\rho|\eta(\mathbf{u},t)\right) \quad \text{and} \quad \frac{\mathrm{d}\mathsf{p}}{\mathrm{d}\mathsf{q}} = \exp\left(-|\rho|\eta(\mathbf{u},t)\right), \tag{52}$$

where $\eta(\mathbf{u},t)$ is given by:

$$\eta(\mathbf{u},t)=\frac{1}{2}\int_t^{t_\mathrm{f}}\mathbf{u}^\intercal(\tau)\mathbf{u}(\tau)\,\mathrm{d}\tau+\frac{1}{\sqrt{|\rho|}}\int_t^{t_\mathrm{f}}\mathbf{u}^\intercal(\tau)\,\mathrm{d}\mathbf{w}^{(1)}(\tau). \tag{53}$$

Substitution of (52) and (53) into inequality (42) yields the following result:

$$-\frac{1}{|\rho|}\log\mathbb{E}_{\mathsf{p}}\left[\exp\left(-|\rho|\mathcal{J}(\mathbf{x},t)\right)\right]\leq\mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)+\eta(\mathbf{u},t)\right]. \tag{54}$$

Since the noise and the control terms are uncorrelated and the expectation of the noise is zero, the expectation on the right side of the inequality in (54) is further simplified as follows:

$$\underbrace{-\frac{1}{|\rho|}\log\mathbb{E}_{\mathsf{p}}\left[\exp\left(-|\rho|\mathcal{J}(\mathbf{x},t)\right)\right]}_{\xi(\mathbf{x},t)}\leq\underbrace{\mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},t)+\tfrac{1}{2}\int_t^{t_\mathrm{f}}\mathbf{u}(\tau)^\intercal\mathbf{u}(\tau)\,\mathrm{d}\tau\right]}_{\text{Total Cost}}. \tag{55}$$

The right-hand side term in the above inequality corresponds to the cost function of a stochastic optimal control problem that is bounded from below by the free energy. Surprisingly, inequality (55) was derived without relying on any principle of optimality. Inequality (55) essentially defines a minimization process in which the right-hand side part of the inequality is minimized with respect to $\eta(\mathbf{u},t)$ and therefore with respect to the corresponding control $\mathbf{u}$. At the minimum, when $\mathbf{u}=\mathbf{u}^*$, the right-hand side of inequality in (55) attains its optimal value $\xi(\mathbf{x},t)$. Under the optimal control $\mathbf{u}^*$, and according to (44), the corresponding optimal distribution takes the form

$$\mathrm{d}\mathsf{q}^*=\frac{\exp\Big(-|\rho|\Phi(\mathbf{x}(t_\mathrm{f}))\Big)\exp\left(-|\rho|\displaystyle\int_t^{t_\mathrm{f}}q(\mathbf{x}(\tau),\tau)\,\mathrm{d}\tau\right)}{\displaystyle\int\exp\Big(-|\rho|\Phi(\mathbf{x}(t_\mathrm{f}))\Big)\exp\left(-|\rho|\int_t^{t_\mathrm{f}}q(\mathbf{x}(\tau),\tau)\,\mathrm{d}\tau\right)\mathrm{d}\mathsf{p}}\,\mathrm{d}\mathsf{p}. \tag{56}$$

The work [118,119] inspired by early mathematical developments in control theory [43, 46], has shown that the value function $\xi(\mathbf{x},t)$ in (55) satisfies the Hamilton-Jacobi-Bellman equation and it has made the connection with more recent work in machine learning [63, 121] on Kullback-Leibler and path integral control.

7.4.2 Connection with Dynamic Programming (DP)

An important question that arises is: What is the link between (55) and the principle of optimality in dynamic programming? To address this question, we show that $\xi(\mathbf{x},t)$ satisfies the Hamilton-Jacobi-Bellman (HJB) equation associated with the optimal control problem (51)-(48) and hence, $\xi(\mathbf{x},t)$ is the corresponding value function of the following minimization problem

$$\begin{aligned}\xi(\mathbf{x},t) &= \min_{\substack{\mathbf{u}(\tau)\\ t\leq\tau\leq t_{\mathrm{f}}}} \mathbb{E}_{\mathsf{q}}\left[\Phi(\mathbf{x}(t_{\mathrm{f}})) + \int_t^{t_{\mathrm{f}}} \left(q(\mathbf{x}(\tau),\tau) + \tfrac{1}{2}\mathbf{u}^{\intercal}(\tau)\mathbf{u}(\tau)\right)\mathrm{d}\tau\right]\\ &= \min_{\substack{\mathbf{u}(\tau)\\ t\leq\tau\leq t_{\mathrm{f}}}} \mathbb{E}_{\mathsf{q}}\left[\mathcal{J}(\mathbf{x},\tau) + \tfrac{1}{2}\int_t^{t_{\mathrm{f}}} \mathbf{u}^{\intercal}(\tau)\mathbf{u}(\tau)\,\mathrm{d}\tau\right], \end{aligned} \tag{57}$$

where the expectation is computed over the trajectories of (51). To see this, we introduce $\Psi(\mathbf{x},t) \triangleq \mathbb{E}_{\mathsf{p}}\left[\exp\left(\rho\mathcal{J}(\mathbf{x},t)\right)\right]$ and apply the Feynman-Kac lemma [50] to arrive at the backward Chapman-Kolmogorov partial differential equation (PDE)

$$-\partial_t\Psi(\mathbf{x},t) = -|\rho|q(\mathbf{x},t)\Psi(\mathbf{x},t) + \mathbf{f}^{\intercal}(\mathbf{x})\nabla\Psi_{\mathbf{x}}(\mathbf{x},t) + \frac{1}{2|\rho|}\,\mathrm{tr}\left(\nabla\Psi_{\mathbf{xx}}(\mathbf{x},t)\mathbf{B}(\mathbf{x})\mathbf{B}(\mathbf{x})^{\intercal}\right) \tag{58}$$

with boundary condition $\Psi(\mathbf{x}(t_{\mathrm{f}}),t_{\mathrm{f}}) = \exp\big(-|\rho|\Phi(\mathbf{x}(t_{\mathrm{f}})\big)$, which governs the evolution of $\Psi(\mathbf{x},t)$ along the trajectories of (51) subject to $\mathbf{x}=\mathbf{x}(t)$. Since $\xi(\mathbf{x},t) = -\log\Psi(\mathbf{x},t)/|\rho|$, it follows that $\partial_t\Psi(\mathbf{x},t) = -|\rho|\Psi(\mathbf{x},t)\partial_t\xi(\mathbf{x},t)$, $\nabla\Psi_{\mathbf{x}}(\mathbf{x},t) = -|\rho|\Psi(\mathbf{x},t)\nabla\xi_{\mathbf{x}}(\mathbf{x},t)$ and $\nabla\Psi_{\mathbf{xx}}(\mathbf{x},t) = |\rho|\Psi(\mathbf{x},t)\nabla\xi_{\mathbf{xx}}(\mathbf{x},t) - |\rho|^2\Psi(\mathbf{x},t)\nabla\xi_{\mathbf{x}}(\mathbf{x},t)\nabla\xi_{\mathbf{x}}^{\intercal}(\mathbf{x},t)$. In this case, it can be shown that $\xi(\mathbf{x},t)$ satisfies the nonlinear PDE

$$\begin{aligned}-\partial_t\xi(\mathbf{x},t) = q(\mathbf{x},t) + \nabla\xi_{\mathbf{x}}^{\intercal}(\mathbf{x},t)\mathbf{f}(\mathbf{x}) - \tfrac{1}{2}\nabla\xi_{\mathbf{x}}^{\intercal}(\mathbf{x},t)\mathbf{B}(\mathbf{x})\mathbf{B}^{\intercal}(\mathbf{x})\nabla\xi_{\mathbf{x}}(\mathbf{x},t)\\ + \frac{1}{2|\rho|}\,\mathrm{tr}\left(\nabla\xi_{\mathbf{xx}}(\mathbf{x},t)\mathbf{B}(\mathbf{x})\mathbf{B}^{\intercal}(\mathbf{x})\right), \end{aligned} \tag{59}$$

subject to the boundary condition $\xi(\mathbf{x}(t_{\mathrm{f}}),t_{\mathrm{f}}) = \Phi(\mathbf{x}(t_{\mathrm{f}}))$. The nonlinear PDE (59) corresponds to the HJB equation associated with the optimal control problem (57) and hence $\xi(\mathbf{x},t)$ is the corresponding minimizing value function [111]. It is important to note, however, that the principle of optimality was not used to derive (59).

7.4.3 Path Integral Control with Initial Sampling Policies

According to (55), one need to sample trajectories under the uncontrolled dynamics and evaluate the left-hand side of (55) on these trajectories in order to find the value function $\xi(\mathbf{x}, t)$. However, in high-dimensional spaces, it is desirable to steer sampling toward specific areas of the state space. To do so, we have to incorporate an initial control policy into the uncontrolled dynamics. Therefore, instead of sampling from the uncontrolled dynamics (50), we sample trajectories based on the following stochastic dynamics:

$$\mathrm{d}\mathbf{x} = \mathbf{f}(\mathbf{x})\,\mathrm{d}t + \mathbf{B}(\mathbf{x})\left(\mathbf{u}_{\text{in}}\,\mathrm{d}t + \frac{1}{\sqrt{|\rho|}}\,\mathrm{d}\mathbf{w}^{(1)}\right), \tag{60}$$

where $\mathbf{u}_{\text{in}}$ is an initial control policy. In [118, 119], the authors derived an iterative PI control without relying on previous policy parameterizations. More precisely, when sampling trajectories from the dynamics (60) the work in [119] and [118] showed that the value function $\xi(\mathbf{x}, t)$ is expressed as

$$\xi(\mathbf{x}, t) = -\frac{1}{|\rho|}\log\left(\int \exp\left(-|\rho| S\left(\mathbf{x}, \mathbf{u}_{\text{in}}(\mathbf{x}, t), t\right)\right)\mathrm{d}\mathsf{q}_{\text{in}}\right)$$

where the term $S(\mathbf{x}, \mathbf{u}_{\text{in}})$ is defined as

$$\begin{aligned} S(\mathbf{x}, \mathbf{u}_{\text{in}}) = &\underbrace{\Phi(\mathbf{x}(\mathrm{t_f})) + \int_t^{t_\mathrm{f}} q(\mathbf{x}(\tau), \tau)\,\mathrm{d}\tau}_{\mathcal{J}(\mathbf{x}, t)} \\ &\underbrace{+ \frac{1}{2}\int_t^{t_\mathrm{f}} \mathbf{u}_{\text{in}}^{\intercal}(\tau)\mathbf{u}_{\text{in}}(\tau)\,\mathrm{d}\tau + \frac{1}{\sqrt{|\rho|}}\int_t^{t_\mathrm{f}} \mathbf{u}_{\text{in}}^{\intercal}(\tau)\,\mathrm{d}\mathbf{w}^{(1)}(\tau)}_{\eta(\mathbf{u}_{\text{in}}, t)}, \end{aligned} \tag{61}$$

where the term $\eta(\mathbf{u}_{\text{in}}, t)$ appears due to sampling based on the dynamics (60), while the term $\mathcal{J}(\mathbf{x}, t)$ is the state-dependent part of the total cost function in (55). The path integral control is now expressed as [118]

$$\mathbf{u}_{\text{PI}}(\mathbf{x}, t)\,\mathrm{d}t = \mathbf{u}_{\text{in}}(\mathbf{x}, t)\,\mathrm{d}t + \delta\mathbf{u}(\mathbf{x}, t), \tag{62}$$

where the term $\delta\mathbf{u}(\mathbf{x}, t)$ is defined by

$$\delta\mathbf{u}(\mathbf{x}, t) = \frac{1}{\sqrt{|\rho|}}\mathbb{E}_{\mathsf{q}^*}\left[\mathrm{d}\mathbf{w}^{(1)}\right] = \frac{1}{\sqrt{|\rho|}}\int \mathrm{d}\mathbf{w}^{(1)}\,\mathrm{d}\mathsf{q}^*, \tag{63}$$

and where the expectation is taken under the optimal probability

$$\mathrm{d}\mathsf{q}^* = \frac{\exp\left(-|\rho| S(\mathbf{x}, \mathbf{u}_{\mathrm{in}})\right)}{\displaystyle\int \exp\left(-|\rho| S(\mathbf{x}, \mathbf{u}_{\mathrm{in}})\right)\, \mathrm{d}\mathsf{q}_{\mathrm{in}}} \mathrm{d}\mathsf{q}_{\mathrm{in}}. \tag{64}$$

During implementation, equation (63) is approximated as

$$\begin{aligned} \delta\mathbf{u}(\mathbf{x}, t) &= \frac{1}{\sqrt{|\rho|}} \sum_{k=1}^{\#\mathrm{traj}} p_k \mathrm{d}\mathbf{w}^{(1)}(\omega_k) \\ p_k &= \frac{\exp\left(-|\rho| S(\mathbf{x}_k, \mathbf{u}_{\mathrm{in}})\right)}{\sum_{\ell=1}^{\#\mathrm{traj}} \exp\left(-|\rho| S(\mathbf{x}_\ell, \mathbf{u}_{\mathrm{in}})\right)} \end{aligned} \tag{65}$$

The initial policy $\mathbf{u}_{\mathrm{in}}$ can be a suboptimal control law, a hand-tuned PD, PID control, or feedforward control. In this chapter, we consider a feedforward control given by the RRT algorithm as the initial control policy. The RRT algorithm is proven to be a simple, iterative algorithm that quickly searches complicated, high-dimensional spaces for feasible paths. It grows a space-filling tree by drawing a random sample from the search space and connecting the nearest point in the tree to the new random sample at each iteration. This helps the tree to grow its branches toward unexplored regions of the search space quickly, i.e., achieving Voronoi bias [39, 81, 83]. In this case, the RRT-based optimal path integral control takes the form

$$\mathbf{u}_{\mathrm{PI}}(\mathbf{x}, t)\, \mathrm{d}t = \mathbf{u}_{\mathrm{RRT}}(t)\, \mathrm{d}t + \delta\mathbf{u}(\mathbf{x}, t). \tag{66}$$

In the next section, we discuss how to use the RRT algorithm to compute the initial control policy $\mathbf{u}_{\mathrm{RRT}}$.

7.5 Trajectory Sampling via Sampling-based Algorithms

In high dimensional state spaces, sampling of useful trajectories from the unforced dynamics can be a tedious task. This issue can be addressed by first computing a "good enough" initial trajectory and then sampling local trajectories in the neighborhood of this trajectory. In the proposed approach, we use a probabilistic algorithm to find $\mathbf{u}_{\mathrm{in}}$ in (60) and compute an initial trajectory quickly. Probabilistic methods have proven to be very efficient for the

solution of motion planning problems with dynamic constraints in high dimensional search spaces. Among them, Rapidly-exploring Random Trees (RRTs) [39, 81, 83] are among the most popular for solving single query motion planning problems. The main body of the RRT algorithm is given in Algorithm 13.

In the proposed approach, we leverage the speed and exploration capabilities of the RRT algorithm to compute an initial policy quickly by modifying the RRT primitive procedures. The proposed algorithm PI-RRT uses the path-integral approach to compute optimal trajectories that incorporate the system uncertainty (i.e., the risk of collision with obstacles). Since both final time and final state are given, the search space is formed by adding an additional time dimension T to the state space $\mathcal{X}$. Our search space, goal set and free space are thus defined as $\mathcal{Z} = \mathcal{X} \times T$, $\mathcal{Z}_{\text{goal}} = \mathcal{X}_{\text{goal}} \times T_{\text{goal}}$, and $\mathcal{Z}_{\text{free}} = \mathcal{Z} \setminus \mathcal{Z}_{\text{goal}}$, respectively. The RRT algorithm is then run to find a trajectory starting from an initial point $z_{\text{init}} = (x_{\text{init}}, t_{\text{init}})$ to the goal set $\mathcal{Z}_{\text{goal}}$ while avoiding the obstacles in $\mathcal{X}$. The primitive procedures borrowed by the RRT algorithm are `Sample`, `Nearest`, `Steer` and `Extend`. In addition, given a trajectory $\boldsymbol{\sigma}$, the Boolean function `ObstacleFree`(σ) checks whether $\boldsymbol{\sigma}$ belongs to $\mathcal{Z}_{\text{free}}$ or not. It returns `True` if the trajectory is a subset of $\mathcal{Z}_{\text{free}}$, i.e., $\boldsymbol{\sigma} \subset \mathcal{Z}_{\text{free}}$, and `False` otherwise. Details for these procedures can be found in [65]. In addition, the PI-RRT algorithm uses the following procedures:

`Steer` : Given two points $\mathbf{z}_1$ and $\mathbf{z}_2$ in $\mathcal{Z}_{\text{free}}$, `Steer` extends $\mathbf{z}_1$ toward $\mathbf{z}_2$ by sampling trajectories from the unforced dynamics of the system. Specifically, the procedure samples a set of trajectories emanating from $\mathbf{z}_1$ and returns the closest end point of this set of trajectories to the point $\mathbf{z}_2$ with respect to a given distance function.

`Extend`: is a function that extends the nearest vertex of the graph $\mathcal{G}$ toward the randomly sampled point z_{rand}. Since time always flows in forward direction, we make sure that `Extend` computes valid connections, i.e., it returns false if the time value of z_{rand} is less than that of the nearest vertex in the graph. The `Extend` procedure of the RRT algorithm is shown in Algorithm 14.

`ExtractPath`: is a function that process on a tree data structure and extracts the information of the collision free trajectory starting from the current state to the goal set and the corresponding the control signal. The RRT Algorithm returns a tree $\mathcal{G}$ which contains the information of control inputs at each vertex of the tree at Line 5. Then, the `ExtractPath` procedure is subsequently called at the next line and it concatenates the control inputs by backtracking from the goal set toward the current state.

`Execute`: is a function that performs some initial portion of a given control signal on the system.

`MeasureState`: is a function that returns the current state of the system after it is executed with some control input. Since the system is subjected to noise, there is usually a difference between the real and the simulated state.

`ComputeVariation`: is a function that implements the path-integral control approach. After the RRT Algorithm computes a feasible trajectory and the corresponding control input, a fixed number of trajectories are locally sampled in the vicinity of this trajectory. Then, the correction term in control is computed as simply a weighted average of the noise profiles that create the local trajectories.

Algorithm 13: Body of the RRT Algorithm

```
1 RRT(z_init, Z_goal, Z)
2     V ← {z_init}; E ← ∅;
3     G ← (V, E);
4     for i = 1 to N do
5         z_rand ← Sample(i);
6         G ← Extend(G, z_rand);
7     return G
```

The body of the path-integral based RRT algorithm is shown in Algorithm 15. It runs in a receding horizon fashion, that is, it computes a "good enough" control input and executes the first portion of the control signal at each time step. The algorithm starts by initializing the current time and state with the initial values in Lines 2-3. The algorithm then computes an initial policy in Line 5 by using the RRT algorithm. The steering procedure in the RRT

Algorithm 14: `Extend` Procedure for RRT Algorithm

1 **Extend**($\mathcal{G}$,$\mathbf{z}$)
2 $(V, E) \leftarrow \mathcal{G}$;
3 $\mathbf{z}_{\text{nearest}} \leftarrow \texttt{Nearest}(\mathcal{G}, \mathbf{z})$;
4 $(\mathbf{z}_{\text{new}}, \boldsymbol{\sigma}_{\text{new}}, \mathbf{u}_{\text{new}}) \leftarrow \texttt{Steer}(\mathbf{z}_{\text{nearest}}, \mathbf{z})$;
5 **if** $\texttt{ObstacleFree}(\boldsymbol{\sigma}_{\text{new}})$ **then**
6 $V \leftarrow V \cup \{\mathbf{z}_{\text{new}}\}$;
7 $E \leftarrow E \cup \{(\mathbf{z}_{\text{nearest}}, \mathbf{z}_{\text{new}})\}$;
8 **return** $\mathcal{G}' \leftarrow (V, E)$

algorithm is slightly modified in order to sample dynamically feasible trajectories. The steering procedure first samples a fixed number of trajectories from the unforced dynamics and then chooses the one that has the closest terminal state toward the desired point. Once a trajectory that reaches the goal set has been computed, the corresponding trajectory $\boldsymbol{\sigma}_{\mathrm{RRT}}$, along with control the signal $\mathbf{u}_{\mathrm{RRT}}$, are extracted from the computed data structure in Line 6. Then, the algorithm proceeds by locally sampling trajectories around $(\boldsymbol{\sigma}_{\mathrm{RRT}}, \mathbf{u}_{\mathrm{RRT}})$ and computes the variation in the control $\delta\mathbf{u}(\mathbf{x}, t)$ according to (63) by using information of local trajectories. Since we have M number of local trajectories, the expectation in (63) is numerically approximated by using the expression in (65). For each local trajectory $\boldsymbol{\sigma}_k$, a cost value is computed as $S(\boldsymbol{\sigma}_k, \mathbf{u}_{\mathrm{RRT}})$ and its desirability value is computed by exponentiating the corresponding cost value, i.e., $d_k = \exp(-|\rho| S(\boldsymbol{\sigma}_k, \mathbf{u}_{\mathrm{RRT}}))$. Then, the variation term in control $\delta\mathbf{u}(\mathbf{x}, t)$ is computed by taking the weighted average of all noise profiles which create the local trajectories and the weight of each trajectory is computed as the normalized desirability value, i.e., $p_k = d_k / \Sigma_{\ell=1}^{N} d_\ell$. The iteration of the algorithm is completed by executing the first τ times of the computed control signal and the algorithm keeps repeating the same steps until the final time is reached.

7.6 Comparison with Existing Methods

In [92], the authors introduced Chance Constrained Rapidly-exploring Random Trees (CC-RRT) algorithm to solve motion planning problems involving uncertainty in the location

Algorithm 15: Body of the PI-RRT Algorithm

PI-RRT($\mathbf{z}_{\text{init}}$, $\mathcal{Z}_{\text{goal}}$, $\mathcal{Z}$**)**
  $(t_{\text{i}}, \mathbf{x}_{\text{i}}) \leftarrow \mathbf{z}_{\text{init}}$; $(t_{\text{f}}, \mathcal{X}_{\text{goal}}) \leftarrow \mathcal{Z}_{\text{goal}}$;
  $\mathbf{z}_{\text{i}} \leftarrow \mathbf{z}_{\text{init}}$;
  while $t_{\text{i}} < t_{\text{f}}$ **do**
    $\mathcal{G} \leftarrow \text{RRT}\,(\mathbf{z}_{\text{i}}, \mathcal{Z}_{\text{goal}}, \mathcal{Z})$;
    $(\boldsymbol{\sigma}_{\text{RRT}}, \mathbf{u}_{\text{RRT}}) \leftarrow \texttt{ExtractPath}(\mathcal{G})$;
    $\delta\mathbf{u}_{[t_{\text{i}}, t_{\text{f}}]} \leftarrow \texttt{ComputeVariation}(\mathbf{u}_{\text{RRT}}, M)$;
    $\mathbf{u} \leftarrow \mathbf{u}_{\text{RRT}} + \delta\mathbf{u}$;
    $\texttt{Execute}(\mathbf{u}_{[t_{\text{i}}, t_{\text{i}}+\tau]})$;
    $\mathbf{x}_{\text{i}} \leftarrow \texttt{MeasureState}(t_{\text{i}} + \tau)$; $t_{\text{i}} \leftarrow t_{\text{i}} + \tau$;
    $\mathbf{z}_{\text{i}} \leftarrow (\mathbf{x}_{\text{i}}, t_{\text{i}})$;

of the obstacles. Their approach is applicable for linear systems subject to process noise and/or uncertain obstacles which are assumed to be convex polyhedra. Due to the uncertainties in the problem, it may not be possible to identify a path guaranteed to be collision free surely. Therefore, the main idea in [92] is to relax this feasibility condition and introduce the notion of chance constraints, which guarantees probabilistic feasibility of computed trajectories. Under the assumption of Gaussian noise, probabilistic feasibility at each time step can be established through simple simulation of the state conditional mean and the evaluation of linear constraints. After some algebraic operations, these probabilistic inequality constraints are converted into deterministic ones that yield conservative bounds in lieu of the inequality constraints representing the obstacles. Although both our approach and the CC-RRT algorithm leverage the RRT algorithm, they differ from each other in several ways. The PI-RRT algorithm can be applied to nonlinear systems which is affine in control and it does not impose any condition on the shape of the obstacles. Also, since the PI-RRT algorithm tries to compute the desirability function, i.e., the value function under exponentiation, it provides guarantees of optimality, whereas the CC-RRT algorithm only ensures path feasibility and relies on random sampling of controls to compute good enough trajectories without any guarantees.

7.7 Numerical Simulations

In this section, we present a series of simulated experiments using a kinematic car model. We are interested in controlling a vehicle, whose motion is described by the following kinematic equations:

$$\dot{x} = v\cos\theta, \quad \dot{y} = v\sin\theta, \quad \dot{\theta} = w/r, \tag{67}$$

where x, y are the Cartesian coordinates of a reference point of the vehicle, v is its speed, w is the control input and r is a positive constant. We assume that the admissible control inputs, are restricted by $w \in [-1, 1]$. We would like to find an optimal policy for the heading rate w to move the vehicle from a given initial configuration $(x_{\mathrm{i}}, y_{\mathrm{i}}, \theta_{\mathrm{i}})^{\intercal}$ to a final configuration $(x_{\mathrm{f}}, y_{\mathrm{f}}, \theta_{\mathrm{f}})^{\intercal}$ within some fixed final time t_{f}.

Let $\mathbf{x}_1 = x$, $\mathbf{x}_2 = y$, $\mathbf{x}_3 = \theta$ be the states and $\mathbf{u} = w$ be the control input of the system. Then (67) can be rewritten as

$$\dot{\mathbf{x}}_1 = v\cos\mathbf{x}_3, \quad \dot{\mathbf{x}}_2 = v\sin\mathbf{x}_3, \quad \dot{\mathbf{x}}_3 = \mathbf{u}/r. \tag{68}$$

Assuming the system is subjected to noise of intensity α in the control channel, (68) can be written in the standard form

$$\begin{pmatrix} \mathrm{d}\mathbf{x}_1 \\ \mathrm{d}\mathbf{x}_2 \\ \mathrm{d}\mathbf{x}_3 \end{pmatrix} = \begin{pmatrix} v\cos\mathbf{x}_3 \\ v\sin\mathbf{x}_3 \\ 0 \end{pmatrix} \mathrm{d}t + \begin{pmatrix} 0 \\ 0 \\ 1/r \end{pmatrix} (\mathbf{u}\,\mathrm{d}t + \alpha\,\mathrm{d}\mathbf{w}), \tag{69}$$

where $\mathbf{f}$, $\mathbf{B}$ and ρ in (60) are defined as follows

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} v\cos\mathbf{x}_3 \\ v\sin\mathbf{x}_3 \\ 0 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 0 \\ 0 \\ 1/r \end{pmatrix}, \quad \rho = -\frac{1}{\alpha^2}.$$

The following parameters were used in the numerical simulations: $\mathbf{x}_0 = \begin{pmatrix} -9 & 0 & 0 \end{pmatrix}^{\intercal}$, $t_0 = 0$, $\mathbf{x}_{\mathrm{f}} = \begin{pmatrix} 9 & 0 & 0 \end{pmatrix}^{\intercal}$, $t_{\mathrm{f}} = 10$, $\mathrm{d}t = 0.1$, $v = 2.0$.

