Sampling-Based Motion Planning with μ-Calculus Specifications

by

Luc Larocque

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Bruce Bruce Professor, Dept. of Philosophy of Zoology, University of Wallamaloo
Supervisor:	Jun Liu Assistant Professor, Dept. of Applied Mathematics, University of Waterloo
Internal Member:	Pamela Python Professor, Dept. of Zoology, University of Waterloo
Internal-External Member:	Deepa Thotta Professor, Dept. of Philosophy, University of Waterloo

Abstract

This is the abstract.

Vulputate minim vel consequat praesent at vel iusto et, ex delenit, esse euismod luptatum augue ut sit et eu vel augue autem feugiat, quis ad dolore. Nulla vel, laoreet lobortis te commodo elit qui aliquam enim ex iriure ea ullamcorper nostrud lorem, lorem laoreet eu ex ut vel in zzril wisi quis. Nisl in autem praesent dignissim, sit vel aliquam at te, vero dolor molestie consequat.

Tation iriure sed wisi feugait odio dolore illum duis in accumsan velit illum consequat consequat ipsum molestie duis duis ut ullamcorper. Duis exerci odio blandit vero dolore eros odio amet et nisl in nostrud consequat iusto eum suscipit autem vero. Iusto dolore exerci, ut erat ex, magna in facilisis duis amet feugait augue accumsan zzril delenit aliquip dignissim at. Nisl molestie nibh, vulputate feugait nibh luptatum ea delenit nostrud dolore minim veniam odio volutpat delenit nulla accumsan eum vero ullamcorper eum. Augue velit veniam, dolor, exerci ea feugiat nulla molestie, veniam nonummy nulla dolore tincidunt, consectetuer dolore nulla ipsum commodo.

At nostrud lorem, lorem laoreet eu ex ut vel in zzril wisi. Suscipit consequat in autem praesent dignissim, sit vel aliquam at te, vero dolor molestie consequat eros tation facilisi diam dolor. Odio luptatum dolor in facilisis et facilisi et adipiscing suscipit eu iusto praesent enim, euismod consectetuer feugait duis. Odio veniam et iriure ad qui nonummy aliquip at qui augue quis vel diam, nulla. Autem exerci tation iusto, hendrerit et, tation esse consequat ut velit te dignissim eu esse eros facilisis lobortis, lobortis hendrerit esse dignissim nisl. Nibh nulla minim vel consequat praesent at vel iusto et, ex delenit, esse euismod luptatum.

Ut eum vero ullamcorper eum ad velit veniam, dolor, exerci ea feugiat nulla molestie, veniam nonummy nulla. Elit tincidunt, consectetuer dolore nulla ipsum commodo, ut, at qui blandit suscipit accumsan feugiat vel praesent. In dolor, ea elit suscipit nisl blandit hendrerit zzril. Sit enim, et dolore blandit illum enim duis feugiat velit consequat iriure sed wisi feugait odio dolore illum duis. Et accumsan velit illum consequat consequat ipsum molestie duis duis ut ullamcorper nulla exerci odio blandit vero dolore eros odio amet et.

In augue quis vel diam, nulla dolore exerci tation iusto, hendrerit et, tation esse consequat ut velit. Duis dignissim eu esse eros facilisis lobortis, lobortis hendrerit esse dignissim nisl illum nulla minim vel consequat praesent at vel iusto et, ex delenit, esse euismod. Nulla augue ut sit et eu vel augue autem feugiat, quis ad dolore te vel, laoreet lobortis te commodo elit qui aliquam enim ex iriure. Ut ullamcorper nostrud lorem, lorem laoreet eu ex ut vel in zzril wisi quis consequat in autem praesent dignissim, sit vel. Dolore at te, vero

dolor molestie consequat eros tation facilisi diam. Feugait augue luptatum dolor in facilisis et facilisi et adipiscing suscipit eu iusto praesent enim, euismod consectetuer feugait duis vulputate veniam et.

Ad eros odio amet et nisl in nostrud consequat iusto eum suscipit autem vero enim dolore exerci, ut. Esse ex, magna in facilisis duis amet feugait augue accumsan zzril. Lobortis aliquip dignissim at, in molestie nibh, vulputate feugait nibh luptatum ea delenit nostrud dolore minim veniam odio. Euismod delenit nulla accumsan eum vero ullamcorper eum ad velit veniam. Quis, exerci ea feugiat nulla molestie, veniam nonummy nulla. Elit tincidunt, consectetuer dolore nulla ipsum commodo, ut, at qui blandit suscipit accumsan feugiat vel praesent.

Dolor zzril wisi quis consequat in autem praesent dignissim, sit vel aliquam at te, vero. Duis molestie consequat eros tation facilisi diam dolor augue. Dolore dolor in facilisis et facilisi et adipiscing suscipit eu iusto praesent enim, euismod consectetuer feugait duis vulputate.

Acknowledgements

Many thanks to my ever-supportive and helpful supervisor, Jun Liu. Your positivity and guidance gave me the motivation I needed to succeed, while leaving me with the independence to choose my own research path. Thank you also to all of the members of the Hybrid Systems Lab. Yinan Li, Milad Farsi, Chuanzheng Wang, Riley Brooks, and Kevin Church, you have all contributed so much to my Master's experience with insightful conversation, interesting presentations, and utmost kindness.

Dedication

This is dedicated to my parents and to Maša: you have made my life an absolute pleasure during my graduate studies.

Table of Contents

List of Tables

List of Figures

Abbreviations

CTL	Computation Tree Logic 5, 12, 14, 28
EST	Expansive Space Trees 16
FMT*	Fast Marching Tree 20, 26, 41
LQR	Linear Quadratic Regulator 29
LTL	Linear Temporal Logic 5, 12, 14, 28, 31
NED	North-East-Down 41
OBVP	optimal boundary value problem 15, 29, 30
PRM	Probabilistic Roadmap 16, 20
RRT	Rapidly-exploring Random Tree 4, 16, 20
SLAM	simultaneous localization and mapping 3
SST	Stable Sparse RRT 15, 25, 29
UAV	unmanned aerial vehicle 2, 3

Chapter 1

Introduction

1.1 Motion Planning

Planning is a fundamental problem in robotics: mobile robots must be able to determine how to move in order to perform tasks. According to LaValle in his titular book on planning algorithms, converting high-level specifications into low-level descriptions of how a robot ought to move is what is generally referred to as motion planning [19]. He further states motion planning, in modern control theory literature, refers to the generation of inputs to a dynamical system which drive it from an initial state to a specified goal state (or set).

In general, motion planning solves problems involving a *state space*, which is the set of all possible states in which a system could find itself. Such a space could be finite, like in the case of Rubik's cube with finitely many configurations, or infinite, such as train with both a position and velocity that can vary continuously in the domain of real numbers. Note that *time* also plays a crucial role in both of these examples: the Rubik's cube allows moves in succession, in some order, and the train's location and current velocity depend on its past position and velocity. Lastly, in order to plan, one must be able to affect the system, i.e., change the state, in the some way. Some systems, like the train, abide by a set of dynamics which govern how the system changes. A train on a steep hill will roll down the hill if it does not have sufficient momentum to crest over the top. However, if we allow the system to accept an *input* or *control*, then the system can be altered to act in a desirable way. Being able to set the engine to full-throttle may make the difference between getting to the destination and ending up stuck at the foot of the hill. For continuous time systems, the dynamics are modeled with ordinary differential equations. Even for systems without dynamics, like with the Rubik's cube (assuming we are not concerned with how the faces of the cube are rotated), there must be a way to specify exactly how an action affects the state of the system.

This thesis will focus only on continuous-time motion planning problems, where the dynamics are modeled by a control system consists of a set of ordinary differential equations modeling the dynamic of the system, an initial condition, a set of allowed states, and a set of admissible controls. Note that this does not rule out the possibility of uncertainties. Dynamical systems model reality but do not necessarily do so with complete accuracy, and disturbances in the environment may also have an effect on the behaviour of a system. A well-designed motion planning algorithm can help to reject disturbances and be robust under uncertainties.

1.1.1 Examples

Motion planning has an immense variety of applications in both virtual and robotic systems. Moreover, the potential for autonomous robots to improve human living conditions is vast, and as yet not fully understood. One especially disruptive emerging technology is autonomous vehicles: cars and trucks that are able to drive from an initial position to a goal location with minimal or no human input [29]. These autonomous vehicles have been gaining popularity in recent years, especially with the media hype of Google and Tesla bringing self-driving cars into the public eye; however, the first research on this topic began in the 20th century. By 1995, Todd Jochem and Dean Pomerleau completed a 2,797 mile journey across America in a van using neural networks to design a vision-based partially automated driving system[1] [12]. This accomplishment demonstrated level 2 automation under the SAE International Standard J3106 [9], which falls short of being described as an "automated driving system" as a human driver is still essential. More recent advances have brought autonomous vehicles to level 3 with Google's self-driving car having over 500,000 miles of autonomous driving in 2012 [24]. Level 3 is labeled as "conditional automation", meaning for certain driving modes the automated driving system can control all aspects of driving with the caveat that a human driver be on standby to respond to requests to intervene. In jumping from level 2 to level 3, a human driver becomes non-essential to the driving task, except as a fallback.

Another practical example of motion planning is controlling an unmanned aerial vehicle (UAV) system, such as a quadrotor. Due to their scalable size and high maneuverability, quadrotors are used for an ever-increasing range of tasks, from aerial photography, to

[1]The trip was titled "No Hands Across America" since the only human input involved braking and accelerating; steering was performed completely autonomously.

mapping dangerous, cluttered, or unexplored regions with simultaneous localization and mapping (SLAM), to light shows from a fleet of Intel's quadrotors, and even for quickly delivering small parcels [35, 25]. Each of these tasks requires a means of determining where and how the quadrotor should move, and advances in motion planning will allow for even more complex and intricate maneuvers, and therefore more applications. However, planning for quadrotors can be quite difficult because, like cars, they are non-holonomic, and are therefore subject to differential constraints. Since they can maneuver in 3D-space instead of being restricted to a 2D plane, quadrotors are described by a 12-dimensional state space (position, velocity, orientation, and rotational velocity). This often means that much computational effort is involved in motion planning for UAV systems. Furthermore, given that the dynamics governing quadrotor motion are highly nonlinear, motion planning is all the more difficult, from path generation to trajectory tracking. Many of these issues will be addressed in Chapter 4.

Overall, the field of robotics is continuing to grow, and with it, motion planning is becoming all the more relevant and important in day-to-day life. Forklifts are being automated to move merchandise around in warehouses, and vacuum cleaners have become small disks that roam around the home, like small robotic pets. In the field of agriculture, robots are improving the lives of farmers by autonomously targeting and removing weeds that negatively impact crops [31]. The recurring theme in all of these examples is that automation, along with the essential component of motion planning, is leading the way in reducing the need for human labour. Robots are becoming increasingly capable of performing complex tasks, especially with the concurrent rise of machine learning and artificial intelligence, and advances in motion planning are leading to improved performance [6], online reactivity to dynamic obstacles [1], and better guarantees [23] in all areas of robotics.

1.1.2 Sampling-Based Motion Planning

The sampling-based approach to motion planning has become particularly popular in robotics applications since it avoids having to explicitly define obstacles (dually, the free space). In essence, the strategy is to sample the state space in order to create a graph which explores a representative portion of the space. Such problems can be solved geometrically, meaning solutions only find a collision-free path in space and do not take into account feasibility, or using kinodynamic planning, which considers the system dynamics and differential constraints to ensure the robot can accomplish the planned task. In this thesis, emphasis is placed on kinodynamic planning.

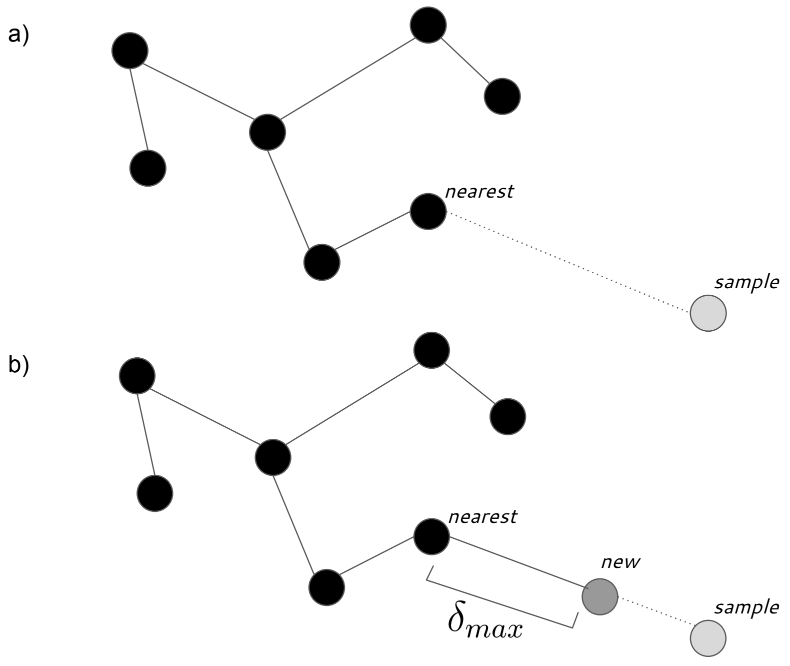

Figure 1.1: RRT sampling and connection procedure. The existing tree is shown with black nodes and solid edges. a) A node is randomly sampled from the state space, and the nearest existing node is found. b) A new node and edge are added to the tree along the line connecting the nearest and sampled nodes, but at a maximum distance δ_{max} from the nearest node.

The general approach to sampling-based motion planning begins with choosing a sampling scheme; that is, samples of the state space are to be taken either deterministically, or according to some probability distribution. These samples are then used to generate new nodes on a graph, where the details involved in connecting each new node to the existing graph vary based on the sampling-based algorithm being applied. For example, as depicted in Figure 1.1, the popular motion planning algorithm Rapidly-exploring Random Tree (RRT) samples a random point in state space, finds the nearest existing point of the tree, and steers from said nearest point towards the sample up to some maximum distance, δ_{max}. The result is a newly added edge (transition from one state to another) and node (state) in the tree.

