Real-Time Graph-Based Path Planning for Autonomous Racecars

by

Elizabeth Keefer

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 5, 2023

Keywords: path planning, graph search, autonomous vehicle

Approved by

David Bevly, Chair, Bill and Lana McNair Distinguished Professor of Mechnaical Engineering
Scott Martin, Assistant Professor of Mechanical Engineering
Thaddeus Roppel, Associate Professor Emeritus of Electrical and Computer Engineering

Abstract

This thesis presents a computationally efficient graph-based motion planner designed for high speed autonomous racing. The emergence of autonomous racing competitions, such as the Indy Autonomous Challenge (IAC), have sought to test the limits of autonomous vehicle technology to accelerate development within the domain. A fast, safe, and reliable motion planning algorithm is developed for an autonomous vehicle operating under high-speed conditions such as the ones in the IAC. A variety of planning methods are investigated for this purpose, such as graph-based planning and sampling-based planning, among others. A graph-based method using the A* search algorithm is selected due to its computational efficiency, reliability, and predictability in structured environments. The proposed planner is augmented with techniques for integrating vehicular constraints with path smoothing and edge generation.

Two versions of the proposed path planner are presented. The version used for the 2021 and 2022 IAC competitions on oval tracks is developed and test results from simulation and running the planner in real time competition are presented. Additionally, improvements to the planner are implemented to enhance the dynamic feasibility of the planned path and allow for use on road courses. The improved planner is tested on an autonomous consumer sedan as well as in simulation. Both iterations of the proposed algorithm are shown to produce dynamically feasible maneuvers in the presence of *a priori* unknown obstacles while maintaining faster than real-time performance.

Acknowledgments

First I would like to acknowledge my parents for being excited, and not mad, that I decided to go to grad school instead of getting a job. Thank you for always being supportive of whatever random endeavours I decide to pursue next. A big thanks Howard for convincing me that I should go to grad school during our weekly meetings in Software for Sensors at the start of the pandemic. I would also like to thank Dr. Bevly and Dr. Martin for providing an opportunity for me to join the GAVLAB. I'll miss the lab lunches, ski trips, and never-ending jokes about tofu. Thank you Dr. Roppel for serving on my committee and also for your enthusiasm for all things robots that made class that much more interesting.

Next I would like to thank my fellow GAVLAB cohort. Thank you Kathleen for being my co-conspirator in all our lab shenanigans, tailgates, and party planning. And to Walt, who was always down for a Birmingham shopping trip or any of our last-minute adventures. Thank you Billy for the daily crossword reminder; without you I might have forgotten. And to Grubes, for allowing all of us to invade your apartment for late-night commiserations over Auburn football, grad school, and life. Thank you to the rest of the Dead Reckoners trivia team; Wednesdays were always a highlight of my week (and the group in the corner was definitely cheating)! And to Will Bryan for being in the top twenty (and his belief in me throughout our time in ATR and during the formulation of this thesis).

A thanks to the many people whom I have not named but have shaped me into the person I am today. These include my friend group from high school, my cycling teammates, my fellow AUMB baritones and music lovers, and all the people in between who have filled my my life with love, passion, and weirdness. I couldn't forget a shout-out to Quentin and Elliot. All the long nights and travels were so that I could give y'all the life you deserve. Lastly, I would like to thank Christian Moomaw for being my forever best friend (and fiancé), biggest cheerleader (and biggest critic), and for continually pushing me to be the best version of myself and loving me regardless.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction and Motivation

The Society of Automotive Engineers (SAE) defines six levels of vehicle autonomy, seen in Figure 1.1. The lowest level, Level 0, describes no autonomy at all, with the human driver having full control of the vehicle, excepting for rudimentary safety technologies such as an anti-lock braking system (ABS) [1] and blind spot detection [2]. Conversely, the highest level, Level 5, describes full autonomous control which never requires human input, even as the vehicle negotiates a wide variety of scenarios, conditions, and terrains. Level 1 and Level 2 technologies such as adaptive cruise control [3] and rudimentary lane keeping [4] are becoming commonplace in vehicles as consumer safety measures. These lower levels (0–2) are categorized more broadly as driver-supported features with which the driver must be *constantly* engaged while operating the vehicle.

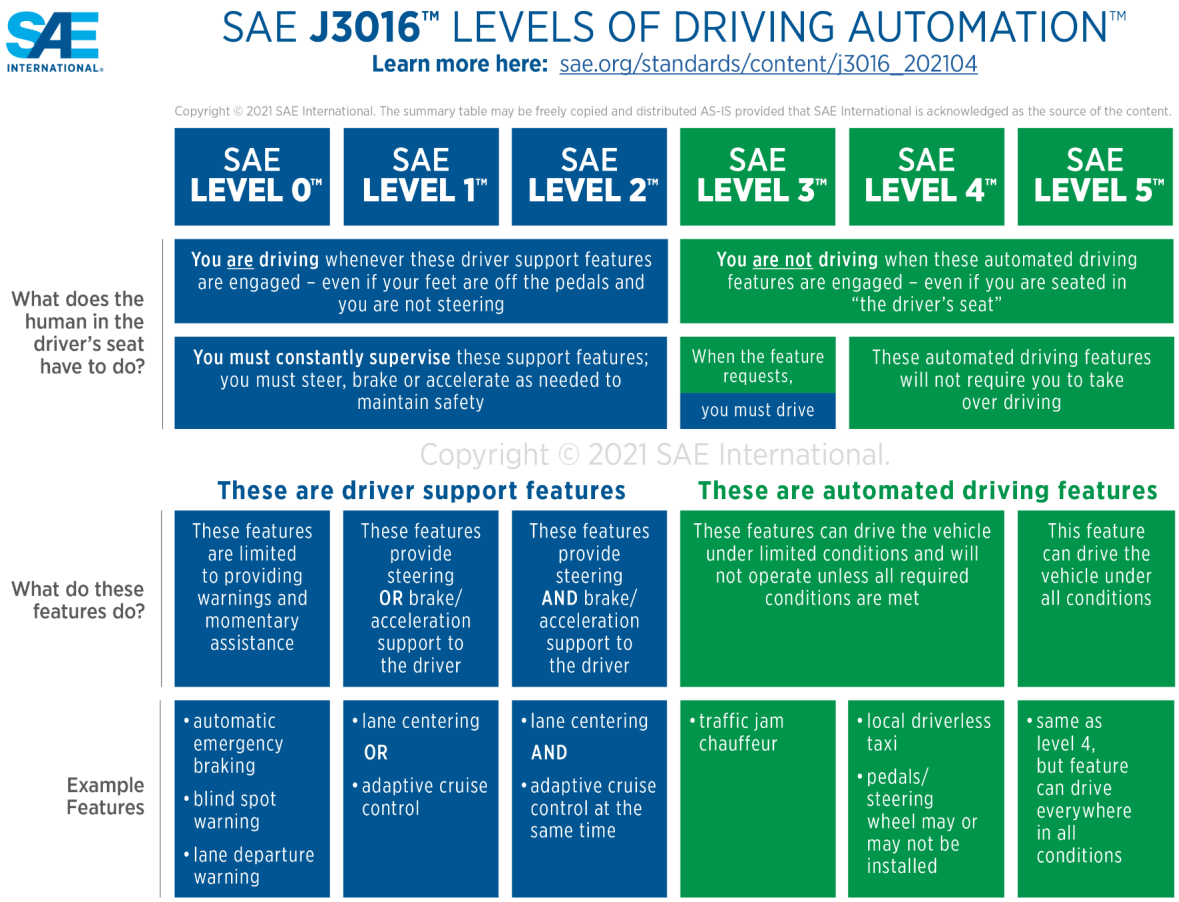


Figure 1.1: 6 Levels of Driving Automation [5]

It is not until systems reach Level 3 autonomy that they begin to resemble the stereotypical idea of vehicle autonomy. This level allows for full vehicle control in *specific* situations but requires human intervention upon requests. Level 3 began to emerge on the consumer market with Honda being one of the first approved manufacturer to sell to consumers in 2001, but only in a very limited market [6]. The technology is called *Traffic Jam Pilot* and is only used under certain conditions, such as congested traffic, but does allow the driver to partially disengage from the driving task. Examples of Level 4 autonomy have been seen in autonomous taxi services such as Waymo [7] (seen in Figure 1.2 and Cruise [8]. However, because Waymo has human monitoring and intervention on standby for edge case scenarios and Cruise has a limited working area, neither company can claim full Level 5 autonomy despite having no physical driver present.

Figure 1.2: Waymo, a Level 4 Autonomous Taxi

To spur on innovation and push the limits of the then current autonomous technologies, the Defense Advanced Research Projects Agency (DARPA) put on a race in the early 2000s. In 2004, they hosted what was to be their first Grand Challenge, which consisted of a 210-mile course through the Mojave Desert [9]. The event ended with no winner, as the last team standing made it only 7.4 miles before their vehicle drove itself partially off the edge of a road.

The following year the second edition of the Grand Challenge was hosted with more compelling results. Five of the 23 participating teams managed to finish the 132-mile course. The winner, a robot named Stanley (seen in Figure 1.3 from the Stanford Racing Team, came in at 6 hours and 53 minutes [10]. All the teams were given was a list of latitude, longitudes, corridor widths, and speed limits associated with almost 3,000 waypoints along the course. It was up to the teams to have prepared for the unknown elements of the course. The Second Grand Challenge led to advancements in research and technologies as well as leading to the founding of many autonomous robotic startups.

Figure 1.3: Stanford's Stanley

Nearly a decade later, another wave of autonomous racing competitions emerged. These include RoboRace [11] (Figure 1.4a) and the Indy Autonomous Challenge [12] (Figure 1.4b.

Roborace was an European based series of autonomous races that takes place in the Roborace Metaverse, described as “a mixed reality that blends the virtual and physical worlds” that started in 2015 and held competitions from 2019 to 2021. It employed a World Endurance Championship (WEC) style chassis with an electric powertrain competing on a mixture of traditional circuits and simulated environments with various goals for each round. The competitions themselves ranged from fastest lap to static and dynamic object avoidance. The races were all only single-vehicle with the competition utilizing ”ghost” cars on track as dynamic objects [13].

(a) RoboRacing (b) IAC 2021

Figure 1.4: Autonomous Racing Series

The IAC, an US-based series, was inspired by the DARPA Grand Challenge and sought to be the first fully autonomous multi-vehicle race. Utilizing a Dallara AV-21 racecar, also used in the Indy Light series, that was retrofitted with a complete sensor package and drive-by-wire capabilities, the series has grown from single vehicle time trials to a two-car passing competition with plans to continue advancing the complexity of the race. The IAC was founded to help address three major issues hurdles facing the advancement of autonomous vehicle (AV) technology in commercial spaces:

1. ***Solving “edge case” scenarios*** - finding and testing scenarios at the limits of the vehicles known capabilities and ensure vehicular control and safety

2. ***Catalyzing New AV technologies and innovators*** - like with DARPA, pushing a new generations of engineers and researchers towards innovation of new technologies and softwares in the field of AV

3. ***Engaging the public to help ensure acceptance and use of AV technologies*** - exposing the general public, who may not have many opportunities for close-up interactions with AVs, to the idea of AV and help improve their understanding of the technologies

Every aspect of an autonomous code stack answers some basic question asked of autonomous systems. Localization, or estimation, answers the questions of "Where am I?", perception answers "What do I see?", and path planning answers "I know where I need to go, but what path should I take to get there?" The last question is answered more broadly by *global* path planning and typically involves a type of optimal solution that does not take into consideration unexpected obstacles.

This thesis focuses on a different way of answering the last question by providing a solution that is generally referred to as a *local* path planner, By using global waypoints, the planner aims to negotiate a high-speed autonomous race car around a track in real-time while handling the avoidance of obstacles not previously accounted for in the global path. The IAC competitions serve as the basis of this thesis.

1.2 Contributions

While path planning for autonomous vehicles is a well-researched field with a diverse set of solutions and algorithms, path planning specifically for high-speed racing vehicles is a new field of study. The principal contribution of this thesis is a performant path planner suitable for a racing environment on an oval and road course. Notable properties of the proposed path planner include:

- Integration with an otherwise full autonomous racing software stack.
- Layer and edge generation that take into account the dynamic capabilities of the vehicle in order to produce viable racing lines.
- Velocity profile generation to ensure dynamically feasible goal velocities along a path.
- A flexible software design that can support future algorithm development supporting more complex environments.

An earlier-in-development (and therefore more limited) version of the planner is also discussed which has been tested in competition on a physical racecar at an oval track. The full version of the proposed planner is tested in a high-fidelity simulation environment and on a low-speed autonomous consumer vehicle.

1.3 Outline

This chapter has served as an introduction to autonomous vehicles and their context in autonomous racing. It also provides context for the thesis by providing the contributions to the field of autonomous racing. Chapter 2 gives further background, exploring different aspects of path planning. This includes a look at various types of path planning algorithms, path smoothing techniques used in the context of autonomous vehicles, and path planning implementations in autonomous racing. Chapter 3 discusses the initial path planner design and its testing in both simulation and in a real-world environment. Chapter 4 discusses changes made to the aforementioned path planner to address its shortcomings and limitations. Chapter 5 discusses two experimental setups and results used to test the proposed path planner. Finally, chapter 6 provides a final summary, conclusion, and future work in this area.

Chapter 2

Background

2.1 Motion Planning

Motion planning, also known as path planning, is as a computational problem of getting from one place to another. One of the classical examples of a path planning problem is the "Piano's Mover Problem" [14]. The idea is to create an algorithm that gives the path of the piano given the inputs of the precise dimensions of the room, the objects, and the piano as well as a start and ending location. The output would include the translations and orientations the piano would have to move in order to get from the starting point to the goal while avoiding collisions with any of the objects.

There is a wide range of motivations for studying path planning techniques. The following list a few such motivations [15]:

- Discrete puzzles, operations, and scheduling [16] – like games such as chess or bridge to more complex operations such as folding sheet metal into a particular object
- Motion planning puzzles [17] – similar to the physical puzzles where movements in a precise order have to be executed to get the pieces apart where simulation environments can be set up in which path planning algorithms solve the order of movements
- Automotive assembly puzzle [18] – instead of making a small scale model of a car to figure out how a piece should be inserted such as Kineo CAM which was developed as a motion planning solver to figure out how best to assemble products

- Navigating mobile robots [19] – robots designed to autonomously explore unknown and/or unfamiliar terrain need to have the ability to safely navigate their changing environments

- Autonomous cars [20] – algorithms that will allow autonomous vehicles to negotiate the world around them, and more specifically in this thesis the path planning problem present is an autonomous race car that needs to safely and efficiently navigate an environment with other high speed autonomous race cars

- Flying through the air or in space [21] – like autonomous driving, bodies in the air, such as UAVs or drones, need to know how to safely navigate spaces that are also filled with other moving objects

- Designing better drugs [22] – in drug design, the "docking problem" is determining if a certain molecule can fit inside of a protein cavity which is similar to an assembly problem and can be solved with similar algorithms

Motion planning algorithms are traditionally classified into the following three categories [23]:

- Graph Search

- Sampling Based

- Interpolating curve planners

A further breakdown of each category is shown below in Figure 2.1

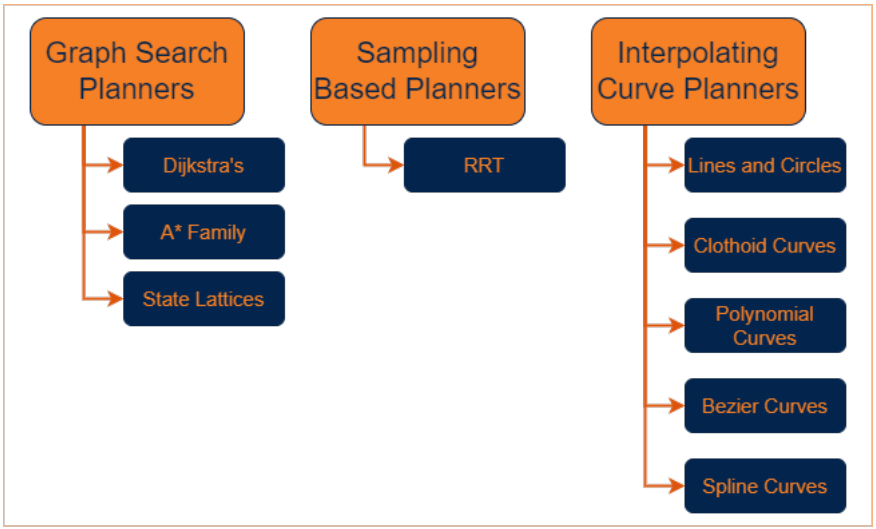


Figure 2.1: Traditional Motion Planning Algorithms

The classification of motion planning algorithms is further expanded in two large subsets: traditional and machine learning algorithms [24]. The machine learning algorithms are subdivided into the following categories:

- Supervised learning
- Optimal value reinforced learning
- Policy gradient reinforced learning

Graph search, also known as discrete search, planners discretize the search space into a graph of *nodes* and *edges*. The nodes represent a point in the search space while the edges connect the nodes to each other and sometimes have a *cost* associated to traversing to them. Discrete searches are often used in vehicle navigation because they do well in low-dimension space. However, in problems with a large search space, using a graph search method could become unwieldy in terms of computational memory and time. Also, graph search methods can only be as good as the ability to properly discretize and characterize the graph which might not be possible in unknown environments.