7.7.1 Example 1: Single-slit Obstacle

The objective in this problem is to find trajectories for the vehicle in a square environment with a box-like obstacle having a single slit. The trajectories computed by the PI-RRT algorithm at different stages are shown in Figure 28. The initial state is plotted as a yellow square and the goal region is shown in blue with magenta border (right-most). The computed path by the RRT algorithm following the unforced dynamics is shown in yellow. The locally sampled trajectories which are bundled around the yellow trajectory are shown in different colors. The trajectory of the vehicle due to execution of the control policy for some finite time horizon is shown in magenta.

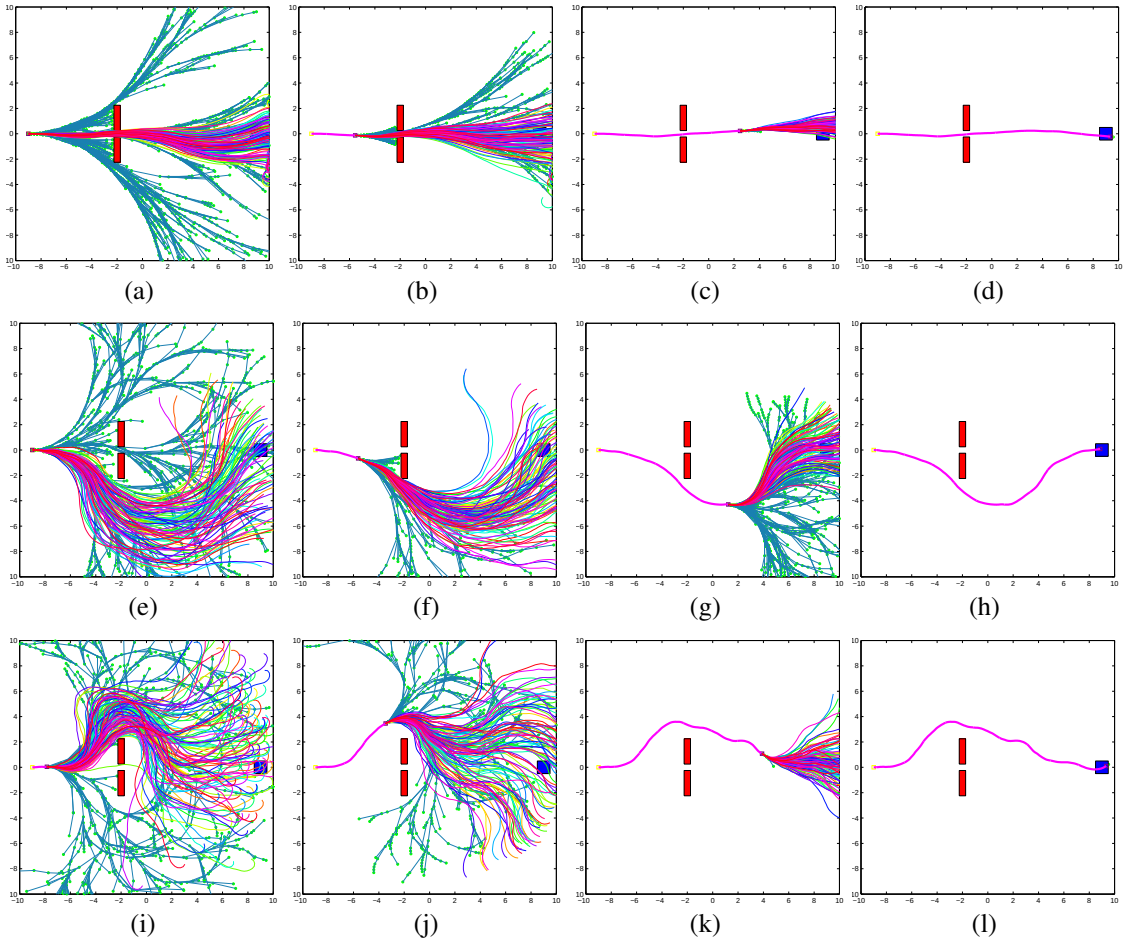


Figure 28: The trajectories computed by the PI-RRT algorithm for stochastic optimal control of the kinematic car model under different levels of noise injected to the control channel: (a)-(c) is with $\alpha = 0.25$, (e)-(g) is with $\alpha = 0.50$, and (i)-(k) is with $\alpha = 1.0$.

To understand how the intensity of the noise level affects the patterns of the trajectories of the system, we run the algorithm and analyzed the situation for three different cases, $\alpha = 0.25$, 0.5 and 1.0 corresponding to low, medium and high intensity noise levels in the control channel. As shown in Figure 28 (a)-(c), the PI-RRT algorithm computes trajectories that pass through the slit most of the time when there is low intensity noise in the control channel. As a first step, the PI-RRT algorithm computes a baseline trajectory using the RRT algorithm. The vertices and the edges of the tree computed by the RRT algorithm are shown in green and blue colors, respectively. During the simulations, it was observed that this baseline trajectory does not necessarily pass through the slit. The RRT algorithm sometimes returns a baseline trajectory that passes close by the upper or the lower sections of the obstacle due to both the noise which is observed in the dynamics and the randomized nature of the algorithm itself. The PI-RRT algorithm then samples a bundle of trajectories around the baseline trajectory in order to compute the variation term for the new control input. The new control input is computed by summing up the baseline control policy returned by the RRT algorithm and the variation term, which is the weighted average of the contribution of each locally sampled trajectory. These weights are computed by using the cost information of each locally sampled trajectory. We observed that the distribution of the trajectories, which pass close to the upper or lower corners or through the slit, changes as the intensity of the noise increases. For higher intensity of the noise, the PI-RRT algorithm computes trajectories which do not pass through the slit but rather pass close to the upper or lower corners. This change in the distribution of trajectories is shown in Figures 28 (e)-(g) for medium intensity noise and in Figures 28 (i)-(k) for high intensity noise.

7.7.2 Example 2: Double-slit Obstacle

Next, we consider a more challenging motion planning problem. In this case, there are two slits on the obstacle block and the length of the slits is longer than in the previous example. The longer length of the slits results in a higher probability of collision while traversing

through the slit, which makes the motion planning problem more challenging.

A study was performed in order to compare the performance of the PI-RRT algorithm with the RRT algorithm. No variation term in the control input was computed for the RRT algorithm, and it was simply executed in a receding horizon fashion. All algorithms were run for 6000 iterations to find a baseline trajectory. The results over 100 trials are shown in Figures 29, 30 and 31. The trajectories that result in collision are plotted in Figure 29 (a), (d) for the low noise level, Figure 30 (a), (d) for the medium noise level, and Figure 31 (a), (d) for the high noise level for the RRT and PI-RRT algorithms, respectively. Also, the distribution of collision-free trajectories is plotted in Figure 29 (c), (f) for the low noise level, Figure 30 (c), (f) for the medium noise level, and Figure 31 (c), (f) for the high noise level for the RRT and PI-RRT algorithms, respectively. The distribution of trajectories and the number of trajectories which result in a collision are summarized in Table I. Under the 'Success' column, the rows of the table contain the number of collision-free trajectories which pass through the bottom corner, bottom slit, top slit and top corner of the block. As shown in Table 3, the PI-RRT computes safer control policies which reduce the risk of having a collision. On the other hand, both the RRT and the PI-RRT compute trajectories that are almost equally distributed over both slits.

In summary, it was observed that the behaviors of both algorithms are similar for the case with high noise level. As the noise level decreases, most of the failed cases, not surprisingly, occur when the algorithms try to compute a path that passes through the slits. Our simulation results demonstrate that the PI-RRT algorithm tends to compute trajectories that have larger clearance from obstacles and hence outperforms the standard RRT algorithm, resulting in a smaller failure rate.

Table 3: Monte-Carlo Results for Double-Slit Obstacle

	$\alpha = 0.25$					$\alpha = 0.50$					$\alpha = 1.00$				
	Success				Fail	Success				Fail	Success				Fail
RRT	0	24	20	0	56	23	8	11	27	31	48	0	0	44	8
PI-RRT	0	44	45	0	11	35	9	8	37	11	47	0	0	49	4

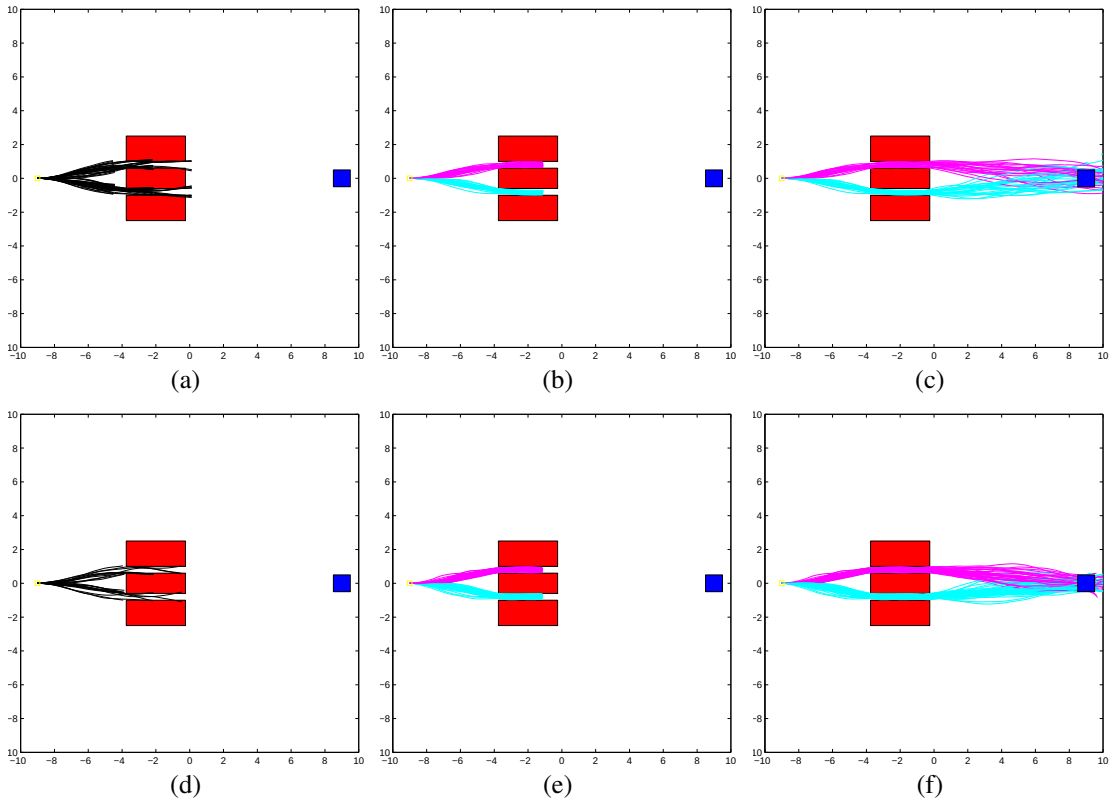


Figure 29: Distribution of trajectories for kinematic car model under low intensity of noise injected to the control channel ($\alpha = 0.25$) is shown in (a)-(c) for the RRT algorithm, and in (d)-(f) for the PI-RRT algorithm. The trajectories which hit the obstacles are shown in (a), (d). The collision-free trajectories at an intermediate stage are shown in (b), (e), and at the final stage are shown in (c), (f).

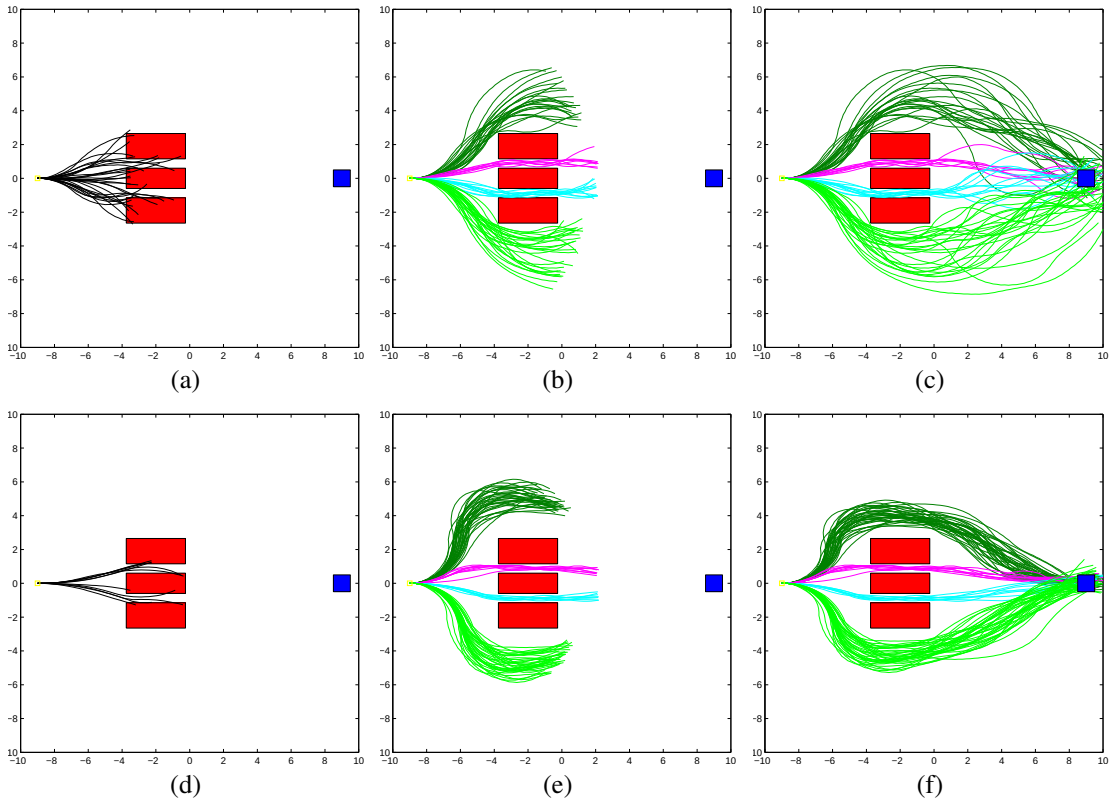


Figure 30: Distribution of trajectories for kinematic car model under medium intensity of noise injected to the control channel ($\alpha = 0.50$) is shown in (a)-(c) for the RRT algorithm, and in (d)-(f) for the PI-RRT algorithm. The trajectories which hit the obstacles are shown in (a), (d). The collision-free trajectories at an intermediate stage are shown in (b), (e), and at the final stage are shown in (c), (f).

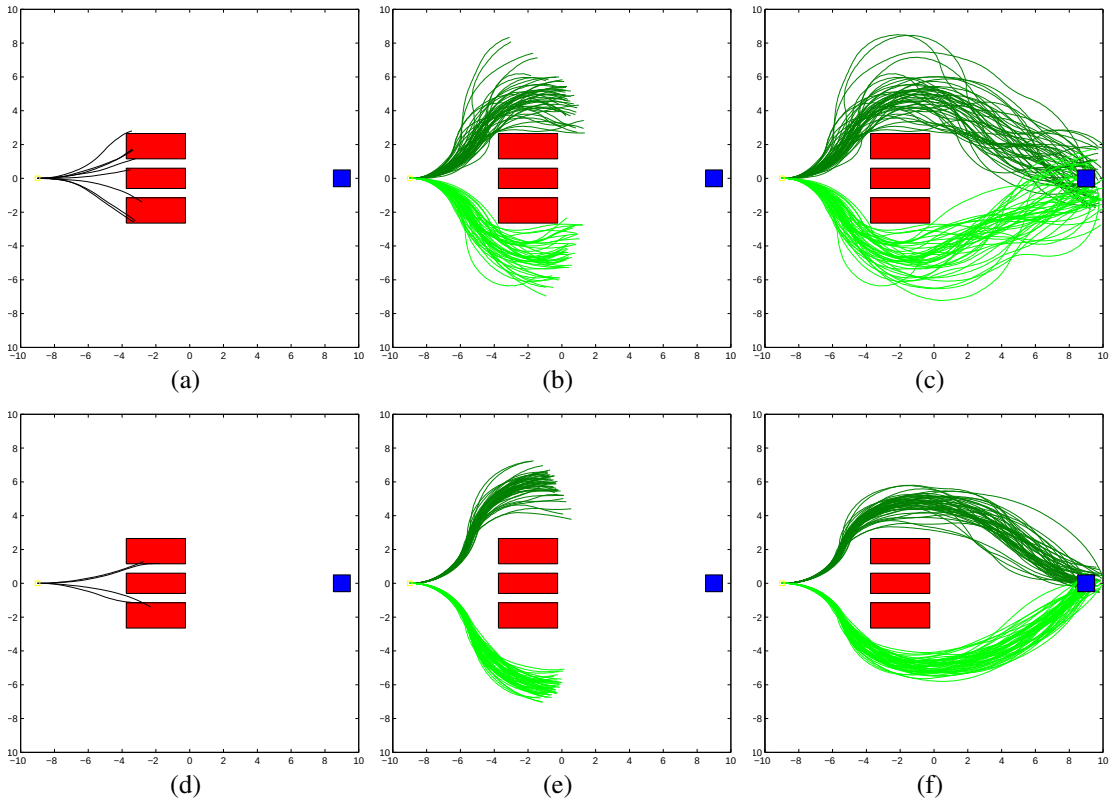


Figure 31: Distribution of trajectories for kinematic car model under high intensity of noise injected to the control channel ($\alpha = 1.0$) is shown in (a)-(c) for the RRT algorithm, and in (d)-(f) for the PI-RRT algorithm. The trajectories which hit the obstacles are shown in (a), (d). The collision-free trajectories at an intermediate stage are shown in (b), (e), and at the final stage are shown in (c), (f).

7.8 Conclusion

In this chapter, the PI-RRT algorithm is proposed in order to solve a class of stochastic optimal control problems. The proposed approach makes a novel connection between incremental sampling-based algorithms and path integral control. The PI-RRT algorithm benefits the nice properties of importance sampling and finds an efficient way to compute an expectation operation over the distribution of trajectories of the uncontrolled dynamics, i.e., the target distribution of trajectories. The expectation operation is essential in order to compute the optimal path integral control law. At each iteration, the initial control policy computed by the RRT algorithm is inputed to the system, and the expectation is performed as a weighted summation over the distribution of trajectories of the controlled dynamics under the initial control policy instead of the target distribution. It is observed that the proposed approach remedies the numerical instability of the original path integral control during the simulations.

Chapter VIII

MOTION PLANNING USING CLOSED-LOOP PREDICTION

8.1 Overview

Motion planning under differential constraints, the *kinodynamic motion planning* problem, is one of the canonical problems in robotics. Currently, state-of-the-art is provided by kinodynamic variants of popular sampling-based algorithms such as Rapidly-exploring Random Trees (RRTs). However, this problem remains challenging due to, for example, issues of how to include complex dynamics and guarantee optimality. Especially, if the open-loop dynamics are unstable, then exploration of high quality solutions by random sampling in control space becomes a very tedious task. Possible ways to remedy these issues are to search for a reference trajectory, which can be easily followed by using a stabilizing tracking controller, and to employ ideas from recently proposed asymptotically optimal sampling-based algorithms to optimize over the space of reference trajectories. In this chapter, we describe a new sampling-based algorithm, called CL-$\mathrm{RRT}^{\#}$, which leverages ideas from the $\mathrm{RRT}^{\#}$ algorithm and a variant of the RRT algorithm that generates trajectories using closed-loop prediction. The idea of planning with closed-loop prediction allows us to handle complex unstable dynamics. The search technique presented in the $\mathrm{RRT}^{\#}$ algorithm provides us to improve the solution quality by searching over alternative reference trajectories. Numerical simulations computed for a car model demonstrate the benefits of the proposed approach.

8.2 Introduction

Motion planning problems have been studied by both the robotics and the controls research communities for a long time, and many algorithms have been developed for their solution [78, 81]. These problems are ubiquitous in many applications, to achieve higher level of autonomy and challenging several reasons, for example, including state and control constraints, high-state dimensionality, complex differential constraints that arise from system dynamics, and solution quality. Loosely speaking, given a system that is subject to a set of differential constraints, an initial state, a final state, a set of obstacles, and a goal region, the *motion planning* problem is to find a control input that drives the system from its initial state to the goal region. This problem is computationally hard to solve and a basic version of the problem, called the piano mover's problem, was shown to be PSPACE-hard [104].

A common approach to solve the motion planning problems is to divide the problem into two or more subproblems. In the first subproblem, an obstacle-free geometric path is computed from the initial point to the goal region. Then, in the next subproblem, a controller is designed to track the computed geometric path with a suitable time parameterization while considering the differential and control constraints. The main drawback of this approach is lack of dynamic feasibility guarantee. Nonetheless, it has been successfully applied to robotic applications in which the underlying system has redundant control authority (e.g., robotic manipulators). Since it is usually nontrivial to capture the differential constraints at a geometric level to guide the search in the first subproblem, it is quite possible that the computed obstacle-free geometric path may not be tracked by the underlying control system or can be tracked at the expense of large control effort at best.

Another important class of algorithms is randomized planners that remedy the issue with dynamic feasibility guarantee by solving the motion planning problem in one step without resorting to any intermediate problems. Notably, the kinodynamic version of the RRT algorithm incrementally grows a tree of trajectories in the state space by sampling random control inputs and simulating the motion of the system with these random control

inputs over a time period [81–83]. Therefore, the trajectories that are generated by the RRT algorithm are dynamically feasible by construction. Recently, the RRT algorithm and its variants were successfully applied to many robotic system [72, 84] and different class of stochastic problems [7, 58]. Unlike the standard RRT algorithm, these variants were usually implemented to compute a solution quickly and improve it in the remaining time until the execution of the motion plan. The quality of the solution computed by the RRT algorithm is still unboundedly suboptimal despite these engineered implementations.

One possible reason of suboptimality of the RRT algorithm is that exploration via random selection of control inputs becomes very inefficient when the dynamics are complex and unstable. In practice, the underlying dynamical systems are desired to exhibit certain motion behaviors that yield high quality solution, but these behaviors may not be achieved easily by randomly chosen control inputs. To remedy this drawback, the authors in [76] proposed a new algorithm, called CL-RRT, that uses closed-loop prediction for trajectory generation. Instead of sampling in the space of controls, the proposed approach grows a tree in the space of reference trajectories and each path of the tree represents a reference trajectory that acts as an input to the closed-loop system. The desired behaviors of the system are prescribed as specifications for a tracking controller that is used to track a given reference trajectory. Each edge of the tree is associated with a segment of a reference trajectory and a state trajectory of the system that is computed by closed-loop prediction. Instead of using Euclidean distance function, some heuristics are used to guide the exploration of the tree. Finally, the CL-RRT algorithm returns a reference trajectory that yields the lowest-cost state trajectory of the system.

The poor suboptimality property of the RRT algorithm has led many researchers to develop different class of sampling-based algorithms with improved solution properties. To this end, the authors in [65] presented the first rigorous analyses regarding the optimality properties of the RRT algorithm. The RRT algorithm computes a solution that converges

to a non-optimal solution with probability one. Furthermore, they introduced a new algorithm with asymptotic optimality guarantee, called RRT^*, and it performs an additional rewiring step that improves the existing solution with constant asymptotic computational overhead. This important result has sparked interests in development of other sampling-based algorithms with optimality guarantees [9, 61]. Later, the same authors developed variants of the RRT^* algorithm to solve motion planning problems under differential constraints [64, 66] and showed that the proposed algorithms are asymptotically optimal when a steering procedure that satisfies certain conditions is provided. However, developing efficient steering procedures that solves point-to-point motion planning without considering obstacles is not trivial for many dynamical systems. Steering procedures are known only for certain dynamical systems, e.g., time-optimal motion planning problems for Dubins vehicle or double integrator.

In this chapter we develop a new asymptotically optimal motion planning algorithm, called CL-$\mathrm{RRT}^{\#}$, that leverages ideas from the CL-RRT [76] and the $\mathrm{RRT}^{\#}$ algorithms [9, 12, 13]. To handle differential constraints, instead of sampling directly in the control space, the proposed approach samples in the output space and incrementally grows a graph whose edges correspond to segments of reference trajectories. The algorithm also keeps another graph to store state trajectories of the closed-loop system when it is inputed with a certain path in the graph of reference trajectories. To improve the quality of solutions, the CL-RRT algorithm searches among alternative paths of the graph of reference trajectories similar to $\mathrm{RRT}^{\#}$. As search progresses, the proposed algorithm checks different reference trajectories and simulates the system model forward in time as needed. Finally, the algorithm chooses and returns the segments of reference trajectories that yield a lower-cost state trajectory of closed-loop system.

The chapter is organized as follows. Section 8.3 gives an overview of the problem formulation and notation. Section 8.4 is devoted to the discussion about the details of the proposed algorithm, the CL-$\mathrm{RRT}^{\#}$algorithm. Numerical results are presented for some

dynamical systems in Section 8.5 to demonstrate the nice features of the algorithm. Finally, the last section contains conclusion and future work.

8.3 Problem Formulation

8.3.1 Notation and Definitions

Let $X \subseteq \mathbb{R}^n$, $Y \subseteq \mathbb{R}^p$ and $U \subseteq \mathbb{R}^m$ be compact sets. We assume that the system dynamics can be described by a nonlinear differential equation of the form

$$\begin{aligned} \dot{x}(t) =& f(x(t), u(t)), \quad x(0) = x_0, \\ y(t) =& h(x(t), u(t)), \end{aligned} \tag{70}$$

where the system state $x(t) \in X$, the system output $y(t) \in Y$, the control $u(t) \in U$, for all $t, x_0 \in X$, and f and h are smooth (continuously differentiable) functions describing the time evolution of the system dynamics. Let $\mathcal{X}$ denote the set of all essentially bounded measurable functions mapped from $[0, T]$ to X for any $T \in \mathbb{R}_{>0}$ and define $\mathcal{Y}$ and $\mathcal{U}$ similarly. The functions in $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{U}$ are called *state trajectories*, *output trajectories*, and *controls*, respectively.