An important concept in motion planning literature is *completeness*, and we say that an algorithm is complete if, for any input, it correctly returns in finite time whether or not there exists a solution to the path planning problem [19]. Clearly, due to the nature of sampling-based algorithms, completeness cannot be achieved. There are, however, weaker notions of completeness which can be useful to study. One common alternative when using a random sampling scheme is the the notion of *probabilistic completeness*, which means that the probability that the algorithm finds a solution (if one exists) converges to 1 as the number of samples tends to infinity. Similarly, we that a motion planning algorithm is *asymptoticallyoptimal* if the probability of finding an optimal solution (based on a pre-established cost function) approaches 1 as the number of samples approaches infinity [14].

1.2 Temporal Logic

Motion planning occurs on various levels of abstraction. A hierarchical control structure would typically have some "vague", high-level description of what the robot or system is supposed to do at the top, guiding the desired behaviour. Below that there is a model, usually encompassing system dynamics, with a control architecture that prescribes how information is passed along in a series of inputs and outputs, which ultimately generates a motion plan to be followed. At the lowest levels, algorithms are used to put the motion plan in action, using a tracking controller (typically with some form of feedback to improve robustness) that determines exactly what inputs ought to be applied to maneuver along the planned path trajectory.

Temporal logics are the language used to express the high-level specifications that preside over the control hierarchy, dictating what the user desires. Most instances of motion planning seek simply to move from one location to another, without any other instructions (except to avoid obstacles). This goal is often hard-coded into the algorithms designed for solving motion planning problems. However, there exists a much broader world of possibility, and allowing users to specify exactly what is desired of a robot opens many avenues for real-world applications, not to mention the benefit of having performance guarantees from the use of formal methods [23]. We choose to work with μ-calculus is a highly expressive temporal logic which permits more diverse and complex specifications than the most widely used temporal logics, including Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and extensions thereof [13], as well as for the relative ease and elegance of model checking that it permits. An in-depth background on μ-calculus is provided Chapter 2.

Chapter 2

Preliminaries

In this chapter, we introduce some core background concepts that arise throughout the thesis. First, we detail the notation and specify the semantics of a our choice of temporal logic, μ-calculus, and provide the model checking tools necessary to use it. A description of the incremental kinodynamic planning algorithm SST* follows, along with a description of the various functions used to implement the planning algorithm. This is contrasted with FMT*, which is another motion planning algorithm that is not incremental.

2.1 μ-Calculus

We begin by defining the syntax and semantics of the full, modal μ-calculus. A simple example Kripke structure is provided to build some intuition surrounding the notation and meaning of some common μ-calculus formulas. We then describe a fragment of μ-calculus called deterministic μ-calculus, which we will be using for the motion planning procedure in Chapter 3. Finally, we present many examples of typical deterministic μ-calculus formulas, describing how to parse and thoroughly understand each one.

2.1.1 Modal μ-Calculus

First, an atomic proposition is a declarative statement that is either true or false, and which cannot be further split into smaller statements. An example of an atomic proposition might be “in free space”, where a state satisfies the atomic proposition if and only if it lies in the defined free space for a problem. On the other hand, the statement “in free space AND not in goal region" is not an atomic proposition, as it can be further deconstructed into the simpler statements "in free space" and "in goal region". As this example demonstrates, more complex statements use logical connectives such as conjunction ($\wedge$, logical AND), disjunction ($\vee$, logical OR), and negation ($\neg$, logical NOT). We will also be using the modal operators $\Diamond$ and $\Box$, and the symbols μ and ν as least and greatest fixed point operators, respectively. Definitions for these fixed point operators will be provided.

Let Π be the set of atomic propositions, and let $\textsc{Var}$ be the set of variables. Then we define the following structure as in [13].

Definition 1. A *Kripkestructure* K over the set of atomic propositions Π is a tuple $(S, S_0, R, \mathcal{L})$ where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a binary relation which indicates a means of transiting from one state to another (usually represented as the edges in a directed graph), and $\mathcal{L} : S \to 2^{\Pi}$ is a labeling function, mapping each state to the subset of propositions that it satisfies. (see Figure 2.1)

In essence, a Kripke structure is a graph with edges representing available transitions between nodes, which represent states that have been added to the graph via some sampling scheme. The distinction to be made between a Kripke structure and any such graph representation is that every node on the graph has an associated label which defines which of the previously defined atomic propositions

Define the set of valid μ-calculus formulas (L_μ) inductively, as follows:

- the symbols `True` and `False` are formulas
- every $p \in \Pi$ and $X \in \textsc{Var}$ is a formula
- if ϕ, ψ are formulas, then $\neg\phi$, $\phi \wedge \psi$, and $\phi \vee \psi$ are also formulas
- if ϕ is a formula then $\Diamond\phi$ and $\Box\phi$ are formulas
- if $\phi[X]$ is a formula where $\phi[X]$ is syntactically monotone in X, then $\mu X.\phi$ and $\nu X.\phi$ are formulas.

As in [7], we write $\phi[X]$ to indicate that ϕ may contain an occurrence of X.

Next, we state definitions that will prove to be useful in the discussion of model checking for our chosen fragment of μ-calculus, deterministic μ-calculus, which will be seen in Subsection 2.1.3.

Definition 2. Given a μ-calculus specification, ϕ, We say that a variable X is *positive* (respectively, *negative*) if it occurs under the scope of an even (respectively, odd) number of negations in ϕ.

Definition 3. A subformula X is *pure* in μ-calculus specification ϕ if all of its occurrences have the same polarity (i.e., all occurrences of X are positive or all occurrences of X are negative).

Definition 4. The formula $\phi[X]$ is *syntacticallymonotone* in X if and only if X occurs with pure polarity in ϕ.

μ-Calculus formulas are interpreted with respect to a Kripke structure $K = (S, S_0, R, \mathcal{L})$ and an *environment* (also called an *evaluation*), $e : \textsc{Var} \to 2^S$, which initializes (i.e., assigns a subset of S to) each free variable, where a free variable is defined in the usual sense and a variable is otherwise said to be bound by a fixed point operator. For every L_μ formula ϕ, given a Kripke structure, K, define $[\![\phi]\!]^e_K \subseteq S$ to be the set of states of S satisfying proposition ϕ (note that the subscript K is often omitted when the Kripke structure being used is clear, and the superscript e is omitted when ϕ contains no free variables). The semantics of the formulas is determined inductively by the following, where $p \in \Pi$, $X \in \textsc{Var}$, and $\phi, \psi \in L_\mu$ are arbitrary formulas [32].

$$[\![\texttt{False}]\!]_K = \emptyset$$
$$[\![\texttt{True}]\!]_K = S$$
$$[\![p]\!]_K = \{s \in S : p \in \mathcal{L}(s)\}$$
$$[\![\neg p]\!]_K = S \setminus [\![p]\!]$$
$$[\![\phi \vee \psi]\!]^e_K = [\![\phi]\!]^e_K \cup [\![\psi]\!]^e_K$$
$$[\![\phi \wedge \psi]\!]^e_K = [\![\phi]\!]^e_K \cap [\![\psi]\!]^e_K$$
$$[\![\Diamond\phi]\!]^e_K = \mathrm{Pre}^e_{K,\exists}(\phi)$$
$$[\![\Box\phi]\!]^e_K = \mathrm{Pre}^e_{K,\forall}(\phi)$$
$$[\![\mu X.\phi]\!]^e_K = \bigcap\{A \subseteq S : [\![\phi]\!]^{e[X \leftarrow A]}_K \subseteq A\}$$
$$[\![\nu X.\phi]\!]^e_K = \bigcup\{A \subseteq S : [\![\phi]\!]^{e[X \leftarrow A]}_K \supseteq A\}$$

The existential and universal *predecessor* functions $\mathrm{Pre}^e_{K,\,\cdot} : L_\mu \to S$ map a μ-calculus formula to the set of states which immediately precede (i.e., have transitions to) states satisfying said formula in the Kripke structure K and under evaluation e:

$$\mathrm{Pre}^e_{K,\exists}(\phi) := \{s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \wedge s' \in [\![\phi]\!]^e_K\}$$
$$\mathrm{Pre}^e_{K,\forall}(\phi) := \{s \in S : \forall s' \in S, (s, s') \in R \implies s' \in [\![\phi]\!]^e_K\}.$$

In words, `True` $= (p \vee \neg p)$ holds for all states in S, and `False` $= (p \wedge \neg p)$ does not hold for any state in S; disjunction and conjunction of formulas is equivalent to the union and intersection of the sets which satisfy them, respectively; $\Diamond$ is the *existential successor* (or "next") operator, and $\Box$ is the *universal successor* operator; lastly, μ and ν are the *least* and *greatest fixed-point* operators, respectively, where $e[X \leftarrow A]$ is a modified evaluation function which maps X to A, i.e., $e[X \leftarrow A](X) = A$. To help build a more intuitive understanding of these last four semantic definitions, a simple example is provided (Example 1).

Example 1. Refer to Figure 2.2.
Define atomic proposition p to be the boolean value corresponding to "in gray region". Then we can specify the labeling function for each node, writing $\mathcal{L}(s_p) = \{p\}$ for nodes $s_p \in \{a, b, c\}$, and $\mathcal{L}(s) = \emptyset$ for nodes $s \in \{d, e, f, g\}$. Note that the environment superscript e is omitted as we are not using any free variables.

$$[\![p]\!]_K = \{s \in S : p \in \mathcal{L}(s)\} = \{a, b, c\}$$
$$[\![\neg p]\!]_K = S \setminus [\![p]\!] = \{\text{d, e, f, g }\}$$
$$[\![\Diamond p]\!]_K = \text{Pre}^e_{K,\exists}(p) = \{a, c, d, f\}$$
$$[\![\Box p]\!]_K = \text{Pre}^e_{K,\forall}(p) = \{\text{a, c}\}$$
$$[\![\mu X.(p \vee \Diamond X)]\!]_K = \bigcap\{A \subseteq S : [\![\phi]\!]^{e[X\leftarrow A]}_K \subseteq A\} = \{a, b, c, d, f\}$$
$$[\![\nu X.(p \wedge \Diamond X)]\!]_K = \bigcup\{A \subseteq S : [\![\phi]\!]^{e[X\leftarrow A]}_K \supseteq A\} = \{a, c\}$$

- $[\![p]\!]_K$ is the set of nodes satisfying p (nodes in the gray region).
- $[\![\neg p]\!]_K$ is the complement of $[\![p]\!]_K$ in S (nodes that are not in the gray region).
- $[\![\Diamond p]\!]_K$ contains those nodes that have a transition to a node satisfying p.
- $[\![\Box p]\!]_K$ contains only nodes such that every transition leads to a node satisfying p.
- $[\![\mu X.(p \vee \Diamond X)]\!]_K$ is the "Reachability" specification (see Specification Examples (i)). In this case, the resulting set consists of the nodes that are in the gray region *or* that can take transitions to eventually reach a node in the gray region.
- $[\![\nu X.(p \wedge \Diamond X)]\!]_K$ is the "Safety" specification (see Specification Examples (ii)). In this case, the resulting set consists of the nodes that are in the gray region *and* can always take a transition to a node that is also in the gray region.

We proceed by discussing the details of implementing a general model checking algorithm. In order to develop a means to evaluate expressions containing fixed-point operators, we rely on the Tarski-Knaster theorem from set theory. The next section introduces some definitions before stating the required theorem.

2.1.2 Tarski-Knaster Theorem

The Tarski-Knaster fixed-point theorem (item 1) we introduce in this section makes an important statement about complete lattices and their fixed-points [27]. We make use of this theorem to formulate an algorithm which computes the least or greatest fixed points for the set of states satisfying a μ-calculus proposition containing a least or greatest fixed point operator. Note that if a proposition does not contain fixed point operators or the existential successor operator, it is easy to evaluate the set of states satisfying such a proposition. We begin by defining the necessary concepts.

Definition 5. Let $(X, \leq)$ be a partially ordered set, and let $A \subseteq X$. Then $\bigvee A$ ($\bigwedge A$) denotes the least upper bound (respectively, greatest lower bound) of A with respect to $\leq$, if it exists. We say that X is a *completelattice* if, for every $A \subseteq X$, then both $\bigvee A$ and $\bigwedge A$ exist in X.

Example 2. Consider $(\mathcal{P}(X), \subseteq)$ for any set X, where $\mathcal{P}(X)$ denotes the power set of X. Note that $\forall A \subseteq \mathcal{P}(X)$, $\bigvee A = \bigcup A$, since the union of all the elements of A gives the smallest set which completely contains the elements of each subset in A, and this union must be an element of $\mathcal{P}(X)$. Similarly, $\bigwedge A = \bigcap A \in \mathcal{P}(X)$. Thus, $(\mathcal{P}(X), \subseteq)$ is a complete lattice.

Definition 6. Let $(A, \leq_A)$ and $(B, \leq_B)$ be partially ordered sets. A function $f : A \to B$ is *monotone* if $a_1 \leq_A a_2 \implies f(a_1) \leq_B f(a_2)$.
A point $a \in A$ is a *fixedpoint* of a function $f : A \to B$ if $f(a) = a$, and we denote the set of all fixed points of f by $\textsc{fix}(f)$.

Theorem 1. (Tarski-Knaster Fixed Point Theorem)
Let $\mathbb{L}$ be a complete lattice and let $F : \mathbb{L} \to \mathbb{L}$ be monotone. Then

1. $\bigvee\{x \in \mathbb{L} \mid x \leq F(x)\} \in \textsc{fix}(F)$,
2. $\bigwedge\{x \in \mathbb{L} \mid F(x) \leq x\} \in \textsc{fix}(F)$, *and*
3. $\textsc{fix}(F)$ *is a complete lattice.*

Proof. Let $\mu F := \{u \in \mathbb{L} \mid u \leq F(u)\}$.
Let $\zeta = \bigvee \mu F$ (ζ exists since $\mu F \subseteq \mathbb{L}$, a complete lattice).
$\forall u \in \mu F$, $u \leq \zeta$, so $u \leq F(u) \leq F(\zeta)$, by monotonicity.
Thus, $F(\zeta)$ is an upper bound for μF, and ζ is the least upper bound, so $\zeta \leq F(\zeta)$.