Sampling-based planning uses samples in the search space, and then constructs a path from those sample points. They are good for high dimensional problems and do not require complete discretization of the search space. However, they do not guarantee optimality and

certain obstacles can prove nearly impossible to navigate past. The more common sampling-based planners are randomized path planner (RPP), probabilistic road maps (PRM), and rapidly exploring random tree (RRT) or one of their derivatives.

Both graph search and sampling-based algorithms will be discussed in the following subsections. Interpolating curve planners will also be discussed, but not in the context of planners but rather in the context of edge generation options. Machine learning algorithms will not be discussed because they were ruled out early on as an option for the path planner. In part due to their complexity and need for training data that would not be readily available or easily reproducible.

2.1.1 Graph Search Algorithms

A graph is defined as a group of *nodes* and *edges* [25]. Nodes are also referred to as a *vertex* and will be used interchangeably in this thesis. In the context of motion planning for autonomous vehicles, each node represents a physical location. An edge connects two nodes that are related in a significant way with only edges the robot is able to realistically traverse considered.

An example of a graph can be seen in Figure 2.2. The vertices are represented by the circles labeled *A*-*I* and are connected by edges, represented by the double lines. In this example, Vertex B (V_B), Vertex C (V_C), Vertex D (V_D), Vertex G (V_G), can be traversed to from V_A using the edges E_{AB}, E_{AC}, E_{AD}, E_{AG}, respectively. In the case of this graph, the traversal from V_A to V_B along E_{AB} is as valid as the reverse traversal from V_B to V_A along, E_{BA}. This type of graph is referred to as an *undirected graph* [26].

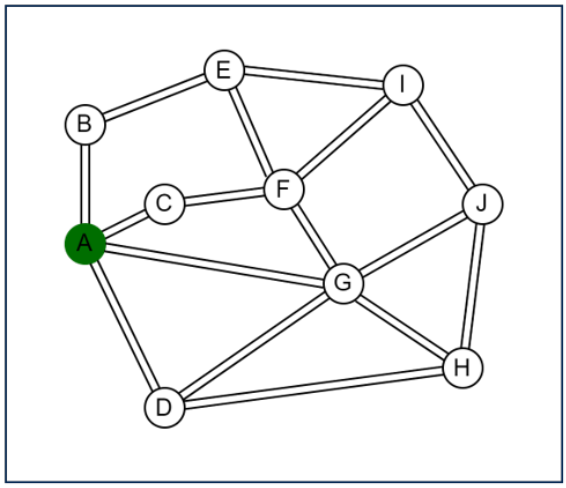


Figure 2.2: Graph Search Example

In the instances where the previous relationship is not true the graph is described as being a *directional* graph. An example of a directional graph is shown in Figure 2.3. Both example figures have similar node placement and edges with E_{AB} being a valid edge for both. However, the inverse edge, E_{BA}, is not valid for Figure 2.3. V_A can be traversed away from to V_B, V_C, or V_D, but the only way to return is from V_G on E_{GA}.

A graph is considered *connected* if a path exists between any two nodes, V_n and V_m for all nodes on the graph. A path is a series of adjacent and connected vertices. For example, a path between V_A and V_I might look like $V_A \xrightarrow{E_{AC}} V_C \xrightarrow{E_{CF}} V_F \xrightarrow{E_{FI}} V_I$. A graph is said to contain a *cycle* if a path exists a where the first and last vertices are the same. These cycles can enclose the entire graph, such as $V_A \xrightarrow{E_{AB}} V_B \xrightarrow{E_{BE}} V_E \xrightarrow{E_{EI}} V_I \xrightarrow{E_{IJ}} V_J \xrightarrow{E_{JH}} V_H \xrightarrow{E_{HD}} V_D \xrightarrow{E_{DA}} V_A$ from Figure 2.2 or can be nested inside the graph such as $V_A \xrightarrow{E_{AC}} V_C \xrightarrow{E_{CF}} V_F \xrightarrow{E_{FG}} V_G \xrightarrow{E_{GA}} V_A$. For the previous examples, direction is not necessary as the path can be traversed either clockwise or counterclockwise due to the undirected nature of the graph.

Looking at Figure 2.3, the directionality of certain edges means that the two cycles that were previously stated do not exist in here. In fact, the graph only contains one short cycle: $V_A \xrightarrow{E_{AD}} V_D \xrightarrow{E_{DG}} V_G \xrightarrow{E_{GA}} V_A$. This cycle can only be traversed in the counterclockwise direction.

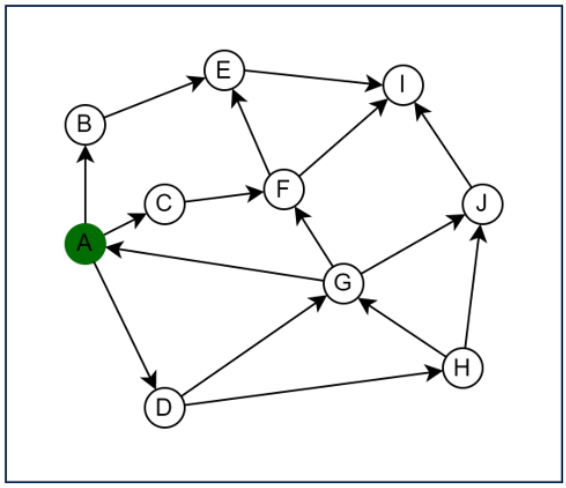


Figure 2.3: Directed Graph Search Example

A special subset of graphs is a known as a *tree* in which searching through is called a *tree traversal*. A tree is defined as directional graph that contains no cycles [25], an example of which can be seen in Figure 2.4.

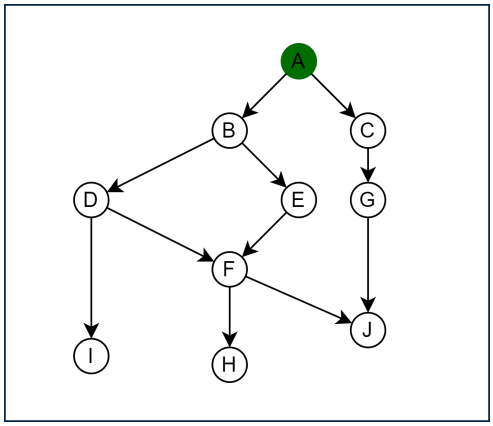


Figure 2.4: Tree Search Example

For graph traversals, there are two broad subsets of traversals that are commonly found in algorithms: *depth-first* and *breadth-first* search algorithms. Algorithms are not limited to only those two types, but many have elements of either one or the other.

A depth-first search algorithm explores along the length of a branch of a root node before moving on to a neighboring node. A better way to describe this may be to visualize a tree such as the one in 2.5. There are several general ways to traverse a graph using depth-first search

algorithms. In this example, the algorithm traverses the tree such that the left most unexplored node is selected and expanded.

Starting at V_A, the root of the tree, the node is expanded and it is discovered that it has three children: V_B, V_C, and V_D. Since the rules of this traversal state that the left most node must be explored and expanded, V_B is explored first and is expanded to show that it has two children: V_E and V_F. V_E, being the left most unexplored node is explored and expanded to show that it has one child, V_I. Once V_I is expanded and shown to not have any children, the exploration moves back to V_E. All the children of V_E have now been explored, so the algorithm once again moves back up to V_B, which has one unexplored child, V_F. Once V_F is explored and expanded the algorithm is done with the branch with the root V_B and it continues this pattern of exploration with the remaining children of V_A, V_C (light blue) and V_D (light orange).

Other types of depth-first search algorithm include variations on which child to explore first (left most or right most) or whether to completely expand a branch and explore for the bottom up. In other words, the original depth-first search algorithm described has the exploration order of:

$$V_A \to V_B \to V_E \to V_I \to V_F \to V_C \to V_G \to V_J \to V_K \to V_D \to V_H \to V_L$$

An algorithm that searches right first would look like:

$$V_A \to V_D \to V_H \to V_L \to V_C \to V_G \to V_K \to V_J \to V_B \to V_F \to V_E \to V_I$$

And an algorithm that expands and does not start exploring until the end of a branch would have an exploration order similar to:

$$V_I \to V_E \to V_F \to V_B \to V_J \to V_K \to V_G \to V_C \to V_L \to V_H \to V_D \to V_A$$

Another way to think of a depth-first search algorithm is equate it to the concept of last-in-first-out, LIFO. The last node explored or expanded is the first one to be explored or expanded next. Depth-first search is also associated *stacks* when talking about data structures.

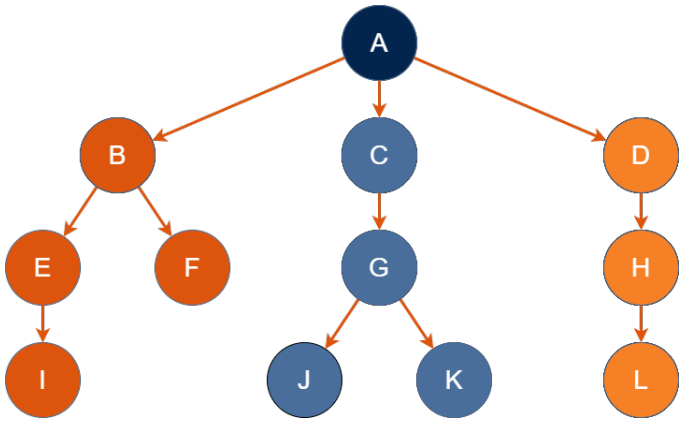


Figure 2.5: Depth First Search

Breadth first search algorithms are algorithms that explore every node at a certain *depth*, or labeled *layer* in Figure 2.6, before moving further [27]. Breadth-first search can be equated to the concept of first-in-first-out, FIFO and is often associated with the date structure type of *queues*. The algorithm starts at V_A and expands that node to find its children to be V_B, V_C, and V_D, same as before. The difference this time is that it will explore all three children before moving on to their children. In other words, the algorithm will explore and expand V_B to find it has children V_E and V_F but will move on to exploring V_C next. The exploration order will result in:

$$V_A \rightarrow V_B \rightarrow V_C \rightarrow V_D \rightarrow V_E \rightarrow V_F \rightarrow V_G \rightarrow V_H \rightarrow V_I \rightarrow V_J \rightarrow V_K \rightarrow V_L$$

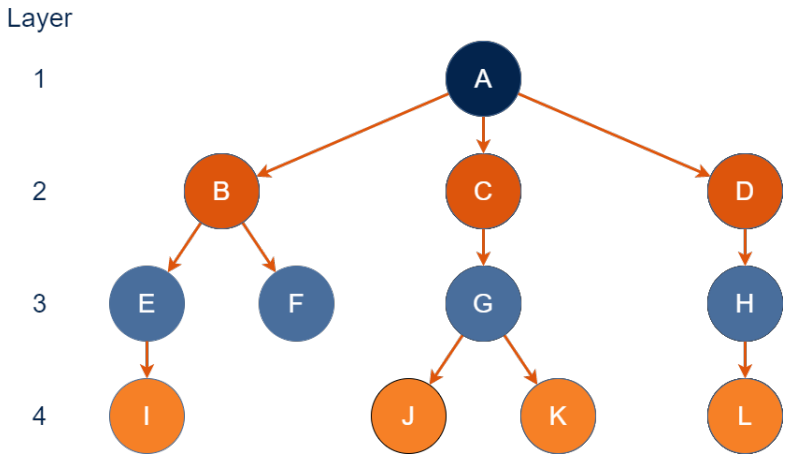


Figure 2.6: Breadth First Search

Dijkstra's algorithm

The most classic graph search algorithm is perhaps Dijkstra's algorithm [28]. Proposed by E. W. Dijkstra in 1959, the algorithm has similarities of a bread-first search algorithm while also being classified as *greedy algorithm*. A greedy algorithm is defined as one that always explores the best local, or immediate, solution [29]. In other words, with every iteration through the algorithm's search loop, the next node to explore is chosen by which one looks like the most optimal step from the immediate node without taking any outside information into consideration. This property can cause the algorithm to run slower than others in the same environment because it will tend to prioritize nodes closer to the start regardless of if that node is heading in the direction of the goal node or not. However, if a solution exists, Dijkstra guarantees it is a globally optimal solution [30].

An example pseudo-code for Dijkstra's Algorithm can be seen below in Algorithm 1. One of the required inputs is the $graph$, which is assumed to include the vertices and its neighboring vertices. At the start of the algorithm, the two arrays $dist$ and $prev$ are initialized, seen in lines 2 and 3. $dist$ is to contain the distance for every vertices to the start. In other words $dist[V]$ would return the distance from the vertex V to the starting vertex, V_{start}. The array is originally initialized to infinity since the vertices have not yet been traversed to and the distance is still unknown. Likewise, the array $prev$ is initialized with no values for each of the vertices. $prev$ will eventually contain directions for the shortest path to V_{start} by indicating which of the previous neighbors of any given node have to be traversed through to get to that node. Each vertex is emplaced in the queue, Q and the distance from the start for V_{start} is updated to 0.

The main loop of the code starts at line 8. In this example, it is implemented as a while loop set to run until Q is empty, but a break condition is implemented in line 19 if the goal node is found. For a more generalized version of Dijkstra's algorithm, the break condition at line 19 would be excluded. This would allow the while loop to iterate over every node in Q. However, since only path between V_{start} and V_{goal} is desired, searching beyond the goal is a computational wasteful task, especially as the scope of the search space widens.

In the while loop itself, the node selected to be searched and expanded is chosen by picking the node in Q that has the minimum distance (line 9). In the case of the first iteration, the only vertex that has a distance value that is not infinity is V_{start}, which has a value of 0. The vertex, u, is then removed from Q, and expanded such that each of its neighbors, n, are found. The distance to each neighboring node via u is then calculated by adding the current distance to u, $dist[u]$, to the edge length between u and n using $edge\_length$ (line 13). $edge\_length$ can be a calculated value or some predetermined value using prior knowledge of the graph. For the former, it could be as simple as finding the Euclidean or Manhattan distance between the two vertices. For the latter, the edge values could represent some real-world metric that is numerically quantified. For example, if the edge between the vertices u and n represent a hilly walking trail in the real-world, the edge length could be given a value of 5. However, if the path is a paved, flat walking trail, the edge length could be given a value of 2 because the time it takes to traverse would be shorter. If $dist\_check$ is less than the current distance value for n, then $dist[n]$ and $prev[n]$ are updated (lines 15 and 16, respectively).

The while loop can be broken out of in two ways, checking all nodes in Q or with the previously mentioned condition of finding the node that is the goal node. If the latter condition is never met and $prev[V_{goal}]$ returns $empty$, then there is no solution for a path between the start and the goal nodes.

The output of the while loop is the array $prev$, which at this point contains back-pointers to the shortest node. The path will have to be reconstructed using an algorithm such as $construct\_path$, given in Algorithm 2. This algorithm checks if the goal node is found in line 4. If it was, then the path is reconstructed from the goal, backwards until the starting node is found.

A quick example on how exactly Dijkstra's algorithm can be seen starting in Figure 2.7. In this graph, there are four nodes, V_A, V_B, V_C, and V_D. The starting node, V_{start}, is A and indicated by the red square and the goal node, V_{goal}, is indicated by the green square. Each edge is indicated by the orange arrows with the cost of traversing each edge also shown.

Algorithm 1 Dijkstra's Algorithm [28]

Require: $graph$, V_{start}, V_{goal}

1: **for** each $Vertex$, V, in $graph$ **do**
2: $dist \rightarrow \infty$
3: $prev \rightarrow []$
4: emplace V in Q.emplace(V)
5:
6: $dist[V_{start}] = 0$
7:
8: **while** Q is **not** empty **do**
9: $u = Q.min(dist(V))$
10: $Q.remove(u)$
11:
12: **for** each neighbor, n, of u **do**
13: $dist\_check = dist[u] + edge\_length(u, n)$
14: **if** $dist\_check < dist[n]$ **then**
15: $dist[n] = dist\_check$
16: $prev[n] = u$
17:
18: **if** $n = V_{goal}$ **then**
19: **Exit**
20:
21: $path = construct\_path(prev, V_{start}, V_{goal})$
22:
23: **return** $path$

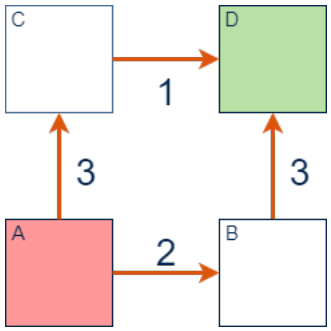


Figure 2.7: Dijkstra's Example

In Algorithm 1, lines 2- 4 are represented by Figure 2.8. Each node is added to $dist$ with the associated value of infinity, $prev$ with an empty back-pointer, and to the queue, Q.