Let X_{obs} and X_{goal}, called the *obstacle space* and the *goal region*, be open subsets of X. Let X_{free}, also called the *free space*, denote the set defined as $X \setminus X_{\text{obs}}$.

The smooth function h describes the output y that we wish to control. Loosely speaking, we are particularly interested in the class of control problems in which we wish to track a time-varying reference trajectory $r(t)$, called the *trajectory-generation* problem. We assume that given a desired output value $y' \in Y$, and a current output value $y \in Y$ of the system, the control law $\phi : (y', y) \mapsto u \in U$ computes a control input such that the closed-loop simulation of the system yields a good tracking performance as time evolves.

8.3.2 Problem Statement

Given the state space X, obstacle region X_{obs}, goal region X_{goal}, and smooth functions f and h that describe the system dynamics, find a reference trajectory $r \in \mathcal{Y}$ with domain

$[0, T]$ for some $T \in \mathbb{R}_{>0}$ such that the unique corresponding state trajectory $x \in \mathcal{X}$, output trajectory $y \in \mathcal{Y}$, and control $u \in \mathcal{U}$ that are computed by closed-loop simulation,

- obeys the differential constraints,

$$\dot{x}(t) = f(x(t), u(t)) \quad x(0) = x_0,$$
$$y(t) = h(x(t), u(t)) \text{ for all } t \in [0, T],$$

- avoids the obstacles, i.e., $x(t) \in X_{\text{free}}$ for all $t \in [0, T]$,
- reaches the goal region, i.e., $x(T) \in X_{\text{goal}}$,
- and minimizes the cost functional $J(x, u, r) = \int_0^T g(x(t), u(t), r(t))\,\mathrm{d}t$

8.3.3 Primitive Procedures

Following are definition of the primitive procedures that are used by the CL-RRT$^{\#}$ algorithm (for details, see [65]).

Sampling Let $\texttt{Sample} : \omega \mapsto \{\texttt{Sample}_i(\omega)\}_{i \in \mathbb{N}_0} \subset Y_{\text{free}}$ be a map from Ω to the sequences of points in Y_{free}, such that the random variables $\texttt{Sample}_i$, $i \in \mathbb{N}_0$, are independent and identically distributed (i.i.d.).

Nearest Neighbor Node Given a graph $\mathcal{G}_y = (V_y, E_y)$, where $V_y \in Y$, a point $\mathbf{Y} \in Y$, the function $\texttt{Nearest} : (\mathcal{G}_y, \mathbf{Y}) \mapsto v_y \in V_y$ returns the node in V_y that is "closest" to $\mathbf{Y}$ in terms of a given distance function. The Euclidean distance is used in the work, and hence

$$\texttt{Nearest}(\mathcal{G}_y = (V_y, E_y), \mathbf{Y}) \coloneqq \operatorname*{argmin}_{v_y \in V_y} \|v_y.\mathbf{Y} - \mathbf{Y}\|.$$

Near Neighbor Nodes Given a graph $\mathcal{G}_y = (V_y, E_y)$, where $V_y \in Y$, a point $\mathbf{Y} \in Y$, and a positive real number $d \in \mathbb{R}_{>0}$, the function $\texttt{Nearest} : (\mathcal{G}_y, \mathbf{Y}, d) \mapsto v_y \in V_y' \subset V_y$ returns the nodes in V_y that are contained in a ball of radius d centered at $\mathbf{Y}$, i.e.,

$$\texttt{Near}(\mathcal{G}_y = (V_y, E_y, r), \mathbf{Y}) \coloneqq \big\{v_y \in V_y : \|v_y.\mathbf{Y} - \mathbf{Y}\| \leq d\big\}.$$

Steering Given two points $y_{\text{from}}, y_{\text{to}} \in Y$, the function $\texttt{Steer} : (y_{\text{from}}, y_{\text{to}}) \mapsto y'$ returns a point $y' \in Y$ such that y' is "closer" to y_{to} than y_{from} is. In this work, the point y' returned by the function `Steer` will be such that y' minimizes $\|y' - y_{\text{to}}\|$ while at the same time maintaining $\|y' - y_{\text{from}}\| \leq \eta$, for a predefined $\eta > 0$, i.e.,

$$\texttt{Steer}(y_{\text{from}}, y_{\text{to}}) \coloneqq \operatorname*{argmin}_{y' \in \mathcal{B}_\eta(y_{\text{from}})} \|y' - y_{\text{to}}\|.$$

Closed-loop Prediction Given a state $x \in X_{\text{free}}$, and an output trajectory $\sigma_y \in \mathcal{Y}$, the function $\texttt{Propagate} : (x, \sigma_y) \mapsto \sigma_x \in \mathcal{X}$ returns the state trajectory that is computed by simulating the system dynamics forward in time with the initial state x, and the reference trajectory σ_y.

Collision Test Given two points $y_{\text{from}}, y_{\text{to}} \in \mathcal{G}_y$, the Boolean function $\texttt{ObstacleFree}(y_{\text{from}}, y_{\text{to}})$ returns `True` if the line segment between y_{from} and y_{to} lies in Y_{free}, i.e., $y = (1-\theta)y_{\text{from}} + \theta y_{\text{to}} \in Y_{\text{free}}, \quad \theta \in [0, 1]$, and `False` otherwise.

Cost-to-come Values Given a graph $\mathcal{G}_y = (V_y, E_y)$, let $\mathtt{g}^*$ denote the optimal cost-to-come value of the node $v_y \in V_y$ that can be achieved in $\mathcal{G}_y$. Each node $v_y \in V_y$ is associated with two estimates of the optimal cost-to-come value (see [9, 71]). The g-value of v_y is the cost of the path to v_y from a given initial state $y_{\text{init}} \in Y_{\text{free}}$. The one step look-ahead g-value of v_y is denoted with $\bar{\mathtt{g}}$ and defined as follows:

$$v_y.\bar{\mathtt{g}} = \begin{cases} 0, & \text{if } v_y.\mathbf{Y} = y_{\text{init}}, \\ \min\limits_{e_y \in E_{y,\text{pred}}} \left(v_{y,\text{pred}}.\mathtt{g} + \texttt{Cost}(\sigma) \right), & \text{otherwise}, \end{cases}$$

where $E_{y,\text{pred}} = \texttt{incoming}(\mathcal{G}_y, v_y)$, $v_{y,\text{pred}} = e_y.\texttt{tail}$, and σ is the state trajectory that is computed via closed-loop prediction, i.e., the dynamical system is simulated forward in time with the initial state $v_{y,\text{pred}}.\mathtt{p}_\sigma.\texttt{back}()$ and the reference trajectory $e_y.\sigma$.

Heuristic Value Given a node $v_y \in V_y$, and an output goal region Y_{goal}, the function `ComputeHeuristic` : $(v_y, Y_{\text{goal}}) \mapsto r$ returns an estimate r of the optimal cost from v_y to Y_{goal}; it return zero if $v_y \in Y_{\text{goal}}$. Heuristic value of each node v_y is stored in $v_y.\mathtt{h}$ during its construction. A heuristic is called *admissible* if it never overestimates the actual cost of reaching Y_{goal}. In this chapter, we always assume that `ComputeHeuristic` is implemented as an admissible heuristic.

Queue Operations Nodes of the computed graphs are associated with some keys and priority queues are used to sort these nodes based on the precedence relation between keys. Following functions are implemented to maintain a given priority queue $\mathcal{Q}$:

- $\mathcal{Q}.\mathtt{top\_key}()$ returns the highest priority of all nodes in the priority queue $\mathcal{Q}$ with the smallest key value if the queue is not empty. If $\mathcal{Q}$ is empty, then $\mathcal{Q}.\mathtt{top\_key}()$ returns a key value of $k = [\infty; \infty]$.
- $\mathcal{Q}.\mathtt{pop}()$ deletes the node with the highest priority in the priority queue $\mathcal{Q}$ and returns a reference to the node.
- $\mathcal{Q}.\mathtt{update}(v_y, k)$ sets the key value of the node v_y to k and reorders the priority queue $\mathcal{Q}$.
- $\mathcal{Q}.\mathtt{insert}(v_y, k)$ inserts the node v_y into the priority queue $\mathcal{Q}$ with the key value k.
- $\mathcal{Q}.\mathtt{remove}(v_y)$ removes the node v_y from the priority queue $\mathcal{Q}$.

Initialization Given an initial point $x_{\text{init}} \in X$, a goal region in the output space $Y_{\text{goal}} \subset Y$, the function `Initialize` : $(x_{\text{init}}, Y_{\text{goal}}) \mapsto (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$ returns a graph $\mathcal{G}_y$ that has only node v_y whose output point is $v_y.\mathbf{Y} = \mathtt{OutputMap}(x_{\text{init}})$, a graph $\mathcal{G}_\sigma$ that has the only node v_σ whose trajectory is a single point $v_\sigma.\sigma = x_{\text{init}}$, and empty priority queues $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ that are used for ordering of nongoal and goal nodes which represent points in Y, respectively.

Exploration Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, where $\mathcal{G}_y$ and $\mathcal{G}_\sigma$ are graphs whose nodes represent points in Y and trajectories in $\mathcal{X}$, respectively, and $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ are priority queues that are used for ordering of nongoal and goal nodes that represent points in Y, a goal region in the output space $Y_{\text{goal}} \subset Y$, and a point $\mathbf{Y} \in Y$, the function $\texttt{Extend} : (\mathcal{S}, Y_{\text{goal}}, \mathbf{Y}) \mapsto \mathcal{S}' = (\mathcal{G}'_y, \mathcal{G}'_\sigma, \mathcal{Q}', \mathcal{Q}'_{\text{goal}})$ includes a new node, multiple edges to $\mathcal{G}_y$ and multiple nodes, edges to $\mathcal{G}_\sigma$, updates the priorities of nodes in $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ and returns an updated tuple $\mathcal{S}'$.

Exploitation Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, where $\mathcal{G}_y$ and $\mathcal{G}_\sigma$ are graphs whose nodes represent points in Y and trajectories in $\mathcal{X}$, respectively, and $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ are priority queues that are used for ordering of nongoal and goal nodes that represent points in Y, the function $\texttt{Replan} : \mathcal{S} \mapsto \mathcal{S}' = (\mathcal{G}'_y, \mathcal{G}'_\sigma, \mathcal{Q}', \mathcal{Q}'_{\text{goal}})$ rewires the parent node of the nodes in $\mathcal{G}_y$ based on their cost-to-come values, includes new nodes and edges in $\mathcal{G}_\sigma$ if necessary, i.e., propagating dynamics of the system for new sequence of reference trajectories, and returns an updated tuple $\mathcal{S}'$.

Construction of Solution Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, the function $\texttt{ConstrSolution} : \mathcal{S} \mapsto \mathcal{T}_x$ returns a tree whose edges and nodes represent simulated trajectories in $\mathcal{X}$ and the corresponding internal states of the nodes of $\mathcal{G}_y$. These trajectories are computed by propagating the dynamics with reference trajectories that are encoded in a tree of $\mathcal{G}_y$, which is formed by the edges between nodes of $\mathcal{G}_y$ and their parent nodes.

Graph Operations The following functions are used in the CL-RRT$^{\#}$ algorithm.

- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{succ}$: $(\mathcal{G}, v) \mapsto V' \subseteq V$ returns the nodes in V that are the heads of the edges emanating from v, that is,

$$\texttt{succ}(\mathcal{G}, v) := \{v' \in V : e.\texttt{tail} = v \text{ and } e.\texttt{head} = v',\ e \in E\}\,.$$

- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{pred} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the nodes in V that are the tails of the edges going into v, that is,

$$\texttt{pred}(\mathcal{G}, v) := \{v' \in V : e.\texttt{tail} = v' \text{ and } e.\texttt{head} = v,\ e \in E\}\,.$$

- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{outgoing} : (\mathcal{G}, v) \mapsto E' \subseteq E$ returns the edges in E whose tail is v, that is, $\texttt{outgoing}(\mathcal{G}, v) := \{e \in E : e.\texttt{tail} = v\}\,.$

- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{incoming} : (\mathcal{G}, v) \mapsto E' \subseteq E$ returns the edges in E whose head is v, that is, $\texttt{incoming}(\mathcal{G}, v) := \{e \in E : e.\texttt{head} = v\}\,.$

List Operations The following functions are used in the CL-$\mathrm{RRT}^{\#}$ algorithm.

- Given a list of nodes V_z, where its nodes represent points in Z, and a point $z \in Z$, the function $\texttt{find} : (V_z, z) \mapsto v_z \in V_z$ returns the node in V_z that satisfies $v_z.z = z$ if there exists any such node, null reference otherwise.

- Given a list of nodes V_z, where its nodes represent points in Z, the function `back` returns a reference to the last node in the list if it is not empty, and null reference otherwise.

- Given a list of nodes V_z, where its nodes represent points in Z, the function `front` returns a reference to the first node in the list if it is not empty, and null reference otherwise.

8.4 The CL-RRT$^{\#}$ *Algorithm*

8.4.1 Details of Data Structures

Each node v_y in the graph $\mathcal{G}_y$ is an `OutNode` data structure, the fields of that are summarized in Table 4. Each node v_y is associated with a reference point $y \in \mathbb{R}^m$. It contains two estimates of the optimal cost-to-come value between the initial reference point and y, namely, cost-to-come value $\mathtt{g}$ and one step look-ahead g-value $\bar{\mathtt{g}}$. It also keeps a heuristic value $\mathtt{h}$ which is an underestimate of the optimal cost value between y and Y_{goal} in order to guide and reduce the search effort. Whenever $\bar{\mathtt{g}}$ is updated during the replanning procedure, the reference node that yields the corresponding minimum cost-to-come value is stored in the parent reference node $\mathtt{p}_y$. Lastly, the trajectory which is computed by closed-loop prediction when the system is simulated with the reference trajectory between the nodes $\mathtt{p}_y$ and v_y is stored in the parent trajectory node $\mathtt{p}_\sigma$ and its terminal state represents the internal state associated with v_y.

Table 4: The node data structure for points in output space (`OutNode`)

field	type	description
y	vector $\in \mathbb{R}^p$	output point associated with this node
$\mathtt{g}$	real $\in \mathbb{R}$	cost-to-come value
$\bar{\mathtt{g}}$	real $\in \mathbb{R}$	one step look-ahead g-value
$\mathtt{h}$	real $\in \mathbb{R}$	heuristic value for the cost between y and $\mathcal{Y}_{\text{goal}}$
$\mathtt{p}_y$	`OutNode`	reference to the parent output node
$\mathtt{p}_\sigma$	`TrajNode`	reference to the parent trajectory node

Each edge e_y in the graph $\mathcal{G}_y$ is a `OutEdge` data structure, the fields of which are summarized in Table 5. Each edge e_y is associated with a trajectory $r \in \mathcal{Y}$. It also contains two output nodes, namely, `tail` and `head`, which represent the tail and the head output nodes of e_y, respectively.

Each node v_σ in the graph $\mathcal{G}_\sigma$ is a `TrajNode` data structure, the fields of which are

Table 5: The edge data structure for trajectories in output space (`OutEdge`)

field	type	description
r	trajectory $\in \mathcal{Y}$	output trajectory associated with this edge
`tail`	`OutNode`	reference to the tail output node
`head`	`OutNode`	reference to the head output node

summarized in Table 6. Each node v_σ is associated with a trajectory $\sigma \in \mathcal{X}$. It contains an output edge e_y which corresponds to the reference trajectory that yields σ as the closed-loop prediction. It also keeps a list of outgoing output edges `outgoing`, and this list is used to compute outgoing trajectory nodes emanating from the terminal state of σ.

Table 6: The node data structure for trajectories in state space (`TrajNode`)

field	type	description
σ	trajectory $\in \mathcal{X}$	state trajectory associated with this node
e_y	`OutEdge`	reference to the output edge
`outgoing`	`OutEdge` array	list of outgoing output edges

Each edge e_σ in the graph $\mathcal{G}_\sigma$ is a `TrajEdge` data structure, the fields of which are summarized in Table 7. Each edge e_σ is associated with a trajectory $\sigma \in \mathcal{X}$. It contains two trajectory nodes, namely, `tail` and `head` which represent the tail and the head trajectory nodes of e_σ, respectively.

Table 7: The edge data structure for trajectories in state space (`TrajEdge`)

field	type	description
σ	trajectory $\in \mathcal{X}$	state trajectory associated with this edge
`tail`	`TrajNode`	reference to the tail trajectory node
`head`	`TrajNode`	reference to the head trajectory node

8.4.2 Details of the Procedures

The body of the CL-RRT$^{\#}$ algorithm is given in Algorithm 16. First, the algorithm initializes the tuple of data structures $\mathcal{S}$ that is incrementally grown and updated as exploration and exploitation are performed (Line 3). The tuple $\mathcal{S}$ contains the graphs $\mathcal{G}_y$ and $\mathcal{G}_\sigma$, which are used to store output nodes and state trajectory nodes, respectively, and the priority queues $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$. The details of the `Initialize` procedure are given in Algorithm 17. The graph $\mathcal{G}_\sigma$ is created with no edges and v_σ as its only node. This node represents a state trajectory that contains only the initial state x_{init}. Then, likewise, the graph $\mathcal{G}_y$ is initialized with no edges and v_y as its only node that represents y_{init}. The g- and $\bar{g}$-values of v_y are set with zero cost value. The parent trajectory node of v_y is set with the reference to the node v_σ.

Algorithm 16: The CL-RRT$^{\#}$ Algorithm

```
1 RRT#(x_init, X_goal, X)
2     Y_goal := OutputMap(X_goal);
3     S ← Initialize(x_init, Y_goal);
4     for k = 1 to N do
5         y_rand ← Sample(k);
6         S ← Extend(S, Y_goal, y_rand);
7         S ← Replan(S);
8     T_x ← ConstrSolution(S);
9     return T_x;
```

The algorithm iteratively builds a graph of collision-free reference trajectories $\mathcal{G}_y$ by first sampling an output point y_{rand} from the obstacle-free output space Y_{free} (Line 5) and then extending the graph towards this sample (Line 6), at each iteration. The cost of the unique trajectory from the root node to a given node v_y is denoted as $\texttt{Cost}(v_y)$. It also builds another graph $\mathcal{G}_\sigma$ to store the state trajectories computed by simulation of the closed-loop dynamics when a reference trajectory is tracked. Once a new node is added to $\mathcal{G}_y$ after the `Extend` procedure, the `Replan` procedure is called to improve the existing solution by propagating the new information (Line 7). The dynamic system is simulated for different reference trajectories as needed during the search process, and the computed state

trajectories are added to the graph $\mathcal{G}_\sigma$ as new nodes along with the corresponding controls information.

Algorithm 17: The `Initialize` Procedure

1 **Initialize**($x_{\text{init}}, Y_{\text{goal}}$)
2 $\sigma \leftarrow \{x_{\text{init}}\}$;
3 $v_\sigma \leftarrow$ `TrajNode(`σ,$\varnothing$,$\varnothing$`)`;
4 $y_{\text{init}} \leftarrow$ `OutputMap(`x_{init}`)`;
5 $v_y \leftarrow$ `OutNode(`y_{init}`)`;
6 $v_y.\mathtt{g} \leftarrow 0$; $v_y.\bar{\mathtt{g}} \leftarrow 0$;
7 $v_y.\mathtt{h} \leftarrow$ `ComputeHeuristic(`y_{init},Y_{goal}`)`;
8 $v_y.\mathtt{p}_\sigma \leftarrow v_\sigma$;
9 $V_y \leftarrow \{v_y\}$; $E_y \leftarrow \varnothing$;
10 $V_\sigma \leftarrow \{v_\sigma\}$; $E_\sigma \leftarrow \varnothing$;
11 $\mathcal{G}_y \leftarrow (V_y, E_y)$; $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$;
12 $\mathcal{Q} \leftarrow \varnothing$; $\mathcal{Q}_{\text{goal}} \leftarrow \varnothing$;
13 **return** $\mathcal{S} \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$;

Finally, when a predetermined maximum number of iterations is reached, the `ConstrSolution` procedure is called to extract the spanning tree of $\mathcal{G}_y$ that contains the lowest-cost reference trajectories (Line 8) and this tree is returned. The details of the `ConstrSolution` procedure is given Algorithm 18.

Algorithm 18: The `ConstrSolution` Solution Procedure

1 **ConstrSolution**($\mathcal{S}$)
2 $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}}) \leftarrow \mathcal{S}$;
3 $(V_y, E_y) \leftarrow \mathcal{G}_y$; $X \leftarrow \varnothing$;
4 **foreach** $v_y \in V_y$ **do**
5 $\sigma \leftarrow v_y.\mathtt{p}_\sigma.\sigma$;
6 $v_x \leftarrow$ `StateNode(`σ`.back())`;
7 $V_x \leftarrow V_x \cup \{v_x\}$;
8 $v_{x,\text{parent}} \leftarrow$ `find(`V_x,σ`.front())`;
9 **if** $v_{x,\text{parent}} = \varnothing$ **then**
10 $v_{x,\text{parent}} \leftarrow$ `StateNode(`σ`.front())`;
11 $V_x \leftarrow V_x \cup \{v_{x,\text{parent}}\}$;
12 $e_x \leftarrow$ `StateEdge(`$v_{x,\text{parent}}$, v_x, σ`)`;
13 $E_x \leftarrow E_x \cup \{e_x\}$;
14 $X \leftarrow X \cup \{\sigma$`.back()`$\}$;
15 **return** $\mathcal{T}_x = (V_x, E_x)$;

The `Extend` procedure is given in Algorithm 19. The procedure first extends the nearest output node $v_{y,\text{nearest}}$ to the output sample $\mathbf{Y}$ (Lines 4-5). The output trajectory that extends

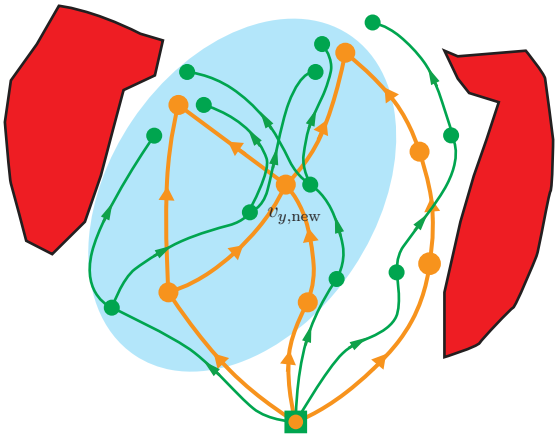


Figure 32: Extension of the graphs computed by the CL-RRT$^{\#}$ algorithm. Trajectories in the output and state spaces are shown in orange and green colors, respectively. Whenever a new node in the output space is added, then several incoming and outgoing edges are included to the graph in the vicinity of the new node, i.e., region colored with cyan.

the nearest output node $v_{y,\text{nearest}}$ towards the output sample $\mathbf{Y}$ is denoted as r_{new}. The final output point on the output trajectory r_{new} is denoted as y_{new}. If r_{new} is collision-free, then a new output node $v_{y,\text{new}}$ is created to represent the new output point y_{new} (Line 8), and the following changes in the vicinity of $v_{y,\text{new}}$ on both graphs are shown in Figure 32. The initial node is shown as a square box, the obstacles are shown in red color, and the graphs $\mathcal{G}_y$ and $\mathcal{G}_\sigma$ are shown in orange and green colors, respectively in Figure 32. The members of the node $v_{y,\text{new}}$ are set as follows. First, the `Near` procedure is called to find the set of neighbor output nodes V_{near} in the neighborhood of the new output point y_{new} (Line 9). Then, the set of incoming edges $E_{y,\text{pred}}$ and outgoing edges $E_{y,\text{succ}}$ of the new output node $v_{y,\text{new}}$ are computed by using the information of the neighbor output nodes (Lines 10-19). Once the new output node $v_{y,\text{new}}$ is created together with the set of incoming edges $E_{y,\text{pred}}$ and outgoing edges $E_{y,\text{succ}}$ connecting it to its neighbor output nodes V_{near}, the `Extend` procedure attempts to find the best incoming edge that yields a segment of a reference trajectory which incurs minimum cost to get to $v_{y,\text{new}}$ among all incoming edges in $E_{y,\text{pred}}$ (Lines 20-34). That is, for any incoming edge e_y in $E_{y,\text{pred}}$, the algorithm first gets the information of the predecessor output node $v_{y,\text{pred}}$ and its internal state x_{pred} by using the

information of the parent state trajectory node $v_{\sigma,\text{pred}}$ (Lines 22-24). Then, the algorithm simulates the system dynamics forward in time with the state x_{pred} being the initial state and $e_y.r$ being the reference trajectory to be tracked, (Line 25). If the state trajectory σ computed by closed-loop prediction is collision-free, a new trajectory node $v_{\sigma,\text{new}}$ is created together with its list of outgoing output trajectories being initialized with $E_{y,\text{succ}}$ (Line 27). Whenever a new trajectory node $v_{\sigma,\text{new}}$ is created, the outgoing state trajectories emanating from the final state of the state trajectory $v_{\sigma,\text{new}}.\sigma$ via closed-loop prediction are not immediately computed for the sake of efficiency. Instead, the algorithm keeps the set of candidate outgoing output trajectories, i.e., the edges in $E_{y,\text{succ}}$, in a list $v_{\sigma,\text{new}}.\texttt{outgoing}$, and the simulation of the system dynamics for these output trajectories is postponed until the head output node of the output edge $v_{\sigma,\text{new}}.e_y$ is selected for the Bellman update during the `Replan` procedure. Once the new state trajectory node $v_{\sigma,\text{new}}$ and the edge between the predecessor state trajectory node $v_{\sigma,\text{pred}}$ and itself are created (Lines 27-28), they are added to the set of nodes and edges of the graph $\mathcal{G}_\sigma$, respectively (Lines 29-30). If the incoming output edge e_y between the predecessor output node $v_{y,\text{pred}}$ and the new output node $v_{y,\text{new}}$ yields a collision-free state trajectory σ that incurs cost less than the current cost of $v_{y,\text{new}}$, then, the $\bar{\text{g}}$-value of $v_{y,\text{new}}$ is set with new lower cost, $v_{y,\text{pred}}$ and $v_{\sigma,\text{new}}$ are made the new parent output node and the new parent state trajectory node of $v_{y,\text{new}}$, respectively (Lines 31-34). After successful creation of the new output node $v_{y,\text{new}}$, it is added to the graph $\mathcal{G}_y$ together with all of its collision-free output edges (Line 36). Likewise, all trajectory nodes and edges created during the simulation of the system dynamics are added to the graph $\mathcal{G}_\sigma$ (Line 37). Lastly, the priority queues $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ are updated accordingly by using the information of the new output node $v_{y,\text{new}}$, i.e., reordering of the priorities after insertion of $v_{y,\text{new}}$ to the queue $\mathcal{Q}$ and reordering the goal output nodes in $\mathcal{Q}_{\text{goal}}$ if $v_{y,\text{new}}$ happens to be a goal output node (Lines 38-39).