By monotonicity, $F(\zeta) \leq F(F(\zeta))$, so $F(\zeta) \in \mu F$, therefore $F(\zeta) \leq \zeta$.
Therefore, $\zeta = F(\zeta) \in \textsc{fix}(F)$.
A similar argument can be made for $\bigwedge\{u \in \mathbb{L} \mid F(u) \leq u\}$.
Lastly, we show that an arbitrary subset $A \subseteq \textsc{fix}(F)$ has a least upper bound and a greatest lower bound, thereby proving $\textsc{fix}(F)$ is a complete lattice.
Define $a := \bigvee A$, and $1_{\mathbb{L}} = \bigvee \mathbb{L}$.
Consider the interval $[a, 1_{\mathbb{L}}] := \{x \in \mathbb{L} \mid a \leq x \leq 1_{\mathbb{L}}\}$, which is a complete lattice.
Then if A has a least upper bound in $\textsc{fix}(F)$, it must lie in $[a, 1_{\mathbb{L}}]$.
Note that it suffices to show that F can be restricted to act as a monotone function $F : [a, 1_{\mathbb{L}}] \to [a, 1_{\mathbb{L}}]$, so that we may apply the first result: a monotone function on the complete lattice $[a, 1_{\mathbb{L}}]$ has a least fixed point, and this point is therefore the least upper bound of $A \subseteq \textsc{fix}(F)$.
Let $x \in A$. Then $x \leq a$ and $x = F(x) \leq F(a) = a$ by monotonicity, so $a \leq F(a)$.
Let $y \in [a, 1_{\mathbb{L}}]$. Then $a \leq y$ and $a = F(a) \leq F(y) \leq 1_{\mathbb{L}}$ by monotonicity.
Therefore $F(y) \in [a, 1_{\mathbb{L}}]$, so we conclude that $F([a, 1_{\mathbb{L}}]) \subseteq [a, 1_{\mathbb{L}}]$ which implies we may restrict the domain and co-domain, $F : [a, 1_{\mathbb{L}}] \to [a, 1_{\mathbb{L}}]$.
We have thus shown that an arbitrary subset $A \subseteq \textsc{fix}(F)$ has a least upper bound in $\textsc{fix}(F)$. It is true for all lattices $\mathbb{L}$ that if every sublattice $S \subseteq \mathbb{L}$ has a least upper bound $\bigvee S$, then S has a greatest lower bound defined by

$$\bigwedge S = \bigvee \left(\bigcap_{s \in S} \{x \in \mathbb{L} : x \leq s\} \right).$$

Therefore, $\textsc{fix}(F)$ is a complete lattice. □

We will use this theorem to perform model-checking over a Kripke structure on propositions with a fixed point operator $\phi = \sigma X.\psi$, where we use σ as a generic symbol to represent either μ or ν.

Corollary 2. *Let $K = (S, S_0, R, \mathcal{L})$ be a Kripke structure, $e : \textsc{Var} \to 2^S$ be an evaluation, and $\phi \in L_1$ be a deterministic μ-calculus formula. Define Q_i^μ and Q_i^ν recursively as follows:*

$$\begin{array}{l|l} Q_0^\mu = \emptyset & Q_0^\nu = S \\ Q_i^\mu = [\![\phi]\!]_K^{e[X \leftarrow Q_{i-1}^\mu]} & Q_i^\nu = [\![\phi]\!]_K^{e[X \leftarrow Q_{i-1}^\nu]} \end{array}$$

then

(i) $\forall i \in \mathbb{N},\ Q_{i-1}^\mu \subseteq Q_i^\mu$ *and* $Q_i^\nu \subseteq Q_{i-1}^\nu$,

(ii) $\exists n, m \in \mathbb{N}$ *such that* $Q^\mu_{n-1} = Q^\mu_n$*,* $Q^\nu_{m-1} = Q^\nu_m$*, and*

(iii) $Q^\mu_n = [\![\mu X.\phi]\!]^e_K$*,* $Q^\nu_m = [\![\nu X.\phi]\!]^e_K$*.*

Corollary 2 presents an intuitive algorithm for finding the least and greatest fixed points satisfying a given μ-calculus proposition with fixed point operators. Note that the complete lattice in question is the power set of the set of states ordered by set inclusion, $(\mathcal{P}(S), \subseteq)$, as in Example 2. The proof relies on the fact that deterministic μ-calculus propositions are inherently *syntacticallymonotone* and therefore *monotone* in their variables [7], as negation is allowed only on atomic propositions. This implies that for any formula of the form $\sigma X.\psi[X]$, $A \subseteq B$ implies $[\![\psi]\!]^{e[X\leftarrow A]}_K \subseteq [\![\psi]\!]^{e[X\leftarrow B]}_K$. Summarizing the algorithm, to evaluate $[\![\mu X.\psi]\!]^e_K$, it suffices to set Q^μ_1 to be the set of states satisfying the formula ψ where $X = Q^\mu_0$ is initialized to be the empty set. We then proceed inductively, letting each subsequent Q^μ_i be the result of finding the states which satisfy ψ where X is replaced by Q^μ_{i-1}. After a finite number n of iterations (since the set of states S of a Kripke structure is finite, see Definition 1), a fixed point $Q_{n-1} = Q_n = [\![\mu X.\psi]\!]^e_K$ will be reached. An analogous algorithm is applied when we wish to evaluate $[\![\nu X.\phi]\!]^e_K$, where $Q^\nu_0 = X$ is initialized to be S.

2.1.3 Deterministic μ-Calculus

We will focus our efforts on a fragment of μ-calculus called *deterministicμ-calculus* which admits efficient model-checking algorithms and is at least as expressive as the commonly used linear time temporal logics LTL and CTL [13]. Deterministic μ-calculus imposes some restrictions on syntax, notably that only atomic propositions may be negated, conjunction can only occur between a formula and an atomic proposition, and the universal successor operator $\Box$ is omitted. We may write this syntax succinctly in Backus-Naur form as follows:

$$\phi := p \mid \neg p \mid X \mid p \wedge \phi \mid \neg p \wedge \phi \mid \phi \vee \phi \mid \Diamond\phi \mid \mu X.\phi \mid \nu X.\phi$$

where $p \in \Pi$ and $X \in \textsc{Var}$. We denote the set of all deterministic μ-calculus formulas by L_1. Note that by restricting negation to atomic propositions only, we ensure that every formula in L_1 is syntactically monotone in its variables.

2.1.4 Specification Examples

This section will introduce several examples of commonly used deterministic μ-calculus specifications. Each example is named, described, and explained in detail to provide some intuition to the reader [13].

(i) **Reachability**: $\phi = \mu X.(p \vee \Diamond X)$

The reachability specification is used to ensure that the system eventually reaches a state which satisfies atomic proposition p. The resulting set $[\![\phi]\!]_K^e$ is the *winning set*, that is, the set of all initial states for which the proposition ϕ holds. In this case, the winning set consists of states which satisfy p or for which there exists a sequence of transitions in R which lead to a state satisfying p. In the context of the Kripke structure $K = (S, S_0, R, \mathcal{L})$, we seek only to show that S_0 is contained in the winning set $[\![\phi]\!]_K^e$.

In this example, we look for the least fixed point because we will start with the empty set and grow through all states that satisfy p or that can reach $[\![p]\!]_K$ with one transition, then two transitions, and so on until the entire winning set is found. This process is guaranteed to terminate since the formula $(p \vee \Diamond X)$ is monotone (ϕ is a deterministic μ-calculus formula), and since the Kripke structure contains finitely many states. Let us apply the algorithm from Corollary 2 to elucidate the procedure:

$$\begin{aligned}
Q_0^\mu &= \emptyset \\
Q_1^\mu &= [\![p \vee \Diamond X]\!]_K^{e[X \leftarrow Q_0^\mu]} \\
&= [\![p]\!]_K \cup [\![\Diamond X]\!]_K^{e[X \leftarrow \emptyset]} \\
&= [\![p]\!]_K \cup \mathrm{Pre}_{K,\exists}^{e[X \leftarrow \emptyset]}(X) \\
&= [\![p]\!]_K \cup \{s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \wedge s' \in \emptyset\} \\
&= [\![p]\!]_K \\
Q_2^\mu &= [\![p \vee \Diamond X]\!]_K^{e[X \leftarrow Q_1^\mu]} \\
&= [\![p]\!]_K \cup \{s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \wedge s' \in [\![p]\!]_K\}.
\end{aligned}$$

This process continues until the least fixed point is reached. See Example 1 for a concrete illustration of the reachability specification.

(ii) **Safety**: $\phi = \nu X.(p \wedge \Diamond X)$

We use the the term "safety" as this specification guarantees that a given atomic proposition will hold on some state trajectory; the atomic proposition may be concerned with being in an obstacle-free space, or it may ensure that constraints on speed or acceleration are observed, for instance. The formula ϕ will hold for all states which satisfy p and which have transitions to states which will themselves satisfy p and in turn have transitions to other states that will satisfy this same condition. For this reason, we start with the entire set of states, S, and repeatedly find intersections to narrow down the states until the greatest fixed point is reached.

$$\begin{aligned}
Q_0^\nu &= S \\
Q_1^\nu &= [\![p \wedge \Diamond X]\!]_K^{e[X \leftarrow Q_0^\nu]} \\
&= [\![p]\!]_K \cap [\![\Diamond X]\!]_K^{e[X \leftarrow S]} \\
&= [\![p]\!]_K \cap \{s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \wedge s' \in S\} \\
&= [\![p]\!]_K \\
Q_2^\nu &= [\![p \wedge \Diamond X]\!]_K^{e[X \leftarrow Q_1^\nu]} \\
&= [\![p]\!]_K \cap \{s \in S : \exists s' \in S \text{ s.t. } (s, s') \in R \wedge s' \in [\![p]\!]_K\} \\
&\vdots
\end{aligned}$$

See Example 1 for a concrete illustration of the safety specification.

(iii) **Reaching a Region Safely**: $\phi = \mu X.(\neg q \wedge (p \vee \Diamond X))$
One way of combining the above two specifications is to ensure that a safety condition is met while trying to reach an objective. This specification is commonly used for motion planning problems, wherein a planner searches for a trajectory which avoids obstacles (represented by states satisfying q), and reaches a given goal (represented by states satisfying p). Obstacle avoidance is guaranteed by the conjunction of the usual reachability subformula with $\neg q$ so that at each iteration, we keep only those states which satisfy the reachability criterion and are not obstacles.

(iv) **Reaching a Safe Region**: $\phi = \mu X.((\nu Y.(p \wedge \Diamond Y)) \vee \Diamond X)$
Another way to combine safety and reachability is the specification to reach a region whose states always satisfy a property p. The subformula $\psi = \nu Y.(p \wedge \Diamond Y)$ is identical to the safety specification listed above; ψ is satisfied by all initial states which give rise to trajectories that always satisfy p. This safety specification is wrapped in the reachability specification $\mu X.(\psi \vee \Diamond X)$, meaning states which satisfy ϕ are either in the safe region already, or can reach the safe region using a finite number of transitions.

(v) **Ordering**: $\phi = \mu X.(q \vee (p \wedge \Diamond X))$
In both LTL and CTL, there is a temporal operator U denoting "until", so that $p\mathrm{U}q$ is satisfied provided p holds at least until q holds; that is, after a finite number of transitions, q must hold, and p may or may not continue to hold. In μ-calculus, we can formulate this specification by building up a set of states where either q is already satisfied, or p is satisfied and there exists a transition to a state satisfying

$q \vee (p \wedge \Diamond X)$. Another way to interpret the formula is to distribute the disjunction to obtain $\mu X.((q \vee p) \wedge (q \vee \Diamond X))$. In this form, we observe the reachability subformula $q \vee \Diamond X$ (in the scope of a least fixed point operator, as usual), so we may conclude that the winning set contains states which eventually satisfy q, and along the way must satisfy p (otherwise q is already satisfied, so p may or may not be satisfied).

(vi) **Liveness**: $\phi = \nu Y.\mu X.((p \wedge \Diamond Y) \vee \Diamond X)$

Liveness, is a specification which guarantees that atomic proposition p is satisfied infinitely often. This example is the first with an alternation depth greater than one, where alternation depth refers to the level of mutually recursive least and greatest fixed point operators [32]. Consider the largest proper subformula of ϕ, $\psi = \mu X.((p \wedge \Diamond Y) \vee \Diamond X)$. This can be seen as a reachability specification where the goal is to eventually reach states satisfying $\eta = p \wedge \Diamond Y$, which itself looks like a safety specification, remarking that η is in the scope of a greatest fixed point operator. We may parse the liveness specification $\phi = \nu Y.\psi = \nu Y.\mu X.(\eta \vee \Diamond X)$ as follows: ψ ensures that a region satisfying $\eta = p \wedge \Diamond Y$ is reached, and η ensures that p is satisfied and that there is a transition to a state satisfying ψ. Having mutually recursive greatest and least fixed point operators in this way allows for this more complex combination of reachability and safety, where ϕ is satisfied by all states which have paths that *always eventually* reach $[\![p]\!]_K$. Note that liveness is also sometimes called the *Büchi objective* in the context of infinite parity games, a topic closely related to μ-calculus [4, 15, 32].

2.2 SST*

In this section, we discuss a probabilistically complete sampling-based kinodynamic motion planning algorithm called Stable Sparse RRT (SST) along with its asymptotically optimal variant, SST*[1]. For completeness, the planner is summarized in this section, although further details and proofs can be found in [21].

