Dynamic Deadlines in Motion Planning for Autonomous Driving Systems

Edward Fang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-58
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-58.html

May 25, 2020

Acknowledgement

I would like to thank Professor Joseph Gonzalez for advising me through the past 3 years. He provided incredible insight and guidance for my academic and professional career. I would also like to thank the ERDOS Team: Ionel Gog, Sukrit Kalra, Peter Schafhalter and Professor Ion Stoica for supporting my research efforts.

I would like to thank Ashley for always supporting me throughout my studies and helping me see things from different perspectives.

I would like to thank my mom, dad and brother for their unconditional and unbounded support.

Dynamic Deadlines in Motion Planning for Autonomous Driving Systems

by

Edward Fang

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Gonzalez
Professor Ion Stoica

Spring 2020

Abstract

Dynamic Deadlines in Motion Planning for Autonomous Driving Systems

by

Edward Fang

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Joseph Gonzalez

Autonomous vehicle technology is posed to increase safety and mobility while decreasing emissions and traffic in the transportation industry. Safe, reliable and comfortable autonomous vehicle technology requires the cooperation of robust driving algorithms, failsafe hardware and real-time operating systems. While all components are important, safety-critical motion planning algorithms are paramount as they are responsible for maneuvering the car through complex, dynamic environments.

Motion planning is still an unsolved task in the autonomous driving realm. There are a variety of challenges and shortcomings associated with current state of the art systems, one primary concern being static deadline motion planning. The objective of this paper is to explore the tradeoffs of a variety of planning algorithms in a practical, realistic driving environment. We show how static deadlines are a limitation in current autonomous vehicle motion planners and lay the groundwork for future work using a dynamic deadline system to allow flexible time constraints. We propose that allowing flexible time constraints in motion planning has significant improvements in safety, comfort and reliability over static deadline alternatives.

To my family and friends.

Contents

List of Figures

Acknowledgments

I would like to thank Professor Joseph Gonzalez for advising me through the past 3 years. He provided incredible insight and guidance for my academic and professional career. I would also like to thank the ERDOS Team: Ionel Gog, Sukrit Kalra, Peter Schafhalter and Professor Ion Stoica for supporting my research efforts.

I would like to thank Professor Fernando Perez and Professor Joshua Hug for allowing me the opportunity to teach alongside them as an undergraduate student instructor. I found our interactions to be invaluable. To my peers, friends and fellow bears, I am indebted to your kindness, wisdom and friendship throughout the last 5 years.

I would like to thank Ashley for always supporting me throughout my studies and helping me see things from different perspectives.

I would like to thank my mom, dad and brother for their unconditional and unbounded support.

Chapter 1

Introduction

Autonomous driving is the metaphorical modern day space race, in the sense that it is an immensely challenging task that spans numerous domains. Successful self-driving systems require innovation not only in artificial intelligence, but also in real-time operating systems, robotic algorithms and failsafe hardware, to name a few. If self-driving matures, there are many positive implications for society. For example, commuters who drive in dense, urban areas waste many hours in congestion every year. Commuters in San Francisco lose roughly 97 hours, while those in New York lose 140 hours annually [32]. Self-driving will also benefit the environment. The transportation industry makes up 29 percent of all greenhouse gas emissions [2]. Within transportation, over 80 percent of emissions come from vehicles and trucks, meaning they contribute roughly 23 percent of all greenhouse gas emissions. The number of vehicles owned has been increasing linearly over the past 2 decades [62], inevitably contributing to emissions. Self-driving cars reduce the demand on consumer-owned vehicles and shift vehicle transportation to a ride-sharing model that would reduce the number of vehicles on the road, simultaneously addressing traffic and emissions concerns. Most important, self-driving vehicles could reduce the number of fatal motor vehicle crashes. The USA saw 34,560 fatal crashes in 2017 [70]. These crashes are due to human error, which self-driving cars can reduce as they become mature.

Again, these are only potential benefits and only realized when autonomous technology surpasses human level performance in terms of safety and comfort. The 34,560 fatal crashes occurred over 3,212 billion miles, meaning human level performance is somewhere near the order of magnitude of 1 accident per 100 million miles. Current leaders in autonomous technology such as Waymo and Cruise have logged only 3 percent of those 100 million miles, combined [52]. In those 3 million miles, Waymo vehicles disengaged roughly once every 13,219 miles and Cruise vehicles disengaged once per 12,221 miles. Evidently, there is a huge gap between current state-of-the-art technology and human-level performance.

The California Department of Motorvehicles defines a disengagement as a "deactivation of the autonomous mode when a failure of the autonomous technology is detected or when the safe operation of the vehicle requires that the autonomous vehicle test driver disengage the autonomous mode and take immediate manual control of the vehicle" [49]. Many of the

disengagements reported by the California Department of Motor Vehicles are fundamentally related to shortcomings of current motion planning algorithms. These disengagements range from slight deviations from the center of the lane to potentially fatal collisions had the safety driver not intervened. Naturally, safer motion planning will improve the true reliability of autonomous vehicles and will be reflected in the disengagement data. While fast, real-time planning algorithms are a must, faster does not necessarily equate to safer. It may be advantageous to search a finer space of feasible trajectories, rather than prioritizing lower latency. In other cases, it may be better to prioritize low latency in high-speed scenarios. Motion planning must share its run-time with other components in the autonomous vehicle stack and thus having a static deadline may be sub-optimal. Thus, we propose dynamic run-time allocation for motion planning as direction towards reliable self-driving technology.

This thesis addresses the trade-offs in various types of motion planners by utilizing ERDOS [26] and Pylot [27] as the frameworks for developing these algorithms. We study three different motion planning algorithms, analyze their shortcomings and strengths in practical, realistic experiments, and propose a dynamic deadline solution to the planning problem. We use Carla [16], a simulator designed to test self-driving vehicles, to substantiate our experiments in physically realistic situations and environments. The rest of the thesis is structured as follows: 2 presents a high-level overview of autonomous vehicle technology. 3 describes motion planning in depth and discusses related works in motion planning. 4 details the motion planning algorithms of interest. 5 describes the experiment setup, including implementation details, vehicle modeling, control setup and simulator configuration. 6 analyzes the experiment results and comments on interpretability of the results. 7 discusses future works.

Chapter 2

Overview of Autonomous Vehicle Technology

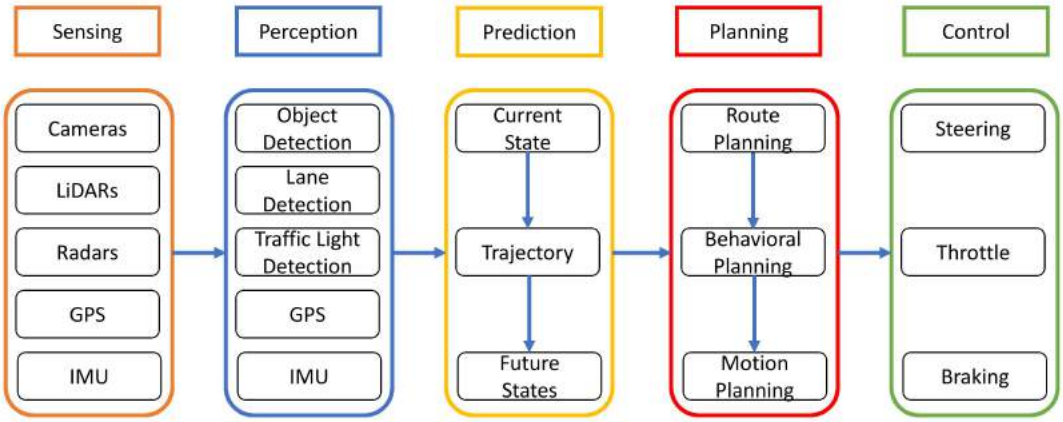

Figure 2.1: A canonical example of the autonomous driving stack as a dataflow graph.

Although end-to-end approaches to self-driving exist [5], the typical autonomous driving technology stack can be decomposed into roughly 5 categories [54]: sensing, perception, prediction, planning and control. While this thesis' focus is in planning, we still broadly describe each component as they are significant in understanding the motivation behind our

method. Then, we discuss the decision-making systems used in planning and control. We refer to our autonomous vehicle of interest as the "ego-vehicle".

2.1 Sensing

The sensing component is responsible for ingesting streams of data regarding the driving environment. This should contain information related to where the ego-vehicle is, where other road users are relative to the ego-vehicle, the pose and inertial measurements of the ego-vehicle, and so on. On-board sensors usually consist of an array of LiDARs, cameras, GPS/INS units and radars. The data is used downstream in conjunction with prior knowledge such as speed limit, traffic regulations, etc.

Managing these sensors is difficult. All sensors need local clocks for synchronization, else the measurements from the different sensors can be inconsistent: eg. radar measurements and LiDAR measurements at different points in time can indicate different world states. Time synchronization is a known research problem and while there are accepted solution frameworks [53, 61, 20], the exact solution to a problem is highly dependent on its setup and requirements. For example, synchronization in autonomous vehicle sensors probably requires a higher degree of precision, as the state of the world can vary drastically given 100 ms vs 10 ms. These sensors also have different resource requirements on energy, storage and bandwidth, further adding difficulty to the problem. While some naive solutions will work fine for a couple of sensors, self-driving vehicles have an atypical amount and variety of sensors to help understand the driving environment.

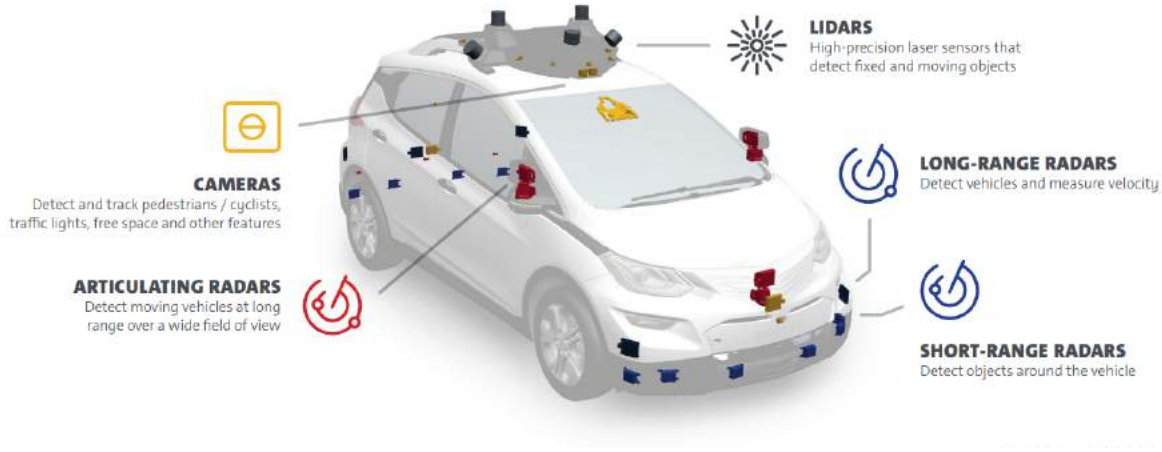

Figure 2.2: Cruise Automation's autonomous vehicle sensor configuration. Adapted from General Motor's 2018 Self-Driving Safety Report. [50]

For example, Cruise autonomous vehicles use 5 LIDARs, 16 cameras, and 21 radars along with IMU, GPS, etc [50]. These streams of data operate at different rates, are variable in size,

and produce 10s of gigabytes per second. While synchronizing and processing this amount of data is a challenge in itself, we highlight the variability in timing as one motivation for dynamic time allocation motion planning. Any variability in the sensing module will generally propagate variation downstream, to perception, prediction, planning and control. Being able to account for this variability in the other modules and in particular, motion planning, can greatly improve safety and robustness of the autonomous vehicle.

2.2 Perception

Figure 2.3: Object detection in Carla. Here we visualize our pipeline running end-to-end in Carla. On screen we visualize the Faster R-CNN [57] bounding box predictions from a first person perspective. Faster R-CNN successfully identifies the truck and pedestrian in this scenario. This information is used by motion planning to produce a safe trajectory between the obstacles. The trajectory is shown as red points.

Perception processes data from LIDARs, cameras and radars to produce an intelligible representation of the environment around the ego-vehicle. It typically uses a combination of deep neural networks, heuristics and sensor fusion to produce a high definition road map. This road map contains information such as driveable lane estimations and traffic light state detection. Localization also takes place in the perception component, where techniques like SLAM [18] can be applied.

For a comprehensive study of computer vision applications in autonomous vehicles, we refer the reader to [33]. Recent advances in computer vision, namely object detection, tracking and activity recognition have greatly improved the onboard perception of autonomous vehicles. Improvement in object detection is driven by region proposal methods [71] and region-based convolutional neural networks [25]. While these models were computationally expensive, methods such as Faster R-CNN improved run-time and turned deep neural net approaches into viable, real-time perception algorithms. Since Faster R-CNN [57], numerous datasets [19, 47, 22, 65, 37, 78], benchmarks [22, 47, 19] and models [33] have been developed in the space of object detection. Tracking has also seen rapid growth, with rigorous benchmarks such as MOT15, 16, 17 etc [43]. Current state of the art includes hybrid deep learning and metric-based models such as DeepSORT [76] and pure deep learning approaches such as Siamese networks [39].

These promising developments are spurred on by hardware innovations that allow for faster, larger models with more learnable parameters. While accuracy and run-time have improved over the years, they have generally done so in a mutually exclusive manner. Some methods such as Faster R-CNN have been able to improve run-time while holding accuracy constant, but usually there is a trade-off. Models with more parameters are better able to fit the data but take longer to run. Faster models have fewer parameters, but can have increased bias as a consequence. As shown in table 5.1 of [33], there is huge variability across the current state of the art object detection models in terms of run-time and accuracy. The highest accuracy model has a run-time of 3.6s/GPU while the lowest accuracy runs at 0.02s/GPU. A review [31] of speed / accuracy trade-offs in 2017 also highlights the variability in internal model run-time, with the maximum being roughly 150 ms. These variations can mainly be attributed to the number of objects in the scene.

In the context of self-driving, if the vehicle is moving at 10 m/s, then in 100ms the car will have traveled 1m. At highway speeds of 25 m/s, timing is even more critical. Thus, slight variations in perception run-time can have a significant impact on the "reaction time" of the vehicle. It may be better to use a faster, less accurate perception model when we want to run a higher precision motion planner, or vice-versa.

2.3 Prediction

Once the ego-vehicle understands its surroundings, it can project what the future state of the environment could look like in the prediction component. The prediction component foretells what other dynamic agents in the environment will do using cues from perception, such as tracking the velocity over time and lane localization. It can also use priors of how non-ego agents typically act under similar circumstances to utilize historical trends [58]. Other methods use a mixture of Gaussians models. For example, Waymo's MultiPath [12] uses a top-down scene features and a set of trajectory anchors to produce a mixture of Gaussians model for forecasting trajectories.

The prediction component processes the state of the world constructed by perception and

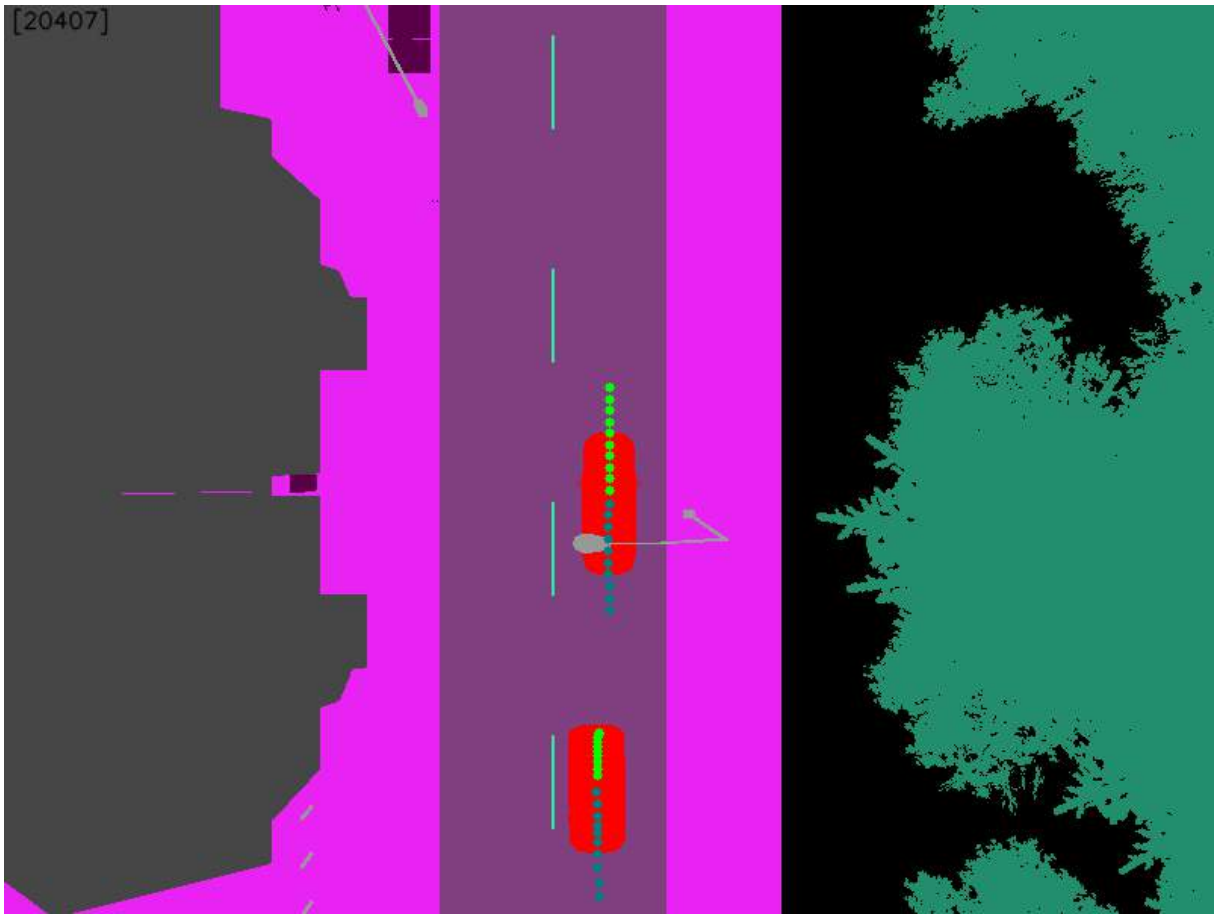

Figure 2.4: Top down view of linear prediction in Carla. Cyan points are past poses and green points are predicted future poses. Here we use the past 10 poses to predict the next 10 poses at 10 hz. In other words, we extrapolate 1 second into the future using a linear predictor. The scene is colored by semantic segmentation.

passes it to motion planning. It is responsible for providing accurate, reasonable information about future states of the world. It is generally accepted as an unsolved challenge as real world behaviors can be truly unpredictable. Especially in complex urban environments, there are numerous pedestrians, cyclists and vehicles to account for, not to mention other oddities like scooters, emergency vehicles and construction workers. Modelling accurate behavior for so many different types of agents is challenging. Data-driven methods such as MultiPath or R2P2 [58] can be susceptible to sampling biases, particularly where and when the data was collected. Underlying distributions can often be correlated with the time and location, so collecting sufficient amounts of data for each slice of time and location can be impractical. In extreme cases like the 6-way intersections in San Francisco, data must be actively collected in order to handle the geometry and possibilities, otherwise it would be impossible to predict

future trajectories.

In the Carla simulator, it is possible to abstract the perception problem and prediction problem by accessing internal information such as non-ego pose and paths. Carla exposes the world state, allowing us to query for topics such as vehicle position or traffic light state. As this paper is concerned primarily with motion planning, our methods and experiments abstract as much of the noise resultant from perception and prediction by utilizing the aforementioned Carla features. In this way, we can analyze the impacts of our research as a self-contained methodology.

2.4 Planning

The planning component must produce a safe, comfortable and feasible trajectory that considers the present and future possible states of the surroundings. It is usually broken down into a high-level route planner, a behavioral layer that provides abstract actions such as "prepare to lane change left", and low-level motion planners that produce a trajectory. We will discuss more related work in 3.

2.5 Control

Finally, a control module actuates throttle, braking and steering to track the trajectory produced by planning. The simplest and most commonly used controller is the proportional integral derivative (PID) controller [38], which is used to track a reference velocity in the case of cruise control. One of the downsides is that it requires extensive tuning, as the controller parameters are affected by different velocity, surface and steering conditions. Yet it is effective and commercially used in many systems, including autonomous driving systems. Model-based controllers are also effective in controlling autonomous vehicles. Many controllers utilize the kinematic bicycle model [41] to compute inputs to drive-by-wire systems [68]. Some popular controller frameworks include pure pursuit (commonly used in the DARPA challenges [11]) and model predictive control which forward simulates an open loop control and re-solves it for each iteration.

Control theory is an extensive field of research and is not solely trajectory tracking. Lower level controllers are crucial for the safety and comfort of the ride, regardless of the trajectory. Traction control maximizes our tire friction when accelerating, braking and turning so that the vehicle does not slip out of control. Anti-lock braking systems ensure that our braking force is maximized and the tires do not skid. We focus on the trajectory tracking aspect of control and discuss our control setup in section 5.

Each of these components are required for a functional self-driving vehicle. Within each component, there is high variability in the run-time and accuracy across different models and internally within each model. Given this high variability, we propose that it is beneficial to dynamically allocate run-time for different components, particularly for planning. Ac-

counting for run-time uncertainty and adapting planning configurations on the fly can have significant impact in safety critical situations. To provide more insight, we will discuss planning in depth in the following section. We show how motion planning is typically structured and cover the strengths and weaknesses of different classes of motion planners.