The `Replan` procedure is given in Algorithm 20 (see [9]). The algorithm improves cost-to-come values of output nodes by operating on the nonstationary and promising nodes of

the graph $\mathcal{G}_y$. It simply pops the most promising nonstationary node from the priority queue $\mathcal{Q}$, if there are any, and this nonstationary node is made stationary by assigning its $\bar{\mathrm{g}}$-value to its g-value (Lines 5-6). Then, the g-value of the output node v_y is used to improve the $\bar{\mathrm{g}}$-values of its neighbor output nodes. Before this operation, the algorithm computes the set of all outgoing state trajectories emanating from internal state of the output node $\mathbf{V}$ (Lines 9-16). To do so, the algorithm first gets the information of the internal state x by using the parent state trajectory node of v_y (Lines 7-8). For any outgoing edge e_y in $v_\sigma.\texttt{outgoing}$, the algorithm first gets the information of the successor output node $v_{y,\mathrm{succ}}$ by using the output edge e_y (Line 10). Then, the algorithms simulates the system dynamics forward in time with the state x being the initial state and $e_y.r$ being the reference trajectory to be tracked (Line 11). If the state trajectory σ computed by closed-loop prediction is collision-free, a new trajectory node $v_{\sigma,\mathrm{succ}}$ is created together with its list of outgoing output trajectories being initialized with the set of outgoing output edges of $v_{y,\mathrm{succ}}$ (Line 13). Also, a state trajectory edge between v_σ and $v_{\sigma,\mathrm{succ}}$ is created (Line 14). Then, the new state trajectory node and edge are tentatively added to the set of nodes and edges of the graph $\mathcal{G}_\sigma$ (Lines 15-16). This step continues until all candidate outgoing output trajectories are processed in the closed-loop simulation, then the list $v_y.\texttt{outgoing}$ is cleared up (Line 17). All newly computed state trajectory nodes and edges are added to the graph $\mathcal{G}_\sigma$ (Line 18). For each outgoing state trajectory σ, the algorithm adds up its cost incurred by reaching to the successor output node $v_{y,\mathrm{succ}}$ to the g-value of v_y and compare this new cost value with the current $\bar{\mathrm{g}}$-value of $v_{y,\mathrm{succ}}$ (Line 22). If the outgoing state trajectory edge σ yields cost less than the current cost of $v_{y,\mathrm{succ}}$, then, the $\bar{\mathrm{g}}$-value of $v_{y,\mathrm{succ}}$ is set with new lower cost, v_y and $v_{\sigma,\mathrm{succ}}$ are made the new parent output node and the new parent state trajectory node of $v_{y,\mathrm{succ}}$, respectively (Lines 23-25). Lastly, the priority queues $\mathcal{Q}$ and $\mathcal{Q}_{\mathrm{goal}}$ are updated accordingly by using the update information of the successor output node $v_{y,\mathrm{succ}}$, i.e., reordering of the priorities after updating the key value of $v_{y,\mathrm{succ}}$ to the queue $\mathcal{Q}$ and reordering the goal output nodes in $\mathcal{Q}_{\mathrm{goal}}$ if $v_{y,\mathrm{succ}}$ happens to be a goal output node (Lines

26-27). These steps are repeated until there is no promising nonstationary output node left in the priority queue $\mathcal{Q}$, i.e., $\mathcal{Q}.\texttt{top\_key}() \succeq \mathcal{Q}_{\text{goal}}.\texttt{top\_key}()$.

The auxiliary procedures used in the `Extend` and `Replan` procedures are shown in Algorithm 21. The `UpdateQueue` procedure is used to maintain the priority queue $\mathcal{Q}$ whenever a new output node is created or key value of an output node that is already in the queue is updated. During a call to the `UpdateQueue` procedure with the priority queue $\mathcal{Q}$ and the output node v_y, there are three possible cases. For the first case, if v_y is a nonstationary output node, i.e., $v_y.\mathrm{g} \neq v_y.\bar{\mathrm{g}}$, key value of v_y is updated and priorities in the queue reordered accordingly (Line 3). For the second case, if v_y is a nonstationary output node and it is not in the queue, then it is inserted to the queue $\mathcal{Q}$ with its key value (Line 5). Lastly, if v_y is a stationary output node, i.e., $v_y.\mathrm{g} = v_y.\bar{\mathrm{g}}$, and it is in the queue $\mathcal{Q}$, then, it is removed from the queue $\mathcal{Q}$ (Line 7).

The constructor procedures for node and edge data structures used in the $\mathrm{CL\text{-}RRT}^{\#}$ algorithm are given in Algorithm 18.

8.4.3 Properties of the Algorithm

The $\mathrm{CL\text{-}RRT}^{\#}$ algorithm provides both dynamic feasibility guarantee, that is, the lowest-cost reference trajectory computed by the algorithm can be tracked by the low-level controller, and asymptotic optimality guarantees, that is, the lowest-cost reference trajectory computed by the algorithm converges to the optimal reference trajectory almost surely. The former property is an immediate result of using closed-loop prediction during the search phase. During extension of the graph $\mathcal{G}_y$, if some segments of a reference trajectory can not be tracked, i.e., not dynamically feasible, then, the corresponding state trajectory is not stored in the graph $\mathcal{G}_\sigma$ constructed by the algorithm. The former property is due to the asymptotic optimality property of the $\mathrm{RRT}^{\#}$ algorithm [9]. The proposed algorithm incrementally grows a graph $\mathcal{G}_y$ in the output space in a similar fashion as the RRG algorithm does [64]. Therefore, the lowest-cost path encoded in $\mathcal{G}_y$ converges to the optimal output

trajectory in the output space almost surely. In addition, the lowest-cost output trajectory encoded in the graph $\mathcal{G}_y$ is extracted at the end of each iteration in a similar fashion as the $\mathrm{RRT}^{\#}$ algorithm does. Given the cost function that associates each edge in $\mathcal{G}_y$ with a non-negative cost values being *monotonic* and *bounded*, the proposed algorithm is asymptotically optimal.

Algorithm 19: The `Extend` Procedure

Extend($\mathcal{S}, X_{\text{goal}}, y$)
  $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}}) \leftarrow \mathcal{S}$;
  $(V_y, E_y) \leftarrow \mathcal{G}_y$; $(V_\sigma, E_\sigma) \leftarrow \mathcal{G}_\sigma$;
  $v_{y,\text{nearest}} \leftarrow$ `Nearest`$(\mathcal{G}_y, y)$;
  $r_{\text{new}} \leftarrow$ `Steer`$(v_{y,\text{nearest}}.y, y)$;
  if `ObstacleFree`(r_{new}) **then**
    $y_{\text{new}} \leftarrow r_{\text{new}}$.`back()`;
    $v_{y,\text{new}} \leftarrow$ `OutNode`(y_{new});
    $v_{y,\text{new}}.\mathtt{h} \leftarrow$ `ComputeHeuristic`$(y_{\text{new}}, Y_{\text{goal}})$;
    $V_{\text{near}} \leftarrow$ `Near`$(\mathcal{G}_y, y_{\text{new}}, |V_y|) \cup \{v_{y,\text{nearest}}\}$;
    $E_{y,\text{succ}} \leftarrow \varnothing$; $E_{y,\text{pred}} \leftarrow \varnothing$;
    foreach $v_{y,\text{near}} \in V_{\text{near}}$ **do**
      $r \leftarrow$ `Steer`$(y_{\text{new}}, v_{y,\text{near}}.y)$;
      if `ObstacleFree`(r) **then**
        $e_y \leftarrow$ `OutEdge`$(v_{y,\text{new}}, v_{y,\text{near}}, r)$;
        $E_{y,\text{succ}} \leftarrow E_{y,\text{succ}} \cup \{e_y\}$;
      $r \leftarrow$ `Steer`$(v_{y,\text{near}}.y, y_{\text{new}})$;
      if `ObstacleFree`(r) **then**
        $e_y \leftarrow$ `OutEdge`$(v_{y,\text{near}}, v_{y,\text{new}}, r)$;
        $E_{y,\text{pred}} \leftarrow E_{y,\text{pred}} \cup \{e_y\}$;
    $V'_\sigma \leftarrow \varnothing$; $E'_\sigma \leftarrow \varnothing$;
    foreach $e_y \in E_{y,\text{pred}}$ **do**
      $v_{y,\text{pred}} \leftarrow e_y.\mathtt{tail}$;
      $v_{\sigma,\text{pred}} \leftarrow v_{y,\text{pred}}.\mathtt{p}_\sigma$;
      $x_{\text{pred}} \leftarrow v_{\sigma,\text{pred}}.\sigma$.`back()`;
      $\sigma \leftarrow$ `Propagate`$(x_{\text{pred}}, e_y.r)$;
      if `ObstacleFree`(σ) **then**
        $v_{\sigma,\text{new}} \leftarrow$ `TrajNode`$(\sigma, e_y, E_{y,\text{succ}})$;
        $e_\sigma \leftarrow$ `TrajEdge`$(v_{\sigma,\text{pred}}, v_{\sigma,\text{new}}, \sigma)$;
        $V'_\sigma \leftarrow V'_\sigma \cup \{v_{\sigma,\text{new}}\}$;
        $E'_\sigma \leftarrow E'_\sigma \cup \{e_\sigma\}$;
        if $v_{y,\text{new}}.\bar{\mathtt{g}} > v_{y,\text{pred}}.\mathtt{g} +$ `Cost`(σ) **then**
          $v_{y,\text{new}}.\bar{\mathtt{g}} \leftarrow v_{y,\text{pred}}.\mathtt{g} +$ `Cost`(σ);
          $v_{y,\text{new}}.\mathtt{p}_y \leftarrow v_{y,\text{pred}}$;
          $v_{y,\text{new}}.\mathtt{p}_\sigma \leftarrow v_{\sigma,\text{new}}$;
    $V_y \leftarrow V_y \cup \{v_{y,\text{new}}\}$; $E_y \leftarrow E_y \cup E_{y,\text{succ}} \cup E_{y,\text{pred}}$;
    $V_\sigma \leftarrow V_\sigma \cup V'_\sigma$; $E_\sigma \leftarrow E_\sigma \cup E'_\sigma$;
    $\mathcal{G}_y \leftarrow (V_y, E_y)$; $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$;
    $\mathcal{Q} \leftarrow$ `UpdateQueue`$(\mathcal{Q}, v_{y,\text{new}})$;
    $\mathcal{Q}_{\text{goal}} \leftarrow$ `UpdateGoal`$(\mathcal{Q}_{\text{goal}}, v_{y,\text{new}}, X_{\text{goal}})$;
  return $\mathcal{S} \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$;

Algorithm 20: `Replan` Procedure

```
1  Replan(S, X_goal)
2      (G_y, G_σ, Q, Q_goal) ← S;
3      (V_σ, E_σ) ← G_σ;
4      while Q.top_key() ≺ Q_goal.top_key() do
5          v_y ← Q.pop();
6          v_y.g ← v_y.ḡ;
7          v_σ ← v_y.p_σ;
8          x ← v_σ.σ.back();
9          foreach e_y ∈ v_σ.outgoing do
10             v_y,succ ← e_y.head;
11             σ ← Propagate(x, e_y.r);
12             if ObstacleFree(σ) then
13                 v_σ,succ ← TrajNode(σ, e_y, outgoing(G_y, v_y,succ));
14                 e_σ ← TrajEdge(v_σ, v_σ,succ, σ);
15                 V_σ ← V_σ ∪ {v_σ,succ};
16                 E_σ ← E_σ ∪ {e_σ};
17         v_σ.outgoing ← ∅;
18         G_σ ← (V_σ, E_σ);
19         foreach v_σ,succ ∈ succ(G_σ, v_σ) do
20             σ ← v_σ,succ.σ;
21             v_y,succ ← v_σ,succ.e_y.head;
22             if v_y,succ.ḡ > v_y.g + Cost(σ) then
23                 v_y,succ.ḡ ← v_y.g + Cost(σ);
24                 v_y,succ.p_y ← v_y;
25                 v_y,succ.p_σ ← v_σ,succ;
26                 Q ← UpdateQueue(Q, v_y,succ);
27                 Q_goal ← UpdateGoal(Q_goal, v_y,succ, X_goal);
28     return S ← (G_y, G_σ, Q, Q_goal);
```

Algorithm 21: Auxiliary Procedures

UpdateQueue($\mathcal{Q}, v_y$)
  if $v_y.\mathtt{g} \neq v_y.\bar{\mathtt{g}}$ *and* $v_y \in \mathcal{Q}$ **then**
    $\mathcal{Q}$.`update(`v_y`,Key(`v_y`));`
  else if $v_y.\mathtt{g} \neq v_y.\bar{\mathtt{g}}$ *and* $v_y \notin \mathcal{Q}$ **then**
    $\mathcal{Q}$.`insert(`v_y`,Key(`v_y`));`
  else if $v_y.\mathtt{g} = v_y.\bar{\mathtt{g}}$ *and* $v_y \in \mathcal{Q}$ **then**
    $\mathcal{Q}$.`remove(`v_y`);`
  return $\mathcal{Q}$;
UpdateGoal($\mathcal{Q}_{\text{goal}}, v_y, Y_{\text{goal}}$)
  $v_\sigma \leftarrow v_y.\mathtt{p}_\sigma$;
  $x \leftarrow v_\sigma.\sigma$.`back();`
  if $x \in X_{\text{goal}}$ **then**
    if $v_y \in \mathcal{Q}_{\text{goal}}$ **then**
      $\mathcal{Q}_{\text{goal}}$.`update(`$v_y$`,Key(`$v_y$`));`
    else
      $\mathcal{Q}_{\text{goal}}$.`insert(`$v_y$`,Key(`$v_y$`));`
  return $\mathcal{Q}_{\text{goal}}$;
Key(v_y)
  return $k = (v_y.\bar{\mathtt{g}} + v_y.\mathtt{h}, v_y.\mathtt{h})$;

Algorithm 22: Node and Edge Constructor Procedures

OutNode(y)
- $v_y.y \leftarrow y$;
- $v_y.\mathtt{g} \leftarrow \infty$; $v_y.\bar{\mathtt{g}} \leftarrow \infty$;
- $v_y.\mathtt{h} \leftarrow 0$;
- $v_y.\mathtt{p}_y \leftarrow \varnothing$; $v_y.\mathtt{p}_\sigma \leftarrow \varnothing$;
- **return** v_y;

OutEdge($v_{y,\mathrm{from}}, v_{y,\mathrm{to}}, r$)
- $e_y.\mathtt{tail} \leftarrow v_{y,\mathrm{from}}$;
- $e_y.\mathtt{head} \leftarrow v_{y,\mathrm{to}}$;
- $e_y.r \leftarrow r$;
- **return** e_y;

TrajNode(σ, e_y, E_y)
- $v_\sigma.\sigma \leftarrow \sigma$;
- $v_\sigma.e_y \leftarrow e_y$;
- $v_\sigma.\mathtt{outgoing} \leftarrow E_y$;
- **return** v_σ;

TrajEdge($v_{\sigma,\mathrm{from}}, v_{\sigma,\mathrm{to}}, \sigma$)
- $e_\sigma.\mathtt{tail} \leftarrow v_{\sigma,\mathrm{from}}$;
- $e_\sigma.\mathtt{head} \leftarrow v_{\sigma,\mathrm{to}}$;
- $e_\sigma.\sigma \leftarrow \sigma$;
- **return** e_σ;

StateNode(x)
- $v_x.x \leftarrow x$;
- **return** v_x;

StateEdge($v_{x,\mathrm{from}}, v_{x,\mathrm{to}}, \sigma$)
- $e_x.\mathtt{tail} \leftarrow v_{x,\mathrm{from}}$;
- $e_x.\mathtt{head} \leftarrow v_{x,\mathrm{to}}$;
- $e_x.\sigma \leftarrow \sigma$;
- **return** e_x;

8.5 Numerical Simulations

The proposed algorithms are tested on a system of unicycle dynamics which is described by the following set of equations

$$\dot{x}_1 = x_4 \sin(x_3), \quad \dot{x}_2 = x_4 \cos(x_3), \quad \dot{x}_3 = u_1, \quad \dot{x}_4 = u_2,$$
$$y_1 = x_1, \quad y_2 = x_2,$$

where x_1, x_2 are the cartesian coordinates of a reference point of the vehicle, x_3 is the vehicle's heading angle, x_4 is the translational velocity of the vehicle, u_1, u_2 are the control inputs for the angular and translational velocity. Each control input takes values in an interval, i.e., $u_i \in [u_i^l, u_i^u]$. A simple controller similar to pure-pursuit controller is designed to track a given reference path [1]. The heading command is generated by following a look-ahead point on a given reference path. The speed command is given as a desired cruise speed v_{crs}. Each command is tracked by using a proportional controller.

In the first scenario, the goal is to navigate the vehicle from one waypoint to another in the counter-clockwise direction on a race track while minimizing the Euclidean path length. The size of the race track is (100m$\times$100m) and the origin is located at its center. The CL-RRT$^{\#}$algorithm was run for 1500 iterations, and the solution trees computed by the algorithm at different stages are shown in Figure 33. Initially, the vehicle stays at the waypoint $(-25, -45)$ with zero heading angle and speed (yellow square at bottom-left), and then it is tasked to move to the waypoint $(48, 33)$ (red square at top-right). As seen in Figure 33(a)-(d), the algorithm incrementally grows a graph in the output space (x_1, x_2), and each path in the graph correspond to a geometric reference path that is inputted to the closed-loop system. It is observed that the CL-RRT$^{\#}$algorithm quickly computes a long reference path. Then, it seeks alternative paths of the graph as more information is explored and improves the existing solution if closed-loop simulation of a new reference path yields lower cost. The nodes and edges of the graph correspond to waypoints and straight line segments. The lowest-cost path is shown in yellow color, and its length is

127.164. The corresponding state trajectories computed during closed-loop simulation are shown Figure 33(e)-(h).

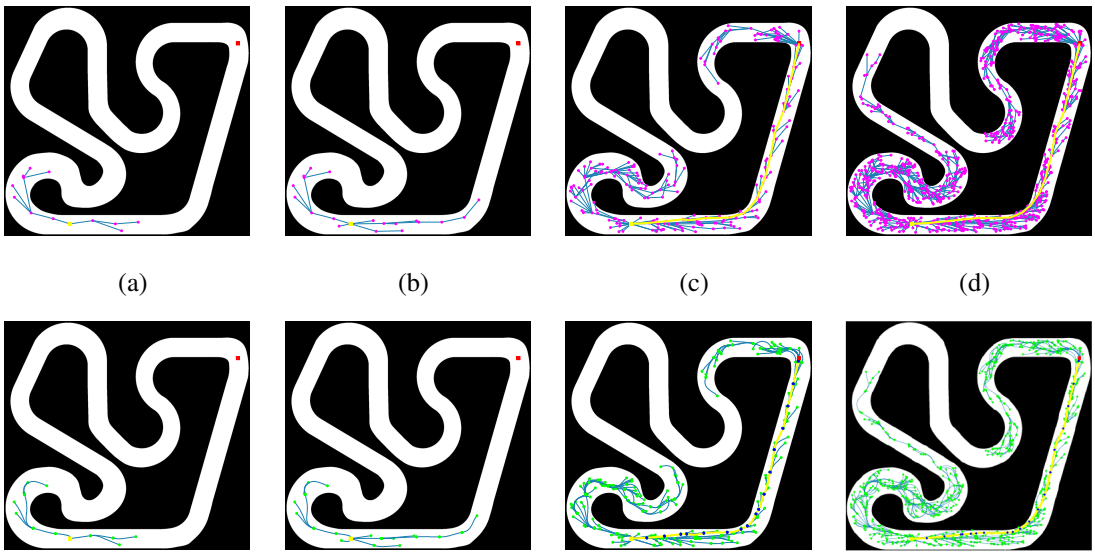


Figure 33: The evolution of the solution trees for reference paths and state trajectories computed by CL-RRT$^{\#}$ are shown in (a)-(d) and (e)-(h), respectively. The trees (a), (e) are at 50 iterations, (b), (f) are at 100 iterations, (c), (g) are at 500 iterations, and (d), (h) are at 1500 iterations.

In the second scenario, the goal is to navigate the vehicle on the race track continuously. The vehicle is tasked to navigate sequentially to a set of waypoints presumably coming from a high-level navigator. In each stage, the CL-RRT$^{\#}$algorithm was run for 1,500 iterations to find a motion plan from the current state of the vehicle to a desired next waypoint. Each next waypoint is sent to the motion planner as the vehicle gets close to the current waypoint. In this simulation, the vehicle is tasked to navigate four waypoints sequentially. The solution trees of reference paths and corresponding state trajectories for each step are shown in Figure 34. As seen during simulations, leveraging the dynamics information of the vehicle during the search phase allows the algorithm to construct dynamically feasible paths and avoid shortest paths that pass near-by the boundary of the track.

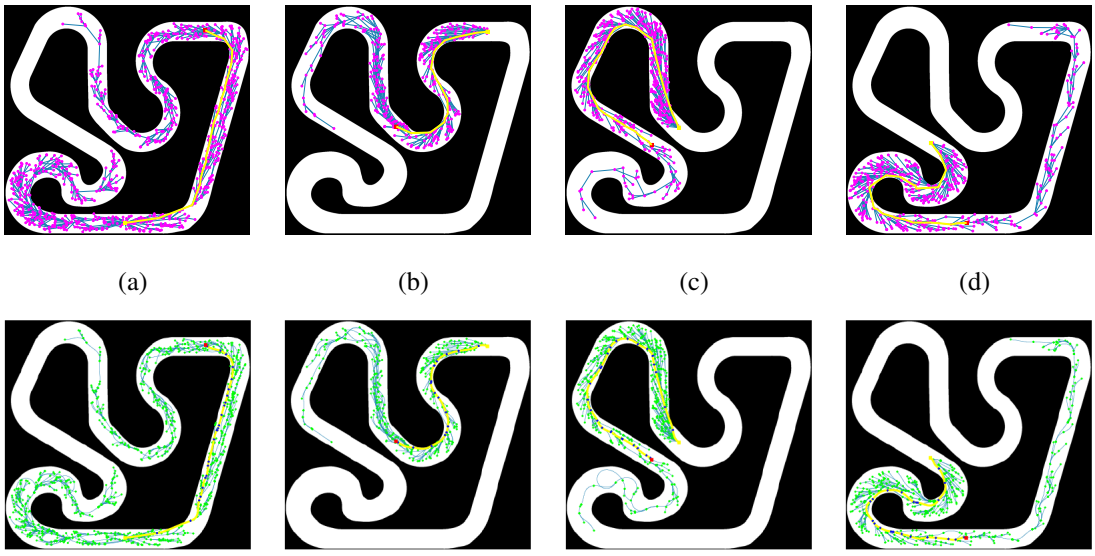


Figure 34: The evolution of the trees for reference paths and state trajectories computed at 1,500 iterations by CL-RRT$^{\#}$ are shown in (a)-(d) and (e)-(h), respectively. The trees (a), (e) are computed during the first waypoint navigation, (b), (f) are computed during the second waypoint navigation, (c), (g) are computed during the third waypoint navigation, and (d), (h) are computed during the fourth waypoint navigation.

8.6 Conclusion

In this chapter, we present a new asymptotically optimal motion planning algorithm, called CL-RRT$^{\#}$, that uses closed-loop prediction for trajectory-generation. The approach is a hybrid of the CL-RRT and the RRT$^{\#}$ algorithms. It incrementally grows a graph of reference trajectories that can be inputted to a low-level tracking controller and chooses the one that yields the lowest-cost state trajectory of the closed-loop system. The proposed approaches provides dynamic feasibility by construction and asymptotic optimality guarantee, i.e., converging to the optimal reference trajectory given a controller. Several simulation results on a system of unicycle dynamics were performed and the proposed approach is used for point-to-point navigation of the vehicle. Simulation results demonstrate the expected nice properties of the algorithm.

Chapter IX

HIGH-LEVEL ROUTE PLANNING FOR AN AUTONOMOUS ROTORCRAFT

9.1 Overview

The Autonomous Aerial Cargo/Utility System (AACUS) is a five-year program announced by the Office of Naval Research (ONR) and its primary focus is the development of control and sensor technologies to enable unmanned Vertical Take Off and Landing (VTOL) air systems to be tasked in cargo and delivery operations. AACUS encompasses the development of technologies that will allow any field personnel with no special training to command and control of unmanned vehicles, e.g., point-to-point navigation, autonomous landing, obstacle detection and avoidance. The developed unmanned systems are expected to be highly reliable for precision cargo delivery evacuating human casualties from remote sites.

Typical mission scenarios and the desired attributes of the AACUS is shown in Figure 35. As seen, during different phases of a mission, the autonomous vehicle is expected to have the capability of solving many point-to-point motion planning problems on-the-fly. As a part of this thesis, some variants of the $\mathrm{RRT}^{\#}$ algorithm is developed for the solution of high-level route planning of an autonomous rotorcraft.

The developed route planner software consists of four important modules: an efficient collision checker, a core planning algorithm, a path smoothing algorithm, and the speed profile computation algorithm. Sampling-based algorithms require efficient collision checkers to query if a given point or line segment yield a violation of feasibility requirement. To this end, an efficient collision checker that leverages the geometry of obstacles is developed. The $\mathrm{RRT}^{\#}$ algorithm is modified in order to compute a path that satisfies some

additional constraints, e.g., heading and flight-path angles constraints, and used as the core planning algorithm of the route planner. A path smoothing algorithm is also developed to reduce the number waypoints of the computed raw path. Finally, a dynamic programming based algorithm is developed in order to determine the speed profile of the vehicle along the smoothed path without violating performance constraints.

