Model-Accuracy Aware Anytime Planning with Simulation Verification for Navigating Complex Terrains

Manash Pratim Das

CMU-RI-TR-22-25

April 27, 2022

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Maxim Likhachev, CMU RI, *chair*
Michael Kaess, CMU RI
Shivam Vats, CMU RI

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

To my Aita and Dhunuma.

Abstract

Off-road and unstructured environments often contain complex patches of various types of terrain, rough elevation changes, deformable objects, etc. An autonomous ground vehicle traversing such environments experiences physical interactions that are extremely hard to model at scale and thus very hard to predict. Nevertheless, planning a safely traversable path through such an environment requires the ability to predict the outcomes of these interactions and generate some form of a traversability map that standard path planning algorithms can utilize.

This thesis proposes an algorithmic framework for planning to navigate complex terrains. The key challenge, as mentioned above, is to know what parts of the map are traversable as quickly as possible because the planning algorithm would require this information. The method to estimate traversability is generally slow, e.g., a physics-simulator. However, we do not need traversability information for the entire map; instead, it is sufficient to cover only the relevant regions for a path-planning problem involving a start and a goal state pair. Our work tightly couples the process of building the traversability map with the graph search involved in planning for a traversable path.

Firstly, we relate traversability to the validity of an edge in the planning graph. This allows us to look at lazy motion planning algorithms that are efficient with edge evaluation. We introduce two edge-evaluators, 1) a high-fidelity physics simulator which we assume to be always correct, and 2) a learned model which can make errors but has orders of magnitude faster inference time.

Secondly, we develop an anytime planning algorithm that is aware of the learned model's errors and utilizes it wherever possible instead of the slower simulation model to speed up planning. Our algorithm, which we call MA3, utilizes the simulator efficiently only to verify paths before returning and to correct the learned model's potential errors.

We provide a theoretical analysis of the algorithm, and its empirical evaluation shows a significant reduction in planning times against contemporary lazy planning algorithms. We believe that our contribution takes a step towards autonomous navigation in complex off-road environments.

Acknowledgments

I would like to begin by thanking my advisor, Prof. Maxim Likhachev. His belief and trust in me when I required the most has helped me fulfill my dream of becoming a roboticists. Three years back in 2019, when I was about to graduate with an undergraduate degree in Biotechnology, I doubted myself whether I would ever be accepted as a roboticists even if I studied and practiced the discipline myself. Although I had tremendous interest in making this switch of discipline from Biotechnology to Robotics, and I had prepared myself for it, but it was hard for me to find someone who would spend the time to look beyond my degree and evaluate me for my knowledge and skills. I thank Prof. Likhachev for taking the leap of faith in me and welcoming me into his lab as a Research Associate. The raw experience and knowledge I gained eventually helped me get accepted and complete my Masters in Robotics. I would truly be ever grateful to him. I would also like to thank him for being the best advisor that I have known. His commitment to his students is phenomenal. Under his purview, not only did I learn the subject, but I was heavily influenced by his critical thinking and methods to communicate research with a variety of audience. I could not thank him enough, and all I can hope is that I put my best efforts to carry his influence in my future endeavors.

I would like to thank Shreyansh Daftry who constantly stood by my side as a mentor and a friend even after my intership with him at NASA, JPL in 2018.

Contents

When this dissertation is viewed as a PDF, the page header is a link to this Table of Contents.

List of Figures

List of Tables

Chapter 1

Introduction

In this work we look at the problem of autonomous navigation in complex off-road terrains. While this problem involves many aspects, our primary focus is on the algorithms that compute motion plans that the autonomous robot can execute to navigate safely over such terrains.

Motion planning is one of the fundamental problems of robotics. It involves reasoning about the world as perceived and making decisions to fulfill a motion objective. It is, in essence, core to artificially intelligent decision-making. In defining a motion objective, we have to specify a state representation containing the configuration of all the agents, including the robot that can participate in the motion. And specify actions that allow transitions from one state to another. One may choose to use discrete variables to represent the states and a finite set of actions, in which case, the traditional approach is to generate a graph and search over it to find the motion plan. There are several textbooks dedicated to motion planning [17, 18], and specifically for planning as search over a graph with heuristics [3, 22].

In this thesis, we focus on search-based motion planning problems. The class of problems aims to search for a feasible path in a graph from a vertex defined as "start" to a vertex defined as "goal". A feasible path consists of a sequence of consecutive edges that are "valid". Each edge will have a cost associated with it. The cost of an edge can also be infinite, which indicates that the edge is not valid. The cumulative sum of the costs associated with the edges that form a path in the graph refers to the path's cost. While the cost of an edge can be negative, in the scope of this work,

Figure 1.1: An example of a human-driven vehicle driving over an off-road terrain. Courtesy of `offroadxtreme.com`.

we assume that the cost is always non-negative.

Moreover, we focus on the sub-class of search-based planning problems, where it is time-consuming to determine the true cost of an edge. Alternatively, if we initially assume that all edges in the graph are valid, it is expensive w.r.t. time to determine if an edge is invalid. The motivation for this work arises from the fact that autonomous navigation of a robot in an off-road terrain requires calling a time-consuming physics-based simulator (as argued below) to evaluate the cost of traversing along an edge. We use the term "robot" and "vehicle" interchangeably throughout the thesis.

In an unstructured off-road environment, the robot can encounter a combination of environmental elements that generate a complex interaction. For example, the environment may contain rocks, gravel, mud, ridges, rough terrain, or deformable objects like tall grass, bushes, low-hanging tree branches, etc. The robot would encounter very complex physical interactions with them, making the interaction practically infeasible to track or predict in closed form. Figure 1.1 shows an off-road vehicle driving over rough terrain. Understanding the physical interactions between the tire, terrain elements, suspension, and power unit is important. In the graph-based setting, to plan a path for the vehicle through this terrain, we can set up a state-lattice graph [24] that spans the region to be traversed. The vertices in this graph represent the states of the robot over the terrain, and the edges will represent a sequence of steering and throttle inputs that "can" make the robot traverse between states. Unlike planning for self-driving vehicles that drive in urban environments, planning

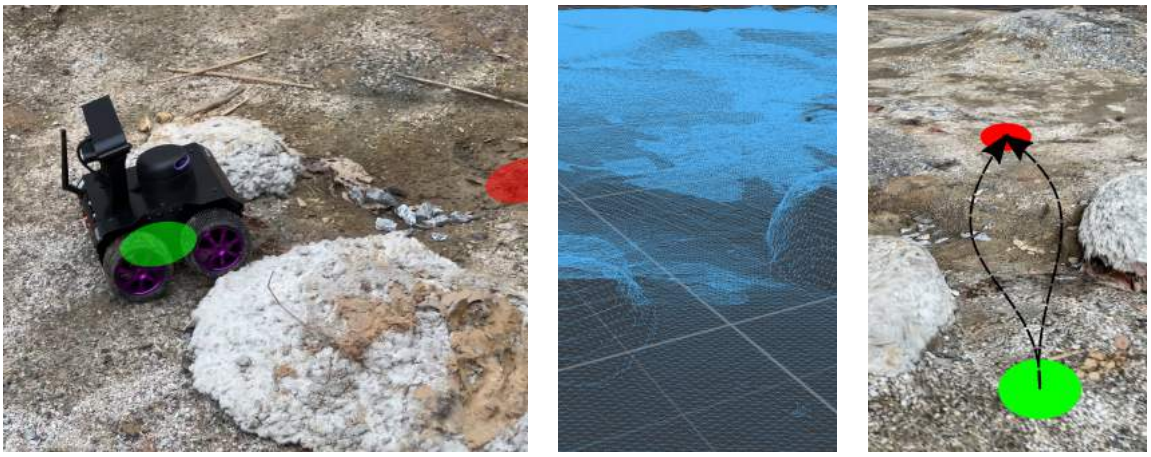

Figure 1.2: An example of a human-driven vehicle driving over an off-road terrain. Courtesy of `offroadxtreme.com`.

for navigating off-road environments requires information about traversability. The Planner would need an "expert" that can predict whether the robot "will" be able to traverse between two states represented in the graph (see Figure 1.2).

We use the term "expert" to refer to an entity that can predict traversability and, in particular, whether an edge is valid in the context of the graph we defined above. Historically analytical models [1, 10], machine learning models [4, 13], and simulation-based [2, 5] models have been popular choices to serve as "experts". Nampoothiri et.al. [21] provides a recent survey of models used in traversability prediction. There is a trade-off between the time required to query a model and its accuracy. Consider the following example: A neural network model can look at the scene in front of the vehicle using onboard sensors to quickly predict traversal. However, their prediction is based on interpolation, which does not reason about the complex physics of interaction. Therefore, their output is only reliable for inputs that are very similar to the data used to train them. On the other hand, a high-fidelity simulator can be engineered to replicate an off-road scene and simulate the complex dynamics for a more reliable prediction.

Figure 1.3: A sequence of images showing a racer riding a motocross bike up the rocky terrain. Courtesy of `https://youtu.be/30-I-SXlwEM`.

1.1 Observations

Based on the above discussion, we present the following observations which guided the development of our work:

1. *Physics-based reasoning is required for traversability estimation*:

   We use a few snapshots from a Motocross race in Figure 1.3 to motivate this point. Assume that we want an autonomous robot to drive over this terrain. Also, assume that in preparation, we have provided the robot with a list of rules that the robot can follow based on the terrain elevation in front of it to determine how to steer and accelerate. Just like the terrain as shown in the images, the world can throw at the robot an environment that is significantly different in its elevation profile than what the list of rules prepares the robot for. Therefore, if the robot has a way to replicate the elevation profile in a physics-based simulator, it could roll out a bunch of simulations to determine what sequence of actions will work. As shown in the images, the racer would

understand the physics and have an intuition of how the motorbike's steering, suspension, and the engine would react to the control inputs.

2. *Planner can inform us about the regions where traversability information is required*:

   A* search [12] uses a heuristic function to evaluate whether a vertex in the graph is important to find the shortest path between the current state of the robot and the desired goal state of the robot. And given an admissible heuristic function, the search is proven to look at the minimal number of states required to find the shortest path. Thus, if we only need the traversability map for navigation purposes, we can focus the efforts of the simulator to obtain traversability information in only the region where the Planner would expand. Concretely, if the cost of the shortest path is c_{π^*} between the start state s_{start} and goal state s_{goal}, A* would expand only the states s which have an f-value

   $$f(s_{\text{start}}, s, s_{\text{goal}}) \leq c_{\pi^*}$$

   where $f(s_{\text{start}}, s, s_{\text{goal}}) = g(s_{\text{start}}, s) + h(s, s_{\text{goal}})$, which is the sum of 1) the cost of the shortest path from the start to the state s given by g, and 2) and underestimate to the cost of the shortest path from the state s to the goal given by the heuristic function h.

   Moreover, since the evaluation of an edge is expensive, and A* would evaluate all the edges coming out of a state when the state is expanded, we can look at the class of algorithms which are proven to evaluate a minimal number of edges in the graph to find the shortest path. This class of planning algorithms is called "Lazy" and is discussed in detail in Chapter 2. They are guaranteed to be optimal in terms of the edge evaluations and have the potential to expand a lesser number of edges than that of A*.