Chapter 3

Motion Planning Survey

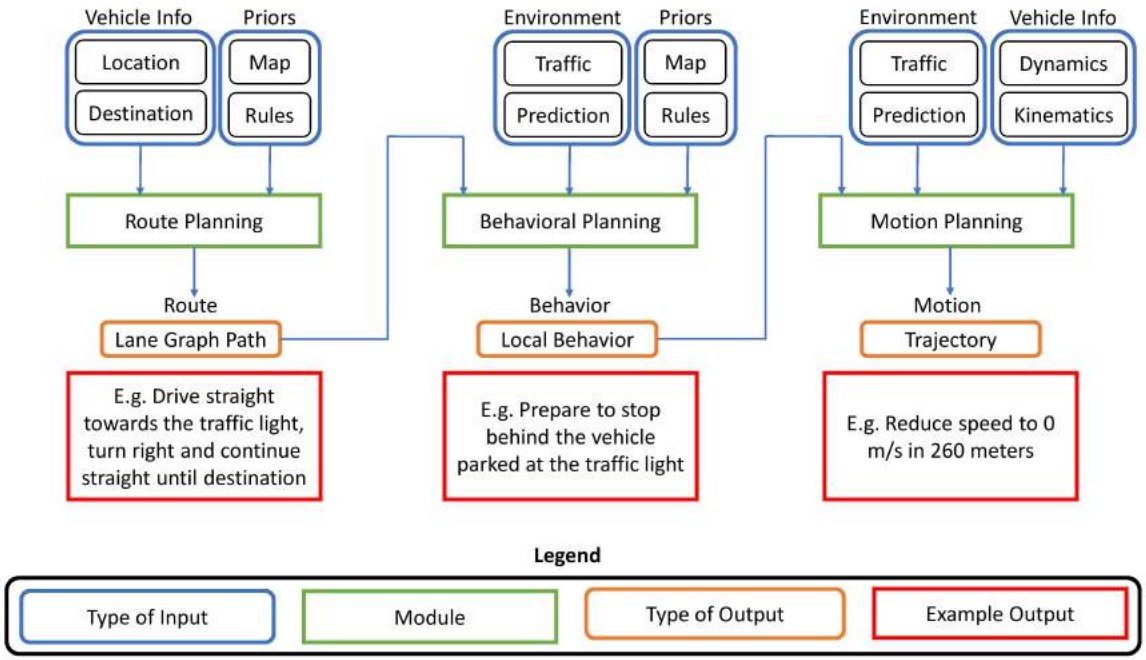

Figure 3.1: A canonical motion planning hierarchy. The rounded boxes at the top represent various modalities of input, while the rounded boxes in the middle represent different outputs. The boxes at the bottom serve as example outputs that feed into downstream layers.

At a high-level, motion planning produces a safe, comfortable and feasible trajectory that accounts for the present and future possible states of the environment. It is usually decomposed into 3 sub-modules: route planning, behavioral planning, and motion planning. We will explain the challenges and solutions for each sub-module.

3.1 Route Planning

Figure 3.2: A* Route planning over the road graph in Carla. This search takes into account the driving direction of the lane, but does not consider static or dynamic obstacles. The nodes in the road graph are coarsely spaced waypoints on the road and the lanes are connections between these waypoints. Here red indicates the path that has already been covered by the ego-vehicle, while blue represents the path that has yet to be covered.

Route planning produces a route through a road graph given a starting and an ending location. Edges in a road graph typically represent connections between lanes, intersections and so on. Nodes represent points on the road and are usually coarse in resolution. For example, vertices could only exist at intersections. Edges would then correspond to roads connecting the intersections. The road graph representation naturally gives rise to weighted graph search algorithms that find a minimum-cost path through the network. Popular algorithms for route planning include A* [51], Dijkstra's [14] and bidirectional search. These algorithms work well for small road graphs, but are infeasible for networks with millions

of nodes and edges. There are numerous methods [28, 23, 6] that work around a one-off pre-processing step that allows for much faster search later on.

3.2 Behavioral Planning

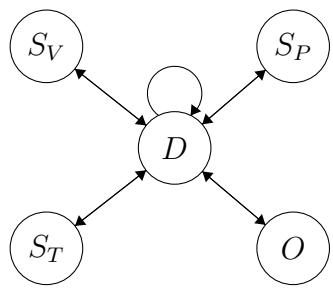

Figure 3.3: A simple behavioral planner represented as a finite state machine. D denotes driving forward along the global route. S_V denotes stopping for a vehicle. S_P denotes stopping for a pedestrian. S_T denotes stopping for traffic rules. O denotes driving around an obstacle.

Behavioral planning uses the route plan and produces a high-level behavior for the vehicle to follow. For example, if the vehicle is in the left-most lane and needs to make a right turn, an appropriate behavior would be: prepare to change lane right. This process can be modeled as a Markov decision process or partially observable Markov decision process (MDP or POMDP, respectively). In the DARPA Urban Challenge, competitors used finite state machines in conjunction with heuristics to model driving scenarios. But in the real world, these state machines can become large and complicated due to the uncertainty of human drivers and other traffic participants. Thus more recent work has focused on probabilistic decision making under machine learning approaches [30, 69], model based methods [73, 77], and MDP or POMDP models [10, 72, 9, 21, 4].

3.3 Motion Planning

For a more comprehensive and detailed overview of motion planning, we refer readers to [42, 35, 55] which serve as the basis for this discussion. We analyze the strengths and weaknesses of a select set of motion planning algorithms which include trajectory generation, graph search, incremental search methods, and real time planning algorithms.

Using the behavior produced by the behavioral layer, the motion planner computes a safe, comfortable and feasible trajectory that moves the vehicle from an initial configuration to a goal configuration. The configuration format depends on the type of motion planner: planners that operate in the Frenet frame [75] may use a target position and lateral offset

Figure 3.4: Planning and prediction in Carla. Here we visualize our planning and prediction modules in the Carla simulator. In this scenario, the ego-vehicle should follow the global route indicated by the blue points. Green points represent predicted trajectories of other agents. We note that the ego-vehicle is stopped behind a car at a red light. The motion planner must account for the global route, dynamic obstacles and traffic state to safely produce a motion plan.

along a path as the goal configuration, while a Rapidly-exploring Random Tree (RRT) [34] planner may uses the final x, y, z position as the configuration. Regardless of the configuration, the motion planner must account for the road shape, the kinematic and dynamic constraints of the vehicle and static and dynamic obstacles in the environment. Most motion planners also optimize an objective function to eliminate uncertainty when there are multiple feasible trajectories. An example of an objective function is minimizing a combination of accumulated lateral jerk and time to destination, which are heuristics for comfort and timeliness.

Motion planning methods can be divided into path planning methods and trajectory planning methods. Path planning methods propose a solution that maps $p(t) : [0, 1] \rightarrow X \in C$, where X is some vehicle configuration and C is the configuration space. This framework is agnostic to time and does not enforce how a vehicle should follow the path, leaving the responsibility to the downstream modules. Trajectory planning methods propose a solution that maps $p(t) : [0, T] \rightarrow X$. This means time is explicitly accounted for in the evolution of a motion plan. Trajectory planning methods prescribe how the vehicle configuration progresses

with time. Under this framework, vehicle dynamics and dynamic obstacles can be directly modeled.

Within the path planning and trajectory planning frameworks, there are numerous subcategories of planning algorithms. Our work focuses on analyzing Hybrid A*, RRT*, and Frenet Optimal Trajectory. Thus in the following sections we will take a look at their corresponding planning subcategories, namely graph search, incremental search, and trajectory generation, respectively. This analysis underscores potential strengths and weaknesses of each planner and allows for further discussion in section 6, with our results.

3.3.1 Graph Search

Graph search methods represent the configuration space as a graph $G = (V, E)$. V is a subset of the configuration space C and E is a set of edges that connect vertices in V with a path. Graph search methods can be further categorized as lane graph, geometric and sampling methods.

Lane graph methods use a manually constructed graph of lanes and can be queried with algorithms such as A* or Dijkstra's. Geometric methods involve constructing geometric representations of driveable regions and obstacles. Some popular geometric representations include generalized Voronoi [66] diagrams and artificial potential fields [79]. Solutions to geometric methods can be directly calculated from the geometric representation.

Sampling methods are generally more effective than its lane graph or geometric counterparts. Lane graph solutions break down when there is a dynamic agent that is unaccounted for during the initial construction of the graph. For example, an autonomous vehicle must be able to drive around a doubly parked car in San Francisco, but cannot do so if the road graph had not accounted for the possibility of an obstacle at that location. Generalizing these uncertainties leads to overly complex lane graphs and excessive manual engineering. Geometric representations can be expensive to produce from raw sensor data. Furthermore, the kinematic and dynamic constraints of real vehicles far exceed those provided by geometric representations. For example, Voronoi diagram path planning yields discontinuous, linear piecewise paths that are not trackable by nonholonomic vehicles. Nonholonomic systems are systems in which the number of controllable degrees of freedom are less than the total degrees of freedom. The car is nonholonomic as we can control steering and throttle, but Cartesian position (x, y) and heading (yaw) are the degrees of freedom. Sampling methods are more popular as they can freely represent kinematic and dynamic constraints and the configuration space. Typically, sampling based methods determine reachability of a goal configuration using steering and collision checking steps. Steering is responsible for producing a path from a given point A to given point B. Collision checks for collisions along the path.

There are many types of steering functions. Random steering applies a sequence of random controls for a period of time. Evidently, this results in suboptimal paths that are not smooth and often far off from the goal location. Heuristic steering addresses some of these shortcomings by applying control that progresses the vehicle towards the goal location.

Dubin's curve [17] and Reeds-Shepp's curve [56] for unicycle models are exact steering procedures. Exact steering procedures are usually optimal with respect to some cost function and do not account for obstacles along the path. In particular, Dubin's and Reeds-Shepp's curves are arc-length optimal under the assumption that there are no obstacles to be considered. Thus, these exact steering procedures are generally used to connect intermediate vertices in a graph, rather than construct a plan from the start to the goal.

Basic collision checking should take into account the vehicle geometries such as the wheelbase length, track length, and so on. This means that pose information must be available. Pose information can be obtained from path plans by approximating the yaw along the path. For example, this can be done with a 2D Cubic Spline, which interpolates a series of points using a cubic polynomial. More advanced collision checking operates in trajectory space. In other words, it considers time as a factor in determining whether a trajectory results in a collision. This is generally better as obstacles in the environment are dynamic, but makes planning more dependent on a robust, accurate prediction model.

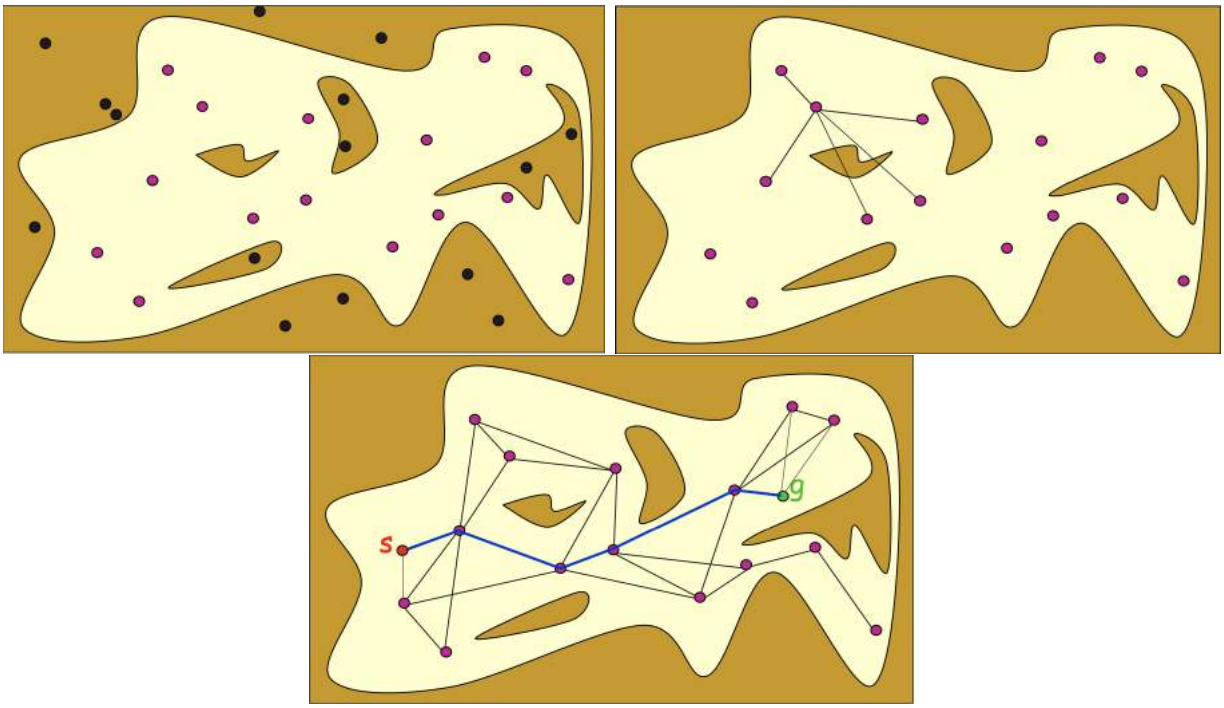

Figure 3.5: The Probabilistic Roadmap planner. The top left figure visualizes configuration sampling. Pink points represent valid configurations, while black points represent invalid configurations. The top right figure visualizes the connection of neighboring configurations and a validation check. The bottom figure visualizes the calculated path for some start, goal configuration in the PRM. These figures were adapted from Pieter Abbeel's CS287 course slides [1].

Some early works in sampling based methods include the Probabilistic Roadmap (PRM) [36] and its variants such as PRM* [34], which iteratively constructs a roadmap or an undirected graph of nodes (robot configurations) and edges (feasible actions connecting configurations). At a high level, PRM initializes a set of points with the start and end configuration. Then it randomly samples points in configuration space, creating a set of valid but potentially unreachable configurations. Then for each sampled configuration, nearby points are connected if they can be reached from each other. Finally it reconstructs the plan by concatenating the sequence of local edge paths into a feasible path from start to goal. The edge paths can be obtained by the aforementioned steer function and verified by the collision function. Dijkstra's or A* can also be used to construct these local paths. Rapidly-exploring Random Graph (RRG) [34] is similar to PRM and PRM* in functionality and time complexity, but has the added benefit of being anytime. This means an initial solution is computed and iteratively refined until the time deadline.

Some of the challenges associated with PRM are that connecting neighboring points is only easy for holonomic systems, ie. systems for which you can move each degree of freedom. This typically also requires solving a boundary value problem, in which we must satisfy some set of ordinary differential equations. Usually this is solved without considering collisions, and then collision checking afterwards. Collision checking, although simple in concept and not limited to just PRM, often takes the majority of time in applications [42]. There are a variety of obstacle representations, state space discretization hyperparameters such as time and space granularity, for example. The simplest obstacle representation is a sphere, ie. Each obstacle has a fixed radius that must be checked for collision. There are also axis-aligned bounding boxes, oriented bounding boxes and convex hulls, each providing a better approximation than the previous but also more expensive to test. Finding a practical yet safe trade-off between resolution and complexity is difficult. Furthermore, graph search techniques suffer from poor discretization of the search space. Poor discretization can lead to infeasibility or highly suboptimal paths. Graph search methods generate a fixed set of nodes in the initialization, so adjusting the discretization granularity across different iterations is not an option. High resolution discretizations are expensive to run, as the edges in the search space increase polynomially. In our experiments, these were some of the most significant, practical shortcomings of graph search methods. Choosing a sufficiently high resolution discretization that could be run real time across all scenarios is challenging, and thus motivates dynamic deadlines as a solution to this balancing problem.

We implement Hybrid A* [15], a graph search algorithm, for our experiments. Hybrid A* associates a continuous state with each cell as well as a cost with the associated state, and we will further discuss it in section 4.

3.3.2 Incremental Search

A key distinction between graph sampling methods and incremental search methods is that graph sampling methods generally precompute a fixed set of nodes ahead of time, whereas incremental search gradually builds a path by sampling the configuration space. Incremental

search techniques are similar to its graph sampling counterparts. It iteratively samples a random point in the configuration space, constructs a candidate set of neighbors and connects the new point to a neighbor using steering functions and validates using collision functions. Again, these sampled configurations are not contained in the graph initially, whereas they are predetermined in PRM, for example. This is one of the disadvantages of fixed graph discretization. Search over the set of paths is limited to the initializations or primitives of the graph discretizations. This results in solutions being infeasible or suboptimal.

Incremental search planners address this concern. They aren't limited by the initial construction of the graph and thus incremental search algorithms are generally probabilistically complete. This means that this class of algorithms will find a solution, if one exists, given sufficient computation time. A related term, asymptotically optimal, describes algorithms that eventually converge to the optimal solution. One can guarantee completeness and optimality by repeatedly solving path planning problems over increasingly finer discretizations of the state space. Still, incremental search techniques are unbounded in computation time. Although a feasible solution may exist, these techniques may take asymptotically infinite time to find it. This is a concern for real time / anytime tasks such as autonomous driving. Another concern is the stability of random algorithms. While there are bounds and expectations on the run-time and effectiveness of incremental search algorithms, the variance and robustness of these algorithms can be an issue. Poor random seeds or slight perturbations in obstacle locations or search space resolution can cause planners to fail. Given limited computation time, incremental search algorithms can often fail to find even suboptimal solutions, when graph search or trajectory generation methods are able to. Allocating a sufficient amount of computation time to these planners highly upon the hyperparameter configuration of the planner and the scenario the planner is operating in. Thus, dynamic allocation may outperform static allocations as the planners are able to adapt to the driving scenario.

Many incremental search algorithms utilize tree-growing strategies in conjunction with discretization tuning in order to get better coverage and exploration in the configuration space. LaValle's Rapidly-exploring Random Tree (RRT) [34] is an effective incremental search algorithm that works well in high-dimensional settings. It builds up a tree by generating next configurations through executing random controls. Many practical implementations will randomly attempt to connect with the end state in order to improve the run-time or promote exploration towards the goal. In order to efficiently find approximate nearest neighbors, optimizations such as KD Trees [7] or locality sensitive hashing [24] are used. These optimizations can also be applied when representing obstacle maps. We often only need to collision check proximal obstacles, which can be efficiently queried for using KD Trees. In the RRT pseudo-code below, select input must solve the two point boundary value problem, or just select the best out of a set of control sequences to find an input that moves the robot from xrand to near. The set of control sequences can be random or some well chosen set of primitives. Dubin's path and Reeds-Shepp's paths are often used as select input for autonomous driving as they can provide arc-length optimal paths between vertices. Many variations of RRT exist, including bi-directional RRT, resolution complete RRT (RC-RRT) and RRT* [34], which uses heuristics to guide the exploration and tree rewiring to shorten

Function RRT($x_{init}, K, \Delta t$)

Result: Tree T initialized at x_{init}
$T.init(x_{init})$
for $k = 1$ **to** K **do**
  $x_{rand} \leftarrow$ RANDOM_STATE()
  $x_{near} \leftarrow$ NEAREST_NEIGHBOR(x_{rand}, T)
  $u \leftarrow$ SELECT_INPUT(x_{rand}, x_{near})
  $x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$)
  $T.add\_vertex(x_{new})$
  $T.add\_edge(x_{near}, x_{new}, u)$
return T

Figure 3.6: RRT pseudocode. RRT utilizes a tree structure to incrementally explore the search space. K represents the number of iterations to run RRT, x_{init} represents the initial state and Δt represents the time step discretization. RANDOM_STATE returns a random state configuration from the configuration space. NEAREST_NEIGHBOR returns the nearest neighbor of a given node. SELECT_INPUT produces a control that connects state configurations. NEW_STATE executes the selected control and validates for collision. Finally, the new node is added to the tree and corresponding edges are updated.

paths through the tree when possible. For our experiments, we implement vanilla RRT*, which is detailed in 4.

3.3.3 Trajectory Generation

Trajectory generation methods account for kinematic, dynamic and road shape constraints when producing a motion plan. This allows trajectory generation methods to better handle dynamic obstacles and consider vehicle kinematics and dynamics, which is helpful when producing controllable, feasible plans. Trajectory generation usually generates path and speed edges separately. Paths are generated by connecting endpoints using different curvature polynomials. After paths are generated, candidate speed profiles are built for individual paths. To better understand the path and speed profile generation process, we describe planning in the Frenet frame.

In the Frenet frame [75], there exists a function $r(l) = [x_r(l), y_r(l), \theta_r(l), \kappa_r(l)]$ where l is the longitudinal offset. Let $p(l, d)$ be a point away from the road center at the given longitudinal offset l and d be the lateral offset away from the road center. Thus all our points are represented as longitudinal, lateral offset pairs relative to the road center. See figure 3.7 for details. This coordinate system yields nice properties, especially for autonomous driving. All points are represented relative to the road center, which is where we usually want our

$$
\begin{aligned}
p(l,d) &= [x_p(l,d), y_p(l,d), \theta_p(l,d), \kappa_p(l,d)] \\
x_p(l,d) &= x_r(l) + dcos(\theta_r(l) + \frac{\pi}{2}) \\
y_p(l,d) &= y_r(l) + dsin(\theta_r(l) + \frac{\pi}{2}) \\
\theta_p(l,d) &= \theta_r(l) \\
\kappa_p(l,d) &= (\kappa_r(l)^{-1} - d)^{-1}
\end{aligned}
$$

Figure 3.7: Frenet coordinate formula. Frenet coordinates can be calculated given a set of points using the above formula. Frenet coordinates are represented by x, y, heading (θ) and curvature (κ). They are parametrized by l and d, the longitudinal position along a path and the lateral offset from the path. x_r, y_r, θ_r, κ_r represent the position, heading and curvature of the target path, from which we can derive the Frenet coordinates using the formula.

autonomous vehicle to drive. It is a natural representation of the configuration space and yields interpretability for discretization parameters. It also gives rise to cost profiles that penalize deviation from the lane center and allows obstacles to also be represented in Frenet coordinates.