One very useful property of SST/SST* is the fact that it does not rely on having the solution to an optimal boundary value problem (OBVP) for the relevant system. Such a solution is called a steering function[2] in the literature, and many motion planning

[1]Note that motion planning algorithms whose acronyms end with an asterisk (*) are usually asymptotically optimal variants of their associated algorithm.

[2]Some authors use different terminology for the steering/OBVP problem. For instance, in their paper on PRM, Kavraki et al. refer to a "local planner" which is used to connect neighbouring nodes in the case

algorithms are contingent upon its availability. For example, some planning algorithms that necessitate using a steering function include RRT* [14] and Probabilistic Roadmap (PRM) [16]. The steering function provides, as the name suggests, optimal inputs to control or *steer* the system from one given state to another; for many planners, the necessity of such a function arises when sampled states (nodes) must be connected together with directed edges, representing that there is a known set of inputs to control the system between such states. The problem is, just as finding analytic solutions to nonlinear differential equations is very difficult or impossible, so too is finding the related solution to an associated OBVP.

To be sure, there exist a small number of sampling-based planning algorithms that do not require a steering function, most notably RRT-Extend [18] and Expansive Space Trees (EST). While EST has been shown to be asymptotically optimal (RRT-Extend is not), the rate of convergence to the (near-) optimal solution is logarithmic, making it impractical at reliably finding high-quality paths. On the other hand, the advantages offered by SST* include being provably asymptotically optimal as well as having good (linear) convergence to high-quality solutions. SST* is further improved by its use of a sparse data structure, where a pruning operation is used to accelerate nearest neighbours searches, thereby ameliorating computational efficiency.

We will now describe the implementation details of the SST* sampling-based planning algorithm. SST* employs `MonteCarlo_Prop` which forward propagates a selected node, $x_{selected}$; that is, a random control vector is sampled from the allowed control-space, $\mathbb{U}$, and supplied as input to the system dynamics for a random duration, up to some specified maximum time, T_{prop}, resulting in trajectories with piecewise constant control inputs (see Algorithm 1).

Algorithm 1 `MonteCarlo_Prop`$(x_{selected}, \mathbb{U}, T_{prop})$

```
1: $t \leftarrow$ Sample$([0, T_{prop}])$
2: $\Upsilon \leftarrow$ Sample$(\mathbb{U})$
3: return $x_{new} \leftarrow x_{selected} + \int_0^t f(x(\tau), \Upsilon) d\tau$
```

SST* also uses a best-first selection strategy to forward integrate from the least-cost node found within a specified radius of the sampled state, thereby improving convergence to high-quality solutions. To elaborate, we use Algorithm 2 called `Best_First_Selection` to sample a random state, x_{rand}, from the state space, $\mathbb{X}$, then it stores in X_{near} the set of all existing states in the tree within a δ_{BN} radius of x_{rand}. If X_{near} is empty, we simply return the nearest node to x_{rand} in the set of all nodes, $\mathbb{V}$. Otherwise, the state in X_{near}

of *holonomic* robots.

with the least cost is selected. In either case, the selected state, $x_{selected}$, is the state from which we propagate and grow the tree using `MonteCarlo_Prop`.

Algorithm 2 `Best_First_Selection`$(\mathbb{X}, \mathbb{V}, \delta_{BN})$

```
1: $x_{rand} \leftarrow$ Sample-State$(\mathbb{X})$
2: $X_{near} \leftarrow$ Near$(\mathbb{V}, x_{rand}, \delta_{BN})$
3: if $X_{near} == \emptyset$ then
4:     return Nearest$(\mathbb{V}, x_{rand})$
5: else
6:     return $\arg\min_{x \in X_{near}}$ Cost$(x)$
```

Furthermore, SST* applies a pruning operation to maintain a sparse data structure. Pruning removes high-cost nodes to improve run-time by accelerating nearest neighbour searches. With this in mind, a graph of witness nodes is maintained, where each witness keeps track of an optimal-cost representative node within a δ_s-radius of the witness. Correspondingly, we must determine whether a new node is the "best" in its neighbourhood in order to decide whether or not to keep it. For this purpose, we use `Is_Locally_Best` presented in Algorithm 3. The algorithm finds the nearest witness state, s_{new}, to the newly propagated state x_{new}. If s_{new} is not within distance δ_s of x_{new}, x_{new} is deemed locally best by default and becomes a new witness node. In this way, x_{new} is added to the set of witness nodes, S. If x_{new} does have a neighbour within a δ_s radius, the chosen cost function, `Cost`, is used to check whether the new state is better than the locally best representative of the witness node.

Algorithm 3 `Is_Locally_Best`(x_{new}, S, δ_s)

```
1: $s_{new} \leftarrow$ Nearest$(S, x_{new})$
2: if dist$(x_{new}, s_{new}) > \delta_s$ then
3:     $S \leftarrow S \cup \{x_{new}\}$
4:     $s_{new} \leftarrow x_{new}$
5:     $s_{new}.rep \leftarrow$ NULL
6: $x_{peer} \leftarrow s_{new}.rep$
7: if $x_{peer} ==$ NULL or Cost$(x_{new}) <$ Cost$(x_{peer})$ then
8:     return True
9: return False
```

Now that there is a means of determining whether a state is locally best, we are ready to define the algorithm which enforces sparsity, `Prune` (Algorithm 4). First, the nearest witness state, s_{new}, to the newly propagated state, x_{new}, is found. We want to set x_{new} to be the representative of s_{new} since it has been deemed locally best if the SST* algorithm has reached this point (see Algorithm 5), but first we must check whether or not s_{new} already has a representative. Since `Prune` is only executed if `Is_Locally_Best` returns `True`, then if s_{new} has a representative, it must have higher cost than x_{new}, and it must now be moved from $\mathbb{V}_{active}$ to $\mathbb{V}_{inactive}$. The next step is to remove inactive leaf nodes recursively, so that if the previous representative of s_{new} is an inactive leaf node, it is removed entirely from the set of all nodes, $\mathbb{V}$, and the check is preformed again with the parent of the removed leaf node.

Algorithm 4 `Prune`($x_{new}, \mathbb{V}_{active}, \mathbb{V}_{inactive}, \mathbb{E}$)

```
1: $s_{new} \leftarrow \texttt{Nearest}(S, x_{new})$
2: $x_{peer} \leftarrow s_{new}.rep$
3: if $x_{peer} \neq \texttt{NULL}$ then
4:     $\mathbb{V}_{active} \leftarrow \mathbb{V}_{active} \setminus \{x_{peer}\}$
5:     $\mathbb{V}_{inactive} \leftarrow \mathbb{V}_{inactive} \cup \{x_{peer}\}$
6: $s_{new}.rep \leftarrow x_{new}$
7: while IsLeaf($x_{peer}$) and $x_{peer} \in \mathbb{V}_{inactive}$ do
8:     $x_{parent} \leftarrow x_{peer}.parent$
9:     $\mathbb{E} \leftarrow \mathbb{E} \setminus \{\overline{x_{parent} \rightarrow x_{peer}}\}$
10: $\mathbb{V}_{inactive} \leftarrow \mathbb{V}_{inactive} \setminus \{x_{peer}\}$
11: $x_{peer} \leftarrow x_{parent}$
```

With the necessary helper functions defined, the pseudocode for SST* is outlined in Algorithm 5. Note that the nested for-loop constitutes the core of SST, and the addition of the update step for δ_s and $delta_{BN}$ is all that is necessary to make the algorithm asymptotically optimal instead of being merely asymptotically near-optimal. This is because, if these two parameters do not tend to zero as the number of iterations goes to infinity, then there will always be some degree of sparsity, which means that not every possible solution is explored. In terms of notation, the overline indicates an edge (trajectory from one state to another), and d, l denote the number of dimensions of the state space and the control space, respectively.

It is important to note that proper tuning of the parameters used in SST* is crucial for effective path planning. The most significant parameters are δ_{BN} and δ_s, which are the radii for selecting low-cost nearby nodes and for pruning in the vicinity of witness nodes, respectively. Choosing a large value of δ_s can decrease the time it takes to find an initial solution at the cost of solution quality, while choosing a large value of δ_{BN}

Algorithm 5 `SST*`$(\mathbb{X}, \mathbb{U}, x_0, T_{prop}, N, \delta_{BN}, \delta_s, \xi)$

```
1: $\mathbb{V}_{active} \leftarrow \{x_0\};\ \ \mathbb{V}_{inactive} \leftarrow \emptyset$
2: $\mathbb{E} \leftarrow \emptyset$
3: $s_0 \leftarrow x_0;\ \ s_0.rep = x_0;\ \ S \leftarrow \{s_0\}$
4: $j \leftarrow 0$
5: while True do
6:     for N iterations do
7:         $x_{selected} \leftarrow$ Best_First_Selection$(\mathbb{X}, \mathbb{V}_{active}, \delta_{BN})$
8:         $x_{new} \leftarrow$ MonteCarlo_Prop$(x_{selected}, \mathbb{U}, T_{prop})$
9:         if CollisionFree$(\overline{x_{selected} \rightarrow x_{new}})$ then
10:            if Is_Locally_Best$(x_{new}, S, \delta_s)$ then
11:                $\mathbb{V}_{active} \leftarrow \mathbb{V}_{active} \cup \{x_{new}\}$
12:                $\mathbb{E} \leftarrow \mathbb{E} \cup \{\overline{x_{selected} \rightarrow x_{new}}\}$
13:                Prune$(x_{new}, \mathbb{V}_{active}, \mathbb{V}_{inactive}, \mathbb{E})$
14:     $\delta_s \leftarrow \xi \cdot \delta_s;\ \ \delta_{BN} \leftarrow \xi \cdot \delta_{BN}$
15:     $j \leftarrow j + 1$
16:     $N \leftarrow N(1 + \log(j))\xi^{-j(d+l+1)}$
17: $\mathbb{V} \leftarrow \mathbb{V}_{active} \cup \mathbb{V}_{inactive}$
18: return $(\mathbb{V}, \mathbb{E})$
```

can improve initial solution quality while increasing the time it takes to find an initial solution. Moreover, the SST* algorithm is *incremental*, meaning states are not sampled all at once, rather during each iteration a new state is sampled to incrementally grow a tree. A consequence of the incremental nature of SST* is that there is no clear way to precompute solution trajectories for online use unless (necessarily static) obstacles and the initial state are specified exactly. In simulations, we have found that feasible trajectories can often be computed in under 10 seconds for simple linear systems with state space dimension three or lower; however, it can take several hundred seconds to compute solutions for higher-dimensional nonlinear systems.

2.3 FMT*

Fast Marching Tree (FMT*) is an asymptotically optimal sampling-based motion planning algorithm that was specifically developed for use in high-dimensional systems [10]. In simulations, Janson et al. consistently found that FMT* converges to high-quality solutions faster than PRM* and RRT* for systems with dimension from 2D to 7D. The improvements were more noticeable in higher dimensions, making FMT* a very promising algorithm for complex motion planning problems such as for those involving quadrotors, which operate in a 12D state space.

Unlike SST*, FMT* is not an incremental algorithm. An important distinction results from this difference in algorithm design, namely that the definition of asymptotic optimality must be altered to accommodate the fact that FMT* uses a predetermined number of sampled states. As mentioned in 1.1.2, the definition of asymptotic optimality essentially involves finding an optimal solution as the number of samples tends to infinity. In such case, each iteration builds upon an existing graph, refining the structure and covering more of the state space. For non-incremental algorithms, the definition of asymptotic optimality can be summarized as follows: the cost of the solution returned by an algorithm must converge in probability to the optimal cost.

For this work, we consider the kinodynamic variant of FMT* by Ross Allen and Marco Pavone called `kinoFMT`. The primary distinction lies in the fact that, due to considerations of differential constraints, there is a fundamental difference between searching for nearest backward-reachable and forward-reachable states. In the case of backward reachability, `Near_Backward`(x, V, J_{th}) finds states in V that can reach state x without exceeding a threshold cost, J_{th}. In contrast, `Near_Forward`(x, V, J_{th}) finds states in V that can be reached by x itself without exceeding J_{th}. In fact, using a cost threshold is another deviation from the standard FMT* algorithm, which uses a distance threshold. Since kinodynamic planning is concerned with feasibility under kinematic and dynamic constraints, a simple distance metric is insufficient for determining which states are easily reachable from another state. See Algorithm 6 for the detailed pseudocode of `kinoFMT`, as shown in [1] except for some slight changes for clarity and implementation simplicity.

Algorithm 6 `kinoFMT`$(x_{init}, X_{goal}, \mathbb{X}, n, J_{th})$

```
 1: $\mathbb{V} \leftarrow \{x_{init}\} \cup \texttt{Sample}(n, \mathbb{X})$
 2: $\mathbb{E} \leftarrow \emptyset$
 3: $W \leftarrow \mathbb{V} \setminus \{x_{init}\}; H \leftarrow \{x_{init}\}$
 4: $z \leftarrow x_{init}$
 5: while $z \notin X_{goal}$ do
 6:     $N_z^{out} \leftarrow \texttt{Near\_Forward}(z, V \setminus \{z\}, J_{th})$
 7:     $X_{near} = N_z^{out} \cap W$
 8:     for $x \in X_{near}$ do
 9:         $N_x^{in} \leftarrow \texttt{Near\_Backward}(x, V \setminus \{x\}, J_{th})$
10:         $Y_{near} \leftarrow N_x^{in} \cap H$
11:         $y_{min} \leftarrow \arg\min_{y \in Y_{near}} \{y.cost + \texttt{Cost}(\overline{y \rightarrow x})\}$
12:         if $\texttt{CollisionFree}(\overline{y_{min} \rightarrow x})$ then
13:             $\mathbb{E} \leftarrow \mathbb{E} \cup \{\overline{y_{min} \rightarrow x}\}$
14:             $x.cost \leftarrow y_{min}.cost + \texttt{Cost}(\overline{y_{min} \rightarrow x})$
15:             $H \leftarrow H \cup \{x\}$
16:             $W \leftarrow W \setminus \{x\}$
17:     $H \leftarrow H \setminus \{z\}$
18:     if $H == \emptyset$ then
19:         return Failure
20:     $z \leftarrow \arg\min_{y \in H} \{y.cost\}$
21: return $\texttt{Path}(z, \mathbb{V}, \mathbb{E})$
```

The main idea behind FMT* and `kinoFMT` is to use forward dynamic programming on a predetermined number of sampled states. The algorithms perform graph construction and graph search simultaneously, thus the final least-cost node lying in the goal region is already known when the algorithm terminates. In other words, by the nature of the expansion of the tree structure, the least-cost nodes on the frontier of expansion are always tracked, so that when the goal is reached, it is not necessary to search the entire tree for the optimal terminal node. Moreover, since tree structures contain no cycles, there exists a unique path from the starting node to the terminal node.