3. *Neural Networks can approximate the simulator and with a much quicker inference time*:

   Theoretically, feed-forward neural networks are universal approximators [14]. Essentially, it can be trained to mimic the output of the simulator such that a wide range of common traversability queries can be cached. Suppose the

Planner queries the simulator to know if the robot can traverse a certain simple off-road terrain that occurs very often. The data generated from the simulator can be used to train the neural network such that next time a similar terrain is observed, we can obtain traversability information from the neural network instead of calling the simulator again. Thanks to the generalizability of neural networks [16], the trained neural networks would also allow approximation to be applied to a wide input space even with sparse training data from the simulator. Although, having access to a simulator means that a lot of training data can be generated to train the network offline and prepare it better when it is required for online planning. Chapter 5 provides an example of how we generated simulation data offline prepared our neural network. However, a major flaw with that the output of the network can be brittle.

4. *A bounded-suboptimal plan, if found quicker, may be preferred over an optimal plan*:

   For the problem of off-road navigation, if we had two options, namely 1) follow a path to the goal that the Planner found within a few seconds, vs. 2) wait for a few extra minutes to search for a path that is guaranteed to be the shortest path, which one would you choose? It can be argued that the first option is more preferred in this context. However, the user should be able to specify an upper bound on the solution quality to prevent arbitrary bad solutions. This implies that our algorithm should have anytime properties and guarantee that the solution is no worse than the threshold set by the user.

5. *Constant local control adjustments are required to navigate complex off-road terrain along with a robust global plan*:

   Consider a four-wheeled robot driving over hilly off-road terrain. We have observed that minor differences in tire placement might lead to widely different interactions and outcomes. Usually, the tire placement and the physical interactions during the execution in the robot will have some difference as compared to the simulation observed during planning. This can be attributed to the sim-to-real gap and the difference in planning and execution in general. Therefore, global planning should have some robustness to minor execution differences.

1.2 Key Ideas

1. *Train a neural network offline using data from simulation for commonly occurring off-road patches*:

   This idea follows from Observation 3. We procedurally generate a wide variety of environments and run simulations to collect data for training the neural network. The intention here is to prepare the neural network for the inputs the learned model would observe during online planning. We can always query the simulator for the cases when we think that the learned model may be wrong. To this end, we require the learned model to provide some kind of a confidence score about its prediction. As discussed in Chapter 5, we use an ensemble of neural networks to compute the confidence score.

2. *We only need to manage the cost of the edges and keep the structure of the graph constant*:

   There are two methods of using the simulator. One method would be to generate the search graph, and the other would be to generate an optimistic graph and only run the simulator to invalidate parts of this optimistic graph. We use the latter method as it allows us to keep the structure of the graph constant and only worry about managing the costs of the edges. This simplifies the theoretical analysis, and the method allows us to leverage observation 2.

3. *Use the simulator as the ultimate source of traversability information*:

   Following observation 1, we make a design choice that the simulator is our ground-truth traversability estimator during planning. We enforce that a plan must be verified in simulation before executing it in the real robot. We acknowledge that there would be a sim-to-real gap. However, we will assume that the simulator is accurate for the purpose of this thesis. In Chapter 4 we discuss our efforts to design and develop a high-fidelity off-road simulator.

   The simulator also helps us correct mistakes that the learned model might have made due to its brittleness. This ensures that regardless of how bad the learned model is if a path exists, it will always be found.

4. *Use two threads for planning*:

Observations 1 and 3 show that the simulation-based model of the robot and the learned model have complementary properties. We allow the search to continue exploring the graph using the learned model in a separate thread while the simulator is processing a query in another thread. Moreover, from observation 4, we know that while the simulator is busy processing a query that might lead to the discovery of the optimal path, we would prefer to continue the search for a sub-optimal path in parallel.

5. *Use parallelization in GPUs to query around an edge with no additional time*: Recall that a query to evaluate an edge evaluates whether the robot will be traverse between the two states connected by the edge. In order to have a robust plan, we also evaluate whether the edge is still valid with some minor perturbations in the location of the states. All of these queries can be made in parallel very efficiently and in no additional time cost.

1.3 Contributions

1. Simulation-assisted planning for accomplishing complex off-road navigation requires access to a high-fidelity simulator that accurately can simulate the effect of an action. We believe that an off-road driving such a simulation framework must support certain basic features, including 1) the ability to generate complex 3D landscapes with physical components such as rocks, grass, snow patches, etc., 2) accurate modeling of road-tire interaction for various off-road terrains, 3) accurate modeling of complex vehicle physics for suspension, transmission, and engine power, 4) ability to simulate various weather conditions, etc. Unfortunately, there is no off-the-shelf and open-source simulator that supports these features. We developed a driving simulator ISimA based on Unreal Engine 4 that particularly targets multi-terrain off-road navigation with support for the above-mentioned features along with certain additional features inherited from the UE4, including 1) photorealistic rendering of simulation world and vision sensors, 2) simulation support for various perception sensors common to self-driving vehicles such as LiDARs, RADAR, Event Camera, GPS, IMU, etc., and 3) a physics simulation engine that runs on GPU providing near

real-time performance. Our simulator also supports CARLA, such that APIs to control the vehicle and other agents in the world, such as pedestrians and other vehicles developed in CARLA, can be used with our simulator.

2. We propose an anytime planning algorithm MA3 that can utilize a learning-based and a simulation-based "expert" to evaluate edges. The learning-based evaluator is approximate but quicker, while the simulation-based evaluator is accurate. The key idea is to use the quicker edge-evaluator to guide the search and use the accurate edge-evaluator "minimally" to verify the solution or correct mistakes that the approximate evaluator might have made. Accounting for the mistakes helps MA3 ensure completeness and bounded-suboptimality guarantees.

Chapter 2

Background

We briefly discuss the related works in the literature.

2.1 Search Based Motion Planning

Algorithms that search for a solution by traversing a graph typically use a heuristic function such as for A* [12]. The *anytime* variants of A* such as Anytime Repairing A* [19] returns a sub-optimal solution quickly by inflating the heuristics and iteratively reducing the inflation to find potentially better paths given more time. However, anytime algorithms are only opportunistic and do not guarantee that a bounded-suboptimal solution will be found within a strict time budget.

2.2 Lazy edge-evaluations

For search-based planning problems where edge evaluations dominate the run-time of the planner, LazySP [20] class of algorithms are proven to evaluate the minimum number of edges (edge-optimal) [11] required to find the optimal path. The main idea behind this class of algorithms is to generate path candidates in order of their potential to be the optimal path and to keep eliminating them if they are invalid until you find the first valid path. Enforced by the order in which these candidates are considered, the first valid path is guaranteed to be optimal. These algorithms are edge-optimal as they use optimistic edge costs to search the graph for the candidate

paths. And query a time-consuming edge evaluator only to eliminate candidates. LazySP utilizes a shortest path planner like A* [12] internally. A variant of A* which aims at reducing edge-evaluations is LazyA* [6].

Our work builds upon the LazySP framework such that in addition to the expensive but error-free edge-evaluator, the planner can also utilize an error-prone but relatively faster edge-evaluator. While [6, 11, 20] are designed to search directly for the optimal path, our Algorithm 1 is an Anytime algorithm. Note that PSMP [15] is also based on LazySP and is an anytime algorithm. However, it is not related to our problem. PSMP utilizes additional global information about the map to accelerate the elimination of path candidates. However, MA3 does not require global information; rather, the additional edge-evaluator only uses local information around the edge. Moreover, PSMP only affects the LazyEdgeSelector function (see Algorithm 1) which we borrowed from LazySP, and hence MA3 can be extended like PSMP to utilize global information.

2.3 Off-road Simulation

Before we started designing the ISimA system, we first evaluated the existing open-source simulation software. We looked at all of the features required to adequately support our goal of simulating an off-road vehicle. In particular, we evaluated each candidate under the following criteria:

1. Physics Engine – fidelity and accuracy
2. Terramechanics – modeling of the tire-ground interface, physics support for deformable surfaces, off-road capabilities, difficulty in adding new tire-ground interface models
3. Customization – ability to generate custom maps and environments, modeling scenarios
4. Open-source/licensing – what are the restrictions on licensing
5. ROS support – capabilities to integrate with ROS or other middleware
6. Photo-realistic rendering – ability to support high-definition camera models for accurate simulation of camera-based sensors

7. GPU support – ability to speed up processing by using the GPU
8. Built-in sensor support – which sensors are available off-the-shelf such as LIDAR, RADAR, GPS, IMU, etc., how difficult to add custom sensors
9. Scenario details – pedestrians, weather conditions, different/custom vehicles, and their associated intelligence capability/interface
10. Plug-in support – ability to automate aspects of the simulation system or tune environmental parameters via plug-in API
11. Operating System support
12. Communications modeling – inter-vehicle communications modeling with the capability to insert delays, simulate packet loss, or model occlusions/dead zones

We examined 13 candidate simulation systems based on these criteria, both open- and closed-source. While none of the candidates could perform all of the required elements, two broad categories met the majority of requirements. Also, they had a significant installed user base developing new capabilities. These two groups were the Unreal Engine-based simulators, including the two derivatives: CARLA (self-driving car-focused), and AirSim (autonomous aerial vehicle focused); and the Unity simulator system. The full comparison and evaluation details can be found in Appendix A.

Ultimately, we selected the CARLA simulator running on Unreal Engine as our baseline simulator for three primary reasons:

1. Large installed user base meant large amounts of support and expertise exist for making modifications and implementing new features
2. Well defined capability to create custom scenarios easily as well as stable API for interacting with simulator system
3. Built-in autonomous vehicle capabilities allow for jumpstarting new project development

Chapter 3

Problem Definition

Consider a 4-wheel drive robot with Ackermann steering. The off-road environment contains only one type of terrain with no obstacles such as rocks, trees, etc. In particular, this terrain is traversable everywhere except for where the elevation is very rough, and the robot cannot traverse over it, this may be either because it does not have enough power or because it gets stuck due to physical limitations.

Setup: Let the state $(x, y, \theta) \in \mathbb{R}^2 \times S^1$ of the robot be defined by the 2D location of the robot on the surface $x, y \in \mathbb{R}^2$ and the heading angle $\theta \in S^1$. Let $(.)_w$ and $(.)_b$ be used to define the coordinates relative to a world frame and the body frame of the vehicle, respectively. Let $\mathcal{G}_{\mathcal{V},\mathcal{E},W}$ represent a graph parameterized by the vertices $\mathcal{V}$, edges $\mathcal{E}$ and a map $W : \mathcal{E} \to \mathbb{R}^+$ which gives the cost of an edge. We generate this graph offline with optimistic edges. During online planning, only the edge-costs are updated to reflect non-traversable edges. The vertices are generated by discretizing the state-space uniformly. The edge connections between edges are generated as follows. We first define a set of local targets M', where each target $m = (dx_b, dy_b, d\theta_w, dt)$ specifies a small-change in state-space. We then check if the vehicle starting from any state say $s = (x_b, y_b, \theta_w)$ can reach a new state determined by this relative change $s' = (x_b + dx_b, y_b + dy_b, \theta_w + d\theta_w)$ within a time-limit dt based on its dynamics constraints and under "optimistic conditions". In our case, this condition refers to a flat world $\mathcal{W}^*$ without any elevation variation, which is traversable everywhere. We store such relative and local targets m that are traversable under optimistic conditions in a set M. This set can be used to generate successors

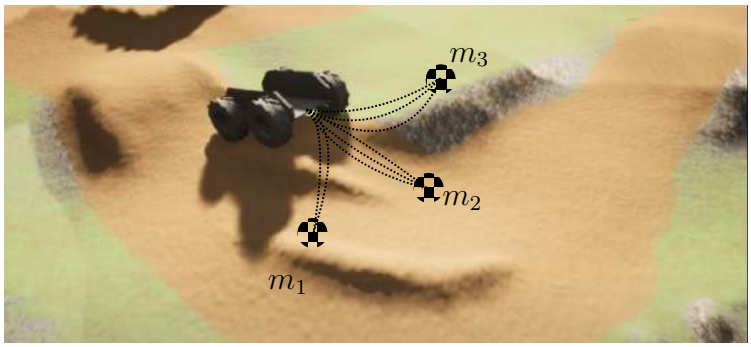

Figure 3.1: Three targets are shown here. The dotted connections indicates that the controller of the vehicle can take any local path to reach the targets. Given the terrain elevation, whether the controller in simulation can reach a target within the time-limit specifies if the edge is valid.

from any state implicitly during planning. Again, note that, in a real-world $\mathcal{W}$, the edge between a state and its successor may not be traversable due to the elevation variations (Figure 3.1).