Thus, path generation can consist of connecting endpoints in the Frenet frame using quartic polynomials. Previous works used cubic polynomials [48], which worked well from single query planning situations. However, given that autonomous vehicles need to frequently replan at a high frequency, cubic polynomials generally yielded highly discontinuous trajectories, as endpoint continuity was not guaranteed. Thus, we use quartic polynomials that guarantee continuous curvature at the endpoints of trajectories. Speed profiles are also generated using quartic polynomials after discretizing the speed space in the Frenet frame. We refer readers to [75, 74] for details on Frenet planning.

The proposed trajectory generation method in the Frenet frame is essentially implemented as a grid search over candidate longitudinal and lateral motion plans. We can iterate over time, lateral, longitudinal, and velocity discretizations to produce an initial set of trajectories, which we then prune using physical constraint checks such as maximum acceleration or curvature. Each candidate can be assigned a cost based on a combination of factors such as lateral deviation, jerk, maximum curvature, and so on for final selection. A practical advantage of this is that it is highly parallelizable. Graph search and incremental search techniques are dependent on previous iterations of the algorithm, whereas Frenet frame trajectory generation candidates are independently generated. Still, this class of techniques suffers from poor discretization and initial configurations. If the discretization parameters aren't high resolution, candidate trajectories that are actually feasible may be omitted in the final selection. Again, this motivates a dynamic planner that can adjust the

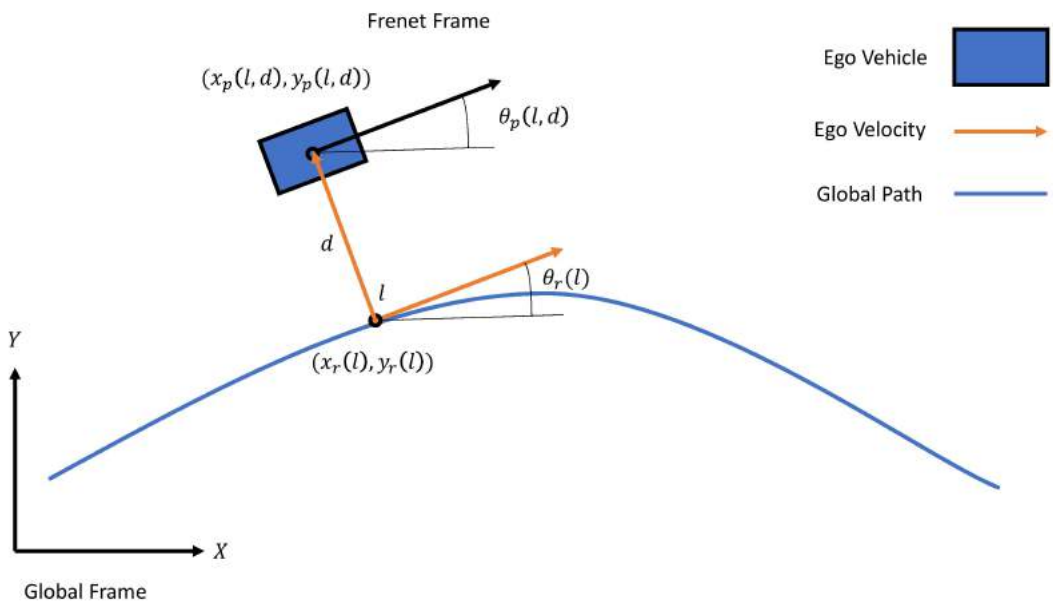

Figure 3.8: Frenet coordinate system. Above we visualize the Frenet coordinate system defined by the formula in figure 3.7.

configuration and consequently run-time based on the driving situation.

We implement the Frenet Optimal Trajectory [75, 74] planner for our experiments, described in 4.

3.3.4 Anytime and Realtime Planning

There are many optimized, real-time variants of the aforementioned motion planners. These real-time planners focus on removing any wasteful recomputation by only updating parts of the state space that have changed as per prediction and reusing information from previous search iterations. Usually these algorithms will provide an initial suboptimal path and then optimize the path until the deadline is up. In this framework, the planner will always be able to produce an executable trajectory given a base amount of time and will produce better trajectories with any extra time. State of the art includes algorithms such as D*, Anytime A* and their variants [63, 64, 29, 45, 46, 40].

Anytime and real time concepts are extremely important in production-level autonomous vehicles. It is necessary to eliminate as much redundant computation as possible and improve run-time. Although we do not analyze the performance of these planners under the dynamic deadline allocation framework, dynamically adjusting these planners can optimize the amount of extra time they have to improve their initial path hypotheses.

3.3.5 Interaction with Prediction and Control

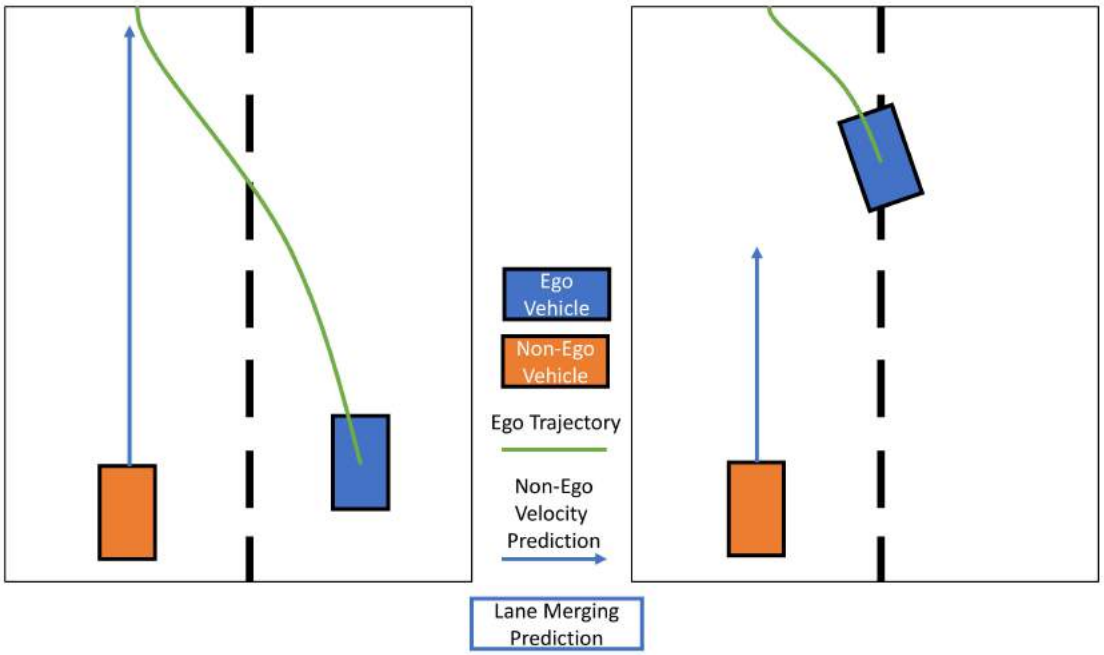

Figure 3.9: Planning's interaction with prediction. Here we visualize a simple scenario in which the ego-vehicle wishes to merge into the left lane. A naive prediction module may predict that the car in the left lane will continue driving forward at its current velocity. Prior knowledge suggests that when the ego-vehicle begins the merging maneuver, the vehicle in the target lane should slow down. On the left we visualize the state at some time $t1$. The blue ego-vehicle wishes to follow the green line, while the orange vehicle is predicted to drive straight at some velocity. On the right we visualize the state at some time $t2 > t1$. When the ego-vehicle begins to merge, it may predict that the vehicle in the left lane will slow down to accommodate the merge maneuver.

Motion planning has a unique position in the stack; it depends highly on accurate prediction in order to produce plans that account for realistic behaviors in the environment. Furthermore, any actions taken by the vehicle could impact the actions of non-ego agents in the environment, implying that the relationship between prediction and planning is undirected. The plans must also be controllable, suggesting that motion planners must be designed with consideration of the control module.

Joint prediction and planning is a rapidly developing research area that studies interactions between autonomous motion planning and non-ego agent intentions. The motivation behind these actions taken by the autonomous vehicle will inevitably impact how other vehicles behave, confounding prior assumptions made by prediction modules. Current research

[13, 60, 44, 59] in this area studies how drivers react to different changes and attempts to integrate this in prediction.

Planning and control are typically seen as separate processes; planning assumes control is a black box and only needs to consider basic kinematic constraints such as maximum acceleration, or turning radius. While control theory is well studied, especially for vehicle kinematics and dynamics, combined planning and control could yield safer, more comfortable trajectories. But combining these modules poses enormous optimization challenges, as the search space becomes enormous. Modularity is also a benefit of keeping the processes separate. Still, once candidate algorithms are identified for a specific application, there is generally some customization across the two processes. For example, the steer function for 4 wheeled motion planning can use the kinematic bicycle model predictive control to produce a trajectory. Ignoring the huge run-time overhead, the resultant path should be more controllable than a Dubin's steering function, as Dubin's assumes a unicycle model.

Chapter 4

Motion Planning Models

In this section we will detail the 3 motion planning algorithms of interest, Hybrid A*, RRT* and Frenet Optimal Trajectory. We will walk through the original, "base" algorithms as prescribed by their respective papers, explain relevant portions and hyperparameters with pseudo code, and describe implementation differences in our motion planners. We build upon the previous section and elaborate on the strengths and weaknesses, both theoretically and practically. We also give qualitative insight on how the algorithms function. We choose 1 motion planner from each subclass of graph search, incremental search and trajectory generation. While our intention is not to compare these planners with each other, we are interested in how dynamic deadlines impact different classes of planners internally and consequently came up with these 3 as candidates.

4.1 Hybrid A*

Hybrid A* [15] builds upon the popular A* search algorithm and is applied to the 3D kinematic state space of the vehicle. It associates the continuous vehicle state with the discretized nodes in A*, guaranteeing kinematic feasibility of the path. It was used by the Stanford Racing Team in the Darpa Urban Challenge and demonstrated superior performance in a variety of urban driving tasks. Although Hybrid A* does not guarantee finding the minimal-cost solution, it is useful in robotic path planning as it produces kinematically feasible paths. Hybrid A* has advantages over other motion planners in situations where discretization errors become critical. For tight maneuvers in a parking lot, the discretization error in linear piecewise A* or linear piecewise RRT may cause collision checking to fail, because the paths are kinematically infeasible for the vehicle. Hybrid A* paths are kinematically feasible and hedge the potential failure of collision checking.

The original paper discretizes the search space into (x, y, θ) configurations, just as in traditional A*. However, their Hybrid A* variation associates a continuous 3D state of the vehicle with each grid cell. The search algorithm is guided by two heuristics, the non-holonomic-without-obstacles cost and the holonomic-with-obstacles cost. The first ignores

```
Function PLAN(m, μ, x_s, θ_s, G)
    Result: Path from x_s to G
    n_s ← (x̃_s, θ̃_s, x_s, 0, h(x_s, G), −)
    O ← {n_s}
    C ← ∅
    while O ≠ ∅ do
        n ← argmin_{n∈O} n_f
        O ← O \ {n}
        C ← C ∪ {n}
        if n_x ∈ G then
            return path from n
        else
            UPDATE(m, μ, O, C, n)
```

```
Function UPDATE(m, μ, O, C, n)
    Result: Explore neighbors of n using μ
    forall δ do
        n′ ← μ(n, δ)
        if n′ ∉ C then
            if COLLISION(n′, m) then
                C ← C ∪ {n′}
            else if ∃n″ ∈ O : n′_x̃ = n″_x̃ then
                n′_g ← g′
                if n′_g < n″_g then
                    replace n″ with n′
            else
                O ← O ∪ {n′}
```

Figure 4.1: Hybrid A* pseudocode. Some of the configurable variables in the code are μ (expansion function), completion criteria (set G), the map representation (m), δ (turning radius), the cost function, and the collision checking function . First, the search space is discretized such that each node has a continuous configuration range. The open set (unexplored set) is initialized with a starting position and associated costs. Then, until there are no more nodes to explore, we select the node with the minimum cost. We remove the node from the open set and add it to the closed set (explored set). If this node is in the goal set G, then we return the reconstructed path. Otherwise, we add viable neighbors to the open set using UPDATE. UPDATE explores new nodes by applying control inputs over a discretized range of possible inputs. Applying the control inputs returns a continuous state, which is associated with a node in the explored set. Association is determined by which node's configuration range the continuous state lies in. If the new continuous state cannot be associated with an existing node, a new node with the continuous state is added to the open set. If it can be associated with an existing node and the new continuous state would have lower cost than the current associated state, we replace the current associated state. If there is a collision when applying the control, the node is added to the closed set. Otherwise, we add the neighbor to the open set.

obstacles but accounts for the non-holonomic nature of the car. It effectively prunes search branches that approach the goal with the wrong heading. The second heuristic accounts for obstacles but assumes a holonomic car. It provides a shortest distance to the goal to guide Hybrid A* search towards a goal. The original paper also calculates the Reeds-Shepp path from a node selected with probability $\frac{1}{N}$ to the end. N decreases as a function of the cost-to-goal heuristic. This improves both the performance and run-time of the planner.

Finally, they use a Voronoi field to define a cost function that provides trade-offs between path length and proximity to obstacles.

Our implementation assumes the following: μ is a function $\mu(p, \delta)$ where p is the current pose of the vehicle and δ is the turning radius of the Dubin's path. Thus all of our exploration is conducted by Dubin's paths with various turning radii. This can easily be adapted to use Reeds-Shepp's paths in case we want to be able to reverse the vehicle. Our completion threshold is the distance between the 2D position of the vehicle and the goal 2D position, and the difference threshold between the final yaw and target yaw. Our heuristic cost function is a combination of the straight line distance to the goal location and the total arc length distance traveled from the start location. Any paths resulting in collision are immediately pruned.

Our implementation is in C++ and has similar run-time compared to the original paper, although in certain cases we see much higher variability. For example, many simple cases involving a few obstacles along a straight path are easily solved in under 20 ms. A simple case may be driving between two staggered cars. However when we need to make a turn around an obstacle, our run-time increases to over 1 second. These run-time statistics are highly dependent on the exact scenario and hyperparameters we have chosen. We attribute these discrepancies to the differences in our implementation, and that of the original paper. We also note that without a behavioral layer to moderate the Hybrid A* search, the plans produced by each planning cycle can be highly variable due to the search space discretization. In the original paper, most of the encountered scenarios involved only static obstacles. This means that frequent replanning was unnecessary and a behavioral layer could easily moderate the behavior of Hybrid A*. But in our dynamic scenarios, replanning every cycle is crucial as new information is provided by perception and prediction. As mentioned in 3, slight perturbations in the starting pose of the vehicle can yield vastly different motion plans due to discretization. Thus, as the vehicle moves slightly from planning cycle to planning cycle, the resulting paths can change frequently causing undesirable behavior in the motion planner. We also note that using Dubin's path to analytically expand the search space can lead to sub-optimal trajectories. Paths between nodes are Dubin's paths, and although kinematically feasible, can result in unnecessary turning. This leads to jerky steering to track the motion plans. We hypothesize that more rigorous hyper parameter tuning is necessary to produce more comfortable, consistent paths. These claims are substantiated in section 6.

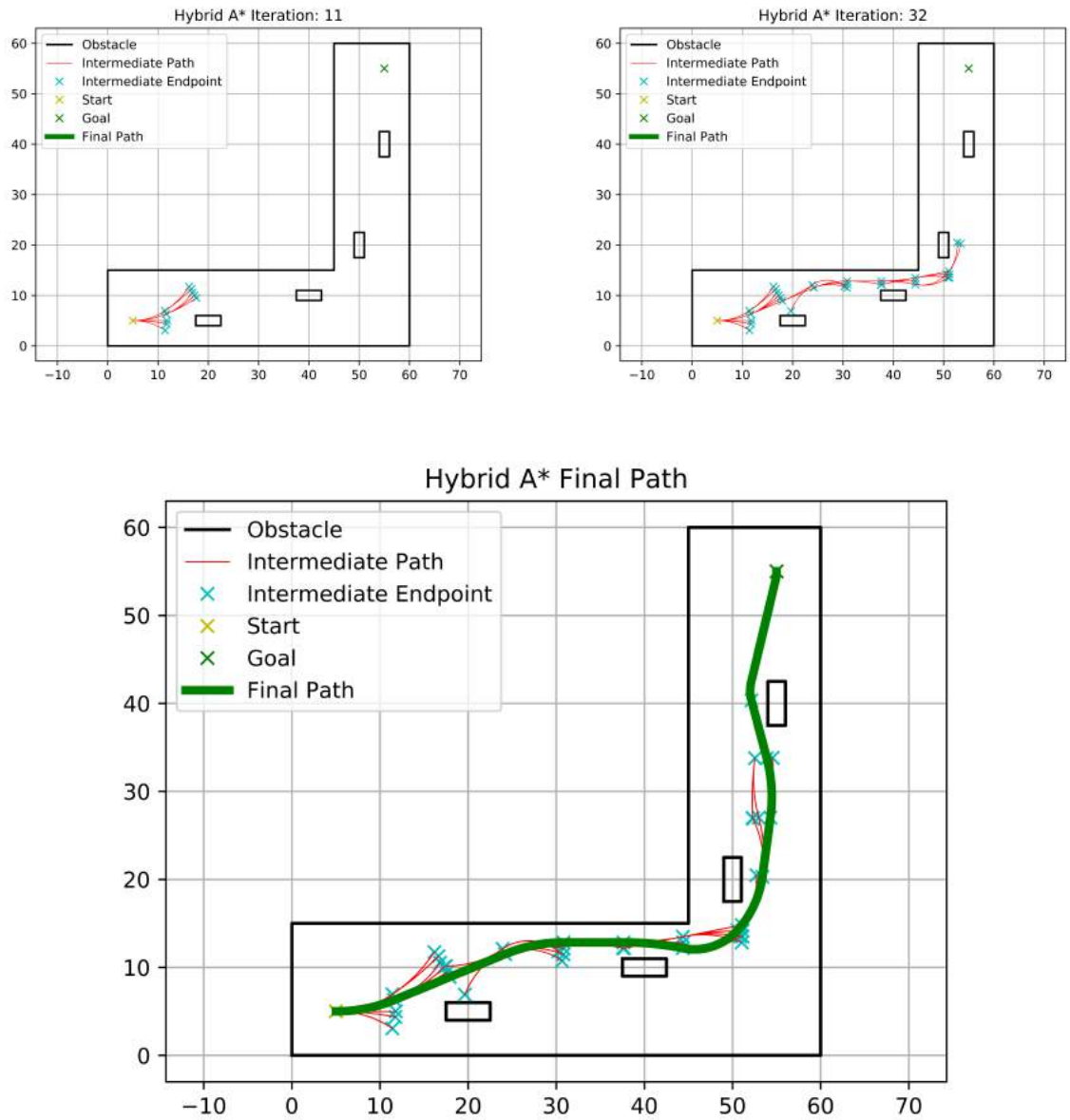

Figure 4.2: Hybrid A* visualization. Here we visualize the Hybrid A* algorithm at various iterations of planning. We see how Hybrid A* search creates a branching pattern from nodes in the graph. The branching pattern is determined by the control input discretization. Here our discretization range is 15 degrees with a grid resolution of 0.1 m. We apply control inputs for 5 meters to explore neighboring nodes. As indicated in the legend, the cyan crosses represent intermediate endpoints while red lines represent paths between the endpoints. Finally, the minimum cost path is selected once we reach the goal. Code is available **here**.

We visualize our Hybrid A* planning algorithm at different iterations in figure 4.2. We note how expansion happens from the intermediate endpoints using various curvature Dubin's paths. Here our collision checker operates on a 0.1 m motion resolution and defines collision as intersection of the bounding box of the car with an obstacle bounding box. In this tight maneuvering scenario, Hybrid A* finds a reasonable path through the static environment.

4.2 RRT*

RRT* builds upon the popular RRT search algorithm. While RRT is probabilistically complete, it is asymptotically suboptimal. RRT* is an adaptation of RRT that is asymptotically optimal. RRT* extends RRT with the following additions: RRT* gradually increases the resolution of search utilizing a hypersphere for connecting sampled vertices. The hypersphere's radius is a function of the size of the tree and dimension of the search space. It also rewires any neighboring vertices if doing so would produce a lower cost path. Finally, it also associates a cost with each node, drawing inspiration from the popular A* search algorithm.

RRT* is very efficient in terms of run-time and memory usage and, in our C++ implementation, is usually the fastest planner in most scenarios and configurations. It also has the benefit of being able to find an initial solution quickly, and then iteratively refining the solution by rewiring the tree until it is interrupted. This is very beneficial for autonomous vehicle motion planning, as time deadlines vary with each cycle. Although it quickly finds a solution, RRT* still suffers from discretization error as a linear, piecewise planner. The path to and from each vertex is linear, resulting in kinematic infeasibility for our nonholonomic vehicle. A workaround for this is to use Dubin's or Reeds-Shepp's paths between each vertex. This improves the kinematic feasibility of inter-vertex paths produced by RRT*, but also complicates collision checking. It also does not guarantee the kinematic feasibility of the global path, as there can be discontinuities at the vertices. Naive RRT* implementations also result in high variance in paths between planning cycles. If the tree from the previous planning cycle is not used to initialize the search space in the current planning cycle, the random factor in RRT* will construct a completely different tree than the previous cycle. This causes the paths to rapidly change with each planning cycle, resulting in undesirable behavior. While maintaining this tree is not difficult, choosing a proper discretization and knowing when to update the tree can be challenging. Because the vehicle operates in a continuous space, the current position of the vehicle will likely not be reflected in previous versions of the tree. Implementations that maintain the tree from cycle to cycle need to properly handle this situation, as well prune branches of the tree that need to be updated due to dynamic obstacles. Ultimately, although this introduces significant run-time variability to the RRT* planner, maintaining the tree is necessary for production motion planners in order to avoid the random replanning.