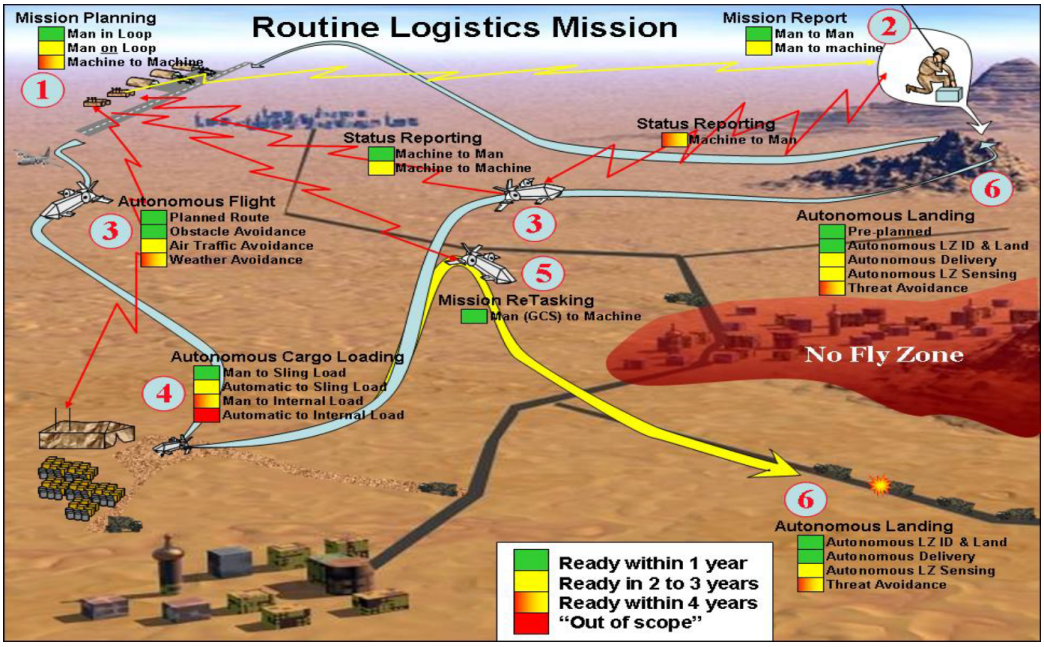


Figure 35: AACUS Logistics Mission with Replanning (courtesy of ONR [42])

The details of the implemented algorithms are given in the following sections. Numerical simulations are performed in several mission scenarios.

9.2 Problem Formulation

9.2.1 Notation and Definition

Let $X \subset \mathbb{R}^3$ denote the *safe-flight space* of the vehicle and X is formed as the union of non-convex polygonal prism. This is the volume of airspace within which flight of the vehicle is allowed. Let X_{obs} and X_{goal}, called the *obstacle space* and the *goal region*, be open subsets of X. Let X_{free}, also called the *free space*, denote the set defined as $X \backslash X_{\text{obs}}$. Obstacle space is a generic term that represents the set within which flight of the vehicle is not possible.

It is composed of different sets X_{terr}, X_{nfs}, and X_{ocs} that denote the *terrain*, the *no-flight space*, and the *occupied space* by objects, respectively, i.e., $X_{\text{obs}} = X_{\text{terr}} \cup X_{\text{nfs}} \cup X_{\text{ocs}}$. The terrain information is given as $X_{\text{terr}} = \{x = (x_1, x_2, x_3) \in X : x_3 \leq h_{\text{terr}}(x_1, x_2)\}$ where the function $h_{\text{terr}} : \mathbb{R}^2 \mapsto \mathbb{R}_+$ gives the maximum altitude of the terrain at a given point in 2D.

A *path* is a mapping $\sigma : [0, 1] \mapsto \mathbb{R}^d$, where $d \in \mathbb{N}$, $d \geq 2$. Let Σ denote the set of all paths, Σ_{free} the set of collision-free paths. Let ψ and γ denote the *heading angle* and *flight path angle* of the vehicle that are defined as follows:

$$\psi(s) = \arctan2\left(\sigma_2'(s), \sigma_1'(s),\right.$$
$$\gamma(s) = \arctan2\left(\sigma_3'(s), \|\sigma_{1,2}'(s)\|\right),$$

where $\sigma' = \dfrac{\mathrm{d}\sigma}{\mathrm{d}s}$ is the tangent vector of the path σ. A *speed profile* is a mapping $V : [0, 1] \mapsto \mathbb{R}_{\geq 0}$ and let $\mathcal{V}$ denote the set of feasible speed profiles of the vehicle.

Let P be a positively oriented, piecewise linear, simply closed curve in x_1x_2-plane and be specified by a sequence of points $\{x^1, x^2, \ldots, x^k\}$ where $x^i \in \mathbb{R}^2$. That is, P is a connected series of line segments in which only consecutive segments intersect and only at their endpoints (simple), also the first point coincides with the last one (closed), and one always has the curve interior to the left when traveling on P. Let $\texttt{Poly}(P)$ denote the region bounded by P and the curve P itself. Given such curve P, and two altitude values $h_1, h_2 \in \mathbb{R}_{\geq 0}$ with $h_1 \leq h_2$, let a polygonal prism be defined as follows:

$$\texttt{PolyPrism}(P, h_1, h_2) = \{x \in \mathbb{R}^3 : x_{1,2} \in \texttt{Poly}(P) \text{ and } h_1 \leq x_3 \leq h_2\}$$

Given a point $x \in \mathbb{R}^3$, and two angle values $\gamma_1, \gamma_2 \in [0, \pi/2]$ with $\gamma_1 \leq \gamma_2$, let a cone, whose pointing vertex is x and opening angle is restricted by γ_1 and γ_2, be defined as follows:

$$\texttt{Cone}(x, \gamma_1, \gamma_2) \triangleq \{x' \in \mathbb{R}^3 : z = x' - x \text{ and } \gamma_1 \leq \arctan2\left(z_3, \|z_{1,2}\|\right) \leq \gamma_2\},$$

where $x_{i,j} \triangleq (x_i \;\; x_j)^\intercal$.

Table 8: Nomenclature for high-level routing problem

x_{init}	=	initial position of the vehicle (x_1, x_2, x_3) (m)
x_{goal}	=	goal position of the vehicle (x_1, x_2, x_3) (m)
$\gamma_{\min}$, $\gamma_{\max}$	=	minimum and maximum values of the flight path angle (rad), respectively
$V_{\min}$, $V_{\max}$	=	minimum and maximum values of the speed of the vehicle (m/s), respectively
$\underline{\gamma}_{\text{to}}$,$\overline{\gamma}_{\text{to}}$	=	minimum and maximum values of angles used to define the left circular cone at x_{init}, respectively
$\underline{\gamma}_{\text{ldg}}$,$\overline{\gamma}_{\text{ldg}}$	=	minimum and maximum values of angles used to define the left circular cone at x_{goal}, respectively
r_{to}, r_{ldg}	=	radius values of the cylinders located at x_{init} and x_{goal}, respectively
ψ_{to},ψ_{ldg}	=	heading angle constraints at the takeoff and landing regions, respectively
h_{r}	=	range of the altitudes $\texttt{BBox}(X)$ (m)
h_{crz}	=	desired cruise altitude above the ground (m), i.e., the distance to the plane $x_3 = 0$

Given a point $x \in \mathbb{R}^3$, a radius $r \in \mathbb{R}_{>0}$, and a length $h \in \mathbb{R}_{>0}$, let a cylinder, that is aligned with vertical altitude direction, be defined as follows:

$$\texttt{Cyl}(x, r, h) \triangleq \{x' \in \mathbb{R}^3 : z = x' - x \text{ and } \|z_{1,2}\| \leq r, z_3 \leq h\}.$$

Given a nonempty set $X' \in \mathbb{R}^3$, let its minimum bounding box be defined as follows:

$$\texttt{BBox}(X') \triangleq \{x \in \mathbb{R}^3 : x_i \in [\underline{x}_i, \overline{x}_i] \text{ where } \underline{x}_i \triangleq \inf_{x \in X'} x_i \text{ and } \overline{x}_i = \sup_{x \in X'} x_i,\ i = 1, 2, 3\}$$

9.2.2 Problem Statement

Given the state space X, obstacle region X_{obs}, and goal region X_{goal}, find a path $\sigma : [0, 1] \mapsto X$ and a speed profile $V : [0, 1] \mapsto [V_{\min}, V_{\max}]$

- starts from the initial state, i.e., $\sigma(0) = x_{\text{init}}$,
- reaches the goal state, i.e., $\sigma(1) = x_{\text{goal}}$,
- avoids the obstacle region, i.e., $\sigma(s) \in X_{\text{free}}$ for all $s \in [0, 1]$
- obeys flight path angle constraint, i.e., $\gamma(s) \in [\gamma_{\min}, \gamma_{\max}]$,

- stays in the takeoff cone, i.e., $\sigma(s) \in \texttt{Cone}(x_{\text{init}}, \underline{\gamma}_{\text{to}}, \overline{\gamma}_{\text{to}})$ for all $z = \sigma(s) - x_{\text{init}}$ such that $\|z_{1,2}\| \leq r_{\text{to}}$
- stays in the landing cone, $\sigma(s) \in \texttt{Cone}(x_{\text{goal}}, \underline{\gamma}_{\text{ldg}}, \overline{\gamma}_{\text{ldg}})$ for all $z = \sigma(s) - x_{\text{goal}}$ such that $\|z_{1,2}\| \leq r_{\text{ldg}}$
- obeys the takeoff direction constraint, i.e., $\psi(s) = \psi_{\text{to}}$ for all $\sigma(s) \in \texttt{Cyl}(x_{\text{init}}, r_{\text{to}}, h_{\text{r}})$
- obeys the landing direction constraint, i.e., $\psi(s) = \psi_{\text{ldg}}$ for all $\sigma(s) \in \texttt{Cyl}(x_{\text{goal}}, r_{\text{ldg}}, h_{\text{r}})$
- satisfies the performance constraint of the vehicle, i.e, $V \in \mathcal{V}$
- $\sigma \in \arg\min_{\sigma' \in \Sigma} J(\sigma')$

9.2.3 Primitive Procedures

Sampling Procedure

The sampling procedure is given in Algorithm 23. It uniformly generates samples on a non-convex set which is formed as a list of polygonal prisms. The algorithm leverages some preprocessed information of polygonal prisms that form $\mathcal{O}_{\text{fly}}$, which is computed in the `InitEnvironment` procedure. First, the algorithm randomly picks a polygonal prism o_{fly} from the list $\mathcal{O}_{\text{fly}}$ (Lines 2-3). Then, it randomly selects one of the convex pieces of o_{fly} (Lines 4-6). Lastly, algorithm easily generates a random point x_{rand} from the minimum bounding box of o_{conv} (Line 8) and subsequently checks if x_{rand} lies inside o_{conv} (Line 9). These steps are repeated until a point within o_{conv} is generated, and then, the random sample x_{rand} is returned.

Polygonal prisms are preprocessed in the `InitEnvironment` procedure in order to speed up sampling and collision checking process. Each polygonal prism is decomposed into convex pieces and their minimum bounding boxes are computed. The `InitEnvironment` procedure is given in Algorithm 24. It creates a polygonal prism object for each entry of the safe-flight space and adds the object to the list $\mathcal{O}_{\text{fly}}$ (Lines 3-9). A flyable polygonal prism

Algorithm 23: Sample Procedure

```
1  Sample(O_fly)
2      i ← RandInt(1,O_fly.size());
3      o_fly ← O_fly[i];
4      O_conv ← o_fly.pieces;
5      i ← RandInt(1,O_conv.size());
6      o_conv ← O_conv[i];
7      do
8          x_rand ← GenerateSample(o_conv.bbox);
9      while ¬OnInterior(o_conv,x_rand)
10     return x_rand;
```

object is created by using information of a set of ordered points P and an altitude interval $[h_{\min}, h_{\max}]$ (Line 6). After its creation, a simple convex decomposition algorithm is used to slice the polygonal prism into convex pieces (Lines 7-8), and the created object is added to the $\mathcal{O}_{\text{fly}}$ list (Line 9). Then, likewise, the same steps are repeated for each entry of the obstacle space and all created objects are added to the $\mathcal{O}_{\text{obs}}$ list (Lines 10-16). Lastly, all geometric objects in both lists are pairwise compared and checked if they have intersection, and an obstacle is added to the list of a flyable polygonal prism if it intersects (Lines 17-20). Finally, both $\mathcal{O}_{\text{fly}}$ and $\mathcal{O}_{\text{obs}}$ are formed into a structure $\mathcal{O}_{\text{env}}$ and the algorithm returns $\mathcal{O}_{\text{env}}$ (Lines 21-22).

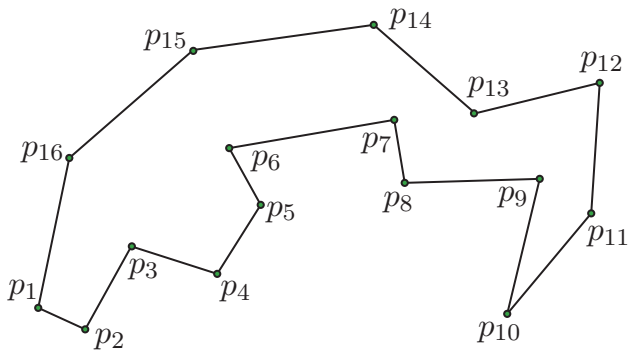


Figure 36: A non-convex polygon

The simple convex decomposition algorithm works in two steps, namely, triangulation and joining the convex pieces. The triangulation algorithm is given in Algorithm 25, and its steps are explained on a simple non-convex polygon in Figure 36. The algorithm starts

Algorithm 24: Initialize Environment Procedure

```
1  InitEnvironment(Θ)
2      (Θ_fly, Θ_obs) ← Θ;
3      O_fly ← ∅;
4      foreach θ ∈ Θ_fly do
5          (P, h_min, h_max) ← θ;
6          o_fly ← PolyPrism(P,h_min,h_max);
7          O_tri ← Triangulate(o_fly);
8          o_fly.pieces ← JoinConvexSets(O_tri);
9          O_fly.push_back(o_fly);
10     O_obs ← ∅;
11     foreach θ ∈ Θ_obs do
12         (P, h_min, h_max) ← θ;
13         o_obs ← PolyPrism(P,h_min,h_max);
14         O_tri ← Triangulate(o_obs);
15         o_obs.pieces ← JoinConvexSets(O_tri);
16         O_obs.push_back(o_obs);
17     foreach o_fly ∈ O_fly do
18         foreach o_obs ∈ O_obs do
19             if HasIntersection(o_fly,o_obs) then
20                 o_fly.obstacles.push_back(o_obs);
21     O_env ← (O_fly, O_obs);
22     return O_env;
```

with the first point of the point set of the polygon and tries to remove the point x from the point set P by forming a triangle around itself. If the line formed by its preceding and succeeding points lies within the polygon formed by P (Line 10), then a triangle is created as a polygonal prism by using the point set formed by its preceding point, itself, and its succeeding point (Lines 11-13). The newly created triangle o_{tri} is added to the list $\mathcal{O}_{\text{tri}}$ (Line 14), and X is removed from P (Line 15). These steps are repeated until no further triangles can be formed, i.e, the size of P becomes less than three (Line 7). The resulted triangulation is shown in Figure 37.

As seen, the triangulation algorithm yields many small triangles, therefore an joining algorithm is subsequently is called to form larger convex sets by joining these small triangles. The joining algorithm is given in Algorithm 26. The algorithm iteratively works on each convex set o in the list $\mathcal{O}$ and attempts to form a larger convex set by join o with

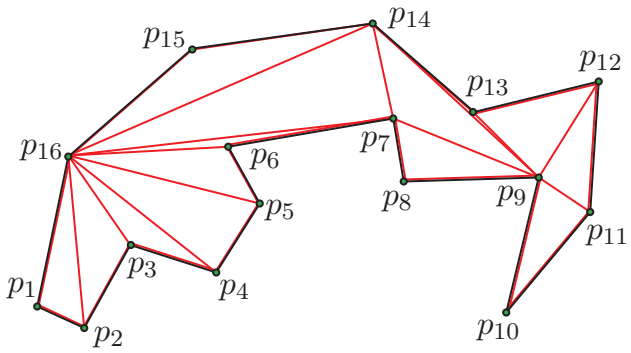


Figure 37: Triangulation of a non-convex polygon

Algorithm 25: Simple Triangulation Algorithm

```
1  Triangulate(o)
2      P ← o.points;
3      h_min ← o.h_min;
4      h_max ← o.h_max;
5      O_tri ← ∅;
6      x ← P.front();
7      while P.size() ≥ 3 do
8          x_prev ← prev(x);
9          x_next ← next(x);
10         if Line(x_prev,x_next) ∈ Polygon(P) then
11             P' ← {x_prev, x, x_next};
12             o_tri ← PolyPrism(P',h_min,h_max);
13             o_tri.p ← o;
14             O_tri.push_back(o_tri);
15             P.remove(x);
16         x ← x_next;
17     return O_tri;
```

its neighbor sets. These steps are repeated until no further convex set can be created by joining. The final convex decomposition of the polygon is shown in Figure 38.

Minimum bounding boxes of convex pieces are computed after the convex decomposition of polygonal prisms. Convex pieces and their minimum bounding boxes are shown in Figure 39.

Whenever a random sample is generated within the minimum bounding box of a convex piece, the Ray Casting Algorithm is used to determine if the point lies inside the convex piece or not [109]. An example case is shown in Figure 40. The algorithm first draws a line

Algorithm 26: Join Convex Sets Procedure

```
JoinConvexSets(𝒪)
    𝒪_conv ← ∅;
    while 𝒪.size() > 0 do
        o ← 𝒪.pop_front();
        𝒪_near ← GetNeighbors(𝒪, o);
        while 𝒪_near.size() > 0 do
            o_near ← 𝒪_near.pop_front();
            if IsConvex(Join(o, o_near)) then
                o ← Join(o, o_near);
                𝒪'_near ← GetNeighbors(𝒪, o_near);
                𝒪_near ← 𝒪_near ∪ 𝒪'_near;
                𝒪.remove(o_near);
        𝒪_conv.push_back(o);
    return 𝒪_conv;
```

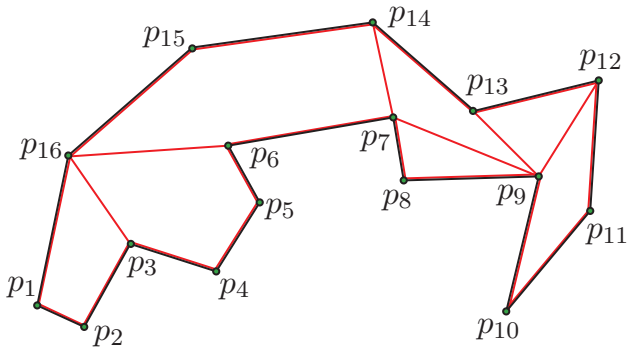


Figure 38: Convex decomposition of a non-convex polygon

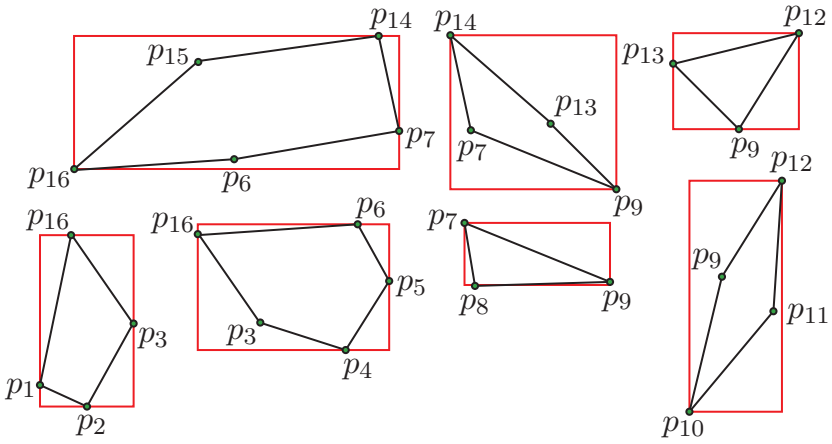


Figure 39: Convex pieces of a non-convex polygon and their minimum bounding boxes

that is parallel to x_1-axis and emanates from the query point, and then, counts the number of intersection of this line with the edges of the convex piece. If the number of intersections is odd, then the query point is decided to be inside the convex pieces.

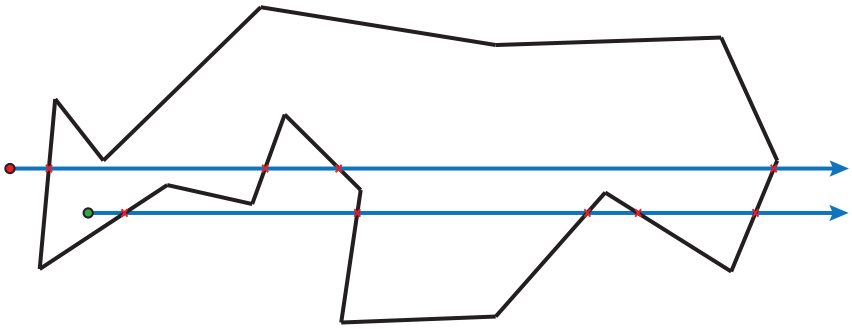


Figure 40: The Ray Casting Algorithm is used for determining interior points of non-convex polygons.

Collision Checking Procedure

The collision checking procedure leverages the geometric information of obstacles and is implemented in the `HasIntersection` procedure. Given a line segment, and a polygonal prism, the `HasIntersection` procedure checks if the line intersects with any face of the polygonal prism. This approach is more efficient than checking if any point of a discrete set of points on the line lies inside the polygonal prism, especially for long line segments.

9.3 Route Planning Algorithm

9.3.1 Details of Data Structures

We assume that the initial and the goal nodes are given as $(x_{\text{init}}, y_{\text{init}}, z_{\text{init}}, \psi_{\text{init}}, V_{\text{init}})$ and $(x_{\text{goal}}, y_{\text{goal}}, z_{\text{goal}}, \psi_{\text{goal}}, V_{\text{goal}})$, respectively.

9.3.2 Details of the Procedures

The body of the RoutePlanner algorithm is given in Algorithm 29. First, the algorithm reads the geometric information about the operating environment and initializes efficient data structures to represent the geometric shapes (Line 2) by calling the `InitEnvironment`

Algorithm 27: Node and Edge Constructor Procedures

StateNode(x)
  $v_x.x \leftarrow x$;
  $v_x.\mathtt{g} \leftarrow \infty$;
  $v_x.\bar{\mathtt{g}} \leftarrow \infty$;
  $v_x.\mathtt{h} \leftarrow 0$;
  $v_x.\mathtt{p}_x \leftarrow \varnothing$;
  return v_x;
StateEdge($v_{x,\mathrm{from}}$,$v_{x,\mathrm{to}}$,σ)
  $e_x.\mathtt{tail} \leftarrow v_{x,\mathrm{from}}$;
  $e_x.\mathtt{head} \leftarrow v_{x,\mathrm{to}}$;
  $e_x.\sigma \leftarrow \sigma$;
  return e_x;

Algorithm 28: Polygonal Prism Constructor Procedure

PolyPrism($P, h_{\min}, h_{\max}$)
  $o.h_{\min} \leftarrow h_{\min}$;
  $o.h_{\max} \leftarrow h_{\max}$;
  $o.\mathtt{points} \leftarrow P$;
  $o.\mathtt{bbox} \leftarrow \mathtt{BoundingBox}(P,h_{\min},h_{\max})$;
  $o.\mathtt{pieces} \leftarrow \varnothing$;
  $o.\mathtt{p} \leftarrow \varnothing$;
  $o.\mathtt{obstacles} \leftarrow \varnothing$;
  return o;

procedure. The non-convex polygonal prisms that form the safe-flight space are decomposed into convex pieces. Intersection between non-convex polygonal prisms of the safe-flight space and regions that represent occupied space and non-flight space is checked. An object is added to the list of a non-convex polygonal prism if they have non-zero intersection. The computed lists are returned with the output parameter $\mathcal{O}_{\mathrm{env}}$. Then, the initial and goal positions of the vehicle together with additional heading and speed constraints at the takeoff and landing regions are read from the parameter file Θ (Lines 3-4). The feasibility and planner parameters are formed subsequently (Lines 5-6). The $\mathrm{RRT}^{\#}$algorithm is called with planner parameters θ_{pln} to find a geometric path between x_{init} and x_{goal} while satisfying feasibility parameters θ_{feas} (Line 7). Once a geometrically feasible path is computed, the raw path is smoothed by calling the `SmoothPath` procedure (Lines 8-9). Finally,

the `ComputeSpeedProfile` is called to compute a feasible speed value for each way point of the smoothed path (Line 10) and the resulted path σ' is returned (Line 11).

Algorithm 29: Body of the Route Planner Algorithm

```
1  RoutePlanner(Θ)
2      O_env ← InitEnvironment(Θ);
3      (x_init, ψ_to, V_to) ← GetInitialState(Θ);
4      (x_goal, ψ_ldg, V_ldg) ← GetGoalState(Θ);
5      θ_feas ← GetFeasibilityParams(Θ);
6      θ_pln ← (O_env, ψ_to, ψ_ldg);
7      (σ, O_env) ← RRT#(θ_pln, θ_feas, x_init, x_goal);
8      θ_opt ← GetOptimizerParams(Θ);
9      σ ← SmoothPath(O_env, θ_feas, θ_opt, σ);
10     σ' ← ComputeSpeedProfile(σ, V_to, V_ldg);
11     return σ';
```

The $\mathrm{RRT}^{\#}$ Algorithm for Route Planner

The $\mathrm{RRT}^{\#}$ algorithm used for the route planner is given in Algorithm 30. First, the algorithm calls the `InitPlanner` procedure to create the search graph with initial and goal nodes and a set feasible edges based on the heading constraints (Line 2). Then, the algorithm incrementally grows the graph for fixed number of iterations (Lines 4-7). Once a fixed number iterations is completed, the algorithm extracts the solution path by starting from the goal node and backtracking over the parent node of each node along the optimal path (Lines 8-13). Finally, the computed path along with a data structured representing the environment are returned (Line 14).

Initialization of the Planner

The `InitPlanner` procedure is given in Algorithm 31. This procedure creates a graph with the initial node, the goal node, and a set of feasible edges satisfying heading angle constraints $(\psi_{\mathrm{to}}, \psi_{\mathrm{ldg}})$. Then, two artificial cylinders centered at x_{init} and x_{goal} are created to block the takeoff and landing regions to prevent further inclusion of new nodes by the planning algorithm.