Let $W_{\text{optimistic}}$ and $W_{\text{sim},\mathcal{W}}$ map the cost of traversing an edge in $\mathcal{W}^*$ and $\mathcal{W}$ respectively, such that $W_{\text{optimistic}}(e) \leq W_{\text{sim},\mathcal{W}}(e)$ holds true for every edge. Let $\mathbb{M}_{\text{sim}}$ and $\mathbb{M}_{\text{ml}}$ denote the simulator and the ML model, respectively. The cost of an edge e in $W_{\text{sim},\mathcal{W}}$ that connects the states say (s, s') determined by the target m can only be revealed by simulating the robot in the world $\mathcal{W}$ at s to see if it can reach s' within dt. This call to the simulator is denoted by QuerySimModel(e) and returns ∞ if s' could not be reached. In contrast, a query to $\mathbb{M}_{\text{ml}}$ is denoted by the function QueryMLModel : $\mathcal{E} \rightarrow \{\text{true}, \text{false}\} \times [0, 1]$ and is expected to return a binary decision on the validity of the queried edge, and a confidence score about its prediction. $\mathbb{M}_{\text{ml}}$ can use local features $f(s, m)$ which depend on s and m to make its prediction. For example, we use an ensemble of convolutional neural network (CNN) classifiers which takes a two-channel image as the input. The first channel contains the elevation map of the region around the robot centered and oriented w.r.t. s_b, and the second channel is a binary image which contains information about m (Please refer to Figure 4.6 in Section 5.2.2). Finally, the problem statement is as follows.

Problem Statement: Given 1) a world $\mathcal{W}$, 2) a graph $\mathcal{G}_{\mathcal{V},\mathcal{E},W}$, 3) a pair of user-specified start and goal states in the graph $(s_{\text{start}}, s_{\text{goal}})$, and 4) two edge-evaluators $\mathbb{M}_{\text{ml}}$, $\mathbb{M}_{\text{sim}}$. We need to find a path $\pi = \{e_1, e_2, \ldots\}$ such that the cost of the path

$c_\pi \leftarrow \sum_{e_i \in \pi} W_{\text{sim},\mathcal{W}}(e_i)$ satisfies 1) $c_\pi \neq \infty$, and 2) $c_\pi \leq \omega_{\text{sub}} c_{\pi^*}$, where c_{π^*} is the cost of the optimal path, and $\omega_{\text{sub}} \in [0, \infty)$ is a user-defined parameter.

Chapter 4

Traversability Models

In this chapter we briefly discuss about the two models the planner can use to evaluate an edge. We first describe our simulator which we refer to as ISimA, and then present details regarding our learned model.

4.1 Physics-Based Simulation

The ISimA framework is built around the Unreal Engine 4 and CARLA functionality to provide the additional flexibility and functionality required to support off-road simulation of multiple vehicles. The primary ISimA components are built on the Unreal Engine foundation with custom CARLA plug-ins to extend the functionality to the CARLA specific capabilities. By using this setup, we can reuse existing capability from both CARLA and Unreal Engine, while allowing for new functionality across both. The overall system is shown in Figure 4.1.

Plugins provide the implementation of objects such as Autonomous Agents, Weather Engine, Communications Emulator, and Sensors. To enable any of these objects in the simulator, they must be spawned into the Virtual World via the Runtime Environment Builder or Autonomous Agent Handler(s).

Autonomous Agent Handler(s) deals with spawning / despawning only autonomous agents (each agent can have its own handler), while Runtime Environment Builder deals with spawning / despawning anything other than autonomous agents, such as landscape, map objects, world sensors, weather engine components, etc.

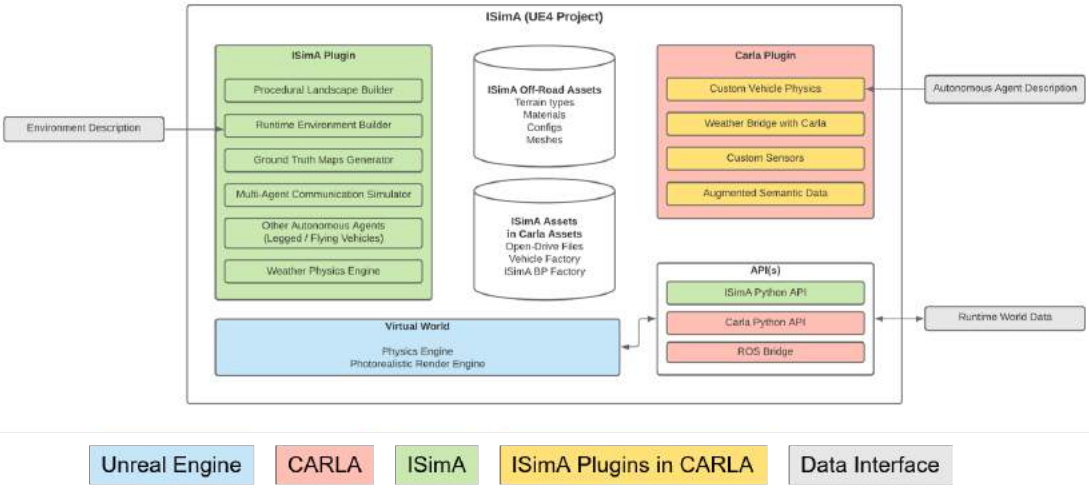

Figure 4.1: ISimA Design Overview

Once all the necessary objects are spawned into the Virtual World, all the runtime data such as controlling the agents, reading scenario performance, world state, ground-truth navigation maps etc., can be accessed from the objects in the world via the APIs as Runtime World Data. Thus, ISimA has only three Data Interfaces: two of them only deal with spawning / despawning (Environment Description and Autonomous Agent Description), and the last one deals with run-time data.

The Unreal Engine based components are:

1. Runtime Environment Builder – creates the simulator environment for a given test run based on the Environment Description provided. This also allows other objects to be spawned in the simulation.
2. Procedural Landscape Builder – constructs the physical landscape in the simulator based on directions from the Runtime Environment Builder
3. Ground Truth Map Generator – creates the maps used by various other components based on the environment and virtual world generated by the Runtime Environment Builder
4. Multi-agent Communication Simulator – provides a simulation pathway for vehicle-to-vehicle and vehicle-to-infrastructure communications
5. Autonomous Agents – extends the existing Unreal Engine framework for wheeled vehicles to allow for additional motion modalities such as tracked, legged, or

aerial vehicles. Additionally, allows for custom physics for wheeled vehicles.

6. Weather Physics Engine – Allows weather effects to modify terrain characteristics and the underlying vehicle physics simulation

Within CARLA there are four planned plug-ins to provide additional capabilities:

1. Custom Vehicle Physics – allows for replacement of the standard NVidia PhysX engine for wheeled vehicles with a custom physics simulation for special case scenarios and to allow for weather-based physics
2. Weather Bridge – ties CARLA weather phenomena to the Weather Physics Engine
3. Custom Sensors – allows for additional sensor modalities as well as weather impacts to sensors
4. Augmented Semantic Data – Provides higher level labeling of elements of the Virtual World to allow for training, testing, and specialized physics

Let us now take a deeper look at some of the most important components we require during planning:

4.1.1 Components based on the Unreal Engine

Runtime Environment Builder

The Runtime Environment Builder is the principal component for establishing and defining the Virtual World within the simulation system and is shown in Figure 4.2. It is responsible for the creation and destruction of elements within the simulator Virtual World (called, respectively, spawning and despawning) such as landscape, map objects, world sensors, weather engine components etc. The one exception is all controlled vehicles1 are spawned / despawned by their respective Autonomous Agent. To accomplish this, the Runtime Environment Builder has access to a library of objects such as 3D meshes and weather engine objects, which can be spawned and initialized in the Virtual World. The Runtime Environment Builder internally calls the Procedural Landscape Builder module to generate 3-D off-road environments without manual 3D modelling. Additionally, it can populate objects for later use by CARLA into the appropriate library. This module can also change the environment during runtime providing for quick scenario changes without requiring time consuming

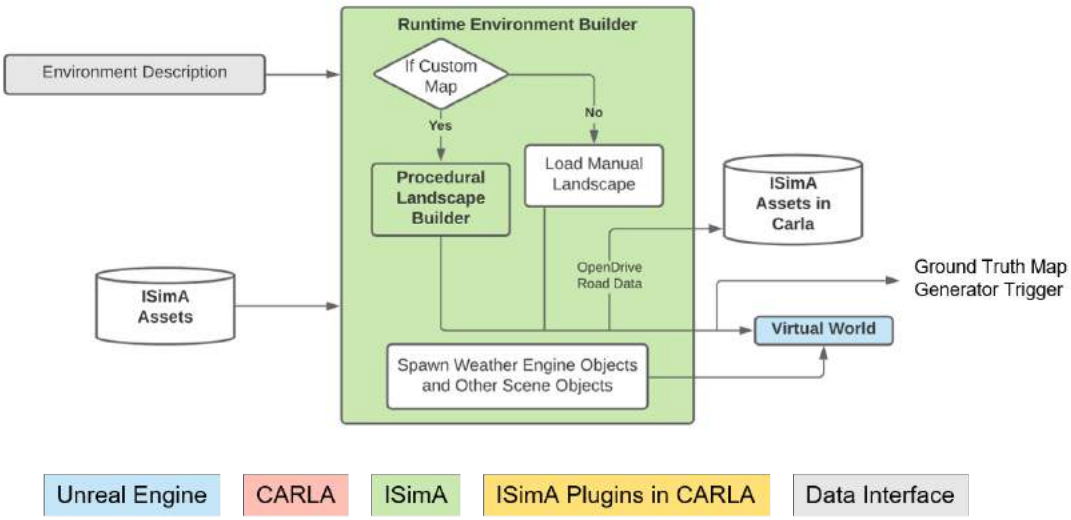

Figure 4.2: Runtime Environment Builder

simulator resets. It can also generate and track numerous sub-worlds inside a single Virtual World.

Procedural Landscape Builder

This module allows for the generation of 3-D terrains without manual construction based on directions from the Runtime Environment Builder. It can programmatically generate height maps, road networks, and terrain types (such as mud, grass, rock, etc.) and use that information for constructing the follow-on OpenDrive formatted map information as well as other required data for the Ground Truth Map Generator. This data is primarily for use by the ISimA system, rather than for the autonomous agents themselves. The Autonomous Agents get their maps from the Ground Truth Map Generator.

Ground Truth Map Generator

Creates the maps used by the Autonomous Agents based on the Virtual World and a trigger from the Runtime Environment Builder. The Ground Truth Maps Generator is depicted in Figure 4.3. Upon trigger, this module will generate all necessary maps such as Elevation Map, Occupancy Grid, Semantic Segmentation Map, etc.