In more detail, `kinoFMT` takes as input an initial state, x_{init}, and goal region, X_{goal}, the state space, $\mathbb{X}$, the number of nodes to sample, and a cost threshold, Jth. The `Sample` function uniformly samples the entire state space[3] (usually with some samples selected directly from the goal region) and stores these sampled states in $\mathbb{V}$ along with x_{init} (line 1). The set W contains unexplored nodes, and the set H contains the nodes forming the frontier of expansion of the tree, which initially includes only x_{init} (line 3). Every iteration, the least-cost node $z \in H$ becomes the pivot about which expansion occurs (lines 6, 20). With the pivot known, we define X_{near} to be the set of as yet unexplored nearest nodes in the forward-reachable set of z (lines 5–7). Then, for each element $x \in X_{near}$, define Y_{near} to be the set of nodes in the frontier that are also in the backward-reachable set of x (lines 8–10). The least-cost state y_{min} is then found from the set of all $y \in Y_{near}$, where the cost is determined to be the sum of the cost of reaching y (along its unique path from x_{init}) and the cost to travel from y to x, recalling that y is in the backward-reachable set of x (line 11). This is the dynamic programming step, where we try to minimize the cost-to-come and the cost-to-go. Once the cost minimizer is found, a collision check is performed (line 12), and if a collision is detected along the transition from y_{min} to x, then the algorithm simply discards this iteration and proceeds to the next. Otherwise, the edge (transition) is added to the tree, the newly connected node's cost is updated and the node is added to the frontier set, H, and it is simultaneously removed from the set of unexplored nodes, W (lines 13–16). Once the for-loop has iterated through the entire set of forward-reachable nodes, X_{near}, the pivot, z, is removed from the frontier and the procedure repeats until either the frontier is empty, at which point "Failure" is returned (lines 18–19), or until z lies within the goal region, at which point the algorithm terminates. The `Path` function simply returns the optimal path, that is, the unique path from x_{init} to the final least-cost goal state, z.

[3]The FMT* algorithm specifies that samples are taken only from the free space (i.e., not including obstacles,) however, sampling from the entire state space allows for a much more general motion planning framework, as discussed in Section 4.2.

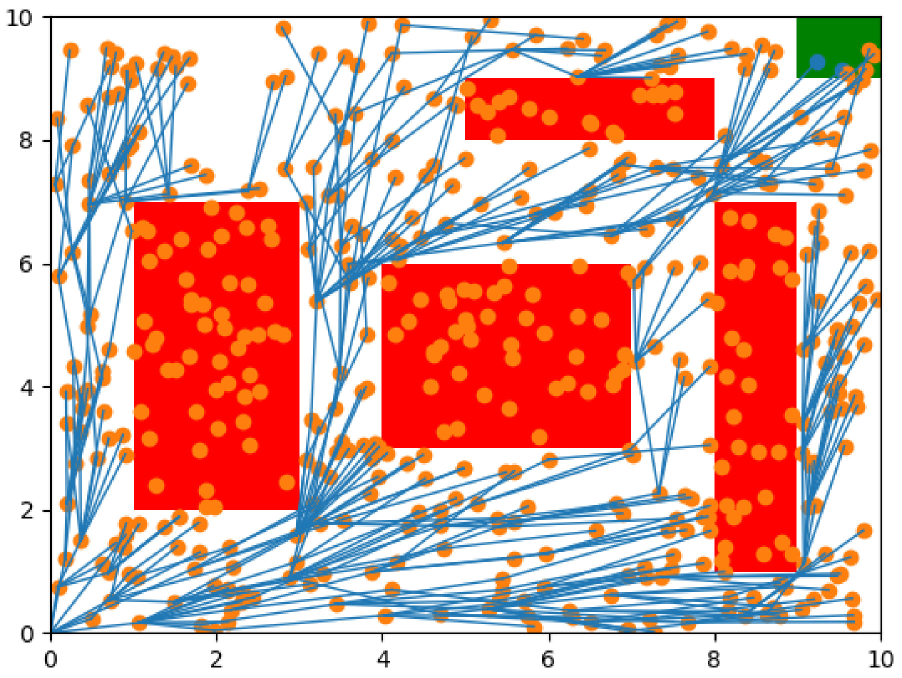

Figure 2.1: Example of a Kripke structure, with obstacles shown in red, free space shown in white, and the goal set shown in green. The set of states, S, is represented by the nodes of the graph, the set of initial states is given by the singleton containing the bottom-left node, and the set of relations is represented by the edges. The labeling function is not defined in this example, though it is nonetheless necessary to properly describe a Kripke structure.

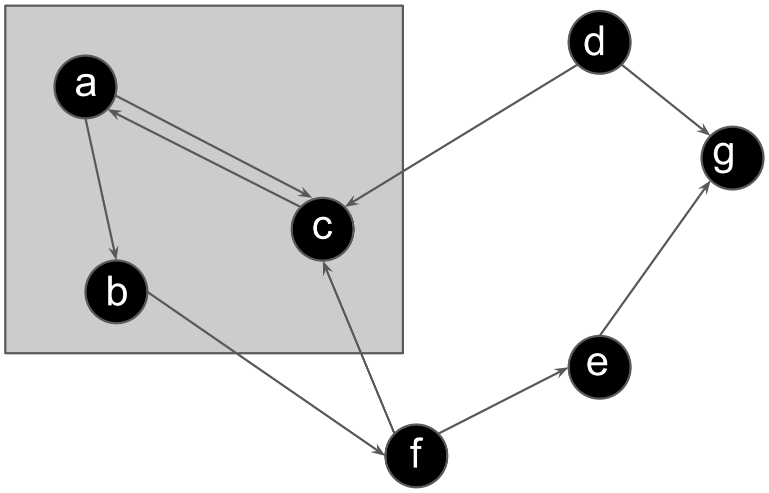

Figure 2.2: μ-Calculus semantics Example 1.

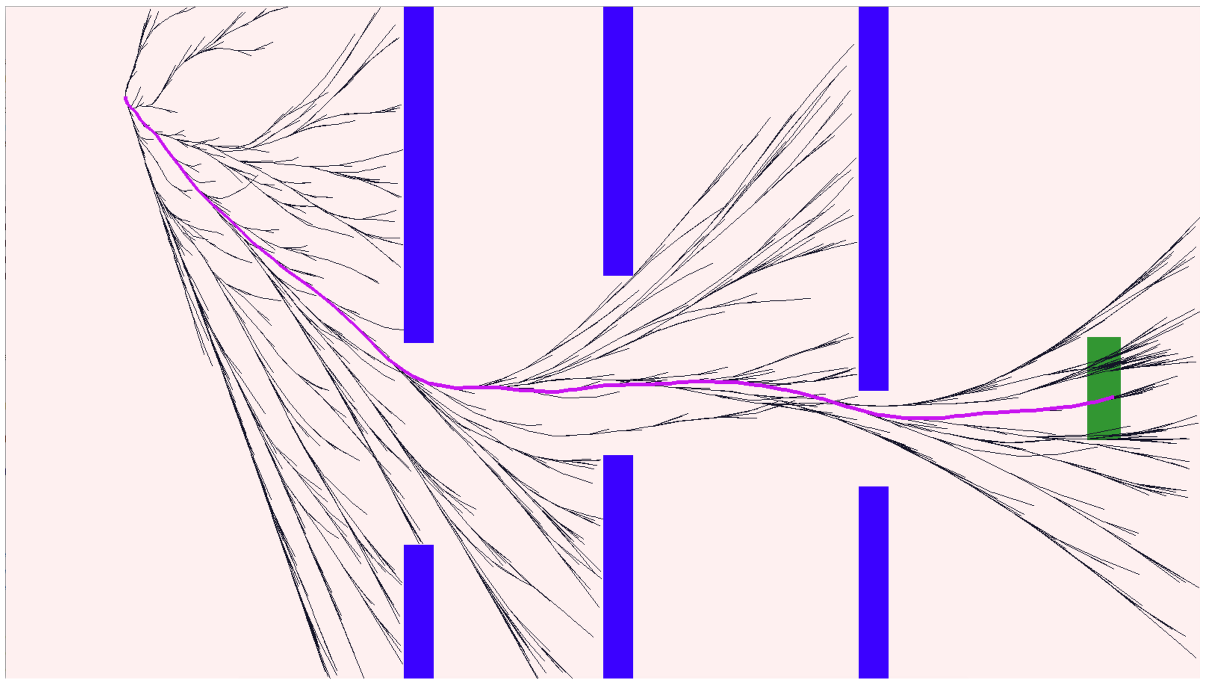

Figure 2.3: An example of a tree produced by SST for the flappy bird problem. The initial state is in the top-left, the point travels at a constant horizontal velocity, and the control-space is a set containing only two options: “do nothing”, or “accelerate up”, each for a random duration. The least-cost solution for this execution is highlighted in pink.

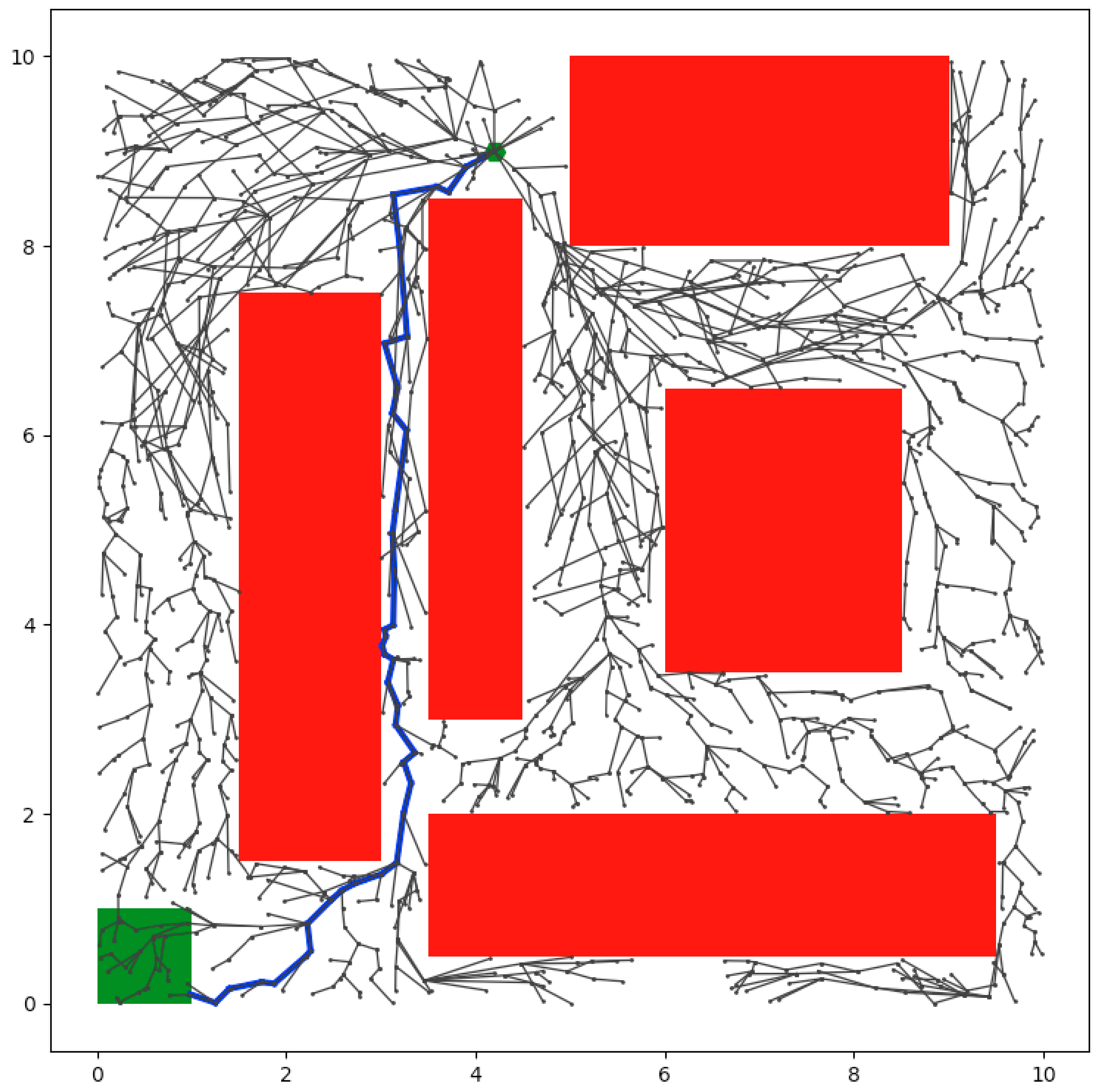

Figure 2.4: Example of a tree generated by FMT* using 2000 nodes. The initial state is shown at the top center as a green hexagon, the goal region lies in the bottom left and is shown in green, and obstacles are represented by red rectangles. The optimal path is highlighted in blue. This example does not use any dynamic model, so the cost function is simply the Euclidean distance.