Our implementation closely follows the original paper and does not maintain the tree between cycles. One difference is that at every iteration, we sample the goal endpoint with

```
Function PLAN(x_init, K, Δt)
  Result: Tree T initialized at x_init
  T.init(x_init)
  for k = 1 to K do
    r = γ ∛(log|T| / |T|)
    x_rand ← RANDOM_STATE()
    x_near ← NEIGHBORS(x_rand, T, r)
    x_par ←
     argmin x_{x∈x_near} c(x) + c(x, x_rand)
    u ←
     SELECT_INPUT(x_rand, x_par)
    x_new ← NEW_STATE(x_par, u, Δt)
    T.add_vertex(x_new)
    T.add_edge(x_par, x_new, u)
    REWIRE(T, x_near, x_new)
  return T
```

```
Function REWIRE(T, x_near, x_new)
  Result: Rewire T
  for x ∈ x_near do
    u ← SELECT_INPUT(x_new, x)
    x' ← NEW_STATE(x_new, u, Δt)
    if c(x_new) + c(x') < c(x) then
      T.add_vertex(x')
      T.add_edge(x_new, u, Δt)
```

Figure 4.3: RRT* pseudocode. For each iteration, RRT* samples a new vertex in the configuration space. It then connects the new vertex to the neighbor that minimizes the path cost from x neighbor to x new. Neighbors are considered using the aforementioned hypersphere. Then a control input is computed and the path is validated for collisions. Any vertex in the vicinity of the new vertex is rewired if the cost to use the new vertex would be lower. Rewiring results in paths that are more direct than RRT paths and is necessary for optimal convergence. The radius of the Nd ball is defined as a function of the size of the tree $\gamma\sqrt[d]{\frac{log|T|}{|T|}}$ and the dimensions of the search space. This is necessary to ensure that the algorithm asymptotically converges to an optimal path. The pseudocode is similar to RRT, with the primary difference being NEIGHBORS accepting the hypersphere radius as a parameter and we REWIRE at the end of each iteration. SELECT_INPUT and NEW_STATE are the same as in RRT.

probability 0.01. This encourages connection with the goal, improving the rate at which RRT* makes progress and ultimately run-time considerably. 0.01 was selected empirically, as there is a trade-off between exploration of neighboring spaces and connecting to the goal. This probability can also be a function that increases with the number of iterations, encouraging exploration early on in the algorithm and completion later on.

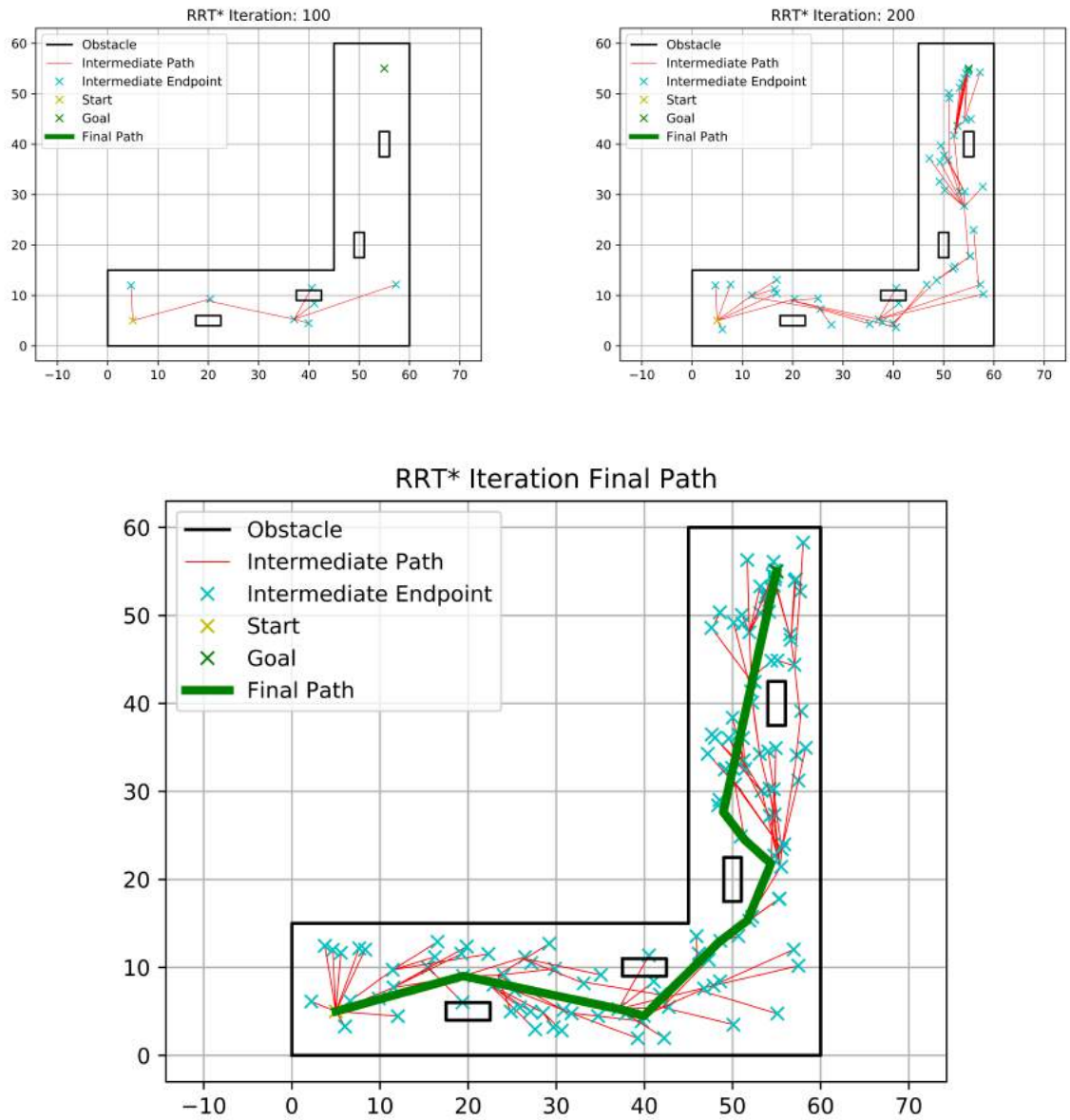

Figure 4.4: RRT* visualization. Here we visualize the RRT* algorithm at different iterations of planning. First, we note that at iteration 100, there is a path that collides with a vehicle. This is due to the coarse discretization of the search space. Between iteration 200 and the final graph, numerous connections are rewired for optimal cost. This rewiring, in conjunction with the hypersphere connection process, tends to produce very direct paths, relative to RRT. These paths are kinematically infeasible for our nonholonomic car, unless we modify the steering function. The base steering function is just the straight line path between vertices for a holonomic robot. Code is available **here**.

We visualize our RRT* implementation in figure 4.4. The paths produced are kinematically infeasible for a nonholonomic vehicle. The collision check is an obstacle intersection check with a padded version of the path. As the number of iterations increase, the initial paths are refined as the tree is rewired. The run-time, although consistently low, can have relatively high variance due to the randomness. In the scenario below we run for 500 iterations and empirically RRT* finds solutions in less than 100 iterations or fails in all 500 iterations.

4.3 Frenet Optimal Trajectory

The original Frenet Optimal Trajectory was proposed by Werling et al. 2010. We use Xu. et al. 2015 adaptation of the Frenet Optimal Trajectory as our reference implementation. Their algorithm consists of trajectory planning and trajectory optimization. Trajectory optimization improves the trajectory by optimizing it with respect to some cost function and set of constraints. Our implementation omits the trajectory optimization step and thus we will not cover it here.

Trajectory planning is broken into path generation and speed generation. Path generation is done by connecting sampled endpoints using quartic curvature polynomials. These guarantee continuous change of curvature rate between planning cycles. Endpoint sampling is done in the Frenet frame, described in section 3. The quartic polynomials are parametrized by x, y, velocity and orientation difference from the start point to the end point, start point curvature and end point curvature. Speed generation is done by first discretizing the speed space, and then using a cubic polynomial parametrized by position, velocity and acceleration. Speeds are discretized linearly given a minimum speed, maximum speed and number of discretizations. The 2015 adaptation uses a variety of costs which differ under static and dynamic situations. Their dynamic costs consist of a blend of efficiency, energy, comfort and safety proxies. They use time duration of trajectory for efficiency, sum of squared velocities for energy, sum of squared acceleration, jerk and centripetal acceleration for comfort, and sum of distance to dynamic obstacles for safety. These costs are weighted and summed to produce a weighted sum cost for each trajectory. Finally, the trajectories are sorted by their weighted sum cost and the lowest cost trajectory is selected. Each trajectory originates from a unique set of the discretization parameters, such as road width sampling distance, maximum time horizon, end velocity sampling range, and so on. These trajectories are also verified against physical constraints such as maximum acceleration or curvature.

Metric	Proxies
time	Efficiency
$\sum_i^n v_i^2$	Energy
$\sum_i^n a_i^2; \sum_i^n j_i^2; \sum_i^n c_i^2$	Comfort
$\sum_i^n \sum_j^m d(o_j, p_i)$	Safety

Figure 4.5: Frenet Optimal Trajectory costs as per [74]. v_i, a_i, j_i are the velocity, acceleration and jerk at time i. c_i is the centripetal force measured from the center of gravity. o_j is an obstacle location. p_i is a pose along the trajectory at a time i. d is the Euclidean distance.

In pseudo-code, the naive implementation is essentially a series of nested for-loops that iterate over the discretization parameters. Each inner loop iteration produces a trajectory that is validated for constraints and assigned a cost. The trajectory is produced as described above. Our C++ implementation has a few differences. For trajectory generation, we use quintic polynomials parametrized by both lateral and longitudinal position, velocity and acceleration in the Frenet frame. For speed generation we use a quartic polynomial parametrized by the longitudinal position, velocity and acceleration, and lateral velocity and acceleration. These high degree polynomials help to ensure continuity in velocity and acceleration in addition to curvature and position. The trajectory and speed profiles are combined to produce a motion plan, which is first validated for collisions and physical constraints and then assigned a cost. Our costs are the accumulated lateral deviation from the lane center, the accumulated lateral velocity, acceleration and jerk, the longitudinal acceleration and jerk, the end speed deviation, the total time taken, and the inverse distance to the closest obstacle. These costs approximate safety, both experienced and perceived comfort, efficiency and energy.

Metric	Proxies
time	Efficiency
$\lvert v_n - v_t \rvert$	Energy
$\sum_i^n a_i^2; \sum_i^n j_i^2 2$	Comfort
$\sum_i^n d(p_i, c_i)$	Perceived Safety
$\sum_i^n \frac{1}{min_j d(o_j, p_i)}$	Safety

Figure 4.6: Frenet Optimal Trajectory costs for our implementation. v_n is the final velocity at the end of the Frenet trajectory, while v_t is the target velocity. c_i is the center lane position at some time i along the target trajectory. The rest of the variables are the same as those in figure 4.5.

In our experiments, the Frenet Optimal Trajectory produces the qualitatively and quantitatively best trajectories in terms of jerk, acceleration and curvature. This is likely because

Frenet Optimal Trajectory accounts for speed profiles when generating a trajectory, whereas RRT* and Hybrid A* are both path planners. Accounting for speed improves collision checking granularity, kinematic feasibility and timing in safety critical scenarios. Ultimately, this leads to better qualitative and quantitative performance. In terms of run-time, it is more consistent than RRT* or Hybrid A*, as the run-time is just dependent on the hyperparameter configuration and the number of obstacles. RRT* and Hybrid A* run-times also depend on the world state, which increases run-time variability. This can further be optimized by simply parallelizing the trajectory and speed profile generation. However, Frenet also suffers from search space discretization. Some coarse discretizations work particularly well in difficult scenarios but fail in simpler scenarios. This is likely to due to overfitting the hyperparameters to the particular scenario. Choosing a proper discretization is an important part in tuning this planner. We utilize grid search on a variety of scenarios to determine appropriate discretization parameters for each of the motion planners.

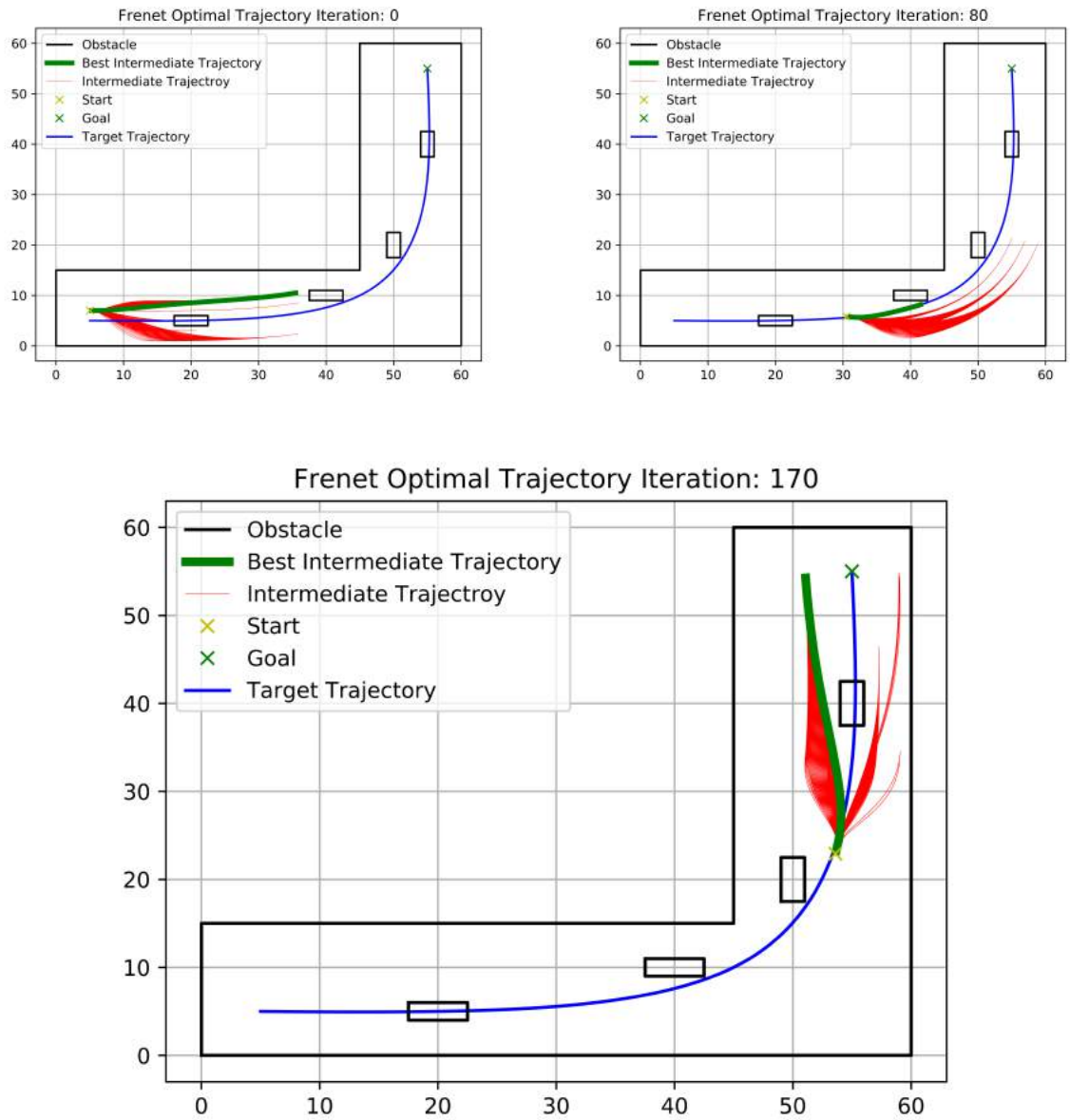

Figure 4.7: Frenet Optimal Trajectory visualization. Here we visualize the Frenet Optimal Trajectory algorithms at various iterations of planning. Each of the thin red lines is a candidate Frenet trajectory. Out of these valid candidate trajectories, the optimal trajectory is selected with respect to our cost functions. The differences in these trajectories reflect different discretization parameters, such as lateral offset and target velocity. Because Frenet Optimal Trajectory does not necessarily compute a path to the end of the trajectory, we update the world state by setting the new configuration to the first configuration in the best candidate trajectory. This moves the simulated vehicle forward and produces a different set of candidate trajectories at each iteration. Code is available **here**.

We visualize our Frenet Optimal Trajectory implementation in figure 4.7. Again, Frenet Optimal Trajectory produces multiple candidate trajectories denoted in red below. Then based on the cost of each path, it selects the minimal cost path to execute, denoted in green. A primary difference is that Frenet Optimal Trajectory relies on a target trajectory, rather than just a goal location. Usually this target trajectory will be the lane center, at a velocity under the speed limit. This has huge benefits for autonomous vehicle motion planning. Tracking the center of a lane provides safety and perceived safety and is an easy heuristic to implement. It also allows better velocity control relative to path curvature constraints and vehicle dynamic constraints. Another difference is that the Frenet Optimal Trajectory does not need to produce a path to the end of the target trajectory. Instead, it is able to produce a path that follows the target trajectory, but not necessarily completes it. Again, this can be beneficial as it simplifies the complexity of the planning problem and provides immediate solutions that progress towards a long term goal.

Chapter 5

Experiment Setup

In this section, we elaborate on our experiment setup. We discuss how our autonomous vehicle stack is designed, what kind of metrics we gather from our experiments and what scenarios we test. Finally we show how to use these results to construct a dynamic deadline planner.

Our experiments address the trade offs in motion planning by utilizing ERDOS and Pylot as the frameworks for developing these algorithms. ERDOS is a streaming data-flow system designed for self-driving car pipelines and robotics applications. It is designed for low-latency throughput and provides determinism through time-ordered watermarks. This is crucial for ensuring integrity of our experiments. The Robot Operating System (ROS) [3], although widely used and with its own set of strengths, does not guarantee message ordering and determinism. Pylot is an autonomous vehicle platform for developing and testing autonomous vehicle components (e.g., perception, prediction, planning). It can be used on a real vehicle, or a vehicle simulated in the Carla simulator. The Carla simulator is designed to evaluate self-driving vehicles and provides physically realistic situations and environments. We utilize the Carla simulator's **scenario runner** to craft complex scenarios to evaluate our motion planning algorithms. The vehicle model we use is a Lincoln MKZ. To control the vehicle, we use a carefully-tuned, planner specific PID controller for steering and acceleration inputs. While decoupling the controls is possible in Carla as we can directly update the ego-vehicle state, control feasibility is a critical aspect of motion planning. We analyze 3 different motion planning algorithms described in section 4 and determine how they can be improved by dynamic deadlines.

Pylot components are implemented as ERDOS operators which are connected by data streams. The set of operators and streams forms the dataflow graph, the representation of the pipeline that ERDOS uses. Pylot and ERDOS provide a time-to-decision operator that is based on the current ego vehicle speed and location of obstacles. This time-to-decision operator outputs the maximum time available to make a decision, which other operators can listen to and adapt themselves accordingly. For example, if there are no obstacles in the environment then the operator can output a generous time-to-decision, but if a pedestrian suddenly jumps out onto the road then the time-to-decision will significantly reduce the

time to decision in order to avoid collision. Motion planning operators can change their configurations appropriately. In emergency cases, we would need to run a faster planner with a more coarsely discretized configuration in order to avoid collision. We discuss how to choose a planner based on time-to-decision in section 6 along with our results.

We abstract the upstream modules of sensing, perception and prediction. We are able to do this by accessing internal Carla states, thereby obtaining present and past information on dynamic obstacles. This information is the ground truth. In our experiments the behavior of non-ego agents is mostly linear, so we use a linear predictor to accurately forecast future poses. Abstracting these modules allows us to eliminate any confounding effects on dynamic deadlines in motion planning, although we will demonstrate our dynamic deadline pipeline running end-to-end, with all modules. Our route planner and behavioral planner are consistent across different motion planners. The route planner outputs the lane center of a straight, long road and the behavioral planner switches between driving straight and obstacle avoidance, toggled by the presence of an obstacle. While more sophisticated route and behavior planners are required in real world driving, these simple modules suffice for our experiments. In this way, we can also hold these quantities constant across the experiments for fair comparison.

Our implementations for Hybrid A*, RRT* and Frenet Optimal Trajectory are in C++ and exposed to Pylot via. a C++ / Python API. They are implemented as per descriptions in section 4. We note that the implementations are different from the original papers, so results here will vary by the exact implementation.

We divide our experiments into 2 categories: Synchronous Carla Experiments and Pseudo-Asynchronous Carla experiments. These experiments show motivation for using a dynamic deadline. Certain hyperparameter configurations are too costly to run or not fine enough in discretization and thus we need to use the time-to-decision operator to select an appropriate configuration for the situation.

5.1 Synchronous Carla Experiments

Synchronous Carla experiments are theoretical experiments that test the implementations in an idealistic way, provide visualization for understanding algorithm details and give insight to what would be important to test within Carla. They lay the groundwork and motivation for the pseudo-asynchronous experiments. We can run Carla synchronously, meaning the simulator waits for our stack to finish running before updating the simulation state. In this sense, the run-time of our modules is irrelevant, as the simulator ticks only when the control module finishes and sends a control command. Under this assumption, we can test the theoretical capabilities of each planner and each configuration by ignoring the run-time. This is advantageous for expensive planners and configurations that can operate over a high-resolution search space, because it leads to more optimal motion plans. In the pseudo-asynchronous experiments, run-time impacts real-world state configurations and therefore the expensive planners may perform worse than the coarse grained planners.