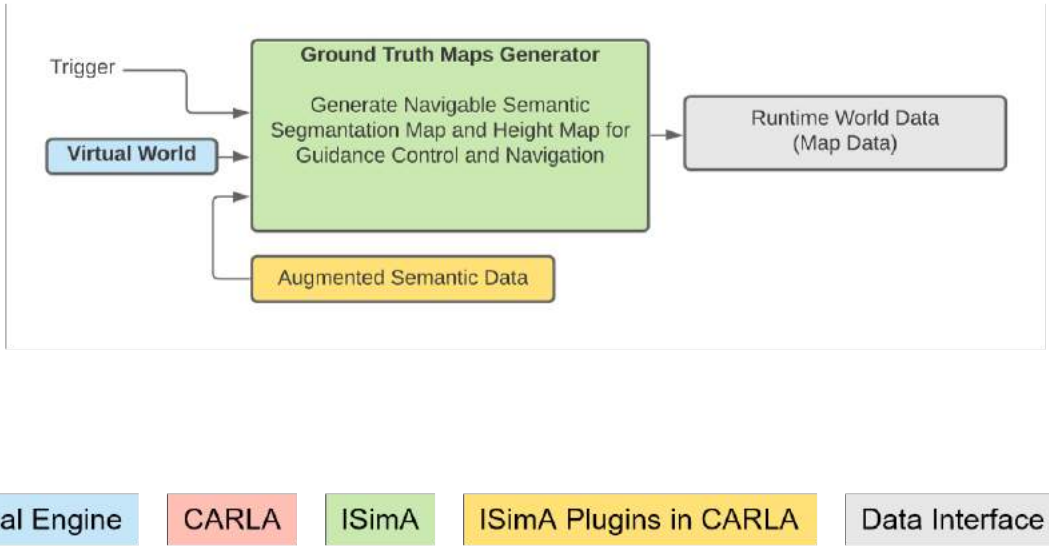

Figure 4.3: Ground Truth Maps Generator

which might be required by the Autonomous Agents for navigation. By separating this functionality from the Procedural Landscape Builder, it becomes possible to introduce mapping errors into the Autonomous Agents' maps while still retaining perfect mapping within the simulator system for other purposes. It is supported by Augmented Semantic Data module which augments the standard CARLA map data to add semantic labels for objects that can be found in off-road environments.

4.1.2 Components based on CARLA

Within CARLA there are four planned plug-ins to provide additional capabilities namely: Custom Vehicle Physics, Weather Bridge, Custom Sensors, Augmented Semantic Data. Out of which we describe only the Custom Vehicle Physics component as the rest of them are not currently used.

Custom Vehicle Physics

This module allows for replacement of the standard NVidia PhysX engine for wheeled vehicles with a custom physics simulation for higher fidelity vehicle simulation and to allow for weather-based physics. This module groups all autonomous agents including wheeled vehicles or Other Autonomous Agents such as legged and flying

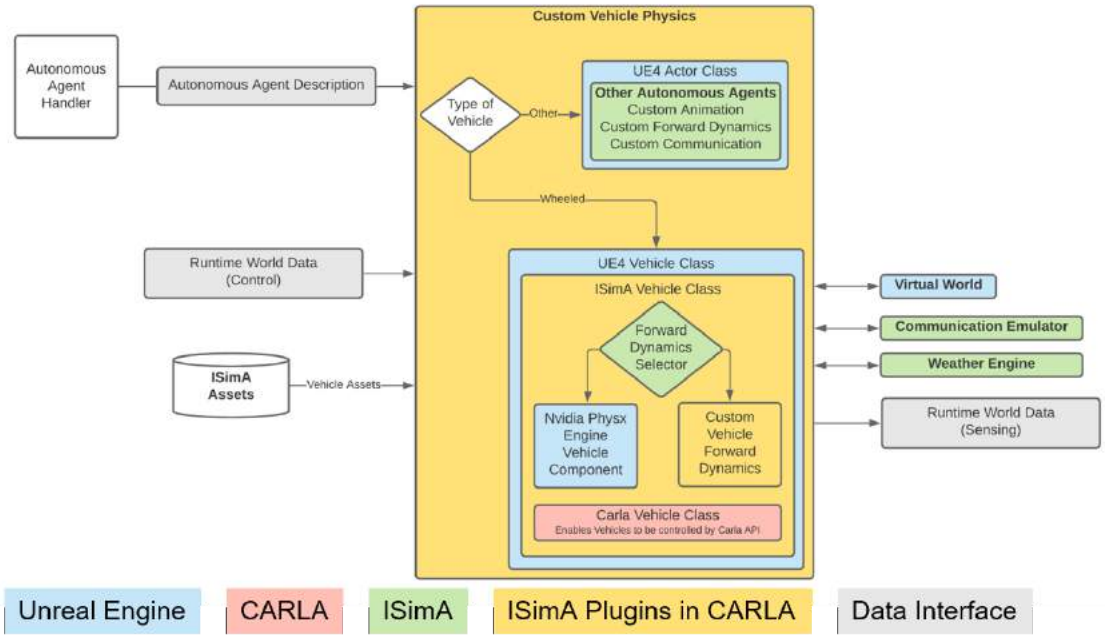

Figure 4.4: Custom Vehicle Physics

robots into a single framework as shown in Figure 4.4. This module however does not include pedestrians, as CARLA natively supports pedestrian motion and controls. CARLA natively provides support for wheeled vehicles (CARLA Vehicle Class), which builds upon the wheeled vehicle model of NVidia PhysX (UE4 Vehicle Class). However, ISimA adds a layer between the two allowing for the integration with user-defined custom vehicle and terramechanics models as well as integration with weather-modified terrain functionality. ISimA does allow a standard wheeled vehicle to use the native CARLA interface for control if desired.

This setup does allow for a user to replace the ISimA custom vehicle physics with a module of their own (or from a third-party) without impacting the remainder of the simulation system.

4.1.3 Setup for the Experiments

We used "**Procedural Landscape Builder**" to replicate the elevation map of the Budds Creek Motocross Track, Maryland, USA, obtained from publicly available laser-scans Figure 4.5. For the purpose of the experiments in Chapter 5, we generate

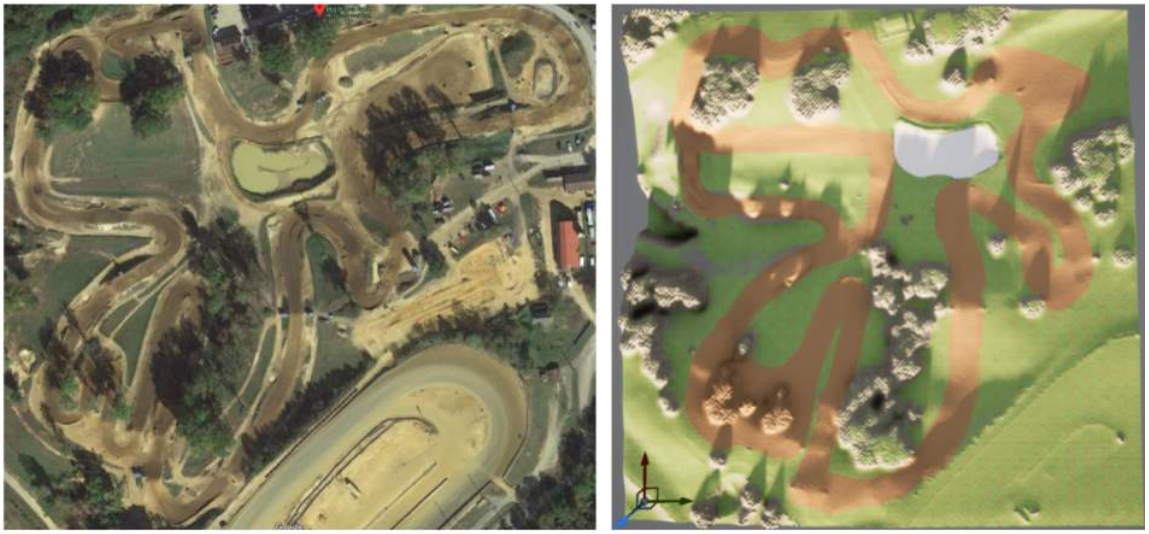

Figure 4.5: Left: Google Earth View of Budds Creek Motorcross Park. Right: Replicated terrain elevation map in UE4.

three maps. These three maps are taken from various parts of this motocross park. While the three maps simply vary in surface area size, the "medium" map is from an area where many paths, many valid paths may exist from any start and goal pair. The vertex count for the graph generated from each map in increasing order are $76,716$, $86,076$ and $411,048$. The maximum branching factor for each vertex was $|M| = 11$.

We simulate a 1.5x scaled-up model of the Clearpath Husky robot using the Nvidia PhysX engine. The low-level controller for the vehicle that accepts a target $m \in M$ (as shown in Figure 3.1) is based on Carla's [9] implementation of lateral and longitudinal PID controllers.

4.2 Machine Learning Model

We use an ensemble of CNN binary-classifiers to provide us with an estimate for the prediction confidence. An example input to the model corresponding to an edge in the graph is shown in Figure 4.6. Note that this model is not trained on any parts of the Budds Creek test environment. It is trained offline.

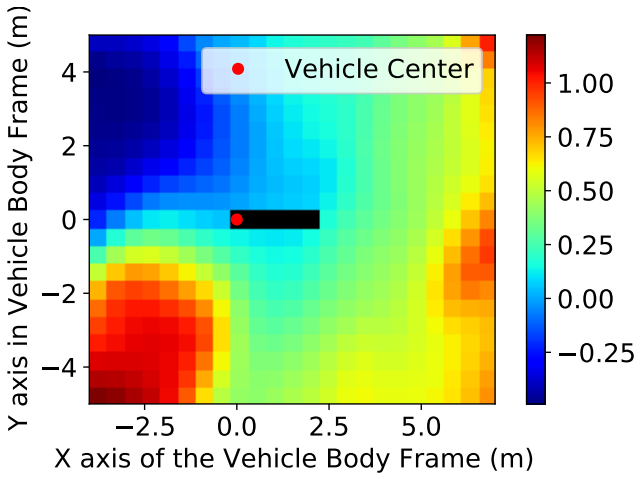

Figure 4.6: An example 2-channel input to $\mathbb{M}_{\text{ml}}$. The first channel encodes the height around the vehicle (in meter). The second channel is a binary image with 'black' pixels encoding the ideal path between the center of the vehicle to a target m. In this case, the target is $2.5m$ along the x-axis (vehicle forward). In this visualization, only the black pixels from the second channel are overlaid on top of the first channel.

4.2.1 Offline Training

We train the neural network with data obtained after rolling out simulations on a series of maps with varying elevations. These elevation maps are not from the real world but generated based on the Perlin noise model [23] with a manually selected range of parameters that generate reasonable variation in elevation roughness and amplitude (see Figure 4.7). We believe that this is a valid process to prepare the model for a real-world map, and pre-training with this data does not require any real-world elevation map. Once the model is trained, it is not changed in the course of the experiments described in the following sections. We employ a trick to make the predictions more robust at no additional ML inference time.