Algorithm 2 Construct Path

Require: $prev$, V_{start}, V_{goal}

1: $u = prev[V_{goal}]$
2: $path = []$
3:
4: **if** u **then**
5: $path = u$
6: **while** u is **not** V_{start} **do**
7: $u = prev[u]$
8: $path.push\_front(u)$
9:
10: **return** $path$

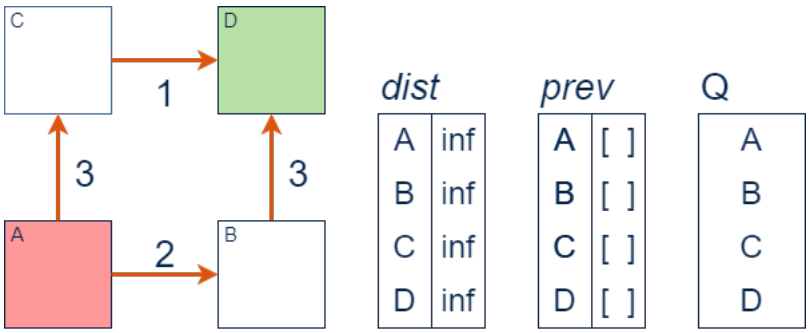


Figure 2.8: Dijkstra's Example - Step 1

Lines 6 and 9 are shown in Figure 2.9. The starting node distance is changed to 0 in $dist$. Then V_curr is selected to be V_A because it is the node in Q with the lowest associated $dist$ value.

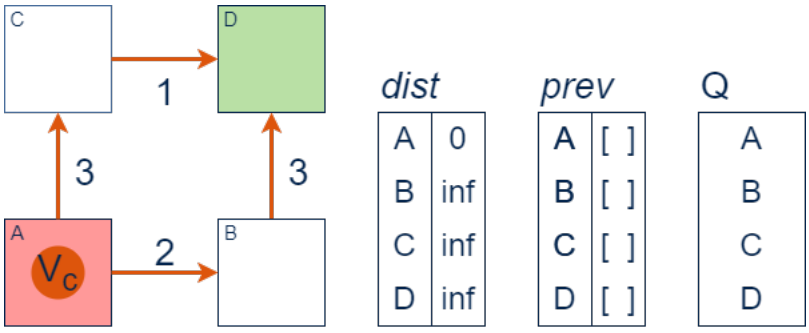


Figure 2.9: Dijkstra's Example - Step 2

V_A is then expanded and found to have two neighbors, V_B and V_C and then is removed from Q. V_B is arbitrarily chosen to be the first neighbor to be explored, indicated by the purple square in Figure 2.10. $dist\_check$ is then calculated, as in line 13, and is found to be equal to 2

units since $dist[V_A]$ and $edge\_length(V_A, V_B)$ have values of 0 and 2 units, respectively. Since $2 < inf$, the value for $dist[V_B]$ is updated to be 2 and the value of $prev[B]$ is updated to be V_A.

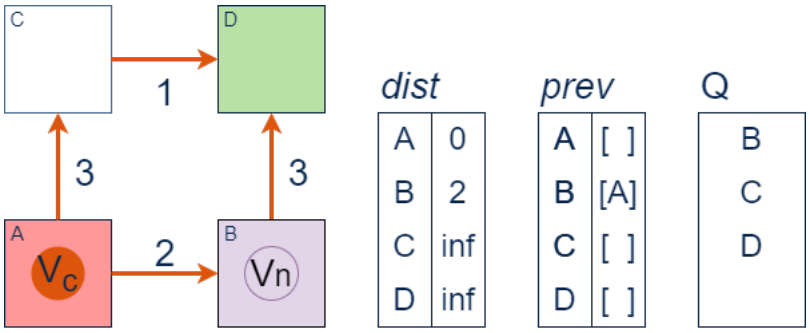


Figure 2.10: Dijkstra's Example - Step 3

Next, V_C by way of V_A is explored and the its respective values in $dist$ and $prev$ are updated to be 3 and V_A (Figure 2.11). Using the current values in the $dist$ array, the shortest path to get from the starting node to V_B is 2 units which can accessed by way of V_A.

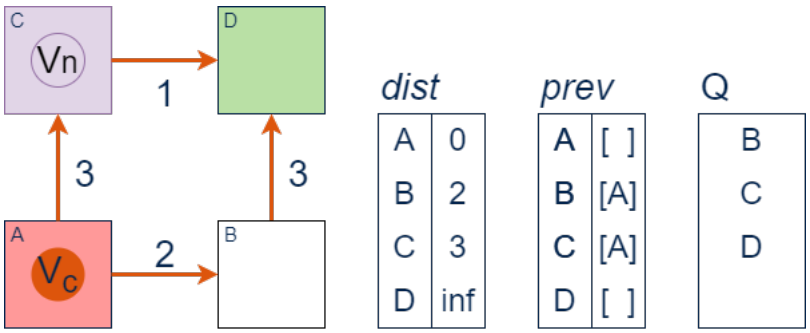


Figure 2.11: Dijkstra's Example - Step 4

Now that both neighbors of V_A are explored, the node is completed and the next V_curr can be selected. This next node being V_B since it has the current lowest $dist$ value of 2 units (Figure 2.12). V_B only has one edge leading out, V_D. $dist\_check$ is calculated by $dist[V_B] + edge\_length(V_D, V_B) = 2 + 3 = 5$ units.

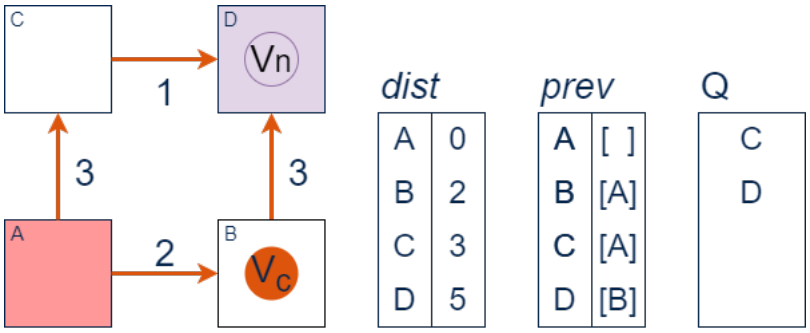


Figure 2.12: Dijkstra's Example - Step 5

Next node to be searched is V_C, whose only has one edge leading away and is also V_D (Figure 2.13). Now, performing the $dist\_check$ results in 4 units. Comparing it to the value $dist[V_D]$ is a little bit different this time. In previous comparisons, the distance values were always infinity since it was the first time encountering the node. However, now $dist[V_D]$ holds the value of 5 units. This is less than the calculated $dist\_check$ for this step and means that a new path to V_D has been found and is shorter than the one previously stored. $dist[V_D]$ is updated to be 4 units and $prev[V_D]$ is updated to be V_C. Now, the shortest path to V_D is by way of V_C.

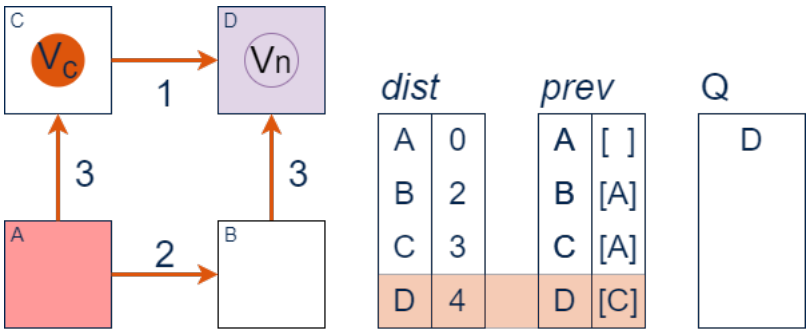


Figure 2.13: Dijkstra's Example - Step 6

The final node left in Q is V_D, which has no neighbors leading out of it but is also the goal node. Either condition would allow for the exit of the while loop. The last step is seen in line 21, reconstructing the path using the output matrix $prev$. This would result in a path shown in 2.14, with a final path of: $V_A \rightarrow V_C \rightarrow V_D$.

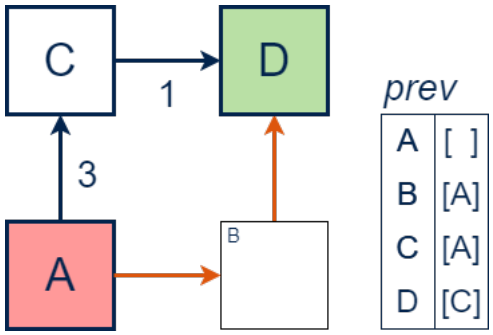


Figure 2.14: Dijkstra's Example - Solution

Dijkstra's simplicity and ease of implementation makes it a good starting point for creating a path planner for a robot, autonomous vehicle, or otherwise, like in [31]-[32]. However, because of the algorithms greedy nature, there have been numerous attempts on adapting the algorithm to improve speed while still maintaining optimality. For example, attempts to combine Dijkstra with fuzzy logic to find a shortest path in a partially uncertain environment is presented [33]. Alternatively, research argues that as distance between the start and goal node increases, the computation time becomes too great to run in real-time [34]. The paper proposes an alternative approach that includes starting the search from both the start and goal nodes and continuing to iterate until the search spaces begin to overlap. That proves to greatly reduce the search time as the distances between the two nodes increases.

A* Algorithm

The A* (or A-Star) algorithm was first proposed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael [35]. It is often seen as an extension of Dijkstra's algorithm, but is capable of finding a solution more efficiently assuming that certain parameters are tuned correctly. Where Dijkstra's algorithm could run regardless of if information about the goal is known *a priori*, A* requires and uses information about the goal when choosing the nodes it explores in order to minimize the total number of nodes that need explored and expanded. This in turn could potentially reduce the run-time of the planner. This knowledge and use of the goal node gives A* the property of being an *informed* algorithm, also known as best-first search [25].

A* attempts to minimize the cost function given in Equation (2.1).

$$f(n) = g(n) + h(n) \tag{2.1}$$

where $g(n)$ is the cost from the starting node to the node being explored and $h(n)$ is the *heuristic* function that estimates the distance from the current node to the goal. The heuristic function is the key to an optimal and efficient A* algorithm.

There are several common ways of estimating the heuristic in low-dimensional problems. They include the *Euclidean*, *Manhattan*, or *Chebyshev* distances seen in Figure 2.15. The two-dimensional Euclidean distance is found with Equation (2.2).

$$dist_{euclidean} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.2}$$

The two-dimensional Manhattan takes the sum of the absolute value of the distance of each coordinate (ex. x and y) as seen in Equation (2.3).

$$dist_{manhattan} = |x_1 - x_2| + |y_1 - y_2| \tag{2.3}$$

Lastly, the Chebyshev uses the largest difference along an axes as the distance value as seen in Equation (2.4).

$$dist_{chebyshev} = max(|x_1 - x_2|, |y_1 - y_2|) \tag{2.4}$$

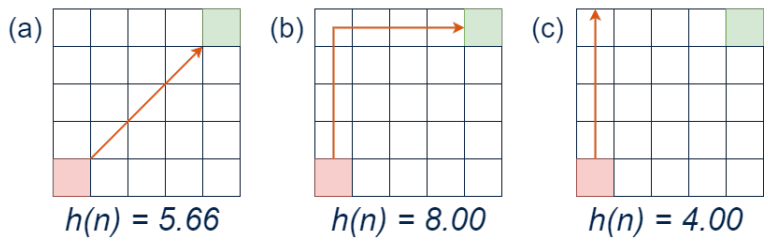


Figure 2.15: (a) Euclidean, (b) Manhattan, (c) Chebyshev distance

Selecting a "bad" heuristic may still return a path, but that path might be aub optimal path and the calculation time taken may be longer than an otherwise better heuristic. A bad heuristic

is one that returns a value that is *more* than the actual distance between the current and goal nodes. The heuristic is considered *admissible* if it always returns a value that is less than or equal to the cost of the actual shortest path from the current node to the goal. In this case, A* is said to be optimal. One special case for A* is if $h(n) = 0$ for all n, then it can be viewed as another case of Dijkstra's algorithm and will perform no better.

An example of pseudo-code for A* can be seen in Algorithm 3.

Algorithm 3 A* Algorithm

Require: $graph$, V_{start}, V_{goal}

let Q be sorted by $min(cost)$

for each $Vertex$, V, in $graph$ **do**
    $visited \rightarrow false$
    $prev \rightarrow []$
    $cost \rightarrow \infty$

$cost[V_{start}]$ = 0.0
$Q.emplace(V_{start}, cost[V_{start}])$

while Q is **not** empty **do**
    $V_c = Q.top()$
    $V_c = Q.pop()$

    for each neighbor, V_n, of V_c **do**
        $cost_{next} = cost(V_c) + edge\_cost(V_c, V_n)$

        if $cost_{next} < cost(V_n)$ **then**
            $prev[V_n] = V_c$
            $cost[V_n] = cost_{next}$
            if !$visited[V_n]$ **then**
                $Q.emplace(V_n, cost[V_n] + heur(V_n))$
                $visited[V_n] = true$

    if $V_c = V_{goal}$ **then**
        exit

$path = construct\_path(prev, V_{start}, V_{goal})$

return $path$

Similar to Dijkstra's algorithm, A* initializes several arrays $visited[V] = false, prev[V] = [\,]$, and $cost[V] = inf$ for each vertex, V, in the graph (lines 4-6). $prev$ serves the same function as it did in Algorithm 1 while the $dist$ array that tracks the shortest path from source and the $cost$ array tracks the cost of that node. The $dist$ is used as the estimate of the shortest potential path *if* the path were to go through that particular node. The $cost$ array is the combination of the shortest calculated cost from the goal node and the estimated heuristic to the goal node. The $visited$ tracks which nodes have been explored.

Also similar to Dijkstra's, A* initially sets $cost[V_{start}] = 0.0$ and emplaces it into Q. For this algorithm, Q is a *minimum-priority* queue, meaning that it sorts the entries from least to greatest by some metric. Which in this case is the cost that is emplaced with the node. After initialization, the algorithm enters the main while loop that continues to run until either Q is empty or the goal node is found (line 25).

The first step each loop is to pick the next node from the top of the queue, which is also the node with the lowest cost (line 12). The node is then *popped*, or removed, from Q (line 13). The current node chosen, V_c is then expanded and its neighbors, $V_{n,i}$, are found. Each neighbor is then is explored. The potential next cost, $cost_{next}$, is calculated in line 16 and is the combination of the shortest path from the start to V_c and the $edge\_cost$, or the cost to traverse from V_c to V_n. Like previously stated, the $edge\_cost$ does not have to be the physical distance from the start to the node, rather its the cost of traversing the specific edges. These costs can be impacted by various factors such as proximity to objects, path conditions, or any other quantifiable metric.

If $cost_{next}$ is less than the current value of $cost[V_n]$, then $prev[V_n]$ and $cost[V_n]$ are updated to be V_c and $cost_{next}$, respectively, in lines 19 and 20. If V_n had not been previously visited, then it is emplaced into Q and marked $true$ in $visited$. However, the cost value emplaced in Q is not just $cost_{next}$. It is the sum of both $cost_{next}$ and the heuristic cost, $heur(V_n)$ (line 22). By including the heuristic in the cost inserted in the queue, the queue will begin to prioritize searching nodes that tend towards the goal. Essentially, if the goal is in "front" of the start, the heuristic will keep the algorithm from searching in too many nodes that are "behind" the start.

If V_c is the goal node, the while loop is exited, else it will continue to iterate until it does eventually find the goal or runs out of nodes to explore. Finally, upon exiting the while loop, the algorithm constructs the path using the $prev$ array.

A step-by-step of example A* starts at Figure 2.16. Similar to the previous example, the start and end nodes are indicated by the red and green squares. The edge costs and directions are also indicated on this graph.