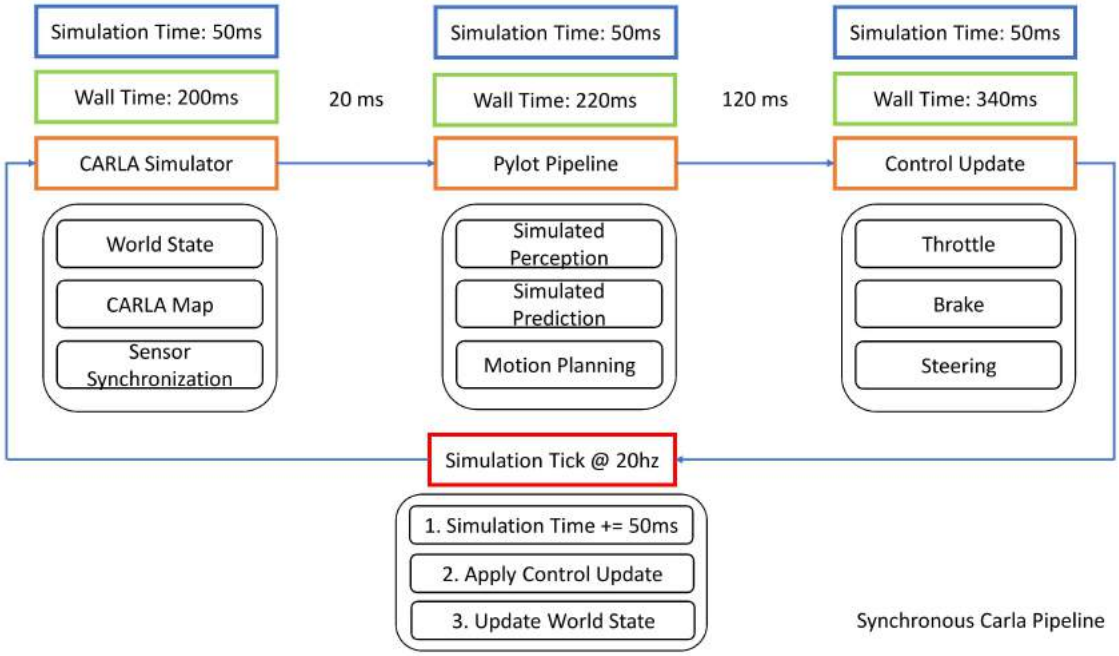

Figure 5.1: Synchronous Carla pipeline. Above we visualize the synchronous Carla Pipeline. The blue boxes at the top represent simulation time, or the internal elapsed time of the simulator. Below in green is the wall time, or the real world time. The Carla simulator passes world state, map information and synchronized sensor information to Pylot. Pylot runs simulated perception and prediction using Carla internal state information and produces a control output. The control output consists of throttle, brake and steering. Once the Carla simulator received the control update, the simulation is ticked forward. At 20 hz, simulation time is updated by 50 ms, the control is applied and the world state is updated for all actors. Note that the simulation is only ticked once the entire pipeline finishes.

5.2 Asynchronous Carla Experiments

While the synchronous Carla experiments test the theoretical capabilities of our planners, the pseudo asynchronous experiments verify the potential real world impacts of dynamic deadlines. First, we can't run Carla in a purely asynchronous mode because it has a variable frames per second (FPS) and can randomly release sensor data late, on the order of 100-200 ms. Instead, we run it synchronously at 200 FPS. This allows us to synchronize the sensors. After sensor synchronization, we measure the run-time of our pipeline. When our pipeline outputs a waypoints message from the planning module, we have the end-to-end run-time of just the pipeline. We apply the results from our pipeline at simulation time equal to the measured sensor time plus the real world run-time of our pipeline. In this way we get to see the effect of run-time on the world. We are also able to run controlled experiments in

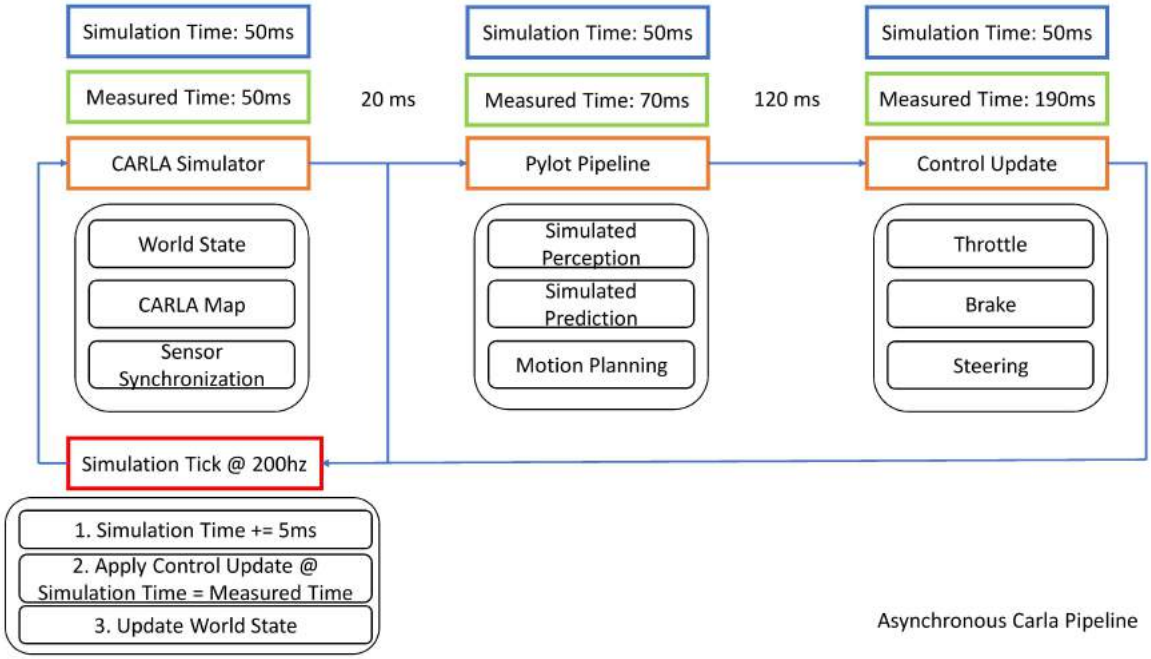

Figure 5.2: Asynchronous Carla pipeline. The asynchronous Carla pipeline differs in a few ways. First, we run at a much higher rate of 200 hz, meaning simulation time updates by 5 ms each tick. Second, instead of wall time, we use measured time. Finally, the simulator only applies received control at the corresponding measured times. Measured time is the time it takes for the Pylot pipeline to produce a control output based on when the Carla simulator outputs data. In the above hypothetical example, it takes a total of 140 ms to produce a control output. 20 ms is due to receiving the Carla data and 120 ms is due to running the Pylot pipeline. The time reference is 50 ms, thus the measured time is 190 ms for this control input. When the simulation time reaches 190 ms, this control input would be applied.

which the sensors are synchronized and we are not affected by Carla's variable FPS. In our configuration, we are able to control the frequency of our sensors as well as the frequency of the PID controller used to control the vehicle. Carla FPS controls the granularity of each simulator tick. At 200 FPS, the simulator is at most $\pm$ 5 ms in the terms of worst case run-time effects, which is acceptable for our experiments. Thus, we are able to run our experiments in pseudo-asynchronous modes to observe the real-world implications of dynamic deadlines.

5.3 Metrics

The responsibilities of motion planning are to produce a safe, comfortable and collision free trajectory. In our experiments, control is coupled with our planning modules so there is slight confounding in our metrics. Regardless, we measure a variety of metrics that approximate abstract terms like safety and comfort and include metrics on collision frequency. Comfort is highly subjective and difficult to measure, but we include metrics such as lateral and longitudinal acceleration and jerk to capture experienced comfort. These are measured using a simulated inertial measurement unit (IMU) mounted at the center of gravity of the vehicle. We also utilize perceptive comfort metrics such as accumulated deviation from lane center, total time spent in oncoming lanes and a timeline of distances to nearest object. These metrics approximate how one perceives the comfort and safety of the motion planner and are calculated using the true pose of the vehicle and obstacles. Collision metrics include a binary table of collision or no collision vs. discretized hyperparameters and collisions vs. target speeds. Collisions are determined using Carla internal collision checkers for accuracy. Finally, we also include run-time CDFs for each of the configurations and planners. These experiments were conducted using an i9-9900k, NVIDIA RTX 2080S and 32GB of ram. We found that our results were highly dependent on the hardware. Slower hardware exhibited worse performance on asynchronous cases, as run-time was slower. Details of these metrics are elaborated in section 6.

5.4 Scenario

Our motivating scenario is a pedestrian jumping out from behind a parked car, crossing the street. The ego-vehicle is driving at moderately high speeds of 16 m/s to 22 m/s on a street with two lanes, one in each direction. In the oncoming lane, there is a large red truck that is parked on the side of the road. As the ego-vehicle approaches the truck, a pedestrian runs across the street and stops on the road, near the sidewalk. This is triggered when the vehicle is 50 meters from the pedestrian. The pedestrian moves at 3.5 m/s for 6.5 m, and ends up in the middle of the road near the sidewalk. The truck, which is roughly 2.2 meters wide, conceals the pedestrian for nearly 0.6 seconds. At 22 m/s, this leaves the motion planner with a little over a second and a half of safety critical planning to avoid collision. At the current vehicle speed, emergency braking is not an option and the planner must devise a maneuver to swerve around the pedestrian. The extremity here is that the pedestrian also stops in the road, instead of moving to the other side. This forces the vehicle to take action and not just apply braking, even at 16 m/s. We verified that it is possible to navigate this scenario at speeds up to 24 m/s using manually engineered trajectories. For all experiments, the vehicle reaches within 0.5 m/s of the target speed.

This is an extreme scenario that rarely occurs and does not reflect the majority of driving situations. Yet under these challenging circumstances, the impact of dynamic deadlines becomes evident. Being able to select a motion planner that is fast enough to react yet high

Figure 5.3: Person Behind Car scenario. The ego-vehicle must succesfully navigate between a tight space formed by the red truck and pedestrian.

resolution enough to find a feasible trajectory is crucial in these situations. Static deadline systems do not allow for this, while dynamic deadline systems do. We elaborate on these experiments, metrics and scenarios in our results, section 6. We will walk through our results with the Frenet Optimal Trajectory planner and provide results on RRT* and Hybrid A* after.

Chapter 6

Results

We provide quantitative and qualitative results from our experiments. First, we walk through all metrics measured using the Frenet Optimal trajectory Planner and analyze the implications of the results. Then we succinctly discuss results for RRT* and Hybrid A*, highlighting any interesting trends in our metrics. Finally, we show how we used our results to implement a dynamic deadline planner that utilizes the time-to-decision operator.

Below we visualize our metrics for the Frenet Optimal Trajectory planner on the pedestrian behind car scenario at speeds of 16, 18, 20, 22 m/s. Out of the tunable hyperparameters, the most significant ones were the road width discretization and the time step discretization. Road width discretization controls the sampling distance between lateral offsets for lateral trajectory generation, while time step discretization determines the temporal spacing between waypoints for collision checking. We varied the road width discretization between 0.1 meters and 1.0 meters spaced 0.1 meters apart, and varied the time step sampling between 0.1 and 0.3 seconds spaced 0.05 seconds apart. All other hyperparameters were fixed between experiments. We run these experiments as a grid search, iterating over every combination of road width discretization and time step discretization. To keep our figures concise and interpretable, we choose to analyze 4 of the possible discretizations for most metrics.

6.1 Runtime

We visualize the run-time CDFs of 4 possible discretizations in figure 6.1. These CDFs provide insight when used in conjunction with other metrics described in this section. We see that finer resolution discretizations have higher run-times, while coarser resolution discretization have lower run-times. This trade-off in resolution and run-time is crucial in the following sections, especially for the asynchronous experiments. Using the 99th percentile of run-times, we are able to construct a run-time to planner configuration mapping that allows the motion planner to effectively utilize the time-to-decision operator. Based on how much time the motion planner has to make a decision, it can dynamically choose what motion planning configuration to use. In emergency scenarios with little time to react, we expect

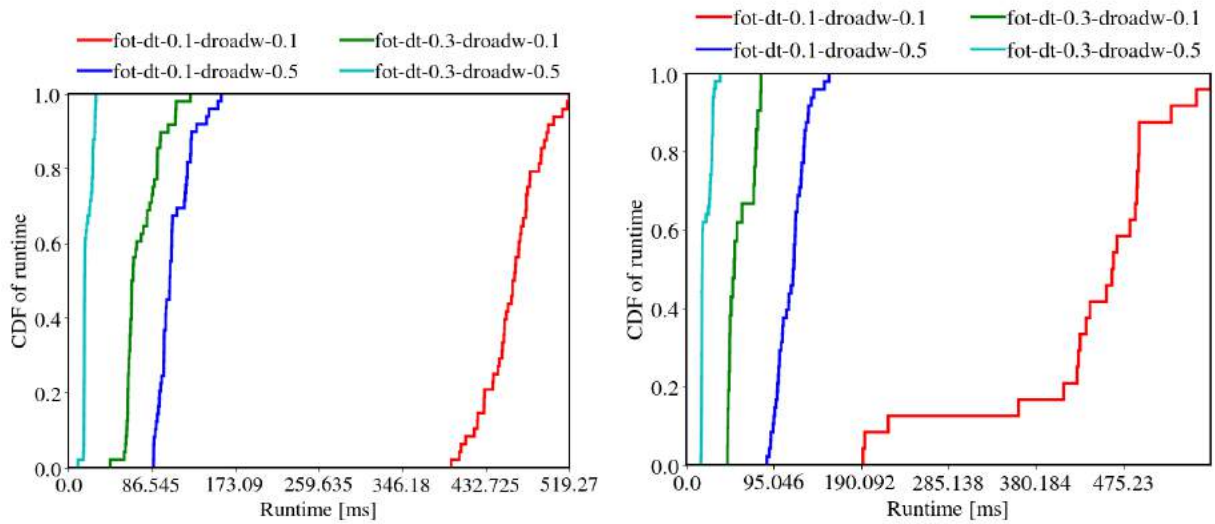

Figure 6.1: Frenet Optimal Trajectory run-time CDF. Synchronous CDFs are on the left and asynchronous CDFs are on the right.

that a compromise between run-time and resolution is necessary to successfully navigate the scenario. We further analyze these claims in section 6.9.

6.2 Collision Table

The collision tables in figures 6.2, 6.3 visualize whether the vehicle successfully navigated the scenario given a particular configuration of road width discretization, time discretization and target speed. These collision tables give us an approximate understanding of how the planner's hyperparameters impact the resulting motion plan and proxies the safety of the motion planner. Generally, finer discretization is superior as it is less susceptible to discretization error. However, finer discretization also increases run-time, which impacts the reaction time of the vehicle and thus its ability to navigate the scenario. The synchronous case serves as a reference when analyzing the asynchronous case. If the synchronous case was able to successfully navigate the scenario, then the hyperparameter discretization is not a limiting factor in determining the success of the planner. For example, at target speed 16 m/s and both discretization parameters set to 0.1, the synchronous case passes yet the asynchronous case collides with the truck. This allows us to infer that it was likely not the discretization parameter that could not generate a collision-free path, but it was the run-time of the planner that resulted in a collision. The inverse of this is generally not true; acknowledging the influence of run-time generally hurts motion planning but in some cases can unintentionally help.

While there are anomalies in the trends, a balance between run-time and resolution per-

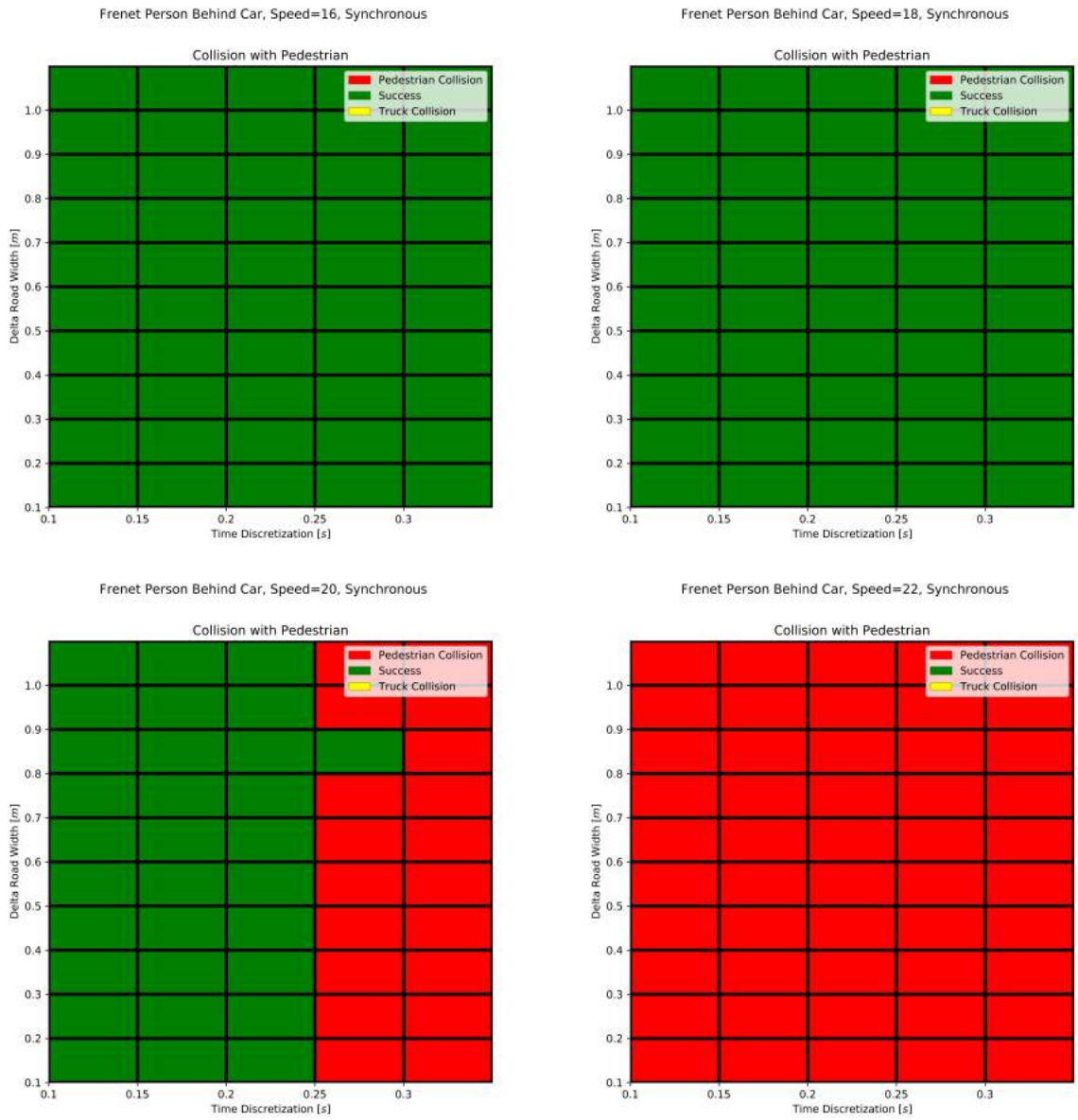

Figure 6.2: Frenet Optimal Trajectory collision tables for synchronous cases.

sists through our experiments, across all planners. In the synchronous case, finer resolution planners are generally more successful than the coarser resolution planners. But for the asynchronous case, we see that there is a real trade-off between the resolution and success. The higher resolution planners are too slow, as indicated by the CDF plots in figure 6.1. A planning time of over 200 ms is considered unacceptable, as the state of the world can change rapidly at high speeds. Consider driving at 22 m/s towards an oncoming vehicle that is also

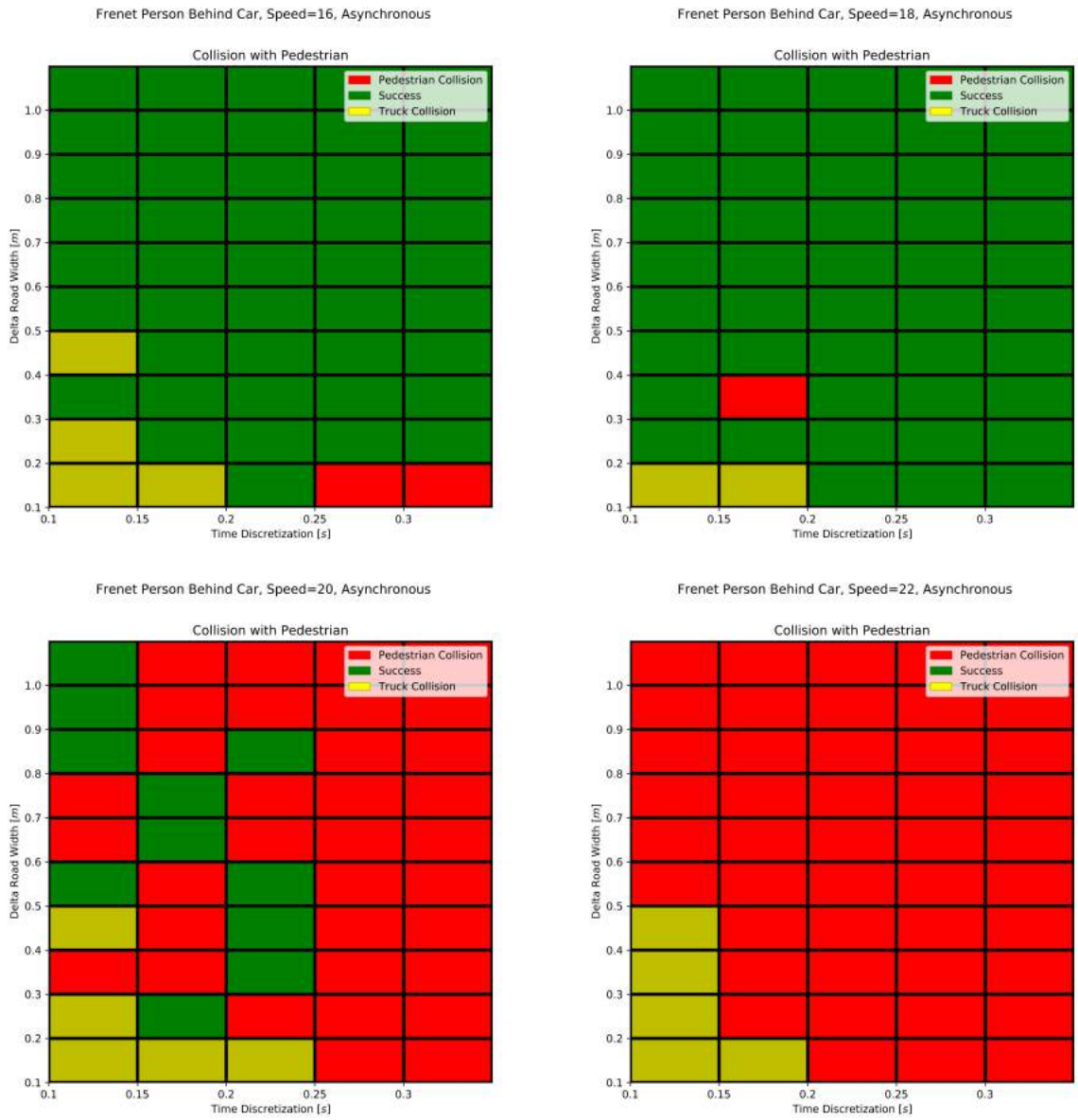

Figure 6.3: Frenet Optimal Trajectory collision tables for asynchronous cases.

moving at 22 m/s. In 200 ms, there is nearly a 9 meter discrepancy between the past location of the vehicle and current location of the vehicle. For this reason, planners that succeeded in both synchronous and asynchronous cases generally were balanced between discretization resolution and run-time. Evidently, being able to dynamically configure the planner based on the time-to-decision could drastically improve the safety of motion planning.