4.2.2 Online Inference

Instead of querying the model for a target $m \in M$ and for a vehicle location given by the vertex, we additionally query the ML model parallelly in the GPU for a set of neighboring vehicle locations for the same target m and then take a vote inversely

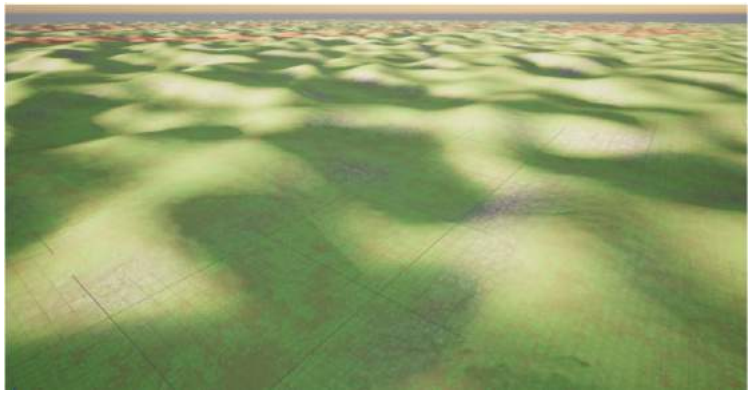

Figure 4.7: A terrain procedurally generated using a perlin-noise heightmap.

weighted by the deviation. Finally, with TensorRT optimization [25], the inference time on an Nvidia Titan V GPU is around $3ms$. Without TensorRT optimization, we observed that the inference time for the same set of input is around $250ms$. Thus, it is very important to optimize the model for inference to gain the advantage of parallel search with the learned model. We use the "**Ground Truth Map Generator**" component to generate the elevation map around the robot as shown in Figure 4.6.

Chapter 5

Planning with a Learned Model and a Simulator

5.1 Approach

The key idea in MA3 is to delay edge-evaluations by $\mathbb{M}_{\text{sim}}$ and carry out the regular LazySP-like search using $\mathbb{M}_{\text{ml}}$. Ideally, the simulator is queried only to verify a path found valid by $\mathbb{M}_{\text{ml}}$. However, some additional simulation queries are made to evaluate 1) edges where the model $\mathbb{M}_{\text{ml}}$ is not confident about its prediction, and 2) edges that have been marked as invalid by $\mathbb{M}_{\text{ml}}$ with high confidence. Figure 5.1 shows an example run-through of MA3.

Notations: First we introduce some additional notations used in Algorithm 1. Relevant line numbers are shown as $\{\dots\}$. Multiple priority values $(p_1, p_2, \dots)$ ordered in descending order of their significance can be used to determine the priority of an element in a priority queue. Lesser significant priority is only used to break ties. Finally, the procedure A*$(s_{\text{start}}, s_{\text{goal}}, W_{\text{current}})$ returns 1) a least-cost path π with regards to the edge costs W_{current}, and 2) the cost of the path c_π. While $W_{\text{current}}[e]$ gives the cost of an edge e, $W_{\text{current}}[e]$.model contains the information about the source of the edge cost, which can take either of the following values {init, temp, ml, sim, ppe}.

- init: cost are based on $W_{\text{optimistic}}$ {7}

- temp: if cost is set to ∞ temporarily {35}
- ml: cost updated based on $\mathbb{M}_{\text{ml}}$ evaluation
- sim: cost updated based on $\mathbb{M}_{\text{sim}}$ evaluation
- ppe: potentially pessimistic edge (explained in 5.1.2)

5.1.1 Lazier Simulation Evaluations

First, let us relate the parts of our algorithm similar to LazySP. Like LazySP, A* always uses a edge-cost-map W_{current} (which is initialized with $W_{\text{optimistic}}$) to compute the least-cost path {9 and 41}. During the search, this data structure is updated by calling the OVERWRITE function to contain the most-relevant cost for all the edges. Specifically, we store two information about an edge, 1) its cost in $W_{\text{current}}[e]$, and 2) the latest model that contributed the edge-cost in $W_{\text{current}}[e]$.model. OVERWRITE also sets the boolean variable is_new_W to "true" when the edge-cost being overwritten is different from its previous value. Doing this allows the search to run A* only when the graph has changed from the previous iteration. {15-36} does very similar work like the main-loop of LazySP, i.e., 1) it considers one path candidate after the other (π) sorted in increasing order of their potential path cost (c_π) {16}, 2) selects edges E from that path based on the LAZYEDGESELECTOR for evaluation, and 3) repeats this process until a path is found where no new edges require evaluation. Let us denote such a path as "evaluated path". Note that you may use any selector as provided in [20].

Here is how MA3 is different. Firstly, it runs two threads, one that performs the graph-search {1-48}, and the other {49-60} that uses $\mathbb{M}_{\text{sim}}$ to evaluate sets of edges that are scheduled in a priority queue $\mathcal{P}_{\text{sim}}$. Secondly, for every edge that has been selected by the LAZYEDGESELECTOR, instead of using only one edge-evaluator, MA3 chooses between $\mathbb{M}_{\text{ml}}$ and $\mathbb{M}_{\text{sim}}$. The more expensive $\mathbb{M}_{\text{sim}}$ is used if the confidence of the approximate $\mathbb{M}_{\text{ml}}$ is below a user-defined threshold ϵ_{conf} {21-36}. SCHEDULEFORSIM is a non-blocking function; thus, MA3 would not wait for the evaluation of an edge in $\mathbb{M}_{\text{sim}}$ to be available. Instead, it will temporarily disable that edge by setting its cost to ∞ {35} and continue with the search to find another path candidate. The idea is that, while the simulator is busy evaluating edges from potentially optimal paths, it might be worth exploring the graph and

finding a potentially sub-optimal "evaluated path" quickly. Additionally, MA3 does extra work to maintain bounded-suboptimality (marked in red) and completeness (marked in blue) properties.

Once an "evaluated path" is found, LazySP will return it as a solution. However, in MA3, this path may consist of edges that were evaluated by only the $\mathbb{M}_{\text{ml}}$ and thus, may be invalid. Therefore, MA3 will verify those edges using $\mathbb{M}_{\text{sim}}$ before returning it as a solution. Again, instead of waiting for verification, these edges will only be scheduled. However, the edges being scheduled for path verification are set to a higher priority (see ScheduleForSim) {61-67}.

GetSimData processes the evaluations completed by the simulator. Once an "evaluated-path" is verified to be valid in simulation, the upper bound on path-cost ub is updated {78-80} and the path is added to $\mathcal{P}_{\text{valid}}$. It is still not ready to be returned as a solution yet. Within the search, when CheckGap {89-97} is called, MA3 can now compare the upper bound ub with the lower-bound lb on path cost to see if the sub-optimality condition is satisfied. This ensures the bounded-suboptimality property of our algorithm, and having found such a path, the anytime solution stored in $\mathcal{P}_{\text{valid}}$ can finally be returned {45-48}.

Now we explain how we maintain the lower-bound lb on path-cost. A* generates least-cost path candidates based on W_{current} which initially contains optimistic costs; thus, the path candidates have the potential to set the lower bound. We add these paths to LB {10 and 43}. A path in LB will lose the potential to set the lower bound if it contains at least one invalid edge. Thus, when discovered, such paths are removed from LB {76}. Alternatively, if a path candidate consists of potentially pessimistic edges (explained in 5.1.2), it can incorrectly raise the lower bound. Thus, in {30} we disallow these candidates to increase the lower bound.

5.1.2 Handle ML Model Inaccuracies

The error-prone model $\mathbb{M}_{\text{ml}}$ can make two types of errors: 1) false-positive errors, which leads to optimistic edge costs, and 2) false-negative errors, which leads to pessimistic edge costs. Optimistic edge costs do not violate completeness guarantees. We handle them lazily, i.e., we let those edges be considered during the search until they are a part of an "evaluated path". In this case, MA3 will reveal their actual

Algorithm 1 Pseudocode for MA3

procedure MA3($s_{\text{start}}, s_{\text{goal}}, \omega_{\text{sub}}, \epsilon_{\text{conf}}$)
  $\mathcal{P}_{\text{valid}} \leftarrow \emptyset$ ▷ List of Sim Validated Paths
  $\mathcal{P} \leftarrow \emptyset$ ▷ Path candidates ordered by path cost
  $\mathcal{P}_{\text{sim}} \leftarrow \emptyset$, $\mathcal{P}_{\text{data}} \leftarrow \emptyset$ ▷ Edge-sets waiting for Sim
  $E_{\text{conf}} \leftarrow \emptyset$ ▷ Edges with high confidence
  $O_{\text{e}} \leftarrow \emptyset$ ▷ Potentially pessimistic edges
  W_{current} ▷ Latest edge costs. Initialized with $W_{\text{optimistic}}$
  is_new_$W \leftarrow$ false
  $(c_\pi, \pi) \leftarrow$ A*$(s_{\text{start}}, s_{\text{goal}}, W_{\text{current}})$
  $LB \leftarrow \{(\pi, c_\pi)\}$ ▷ Paths that determine lower-bound
  $ub \leftarrow \infty$ ▷ upper bound on path cost
  Insert π in $\mathcal{P}$ with priority c_π
  Start SimulationWorker Thread
  while ($|\mathcal{P}| > 0$ or $|\mathcal{P}_{\text{sim}}| > 0$) **do**
    if $|\mathcal{P}| > 0$ **then**
      $(c_\pi, \pi) \leftarrow$ POP($\mathcal{P}$)
      $E \leftarrow$LazyEdgeSelector(π)
      if E is $\emptyset$ **then**
        ScheduleForSim(π, c_π)
      else
        for all $e \in E$ **do**
          $c_e \leftarrow W_{\text{current}}[e]$
          (is_valid, conf) $\leftarrow$ QueryMLModel(e)
          if is_valid = false **then**
            $c_e \leftarrow \infty$
          if conf $> \epsilon_{\text{conf}}$ **then**
            $W_{\text{current}} \leftarrow$ Overwrite(e, c_e, 'ml')
            Insert e in E_{conf}
            if $c_e = \infty$ **then**
              Update the path-cost of π in LB with $\min_{(\pi, c_\pi) \in LB} c_\pi$
              Insert (e, c_π) in O_{e}
              Break **for** loop
          else
            ScheduleForSim($\{e\}, c_\pi$)
            $W_{\text{current}} \leftarrow$ Overwrite(e, ∞, 'temp')
            Break **for** loop
    else
      Wait until $|\mathcal{P}_{\text{data}}| > 0$
    GetSimData()
    if is_new_W **then**
      $(c_\pi, \pi) \leftarrow$ A*$(s_{\text{start}}, s_{\text{goal}}, W_{\text{current}})$
      Insert π in $\mathcal{P}$ with priority c_π
      Insert (π, c_π) into LB
      is_new_$W \leftarrow$ false
    if CheckGap() **then**
      Break **while** loop ▷ Anytime solution found
  Terminate SimulationWorker Thread
return Least cost path from $\mathcal{P}_{\text{valid}}$

procedure SimulationWorker
  while Thread not terminated **do**
    (priorities $(p_1, p_2), E) \leftarrow$ POP($\mathcal{P}_{\text{sim}}$)
    continue if $p_1 = 1$ and $p_2 > ub$
    $C \leftarrow \emptyset$
    for all $e \in E$ **do**
      if $W_{\text{current}}[e]$.model = 'sim' **then**
        $c_e \leftarrow W_{\text{current}}[e]$
      else
        $c_e \leftarrow$ QuerySimModel(e)
      Insert c_e in C while preserving order
    Insert (p_1, p_2, E, C) in $\mathcal{P}_{\text{data}}$

Algorithm 1 Pseudocode for MA3 (continued)

```
61: function ScheduleForSim(E, c)
62:     if the set of edges E represent a path then
63:         π ← E                                          ▷ A path is a set of edges
64:         Insert π in P_sim with priority (1, c)
65:     else
66:         {e} ← E
67:         Insert {e} in P_sim with priority (2, c)

68: function GetSimData
69:     P_data ← Get the edge-sets that have been evaluated
70:     for all (p1, p2, E, C) ∈ P_data do
71:         path_verified ← true
72:         for all e ∈ E and the corresponding c_e ∈ C do
73:             W_current ← Overwrite(e, c_e, 'sim')
74:             Insert e in E_conf
75:             if c_e = ∞ then
76:                 Remove from LB path π that generates e
77:                 path_verified ← false
78:         if p1 = 1 and path_verified = true then
79:             Insert (E, Σ_{c_e∈C} c_e) in P_valid
80:             ub ← min(ub, Σ_{c_e∈C} c_e)
81:     P_data ← ∅

82: function LazyEdgeSelector(π)
83: return At least one edge from π that is not in E_conf or ∅ if all edges of π are in E_conf .

84: function Overwrite(e, c_e, model)
85:     if W_current[e] ≠ c_e then
86:         W_current[e] ← c_e
87:         is_new_W ← true
88:     W_current[e].model ← model

89: function CheckGap
90:     lb ← min_{(π,c_π)∈LB} c_π
91:     if (ub/lb) > ω_sub then
92:         if no paths left to consider in P and P_sim then
93:             (e, c_π) ← argmin_{O_e} c_e
                          s.t. W_current[e].model = 'ml'
94:             W_current[e].model ← 'ppe'
95:             ScheduleForSim({e}, c_est)
96:         return false
97:     else return true
```

edge cost when verifying the "evaluated path".