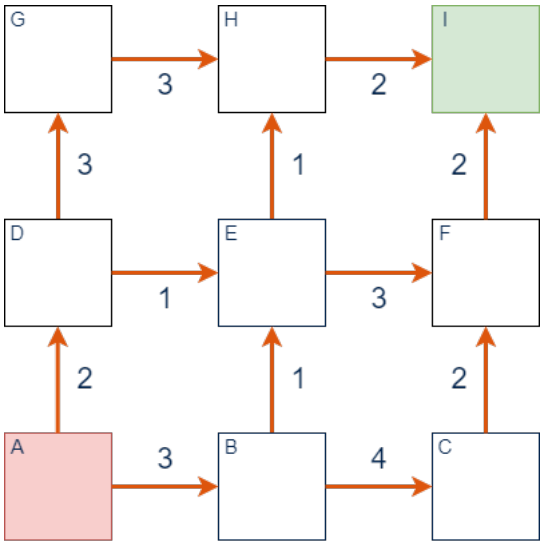
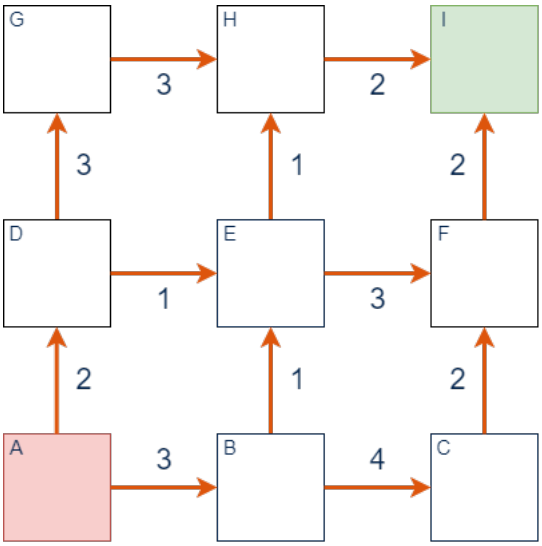


Figure 2.16: A* Example

In Figure 2.17, the $visted$, $prev$, and $cost$ array are initialized to their starting values like in lines 4-6. Also shown in this step is V_{start}, V_A, being emplaced in Q along with the cost value of 0.0 in second column (line 9).



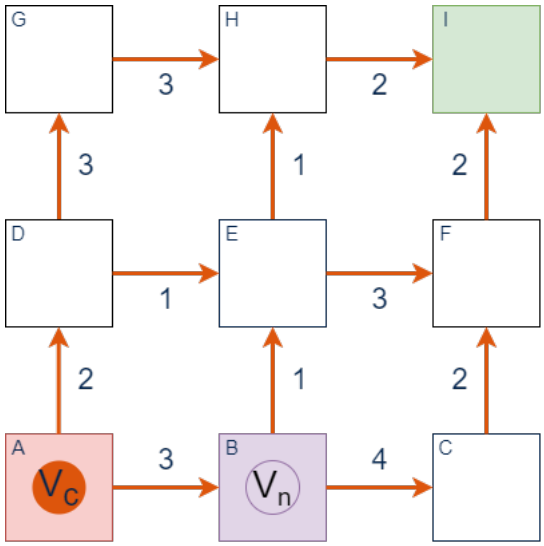
	visited	prev	cost
A	true	[ ]	0
B	false	[ ]	inf
C	false	[ ]	inf
D	false	[ ]	inf
E	false	[ ]	inf
F	false	[ ]	inf
G	false	[ ]	inf
H	false	[ ]	inf
I	false	[ ]	inf

Q	
A	0.0

Figure 2.17: A* Example - Step 1

Since V_A is the only node in Q, it is first selected as V_c. V_A is expanded and its neighbors are found with V_B arbitrarily being chosen as the first one to investigate. $cost_{next}$ is calculated in line 16 and is the sum of the cost to travel to V_A plus the $edge\_cost$, which in this case is 0 and 3, respectively. The calculated $cost\_next$ is greater than the value $cost[V_B]$ and so both $cost[V_B]$ and $prev$ are updated to new values. Since V_B has also not been visited yet, its status is updated in $visited$ and it is emplaced in Q. One thing to note, the value that it is placed in is not the same value that is in $cost$. Rather, it is the value in *cost plus* the heuristic, as seen in line 22.

In this example, the heuristic function chosen is the Euclidean distance between the node and the goal. Each square is estimated to be, at best, 1 unit away from its neighbors, so the $heur(V_B, V_{goal}) = \sqrt{1^2 + 2^2} = 2.24$ units. If we were to use the Manhattan distance, the heuristic function would result in 3 units while the Chebyshev distance would be 2 units.



	visited	prev	cost
A	true	[ ]	0
B	true	[A]	3
C	false	[ ]	inf
D	false	[ ]	inf
E	false	[ ]	inf
F	false	[ ]	inf
G	false	[ ]	inf
H	false	[ ]	inf
I	false	[ ]	inf

Q	
B	5.24

Figure 2.18: A* Example - Step 2

Something to note, it was previously stated that the heuristic function had to result in an *optimistic* estimation. That would imply that in this case, the Manhattan distance would not be admissible because it has a greater value than the Euclidean distance. However, looking closer, the maximum number of neighbors to or from a node is four. This is similar to a 4-connected graph, or a *von Neumann* neighborhood [36]. With this type of connectivity, diagonal movement is not allowed, only movement to a node that shares an edge is allowed. In this type of edge configuration, the shortest path from a node to the goal is actually the Manhattan distance, given that all edge lengths are greater than or equal to 1. Therefore, the Manhattan distance is admissible in this particular case. An example of where it wouldn't be admissible can be seen below in Figure 2.19(b) for an 8-connected neighborhood.

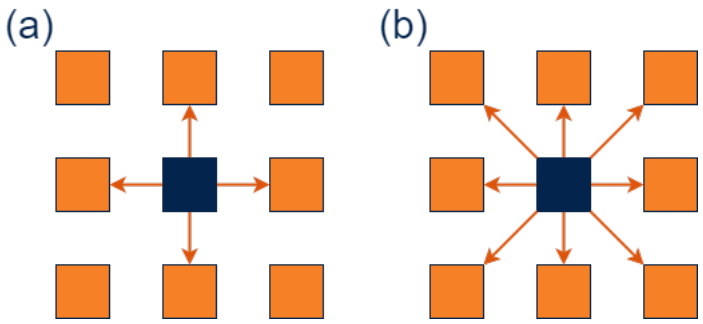
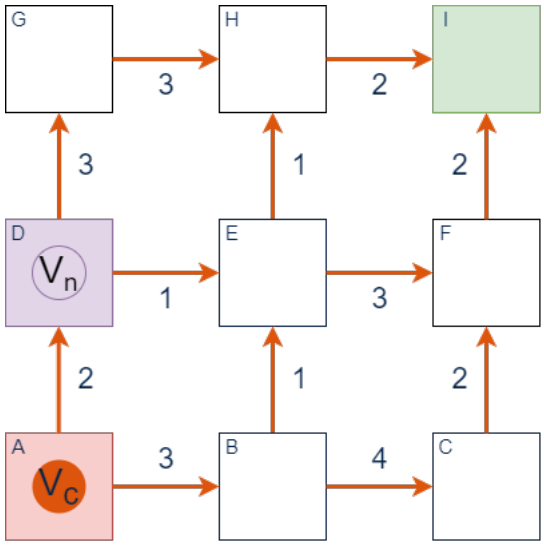


Figure 2.19: (a) 4-connected (b) 8-connected

Next, V_D is investigated. For this node, the cost to traverse is found to be only 2 units while the heuristic function also returns 2.24 units. Since this node has also not been visited prior to this, its $visited[V_D]$ is updated as well as being emplaced in Q. This time $cost[V_D] + heur(V_D, V_{goal}) = 4.24$ which is greater than the current queue cost of V_B, V_D is placed ahead of V_B, as seen in Figure 2.20.

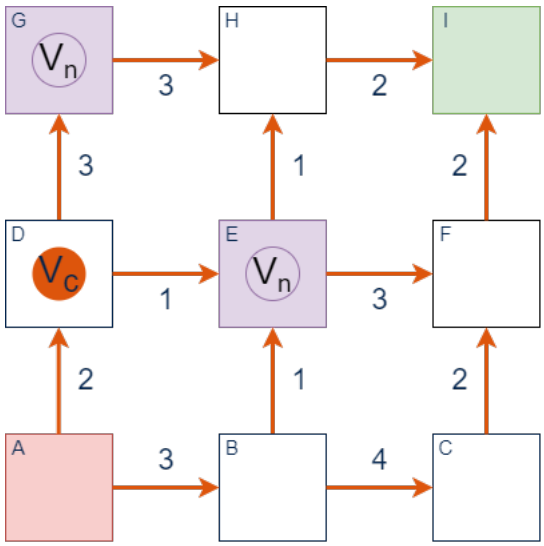


	visited	prev	cost
A	true	[ ]	0
B	true	[A]	3
C	false	[ ]	inf
D	true	[A]	2
E	false	[ ]	inf
F	false	[ ]	inf
G	false	[ ]	inf
H	false	[ ]	inf
I	false	[ ]	inf

Q	
D	4.24
B	5.24

Figure 2.20: A* Example - Step 3

In Figure 2.21, it is shown that V_D is selected as the next V_c. Its expanded and its neighbors, V_E and V_G, are explored for the first time and emplaced in the queue.

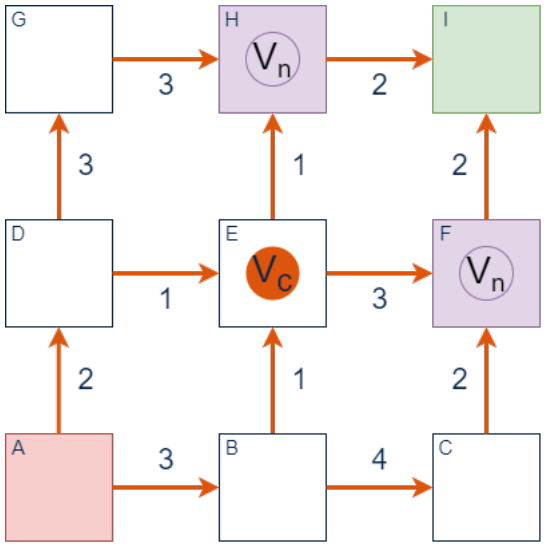


	visited	prev	cost
A	true	[ ]	0
B	true	[A]	3
C	false	[ ]	inf
D	true	[A]	2
E	true	[D]	3
F	false	[ ]	inf
G	true	[D]	5
H	false	[ ]	inf
I	false	[ ]	inf

Q	
E	4.41
B	5.24
G	7.00

Figure 2.21: A* Example - Step 4

Continuing another iteration, V_E is next in Figure 2.22.

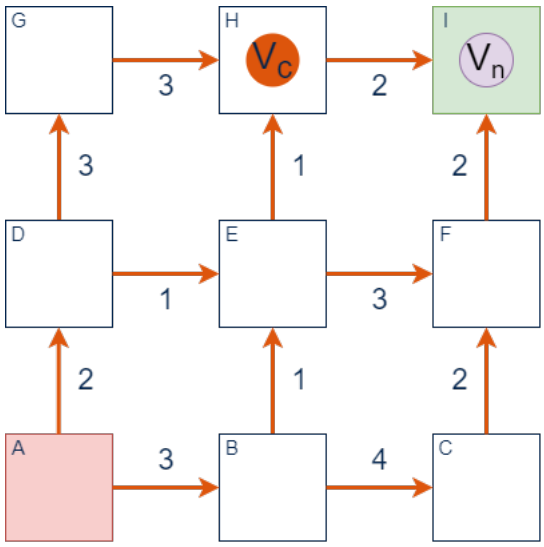


	visited	prev	cost
A	true	[ ]	0
B	true	[A]	3
C	false	[ ]	inf
D	true	[A]	2
E	true	[D]	3
F	true	[E]	6
G	true	[D]	5
H	true	[E]	4
I	false	[ ]	inf

Q	
H	5.00
B	5.24
G	7.00
F	7.00

Figure 2.22: A* Example - Step 5

In Figure 2.23, V_H is shown to only have one neighbor, V_I, which also happens to be the goal node. However, the while loop will continue another iteration because in order to break out of the while loop with goal condition, V_c, not V_n, must equal V_{goal}.

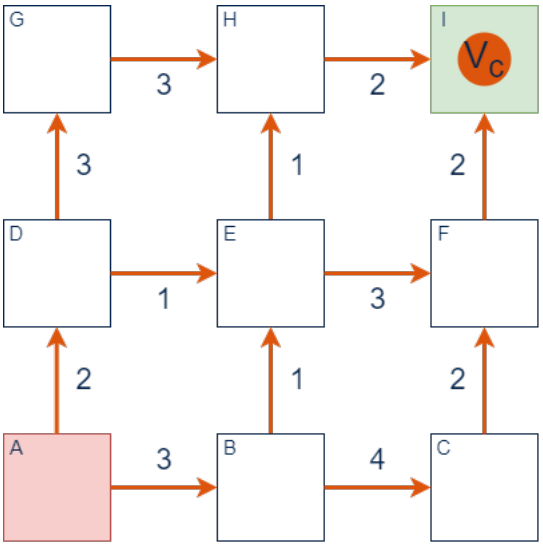


	visited	prev	cost
A	true	[ ]	0
B	true	[A]	3
C	false	[ ]	inf
D	true	[A]	2
E	true	[D]	3
F	true	[E]	6
G	true	[D]	5
H	true	[E]	4
I	false	[H]	5

Q	
I	5.00
B	5.24
G	7.00
F	7.00

Figure 2.23: A* Example - Step 6

Finally, $V_c = V_{goal}$ and the while loop is ended before exhausting the queue.

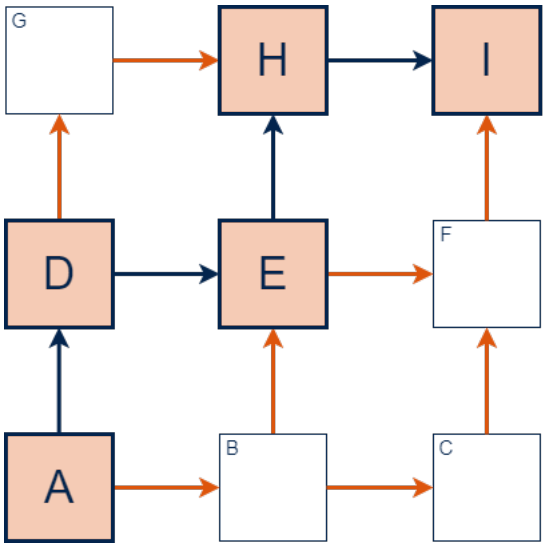


	visited	prev	cost
A	true	[ ]	0
B	true	[A]	3
C	false	[ ]	inf
D	true	[A]	2
E	true	[D]	3
F	true	[E]	6
G	true	[D]	5
H	true	[E]	4
I	false	[H]	5

Q	
B	5.24
G	7.00
F	7.00

Figure 2.24: A* Example - Step 7

Using $prev$, the path can be reconstructed like in Figure 2.25.



	prev
A	[ ]
B	[A]
C	[ ]
D	[A]
E	[D]
F	[E]
G	[D]
H	[E]
I	[H]

Figure 2.25: A* Example - Step 8

Performance of A* vs Dijkstra's Algorithm

Figure 2.26(a) highlights the nodes that were expanded before the goal was found. It happens that in this particular scenario that the algorithm explored the nodes that were to become the final path, but that is not always guarunteed as the search space grows or objects are introduced.

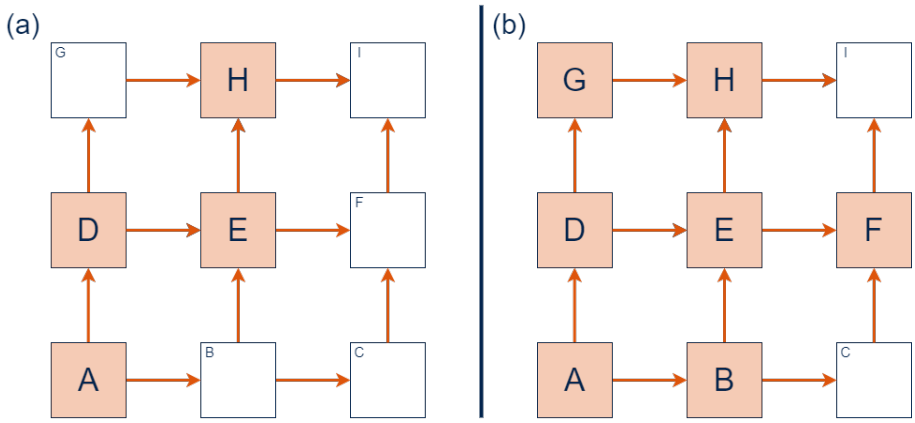


Figure 2.26: Comparing the visited nodes for A* (a) and Dijkstra's Algorithm (b)

The nodes that would have been searched if Dijkstra's algorithm were run for the same graph are shown in Figure 2.26(b). In terms of efficiency, A* searched 3 less nodes than Dijkstra's. While that might not seem like a lot, the efficiency of A* really shines the larger the search space and the farther apart the start and goal nodes are. For example, in Figure 2.27, a 10 by 10 grid is shown with 20 nodes that are not reachable denoted with the black squares. Both Dijkstra's algorithm and A* was used to find the shortest path between the blue (start) and orange (goal) nodes. However, in Figure 2.27(a), 77 of the 80 searchable nodes were searched while only 22 were searched in Figure 2.27(b) for the same configurations.