Algorithm 30: Body of the RRT$^{\#}$ Algorithm

```
1  RRT#(θpln,θfeas,xinit,xgoal)
2      (G, Oenv) ← InitPlanner(θpln,θfeas,xinit,xgoal);
3      (Ofly, Oobs) ← Oenv;
4      for k = 1 to N do
5          xrand ← Sample(Ofly);
6          G ← Extend(Oenv,θfeas,G,xrand);
7          G ← Replan(G);
8      σ ← ∅;
9      (V, E) ← G;
10     vx ← find(V,xgoal);
11     while vx ≠ ∅ do
12         σ.push_front(vx.x);
13         vx ← vx.px;
14     return (σ, Oenv);
```

First, the heading constraints are read from the planner parameter structure θ_{pln} (Line 1). Then, the initial node is created by using the initial state x_{init}, and it is added to node set after its g-, $\bar{\text{g}}$- and h-values are set properly (Lines 3-7). To enforce takeoff heading constraints, the algorithm computes the set of points which yields feasible edges satisfying the heading angle ψ_{to} constraint in the `ComputeTakeoffPoints` procedure (Line 8). Additional nodes and edges are created for the set of points X_{pts} (Lines 8-18). Similar steps are repeated for the goal state x_{goal} and the set of points that yields feasible edges satisfying the heading angle ψ_{ldg} constraint (Lines 19-27). Once all feasible edges are created on both takeoff and landing regions, artificial obstacles are placed there in order to prevent the RRT$^{\#}$ algorithm to create new nodes in the neighborhood of x_{init} and x_{goal}. Cylinders centered at x_{init} and x_{goal} are created and added to the obstacle set $\mathcal{O}_{\text{obs}}$ (Lines 28-39). Finally, the constructed graph $\mathcal{G}$ and the list $\mathcal{O}_{\text{env}}$ are returned (Line 41).

Algorithm 31: The Initialize Planner Procedure

InitPlanner(θ_{pln},θ_{feas},x_{init},x_{goal})
  $(\mathcal{O}_{\text{env}}, \psi_{\text{to}}, \psi_{\text{ldg}}) \leftarrow \theta_{\text{pln}}$;
  $v_{x,\text{init}} \leftarrow$ `StateNode(`x_{init}`)`;
  $v_{x,\text{init}}.\mathtt{g} \leftarrow 0$;
  $v_{x,\text{init}}.\bar{\mathtt{g}} \leftarrow 0$;
  $v_{x,\text{init}}.\mathtt{h} \leftarrow$ `ComputeHeuristic(`x_{init},x_{goal}`)`;
  $V \leftarrow \{v_{x,\text{init}}\}$;
  $X_{\text{pts}} \leftarrow$ `ComputeTakeoffPoints(`$\mathcal{O}_{\text{env}}$,θ_{feas},x_{init},ψ_{to}`)`;
  foreach $x \in X_{\text{pts}}$ **do**
    $v_x \leftarrow$ `StateNode(`x`)`;
    $\sigma \leftarrow$ `Steer(`x_{init},x`)`;
    $v_x.\bar{\mathtt{g}} \leftarrow$ `Cost(`σ`)`;
    $v_x.\mathtt{g} \leftarrow v_x.\bar{\mathtt{g}}$;
    $v_x.\mathtt{h} \leftarrow$ `ComputeHeuristic(`x,x_{goal}`)`;
    $v_x.\mathtt{p}_x \leftarrow v_{x,\text{init}}$;
    $e_x \leftarrow$ `StateEdge(`$v_{x,\text{init}}$,v_x, σ`)`;
    $V \leftarrow V \cup \{v_x\}$;
    $E \leftarrow E \cup \{e_x\}$;
  $v_{x,\text{goal}} \leftarrow$ `StateNode(`x_{goal}`)`;
  $V \leftarrow V \cup \{v_{x,\text{goal}}\}$;
  $X_{\text{pts}} \leftarrow$ `ComputeLandingPoints(`$\mathcal{O}_{\text{env}}$,θ_{feas},x_{goal},ψ_{ldg}`)`;
  foreach $x \in X_{\text{pts}}$ **do**
    $v_x \leftarrow$ `StateNode(`x`)`;
    $v_x.\mathtt{h} \leftarrow$ `ComputeHeuristic(`x,x_{goal}`)`;
    $e_x \leftarrow$ `StateEdge(`v_x,$v_{x,\text{goal}}$,σ`)`;
    $V \leftarrow V \cup \{v_x\}$;
    $E \leftarrow E \cup \{e_x\}$;
  $(\mathcal{O}_{\text{fly}}, \mathcal{O}_{\text{obs}}) \leftarrow \mathcal{O}_{\text{env}}$;
  $r_{\text{to}} \leftarrow \theta_{\text{feas}}.r_{\text{to}}$;
  $(f, o_{\text{init}}) \leftarrow$ `OnFlyZone(`$\mathcal{O}_{\text{fly}}$,x_{init}`)`;
  $h_{\min} \leftarrow o_{\text{init}}.h_{\min}$; $h_{\max} \leftarrow o_{\text{init}}.h_{\max}$;
  $o_{\text{cyc}} \leftarrow$ `Cylinder(`x_{init},r_{to},$h_{\min}$,$h_{\max}$`)`;
  $\mathcal{O}_{\text{obs}}$`.push_back(`o_{cyc}`)`;
  $r_{\text{ldg}} \leftarrow \theta_{\text{feas}}.r_{\text{ldg}}$;
  $(f, o_{\text{goal}}) \leftarrow$ `OnFlyZone(`$\mathcal{O}_{\text{fly}}$,x_{goal}`)`;
  $h_{\min} \leftarrow o_{\text{goal}}.h_{\min}$; $h_{\max} \leftarrow o_{\text{goal}}.h_{\max}$;
  $o_{\text{cyc}} \leftarrow$ `Cylinder(`x_{goal},r_{ldg},$h_{\min}$,$h_{\max}$`)`;
  $\mathcal{O}_{\text{obs}}$`.push_back(`o_{cyc}`)`;
  $\mathcal{O}_{\text{env}} \leftarrow (\mathcal{O}_{\text{fly}}, \mathcal{O}_{\text{obs}})$;
  $\mathcal{G} \leftarrow (V, E)$;
  return $(\mathcal{G}, \mathcal{O}_{\text{env}})$;

Bisection Based Feasibility Checking Algorithm

The `IsFeasible` procedure is given in Algorithm 32. Each segment added to the graph needs to satisfy the following conditions:

- lying entirely in the safe-flight space
- avoids occupied and no-flight spaces
- satisfy flight path angle constraints
- lying within the cones centered at the takeoff and landing regions
- satisfying heading angle constraints

One possible approach to check feasibility of a line segment could be to get a discrete set of points on the line segment and to check if each individual point satisfies the desired constraints. Although this approach is easier to implement, it is less efficient and requires more computation since the number of points increases with the path length for a desired resolution. Another approach is to develop a bisection based feasibility checking algorithm that leverages the geometric information (e.g. convexity) of the safe-flight spaces. The idea is to divide a long line segment into short segments and to check if each individual segment lies entirely in the safe-flight space. If both end points of the line segment lie inside the same convex piece, then, this implies that it belongs to the safe-flight space and no further division is required.

Given the environment $\mathcal{O}_{\text{env}}$, the feasibility parameters θ_{feas}, and the end points x and x', the `IsFeasible` procedure begins by forming a line segment that has end points x and x' and adds it to the list $\mathcal{L}$ (Lines 3-4). Then, the individual line segments are checked for feasibility and subdivided into two pieces as required in a loop. First, the line segment at the top of the list $\mathcal{L}$ is popped and set to l at the beginning of the loop (Line 6). Then, both end points of l are checked if they lie inside the safe-flight space $\mathcal{O}_{\text{fly}}$ in the `OnFlightSpace` algorithm. If so, the corresponding polygonal prisms that contain the end points are also

returned (Lines 7-8). If any of end points is outside of $\mathcal{O}_{\text{fly}}$, then the algorithm returns `False`. If the length of l is less than or equal to a predefined value, then, the feasibility of the end points are checked individually and the algorithm returns `False` if there is a collision (Lines 11-13). For the last case, i.e., both end points lie inside the safe-flight space, the algorithm first finds the corresponding convex pieces that contain the end points (Lines 15-16). If both end points belong to the same convex piece, then, the line segment l is not subdivided further. Its feasibility value is subsequently checked by the `HasCollision` procedure and `False` is returned if there is a collision (Lines 17-19). If both end points belong to different convex pieces, then, l is subdivided into two equal line segments, and they are added to the list $\mathcal{L}$ (Lines 21-25). Finally, if the algorithm does not return with `False` value due to a collision or finding a line segment that lies outside the safe-flight space, then, it returns `True` (Line 26).

Algorithm 32: Feasibility Check Procedure

```
1  IsFeasible(O_env, θ_feas, x, x')
2      (O_obs, O_fly) ← O_env;
3      ℓ ← Line(x, x');
4      L.push_back(ℓ);
5      while L.size() > 0 do
6          ℓ ← L.pop_front();
7          (f_p, o_p) ← OnFlightSpace(O_fly, ℓ.p);
8          (f_q, o_q) ← OnFlightSpace(O_fly, ℓ.q);
9          if ¬f_p ∨ ¬f_q then
10             return False;
11         else if |ℓ| ≤ d_min then
12             if HasCollision(O_env, θ_feas, ℓ.p, ℓ.q) then
13                 return False;
14         else
15             o_p,conv ← GetConvexPiece(o_p, ℓ.p);
16             o_q,conv ← GetConvexPiece(o_q, ℓ.q);
17             if o_p,conv = o_q,conv then
18                 if HasCollision(O_env, θ_feas, ℓ.p, ℓ.q) then
19                     return False;
20             else
21                 x_middle ← (ℓ.p + ℓ.q)/2;
22                 ℓ' ← Line(p, x_middle);
23                 L.push_back(ℓ');
24                 ℓ' ← Line(x_middle, q);
25                 L.push_back(ℓ');
26     return True;
```

Path Smoothing Algorithm

The raw path computed by the $\mathrm{RRT}^{\#}$ algorithm usually contains many waypoints and short segment. This raw path is smoothed by using an algorithm similar to line-of-sight filter. The path smoothing algorithm is given in Algorithm 33. Since the first and last line segments of the raw path are precomputed and satisfy the heading angle constraints, these segments are first excluded from smoothing process (Lines 2-3). Then, the path optimizer parameters are read, and intermediate segments of the raw path are optimized by the `OptimizePath` procedure (Lines 4-5). Finally, the first and last segments are integrated to the optimized path, and the resulted path is returned (Lines 6-8).

Algorithm 33: The Smooth Path Procedure

```
1 SmoothPath(O_env,θ_feas,θ_opt,σ)
2     x_init ← σ.pop_front();
3     x_goal ← σ.pop_back();
4     θ_opt ← GetOptimizerParams(Θ);
5     σ ← OptimizePath(O_env,θ_feas,θ_opt,σ);
6     σ.push_front(x_init);
7     σ.push_back(x_goal);
8     return σ;
```

The path optimizer algorithm is given in Algorithm 34. First, the algorithm reads the parameters for path shortening and relaxing algorithms (Lines 2-3). Then, the raw path is shortened by using a line-of-sight filter in the `ShortenPath` procedure. This step yields a path with fewer waypoints and of lower cost (Line 4). The shortened path usually passes very close to the obstacles, and this may reduce the clearance of the path in certain directions. In order to provide enough clearance for the path, the algorithm first increases the number of waypoints on the optimized path in the `RefinePath` procedure, then it subsequently attempts to perturb these waypoints away from the obstacles in the `RelaxPath` procedure (Lines 5-6). Then, the relaxed path is shortened one more time in the forward direction to reduce the number of waypoints, and length of each segment is made to be less or equal than $l_{\max}$ (Lines 8-10). Finally, the computed path is returned (Line 11).

Algorithm 34: Path Optimization Procedure

```
1  OptimizePath(O_env,θ_feas,θ_opt,σ)
2      (N_s,fb, f_ref, l_disc) ← GetShortenParams(θ_opt);
3      (N_r,fb, N_points, l_per) ← GetRelaxParams(θ_opt);
4      σ ← ShortenPath(O_env,θ_feas,N_s,fb,f_ref,l_disc,σ);
5      σ ← RefinePath(l_disc,σ);
6      σ ← RelaxPath(O_env,θ_feas,N_r,fb,N_points,l_per,σ);
7      N_s,fb ← 1; f_ref ← False; l_disc ← 0;
8      σ ← ShortenPath(O_env,θ_feas,N_s,fb,f_ref,l_disc,σ);
9      l_max ← θ_opt.l_max;
10     σ ← RefinePath(l_max,σ);
11     return σ;
```

The path refinement algorithm is given in Algorithm 35. Given a positive length value

$l_{\max} \in \mathbb{R}_{>0}$, and a path $\sigma \in \Sigma_{\text{free}}$, it processes over each segment and ensure its length to be less than or equal to $l_{\max}$ by adding additional middle waypoints if necessary.

Algorithm 35: Path Refinement Procedure

```
1  RefinePath(l_max,σ)
2      x ← σ.front();
3      σ'.push_back(x);
4      while x ≠ σ.back() do
5          l ← ‖σ.next(x) − x‖;
6          N ← ceil(l/l_max);
7          Δx ← (σ.next(x) − x)/N;
8          x' ← x + Δx;
9          for n = 1 to N do
10             σ'.push_back(x');
11             x' ← x' + Δx;
12         x ← σ.next(x);
13     return σ';
```

The path shortening algorithm is given in Algorithm 36. It uses a line-of-sight filter and runs in a forward-backward scheme to improve quality of a given path.

The path relaxation algorithm is given in Algorithm 37. It relaxes a given path by running in a forward-backward scheme and perturbing the waypoints of the path slightly. In each iteration, the algorithm detects where the obstacles are located and moves the waypoints in the opposite direction.

Algorithm 36: Path Shortening Procedure

ShortenPath($\mathcal{O}_{\text{env}}$,θ_{feas},N_{fb},f_{ref},$l_{\max}$,σ)
  for $k = 1$ *to* N_{fb} **do**
    if $f_{\text{ref}} =$ `True` **then**
      $\sigma \leftarrow$ `RefinePath(`σ,$l_{\max}$`);`
    $\sigma' \leftarrow \varnothing$;
    $x_{\text{from}} \leftarrow \sigma$`.front();`
    σ'`.push_back(`x_{from}`);`
    $x_{\text{to}} \leftarrow \sigma$`.next(`$x_{\text{from}}$`);`
    while $x_{\text{to}} \neq \sigma$`.back()` **do**
      if $\neg$`IsFeasible(`$\mathcal{O}_{\text{env}}$,θ_{feas},x_{from},σ`.next(`x_{to}`))` **then**
        σ'`.push_back(`x_{to}`);`
        $x_{\text{from}} \leftarrow x_{\text{to}}$;
      $x_{\text{to}} \leftarrow \sigma$`.next(`$x_{\text{to}}$`);`
    σ'`.push_back(`x_{to}`);`
    $\sigma \leftarrow \sigma'$`.reverse();`
  if N_{fb} *is even* **then**
    $\sigma' \leftarrow \sigma$;
  return σ';

Algorithm 37: Path Relaxation Procedure

RelaxPath($\mathcal{O}_{\text{env}}$,θ_{feas},N_{fb},N_{points},l_{per},σ)
  for $k = 1$ *to* N_{fb} **do**
    $\sigma' \leftarrow \varnothing$;
    $x \leftarrow \sigma$.front();
    σ'.push_back(x);
    $x \leftarrow \sigma$.next(x);
    while $x \neq \sigma$.back() **do**
      if $\|x - \sigma$.prev(x)$\| < l_{\text{min}} \vee$
        $\|\sigma$.next(x) $- x\| < l_{\text{min}}$ **then**
        $x' \leftarrow \sigma$.next(x);
        while $x' \neq \sigma$.back() $\wedge$ IsCollinear(σ.prev(x),x,x') **do**
          $x' \leftarrow \sigma$.next(x');
        $x_{\text{new}} \leftarrow$ PerturbPoint(x,σ.prev(x),x');
        $f_{\text{pn}} \leftarrow$ IsFeasible($\mathcal{O}_{\text{env}}$,θ_{feas},σ.prev(x),x_{new});
        $f_{\text{nn}} \leftarrow$ IsFeasible($\mathcal{O}_{\text{env}}$,θ_{feas},x_{new},σ.next(x));
        if $f_{\text{pn}} \wedge f_{\text{nn}}$ **then**
          $x \leftarrow x_{\text{new}}$;
      σ'.push_back(x);
      $x \leftarrow \sigma$.next(x);
    σ'.push_back(x);
    $\sigma \leftarrow \sigma'$.reverse();
  if N_{fb} *is even* **then**
    $\sigma' \leftarrow \sigma$;
  return σ';

9.4 Numerical Simulations

The simulation framework is implemented in standard C++ language and communicates with different components via Robot Operating System (ROS) [102]. Therefore, the route planner is integrated to the framework as a ROS package. It gets ROS message which contains mission specific information, computes a feasible route quickly, and publishes the solution as a ROS message to the ROS nodes of interest. In these numerical simulations, the route planner is used to solve several high-level route planning problems in order to demonstrate its efficiency and capabilities. Each mission takes place at locations of different terrain characteristics, non-convex safe-flight space, and no-flight space. For each mission, the computed route is shown in blue color from top and side views in Figures 41, 45, 49. Due to safety requirements, the safe-flight space is shrunk, and the no-flight spaces are expanded by some margins. This allows the algorithm to compute a route which has enough clearance from the boundaries as seen in Figures 42, 46, 50. The nodes that are populated by the $\mathrm{RRT}^{\#}$ algorithm as iterations go are also shown in the same figures. They are plotted with different colors depending on the iteration at which they are created. The nodes that are created at earlier and late iterations are shown in blue and red colors, respectively. The cylinders of different radius and desired heading angle directions at take-off and landing regions are shown in black. The same results are plotted in 3D from a different perspective in Figures 43, 47, 51, and with the nodes in Figures 44, 48, 52.

In addition to simulations, the route planner was also tested extensively during the real flight tests on June 2015. During a test scenario, the pilot invoked the planner with specific mission information before the take-off, and the route information was shown on pilot's display once it was computed. Then, the pilot tried to fly the helicopter by tracking the computed route. The implemented route planner successfully completed all piloted flights.

Mission 1

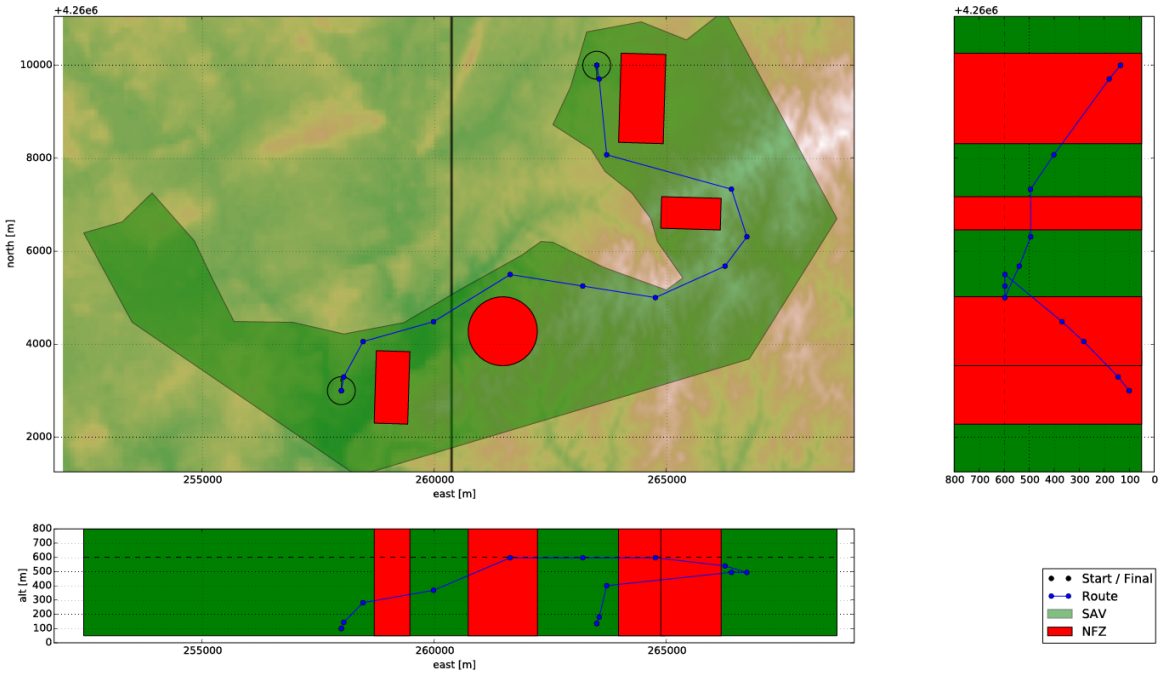


Figure 41: The computed route by the planner for Circus mission. The flight path angle constraints are seen clearly from the side view.

Figure 42: The nodes which are populated by the planner during Circus mission. The cone constraints at take-off and landing regions are seen clearly from the side view.

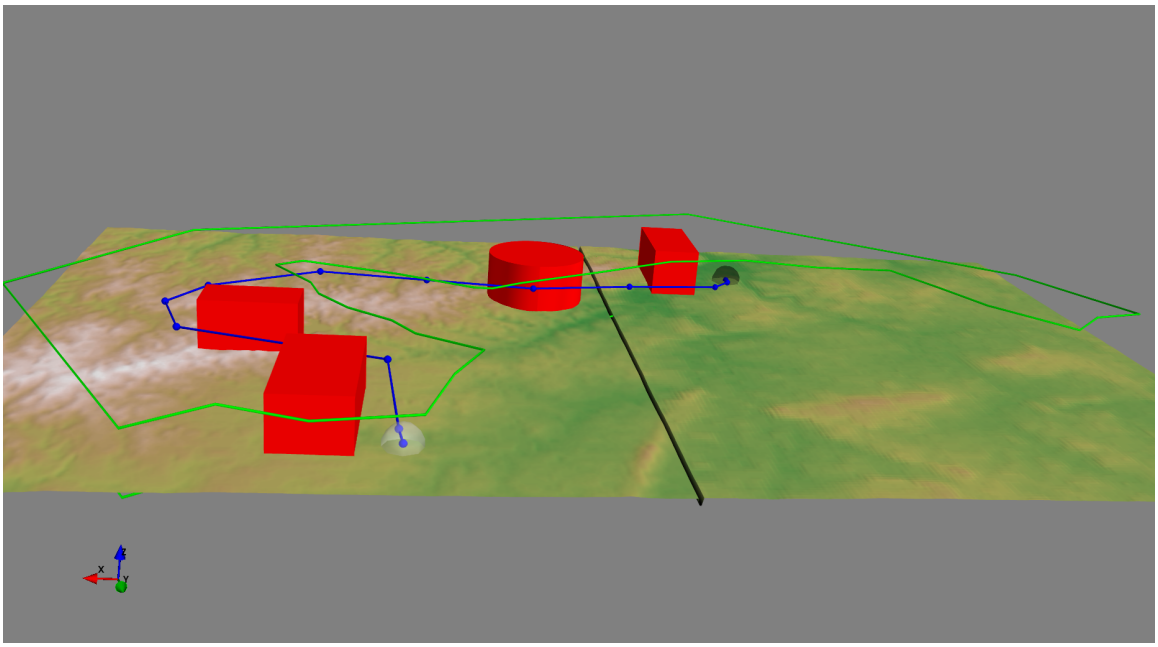


Figure 43: The computed route by the planner for Circus mission (3D view). The planner computes a path that climbs to the desired cruise altitude quickly as seen in the middle of the route.

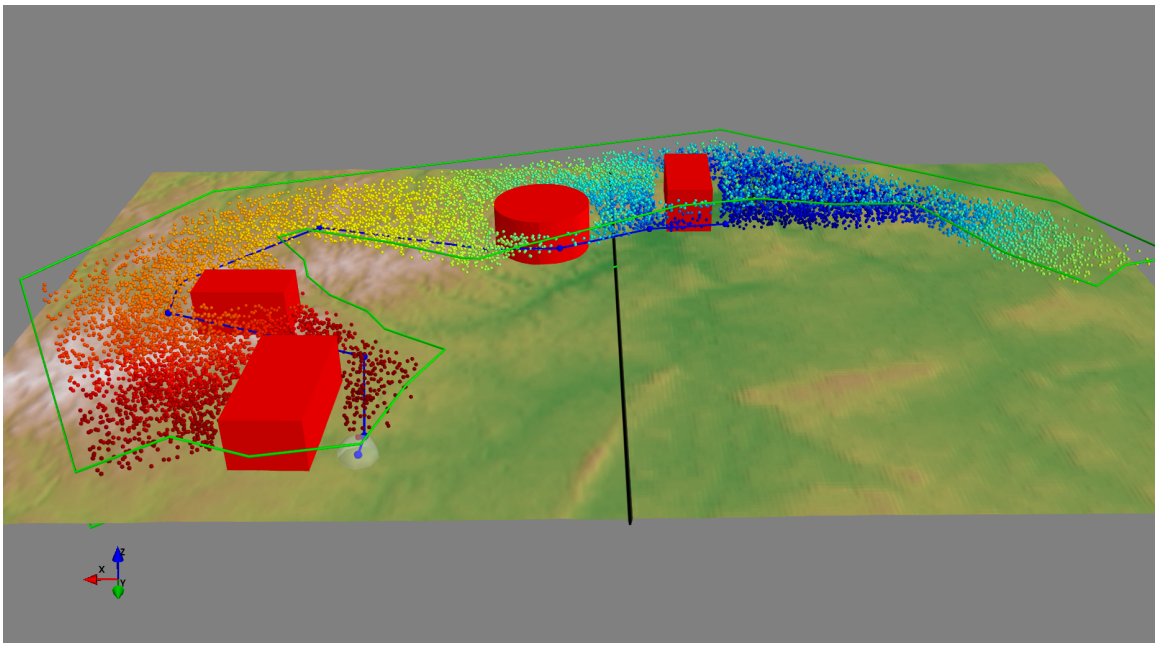


Figure 44: The nodes which are populated by the planner during Circus mission (3D view). Different colors of the nodes show the growth direction of the underlying graph.