In contrast, the pessimistic edges may lead the search not to discover a valid path. An edge that is invalidated by $\mathbb{M}_{\text{ml}}$ with high confidence is potentially pessimistic. We keep track of these edges in O_e {31}. Now, it is a matter of selecting edges from this list and scheduling them for simulation to reveal their actual cost. We do this in CheckGap {92-95}. We choose to schedule these edges for simulation after the search has exhausted all possible path candidates to find a valid path that satisfies the suboptimality gap. And we select them in increasing order of the path-cost corresponding to the path that contains the edge. Once scheduled, we mark the

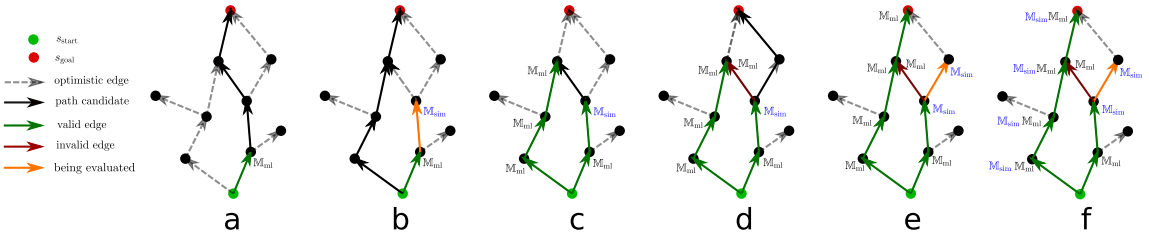

Figure 5.1: Assuming that $\mathbb{M}_{\text{ml}}$ makes no mistakes, a typical run of MA3 would like like this. In (a) a shortest-path-candidate is found and its first edge is selected for evaluation. Since $\mathbb{M}_{\text{ml}}$ was confident about this edge, MA3 accepts its evaluation. In (b) the next edge is selected for evaluation, however, the $\mathbb{M}_{\text{ml}}$ is not confident and hence $\mathbb{M}_{\text{sim}}$ is used. Let us assume that querying $\mathbb{M}_{\text{sim}}$ takes 3x that of $\mathbb{M}_{\text{ml}}$. Thus, while $\mathbb{M}_{\text{sim}}$ is evaluating that edge MA3 will attempt to continue the search through the next shortest-path-candidate (marked in black). In (c) we see that $\mathbb{M}_{\text{ml}}$ is confident and hence has evaluated three edges until $\mathbb{M}_{\text{sim}}$ evaluates its edge. Since $\mathbb{M}_{\text{sim}}$ found this edge to be valid, the search can now resume along the previous shortest-path-candidate. In (d) $\mathbb{M}_{\text{ml}}$ invalidated an edge and similarly the next shortest-path-candidate is considered. Again in (e) as we wait for $\mathbb{M}_{\text{sim}}$ evaluation, MA3 considers the next shortest-path-candidate $\mathbb{M}_{\text{ml}}$ and employs $\mathbb{M}_{\text{ml}}$ to evaluate the last edge that connects the goal. At this point, we have an "evaluated-path" and thus in (f) the simulator is made to verify this path with high priority. Since a valid path is found MA3 will return this solution if it meets the suboptimality criteria.

model of these edges to be 'ppe' such that MA3 does not add them again to O_e.

5.1.3 Theoretical Analysis

Claim: If there exists an optimal path π^* that is valid according to $\mathbb{M}_{\text{sim}}$, then given an admissible heuristic function, Algorithm 1 is guaranteed to return a path $\hat{\pi}$ such the its cost $c_{\hat{\pi}}$ is no more than ω_{sub} times the cost c_{π^*} of π^*, and such that $\hat{\pi}$ is valid according to $\mathbb{M}_{\text{sim}}$.

Lemma 5.1.1. *All paths returned by Algorithm 1 are valid according to the given* $\mathbb{M}_{sim}$.

Proof. This theorem is enforced by construction as GetSimData function only inserts paths π marked for verification (distinguished by priority $p_1 = 1$) when all the edges in the path have non-infinite costs. Note that some of these edge costs (for edges with model 'sim') are being taken from storage {55,56} to prevent duplicate

evaluations. Because MA3 is a multi-threaded program, we ensure that the model is updated strictly after the correct edge costs are overwritten in W_{current} {84-88}. This also ensures that MA3 corrects false-positive errors made by $\mathbb{M}_{\text{ml}}$. □

Theorem 5.1.2 (Completeness)**.** *Algorithm 1 is complete if the heuristic function is admissible.*

Proof. Given an admissible heuristic function, A* and LazySP are proven to be complete. Hence, given an admissible heuristic function, MA3 will generate path candidates that are optimal given the current edge costs W_{current}. Only false-negative errors in W_{current} made by $\mathbb{M}_{\text{ml}}$ will prevent MA3 to find a feasible path. Suppose that there are no path-candidates left {92}, then by construction, in {93-95} MA3 ensure that all the potentially pessimistic edges get an opportunity to get scheduled for $\mathbb{M}_{\text{sim}}$. Note that the condition specified in {93} skips some potentially pessimistic edges from O_e. However, these edges are those which 1) have already been evaluated by $\mathbb{M}_{\text{sim}}$ or 2) have appeared as potentially pessimistic edge before and thus are already scheduled for $\mathbb{M}_{\text{sim}}$. Figure 5.2 shows all possible ways in which the model of an edge evolves. An edge with source 'ml' will only upgrade to 'sim' either directly or via 'ppe' {94}. Thus, the condition $W_{\text{current}}[e]$.model = 'ml' in {93} will never miss to schedule a potentially pessimistic edge. If all potentially pessimistic edges are eventually evaluated in $\mathbb{M}_{\text{sim}}$, all the false-negative errors in W_{current} will eventually be corrected to find the optimal path. Hence MA3 is complete. □

Theorem 5.1.3 (Bounded Suboptimality)**.** *The path $\hat{\pi}$ returned by Algorithm 1 is guaranteed to satisfy $c_{\hat{\pi}} \leq \omega_{sub} c_{\pi^*}$.*

Proof. The best valid path found by $\mathbb{M}_{\text{sim}}$ is used to update the upper bound *ub* {78-80} on the path-cost as once a path is found, we only want to find a better path. Thus, the upper bound informs about the best valid path found so far. Next, we discuss the lower bound on path cost. The first path π_1 found by A* when $W_{\text{current}} = W_{\text{optimistic}}$ contains only optimistic edge costs. The cost of a valid path can only be greater than or equal to c_{π_1}. Hence, if $lb \leftarrow c_{\pi_1}$, then any path π that satisfies $c_\pi / lb \leq \omega_{\text{sub}}$ is bounded-suboptimal. However, if we discover that an edge in π_1 is invalid, then the lower bound as described above will only be a loose bound. Thus, we raise the lower bound by considering the next optimal least-cost path. Therefore, it

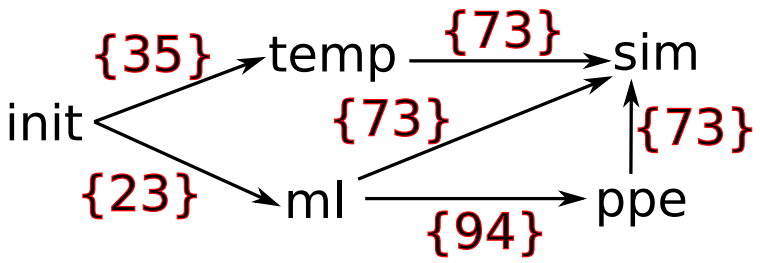

Figure 5.2: Flow Diagram For Model Sources

now remains to prove that the lower bound is not raised incorrectly. As described in Section 5.1.2, if $\mathbb{M}_{\text{ml}}$ only makes false-positive (optimistic) errors in W_{current}, the path costs returned by A* will also be optimistic. However, if a path candidate returned by A* contains an edge that is potentially pessimistic, the path cost may be higher than the actual path cost, in which case, this path cost should not be used to determine the lower bound. Hence, by ignoring such paths {30}, MA3 ensures that the lower bound is never raised incorrectly. □

5.2 Experiments

We are interested in three primary metrics, namely:

1. Planning times: MA3 is primarily designed to reduce calls to the time expensive edge-evaluator $\mathbb{M}_{\text{sim}}$ as in general the time spent in search and querying the approximate evaluator $\mathbb{M}_{\text{ml}}$ is negligible when compared to the time required to query the $\mathbb{M}_{\text{sim}}$. Therefore planning time is proportional to the number of calls.
2. Rate of success in finding a valid solution: False-positive errors made by $\mathbb{M}_{\text{ml}}$ can invalidate edges that are required for the solution. So, we are interested in knowing how many times does the planner fails to find a solution.
3. Path cost: This metric allows us to empirically compare the solution quality of the planner to the optimal solution.

5.2.1 Baselines

All our baselines are based on LazySP [11] as it is proven to be edge-optimal with one edge-evaluator, which means that there is no algorithm that can evaluate a lesser

number of edges compared to LazySP for the same planning problem with the same information. So, we only generate baselines that gradually employ smarter techniques to employ two edge-evaluators with the properties as discussed above. We run MA3 with parameters $\omega_{\text{sub}} = 2$, $\epsilon_{\text{conf}} = 0.6$.