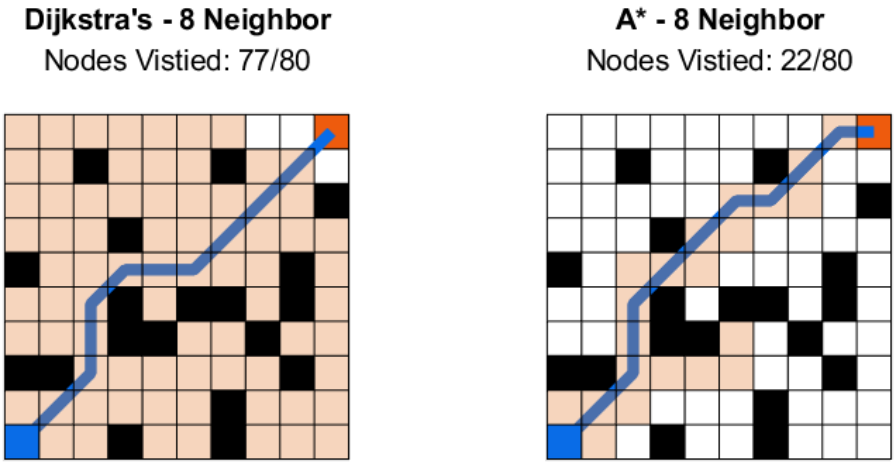


Figure 2.27: (a) Dijkstra vs (b) A*

Both algorithms seem to start there search similarly, with the three nodes neighboring the starting node are all explored. However, as both algorithms continued, A* began to explore nodes that tended nearer the straight line path from the start to goal node. In the beginning,

Table 2.1: Dijkstra vs A* Performance

Graph Size	# of Nodes	# of Objects	Dijk. Time (ms)	A* Time (ms)	Diff	Dijk. Visited	A* Visited	Diff
5 x 5	25	5	1.70	1.10	1.55	15	6	2.47
10 x 10	100	80	6.23	2.27	2.74	76	19	3.92
15 x 15	225	45	13.7	3.88	3.53	317	70	4.51
20 x 20	400	80	26.5	7.51	3.53	317	70	4.51
25 x 25	625	125	42.7	9.98	4.28	497	91	5.44
50 x 50	2,500	500	226	50	4.56	1,997	391	5.10
75 x 75	5,724	1,125	940	133	7.05	4,496	678	6.63
100 x 100	10,000	2,000	2,986	533	5.60	7,996	1,491	5.36

when the algorithm comes to the L-shaped set of objects, it begins exploring both paths around because they both appear to be valid solutions. As it goes on, the path to the right ultimately continues to run into more objects, which would push it farther and farther right while the path upwards eventually finds a hole in the objects and is able to continue towards the goal, lowering the output of the heuristic function.

Table 2.1 represents the results of 100 simulations of increasingly larger graph sizes. Each graph had a fifth of its nodes randomly selected as objects and each of the algorithms were implemented for the same graph configurations. In these simulations, the graphs are 8-connected, and so diagonal movement is allowed. For smaller graphs, like the 5x5, the performance difference is subtle. A* performs 1.55x faster than Dijkstra while searches 2.47 fewer nodes.

Revisiting the role of the heuristic function and its affects on the algorithm, Table 2.2 shows the results of simulations with changing values of the heuristic. The simulations are set up similar to the one described previously for Table 2.1. In this simulation, the Euclidean distance is used as the heuristic function and would be considered admissible because the edge lengths are never less than 1. However, since these graphs are 8-connected, using the Manhattan distance as the heuristic would not be considered ”optimistic” because the shortest path could be shorter than the result of that function due to the diagonal travel.

For this set of simulations, the calculated value of the heuristic is scaled by various values seen in the first column. For example, if the heuristic results in a value of 3.4 units using the Euclidean distance and the scaling is 0.75, the heuristic function would actually return 2.55 units. As the the scaling decreases, the number of nodes visited increases, with A* only being

Table 2.2: Dijkstra vs A* Performance with Scaled Heuristic

Heuristic Scaling	**Dijk. Visited**	**A* Visited**	**Diff**
1.00	396	139	2.84
0.90	396	203	1.95
0.80	396	253	1.56
0.75	396	273	1.45
0.50	396	351	1.13
0.25	396	395	1.00
0.10	396	396	1.00
0.00	396	396	1.00

able to maintain a higher efficiency with heuristic scaling of about 50%. The two nodes converge and perform identically from 25% and lower. This corresponds with the earlier statement that if $h(n) = 0$, A* is functionally no better than Dijkstra's.

To better visualize what is happening as the scaling goes down, Figure 2.27 shows the comparison of the various scalings of a 20x20 grid with about a fifth of the nodes treated as objects. Figure 2.28a and 2.28b show the results of running Dijkstra's algorithm and A* just as before. A* appears to have worked very efficiently, finding the goal with exploring much fewer nodes. In the next pane, Figure 2.28c, the estimation of the heuristic is 90% of the Euclidean distance. This causes the visited nodes to fan out more because the algorithm wrongly prioritizes nodes it believes to be closer to the goal. Continuing through the panes, Figures 2.28d-2.28i, the number of nodes continues to increase until the visited nodes begins to look like result of the Dijkstra's graph.

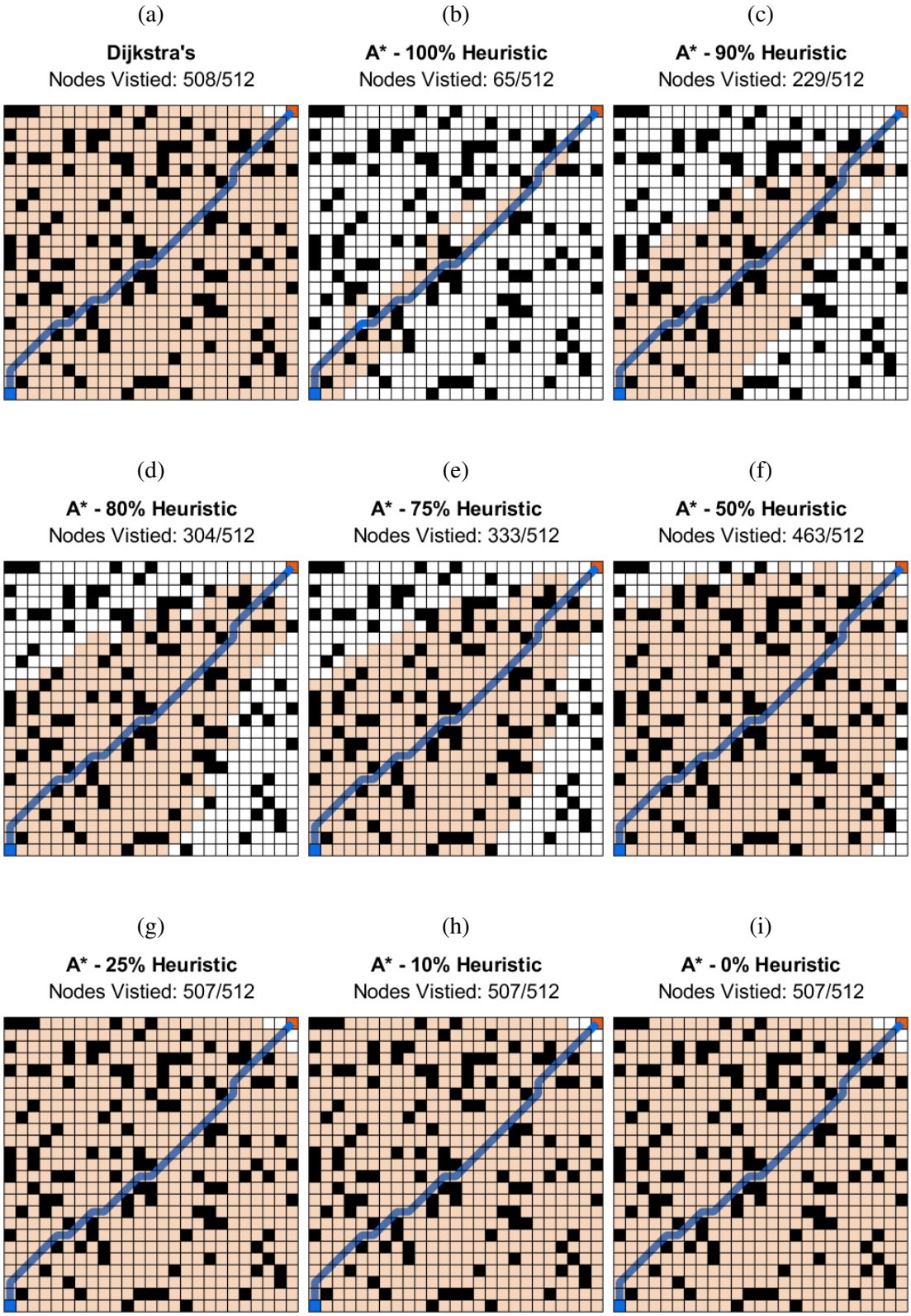


Figure 2.28: Effects of A* Heuristic Function

A* Variations, D*, and Others

The simplicity and versatility of A* has led to many derivatives since its first introduction. These derivatives include trying to improve various properties of A* such as computation time or node-expansion efficiency. Weighted A* adapts the queue cost calculation in Algorithm 3 Line 22 with Equation (2.5).

$$f_{weighted}(n) = g(n) + w * h(n), w > 1 \tag{2.5}$$

In general, the this would speed up finding a solution with a trade off that the solution may not be the most optimal, or the shortest path [37].

Another class of A* derivatives is the Anytime A* family, which includes *Anytime Weight A** [38], *Anytime Weighted A** [39], and *Anytime Repairing A** [40] among others. The main characteristics of these algorithms is they trade off optimality for efficiency in order to have a solution *anytime*. For the most part, the algorithms are designed to quickly find a solution then iterate over the algorithm by incrementally changing a parameter resulting in a longer computation time but more optimal solution. However, the algorithm could be terminated at any time, using the most recently computed path as its solution. Algorithms like this would most likely shine in environments with lower computation ability, rapidly changing environments, or some combination of both.

Other derivatives of A* take a different approach when adapting the original algorithm in that they are designed for situations where there is some understanding of the graph during the initialization of the algorithm, but can handle unknowns such as unexpected objects. Lifelong Planning A* (LPA*) is one such algorithm introduced by Koenig as an incremental version of A* that reuses parts of the graph that are identical in order to save time re-planning due to changing edge costs or nodes, among other things [41]. An example of the LPA* algorithm can be seen in Algorithm 4. Note, that oftentimes other graph search methods, such as A* are used to find the initial path and various costs.

Like previous algorithms, LPA* initializes a few arrays, g and rhs, with values of infinity for each vertex. The g array is similar to array in A* that tracks the distance from the start to the

current node. However, the array rhs is an array that has not been used before and represents one-step look-ahead values that are based on g values and could be better informed.

Also differing in this algorithm is how the queue, Q, is sorted. Rather than cost like in A*, it is sorted by the key. However, a closer look at what $calcKey(v)$ (line 1) returns shows that it returns two values, $k(v) = [k_1(v), k_2(v)]$ where $k_1 = min(g[v], rhs[v] + heur(v, V_{goal})$ and $k_2 = min(g[v], rhs[v])$. Q is sorted by k_1 and then by k_2. For example, for two keys, $K_a = [k_{a,1}, k_{a,2}]$ and $K_b = [k_{b,a}, k_{b,2}]$, K_a will always be smaller if $k_{a,1} \leq k_{b,1}$ or if $(k_{a,1} = k_{b,1}, k_{a,2} \leq k_{b,2})$.

The algorithm then enters the main loop in lines 33-37 that would run until stopped. At every iteration, the shortest path is constructed using $GetShortestPath$ function in lines 14-23. Essentially, the function does the same thing as Algorithm 4 by starting at the goal vertex and working backwards and checking the shortest paths from the nodes *predecessors*. This process is repeated until the start vertex is found.

Between iterations of the loop, there may be changes in the edge costs. If that is the case then each edge, composed of u (start) and v vertices, is iterated and its costs are updated and $UpdateVertex(v)$ is called. In $UpdateVertex$ (line 4), each of the *predecessor* vertices rhs values are updated in line 8. Then the vertex is checked whether it is in Q and removed if it is (line 9. Finally, if the g and rhs values do not matched, the vertex is emplaced in Q and a new key is calculated reflecting updated values.

While the first iteration of LPA* runs the same as A*, the strengths of the algorithm comes by reducing the re-planning times for subsequent iterations because the algorithm is able to reuse results from previous searches and thus reduces the amount of nodes that need to be explored [42].

Other popular and often used derivatives of A* that can *repair* the graph due to unforeseen changes, often classified as *dynamic* variants, include:

- *D** [43] - Orginially proposed by Anthony Stentz, assumes some *apriori* knowledge of the graph and the search space and initially plans a shortest path between the start and goal. However, as the path is traversed and new graph information is uncovered (such as unknown objects), the algorithm re-plans from the current cell to the goal. D* is often

Algorithm 4 LPA* Algorithm

Require: $graph$, V_{start}, V_{goal}

CalcKey(v**)**
return $key\_cost = [min(g[v], rhs[v] + heur(v, V_{goal}), min(g[v], rhs[v])]$

UpdateVertex(v)
if $v! = V_{start}$ **then**
    $rhs[v] = \infty$
    for each predecessor, $pred$ of v **do**
        $rhs[pred] = min(rhs[v], g[pred] + EdgeCost[pred, v])$
    if $Q.isIn(v)$ **then**
        $Q.remove(v)$
    if $g[v]! = rhs[v]$ **then**
        $Q.emplace(v, CalcKey(v))$

GetShortestPath()
while $Q.top().key < CalcKey(V_{goal}) || rhs[V_{goal}]! = g[V_{goal}]$ **do**
    $V_c = Q.pop()$
    if $g[V_c] > rhs[V_c]$ **then**
        $g[V_c] = rhs[V_c]$
    else
        $g[V_c] = \infty$
        $UpdateVertex(V_c)$
    for each successor, $succ$, of V_c **do**
        $UpdateVertex(succ)$

Main()
let Q be sorted by key
for each Vertex, V, in $graph$ **do**
    $g \rightarrow \infty$
    $rhs \rightarrow \infty$
$rhs[V_{start}] = 0$
$Q.emplace(V_{start}, CalcKey(V_{start}))$

while true **do**
    GetShortestPath()
    for each edge, $[u, v]$, with changed costs **do**
        $cost[u, v] = newEdgeCost(u, v)$
        $UpdateVertex(v)$

seen as a fairly complicated algorithm that is hard to implement, with the potential need for a large amount of memory storage being one of the main drawbacks [42].

- *Focused D** [44] - By the original author of D*, Stentz further improved the algorithm by introducing the Focused D* algorithm. This algorithm is an informed heuristic search algorithm that was proven to run faster than the original D* by several orders of magnitude. This is accomplished by focusing updates of the graph to only nodes that matter.

- *D* Lite* [45] - Rather than being based on D*, D* lite draws its inspiration from LPA*. When implemented, the algorithm explores fewer nodes than LPA* and thus is able to run faster than its predecessor.

2.1.2 Sampling Based Algorithms

Like its name suggest, sampling based planning is a type of planning algorithm that samples the search space as opposed to discretization. In general, sampling is able to converge on a solution very quickly for more complex problems such as paths in high dimension spaces. For example, a UAV has a minimum of 6 degrees of freedom: $x, y, z, \phi, \theta, \psi$. In a 2D scenario where there were 100 vertices in a graph, the 6D case would require 1,000,000 vertices [46]. In fact, increasing a graph by *n*-dimensions increases the number of vertices by 10^n. For this reason, sampling based planning techniques are often implemented for problems with 5 or more dimensions.

However, one of the drawbacks to sampling based algorithms is that they cannot be considered to be *complete*. To be *complete*, an algorithm will return a solution, if one exists, in a finite amount of time [15]. Algorithms such as A* and D* are considered complete, because given enough time and it exists, a solution will always be found. In certain edge cases, that solution may require searching the entire graph, but the algorithm will always converge. The same can not be said of sampling based algorithms and because of this a weaker idea of completeness is tolerated when considering various sampling based algorithms. Instead, an algorithm can be considered to be *probabilistically complete*. Which means that given enough

time, and assuming a solution exists, it will be found. This is a key metric to assess this quality being how quickly can the algorithm converge to a solution.

Randomized Potential Planner (RPP)

One of the earliest sample based methods developed to better solve the more complex problems that discrete planning could not is the *randomized potential fields* or the *randomized potential planner* [47]. The algorithm can best explained with Figure 2.29.