Mission 2

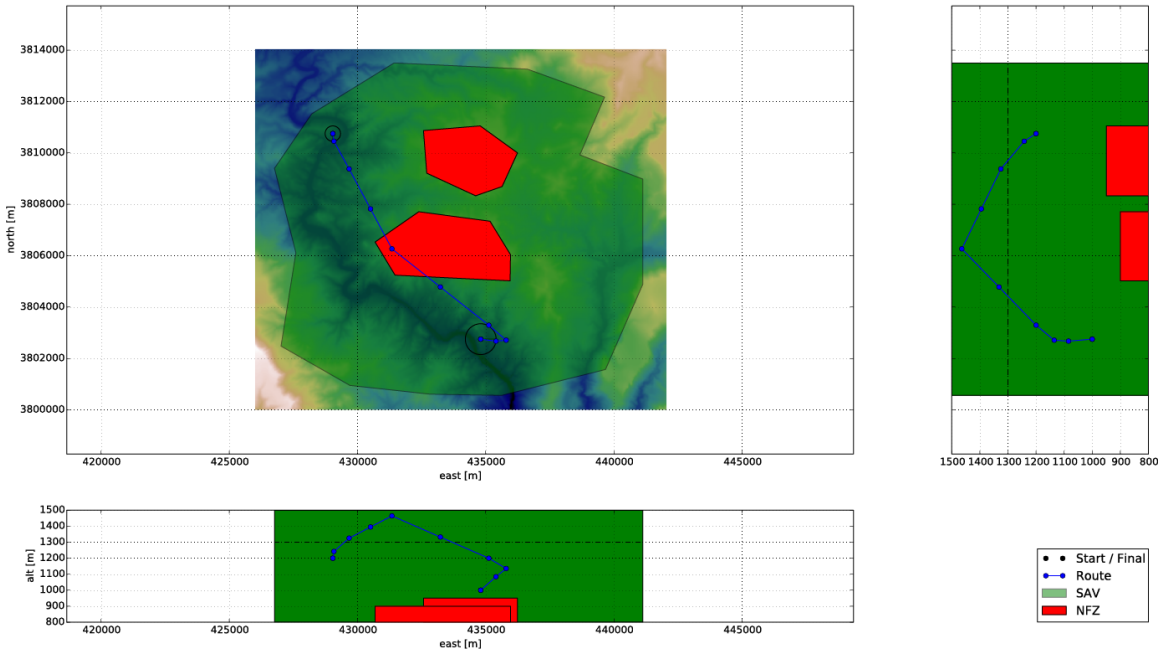


Figure 45: The computed route by the planner for Verde River, Arizona mission. The flight path angle constraints are seen clearly from the side view.

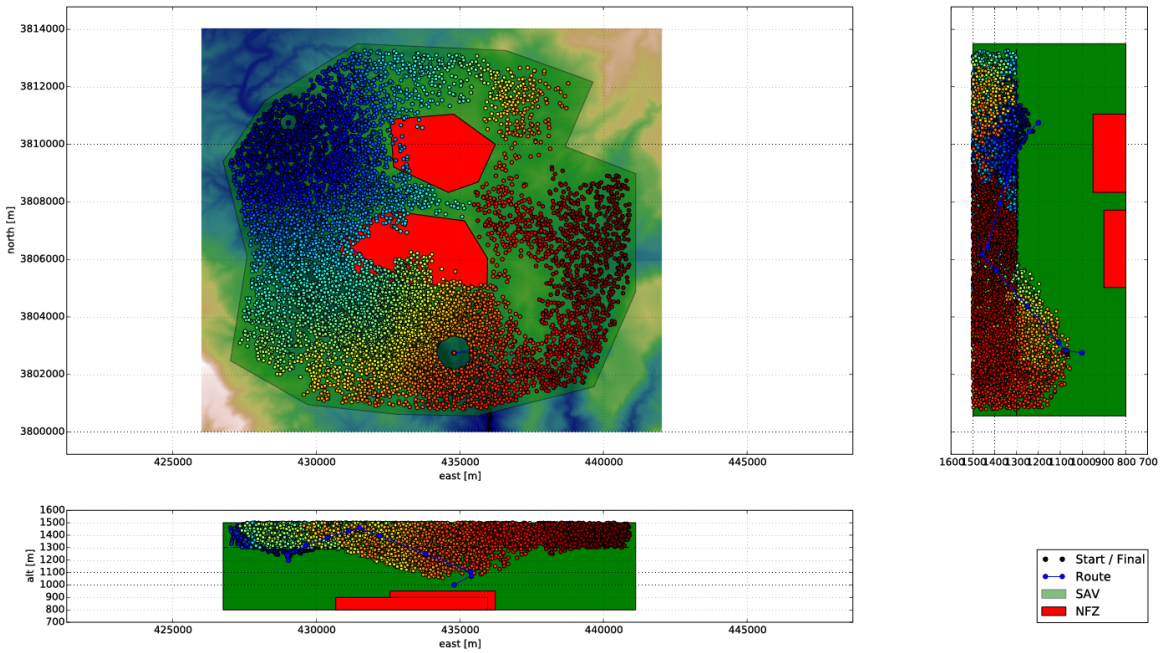


Figure 46: The nodes which are populated by the planner during Verde River, Arizona mission. The cone constraints at take-off and landing regions are seen clearly from the side view.

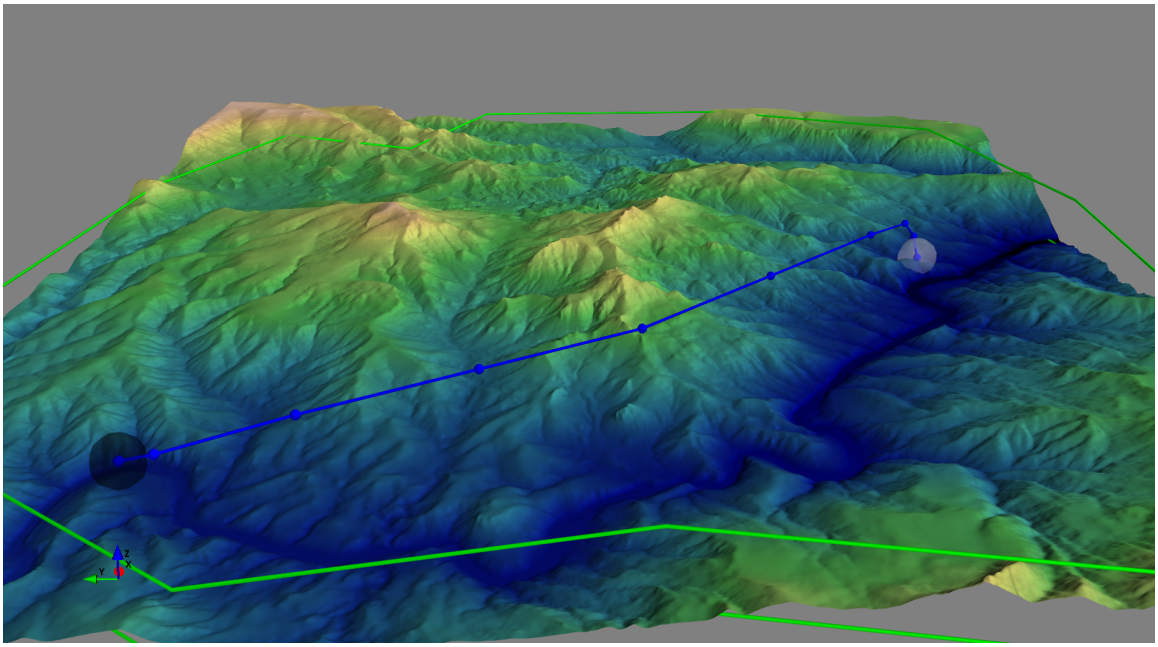


Figure 47: The computed route by the planner for Verde River, Arizona mission (3D view). The planner computes a path that climbs to the desired cruise altitude quickly as seen in the middle of the route.

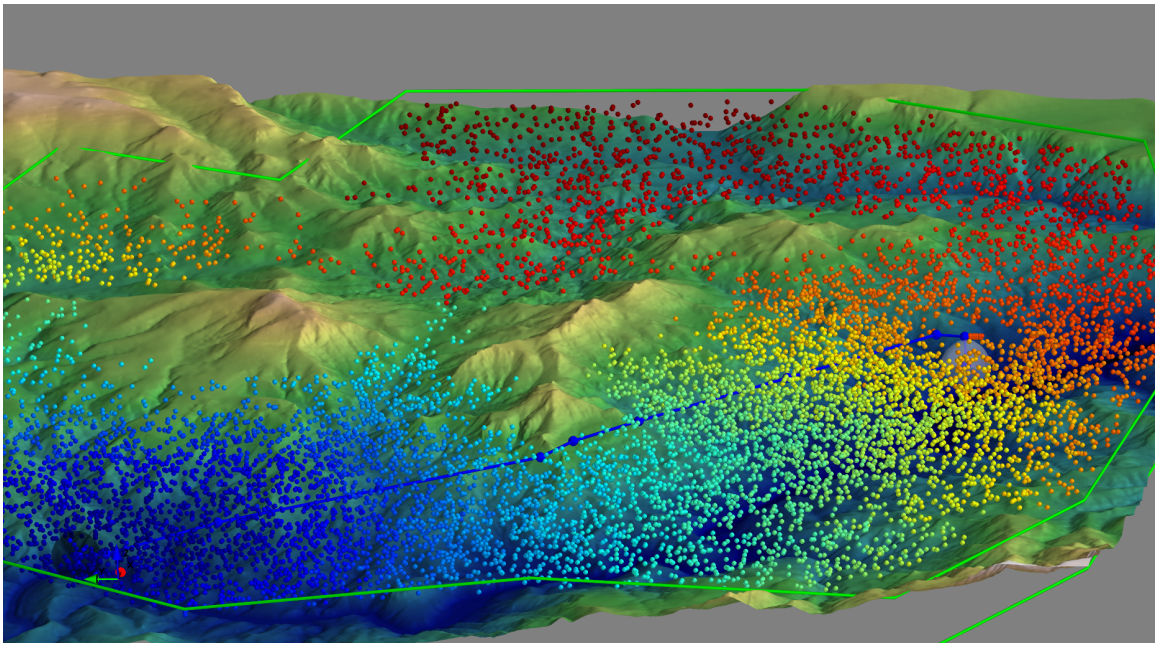


Figure 48: The nodes which are populated by the planner during Verde River, Arizona mission (3D view). Different colors of the nodes show the growth direction of the underlying graph

Mission 3

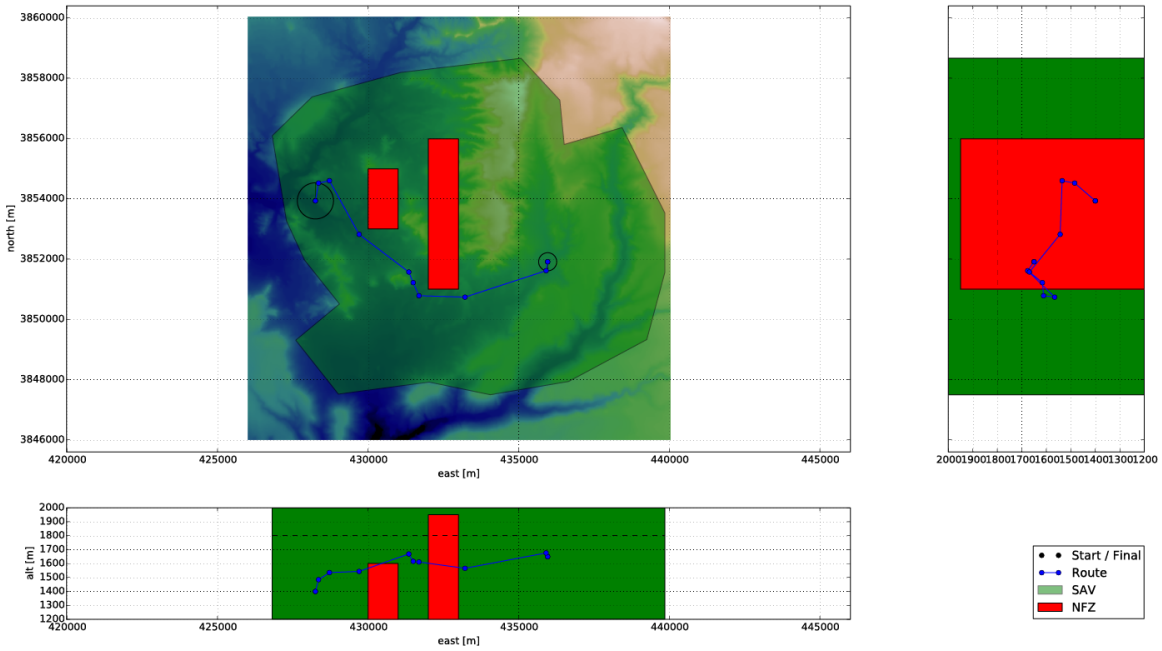


Figure 49: The computed route by the planner for Sedona, Arizona mission. The flight path angle constraints are seen clearly from the side view.

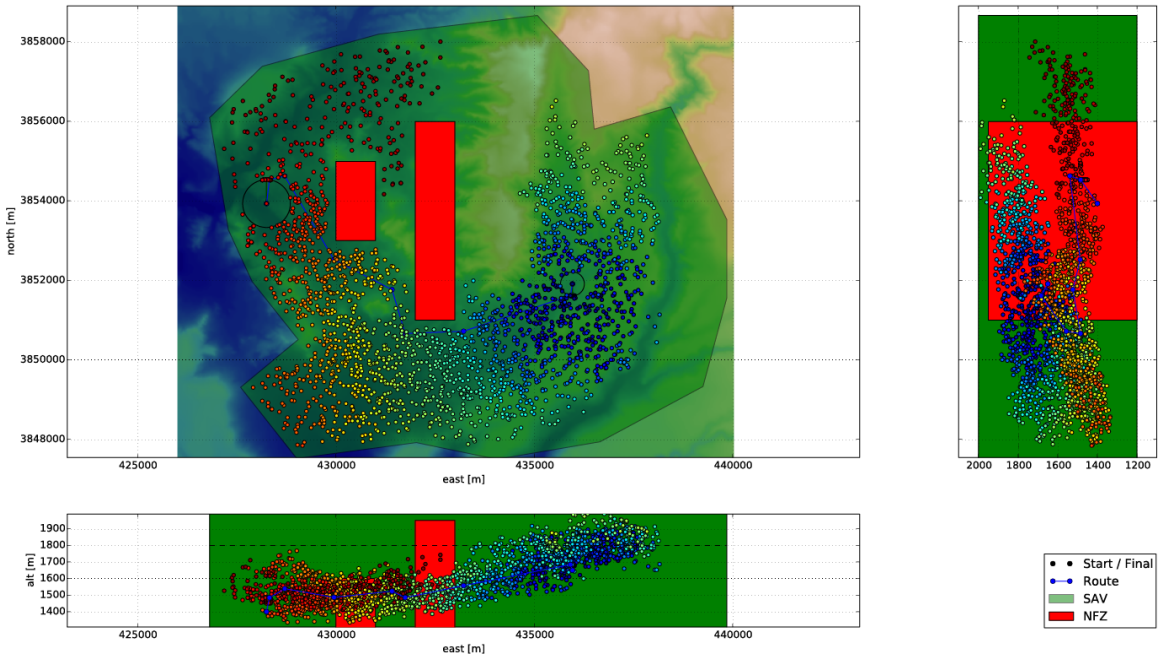


Figure 50: The nodes which are populated by the planner during Sedona, Arizona mission. The cone constraints at take-off and landing regions are seen clearly from the side view.

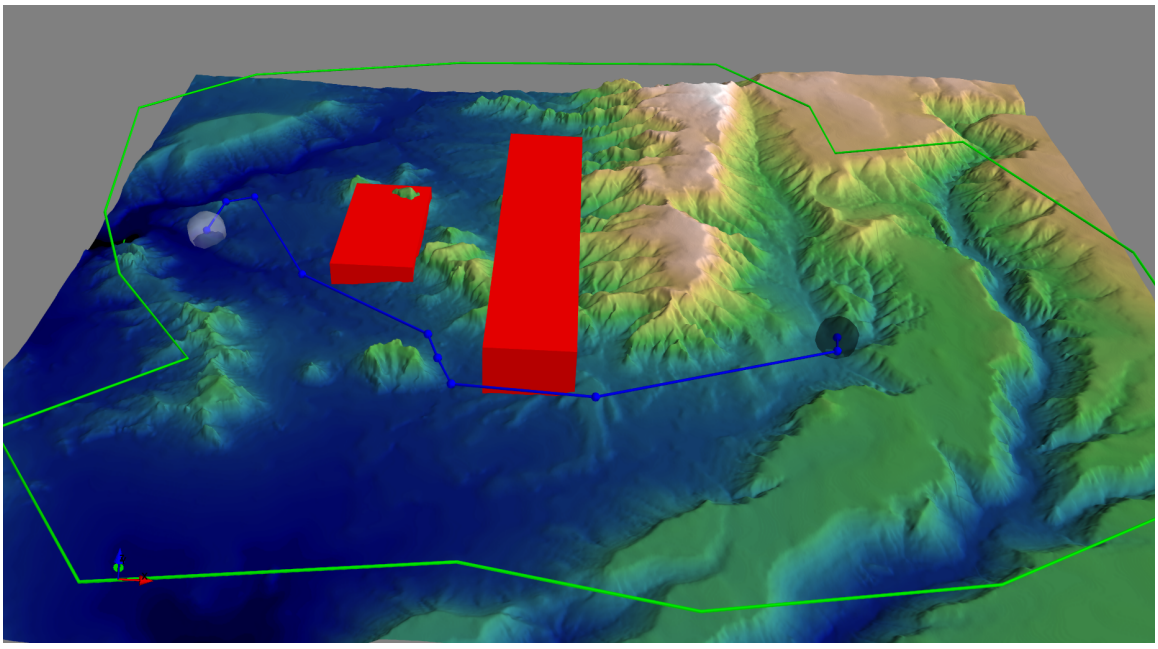


Figure 51: The computed route by the planner for Sedona, Arizona mission (3D view). The planner computes a path that climbs to the desired cruise altitude quickly as seen in the middle of the route.

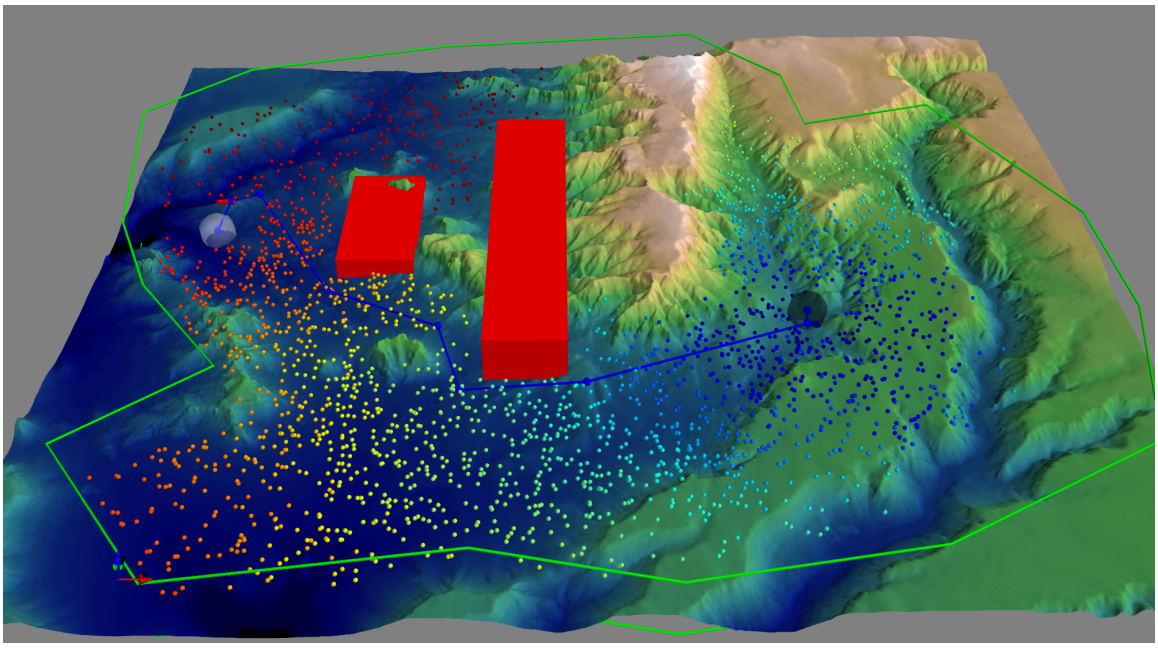


Figure 52: The nodes which are populated by the planner during Sedona, Arizona mission (3D view). Different colors of the nodes show the growth direction of the underlying graph

9.5 Conclusion

In this chapter, we present a demonstration of the proposed algorithms on a real-world engineering problem, i.e., the high-level route planning of a rotorcraft. Several numerical simulations were performed in order to demonstrate the capabilities of the route planner on different missions. The implemented route planner was also tested in real flight tests, and it performed successfully during piloted flights.

Chapter X

CONCLUSION

10.1 Contributions

10.1.1 Novel Connection Between DP and Sampling-based Motion Planning

We presented a new interpretation of optimal motion planning problem as a form of machine learning problem. This new interpretation defines two subtasks, namely, exploitation and exploration, and presents how these tasks can be implemented within the framework of sampling-based algorithms. We used DP and ML algorithms to implement the exploitation and exploration tasks, respectively. Specifically, VI and PI methods are used to exploit the information collected so far and extract the best solution encoded in the underlying data structure. Each method yields different implementation model, e.g., sequential or parallel. It is shown that PI methods yield a massively parallel implementation that can leverage state-of-the-art GPUs. We presented numerical simulations to demonstrate the efficiency of the proposed sampling-based motion planning algorithms.

10.1.2 Machine Learning Guided Exploration

Collecting good samples can be a very tedious task in high-dimensional search problems. To achieve better exploration, we utilized ML algorithms to develop adaptive sampling strategies without incurring high computational overhead. Most of the sampling-strategies aim to collect samples from free space. We took this goal one step further and identified the relevant region of a query, i.e., a subset of the free space that contains the optimal solution. We introduced a two-step test that quickly decides whether a given sample has the potential to improve the existing solution. The proposed adaptive sampling method can be seamlessly integrated to any single-query sampling-based motion planning algorithm. We presented numerical simulations to demonstrate the efficiency of the proposed adaptive

sampling methods.

10.1.3 Stochastic Motion Planning

We extended the applications of sampling-based algorithms and utilized them in order to solve a class of stochastic optimal control problems. Specifically, path integral control framework gives another way to compute optimal policies for stochastic motion planning problems. This alternative approach requires sampling of trajectories from uncontrolled dynamics and use them to compute path integral control law. We used the RRT algorithm to sample useful trajectories from challenging dynamics and presented the first sampling-based algorithm which leverages path integral framework. We presented numerical simulations to demonstrate the efficiency of the proposed Path Integral - RRT algorithm.

10.1.4 Optimal Motion Planning via Closed-loop Prediction

We presented the first asymptotically optimal sampling-based motion planning algorithm which uses closed-loop prediction. Several approaches have been developed to solve optimal kinodynamic motion planning problem. A widely used approach is to solve the problem in two steps: first finding geometrically feasible path and second computing the controls by tracking this path via a low-level controller. Despite its applicability to many systems, this approach lacks dynamic feasibility guarantee. The other popular approach is kinodynamic sampling-based motion planners which randomly sample controls and grow a graph in the state space. The main drawback of kinodynamic motion planners with optimality guarantees is that they require complex steering procedures which are not readily available. To remedy these challenges, we developed the CL-$\mathrm{RRT}^{\#}$ algorithm which is a hybrid of both approaches. Given a low-level tracking controller, it grows a graph in the output space and computes the lowest-cost reference trajectory that can be inputted to the controller. The controls are not sampled directly, instead they are computed during closed-loop simulation of the system. The CL-$\mathrm{RRT}^{\#}$ algorithm provides dynamic feasibility guarantees by construction and is asymptotic optimal. We presented numerical

simulations and demonstrated the efficiency of the proposed CL-RRT$^{\#}$ algorithm.

10.1.5 Knowledge Transfer from Academia to Industry

We implemented the RRT$^{\#}$ algorithm for the high-level route planning of a rotorcraft under AACUS project. The implemented route planner was tested through extensive simulations and used in real flight tests. It was integrated to the flight control system of a full scale helicopter and performed very well during piloted flight tests.

10.2 Future Work and Open Problems

10.2.1 Real-time Motion Planning

As shown in Chapter 3, some versions of the RRT$^{\#}$ algorithm have great potential for massive parallelization. These algorithms can be implemented on state-of-the-art GPUs and tested on high-dimensional robotic system for real-time motion planning.

10.2.2 Different ML Algorithms for Exploration

In this thesis, we used a few ML algorithms to solve exploration for motion planning. We proposed an approach which boils down to solve classification and regression problems. There is a plethora of algorithms to solve these central problems. A nice follow-up work can be to implement different ML algorithms for exploration and benchmark their performance. Also, for the sake of simplicity we used the configuration of the robot as the feature vector of the machine learning problem. The underlying configuration space can have very different topology depending on the geometry of the robot, location of the obstacles in the workspace and their geometric properties. Another interesting work can be to learn good features for the classification and regression problems and to investigate if alternative features yield better performance or not.

10.2.3 Applications of the CL-RRT$^{\#}$ Algorithm

It was shown that the performance of the CL-RRT$^{\#}$ algorithm highly depends on the designed low-level tracking controller. One possible future work can be to use different tracking controllers on the same problem and compare their solutions. Another nice follow-up work can be to apply the CL-RRT$^{\#}$ algorithm to complex dynamical systems for which standard kinodynamic planners are not applicable.

10.2.4 Sampling-based Path Integral Control Algorithms

We presented only the standard RRT algorithm within the path integral control framework. There is a plethora of singe-query sampling-based motion planning algorithms that can be used in the same way. A nice follow-up work can be integration of different sampling-based algorithms to path integral control framework and compare of their performance. Due to the formulation of the path integral control, many algorithms require sampling of trajectories and expectation operation which can be done in parallel.

First, a parallel version of the algorithm can be implemented by sampling local trajectories or computing several initial trajectories simultaneously. Second, since there exist many variants of the standard RRT algorithm, one can implement different sampling-based algorithms to compute initial trajectories and incorporate them within the path integral framework. For example, the RRT* and the RRT$^{\#}$ algorithms, which are both asymptotically optimal, can be used to compute bundles of good initial trajectories in a single pass; however, such an algorithm would require more elaborate computations for implementing the steering function, e.g., backward integration of a stochastic differential equation. Another work can be the implementation of parallel versions of the proposed algorithm on GPUs and application of the PI-RRT algorithm to robotic systems with many states/degrees of freedom.