In some asynchronous cases, the motion planner was able to achieve a better result than

the synchronous case. For target speed of 22 m/s, the Frenet Optimal Trajectory planner collides with the pedestrian in all hyperparameter discretizations for the synchronous case. However, in the asynchronous case, the planner collides with a truck for a small subset of the configurations. It is challenging to say what causes this discrepancy, but it is likely due to overfitting of the hyperparameter configuration to the particular scenario. Thus these trends should be analyzed cautiously, as there exists a certain amount of overfitting for every planner and scenario. More experiments in different scenarios would be needed to decrease motion planning variance.

For Frenet Optimal Trajectory, we suspect that some other discretization parameters need to be adjusted. At a target speed of 22 m/s, Frenet Optimal Trajectory fails to produce a motion plan. Further analysis indicates that at that speed, it fails to find a trajectory around the pedestrian and results in emergency braking until collision. Parameters such as the target speed sampling range could be increased. This parameter allows for a range of target speeds and improves the flexibility of the planner in producing viable speed profiles. Obstacle clearance for prediction could also be tuned; many of the proposed Frenet trajectories were invalidated by obstacle collision checking, suggesting our obstacle clearance threshold was too restrictive. This could also be related to how we configure our linear predictor, in which we use 4 seconds of linear prediction for the pedestrian. Given that the pedestrian moves in a straight line, this effectively creates a wall of obstacles for which the Frenet planner has no solution. Ultimately, the Frenet Optimal Trajectory is more susceptible to the discretization parameters, as trajectories are entirely predetermined by the discretization. We will see in section 6.7 that RRT* handles this high speed much better, as it is not limited by construction.

6.3 Lateral and Longitudinal Metrics

The lateral and longitudinal acceleration and jerk plots visualize acceleration and jerk over the course of the simulation. Here simulation time is defined as the elapsed time in the simulation as per the simulation update frequency. We truncate the plots to only show the safety critical section of the experiment. These metrics proxy the experienced comfort of the motion plan. Lower magnitudes of acceleration and jerk are better.

Generally, the coarser resolution configurations yield trajectories with higher lateral jerk. This is expected as coarser resolution configurations will produce trajectories that deviate more from the lane center, thus increasing lateral jerk and acceleration. Longitudinal jerk is fairly consistent across configurations and is more dependent on the PID controller than the planner. We note that Frenet Optimal Trajectory significantly outperforms Hybrid A* and RRT* in these metrics, as the lateral acceleration and jerk are encoded in the cost function. Furthermore, it utilizes quartic polynomials for lateral trajectory generation and speed profile generation, improving continuity across planning cycles. As expected, planners that account for kinematic and dynamic feasibility exhibited better lateral and longitudinal measurements and thus a more comfortable ride. Again, there is a substantial trade-off in

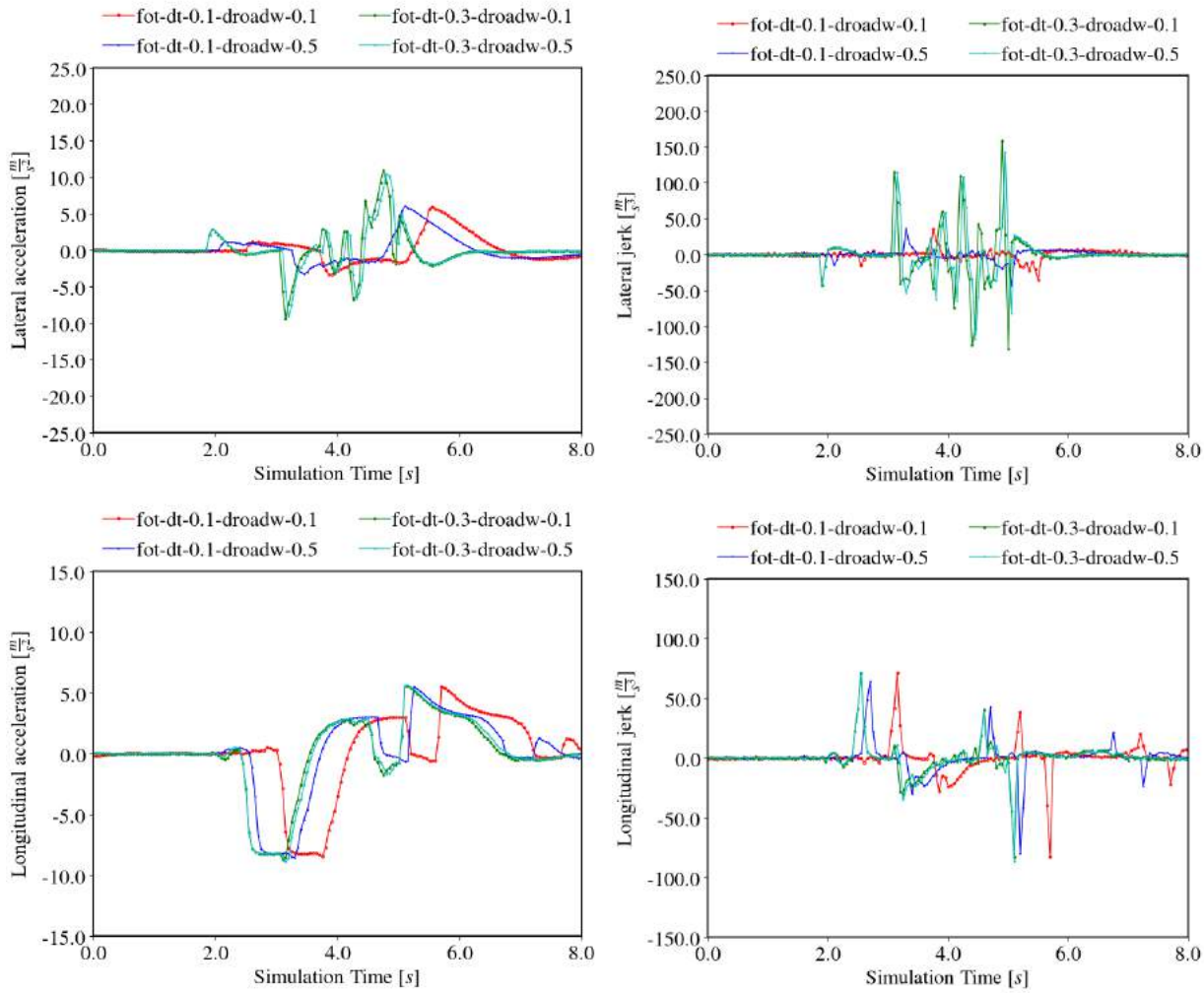

Figure 6.4: Frenet Optimal Trajectory acceleration and jerk measurements for synchronous cases.

comfort and resolution in this extreme scenario. Finding a collision free path is paramount, but lowering the risk of injury for passengers is also an important factor to consider. If there is an emergency situation that involves an avoidance maneuver, dynamically choosing the planner configuration to maximize success and comfort could be beneficial.

The metrics for the asynchronous case are slightly worse than the synchronous case. The difference in maximum lateral jerk is roughly 50 m/s^3 between the synchronous and asynchronous case. This is due to run-time delay and the resulting need to overcorrect for previously suboptimal plans. The run-time delay creates lag between the true simulator state and the perceived simulator state, thus each motion planner is planning for the past. As there were collision cases in some of these hyperparameter configurations, some of the

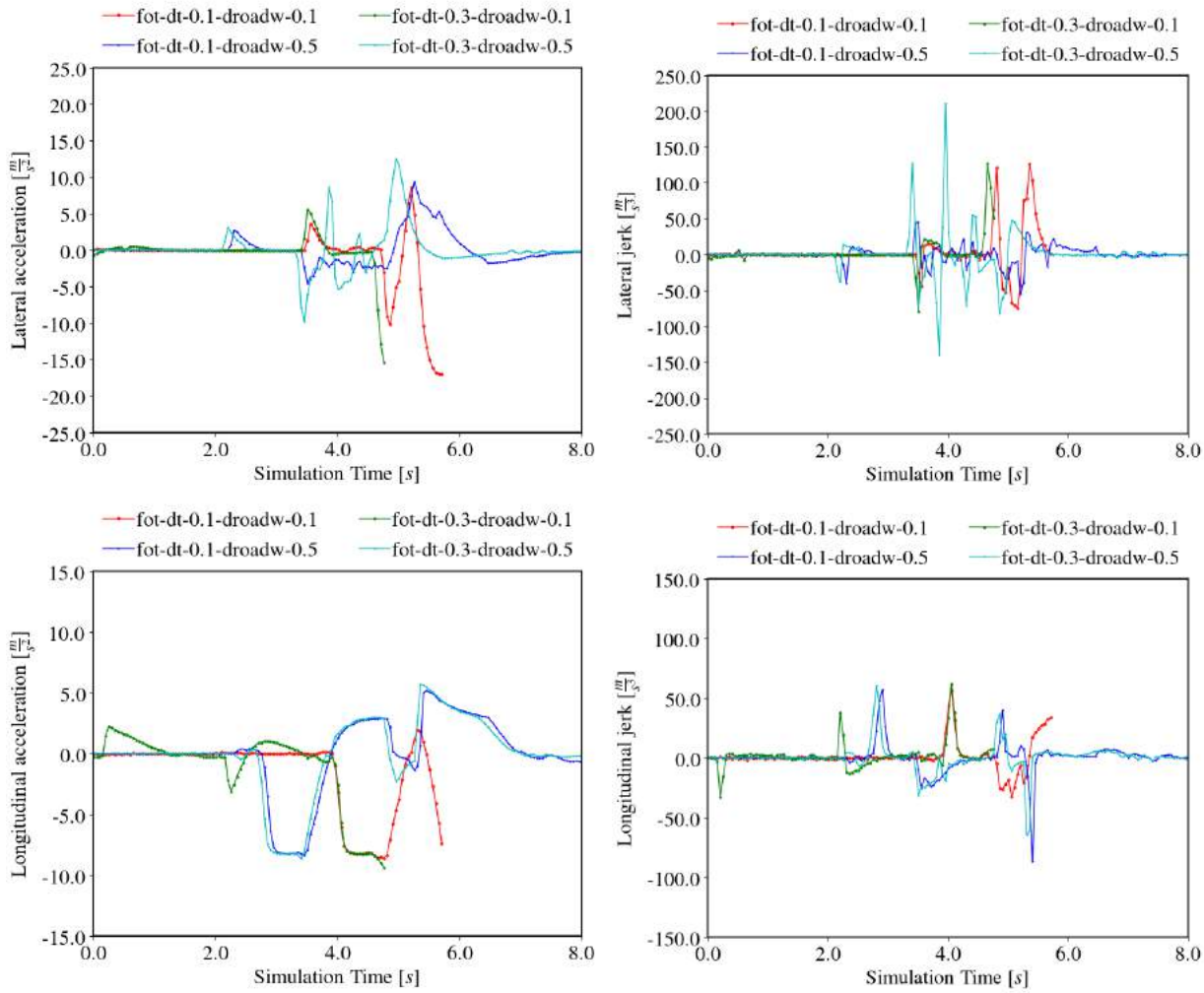

Figure 6.5: Frenet Optimal Trajectory acceleration and jerk measurements for asynchronous cases.

plots are truncated early in the asynchronous case.

6.4 Lane Center Deviation and Lane Invasion Time

Deviation from lane center and lane invasion time measure the safety and perceived safety of the vehicle. Generally, it is better to minimize the time spent in an oncoming lane and avoid unnecessary deviations from the lane center when possible. In figures 6.6, 6.7 we measure the total time spent in the oncoming lane and the cumulative deviation from the lane center. Time spent in the oncoming lane is measured as the center of the vehicle crossing the lane

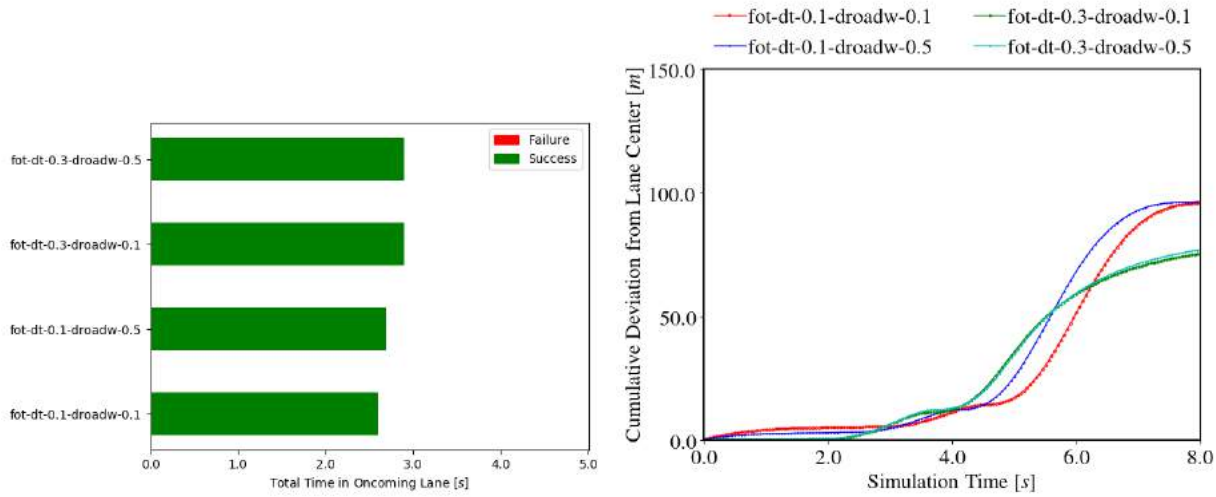

Figure 6.6: Frenet Optimal Trajectory lane invasion and deviation metrics for synchronous cases.

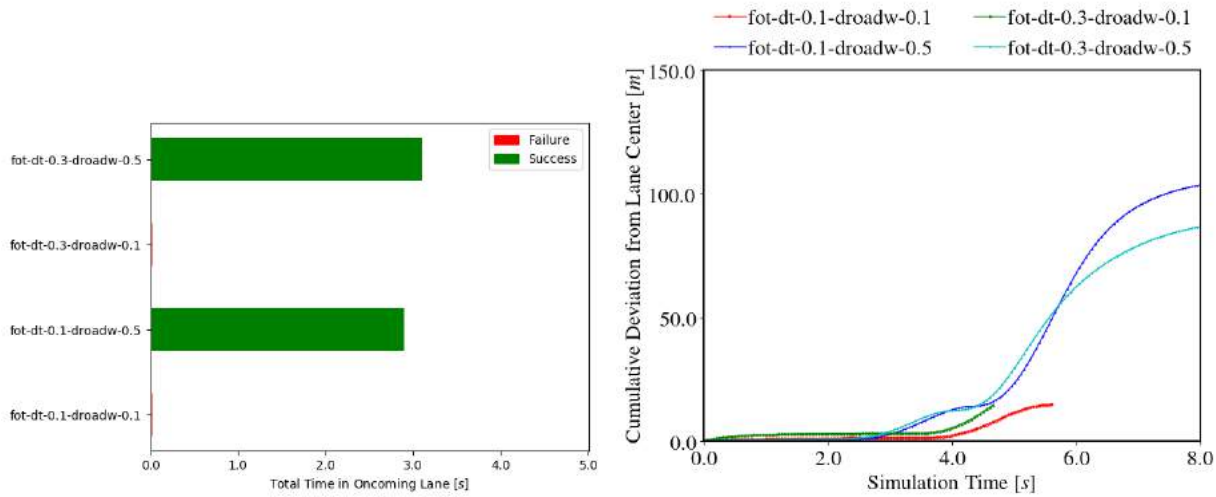

Figure 6.7: Frenet Optimal Trajectory lane invasion and deviation metrics for asynchronous cases.

boundary and cumulative deviation is measured at each waypoint in the motion plan. The number of waypoints is consistent across motion planners, thus this comparison is fair.

Again, the asynchronous case has truncated or missing data due to failure of the planner. Overall the Frenet Optimal Trajectory planner is fairly consistent in these metrics relative to Hybrid A* and RRT*. Most of the accumulation in lane deviation occurs from simulation time t=2s to t=6s, the approximate safety critical portion of the scenario. The finer reso-

lution planners spend less time in the oncoming lane and deviate less from the lane center. In this scenario, lane deviation and time spent in the oncoming lane are not crucial metrics. Avoiding collision and minimizing lateral acceleration are the primary factors related to injury and comfort. Still, it is useful for comparing the characteristics between planning configurations and different motion planners.

6.5 Distance to Obstacles

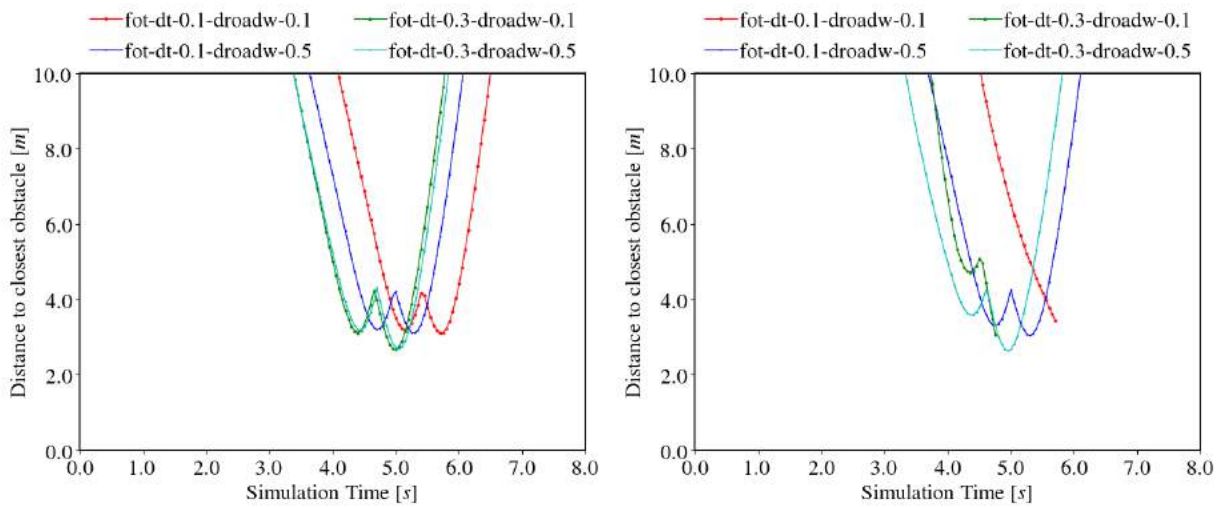

Figure 6.8: Frenet Optimal Trajectory obstacle proximity metrics. Synchronous is on the left, asynchronous is on the right.

We plot the distance of the vehicle to the nearest obstacle as a function of the simulation time in figure 6.8. Distance to the nearest obstacle is calculated as the closest corner on the vehicle to the center of an obstacle. Computing the closest point on the vehicle to the closest point on the obstacle is computationally expensive, so this approximation is used. Thus, a collision can occur even if the distance is not 0. We see that the configuration of time discretization as 0.1 and road discretization as 0.1 results in a collision, but the distance is greater than 0. The distance to the nearest obstacle is a good proxy for safety and perceived safety as maximizing space between agents leaves room for error.