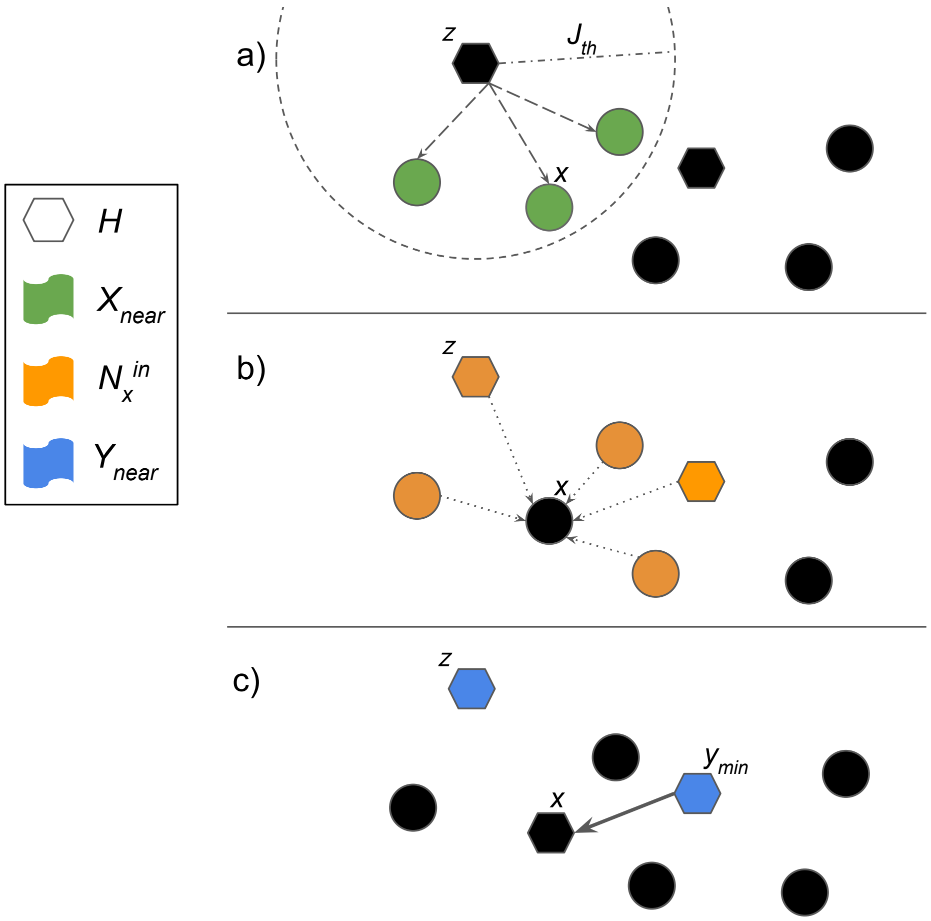

Figure 2.5: Illustration of the FMT* algorithm.
a) Of the two nodes in the frontier, H, the one with least cost is selected to be z. Distance is used as the cost, with search radius J_{th}, and three unvisited nodes are found to lie within this distance from z. These are the nodes, belonging to X_{near}. The algorithm will iterate through all three nodes in X_{near}, but we focus on one for this example.
b) For the current iteration, add all nodes whose search radius includes node x to N_x^{in} (not necessarily all nodes within the search radius of x); for non-symmetric cost functions, this distinction could have a significant impact (e.g., the `kinoFMT` algorithm).
c) Of the nodes in N_x^{in}, only those that are also in the frontier, H, are included in the set Y_{near}. For each node $y \in Y_{near}$, the cost of each, $y.cost + \texttt{Cost}(\overline{y \rightarrow x})$, is computed and the least-cost node, y_{min}, is determined. Provided there is no collision when connecting y_{min} to x, the edge is added to the tree and x is added to the set H.

Chapter 3

Sampling-Based Motion Planning with μ-Calculus Specifications without Steering

This chapter presents the work I have published in [17].

3.1 Introduction

Motion planning in complex environments has seen a shift towards using sampling-based planning algorithms. By using a predetermined sampling scheme, a random snapshot of the workspace can be taken and incrementally improved upon without any prior knowledge of the environment. Furthermore, combining sampling-based motion planning with temporal logic specifications grants users a much more sophisticated toolbox of high-level behaviors that can be specified.

Temporal logics present a means of formally expressing high-level specifications for use in various problems in mathematics, robotics, and computer science. In particular, temporal logic specifications are well-suited for motion planning problems, allowing a user-defined specification to describe the desired behavior of an autonomous vehicle or robot [23, 33]. The modal μ-calculus is a highly expressive temporal logic which permits more diverse and complex specifications than the most widely used temporal logics, including LTL, CTL, and extensions thereof. On the other hand, it is typically much easier to understand LTL specifications at a glance, whereas μ-calculus formulas can be much more difficult to intuit. The added complexity of μ-calculus formulas is not without benefit, however, as a major advantage of using μ-calculus to formulate specifications is its predisposition for simple model-checking. While LTL specifications are a useful tool for high-level planning, they must usually be translated into automata for the purposes of model-checking. The equivalent μ-calculus specification, however, can be checked directly without any intermediate steps via the Tarski-Knaster fixed point theorem [5, 27].

In this thesis, a fragment of the full μ-calculus called deterministic μ-calculus [13] is used, allowing for efficient model checking while maintaining the ability to specify complex tasks. Some such specifications include reaching a goal while avoiding obstacles, and a property known as liveness, which involves satisfying one or many propositions infinitely often. Furthermore, using the model checking algorithm discussed in Section 3.3, it is possible to formally synthesize a control policy that provably satisfies a given deterministic μ-calculus specification.

Summarizing our contribution, we have successfully solved a kinodynamic planning problem with temporal logic specifications without the need for a solution to an OBVP, also called a steering function. While using temporal logic specifications with motion planning has been heavily researched, e.g., [2, 3, 13, 23, 33], it is often difficult or impossible to find a steering function, allowing motion planning only for simple dynamical systems. Addressing this issue, we have developed a means of combining the asymptotically optimal and probabilistically complete kinodynamic planning algorithm SST* from [21] with the model checking procedure from [13] to create a motion planning algorithm with deterministic μ-calculus specifications that does not rely on a steering function. By merging information obtained from multiple Kripke structures, we are able to create one abstracted Kripke structure which stores the most cost-efficient paths that reach other proposition regions of the state space. To connect the trajectories found from multiple Kripke structures, a Linear Quadratic Regulator (LQR) feedback control policy is used to track the candidate trajectories stored in the abstracted Kripke structure. Simulations demonstrate that it is possible to satisfy a complex liveness specification for infinitely often reaching three regions of state space using only forward propagation.

3.2 Problem Formulation

Consider a time-invariant continuous dynamical control system given by the differential equation

$$\dot{x}(t) = f(x(t), u(t)), \quad x(0) = x_0 \tag{3.1}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the control input, and $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is a locally Lipschitz function. Let Π be a set of *atomic propositions*, which are the simplest form of declarative statements that are either true or false. Lastly, let $L : \mathbb{R}^n \to 2^{\Pi}$ be a labeling function which assigns to a state all of the atomic propositions that it satisfies.

The goal is to design a controller such that the possibly infinite state trajectory $x(t)$ satisfies a given temporal logic specification, Φ. For this work, we choose to work with deterministic μ-calculus. It is important to note that model checking of the temporal logic specification must be performed on a finite model of the dynamical system. To this end, a sampling-based motion planner is used to generate a discretization of the set of possible trajectories from the initial condition.

3.2.1 Problem Statement

A precise formulation of the problem to be solved can now be made. We seek to bring together the dynamical control system with the notion of a Kripke structure and deterministic μ-calculus specifications to determine whether a given specification is satisfied by a Kripke structure that models the dynamical control system.

Definition 7. *A continuous-time dynamical control system of the form (3.1) is said to satisfy a deterministic μ-calculus specification Φ at some initial state x_0 if and only if there exists a Kripke structure $K^* = (S^*, \{x_0\}, R^*, \mathcal{L}^*)$ modeling the system, and such that $x_0 \in [\![\Phi]\!]_{K^*}$.*

In contrast to [13], we allow the dynamical control system to be continuous in time. The problem statement is as follows:

Given a continuous-time dynamical control system (3.1) with initial state x_0 and a deterministic μ-calculus formula Φ, return a control policy u which gives rise to a trajectory satisfying Φ obtained from a Kripke structure that models the system, or return failure if such a trajectory is not found.

3.3 Kripke Structures and Model Checking

The main goal of this paper is to design an algorithm that finds near-optimal trajectories satisfying a given temporal logic specification without relying on an OBVP (steering function). The motivation for such an algorithm is that in a differentially constrained system, finding an optimal trajectory between two states is difficult in general. Some research has been done to first linearize system dynamics to find a solution to the BVP [30], while others have found some success in numerical solutions to BVPs with nonlinear dynamics using sequential quadratic programming [34]. However, neither approach fully addresses the crux of the problem: some planning problems, such as systems simulated on a physics engine, allow only forward propagation. We opt to use the asymptotically optimal variant of SST called SST* by Li et al. [21] which does not require a steering function to plan high-quality trajectories.

3.3.1 Model Checking with μ-Calculus Specifications

It is necessary to determine whether or not the μ-calculus specification Φ is satisfied during the incremental tree expansion (refining the discretized state space) with SST*. To perform such a verification, we will use a local model checking algorithm tailored specifically to run efficiently for deterministic μ-calculus. The procedure requires as input a Kripke structure, K, the μ-calculus specification, Φ, an initial state, s, and the subformula to be checked in the current execution, ϕ. We further require a function `succ`(s), which returns all states in K which may be reached from s via one relation (edge of the tree), and `BoundFormula`(X), which maps the input variable X to the subformula of Φ of the form $\sigma X.\psi$, that is, the smallest subformula that binds the variable X to a least or greatest fixed-point operator. Note that the first two arguments of `ModelCheck` are omitted in recursive calls for brevity as they remain unchanged. The local model checking algorithm presented here is based on Algorithm 3 presented in [13].

Algorithm 7 is a recursive function which returns a boolean value without having to create any intermediary graphs unlike the proposed *incremental* model checking algorithms from [5] and [13], and also without the need for generating an automaton on which to perform model checking, unlike when using LTL, for example. Since model checking will be performed on a very small abstracted Kripke structure, using the local, non-incremental model checking algorithm is sufficient. The algorithm presented here is based on a similar algorithm from [26], wherein correctness is proved.

3.3.2 Abstracted Kripke Structure and Planning

The primary contribution we make is to simplify model checking over several data structures by creating an abstracted Kripke structure. We generate this new abstracted structure $\tilde{K}_{\Pi^+(\Phi)} = (\tilde{S}, \{\tilde{s_0}\}, \tilde{R}, \mathcal{L})$ from $p = |\Pi^+(\Phi)|$ Kripke structures of the form

Algorithm 7 `ModelCheck`(K, Φ, s, ϕ)

```
switch $\phi$ do
    case $p$ where $p \in \Pi$
        return $p \in \mathcal{L}(s)$
    case $\neg p$ where $p \in \Pi$
        return $p \notin \mathcal{L}(s)$
    case $p \wedge \varphi$
        return $p \wedge$ ModelCheck$(s, \varphi)$
    case $\neg p \wedge \varphi$
        return $\neg p \wedge$ ModelCheck$(s, \varphi)$
    case $\psi \vee \varphi$
        return ModelCheck$(s, \psi) \vee$ ModelCheck$(s, \varphi)$
    case $\Diamond\varphi$
        for $s' \in$ succ$(s)$ do
            if ModelCheck$(s', \varphi)$ then
                return True
        return False
    case $\sigma X.\varphi$ where $\sigma \in \{\mu, \nu\}$
        set $\leftarrow$ set $\cup \{(s, \varphi)\}$
        value $\leftarrow$ ModelCheck$(s, \varphi)$
        set $\leftarrow$ set $\setminus \{(s, \varphi)\}$
        return value
    case $X$ where $X \in$ Var
        if $(s,$ BoundFormula$(X)) \in$ set then
            switch BoundFormula$(X)$ do
                case $\mu X.\varphi$
                    return False
                case $\nu X.\varphi$
                    return True
        else
            return ModelCheck$(s,$ BoundFormula$(X))$
```

$K_i = (S_i, \{s_{i0}\}, R_i, \mathcal{L}), i = \{1, \ldots, p\}$, generated via SST*, where $\Pi^+(\Phi) \subseteq \Pi$ is the set of atomic propositions which appear positively in specification Φ. The reason for restricting the abstracted Kripke structure to use only the positively-appearing atomic propositions is that these are the specifications we explicitly want to fulfill (e.g., reach a certain goal), as opposed to something we do not want our system to do (e.g., run into obstacles). In this way, a directed edge is added to the abstracted Kripke structure only if a trajectory in one of the K_i is found to initially satisfy one atomic proposition, and reaches a state satisfying another atomic proposition, all the while ensuring any negatively-appearing atomic propositions are respected so that the associated regions of state space are avoided. If any of these conditions does not hold, the edge is not added to the graph. The planning procedure, including the generation of the abstracted Kripke structure and the model checking to be performed on this structure, is as follows.

Algorithm 8 `KinoSpecPlan`$(f, x_0, \Phi,$ `regions`$, \mathcal{L})$

```
1: $\tilde{K} \leftarrow (\{x_0\}, \{x_0\}, \emptyset, \mathcal{L})$
2: $K \leftarrow \emptyset$
3: $p \leftarrow$ length(regions)
4: for $i \in \{1, \ldots, p\}$ do
5:     $s_{i0} \leftarrow$ ChooseInit(regions$[i], x_0)$
6:     $K_i \leftarrow (\{s_{i0}\}, \{s_{i0}\}, \emptyset, \mathcal{L})$
7:     $K \leftarrow K \cup \{K_i\}$
8: while $\neg$ModelCheck$(\tilde{K}, \Phi, x_0, \Phi)$ do
9:     for $n \in \{1, \ldots, p\}$ do
10:        $K_i \leftarrow$ SST*$(K_i)$
11:    $\tilde{K} \leftarrow$ AbstractUpdate$(\tilde{K}, K)$
12: return ConstructPath$(\tilde{K}, K)$
```

The `KinoSpecPlan` algorithm takes as input the dynamics f from (3.1), the initial condition x_0, the deterministic μ-calculus specification Φ, an array called `regions` containing p subsets of the state space (one for each positively-appearing atomic proposition in Φ), and the labeling function $\mathcal{L}$. The function `ChooseInit` samples a state from the input region to use as the initial state for Kripke structure K_i, or x_0 if it exists in the given region. SST* takes a Kripke structure and incrementally grows the structure using forward propagation, thereby updating its set of states and relations. `AbstractUpdate` verifies the existence of any paths spanning from one region to another in the list K of all Kripke structures. If so, the corresponding relation is added to the list of relations maintained in the abstracted Kripke structure $\tilde{K}$. Furthermore, the first time it is run, `AbstractUpdate` adds the necessary states from the initial states of each of the K_i to its own set of states. Using the abstracted Kripke structure and the list of all Kripke structures, `ConstructPath` creates a single time-parameterized state trajectory along the least-cost path satisfying Φ, where each individual candidate path from the Kripke structures is combined head-to-tail as necessary.