10.2.5 Bi-directional Search Algorithms

For the proposed algorithms, it is crucial to reach the target set as early as possible in order to converge to the optimal solution faster. In that respect, a bi-directional version of the $\mathrm{RRT}^{\#}$ algorithm (like the RRT-connect in [73]) can be developed in order to shorten the first time-to-connect to the goal set, and this will help to identify the relevant region in very early iterations. Hence, the bi-directional version can improve the exploration, convergence rate, and memory requirement further by guiding samples towards the relevant region.

10.3 Final Remarks

There are a lot of open problems to solve optimal motion planning problems in an efficient way. Historically, these problems have been studied by different people in robotics, control and AI communities. Each community focuses on different aspects of the same problem. The high-dimensionality of the search space and the geometry of the robot are more challenging for robotics problems whereas differential constraints become more difficult to address in many control problems due to under-actuation. Perhaps it is time for a closer collaboration between the different communities in order to unify the strengths and the expertise of the various available approaches.

REFERENCES

[1] AMIDI, O., "Integrated mobile robot control," Tech. Rep. CMU-RI-TR-90-17, Robotics Institute, Carnegie Mellon University, Pittsburgh, Philadelphia, USA, May 1990.

[2] ARSLAN, O., ARMAGAN, B., and INALHAN, G., "Development of a mission simulator for design and testing of C2 algorithms and HMI concepts across real and virtual manned-unmanned fleets," in *Optimization and Cooperative Control Strategies, Lecture Notes in Control and Information Sciences*, vol. 381, pp. 431–458, Springer, 2009.

[3] ARSLAN, O., BERNTORP, K., and TSIOTRAS, P., "Sampling-based algorithms for optimal motion planning using closed-loop prediction," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Stockholm, Sweden), May 16–21 2016. (Note: under review).

[4] ARSLAN, O., DADKHAH, N., MOSER, M., and TSIOTRAS, P., "Development of a route planning system for an autonomous rotorcraft," in *AIAA Atmospheric Flight Mechanics Conference*, (Washington, D.C., USA), June 13–17 2016. (Note: in preparation).

[5] ARSLAN, O. and INALHAN, G., "Design of a decision support architecture for human operators in UAV fleet C2 applications," in *Proceedings of the International Command and Control Research and Technology Symposium*, (Washington, D.C., USA), June 15–17 2009.

[6] ARSLAN, O. and INALHAN, G., "An event driven decision support algorithm for command and control of UAV fleets," in *Proceedings of the IEEE American Control Conference*, (St. Louis, Missouri, USA), pp. 5198–5203, June 10–12 2009.

[7] ARSLAN, O., THEODOROU, E. A., and TSIOTRAS, P., "Information-theoretic stochastic optimal control via incremental sampling-based algorithms," in *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, (Orlando, Florida, USA), pp. 1–8, December 9–12 2014.

[8] ARSLAN, O. and TSIOTRAS, P., "An efficient sampling-based algorithm for motion planning with optimality guarantees," Tech. Rep. DCSL-12-09-010, School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, Georgia, USA, September 2012.

[9] ARSLAN, O. and TSIOTRAS, P., "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *Proceedings of the IEEE International*

Conference on Robotics and Automation, (Karlsrühe, Germany), pp. 2413–2420, May 6–10 2013.

[10] ARSLAN, O. and TSIOTRAS, P., "Machine learning for sampling-based motion planning," in *Amazon Graduate Research Symposium*, (Seattle, Washington, USA), November 7 2014.

[11] ARSLAN, O. and TSIOTRAS, P., "Dynamic programming and machine learning algorithms for motion planning and control," in *Microsoft Research New England Machine Learning Day*, (Cambridge, Massachusetts, USA), May 18 2015.

[12] ARSLAN, O. and TSIOTRAS, P., "Dynamic programming guided exploration for sampling-based motion planning algorithms," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Seattle, Washington, USA), pp. 4819–4826, May 26–30 2015.

[13] ARSLAN, O. and TSIOTRAS, P., "Dynamic programming principles for sampling-based motion planners," in *ICRA Workshop on Optimal Robot Motion Planning*, (Seattle, Washington, USA), May 30 2015.

[14] ARSLAN, O. and TSIOTRAS, P., "Machine learning guided exploration for sampling-based motion planning algorithms," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Hamburg, Germany), September 28–October 02 2015.

[15] ARSLAN, O. and TSIOTRAS, P., "Incremental sampling-based motion planners using policy iteration methods," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Stockholm, Sweden), May 16–21 2016. (Note: under review).

[16] ARSLAN, O., TSIOTRAS, P., and HUO, X., "Solving shortest path problems with curvature constraints using beamlets," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (San Francisco, California, USA), pp. 3533–3538, September 25–30 2011.

[17] ARSLAN, O., ULUALAN, B. G., and INALHAN, G., "Design and implementation of communication and information distribution modules for cooperative unmanned-manned vehicle networks," in *8th International Conference on Cooperative Control and Optimization*, (Gainesville, Florida, USA), January 30–February 1 2008.

[18] ATKESON, C. G., MOORE, A. W., and SCHAAL, S. A., "Locally weighted learning," *Artificial Intelligence Review*, vol. 11(1-5), pp. 11–73, 1997.

[19] BARRAQUAND, J., KAVRAKI, L. E., LATOMBE, J.-C., LI, T., and MOTWANI, R.AND RAGHAVAN, P., "A random sampling scheme for path planning," *The International Journal of Robotics Research*, vol. 16, no. 6, pp. 759–774, 1997.

[20] BELLMAN, R. E., "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[21] BELLMAN, R. E. and DREYFUS, S. E., *Applied Dynamic Programming*. Princeton, New Jersey, USA: Princeton University Press, 1962.

[22] BERENSON, D., KUFFNER, JR., J. J., and CHOSET, H., "An optimization approach to planning for mobile manipulation," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Pasadena, California, USA), pp. 1187–1192, May 19–23 2008.

[23] BERTSEKAS, D. P., *Dynamic Programming and Optimal Control*, vol. 1. Athena Scientific, Belmont, MA, 2000.

[24] BERTSEKAS, D. P., *Abstract Dynamic Programming*. Belmont, Massachusetts, USA: Athena Scientific, 2013.

[25] BERTSEKAS, D. P. and TSITSIKLIS, J., *Parallel and Distributed Computation: Numerical Methods*. Belmont, Massachusetts, USA: Athena Scientific, January 1997.

[26] BHATIA, A. and FRAZZOLI, E., "Incremental search methods for reachability analysis of continuous and hybrid systems," *Hybrid Systems: Computation and Control*, pp. 451–471, 2004.

[27] BIALKOWSKI, J., KARAMAN, S., and FRAZZOLI, E., "Massively parallelizing the RRT and the RRT*," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (San Francisco, California, USA), pp. 3513–3518, September 25–30 2011.

[28] BIALKOWSKI, J., KARAMAN, S., OTTE, M., and FRAZZOLI, E., "Efficient collision checking in sampling-based motion planning," in *Algorithmic Foundations of Robotics X*, pp. 365–380, Springer, 2013.

[29] BISHOP, C., *Pattern Recognition and Machine Learning*. New York: Springer, 2006.

[30] BRANICKY, M. S., CURTISS, M. M., LEVINE, J., and MORGAN, S., "Sampling-based planning, control and verification of hybrid systems," *IEE Proceedings in Control Theory and Applications*, vol. 153, no. 5, pp. 575–590, 2006.

[31] BRANICKY, M. S., CURTISS, M. M., LEVINE, J. A., and MORGAN, S. B., "RRTs for nonlinear, discrete, and hybrid planning and control," in *Proceedings of the IEEE Conference on Decision and Control*, vol. 1, (Maui, Hawaii, USA), pp. 657–663, December 9–12 2003.

[32] BRANICKY, M. S., LAVALLE, S. M., OLSON, K., and YANG, L., "Quasi-randomized path planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 2, (Seoul, Korea), pp. 1481–1487, May 21–26 2001.

[33] BRUCE, J. and VELOSO, M., "Real-time randomized path planning for robot navigation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, (Lausanne, Switzerland), pp. 2383–2388, September 30–October 4 2002.

[34] BUCHLI, J., STULP, F., THEODOROU, E. A., and SCHAAL, S., "Learning variable impedance control," *The International Journal of Robotics Research*, vol. 30, pp. 820–833, June 2011.

[35] BURNS, B. and BROCK, O., "Information theoretic construction of probabilistic roadmaps," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Las Vegas, Nevada, USA), pp. 650–655, October 27–31 2003.

[36] BURNS, B. and BROCK, O., "Sampling-based motion planning using predictive models," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Barcelona, Spain), pp. 3120–3125, April 18–22 2005.

[37] BURNS, B. and BROCK, O., "Single-query motion planning with utility-guided random trees," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Rome, Italy), pp. 3307–3312, April 10–14 2007.

[38] CETINKAYA, A., KARAMAN, S., ARSLAN, O., AKSUGUR, M., and INALHAN, G., "Design of a distributed C2 architecture for interoperable manned/unmanned fleets," in *7th International Conference on Cooperative Control and Optimization*, (Gainesville, Florida, USA), January 31–February 2 2007.

[39] CHOSET, H., LYNCH, K., HUTCHINSON, S., KANTOR, G., BURGARD, W., KAVRAKI, L. E., and THRUN, S., *Principles of Robot Motion: Theory, Algorithms, and Implementations*. The MIT Press, 2005.

[40] COHN, D. A., GHAHRAMANI, Z., and JORDAN, M. I., "Active learning with statistical models," *Journal of Artificial Intelligence Research*, vol. 4, pp. 129–145, 1996.

[41] CORTÉS, J., JAILLET, L., and SIMÉON, T., "Molecular disassembly with RRT-like algorithms," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Roma, Italy), pp. 3301–3306, April 10–14 2007.

[42] CUMMINGS, M. and COLLINS, A., "Autonomous Aerial Cargo Utility System (AACUS) - Concept of Operations (CONOPs)," Tech. Rep. 12-SN-0002, The Office Of Naval Research, October 2011.

[43] DAI PRA, P., MENEGHINI, L., and RUNGGALDIER, W., "Connections between stochastic control and dynamic games," *Mathematics of Control, Signals, and Systems*, vol. 9, pp. 303–326, December 1996.

[44] DIJKSTRA, E. W., "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[45] FINN, P. W. and KAVRAKI, L. E., "Computational approaches to drug design," *Algorithmica*, vol. 25, no. 2-3, pp. 347–371, 1999.

[46] FLEMING, W. H. and MCENEANEY, W. M., "Risk-sensitive control on an infinite time horizon," *SIAM Journal on Control and Optimization*, vol. 33, pp. 1881–1915, November 1995.

[47] FORD, L. R., "Network flow theory," tech. rep., RAND Corporation, Santa Monica, California, USA, 1956.

[48] FRAZZOLI, E., DAHLEH, M. A., and FERON, E., "Real-time motion planning for agile autonomous vehicles," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 25, no. 1, pp. 116–129, 2002.

[49] FREDMAN, M. L. and TARJAN, R. E., "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the Association for Computing Machinery*, vol. 34, no. 3, pp. 596–615, 1987.

[50] FRIEDMAN, A., *Stochastic Differential Equations and Applications*. Dover Books on Mathematics, Dover Publications, December 2006.

[51] GERAERTS, R. and OVERMARS, M. H., "A comparative study of probabilistic roadmap planners," in *Algorithmic Foundations of Robotics V*, pp. 43–57, Springer, 2004.

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

[53] HSU, D., JIANG, T., REIF, J., and SUN, Z., "The bridge test for sampling narrow passages with probabilistic roadmap planners," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Taipei, Taiwan), pp. 4420–4426, September 14–19 2003.

[54] HSU, D., KAVRAKI, L. E., LATOMBE, J.-C., MOTWANI, R., and SORKIN, S., "On finding narrow passages with probabilistic roadmap planners," in *The International Workshop on Algorithmic Foundations of Robotics*, pp. 141–154, 1998.

[55] HSU, D., KINDEL, R., LATOMBE, J.-C., and ROCK, S., "Randomized kinodynamic motion planning with moving obstacles," *The International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, 2002.

[56] HSU, D., LATOMBE, J.-C., and KURNIAWATI, H., "On the probabilistic foundations of probabilistic roadmap planning," *The International Journal of Robotics Research*, vol. 25, no. 7, pp. 627–643, 2006.

[57] HSU, D., LATOMBE, J.-C., and MOTWANI, R., "Path planning in expansive configuration spaces," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 3, (Albuquerque, New Mexico, USA), pp. 2719–2726, April 20–25 1997.

[58] HUYNH, V. A., KARAMAN, S., and FRAZZOLI, E., "An incremental sampling-based algorithm for stochastic optimal control," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (St. Paul, Minnesota, USA), pp. 2865–2872, May 14–18 2012.

[59] IJSPEERT, A. J., NAKANISHI, J., HOFFMANN, H., PASTOR, P., and SCHAAL, S., "Dynamical movement primitives: Learning attractor models for motor behaviors," *Neural Computation*, vol. 25, no. 2, pp. 328–373, 2013.

[60] JACOBSON, D. and MAYNE, D., *Differential Dynamic Programming*. Elsevier, 1970.

[61] JANSON, L., SCHMERLING, E., CLARK, A., and PAVONE, M., "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *The International Journal of Robotics Research*, vol. 34, pp. 883–921, June 2015.

[62] KALAKRISHNAN, M., CHITTA, S., THEODOROU, E. A., PASTOR, P., and SCHAAL, S., "STOMP: Stochastic trajectory optimization for motion planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Shanghai, China), pp. 4569–4574, May 9–13 2011.

[63] KAPPEN, H. J., GÓMEZ, V., and OPPER, M., "Optimal control as a graphical model inference problem," *Machine Learning*, vol. 87, no. 2, pp. 159–182, 2012.

[64] KARAMAN, S. and FRAZZOLI, E., "Optimal kinodynamic motion planning using incremental sampling-based methods," in *Proceedings of the IEEE Conference on Decision and Control*, (Atlanta, Georgia, USA), pp. 7681–7687, December 15–17 2010.

[65] KARAMAN, S. and FRAZZOLI, E., "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, pp. 846–894, June 2011.

[66] KARAMAN, S., WALTER, M. R., PEREZ, A., FRAZZOLI, E., and TELLER, S., "Anytime motion planning using the RRT*," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Shanghai, China), pp. 1478–1483, May 9–13 2011.

[67] KAVRAKI, L. E., KOLOUNTZAKIS, M. N., and LATOMBE, J.-C., "Analysis of probabilistic roadmaps for path planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 166–171, 1998.

[68] KAVRAKI, L. E. and LATOMBE, J.-C., "Randomized preprocessing of configuration for fast path planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (San Diego, California, USA), pp. 2138–2145, May 8–13 1994.

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

[70] KOENIG, S. and LIKHACHEV, M., "D^* lite," in *AAAI Conference on Artificial Intelligence*, (Menlo Park, California, USA), pp. 476–483, American Association for Artificial Intelligence, 2002.

[71] KOENIG, S., LIKHACHEV, M., and FURCY, D., "Lifelong planning A*," *Artificial Intelligence J*, vol. 155, no. 1–2, pp. 93–146, 2004.

[72] KUFFNER, JR., J. J., KAGAMI, S., NISHIWAKI, K., INABA, M., and INOUE, H., "Dynamically-stable motion planning for humanoid robots," *Autonomous Robots*, vol. 12, no. 1, pp. 105–118, 2002.

[73] KUFFNER, JR., J. J. and LAVALLE, S. M., "RRT-connect: An efficient approach to single-query path planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (San Francisco, California, USA), pp. 995–1001, April 24–28 2000.

[74] KUNZ, T., *Time-Optimal Sampling-Based Motion Planning for Manipulators with Acceleration Limits*. College of Computing, Georgia Institute of Technology, 2015.

[75] KUWATA, Y., KARAMAN, S., TEO, J., FRAZZOLI, E., HOW, J. P., and FIORE, G., "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, 2009.

[76] KUWATA, Y., TEO, J., KARAMAN, S., FIORE, G., FRAZZOLI, E., and HOW, J. P., "Motion planning in complex environments using closed-loop prediction," in *AIAA Guidance, Navigation, and Control Conference*, (Honolulu, Hawaii, USA), August 18–21 2008.

[77] LADD, A. M. and KAVRAKI, L. E., "Measure theoretic analysis of probabilistic path planning," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 2, pp. 229–242, 2004.

[78] LATOMBE, J.-C., *Robot Motion Planning*. Boston, Massachusetts, USA: Kluwer Academic Publishers, 1991.

[79] LATOMBE, J.-C., "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *The International Journal of Robotics Research*, vol. 18, pp. 1119–1128, November 1999.

[80] LAVALLE, S. M., "Rapidly-exploring random trees: A new tool for path planning," Technical Report TR 98-11, Computer Science Department, Iowa State University, October 1998.

[81] LAVALLE, S. M., *Planning Algorithms*. New York: Cambridge University Press, 2006.

[82] LAVALLE, S. M. and KUFFNER, JR., J. J., "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, pp. 378–400, May 2001.

[83] LaValle, S. M. and Kuffner, Jr., J. J., "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions* (Donald, B. R., Lynch, K. M., and Rus, D., eds.), pp. 293–308, A. K. Peters, Wellesley, 2001.

[84] Leonard, J. and others, "A perception-driven autonomous urban vehicle," *Journal of Field Robotics*, vol. 25, no. 10, pp. 727–774, 2008.

[85] Levine, S., *Motor Skill Learning with Local Trajectory Methods*. Department of Computer Science, Stanford University, 2014.

[86] Likhachev, M., Gordon, G. J., and Thrun, S., "ARA*: Anytime A* with provable bounds on sub-optimality," in *Advances in Neural Information Processing Systems*, pp. 8–13, December 2003.

[87] Lindemann, S. R. and LaValle, S. M., "Current issues in sampling-based motion planning," *Springer Tracts in Advanced Robotics*, pp. 36–54, 2005.

[88] Liu, Y. and Badler, N. I., "Real-time reach planning for animated characters using hardware acceleration," in *IEEE International Conference on Computer Animation and Social Agents*, pp. 86–93, 2003.

[89] Lozano-Perez, T., "Spatial planning: A configuration space approach," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 108–120, 1983.

[90] Lu, Y., Huo, X., Arslan, O., and Tsiotras, P., "Incremental multi-scale search algorithm for dynamic path planning with low worst-case complexity," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 41, no. 6, pp. 1556–1570, 2011.

[91] Lu, Y., Huo, X., Arslan, O., and Tsiotras, P., "Multi-scale LPA* with low worst-case complexity guarantees," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (San Francisco, California, USA), pp. 3507–3512, September 25–30 2011.

[92] Luders, B., Kothari, M., and How, J. P., "Chance constrained RRT for probabilistic robustness to environmental uncertainty," in *AIAA Guidance, Navigation, and Control Conference*, (Toronto, Canada), August 2010.

[93] Nilsson, N. J., *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill Inc., 1971.

[94] Øksendal, B., *Stochastic Differential Equations : An Introduction with Applications*. Berlin; New York: Springer, 6th ed., 2003.

[95] Otte, M. and Frazzoli, E., "RRT-X: Real-Time motion planning/replanning for environments with unpredictable obstacles," in *The International Workshop on Algorithmic Foundations of Robotics*, pp. 461–478, Istanbul, Turkey: Springer, 2014.

[96] PAN, J., CHITTA, S., and MANOCHA, D., "Faster sample-based motion planning using instance-based learning," in *Algorithmic Foundations of Robotics X*, pp. 381–396, Springer, 2013.

[97] PEARL, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Pub (Sd), 1984.

[98] PENROSE, M., *Random Geometric Graphs*, vol. 5. Oxford University Press, 2003.

[99] PETERS, J., MÜLLING, K., and ALTUN, Y., "Relative entropy policy search.," in *Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence*, 2010.

[100] PONTRYAGIN, L. S., *Mathematical Theory of Optimal Processes*. Pergamon Press, 1964.

[101] PRENTICE, S. and ROY, N., "The belief roadmap: Efficient planning in belief space by factoring the covariance," *The International Journal of Robotics Research*, vol. 28, November–December 2009.

[102] QUIGLEY, M., CONLEY, K., GERKEY, B., FAUST, J., FOOTE, T., LEIBS, J., WHEELER, R., and NG, A., "ROS: An open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, p. 5, 2009.

[103] RATLIFF, N., ZUCKER, M., BAGNELL, J. A., and SRINIVASA, S. S., "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Kobe, Japan), pp. 489–494, May 12–17 2009.

[104] REIF, J. H., "Complexity of the mover's problem and generalizations," in *Proceedings of the IEEE Conference on Foundations of Computer Science*, pp. 421–427, 1979.

[105] REIF, J. H., *Complexity of the Generalized Mover's Problem*. Aiken Computation Laboratory, Harvard University, 1985.

[106] SAMET, H., *Foundations of Multidimensional and Metric Data Structures*. Amsterdam Boston: Morgan Kaufmann, 2006.

[107] SCHWARTZ, J. T. and SHARIR, M., "On the piano movers' problem : I. the case of a two-dimensional rigid polygonal body moving amidst polygonal barriers," *Communications on Pure and applied Mathematics*, vol. 36, no. 3, pp. 345–398, 1983.

[108] SCHWARTZ, J. T. and SHARIR, M., "On the piano movers' problem. II. general techniques for computing topological properties of real algebraic manifolds," *Advances in Applied Mathematics*, vol. 4, no. 3, pp. 298–351, 1983.

[109] SHIMRAT, M., "Algorithm 112: position of point relative to polygon," *Communications of the Association for Computing Machinery (ACM)*, vol. 5, no. 8, p. 434, 1962.

[110] SILVERMAN, B. W., *Density Estimation for Statistics and Data Analysis*. London New York: Chapman and Hall, 1986.

[111] STENGEL, R. F., *Optimal Control and Estimation*. New York: Dover Publications, 1994.

[112] STENTZ, A., "The focussed D* algorithm for real-time replanning.," in *International Joint Conference on Artificial Intelligence*, vol. 95, (Montreal, Canada), pp. 1652–1659, August 1995.

[113] STILMAN, M., SCHAMBUREK, J., KUFFNER, JR., J. J., and ASFOUR, T., "Manipulation planning among movable obstacles," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Rome, Italy), pp. 3327–3332, April 10–14 2007.

[114] STULP, F., THEODOROU, E. A., and SCHAAL, S., "Reinforcement learning with sequences of motion primitives for robust manipulation," *IEEE Transactions on Robotics*, vol. 28, no. 6, pp. 1360–1370, 2012.

[115] TEDRAKE, R., MANCHESTER, I. R., TOBENKIN, M., and ROBERTS, J. W., "LQR-trees: Feedback motion planning via sums-of-squares verification," *The International Journal of Robotics Research*, vol. 29, pp. 1038–1052, July 2010.

[116] TELLER, S. and OTHERS, "A voice-commandable robotic forklift working alongside humans in minimally-prepared outdoor environments," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Anchorage, Alaska, USA), pp. 526–533, May 3–9 2010.

[117] THEODOROU, E. A., BUCHLI, J., and SCHAAL, S., "A generalized path integral approach to reinforcement learning," *Journal of Machine Learning Research*, no. 11, pp. 3137–3181, 2010.

[118] THEODOROU, E. A., KRISHNAMURTHY, D., and TODOROV, E., "From information theoretic dualities to path integral and kullback-leibler control: Continuous and discrete time formulations," in *The Sixteenth Yale Workshop on Adaptive and Learning Systems*, (New Haven, Connecticut, USA), June 5–7 2013.

[119] THEODOROU, E. A. and TODOROV, E., "Relative entropy and free energy dualities: Connections to path integral and kl control," in *Proceedings of the IEEE Conference on Decision and Control*, (Maui, Hawaii, USA), pp. 1466–1473, December 10–13 2012.

[120] TODOROV, E., "Linearly-solvable markov decision problems," in *Advances in Neural Information Processing Systems*, pp. 1369–1376, 2006.

[121] TODOROV, E., "Efficient computation of optimal actions," *Proceedings of the National Academy of Sciences*, vol. 106, no. 28, pp. 11478–11483, 2009.

[122] YANG, J. and KUSHNER, J. H., "A Monte Carlo method for sensitivity analysis and parametric optimization of nonlinear stochastic systems," *SIAM Journal in Control and Optimization*, vol. 29, no. 5, pp. 1216–1249, 1991.

[123] YERSHOVA, A., JAILLET, L., SIMÉON, T., and LAVALLE, S. M., "Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Barcelona, Spain), pp. 3856–3861, April 18–22 2005.

[124] ZUCKER, M., KUFFNER, JR., J. J., and BRANICKY, M. S., "Multipartite RRTs for rapid replanning in dynamic environments," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Rome, Italy), pp. 1603–1609, April 10–14 2007.

VITA

Oktay Arslan was born to a Kurdish family in Istanbul, Turkey. He obtained his B.S. degrees in Electrical Engineering (with specialization in Control Engineering) in 2006 and Computer Engineering in 2007, and M.S. degree in Defense Technologies in 2009 all from Istanbul Technical University, Turkey. He was on the high honor list and graduated with the first rank in Department of Electrical Engineering. He was the recipient of numerous awards during his education including the Control Engineering Best Student Award given by Istanbul Technical University, Werner von Siemens Excellence Award given by Siemens AG, Graduate Research Fellowship given by The Scientific and Technological Research Council of Turkey, and EducationUSA Opportunity Grant given by The Turkish Fulbright Commission. He also obtained additional M.S. degrees in Aerospace Engineering in 2012 and Computer Science in 2015 from Georgia Institute of Technology, Atlanta, Georgia while working towards his doctoral degree in Robotics.

He worked as a Guidance, Navigation, and Control Engineering Intern at the Aurora Flight Sciences Corporation from June to August 2013, following which he was hired part-time as a Robotics/Software Engineer at the same company from August 2014 to May 2015 where he contributed to the route planning effort for the Autonomous Aerial Cargo Utility System (AACUS) project. During the same time interval, he was a visiting researcher at Laboratory for Information and Decision Systems at Massachusetts Institute of Technology. Lastly, he worked as a Research Intern with the Mechatronics Group at the Mitsubishi Electric Research Laboratories, where he worked on motion planning and control for dynamical systems.

His research interests are autonomy, perception, planning, learning and control for autonomous systems and their applications to aerospace and robotics.