Frenet Optimal Trajectory gets within roughly 2.5 meters of the pedestrian or truck, while Hybrid A* and RRT* get within 2.0 meters. 2.5 meters is close to the optimal path, as there is less than 7 meters of space between the two closest points on the truck and pedestrian. These results are intuitive, as we encode the inverse distance to the nearest obstacle as part of the Frenet Optimal Trajectory cost. Results across the synchronous and asynchronous

case are consistent. Overall, this suggests that Frenet Optimal Trajectory produced safer trajectories at lower speeds, although it failed at higher speeds.

6.6 Qualitative Results

Figure 6.9: Visualization of the Frenet Optimal Trajectory planner in the Carla pedestrian behind car scenario.

We visualize the successful synchronous experiments for Frenet Optimal Trajectory in figure 6.9. The target speed is 16 m/s and each planner is configured to successfully navigate the scenario. Qualitatively, the trajectory is smoother than RRT* and Hybrid A* counterparts, discussed in 6.7 and 6.8. The trajectories look more feasible and exhibit human-like behaviors, such as maximizing distance between both cars, minimizing changes to steering, and smoothly adjusting the curvature. RRT* and Hybrid A* are notably less human-like, which shows in their lateral metrics and lane deviation metrics.

An aforementioned benefit of Frenet Optimal Trajectory is that it follows a trajectory, rather than a particular target. This is beneficial for motion planning as it allows the planner to output a trajectory that naturally tracks the curvature of a path. For RRT* and Hybrid A*, we had to fix a target waypoint that was a constant distance along the global path. Furthermore, the path RRT* or Hybrid A* takes does not innately consider

the geometry of the road. For Hybrid A* we found that limiting the target waypoint to be 20 meters ahead yielded the best results in terms of success rate. Increasing the distance to 30 meters ahead exponentially increased the number of searched nodes and search time, making the planner infeasible in run-time. For reference, state of the art vision systems have approximately 40 meters of accurate range. Planning for a longer horizon is important for motion planning, as actions in the present can drastically alter the outcome in the future. Choosing the target waypoint dynamically for Hybrid A* and RRT* is possible and could be related to the dynamic deadline planning framework. Still, relying on a target trajectory is a better solution as it eliminates the need to choose the target waypoint. For this reason, Frenet Optimal Trajectory better handled the horizon planning problem in this situation. We expect this result to hold in other situations as well.

6.7 RRT* Results

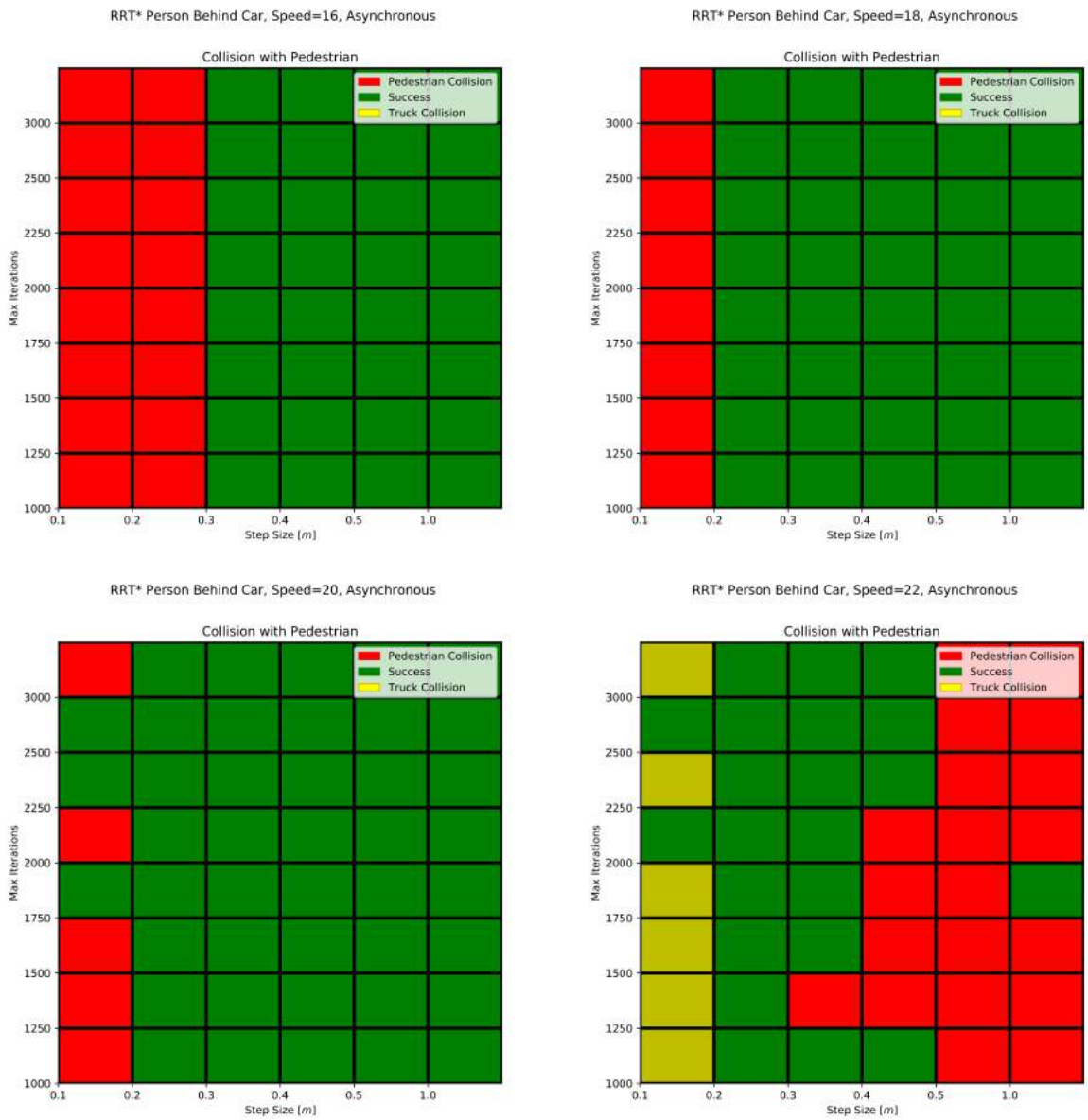

Figure 6.10: RRT* collision tables for asynchronous cases.

RRT* has the highest success rate in synchronous and asynchronous cases. It has a nearly 100 percent success rate for the synchronous case at all speeds. In figures 6.10, 6.11 we plot the asynchronous experiments for each target speed and the run-time CDF. Here, the trade-off between resolution and run-time is apparent. In many of the finer resolution configurations,

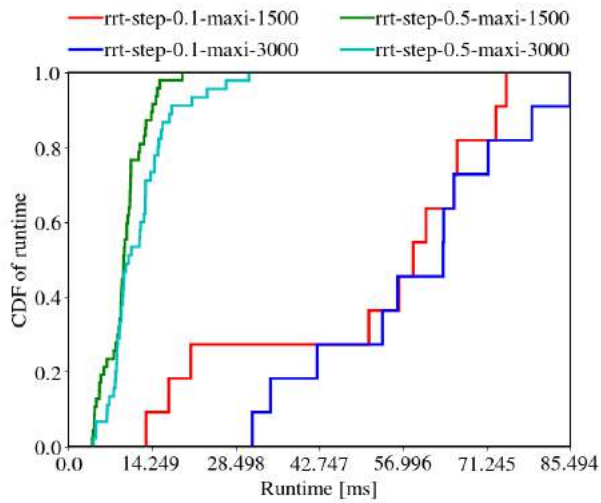

Figure 6.11: RRT* run-time CDFs for asynchronous cases.

the planner fails due to run-time. We can deduce that failure was likely attributed to run-time as the same configuration succeeded in the synchronous experiments. The finest resolution planner was generally too slow or too variable in run-time to reliably produce a motion plan, thus resulting in collision with the pedestrian, as shown in the CDF. At the target speed of 22 m/s, the trade-off between resolution and run-time is most pronounced. The finest resolution planner using step size of 0.1 m tends to collide with the truck, for the asynchronous case. Note that in the synchronous case, it successfully navigated the scenario. This collision is likely due to the increased run-time of the planner. Furthermore, the coarse resolution planners fail to find a safe path, although their run-time is lower than the fine resolution planners. These tables suggest that under these extreme scenarios, there is a fine balance between the run-time, resolution and success of a planner. Dynamic deadlines can help maintain this balance in different driving circumstances

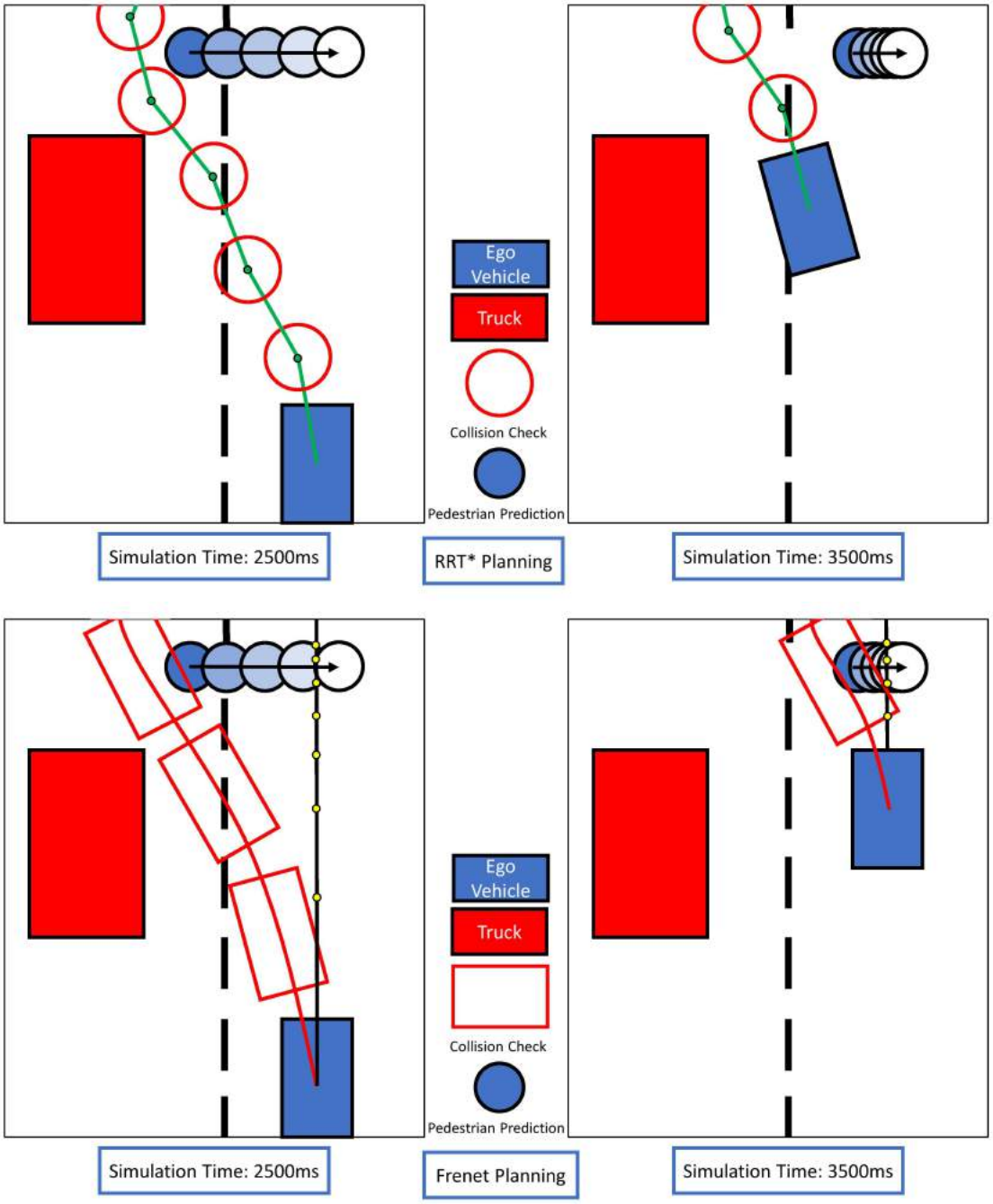

Figure 6.12: RRT* vs. Frenet Optimal Trajectory example. The green path represents a collision free path while the red path represents a path with collision. The red outlines represent the collision checking function. RRT* uses a radius around points along the path, while Frenet Optimal Trajectory uses a bounding box collision check. Green points represent the motion planner path plan without emergency braking. Yellow points represent the motion planner path plan with emergency braking. The red rectangle is the truck, blue rectangle is the ego-vehicle, and the blue circles are the pedestrian predictions.

The success rate of RRT* is likely related to its collision checking function and obstacle clearance parameters. Relative to the other planners, it has the coarsest collision checking and least restrictive obstacle clearance. In conjunction with the fast run-time, the lax collision checking allows RRT* to consistently produce path plans, even if those paths would be invalidated by stricter collision checking methods that utilize obstacle bounding boxes. This works in this scenario because the pedestrian continues to move across the street. Thus the earlier parts of the safety critical section guides the car in between the pedestrian and truck. By the time the pedestrian crosses the street, there is enough space for the vehicle to continue moving between the pedestrian and truck. The other planners use stricter collision checking that utilizes the bounding box of obstacles and the vehicle, which may result in failed planning cycles. This would produce an emergency braking command which does not guide the vehicle towards the optimal trajectory. By the time the pedestrian has fully cleared the feasible trajectory, it may be too late for other planners to react. This is visualized in the figure 6.12.

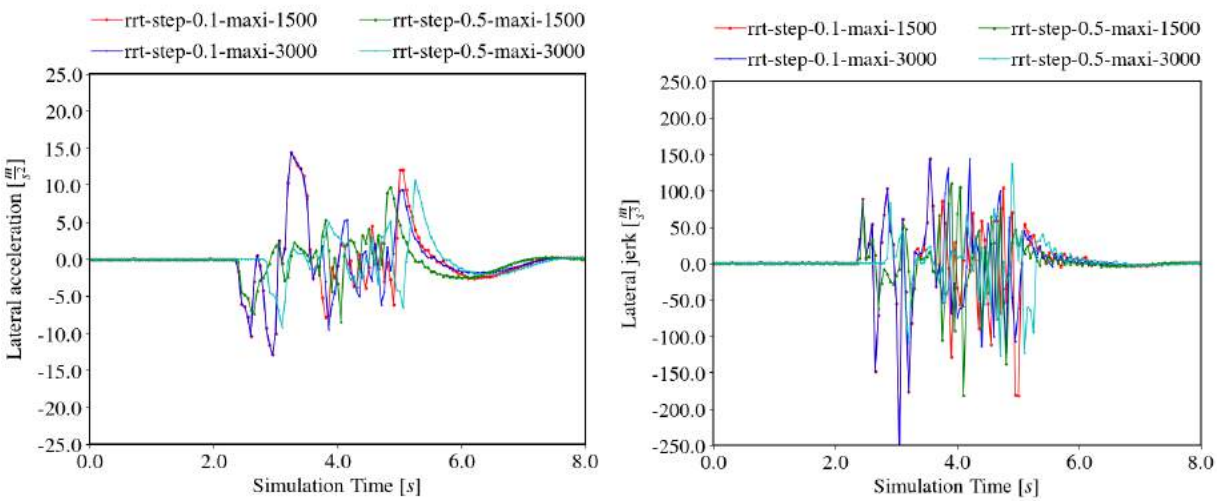

Figure 6.13: RRT* lateral metrics for synchronous cases.

Regarding lateral jerk and acceleration, RRT* has the worst performance. In figure 6.13 we visualize the lateral jerk and acceleration timelines for the synchronous experiments. The jerk and acceleration are extremely noisy. These noisy measurements are not attributed to the simulated IMU, as that perfectly measures the acceleration of the vehicle. The RRT* path visualized in figure 6.14 explains the noise. Without any trajectory optimization, the PID controller has to track a randomly varying path. This results in constant changes to steering, which produce frequent changes to acceleration and jerk. This can be ameliorated by introducing trajectory optimization or model predictive control to improve kinematic feasibility after the path is produced. Alternatively, we can adjust the step size of RRT* or improve the steering function between nodes to improve the smoothness of the path. As per

Figure 6.14: Visualization of the RRT* planner in the pedestrian behind car scenario

section 4, adjusting these parameters introduces other concerns, such as coarse discretization error and collision checking requirements.

6.8 Hybrid A* Results

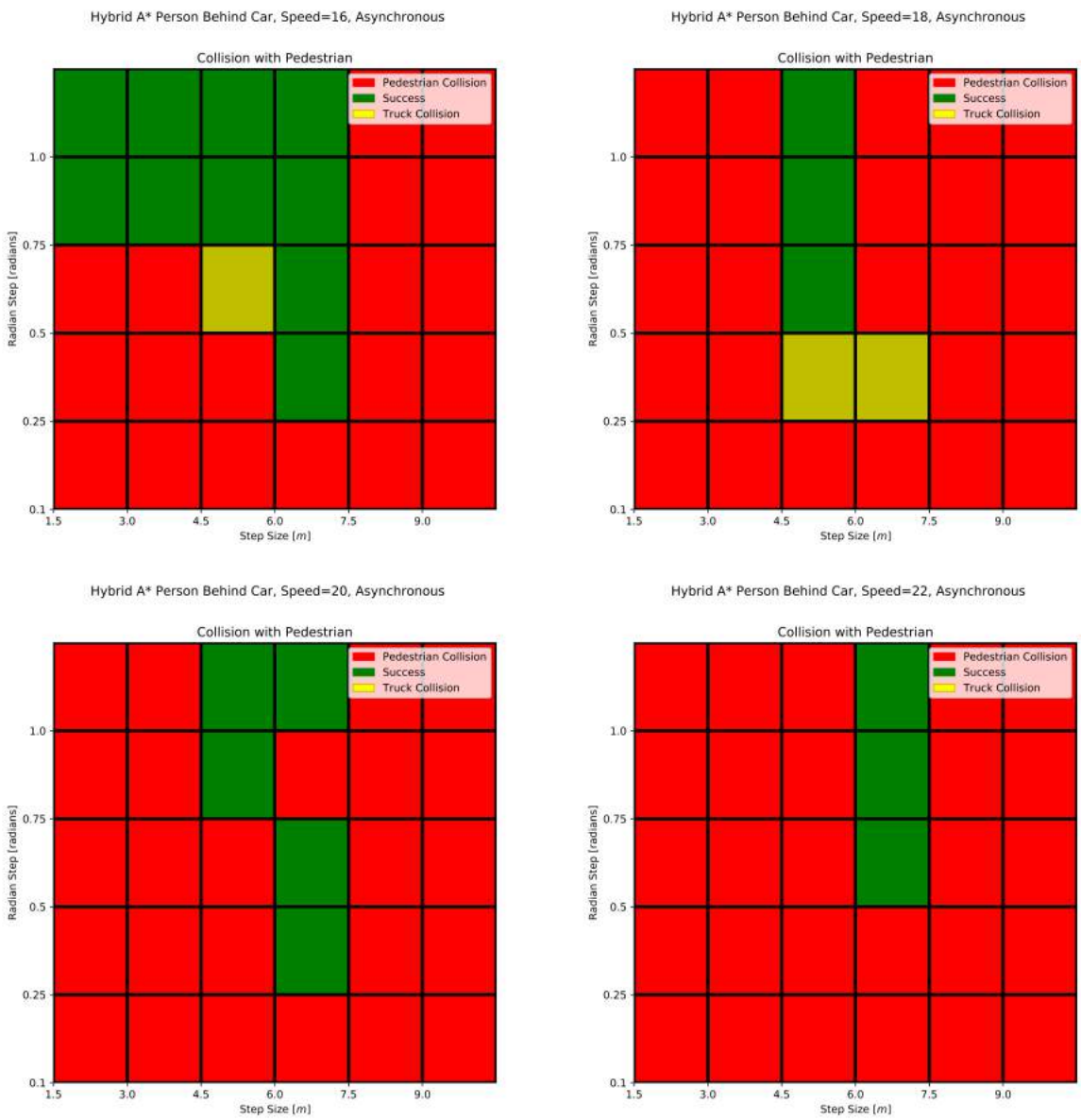

Figure 6.15: Hybrid A* collision tables for asynchronous cases.

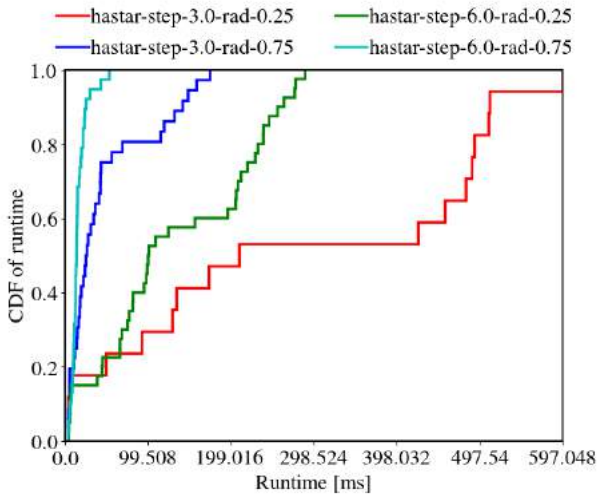

Figure 6.16: Hybrid A* run-time CDFs for asynchronous cases.