In general, the algorithm uses a *potential function* as a type of psuedo-heuristic or metric for determining the distance to the goal. The potential function generally has an *attractive* term that is related to the distance to the goal and a *repulsive* term that is related to the proximity to objects. However, unlike the heuristics described earlier, this function does not need to be optimistic, meaning it is allowed to overestimate.

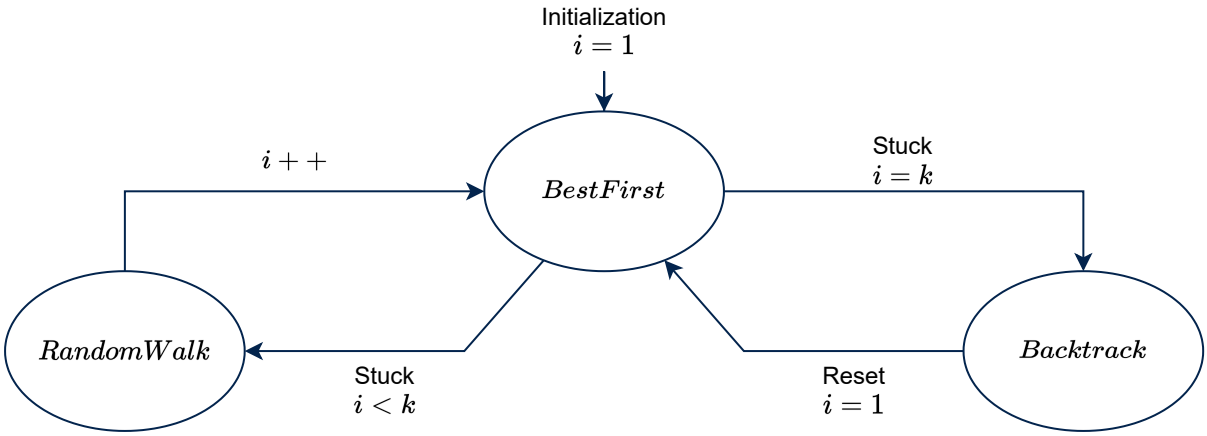


Figure 2.29: RPP State Graph

At initialization, the planner enters the *BestFirst* state. In this state, a vertex and edge are produced using *gradient descent*, or in other words, minimizing the potential function, g. The planner will stay in this state as long as g continues to decrease with every iteration. During this time, the counter i is kept at $i = 1$.

If the planner is unable to reduce g, it enters the $RandomWalk$ state. In this state, a random walk is executed and the algorithm returns to $BestFirst$ and the counter i is incremented by one. If g is still unable to be minimized, the algorithm returns to $RandomWalk$. This process is repeated, incriminating i each time $BestFirst$ is unsuccessful. If $i = k$, where k is

a pre-determined parameter that is the maximum number of iterations of $RandomWalk$, then the algorithm switches to $Backtrack$. In the $Backtrack$ state, the planner randomly selects one of the vertices produced during the previous sets of random walks. The counter i is reset to 1 and the state is changed back to $BestFirst$.

The introduction of random walk into the algorithm is useful for escaping local minima in the potential fields. However, in order to do so a large amount of parameters must be hand selected and tuned in increasing numbers as the number of dimensions increased. While RPP, has been shown to solve problems with 25 or more dimensions, the task of finding the correct parameters led to the developments of different sampling based algorithms.

Probabilistic Roadmaps (PRM)

One of the major sampling algorithm families are ones that deal with *roadmaps* called the Probabilistic Roadmap Method (PRM) [48]. In essence, a roadmap planner samples the search space randomly, connecting nodes with edges in order to find a solution between the start and the goal. In the original algorithm, the process is broken down into two parts: *learning phase* and *query phase*. The learning phase is then broken down into steps: *construction* and *expansion*.

In the construction phase, nodes are placed on the graph and checked if it belongs to the free space of the graph, or in other words if the node was placed on an obstacle or not. If it is in free space, it is then connected to neighbors. There are many methods to choosing which neighboring nodes to connect to including *k*-nearest neighbors or connecting all nodes within a given radius. Each of the selected neighbors is checked that the path between them is completely in the free space, or that by traversing the path there is no collision with an object. This process is continued until a pre-determined density of nodes is met.

The expansion phase starts right after the construction phase ends. In this phase, areas of the graph that are deemed more difficult are added to increase connectivity. For example, if there is a narrow passage between two objects, there is more likely to be disconnected parts of the graph. Additional nodes and edges are added to regions around nodes that are deemed to exist in “difficult” regions.

The second part of the PRM algorithm is the query phase. In this phase, the start and goal locations, V_s and V_g, respectively, are connected to the graph. This phase assumes that the graph as a whole is connected, and so by adding the start and goal locations, the algorithm assumes a path between the two can be found. A search algorithm is then run to find such path. If no path is found, then the query phase is said to have failed. In this case it can not actually be said if a path does or does not exist, just that this particular density and configuration can not find said path. However, if a path is found then, because of previous checks, it can be assumed that it is collision free. A path smoothing technique might need to be employed because of the disjointed nature of the results from this algorithm.

An example of how PRM works can be seen in Figure 2.30. Figure 2.30a represent the search space, with the large orange shapes being various objects. In Figures 2.30b-2.30d, nodes are added to the graph at random with edges being added between nodes in close enough proximty. The final graph is shown in Figure 2.30e, ending the learning phase. Figure 2.30f represents the addition of V_s and V_g. A search algorithm is employed to find a path connecting them, with the final result shown in Figure 2.31. The final path, highlighted in orange, would most definitely need to be smoothed out for an autonomous vehicle because of the sharp corners.

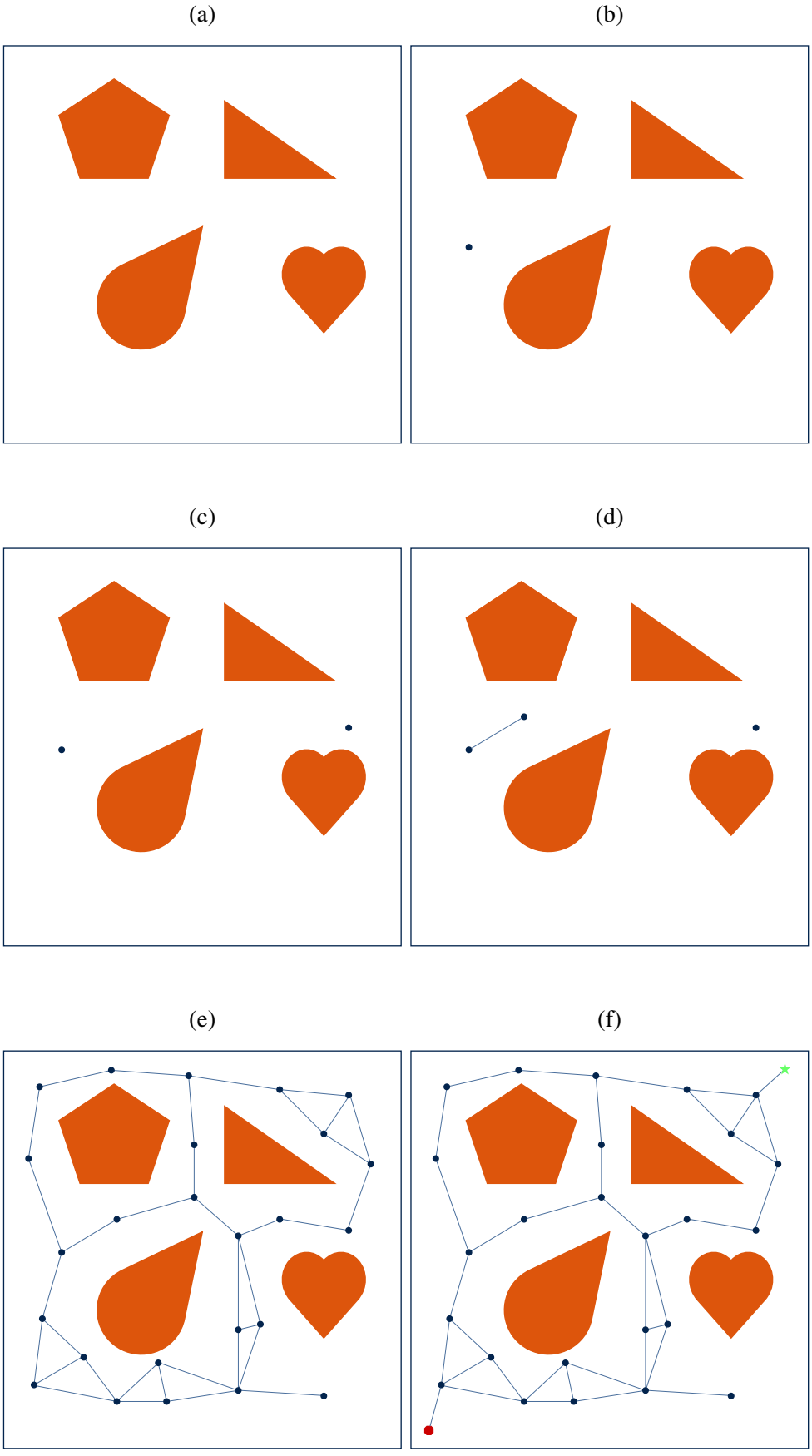


Figure 2.30: PRM Example

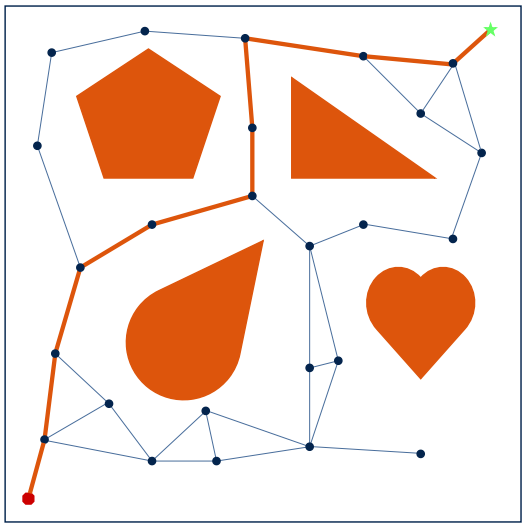


Figure 2.31: PRM Example Results

Various derivatives of PRM were developed to address different aspects of the PRM [49]. Others include:

- **Collision Checking** - In early methods, collision checking was done incrementally along the path. Later a *binary* method was developed to first check for a collision in the middle of the path. If there is no collision there, the halfway points on either side are then checked. This is done recursively until either the points checked are close enough or a collision is detected, as presented [50].

- **Sampling** - Originally, sampling was done randomly within the possible configurations, but other methods of sampling were proposed such as quasi-random methods using Halton points that appear to be randomly placed, but generate a more evenly spread of points in a given area [51]. Another method, Obstacle Based PRM (OBPRM) adds nodes that are in the free space, but when a node is not then a random direction is picked and the sample is moved and re-checked [52]. This is done until the sample chosen becomes free and is then added. This will lead to more nodes closer to obstacles. In problems where the solution requires travel through narrow or complicated configurations, this method may help find a path.

- **Neighbors** - Tuning this element of the PRM is important because of how computationally expensive collision checking along a path is. Choosing neighbors with a relatively

high separation distance increases the time needed for checking while also significantly increasing the change of a collision occurring, essentially wasting that effort. The previously mentioned method of *k*-nearest neighbors for determining neighbors generally has good results [49]. Other methods include *component*, which connects a new node to other nearest nodes that are already in a connected component. *Visibility* is another common alternative method [53]. In this method, nodes are kept only if they are deemed useful. To be useful the node has to either be able to connect to at least two other nodes or to none (waiting to eventually be connected). This reduces the number of nodes on a graph in an attempt to keep the number of kept nodes low so that a higher number of edge combinations can be attempted.

Rapidly-exploring Random Trees (RRT)

Rapidly-exploring Random Trees (RRT) was developed and published around the same time as PRM [54]. This algorithm serves as the basis of a broader family of algorithms by utilizing "space-filling trees". This family of algorithms randomly samples the free space of a problem, building a tree linked to the starting point. One of the problems that RRT addresses was the perceived bias towards exploring areas that have already been explored during random-walk algorithms. Instead, RRT has a tendency towards previously unexplored areas. Also, in the original paper, it was claimed that RRT is probabilistically correct, but no comment could be made on the convergence. However, in general RRT could outperform the PRM algorithm.

An overview of a basic RRT implementation can be seen in Algorithm 5. The algorithm needs the starting location, V_{start} and a maximum number of iterations, K. Upon initialization, V_{start} is added to the graph, G. In this implementation, a for-loop is set up to for K-iterations. For each loop, a node, or vertex, is generated at random within the free space and is called V_{rand}. Then the nearest neighbor is found in the point that already belongs in G. This nearest neighbor does not have to be a vertex but could also be the shortest distance to a point on an edge that also belongs in G. An edge is created between the two points and it is checked that there are no collisions. If that is true, then V_{rand} and $edge(V_{rand}, V_{near})$ are added to G.

Algorithm 5 RRT Algorithm

Require: V_{start}, K

$G.init(V_{start})$

for $i = 1 : K$ **do**

  $V_{rand} = GenerateRandomV()$

  $V_{near} = NearestNeighbor(V_{rand}, G)$

  if $edge(V_{rand}, V_{near})$ is in G_{free} **then**

    $G.addVertex(V_{rand})$

    $G.addEdge(edge(V_{rand}, V_{near}))$

return G

The algorithm can be adjusted for a path planner in a couple of different ways. The main loop can terminate if a random vertex is in the same spot or a certain radius of the goal node and is able to be connected to the graph. Or the main loop can continue on a K number of iterations and the cheapest path is selected. Or the main loop can terminate after a certain amount of time has passed.

A major sub-class of RRT algorithms is the [55] based on the RRT* algorithm. RRT* was developed to address the optimality issues of most sampling-based approaches. While it can not guarantee the most optimal path, it does generally begin to converge on the optimal path and outperforms RRT by the same metric. RRT* does this in two ways. The first is by recording the cost to travel to each vertex and checking the costs of all the neighbors within a given radius. If the new cost is cheaper than one of the neighboring node, then the new node replaces it. This addition to the algorithm gives RRT* a fan like shape as opposed to RRT's disjointed shape. The second addition to RRT* is the "rewiring" of nodes when a cheaper vertex is added to the neighborhood. In other words, if connecting to the new node causes the overall cost to decrease, the edges are rewired to go through that new vertex which creates smoother paths.

However, RRT* tends to trade efficiency for the gains in optimality. The two additions can be computationally heavy and an increase in required computation time is likely. In certain experiments, the run-time difference between RRT and RRT* shows RRT* taking over four times as long as RRT for the same amount of iterations, but was able to to output a more optimal path [56]. This trade off might be worth it for situations where time is not as much of an issue or where smoother and more optimal paths are needed.

RRT*-Smart, seen as an extension of RRT*, adds two more elements to RRT* in an attempt to address some of its predecessor's issues [57]. This algorithm begins similarly to RRT*, searching for an initial path between the start and the end. Once RRT* is terminated, the path is optimized by checking along the path for visibility of successive nodes. For example, in the chain $V_A \rightarrow V_B \rightarrow V_C \rightarrow V_D$, the path between V_A and V_C are checked for collision (the edge between V_A and V_B having already been checked during the initial part of the algorithm). If there is no collision, V_B is eliminated and an edge between V_A and V_C are then connected with an edge. This guarantees a shorter path because of the concept of Triangular Inequality, shown in Figure 2.32. This elimination of node along an edge continues while it is still collision free. At the end of this phase, the nodes that are left are called *Beacons*, and are used for the second addition, *Intelligent Sampling*.

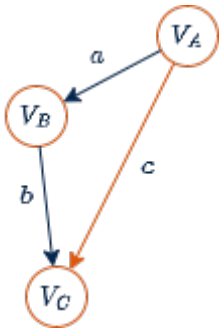


Figure 2.32: Triangular Inequality

In Intelligent Sampling, the graph is resampled, with samples biased towards a radius around each of the beacons. The idea is that the location of the beacons gives an indication as to location of the objects. Sampling near the beacons tend to increase the optimization of the final path at the turns about an object. In the end, RRT*-Smart improves upon the final path cost when compared to both RRT and RRT* in the same experiments mentioned earlier. However, for run time, RRT*-Smart performed slightly slower than RRT, but was able to finish three times faster than RRT*. While RRT*-Smart was able to improve upon RRT in terms of path costs, the computational tradeoffs could still be problematic in higher dimensional spaces or in much more complex problems.

Still, other variations of RRT* add elements similar to ones already discussed in the previous sections. For example, *Informed RRT** adds an *elliptical heuristic* in order to converge

on a solution quicker and more optimal than RRT* [58]. The addition is similar to that of the heuristic addition of A* algorithm. In Informed RRT*, once a solution is found, a theoretical ellipse is drawn around the start node, end node, and path. The algorithm then only searches within these ellipses, making the assumption that any improvement to the algorithm must lie within this region. As better solutions are found the ellipse narrows, continuing to decrease the search space and increasing the likeliness of converging on an optimal solution in a timely manner.