In essence, the algorithm works as follows. First, the abstracted Kripke structure $\tilde{K}$ is initialized with only the initial condition and an empty set of relations, and K, the list of Kripke structures, is initialized to be empty (lines 1–3). In lines 4–7, the for-loop initializes each Kripke structure K_i, $i \in \{1, \dots, p\}$, to be the trivial graph consisting only of the initial condition s_{i0}, which is chosen from among any of the states in the region in state space associated with atomic proposition $\pi_i \in \Pi^+$; that is, the initial node s_{i0} of K_i satisfies $\pi_i \in \mathcal{L}(s_{i0})$.

Next, each Kripke structure is expanded in parallel using the SST* motion planning algorithm (lines 8–10). The abstracted Kripke structure $\tilde{K}_{\Pi^+}$ is then updated in line 11 so that each of its p nodes $\tilde{s_i} \in \tilde{S}$ corresponds to a positively-appearing atomic proposition so $\pi_i \in \mathcal{L}(\tilde{s_i})$, $i \in \{1, \dots, p\}$. The relations in $\tilde{R}$ are also updated and relation $(\tilde{s_a}, \tilde{s_b})$ is added if and only if there exists a path in the Kripke structure K_a from $s_{a0} \in S_a$ to a node $s_b \in S_a$ satisfying $\pi_b \in \mathcal{L}(s_b)$.

The deterministic μ-calculus model checking algorithm `ModelCheck` verifies whether the abstracted Kripke structure $\tilde{K}$ satisfies the given specification, Φ. If the specification is not satisfied, the while-loop repeats lines 8–11.

Once the specification is found to be satisfied by the abstracted Kripke structure $\tilde{K}$, a time-parameterized trajectory is created from the combination of the best paths found among the relevant Kripke structures K_i (line 12).

Note that in order to use the path returned by Algorithm 8, we track it using an LQR controller obtained by linearizing the dynamical system (3.1) at the current state. The state error can then be obtained and the appropriate feedback control can be applied until the next time step, at which point the LQR algorithm must be run again.

3.3.3 LQR Tracking

Once the abstracted Kripke structure $\tilde{K}_{\Pi^+(\Phi)}$ is generated and the model checking algorithm confirms the satisfaction of the μ-calculus specification Φ, it is necessary to connect the candidate trajectories from the various K_i structures. Accordingly, for all paths in $\tilde{K}_{\Pi^+(\Phi)}$ satisfying Φ, the corresponding candidate trajectories in the K_i structures are

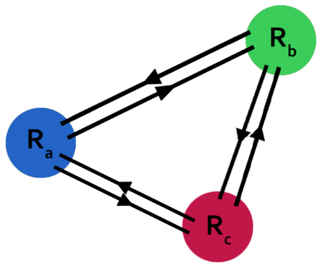

Figure 3.1: Representation of the abstracted Kripke structure of the provided example. This structure is verified with the local model checking algorithm (Algorithm 7) to ensure satisfaction of the deterministic μ-calculus specification.

bridged together within the appropriate proposition region, i.e., the region in state space where every state x satisfies a particular proposition $p \in \Pi^{+}(\Phi)$. To do so, we apply an LQR controller as in [28].

First, the dynamical system is linearized to be of the form $\dot{x} = Ax + Bu$. We then define the quadratic cost function over the time interval $[t_0, t_1]$ to be

$$J = \int_{t_0}^{t_1} \left(\bar{x}^T Q \bar{x} + \bar{u}^T R \bar{u}\right) dt \tag{3.2}$$

where Q is a symmetric positive semi-definite state cost matrix, R is a symmetric positive definite control cost matrix, the tracking error is $\bar{x} = x - x_c$ where x_c is the time-parameterized total candidate trajectory found by patching together the individual candidate trajectories in sequence, and $\bar{u} = u - u_c$ where u_c is the corresponding control signal for x_c. To proceed, the steady-state solution P to the continuous algebraic Ricatti equation (CARE)

$$A^T P + PA - PBR^{-1}B^T P + Q = 0 \tag{3.3}$$

must be found. The feedback control is then given by $u = u_c - K\bar{x}$, where $K = R^{-1}B^T P$.

3.4 Example

To demonstrate the effectiveness of our method, we provide the following pertinent example. We use continuous double integrator dynamics on two spatial dimensions, resulting in a 4D state space. State and control vectors take the form

$$\vec{x} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}, \ \vec{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} \tag{3.4}$$

and the dynamical system is given by

$$\dot{\vec{x}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \vec{x} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \vec{u}, \tag{3.5}$$

with initial condition $\vec{x}_0 = [100, 400, 0, 0]$. We choose the cost function for SST* to be the duration of the trajectory, T, along with a control cost term, with cost matrix R_{sst}:

$$J_{SST} = \int_0^T \left(1 + u^T R_{sst} u\right) dt. \tag{3.6}$$

The specification we wish to satisfy is to visit three distinct regions of the state space, R_a, R_b, and R_c infinitely often while avoiding the obstacle regions collectively called R_o. Define atomic propositions p_i, $i \in \{a, b, c, o\}$, such that $p_i \in \mathcal{L}(s)$ if and only if $s \in R_i$. We write the deterministic μ-calculus formula Φ as follows

$$\begin{aligned} &\mu M.[(\neg o \wedge \Diamond M)\vee \\ &\qquad \nu W.\{(a \wedge \mu X.[\neg o \wedge (((b \vee c) \wedge W) \vee \Diamond X)])\vee \\ &\qquad\quad (b \wedge \mu Y.[\neg o \wedge (((a \vee c) \wedge W) \vee \Diamond Y)])\vee \\ &\qquad\quad (c \wedge \mu Z.[\neg o \wedge (((a \vee b) \wedge W) \vee \Diamond Z)])\}]. \end{aligned}$$

Upon running SST* in three parallel instances starting in the center of each proposition's associated region in state space, we obtained Figure 3.2. Note that there are six candidate trajectories, each representing the best path from one region to another in terms of the cost, J_{SST}. In general, for p positively-appearing atomic propositions in the μ-calculus specification, p trees are incrementally updated in parallel, and we search for $p - 1$ candidate trajectories for each tree.

The model checking algorithm ensures that at least one cycle of length three may be formed in the abstracted Kripke structure, which itself is constructed with three nodes (one for each proposition region), and whose directed edges represent the candidate trajectories that begin in one proposition region and end in another one. The procedure was run until all six candidate trajectories successfully reached their appropriate goal regions, allowing for the abstracted Kripke structure to be fully connected[1]. The resulting structure contains two infinite paths satisfying proposition Φ above, noting that the initial condition is situated in the blue region: (i) head to the green region first, then the burgundy region, and returning to the blue region to repeat, or (ii) head to the burgundy region first, then the green region, and repeating upon returning to the blue region.

In order to determine which of the cycles to take, a simulation is run using the LQR controller discussed in Subsection 3.3.3, tracking each of the proposed paths and bridging the gap between the end of one trajectory and the beginning of another. The state-cost matrix Q was set to $\texttt{diag}(25, 25, 25, 25)$ and the control-cost matrix U was chosen to be $\texttt{diag}(1, 1)$. Using total cost to determine the better of the two proposed solutions, it is found that option (i) results in a faster circuit between the three regions. See Figure 3.3.

3.5 Conclusions

The method presented here is a novel use of SST*, a kinodynamic planning algorithm that avoids reliance on a steering function, and μ-calculus model checking that uses an abstracted Kripke structure to determine satisfaction of deterministic μ-calculus specifications. Construction of the abstracted Kripke structure is performed by creating multiple Kripke structures and merging the most cost-efficient solutions into one structure, and the appropriate trajectories satisfying the given specification are tracked using an LQR feedback control policy.

This research opens many avenues for future work. Investigation into other types of feedback controllers may help to improve tracking, especially for nonlinear systems. Tracking also introduces the problem of collision avoidance, since despite guaranteeing a collision-free trajectory with an appropriate μ-calculus specification, tracking errors may yet cause collisions. Furthermore, completeness of our method remains to be proven given the multilayer approach, as SST* guarantees (probabilistic) completeness for each individual Kripke structure, however it is uncertain what can be said of the use of multiple Kripke structures in satisfying a single specification.

[1]This was done in order to compare the cost associated with each direction of the possible 3-cycle, although Algorithm 8 as it is written would return as soon as any satisfactory path is found.

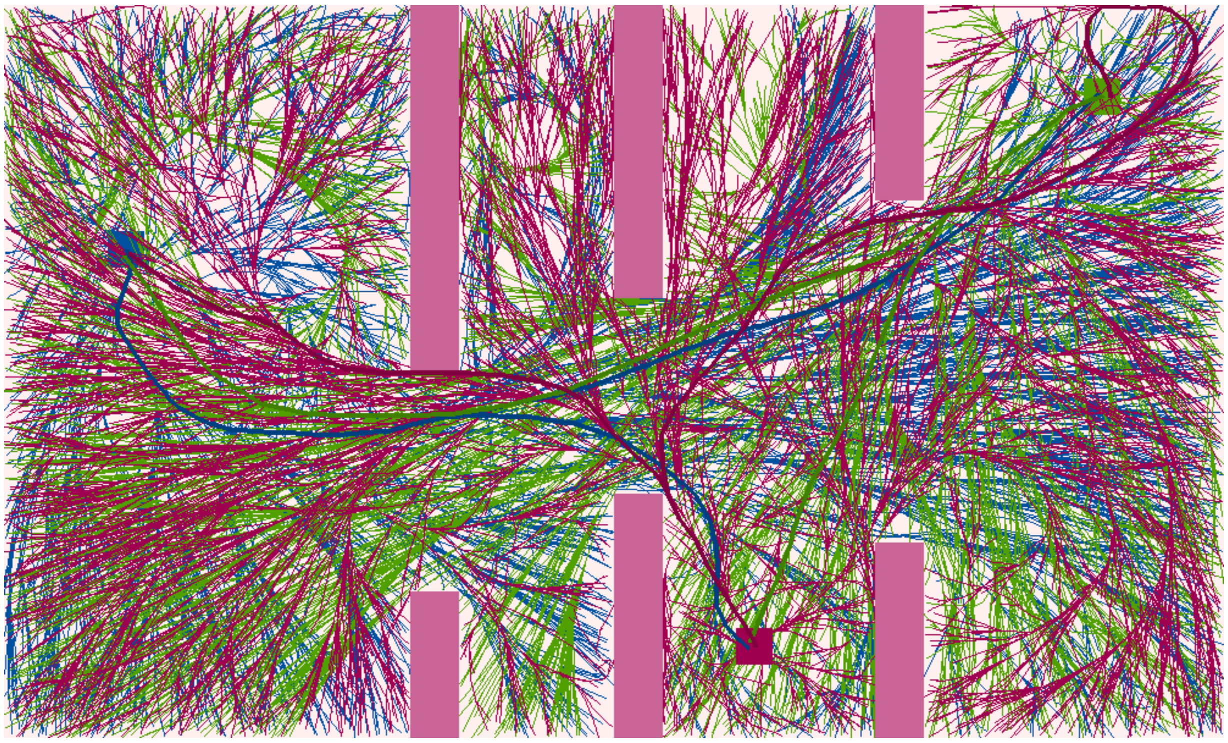

Figure 3.2: SST* is performed three times, producing a Kripke structure for each of the regions shown here in blue (left), burgundy (bottom center), and green (top right). Obstacles are represented as pink rectangles. The color of each line matches the color of the region of the Kripke structure to which it belongs, and the bolded curves are the lowest-cost trajectories that reach another region.