Hybrid A* had the lowest success rate across all planners and both synchronous and asynchronous scenarios. The original algorithm showed success in mostly static situations, but our experiments involve highly dynamic challenges. In the collision tables for the asynchronous experiments, we see that there is a very narrow set of configurations that results in success. The step size and radian step discretization are very brittle, as slight increases could exponentially increase the number of nodes explored during the search. This trend is exhibited in the run-time CDF. The finest resolution has significant variance in the run-time, with the worst case being over 500 ms. In many planning cycles, Hybrid A* simply fails to find a solution due to the discretization parameters. Fine discretizations are needed in these tight maneuvers, but higher resolution leads to huge increases in run-time.

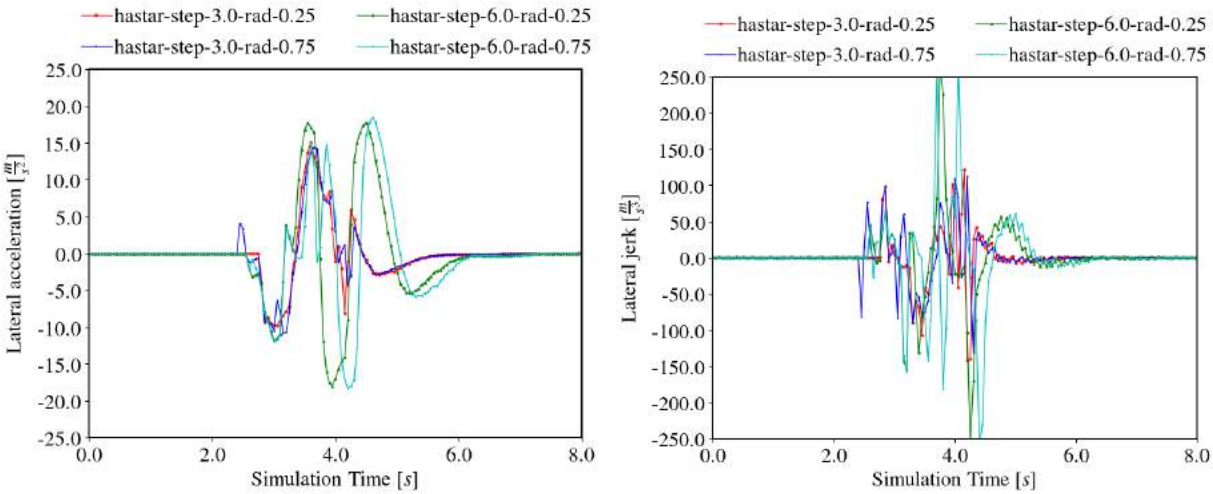

Figure 6.17: Hybrid A* lateral metrics for synchronous cases.

Figure 6.18: Visualization of the Hybrid A* planner in the pedestrian behind car scenario.

Qualitatively, the paths were smooth but generally took unnecessarily sharp turns. This indicates that the radian step size was too coarse. These sharp turns are also reflected in

the lateral jerk and acceleration plots. While the jerk and acceleration were not quite as variable as those of RRT*, the magnitude of jerk and acceleration were the highest across all planners. This would make the maneuver more unpleasant for any passengers in the car. Overall, there are newer variations of Hybrid A* that could be explored in future work to better handle these dynamic scenarios. Further analysis of the relevant hyperparameters could be beneficial as well.

6.9 End-to-end with Time to Decision

Finally, we can utilize our results above to construct an end-to-end autonomous driving pipeline that utilizes the time to decision operator to dynamically allocate time. We use the 99th percentile of motion planning run-times to determine the expected maximum run-time of the planner. This gives us a mapping of expected maximum run-time to a planner hyperparameter configuration. Given a time to decision, the motion planner can select a configuration that meets the time to decision criteria. We can also use the comfort metrics and safety metrics to prune bad configurations. For example, the Hybrid A* planner had multiple configurations that successfully navigated the scenario at 16 m/s. But the maximum jerk can differ by more than 100 m/s^3 across the configurations. We could prune the configurations that resulted in unsafe amounts of jerk and keep those that comfortably navigated the scenario. We would still need to conduct further testing on a variety of scenarios to substantiate our findings in comfort and safety. In this manner, we incorporate our findings on the strengths and weaknesses of each planner into a dynamic deadline system. This allows us to dynamically allocate time to produce the safest, most comfortable motion plans across a variety of situations.

Now we describe our end-to-end autonomous driving pipeline implementation. For sensing, we utilize Carla simulated cameras, LiDARs and IMUs. The cameras and LiDARs allow us to run object detection, tracking and prediction while the IMU provides measurements for planning and control. We use EfficientDet [67] for object detection, SORT [8] for tracking and linear prediction. We use the same route planner and behavioral planner as per our experiment setup in section 5. We use the Frenet Optimal Trajectory planner for motion planning and a PI controller for throttle, braking and steering. We use the same pedestrian behind car scenario for evaluation. The time to decision uses a base decision deadline of 400 ms at 10 m/s. For every 1 m/s over the base 10 m/s, the time to decision is decreased by 10 ms. We use the 99th percentile of Frenet Optimal Trajectory run-times for each configuration as the expected maximum run-time. Then, we can adaptively select which motion planner to utilize in each situation. For example, the time to decision is 300 ms when the vehicle is moving at 20 m/s. After accounting for the perception, tracking and prediction run-times, we can select the optimal planner to use. If the upstream components take 180 ms to complete, then we can select a Frenet Optimal Trajectory configuration that runs in at most 120 ms. To minimize the run-time variance of perception and prediction, we fixate the model parameters and number of obstacles in the scene. We note that this process

for dynamically configuring motion planning models can also be applied to the rest of the components. The result is a dynamic deadline, end-to-end autonomous driving system that more safely navigates difficult scenarios compared to similar static deadline systems.

Chapter 7

Future Work

Our motivational work is limited to a few hand-engineered scenarios and a few planners. It has yet to be validated on a breadth of driving situations that better represent real world driving. Future work should be aimed at verifying these trends across different driving scenarios and testing a wider variety of motion planners. More elegant behavioral layers could also be important in determining how dynamic deadlines can impact motion planning. Using more tailored prediction modules such as MultiPath can also improve the quality of our results. Hybrid A*, RRT* and Frenet Optimal Trajectory were presented in their simplest forms and there are many optimizations and improvements that could decrease the overfitting of planner configurations to scenarios. For example, we could implement the trajectory optimization step for Frenet Optimal Trajectory or introduce Dubin's steering for RRT*. Utilizing better control methods such as model predictive control or pure pursuit could also eliminate some of the confounding introduced by static PID control gains. Overall, a more diverse set of experiments and state of the art autonomous vehicle stack could better motivate the need for dynamically configuring motion planners.

Our results in dynamically configured motion planners are highly preliminary. So far we have provided motivation for why dynamic deadlines are necessary and demonstrated success on a single scenario and planner. Some future exploratory work includes refining the time to decision computation, creating a more rigorous process for selecting motion planner configurations by run-time, and establishing motion planning benchmarks in which a broader range of driving scenarios can be tested. Our time to decision computation is simple and does not take into account obstacles. Ideally it should be extended to include obstacle information, vehicle dynamic constraints, and so on. We also selected motion planner configurations manually. We took the successful planning configurations at various speeds and computed their maximum expected run-times to create a mapping. Yet there can be multiple configurations that lead to the same run-time, or configurations with longer run-times that have better jerk and acceleration metrics. Developing a more thorough and scientific method for selecting these configurations is left as future work. This would likely include establishing a motion planning benchmark, on which we can evaluate our planners and obtain performance metrics.

Chapter 8

Conclusion

This thesis addresses the trade-offs in 3 different motion planning algorithms and proposes dynamic deadlines as an alternative to static deadline motion planning. We build our autonomous vehicle stack and motion planning algorithms on top of ERDOS and Pylot, which provide a time-to-decision operator that utilizes vehicle speed and environment state to produce a time-to-decision. We use Carla, a simulator designed to test self-driving vehicles, to substantiate our experiments in physically realistic situations and environments. We presented a high-level overview of autonomous vehicle technology and provided motivation for dynamic deadlines as a solution to variable run-times and constraints. We described motion planning in depth, covering the structure of motion planning modules and discussing the trade-offs of different motion planners. We detailed 3 motion planning algorithms of interest, including low level implementation details and insight to their weaknesses and strengths. We explained our experiment setup in Carla and covered the metrics we measure. We analyzed our experiment results and commented on their implications. Finally, we showed how to construct dynamic deadline system in an end-to-end autonomous driving vehicle in Carla. Ultimately, while our work is preliminary, it establishes the foundation for future research in dynamical deadline motion planning.

Bibliography

[1] Pieter Abbeel. *CS287 Advanced Robotics, Fall 2019, Motion Planning*. 2019.

[2] United States Environmental Protection Agency. *U.S. Transportation Sector Greenhouse Gas Emissions 1990-2017*. 2017. URL: https://www.epa.gov/greenvehicles/fast-facts-transportation-greenhouse-gas-emissions/.

[3] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org.

[4] T. Bandyopadhyay et al. "Intention-aware motion planning". In: *Algorithmic Foundations of Robotics X*. Springer International Publishing, 2013, pp. 475–491.

[5] Mayank Bansal, Alex Krizhevsky, and Abhijit S. Ogale. "ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst". In: *CoRR* abs/1812.03079 (2018). arXiv: 1812.03079. URL: http://arxiv.org/abs/1812.03079.

[6] Hannah Bast et al. "Route Planning in Transportation Networks". In: *CoRR* abs/1504.05140 (2015). arXiv: 1504.05140. URL: http://arxiv.org/abs/1504.05140.

[7] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[8] Alex Bewley et al. "Simple Online and Realtime Tracking". In: *CoRR* abs/1602.00763 (2016). arXiv: 1602.00763. URL: http://arxiv.org/abs/1602.00763.

[9] Dillmann Brechtel Gindele. "Probabilistic Decision Making Under Uncertainty for Autonomous Driving using Continuous POMDPs". In: 2014, pp. 392–399.

[10] Sebastian Brechtel, Tobias Gindele, and Rüdiger Dillmann. "Probabilistic MDP-behavior planning for cars". In: Oct. 2011. DOI: 10.1109/ITSC.2011.6082928.

[11] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642039901.

[12] Yuning Chai et al. *MultiPath: Multiple Probabilistic Anchor Trajectory Hypotheses for Behavior Prediction*. 2019. arXiv: 1910.05449 [cs.LG].

[13] Yu Fan Chen et al. "Socially aware motion planning with deep reinforcement learning". In: Sept. 2017, pp. 1343–1350. DOI: 10.1109/IROS.2017.8202312.

[14] E.W. DIJKSTRA. "A Note on Two Problems in Connexion with Graphs." In: *Numerische Mathematik* 1 (1959), pp. 269–271. URL: http://eudml.org/doc/131436.

[15] Dmitri Dolgov et al. "Practical Search Techniques in Path Planning for Autonomous Driving". In: *AAAI Workshop - Technical Report* (Jan. 2008).

[16] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *CoRR* abs/1711.03938 (2017). arXiv: 1711.03938. URL: http://arxiv.org/abs/1711.03938.

[17] L. E. Dubins. "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents". In: *American Journal of Mathematics* 79.3 (1957), pp. 497–516. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2372560.

[18] Hugh Durrant-Whyte and Tim Bailey. ""Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms"". In: *Robotics and Automation Magazine* 13 (Jan. 2006).

[19] M. Everingham et al. "The Pascal Visual Object Classes Challenge: A Retrospective". In: *International Journal of Computer Vision* 111.1 (Jan. 2015), pp. 98–136.

[20] Lex Fridman et al. "Automated Synchronization of Driving Data Using Vibration and Steering Events". In: *CoRR* abs/1510.06113 (2015). arXiv: 1510.06113. URL: http://arxiv.org/abs/1510.06113.

[21] Eustice Galceran Cunningham and Olson. "Multipolicy decision-making for autonomous driving via changepoint-based behavior prediction". In: 2015, p. 2.

[22] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[23] Robert Geisberger et al. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* 46 (Aug. 2012), pp. 388–404. DOI: 10.2307/23263550.

[24] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. "Similarity Search in High Dimensions via Hashing". In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 518–529. ISBN: 1558606157.

[25] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[26] Ionel Gog et al. *Elastic Robot Dataflow Operating System*. Version 0.2.0. May 15, 2020. URL: https://github.com/erdos-project/erdos.

[27] Ionel Gog et al. *Pylot*. May 15, 2020. URL: https://github.com/erdos-project/pylot.

[28] Andrew V. Goldberg and Chris Harrelson. "Computing the Shortest Path: A* Search Meets Graph Theory". In: (2005).

[29] Eric A. Hansen and Rong Zhou. "Anytime Heuristic Search". In: *CoRR* abs/1110.2737 (2011). arXiv: 1110.2737. URL: http://arxiv.org/abs/1110.2737.

[30] Frank Havlak and Mark E. Campbell. "Discrete and Continuous, Probabilistic Anticipation for Autonomous Robots in Urban Environments". In: *CoRR* abs/1309.0766 (2013). arXiv: 1309.0766. URL: http://arxiv.org/abs/1309.0766.

[31] Jonathan Huang et al. "Speed/accuracy trade-offs for modern convolutional object detectors". In: *CoRR* abs/1611.10012 (2016). arXiv: 1611.10012. URL: http://arxiv.org/abs/1611.10012.

[32] INRIX. *INRIX 2019 Global Traffic Scorecard*. 2019. URL: https://inrix.com/scorecard/.

[33] Joel Janai et al. "Computer Vision for Autonomous Vehicles: Problems, Datasets and State-of-the-Art". In: *CoRR* abs/1704.05519 (2017). arXiv: 1704.05519. URL: http://arxiv.org/abs/1704.05519.

[34] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: *CoRR* abs/1105.1186 (2011). arXiv: 1105.1186. URL: http://arxiv.org/abs/1105.1186.

[35] Christos Katrakazas et al. "Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions". In: *Transportation Research Part C: Emerging Technologies* 60 (2015), pp. 416–442. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2015.09.011. URL: http://www.sciencedirect.com/science/article/pii/S0968090X15003447.

[36] Lydia Kavraki et al. *Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*. Tech. rep. Stanford, CA, USA, 1994.

[37] R. Kesten et al. *Lyft Level 5 AV Dataset 2019*. urlhttps://level5.lyft.com/dataset/. 2019.

[38] Kiam Heong Ang, G. Chong, and Yun Li. "PID control system analysis, design, and technology". In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576.

[39] Gregory R. Koch. "Siamese Neural Networks for One-Shot Image Recognition". In: 2015.

[40] S. Koenig and M. Likhachev. "Fast replanning for navigation in unknown terrain". In: *IEEE Transactions on Robotics* 21.3 (2005), pp. 354–363.

[41] J. Kong et al. "Kinematic and dynamic vehicle models for autonomous driving control design". In: *2015 IEEE Intelligent Vehicles Symposium (IV)*. 2015, pp. 1094–1099.

[42] Steven M. LaValle. *Planning Algorithms*. USA: Cambridge University Press, 2006. ISBN: 0521862051.

[43] Laura Leal-Taixé et al. "MOTChallenge 2015: Towards a Benchmark for Multi-Target Tracking". In: *CoRR* abs/1504.01942 (2015). arXiv: 1504.01942. URL: http://arxiv.org/abs/1504.01942.

[44] N. Li et al. "Game Theoretic Modeling of Driver and Vehicle Interactions for Verification and Validation of Autonomous Vehicle Control Systems". In: *IEEE Transactions on Control Systems Technology* 26.5 (2018), pp. 1782–1797.

[45] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. "ARA*: Anytime A* with Provable Bounds on Sub-Optimality". In: *Proceedings of the 16th International Conference on Neural Information Processing Systems*. NIPS'03. Whistler, British Columbia, Canada: MIT Press, 2003, pp. 767–774.

[46] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm". In: *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*. ICAPS'05. Monterey, California, USA: AAAI Press, 2005, pp. 262–271. ISBN: 1577352203.

[47] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[48] Matthew McNaughton et al. "Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice". In: *Proceedings of the International Conference on Robotics and Automation*. May 2011, pp. 4889–4895.

[49] California Department of Motor Vehicles. *Autonomous Vehicles Disengagement Reports*. 2020. URL: https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/auto.

[50] General Motors. *Self-Driving Safety Report*. 2018. URL: https://www.gm.com/content/dam/company/docs/us/en/gmcom/gmsafetyreport.pdf.

[51] Nils J. Nilsson. "A Mobile Automaton: An Application of Artificial Intelligence Techniques". In: (1969).

[52] Alan Ohnsman. "Waymo and Cruise Vie for Supremacy in Murky California Self-Driving Data". In: (2020). URL: https://www.forbes.com/sites/alanohnsman/2020/02/26/waymo-and-cruise-vie-for-supremacy-in-murky-california-self-driving-data/#332144017604.

[53] E. Olson. "A passive solution to the sensor synchronization problem". In: Nov. 2010, pp. 1059–1064. DOI: 10.1109/IROS.2010.5650579.

[54] Brian Paden et al. "A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles". In: *CoRR* abs/1604.07446 (2016). arXiv: 1604.07446. URL: http://arxiv.org/abs/1604.07446.

[55] Brian Paden et al. "A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles". In: *CoRR* abs/1604.07446 (2016). arXiv: 1604.07446. URL: http://arxiv.org/abs/1604.07446.

[56] J. A. Reeds and L. A. Shepp. "Optimal paths for a car that goes both forwards and backwards." In: *Pacific J. Math.* 145.2 (1990), pp. 367–393. URL: https://projecteuclid.org:443/euclid.pjm/1102645450.

[57] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[58] Nicholas Rhinehart, Kris M. Kitani, and Paul Vernaza. "r2p2: A ReparameteRized Pushforward Policy for Diverse, Precise Generative Path Forecasting". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 794–811. ISBN: 978-3-030-01261-8.

[59] Dorsa Sadigh et al. "Planning for Autonomous Cars that Leverage Effects on Human Actions". In: June 2016. DOI: 10.15607/RSS.2016.XII.029.

[60] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. "Planning and Decision-Making for Autonomous Vehicles". In: *Annual Review of Control, Robotics, and Autonomous Systems* 1.1 (2018), pp. 187–210. DOI: 10.1146/annurev-control-060117-105157. eprint: https://doi.org/10.1146/annurev-control-060117-105157. URL: https://doi.org/10.1146/annurev-control-060117-105157.

[61] F. Sivrikaya and B. Yener. "Time synchronization in sensor networks: a survey". In: *IEEE Network* 18.4 (2004), pp. 45–50.

[62] statistia. *Number of Motor Vehicle Registered in the United States from 1990 to 2018*. 2020. URL: https://www.statista.com/statistics/183505/number-of-vehicles-in-the-united-states-since-1990/.

[63] Anthony Stentz. "Optimal and efficient path planning for partially-known environments". In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* (1994), 3310–3317 vol.4.

[64] Anthony Stentz. "The Focussed D* Algorithm for Real-Time Replanning". In: *Proc Int Joint Conf on Artificial Intelligence* (Feb. 2000).

[65] Pei Sun et al. *Scalability in Perception for Autonomous Driving: Waymo Open Dataset*. 2019. arXiv: 1912.04838 [cs.CV].

[66] O. Takahashi and R. J. Schilling. "Motion planning in a plane using generalized Voronoi diagrams". In: *IEEE Transactions on Robotics and Automation* 5.2 (1989), pp. 143–150.

[67] Mingxing Tan, Ruoming Pang, and Quoc V. Le. *EfficientDet: Scalable and Efficient Object Detection*. 2019. arXiv: 1911.09070 [cs.CV].

[68] Ayushi Thakur. "An overview of "Drive by wire" technology for Automobiles". In: Jan. 2019.

[69] Q. Tran and J. Firl. "Online maneuver recognition and multimodal trajectory prediction for intersection assistance using non-parametric regression". In: *2014 IEEE Intelligent Vehicles Symposium Proceedings*. 2014, pp. 918–923.

[70] United States Department of Transportation. *National Statistics*. 2020. URL: https://www-fars.nhtsa.dot.gov/Main/index.aspx.

[71] Jasper Uijlings et al. "Selective Search for Object Recognition". In: *International Journal of Computer Vision* 104 (Sept. 2013), pp. 154–171. DOI: 10.1007/s11263-013-0620-5.

[72] Simon Ulbrich and Markus Maurer. "Probabilistic online POMDP decision making for lane changes in fully automated driving". In: Oct. 2013.

[73] R Verma and D Vecchio. "Semi-autonomous multi-vehicle safety". In: *Robotics and Automation Magazine* 18 (2001), pp. 44–54.

[74] Wenda Xu et al. "A real-time motion planner with trajectory optimization for autonomous vehicles". In: *2012 IEEE International Conference on Robotics and Automation*. 2012, pp. 2061–2067.

[75] Moritz Werling et al. "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame". In: June 2010, pp. 987–993. DOI: 10.1109/ROBOT.2010.5509799.

[76] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. "Simple Online and Realtime Tracking with a Deep Association Metric". In: *CoRR* abs/1703.07402 (2017). arXiv: 1703.07402. URL: http://arxiv.org/abs/1703.07402.

[77] Sze Zheng Yong, Minghui Zhu, and Emilio Frazzoli. "Generalized innovation and inference algorithms for hidden mode switched linear stochastic systems with unknown inputs". In: *Proceedings of the IEEE Conference on Decision and Control* 2015 (Feb. 2015), pp. 3388–3394. DOI: 10.1109/CDC.2014.7039914.

[78] Fisher Yu et al. "BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling". In: *CoRR* abs/1805.04687 (2018). arXiv: 1805.04687. URL: http://arxiv.org/abs/1805.04687.

[79] Q. Zhu, Y. Yan, and Z. Xing. "Robot Path Planning Based on Artificial Potential Field Approach with Simulated Annealing". In: *Sixth International Conference on Intelligent Systems Design and Applications*. Vol. 2. 2006, pp. 622–627.