2.2 Path Smoothing

Most path planning algorithms will produce a path with sharp edges and turns as a result of the edges chosen for the final path. However, in practical applications this is not the desired output from a planner. These sharp turns are most likely not dynamically feasible for the vehicle to traverse or, at the very least, cause some reduction in speed. In problems where speed is essential, the time necessary to follow a jagged path might be detrimental. In other applications, such as a vehicle carrying delicate cargo or a human, the sharp turns could lead to accidents or discomfort to the passenger. Fortunately, path smoothing has been extensively researched in both the fields of mathematics and robotics.

One important characterization of a smooth path is the idea of *continuity*. There are two types: *geometric* continuity (G^i) and *parametric* continuity (C^i) [59]. Geometric continuity is when the end points of two splines meet and their respective tangent vectors are in the same direction. Parametric continuity is similar but with the additional characteristic of the tangent vectors also having equal magnitudes. In other words, parametric continuity assumes geometric continuity but the reverse can not be assumed.

A path is said to be C^i continuous between two curves if at point p, where they meet, their ith-derivatives are also equal. A path that is C^i continuous is also C^n continuous for all values of $n < i$. Figure 2.33a shows a discontinuous path, so for no values of i can the path be said to be continuous. A C^0 continuous path is shown in Figure 2.33b where the two halves of the path meet and the left path's end point is equal to the right path's start point. However, the slope of the paths are not equal, and so there is no higher order of continuity. Figure 2.33c shows a

C^1 continuous path, where the slopes of the two paths are equal at the point where they meet. This path is said to have curvature discontinuity though, because the curvatures at the point are not equal. The first half of the graph is said to have no curvature since it is a straight line, but immediately transitions to a graph with high curvature. Finally, Figure 2.33d shows a C^2 continuous path, where the derivatives of curvature are equal no matter where the path is split. In fact, the two halves of the path are indistinguishable.

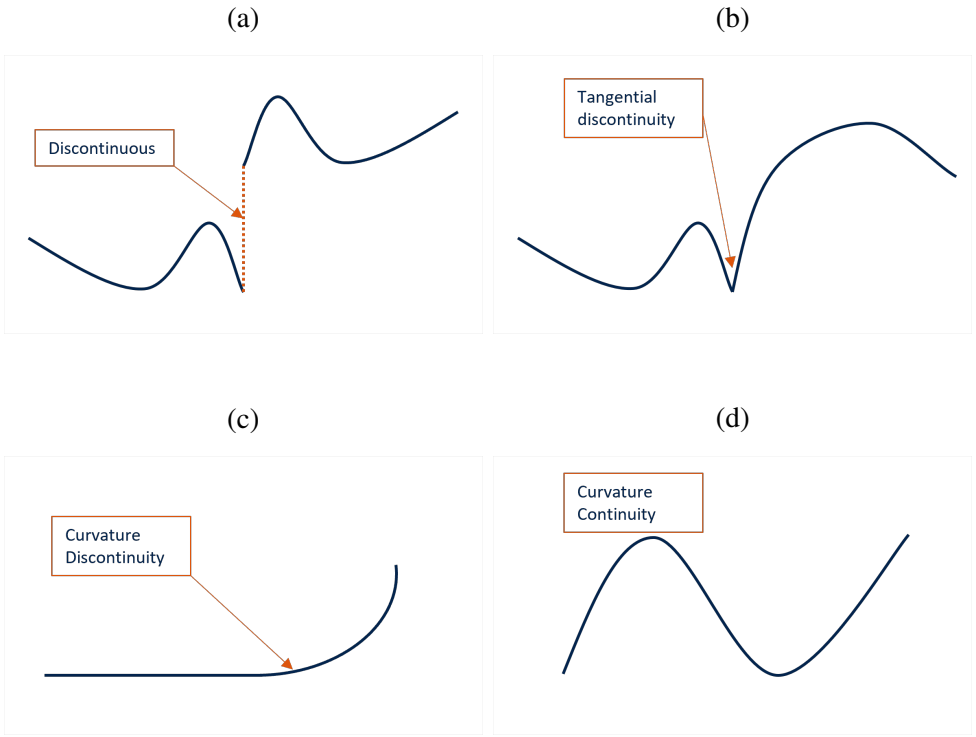


Figure 2.33: Examples of continuity

For the most part, C^1 and C^2 are sought in motion planning for robots. Higher order continuity begins to deal with properties not as relevant to the robots motion. Some common path smoothing techniques used in robotic motion planning will be discussed next.

Dubin's Curve

Given desired poses (position and orientation) at two points, Dubin's curve uses arcs and straight line segments to create a path between the points [60]. Figure 2.34 shows an example of how a path would be pieced together using Dubin's curve. Starting from the left, the path

to the first point (blue dot) is straight, but transitions to follow around a circular path until the second point is reached. Then the edge between the second and third is straight before also transitioning to follow a circular path until the fourth point is reached.

Dubin's curve has been shown to successful in implementation in a 3D UAV problem [61]. In that application, a 2D Dubin's curve was created and then extended into the third dimension. The UAV's kinematic model is propagated along the 3D curve to check for any collisions, throwing out any edge that does have a collision and keeping the edge that makes it collision-free. The author's of the paper gave two reasons for choosing Dubin's curve: 1) the curve takes into account the initial and final headings and 2) it produces the shortest path given a curvature constraint, which in this case would be the UAV's turning radius. The method for applying Dubin's curve for an autonomous vehicle is used in a similar fashion in the water domain as well [62].

Due to it's relatively low computational expense, Dubin's curves make an excellent path smoothing technique in environments where computing space is limited. They also can guarantee a shortest path around an object. However, these curves are traditionally used in 2D spaces, so for applications in slightly higher dimensions, interpolation might be required. The paths also do not guarantee C^2 continuity, which can cause high jerk in the robot's movement.

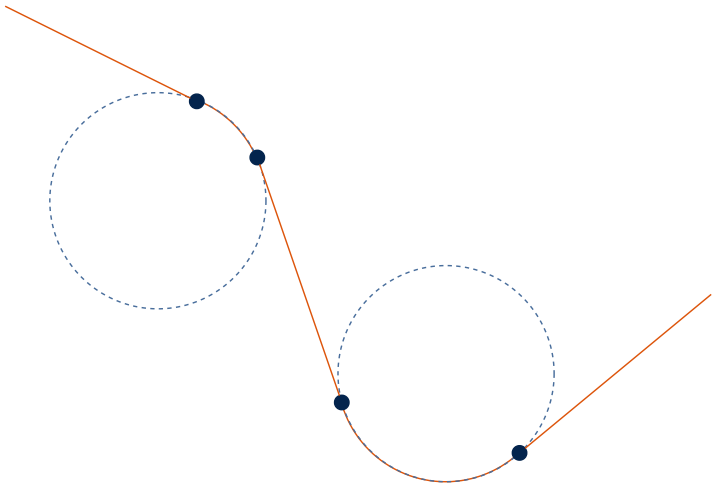


Figure 2.34: Dubins Curve

Clothoids

Clothoids, also known as Euler Spirals, are a type of curve with a special property that their curvature changes as a function of its length. Therefore, as the the curve traverses, the radius of curvature changes linearly, which also means that a vehicle traversing the curve will have a constant rate of angular acceleration. This property tends to keep the overall path to be smoother than other options, which makes clothoids a good choice for autonomous vehicle path planning.

In [63], clothoids are combined with arc and straight line segments for path generations. The authors chose clothoids specifically because they believe that they require minimal steering, emulating the way a human driver would drive. Others have created a piece-wise path using clothoids for autonomous navigation in urban environments [64]. Their experiments included roundabouts, lane changes, straight and curvy road segments and were interested in producing paths that would be most comfortable for their passengers. Clothoids have also been implemented using a look-up table to approximate the desired shape to increase the overall efficiency of their planner and to run in real-time [65].

Bézier Curve/B-Splines

Bézier Curves were were invented in 1962 as method for designing automobile bodies and a curve with degree *n* is represented by,

$$P_{[t_0,t_1]}(t) = \sum_{i=0}^{n} B_i^n(t) P_i \tag{2.6}$$

where P_i are control points such that $P(t_0) = P_0$ and $P(t_1) = P_n$ [66]. $B_i^n(t)$ is called the Bernstein polynomial and is represented by

$$B_i^n(t) = \binom{n}{i}) \left(\frac{t_1 - t}{t_1 - t_0}\right)^{n-1} \left(\frac{t - t_0}{t_1 - t_0}\right)^{1}, i \in \{0, 1, ..., n\} \tag{2.7}$$

Bézier curves guarantee that the spline produced will pass through P_0 and P_n and the path will always lie within the convex shape formed by the control points, which can be seen in Figure 2.35.

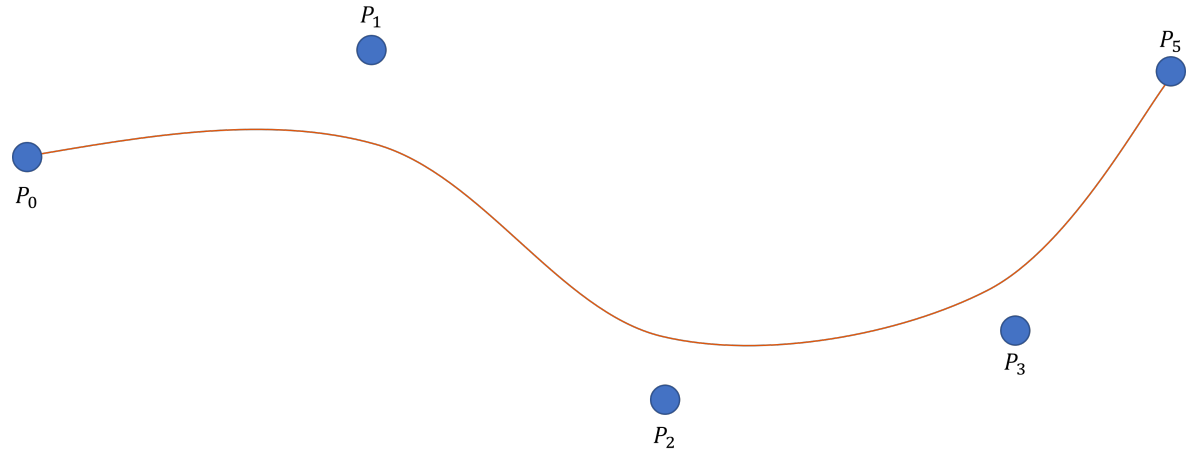


Figure 2.35: Bézier Curve

A more general form of Bézier Curves is known as B-Splines and is a linear combination of Bézier Curves. B-splines use *knots* x_i when formulating the function where $x_i, x_0 <= x_1 <= ... <= x_m - 1$ for m-number of points [67]. The piece-wise function is represented by

$$S(x) = \sum_{i=0}^{m-n-2} B_i^n(x) P_i, x \in [x_n, x_{m-n-1}] \tag{2.8}$$

where P_i are the control points where the number of control points is equal to $m - n - 1$. The B-splines are defined below in Equation (2.9).

$$B_{j,n} = \frac{x - x_j}{x_{j+n} - x_i} B_{j,n-1}(x) + \frac{x_{j+n+1} - x}{x_{j+n+1} - x_{j+1}} B_{j-1,n-1}(x), j = \{0, ...m - 1\} \tag{2.9}$$

Note the recursive nature of Equation (2.9). In order to calculate $B_{j,n}$, both $B_{j,n-1}$ and $B_{j-1,n-1}$ must be calculated. This in turn means that for $B_{j,n-1}$ to be calculated, $B_{j,n-2}$ and $B_{j-1,n-2}$ also must be calculated.

Bézier Curves and B-Splines will often produce similar results in paths. While Bézier Curves are simpler to implement in practice, B-Splines allow for more flexibility when it comes to control points. A B-Spline formulation requires control points, the degree, and the knot vector. Changing the latter two of those can drastically change the characteristics of the path,

causing tuning to be necessary. However, adding control points to the path does not effect the path as a whole, just the local area of the control point since B-Splines are a combination of Bézier Curves, blended together using the knot vector. This is unlike Bézier Curves where an additional control point affects the path entirely.

Both techniques have been used in path planning. In [68], the authors present a real-time implementation of a path planner based off Bézier curves. The simplicity of the curve allows them to create a safe and and comfortable path at quick enough speeds to effectively avoid collisions with objects encountered on the road. A B-Spline implementation of a swarm of UAVs has also been [69]. They were chosen because the splines only require as few as three control points rather than a segment with thousands of points. They also selected to implement a third-order version of B-Splines to ensure the path generated stays near the control points, reducing the risk of a collision. B-Splines were also successfully used in cooperative driving of multiple autonomous vehicles [70]. This was due to the parameterized nature of the curve, allowing for a much lower communication bandwidth between vehicles when passing information back and forth about their respective trajectories. Finally, because of the piece-wise nature of B-Splines, a path can quickly be updated upon encountering a previously indicted object by only updating the B-Splines around the said object [71].

Other Smoothing Techniques

Another common path smoothing technique include non-uniform rational B-Spline curve, or *NURBS*. A NURBS curve is defined by Equation (2.10) below.

$$C(t) = \frac{\sum_{i=0}^{n} N_{i,p}(t) w_i P_i}{\sum_{i=0}^{n} N_{i,p}(t) w_i} \tag{2.10}$$

for a p-order curve where $N_{i,p}$ is a B-spline basis function with control points P_i, and w_i is the weight [72]. Having both control points and weights allow for NURBS to be flexible when generating the shapes of desired trajectories. Conversely, poorly chosen weights can cause for poor parametrization of the curve. NURBS Curves are traditionally used in Graph Design tools but have been utilized in various path planning tools [73, 74, 75].

Cubic Splines are also often used in path smoothing because it is the minimal degree needed to have C^2 approximations and are usually smooth enough in the presence of small curves [67]. A cubic spline, like B-Splines, are a piece-wise polynomial that is shaped to pass through a set of global way-points, or control points. The general form of a one-dimensional cubic spline in a set of $n+1$ piece-wise set $[y_0, y_1, ..., y_n]$ is defined by Equation (2.11).

$$Y_i(t) = a_i + b_i t + c_i t^2 + d_i t^3 \tag{2.11}$$

where i represents the *i*-th piece of the spline and $t \in [0, 1]$ [76]. Examples of cubic spline applications in path planning can be found in [77, 78, 79].

2.3 Path Planning in Autonomous Racing

In previous sections, various aspects of path planning and their implementations in the context of autonomous vehicles were presented. However, autonomous robotic, or vehicular, navigation has a wide range of applications, from subterranean robots to UAVS to land vehicles. Each type of robot has different requirements for its planning algorithm due to its physical limitations, domain, and specific use-case. These requirements might mean sacrificing efficiency for optimality or complexity for fidelity and vice versa.

For this particular thesis, the vehicle domain is a race car, and the specific use-case is high-speed autonomous racing. This particular set of factors is rather unique and has been a more recent development in the field of autonomous vehicles, thanks in part to the IAC and RoboRacing. The needs for this problem are reliability, predictability, and speed. The planner must be reliable in that it will always output a path, whether that is for object avoidance or for last-second braking. It must be predictable in that it will always produce a similar path for relatively similar scenarios so that other aspects of the software stack can always depend on it. Lastly, it must computationally fast in order to give the car the most up-to-date safe path and not take up more computational power than it needs to allow for other systems to be able to run with enough frequency to operate safely.

There are also other sacrifices that can be made such as less robustness to various scenarios and less graph complexity. The vehicle will, in theory, never leave the race track and will hopefully never encounter unknown objects. This means the algorithm does not need to account for various road variables that a street car might encounter. Also, the algorithm should only encounter other vehicles that are similar to itself and also going in the same direction, albeit at different velocities. It will not have to encounter objects of varying size or with trajectories not unlike its own. With the vehicle only traveling on the race course, and only in the ”forward” direction, the graph traverses can be simplified to a directionally cyclic graph which greatly simplifies the needs of the graph.

Other authors attempt to solve the path planning problem for an autonomous race car [80]. They consider using an model predictive control, or MPC, type solution but found that it requires too much computational power to be practical in a race scenario. Instead, they settle on representing the track as a graph. Offline they determine the sequence of maneuvers that are associated with the optimal race line and associate them with the nodes on the constructed graph. When encountering another object, the algorithm assumes that it is following the same minimal-time trajectory and propagates the objects position forward in order to optimally determine the path around the obstacle to avoid a collision.