1. LSP w SIM Evaluation: This is the primary baseline which is simply LazySP with the accurate edge evaluator $\mathbb{M}_{\text{sim}}$. Since $\mathbb{M}_{\text{sim}}$ is accurate, this baseline is also guaranteed to be complete. Also, note that LazySP is not an anytime, so this will always return the optimal solution.
2. LSP w ML Evaluation: One might choose to not use $\mathbb{M}_{\text{sim}}$ with LazySP but rather use an approximate edge-evaluator $\mathbb{M}_{\text{ml}}$. While the edge-evaluation time for these baselines will be very low, it does not guarantee completeness. We verify the path returned by this baseline in $\mathbb{M}_{\text{sim}}$ to compute the success rate.
3. LSP w ML + SIM Verification: This baseline is very similar to Baseline 2 where $\mathbb{M}_{\text{ml}}$ is only used during the search, but before a path is returned, it is verified in $\mathbb{M}_{\text{sim}}$. If the path is found to be invalid, the search will resume until a valid path is verified. Note that it also does not guarantee completeness as this baseline will fail to correct false-negative errors. It will return the first valid path without any guarantees to return a bounded-suboptimal path. This is analogous to running MA3 with $\epsilon_{\text{conf}} = 0$, $\omega_{\text{sub}} = \infty$ and without inserting any edge to O_e in {31}.
4. LSP w ML + SIM Evaluation & Verification: This baseline builds on Baseline 3, in that it makes a choice between $\mathbb{M}_{\text{sim}}$ and $\mathbb{M}_{\text{ml}}$ depending on whether the model $\mathbb{M}_{\text{ml}}$ is confident about its prediction (similar to {26} in MA3). This is analogous to running MA3 with $\epsilon_{\text{conf}} =$ same as MA3, $\omega_{\text{sub}} = \infty$ and without inserting to O_e.
5. Single Thread MA3 (LSP w ML + SIM Evaluation, Verification & Correction): Baseline 2,3, and 4 lack completeness guarantees and hence would not have a way to guarantee bounds on the solution quality. Thus, they return the first solution they find. Baseline 3 and 4 also use two threads like MA3. Moreover, MA3 also returns the first solution that meets the bounded suboptimality criteria and utilizes two parallel threads. In contrast, Baseline 1 is designed to run on a single thread and return the optimal solution. Therefore, we add this

baseline to compare how would MA3 perform if it were to not use the benefits of two parallel threads and return the optimal solution. This is analogous to running MA3 by replacing line {38} with "Wait until $|\mathcal{P}_{\text{sim}}| = 0$" and setting $\omega_{\text{sub}} = 1$.

5.2.2 Off-Road Navigation Domain

We use three maps with varying sizes, which directly dictates the size of the graph.

Table 5.1: Speed-up in planning times w.r.t. LSP w SIM (Factor x)

Maps		LSP w SIM	LSP w ML + SIM V	LSP w ML + SIM E&V	Single Thread MA3 (Optimal)	MA3
Small	Avg : 95% CI	1	0.99 : [0.87, 1.11]	1.07 : [0.86, 1.27]	0.77 : [0.63, 0.92]	1.54 : [0.73, 2.35]
	Min, Max	1, 1	0.62, 1.61	0.62, 2.24	0.13, 1.31	0.62, 8.24
Medium	Avg : 95% CI	1	2.93 : [1.40, 4.46]	2.57 : [1.31, 3.84]	1.38 : [1.10, 1.66]	2.54 : [1.28, 3.80]
	Min, Max	1, 1	0.96, 11.04	1.00, 9.34	0.93, 2.45	0.97, 8.53
Large	Avg : 95% CI	1	5.03 : [3.36, 6.70]	3.71 : [2.69, 4.72]	1.04 : [0.95, 1.13]	3.46 : [2.56, 4.36]
	Min, Max	1, 1	0.66, 33.61	0.75, 15.03	0.38, 1.77	0.46, 12.93

Table 5.2: Failure Rate of the Planners (%)

Maps	LSP w SIM	LSP w ML	LSP w ML + SIM V	LSP w ML + SIM E&V	Single Thread MA3 (Optimal)	MA3
Small	0	26.31	10.52	5.26	0	0
Medium	0	31.25	0	0	0	0
Large	0	57.64	2.35	1.17	0	0

Table 5.3: Suboptimality in path cost w.r.t LSP w SIM

Maps		LSP w SIM	LSP w ML	LSP w ML + SIM V	LSP w ML + SIM E&V	MA3
Small	Avg : 95% CI	1	1.01 : [1.00, 1.03]	1.01 : [1.00, 1.03]	1.00 : [1.00, 1.02]	1.00 : [1.00, 1.02]
	Min, Max	1, 1	1.00, 1.08	1.00, 1.07	1.00, 1.08	1.00, 1.08
Medium	Avg : 95% CI	1	1.01 : [1.00, 1.03]	1.01 : [1.00, 1.03]	1.00 : [1.00, 1.02]	1.00 : [1.00, 1.02]
	Min, Max	1, 1	1.00, 1.08	1.00, 1.07	1.00, 1.08	1.00, 1.08
Large	Avg : 95% CI	1	1.00 : [1.00, 1.00]	1.00 : [1.00, 1.00]	1.00 : [1.00, 1.00]	1.00 : [1.00, 1.00]
	Min, Max	1, 1	1.00, 1.01	1.00, 1.02	1.00, 1.01	1.00, 1.03

Table 5.1, Table 5.2 and Table 5.3 present the performance comparison for MA3 and the baselines for various metrics over 100 planning episodes each for the three maps. For the planning episodes, a random goal state was selected once, and then 100 random start locations were selected. Note that the planners are given the same start-goal pair per episode. We then ran a backwards Dijkstra's search [8] to solve the single-source shortest-path problem with edge-costs from $W_{\text{optimistic}}$. This allowed us to compute the cost-to-go values once, which can then be used as a consistent and admissible heuristic in all the 100 planning episodes. MA3 and all the baselines are implemented in C++. The average query time for $\mathbb{M}_{\text{sim}}$ is around $450ms$ while that of $\mathbb{M}_{\text{ml}}$ is around $3ms$. We could not manage to make the simulation faster without decreasing the fidelity. We chose Nvidia PhysX Engine as it utilizes GPU parallelization, and we disabled UE4 rendering to speed up physics simulation.

5.2.3 Speed-up in planning times

For each planning episode, we compute the pairwise speed-up achieved in planning time for a planner as compared to that of LazySP w SIM. Table 5.1 shows the various statistics over 100 planning episodes. We present 1) average with 95% confidence interval (within []), and 2) the minimum and maximum values observed. We did not present the speed-up for **LSP w ML** due to space-constraints as it is trivially very fast since it does not query $\mathbb{M}_{\text{sim}}$ and is around 10x. We observe that the speed-up achieved correlates with the size of the graph as expected. **MA3** achieved an average speed-up of 3.46x in the "Large" map. Although **LSP w ML + SIM V** and **LSP w ML + SIM E&V** baselines achieves a higher speed-up, as we will observe in Section 5.2.4, these baselines did not solve all the planning problems. In a particular case, **MA3** was around two times slower (speed-up of 0.46) which is because $\mathbb{M}_{\text{ml}}$ made significant errors. False-positive errors waste simulation efforts for path verification and false-negative errors lead the search away from valid solutions requiring the evaluation of potentially-pessimistic edges. Section 5.2.5 presents how the errors in $\mathbb{M}_{\text{ml}}$ affect performance of **MA3**.

5.2.4 Success Rate and Solution Cost

It is important to note that the metrics in Table 5.1 and Table 5.3 are computed only if the planner found a path which is valid in $\mathbb{M}_{\text{sim}}$. Table 5.2 presents the success rate for each planner. MA3 achieves a 100% success rate thanks to its completeness properties. The "medium" map provides a lot of alternate routes, which is why Baseline 3 and 4 did not fail in the random planning episodes we tested them on.

Table 5.1 presents the speed-up for the planners which might return sub-optimal solution. Recall that Baselines 2,3 and 4 return the first solution, Baseline 5 returns optimal solution and MA3 returns bounded-suboptimal solution $\omega_{\text{sub}} = 2$. Therefore, compared to **LazySP w SIM**, and except for Baseline 5, the planners trade optimality for planning times. Thus, we present the ratio of the path cost found by the planner as compared to that of the optimal path in Table 5.3. We observe that a significant speed-up can be achieved for certain planning problems without sacrificing too much on the solution quality. We use the bootstrap method [7] to compute the confidence intervals as suboptimality cannot be lower than 1.

5.2.5 Ablation Studies

The ML model is error-prone, while the simulator is relatively slower to query. Therefore, in these experiments, we wanted to measure how the performance of the planners is affected by the characteristics of the edge-evaluators.

Performance vs ML model accuracy

In order to control the model accuracy, we employed the following trick. We used the same ML model trained as described in Section 5.2.2 which provides a binary prediction and its confidence score, but instead of using the prediction, we flip the ground-truth prediction with a probability to emulate a learned model with a certain rate of accuracy in the test environment. Figure 5.3 shows the speed-up achieved by **MA3** in the "Medium" map with various ω_{conf}. Setting $\omega_{\text{conf}} = 0$ makes MA3 rely more on $\mathbb{M}_{\text{ml}}$ for initial edge evaluation, thus we observe that MA3 has higher speed-ups when the $\mathbb{M}_{\text{ml}}$ is more accurate. However, as the accuracy reduces, the speed-up also reduces. Interestingly the speed-up with an ML model with 50%

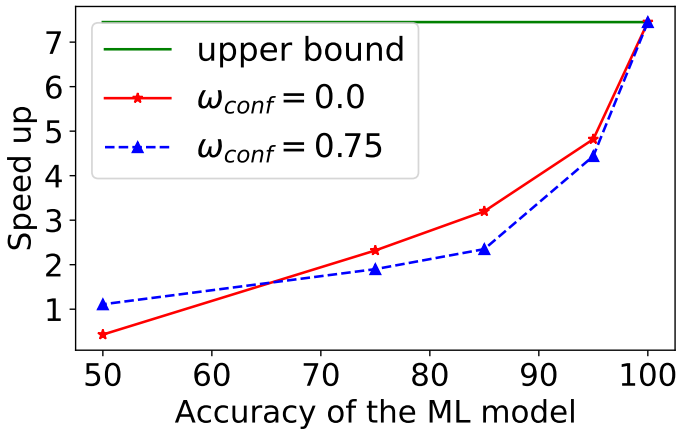

Figure 5.3: Speed-up with varying model accuracy and confidence thresholds

accuracy is lower when $\omega_{\text{conf}} = 0$ as compared to when $\omega_{\text{conf}} = 0.75$. From a user's perspective, ω_{conf} should be set based on the accuracy of $\mathbb{M}_{\text{ml}}$.

Performance vs Simulation Query Time

As expected we observe in Figure 5.4 that MA3 shows significant speed-up when $\mathbb{M}_{\text{sim}}$ is relatively slower than $\mathbb{M}_{\text{ml}}$. For this experiment, we cached the results from the simulator so that they can be made available with any custom delay to emulate any simulation query time. The intuition behind this is that if SimulationWorker has high throughput, it may perform additional simulations before the main thread reaches GetSimData and realizes that all the evaluations required to return a valid path are available.

5.3 Discussion

The experimental results show that the proposed algorithm is opportunistic and can perform really well when provided with an accurate learned model. However, since there is no free lunch, we also observe that the performance can be worse if the model is terribly inaccurate. We also observe that for the baseline **LazySP w ML** is extremely fast although it lacks completeness guarantees. MA3 essentially spends

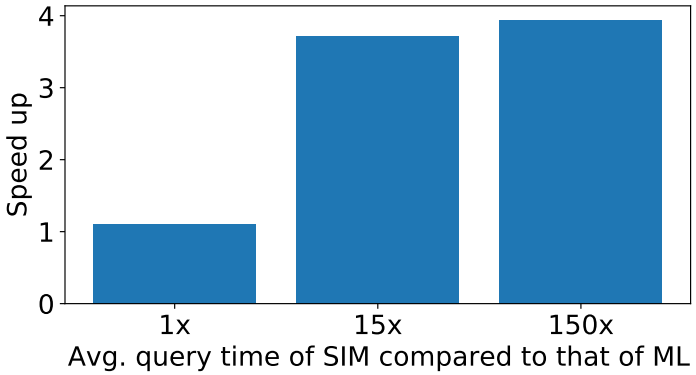

Figure 5.4: Speed-up with simulation query time

a lot of efforts in maintaining completeness guarantees. It does it by taking care of the edge cost of all the edges that take part in the search, and ensuring that only edges evaluated by the simulator finally appear in the solution.