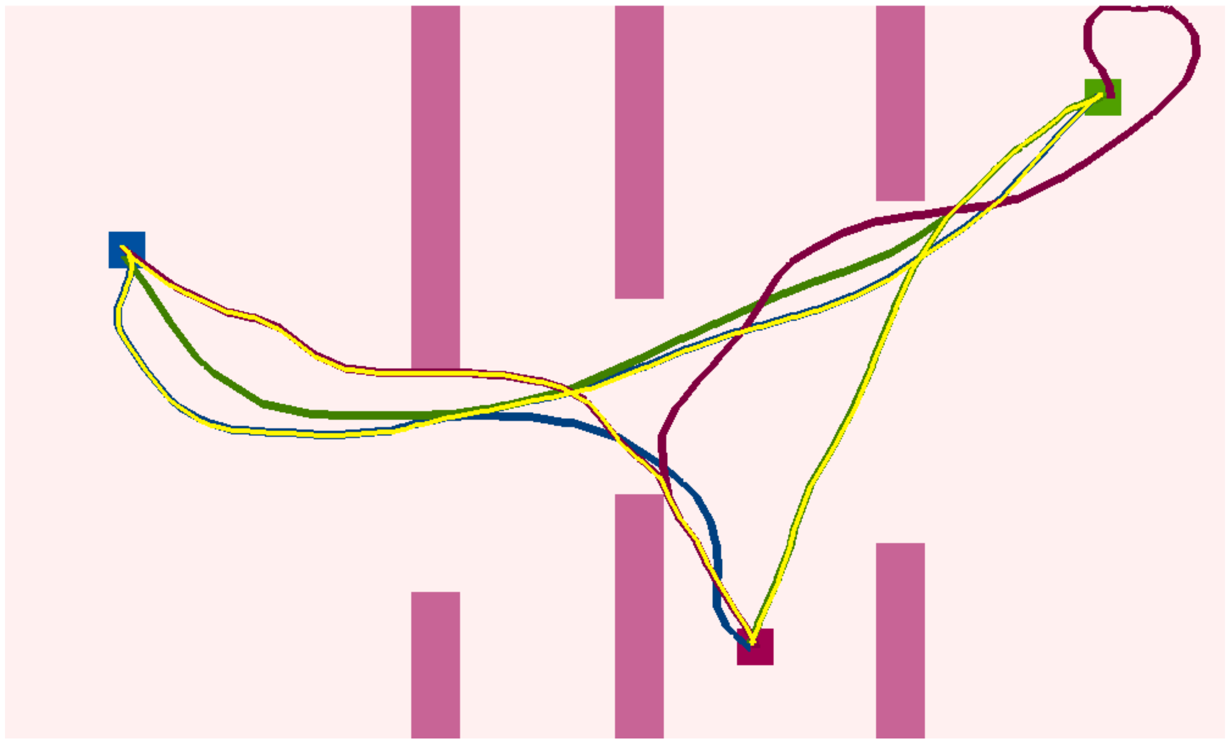

Figure 3.3: The six candidate trajectories are shown, where curves of the same color are selected from the same Kripke structure. The solution trajectory is shown in yellow, starting in the center of the blue region and tracking the infinite path with least cost that satisfies specification Φ.

Chapter 4

Quadrotor Motion Planning

Quadrotors, as discussed in Chapter 1, are growing in popularity for their many uses. Autonomous navigation for quadrotors is still in the early stages of development, although there are already some basic autonomous behaviours in use commercially; for example, many drones now support following a moving person to capture video footage. What makes quadrotor motion planning a truly interesting challenge is the fact that they occupy a 12D state space, and they are non-holonomic vehicles that are governed by nonlinear dynamics. The combination of high-dimensionality and nonlinearity render many current methods ineffective in the context of motion planning, for example the interval method [11, 22], which discretizes the state and control space with intervals, and suffers from the curse of dimensionality [8].

Before proceeding to the mathematics involved, it may be of interest to the reader to discuss our choice to use the term "quadrotor". But first, we begin by seeing that the word "helicopter" can be broken down into "helico", which is itself a combining form of "helix" (screw), and "pter" meaning "wing". Some people have referred to four-rotor helicopter UAVs as "quadcopters", but this terminology is ignorant of the Ancient Greek etymology of such words, as "heli/copter" is not the correct partitioning of the English word. So, the term "quadrotor" avoids the issue altogether, and its meaning is self-evident.

Now, in order to overcome the hurdles inherent in motion planning for such a complex system as a quadrotor, Ross Allen and Marco Pavone put forth a "full-stack approach" for real-time kinodynamic planning of such aerial vehicles [1]. First and foremost, a sampling-based planning approach was deemed necessary to deal with unknown environments online, and while SST* can handle the high-dimensionality and nonlinearity of the quadrotor system without needing a steering function, the incremental nature cripples its ability to plan effectively in an online setting. On the other hand, FMT*, with its pre-sampled states and its flexibility in allowing to perform much of the necessary pre-computation offline, is much more suited to online planning. For this reason, the authors centre their planning framework around a variant of FMT* called `kinoFMT`, introduced in Chapter 2. Once an approximate trajectory is found using this planning algorithm, trajectory smoothing is applied, and due the differentially flat nature of the quadrotor dynamics, it is then feasible to track the smooth trajectory with the proposed controller.

This chapter begins by analyzing the dynamical system that models quadrotor dynamics. Then, much of the work from [1] is described in various levels of detail, and conclusions are drawn from the efficacy of this approach. Finally, the main idea of using high-level temporal logic specifications using an abstracted Kripke structure, presented in Chapter 3, is applied to quadrotor kinodynamic planning *with* steering under the aforementioned real-time planning framework.

4.1 Quadrotor Model

Before delving into the laws of motion for quadrotor systems, we begin by defining the underlying coordinate systems used in our analysis. In order to describe the position and velocity of the quadrotor, we need a fixed inertial reference frame (sometimes called the world frame) where Newton's laws hold. We will use the North-East-Down (NED) configuration for this purpose with basis vectors $\{e_1, e_2, e_3\}$. Along with the inertial frame, we define a body-fixed frame with basis $\{b_1, b_2, b_3\}$ whose origin coincides with the centre of mass of the quadrotor. The vectors b_1 and b_2 lie in the plane formed by the four rotors, and b_3 points in the direction opposite the applied thrust. See Figure 4.1.

Controlling a quadrotor involves adjusting the thrust applied to each of the four rotors. Note that opposite rotors rotate in the same direction, and adjacent rotors rotate in opposite directions; consequently, if all four rotors apply the same amount of thrust, the quadrotor will fly directly upward, and the angular momentum contributed by each rotor cancels so that there is zero rotational motion. Quadrotor motion is described in the space of all rigid body motions, namely the special Euclidean group SE(3). This space has six degrees of freedom: translation in three dimensions, and rotation about each of the three body-fixed axes. However, given that there are only four control inputs, the quadrotor system is underactuated.

Rotational motion is often described by the Euler angles measuring yaw (about b_3), pitch (about b_2), and roll (about b_1). The use of Euler angles as state variables is not

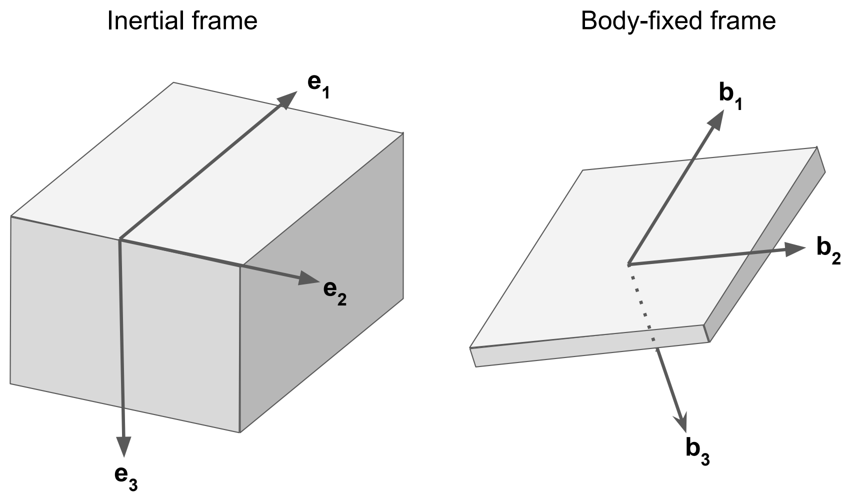

Figure 4.1: The inertial frame and the body-fixed frame are shown here, abiding by the NED convention.

ideal, though, as singularities and jump-discontinuities arise as a result of restricting the domain of such angles. Recent work by Taeyoung Lee et al. instead takes a geometric control approach with a globally defined model to avoid such issues [20], and we will make use of their work in modeling quadrotor dynamics. The important change they make is to replace the three Euler angle state variables with a single 3×3 matrix in the special orthogonal group, SO(3), defined as follows:

$$\mathrm{SO}(3) = \{R \in \mathbb{R}^{3\times 3} \mid R^T R = I,\ \det(R) = 1\}. \tag{4.1}$$

The elements $R \in \mathrm{SO}(3)$ are called rotation matrices, and they describe the current (or desired) attitude of the quadrotor. Given a vector $\vec{v}$ in the inertial frame,

4.2 Real-Time Motion Planning

Blah.

References

[1] Ross Allen and Marco Pavone. A Real-Time Framework for Kinodynamic Planning with Application to Quadrotor Obstacle Avoidance. *{AIAA} Conf. on Guidance, Navigation and Control*, pages 1–18, 2016.

[2] A. I. Medina Ayala, S. B. Andersson, and C. Belta. Temporal logic motion planning in unknown environments. *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5279–5284, 2013.

[3] Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. Sampling-based motion planning with temporal goals. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 2689–2696, 2010.

[4] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 368–377, 1991.

[5] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. *On model checking for the μ-calculus and its fragments*, volume 258. 1999.

[6] Melissa Greeff and Angela P. Schoellig. Model Predictive Path-Following for Constrained Differentially Flat Systems. In *IEEE International Conference on Robotics and Automation*, Brisbane, Australia, 2018.

[7] Arie Gurfinkel and Marsha Chechik. Extending extended vacuity. *Formal Methods in Computer-Aided Design*, pages 306–321, 2004.

[8] Piotr Indyk and Rajeev Motwd. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *Proceedings of the thirtieth annual ACM symposium on Theory of computing. ACM*, pages 604–613, 1998.

[9] SAE international. U.S. Department of Transportation's New Policy on Automated Vehicles Adopts SAE International's Levels of Automation for Defining Driving Automation in On-Road Motor Vehicles. *SAE international*, page 1, 2016.

[10] Lucas Janson, Edward Schmerling, Ashley Clark, and Marco Pavone. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *International Journal of Robotics Research*, 34(7):883–921, 2015.

[11] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis*. Springer London, London, 2001.

[12] T. Jochem, D. Pomerleau, B. Kumar, and J. Armstrong. PANS: a portable navigation platform. *Proceedings of the Intelligent Vehicles '95 Symposium*, pages 107–112.

[13] Sertac Karaman and Emilio Frazzoli. Sampling-based motion planning with deterministic mu-calculus specifications. *Proceedings of the 48th IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, (0):2222–2229, 2009.

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

[15] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning with deterministic mu-calculus specifications. *2012 American Control Conference*, 2012.

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

[17] Luc Larocque and Jun Liu. Sampling-Based Motion Planning with mu-Calculus Specifications without Steering. In *IEEE International Conference on Robotics and Automation*, Brisbane, Australia, 2018.

[18] S. M. LaValle and James J Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, 2001.

[19] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 1 edition, 2006.

[20] Taeyoung Lee, Melvin Leok, and N. Harris Mcclamroch. Geometric Tracking Control of a Quadrotor UAV on SE (3). *49th IEEE Conference on Decision and Control*, (3):5420–5425, 2010.

[21] Yanbo Li, Zakary Littlefield, and Kostas E. Bekris. Asymptotically optimal sampling-based kinodynamic planning. *The International Journal of Robotics Research*, 35(5):528–564, 2016.

[22] Yinan Li and Jun Liu. ROCS: A Robustly Complete Control Synthesis Tool for Nonlinear Dynamical Systems. *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week) - HSCC '18*, pages 130–135, 2018.

[23] Hai Lin. Mission Accomplished: An Introduction to Formal Methods in Mobile Robot Motion Planning and Control. *Unmanned Systems*, 02(02):201–216, 2014.

[24] Jerome M. Lutin, Alain L. Kornhauser, and Eva Lerner-Lam. The revolutionary development of self-driving vehicles and implications for the transportation engineering profession. *ITE Journal (Institute of Transportation Engineers)*, 83(7):28–32, 2013.

[25] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial trajectory planning for quadrotor flight. *International Conference on Robotics and Automation*, (Isrr):1–16, 2013.

[26] Klaus Schneider. *Verification of Reactive Systems*. Springer Science & Business Media, 2004.

[27] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[28] Russ Tedrake. LQR-trees: Feedback motion planning on sparse randomized trees. *Robotics: Science and Systems V*, page 8, 2009.

[29] Sebastian Thrun. Toward robotic cars. *Communications of the ACM*, 53(4):99–106, 2010.

[30] Dustin J. Webb and Jur Van Den Berg. Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 5054–5061, 2013.

[31] Alexander Wendel and James Underwood. Self-supervised weed detection in vegetable crops using ground based hyperspectral imaging. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5128–5135, 2016.

[32] Thomas Wilke. Alternating tree automata, parity games, and modal Âţ-calculus. *Bull. Soc. Math. Belg*, 8(2):2001, 2001.

[33] Eric M Wolff. *Control of Dynamical Systems with Temporal Logic Specifications*. PhD thesis, California Institute of Technology, 2014.

[34] Christopher Xie, Jur van den Berg, Sachin Patil, and Pieter Abbeel. Toward Asymptotically Optimal Motion Planning for Kinodynamic Systems using a Two-Point Boundary Value Problem Solver. *IEEE International Conference on Robotics and Automation*, pages 4187–4194, 2015.

[35] Kwangjin Yang, Seng Keat Gan, and Salah Sukkarieh. A Gaussian process-based RRT planner for the exploration of an unknown and cluttered environment with a UAV. *Advanced Robotics*, 27(6):431–443, 2013.

APPENDICES

Appendix A

Matlab Code for Making a PDF Plot

A.1 Using the GUI

Properties of Matab plots can be adjusted from the plot window via a graphical interface. Under the Desktop menu in the Figure window, select the Property Editor. You may also want to check the Plot Browser and Figure Palette for more tools. To adjust properties of the axes, look under the Edit menu and select Axes Properties.

To set the figure size and to save as PDF or other file formats, click the Export Setup button in the figure Property Editor.

A.2 From the Command Line

All figure properties can also be manipulated from the command line. Here's an example:

```
x=[0:0.1:pi];
hold on % Plot multiple traces on one figure
plot(x,sin(x))
plot(x,cos(x),'--r')
plot(x,tan(x),'.-g')
title('Some Trig Functions Over 0 to \pi') % Note LaTeX markup!
legend('{\it sin}(x)','{\it cos}(x)','{\it tan}(x)')
hold off
```

```
set(gca,'Ylim',[-3 3]) % Adjust Y limits of "current axes"
set(gcf,'Units','inches') % Set figure size units of "current figure"
set(gcf,'Position',[0,0,6,4]) % Set figure width (6 in.) and height (4 in.)
cd n:\thesis\plots % Select where to save
print -dpdf plot.pdf % Save as PDF
```