Chapter 6

Conclusions

In this work, we looked at the off-road navigation-planning problem. We attempt to determine traversability using a simulator, assuming that it is accurate, and use a learned model trained with simulation data to approximate traversability quickly during online planning. To that end, we provide a novel planning algorithm, MA3, which can exploit the complementary properties of these models to find bounded-suboptimal paths in significantly less time.

We justified the use of the learned model with two conclusions

1. In Section 5.2.2 we observed that the learned model leads to significant speed up in planning times for the off-road navigation domain.
2. In Section 5.2.5 we observed that regardless of how bad the learned model, the planner is able to still generate a valid plan albeit the time it might take to find that solution might increase to that point that using the learned model would actually slow down planning. Therefore, our algorithm would work well when an accurate learned model is provided. This requirement is easy to fix as we can generate data from the simulator to train the learned model offline.

To support our motion planning algorithm, we also developed a simulator to easily setup and simulate an off-road driving simulation. The native Unreal Engine/CARLA simulator system uses the NVidia PhysX engine for wheeled vehicles by default. This introduces a few known limitations in its accuracy:

1. Each wheel is only collision checked along a single ray

2. Terrain (and thus physics) not modified by weather
3. Terrain is not deformable (no high-centering on snow or mud, for example)

ISimA implements a modified terrain model to improve fidelity of off-road simulation and provide a proof-of-concept for more advanced terrain interaction models. This module plugs into CARLA and is implemented via an interface with the '**Custom Vehicle Physics**' module as detailed in Figure 4.4.

The prototype implementation provides improved fidelity by determining the terrain type under each point of contact of the vehicle. In this way, torsional effects can be accurately modeled (for example if the left wheels are on mud while the right wheels are on solid ground) as well as effects from having one (or more) wheels lose contact with the ground (for example, while traversing uneven terrain, it may be possible to lift one wheel off the ground and the wheel-terrain model would update).

6.1 Future Work

An immediate direction of future work would be to employ a more intelligent method to schedule the evaluation of the potentially pessimistic edges. Currently (see Section 5.1.2) we schedule their evaluation only when there are no path candidates left to consider. However, if a very bad learned model is provided and the search graph is very large, the search may be guided towards very sub-optimal paths. This will not only cause delays in finding a path which satisfies the sub-optimality bound, but also delay the correction of mistakes the learned model might have made.

6.2 Final Remarks

In this work we rely on the ability of the robot to perceive the scene around it and recreate the scene in a simulator. While we agree that this is a difficult requirement to fulfill, however we also think that a physics-based simulator is going to be essential to reason about complex terrains. This claim is supported by the investment in simulation technology that we see in the self-driving industry today. However, we believe that with the onset of differentiable simulators, it would become easier to use data to bridge the gap between simulation and reality.

Appendix A

Comparison of Existing Simulation Systems

Table A.1 shows the detailed comparison between 13 simulator platform candidates.

Table A.1: Yellow – partial support for requirements; Red – no or limited support for requirements; Blank – unknown level of support

Name Base Simulation System	Physics Engine	State of tire / road interaction physics Physics for driving simulation Any off-road limitations	Ease and quality of Custom environment Off-road	Open source / Free License	ROS	Easy to scale up to Photo realistic rendering	GPU vs CPU for Physics Engine Real time factor	Sensors Existing sensors + ability to add new	Pedestrian, weather and vehicles support	Plugin Support + Documentation	Operating System + Cloud Support	Ability to simulate Communication delays	How would we be able to modify road-tire interactions: 1) Change params of an in-built model 2) Change the model 3) None
Unreal Base for a lot of high-fidelity simulators	Yes. Very good physics engine made for Games and Simulators Uses Nvidia PhysX	Supports basic interaction. Some attempts to add a high-fidelity road-tire interaction model in progress. No	Has Unreal editor to edit environments. Tools to add in common elements like foliage, landscape. Extensive tools to create landscape and terrains	Free to use for our use-case	[1] Provides a wrapper for ROS. Not sure about 2.0	State of the art	GPU RTF Y	Camera exists. [1] provides GPS, Lidar, IMU. Can add new sensors	Yes, has a pedestrian modelling system. Lots of 3rd party pedestrian AI plugins exits	Support visual scripting language (Blueprint) for prototyping. Which can be converted to efficient C++. Support python scripting.	Windows recommended for manually modeling environments Supports Linux	have to emulate communication delay bw agents	Change params of the PhysX model Can also change model
Carla Built on Unreal with driving features	Uses Unreal Engine	Same as Unreal. Except, addition such as [1] might break other features	One can create custom maps, but need to use specific description for road, traffic lights, foot-paths etc.	Yes MIT license But requires UE4 license	ROS bridge exists. No stable support for 2.0 yet.	Uses UE4	GPU RTF Y	Yes, has all autonomous driving related sensors. Can add new ones.	Has support to control weather. Controller pedestrian and other traffic might be somewhat restrictive.	Based on C++ and open source. Large community	Supports Both	-same-	Same as above for vehicle. Can add cuboidal patches in the env with diff friction
AirSim Built on Unreal	Uses Unreal	Ability to write new physics model:	Can load custom UE4 environments easily	But requires UE4 license	Partial support	Uses UE4	GPU RTF Y	Has the common sensors	No pedestrians. Can add other vehicles and weather	Opensource. Not good documentation support	Support both	-same-	Can change both model and params.
Nvidia Drive Sim		Very high-fidelity road-tire interaction.		Not open source									
Bullet	Has better n-body simulation than Nvidia PhysX	Has a very simple vehicle model. But Bullet is highly configurable	Not easy. Not developed to support off-road terrains	zlib license	Good ROS support	Does not support photoreal rendering. In future might get integrated with Unity	GPU RTF Y	Camera, Lidar. Not high-fidelity	None	Large community. Python and C++ plugins	Both	-same-	It is a barebones physics engine. Need to model everything ourselves.
Gazebo Link	ODE,Bullet,Simbody, DART	Vehicle plugin exists. But not does not have realsic models	Difficult	Yes	Yes	Uses Ogre3D. Not photoreal	CPU RTF N	Yes	Weather: not off the shelf.	Very configurable with plugins	Linux preferred	-same-	Custom model
Coppeliasim Link	Bullet Physics, ODE, Newton and Vortex	Vehicle plugin exists. But not does not have realistic models	Difficult	Yes. Source code only for EDU	Yes	No	CPU RTF N	Yes	Weather: not off the shelf.	Plugins can be written in Python, Lua, C, Java, Matlab	Linux preferred	-same-	Custom model
Unity	Uses Nvidia PhysX	High-fidelity pay-ware plug-ins available for vehicle physics	Has user friendly tools to generate off-road terrains	Not free. $1800 / year	None	Has photorealistic rendering	GPU	Camera yes. Rest: Not off the shelf. Need to develop or buy 3rd party assets	Weather yes. Off the shelf pedestrian no.	Mostly close source and 3rd party asset on the store	Windows is preferred.	-same-	Probably only params
rfpro	High quality vehicle dynamics model		Can use PC for terrain	Not free	Not sure	Yes	GPU	Yes	High quality weather	Closed source	-	-same-	Not enough information
SUMO			Not for off-road			Someone used Unity for viz						C2X communication technologies by coupling to a communication network simulator (OMNeT++ or ns-3)	
LGSimulator	Nvidia PhysX		Unity	Has free + pro versions	Yes	Yes	GPU	Yes	Not sure	Possible alternative in the future, but not enough support at current time			
ADAMS				Not free						Possible alternative in the future, but not enough support at current time			
Anvel				Not available									

Bibliography

[1] Said Al-Milli, Kaspar Althoefer, and Lakmal D Seneviratne. Dynamic analysis and traversability prediction of tracked vehicles on soft terrain. In *2007 IEEE International Conference on Networking, Sensing and Control*, pages 279–284. IEEE, 2007. 1

[2] Said Al-Milli, Lakmal D Seneviratne, and Kaspar Althoefer. Track–terrain modelling and traversability prediction for tracked vehicles on soft terrain. *Journal of Terramechanics*, 47(3):151–160, 2010. 1

[3] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001. 1

[4] R Omar Chavez-Garcia, Jérôme Guzzi, Luca M Gambardella, and Alessandro Giusti. Image classification for ground traversability estimation in robotics. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 325–336. Springer, 2017. 1

[5] R Omar Chavez-Garcia, Jérôme Guzzi, Luca M Gambardella, and Alessandro Giusti. Learning ground traversability from simulations. *IEEE Robotics and Automation letters*, 3(3):1695–1702, 2018. 1

[6] Benjamin Cohen, Mike Phillips, and Maxim Likhachev. Planning single-arm manipulations with n-arm robots. In *International Symposium on Combinatorial Search*, volume 6, 2015. 2.2

[7] Thomas J DiCiccio and Bradley Efron. Bootstrap confidence intervals. *Statistical science*, 11(3):189–228, 1996. 5.2.4

[8] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. 5.2.2

[9] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017. 4.1.3

[10] Donald B Gennery. Traversability analysis and path planning for a planetary rover. *Autonomous Robots*, 6(2):131–146, 1999. 1

[11] Nika Haghtalab, Simon Mackenzie, Ariel Procaccia, Oren Salzman, and Siddhartha Srinivasa. The provable virtue of laziness in motion planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, pages 106–113, 2018. 2.2, 5.2.1

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 2, 2.1, 2.2

[13] Noriaki Hirose, Amir Sadeghian, Marynel Vázquez, Patrick Goebel, and Silvio Savarese. Gonet: A semi-supervised deep learning approach for traversability estimation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3044–3051. IEEE, 2018. 1

[14] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 3

[15] Brian Hou, Sanjiban Choudhury, Gilwoo Lee, Aditya Mandalika, and Siddhartha S Srinivasa. Posterior sampling for anytime motion planning on graphs with expensive-to-evaluate edges. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4266–4272. IEEE, 2020. 2.2

[16] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic generalization measures and where to find them. *arXiv preprint arXiv:1912.02178*, 2019. 3

[17] Jean-Claude Latombe. *Robot motion planning*, volume 124. Springer Science & Business Media, 2012. 1

[18] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006. 1

[19] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. *Advances in neural information processing systems*, 16, 2003. 2.1

[20] Aditya Mandalika, Sanjiban Choudhury, Oren Salzman, and Siddhartha Srinivasa. Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 745–753, 2019. 2.2, 5.1.1

[21] MG Nampoothiri, B Vinayakumar, Youhan Sunny, and Rahul Antony. Recent developments in terrain identification, classification, parameter estimation for the navigation of autonomous robots. *SN Applied Sciences*, 3(4):1–14, 2021. 1

[22] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984. 1

[23] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):

287–296, 1985. 4.2.1

[24] S. Singh, R. Simmons, T. Smith, A. Stentz, V. Verma, A. Yahja, and K. Schwehr. Recent progress in local and global traversability for planetary rovers. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 1194–1200 vol.2, 2000. doi: 10.1109/ROBOT.2000.844761. 1

[25] Han Vanholder. Efficient inference with tensorrt. In *GPU Technology Conference*, volume 1, page 2, 2016. 4.2.2