The approach taken in [81] was not for a race scenario, but rather an of-road driving course with potential for encounters with unknown variables. Like the previous paper, these authors assume some knowledge of the track beforehand, in their case using global way-points. Using the way-points, a parametric curve connecting the points is calculated with a C^2 continuity. This curve is named the *base frame*. Using information from the base frame, *path candidates* are generated in real time based on current positioning. These paths are designed so that at the end, the orientation of the car matches the curvature of the candidate frame at that point. An example from this approach can be seen in Figure 2.36. Using various characteristics of each candidate path and checking for collisions, the algorithm determines the best path for the vehicle.

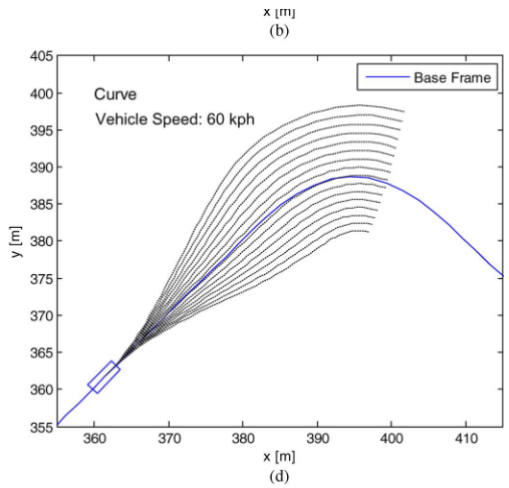


Figure 2.36: Candidate Paths from an Algorithm

This algorithm also determines target speed for the vehicle along the selected path. The speed limit of the path due to the curvature is calculated in Equation (2.12).

$$v_\kappa = \sqrt{\frac{|a_y|_{max}}{max(\kappa_{path})}} \tag{2.12}$$

Also, factored in to the speed limit is the proximity to other objects and the width of drivable portion of the road. This is to reduce the risk of an accident or collision. The author's point to the algorithms ability to perform in simulation, as well as competition, outputting smooth, consistent, and safe paths.

Researchers have also approached the problem by using an MPC in their path planner while simulating the Suzuka race track [82]. Their experiments consisted picking points on the track and calculating the ideal trajectory while measuring the time for that iteration. With their setup, the MPC was able to produce a trajectory in about 0.02 seconds. However, their planner was not implemented on any vehicle or vehicle simulator. This, combined with prior information regarding the computational time needed for an MPC controller led to other path planning methods with simulation or real-world results to be explored.

Other research have also utilized an architectures into their planner that went on to be implemented in an autonomous Dallara AV-21 and Formula Student race cars, respectively utilized [83] and [84]. Both planners have an offline elements that include a global trajectory optimization, which is common in planners where knowledge of the track is known *a priori*.

The authors in [83] simplify the problem by moving the calculations into Frénet space. Or in other words, they represent their race lines and trajectories in terms of (s, d) where s is the length along a given path and d is the latter offset. Multiple trajectories are produced with varying degrees of convergence rates to the predefined line. Using potential collision checking and various safety checks, a trajectory is ultimately chosen and used in the controller.

Another high-speed planner was presented in [85]. This implementation wholly defines the graph offline, also in Frénet space, with each discrete point represented by $[x(s), y(s), \theta(s), \kappa(s)]$. Edge selection is based off a function of three weights, κ_κ, κ_{trans}, and κ_{route} where each is a function of the distance to an object cluster , the road curvature, and the distance to the racing line, respectively.

Similarly, [86] discretizes a track completely with nodes connected by cubic splines that can guarantee C^1 continuity. Some of their edge costs can be calculated offline as well, including the costs due to the lateral displacement from the race line and the maximum curvature of the line. Moving calculations like this to the offline phase helps reduce the computational load at run time. During each planner iteration, the portion of the graph between the ego vehicle and the goal horizon is extracted from the whole graph. Nodes that have the potential to plan a collision into another vehicle are removed, along with the nodes in the vicinity of the other vehicle. A shortest path is calculated and re-processed to give it C^2 continuous splines.

Chapter 3

Graph Based Planner for Oval Tracks

3.1 Background and Motivation

In May of 2020, teams participating in the inaugural Indy Autonomous Challenge (IAC) in Indianapolis, had access to a fully autonomous drive-by-wire retrofitted Indy Lights race car. Over the course of the next five months, teams wrote and tested their software stacks in hopes of competing in the solo time trial in October of 2020 that consisted of two fast laps and an object avoidance lap. While the solo competition was not what the organizers originally envisioned for the competition, it was the first step towards a true head-to-head multi-car racing competition.

The next competition came quickly on the heels of the first, being scheduled for January of 2021 at the Las Vegas Motor Speedway in conjunction with the Consumer Electronics Show (CES). This competition consisted of a tournament bracket style setup with the two-team rounds of a *passing* competition. The format of each round included two cars alternating overtaking maneuvers at increasing speeds until one of them failed to complete the maneuver. The *defending* vehicle had to maintain a specified speed and hold an inside line while the *attacking* vehicle attempted to safely pass and merge back onto the inside line. Each time both vehicles successfully completed the overtake at a given speed, the defender speed was increased. The defender speed started at 130 kph (36 m/s) and gradually increased each round to speeds over 270 kph (75 m/s).

For a motion planner to be effective in this competition, it needed to be safe, efficient, stable, and predictable. This planner would also have to be written from scratch and successfully integrated with the existing software stack in less than three months. After a survey of

potential planner algorithms, the A* algorithm was chosen as the basis for this planner. The A* algorithm's strength comes from combining elements of Dijkstra's Algorithm and principles of greedy best-first-search algorithms. Dijkstra's algorithm guarantees optimality, and the adding of the heuristic allows A* to find the solution in a much more efficient manner by searching the most promising vertices first [87].

3.2 Planner Architecture

Upon startup, the planner would load an existing graph or generate a new graph with the desired parameters such as distance between lanes or layers. The planner continuously took in vehicle state estimates, object detections/predictions, race control flags, and the desired race line trajectory. It then performed the graph search using the A* algorithm in presented in section 2.1.1. Before outputting the desired path, the planner up-samples and smoothed the optimal path to provide an improved reference for the motion control modules. A more detailed description of the planner architecture can be found in Appendix A.1.

3.2.1 Graph Generation

The graph generation could be done offline because the racetrack bounds were known in advance. It was safe to make a couple assumptions: (1) the ego vehicle should remain inside the track bounds and (2) the ego vehicle will always be moving in the forward direction (track dependent). These assumptions allowed the generated graph to be simplified to a directionally cyclic graph which greatly reduced the complexity of the graph. This also allowed for the edges to be computed at run-time without greatly affecting the speed of the calculation. However, if computation time was an issue, the graph could also be pre-computed during initialization.

Edges were generated from a vertex to the dynamically feasible neighbors using a lateral acceleration limit. This lateral acceleration limit acts as a tuning parameter, and in practice it determines how many layers forward the car must travel to move over laterally one vertex. Figure 3.1 gives an example of a few vertices in the graph and their edges.

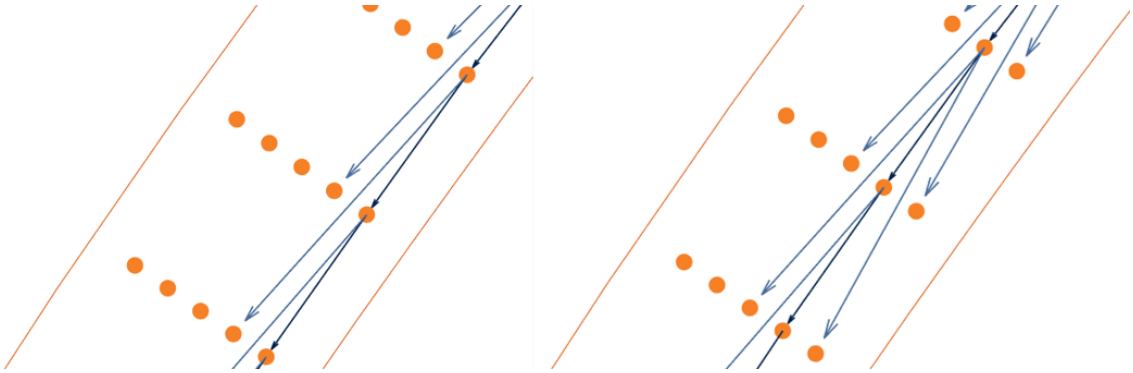


Figure 3.1: Vertices and edges.

For purposes of this thesis, the graph is described by two coordinates, *lanes* and a *layers*. A layer spans the width of the track and are set equal distances apart from each other around the track. An equal number of lanes are placed in each layer. This is can be seen in Figure 3.2 where there are 5 lanes in the graph.

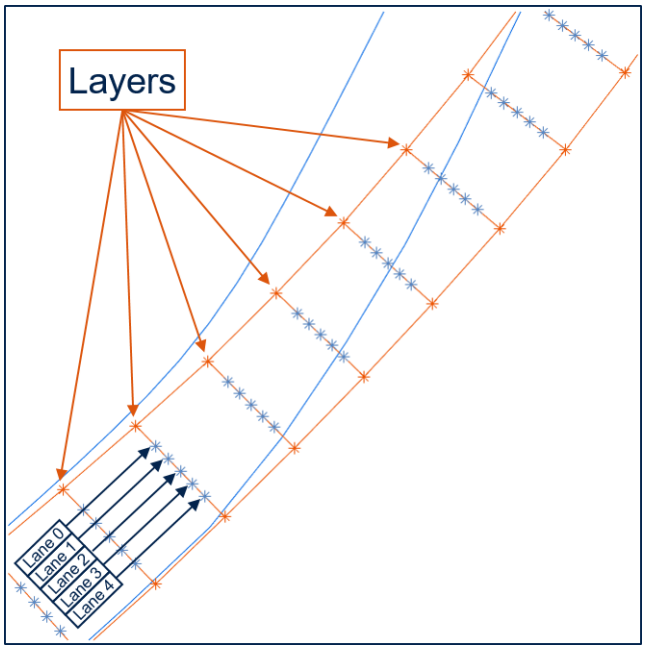


Figure 3.2: Lanes and layers.

Upon the graph generation, several parameters could be changed and tuned to better fit a track or a scenario. The distance between layers could be adjusted, with smaller distances creating a denser graph but requiring more computation time. For this testing, the layer distance was set to be 10 meters. The number of lanes could also be increased or decreased depending on how finely the lateral movement needed to be to achieve a successful pass. Five lanes was

shown to be a sufficient number for the purposes of this competition. In Figure 3.2, the blue stars represent the graph points while the orange stars represent the corresponding perpendicular points on the inner and outer bounds. The adjustment of where the lanes start and end relative to the inner and outer bounds allowed for a margin of safety within the paths outputted by the planner.

3.2.2 Graph Search

During the graph search portion of the planner, the planner calculates the cost of each vertex. Four different factors make up the total cost of each vertex: object cost, dynamic cost, distance cost,and heuristic cost.

Object Cost

The object cost is a penalty associated with a given vertex based on its proximity to a detected object's predicted path. The cost of the vertices decrease as they move further from an object's predicted trajectory, causing the planner to avoid the object. The implemented object cost configuration is shown in Figure 3.3. The light, medium, and dark blue areas correspond to *soft*, *medium*, and *hard* weights, respectively, that are applied to the cost of the vertex. The width of the bands represent how far those weights carry laterally from the object. The weights do not necessarily have to correspond with certain lane(s) or vertex, rather they correspond with an actual distance from the object. The weights are additive in the event there are multiple objects in close proximity.

There are several parameters around the object cost that can be adjusted based on the given scenario. Most important of which are the soft, medium, and hard weights and their respective distances from the detected object. Too similar of weights or too wide of a distances from the object could potentially cause the algorithm to plan a path straight through the object. For the best case, a notable gradient at a reasonable distance from the object is required for a successful object avoidance.

Another important parameter is the distances longitudinally the weights carry for each object. In other words, how far forward and how far back from an object do the weights still

hold. For this testing, the distance was set at initialization, but it could have been rewritten to scale the distances based on the speed of the defending vehicle.

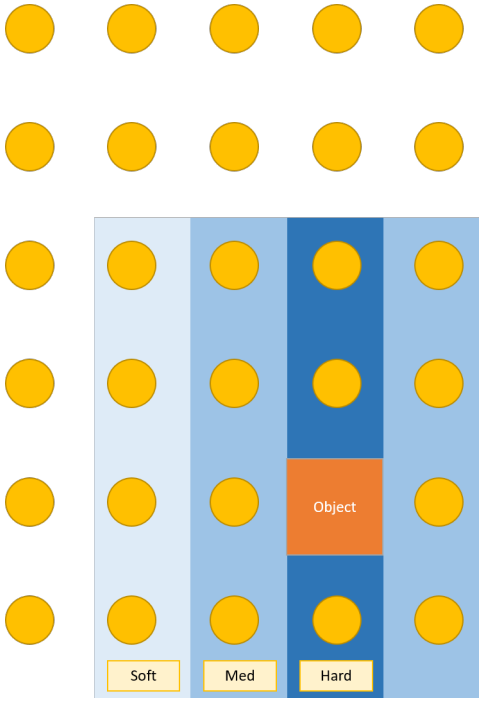


Figure 3.3: Object cost assignment.

Dynamic Cost

A dynamic cost was included to prevent the planner from outputting a path that is not dynamically feasible for the ego vehicle, and to limit any unnecessary lateral movement. Several methods, such as the lateral velocity, the lateral acceleration, or a vehicle model, can be used to calculate the dynamic cost. Here, the lateral velocity is used as the basis of the cost calculation as shown in Equation (1) and in Figure 3.4.

$$J_{dyn} = \sqrt{LD^2 + SD^2} * \sqrt[3]{V_x} * DW \tag{1}$$

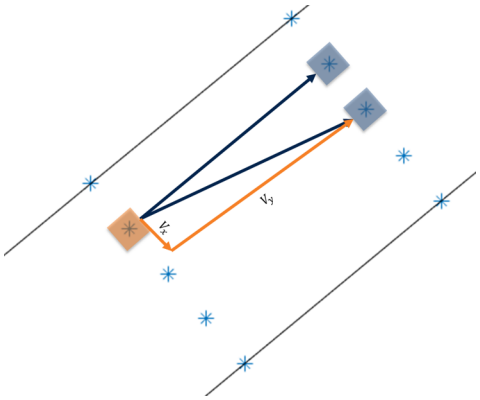


Figure 3.4: Dynamic cost.

where LD represents the distance between the lanes, SD is the distance between layers, and DW is the dynamic weight. With this formulation, the cost is scaled at higher speeds to stay within the range of dynamic feasibility. Figure 3.5 shows how the dynamic cost changes with lateral velocity with a selected dynamic weight of 0.1884.

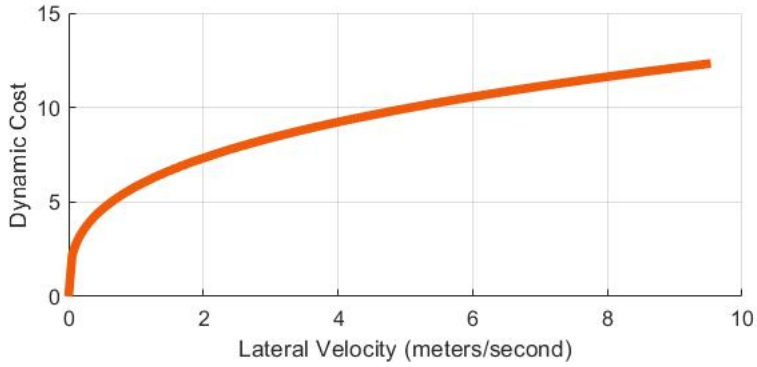


Figure 3.5: Dynamic cost assignment.

Distance Cost

The distance cost is related to a vertex's distance from the race line, seen in Figure 3.6. The cost is calculated as a linear function of the perpendicular distance from the desired racing line. The purpose of the distance cost is to keep the path on the optimal race line when obstacles are not present. For the purposes of the IAC competition, the desired race line is generally set to be the inner most lane of the graph. It could be changed at the planner initialization to be any other desired line.

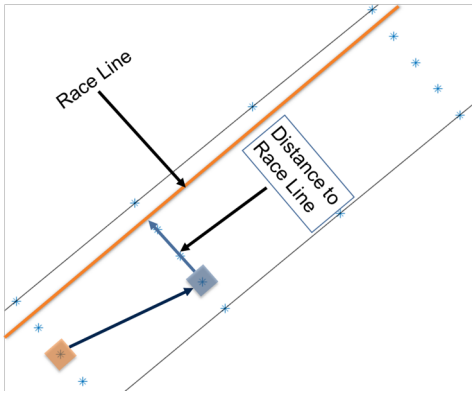


Figure 3.6: Distance cost.

Heuristic Cost

The heuristic cost is the same as most grid based A* planners, and is calculated as the Euclidean distance from the vertex to the goal, as shown in Figure 3.7. The purpose of this cost is to prioritize the search of the vertices that are nearer to the goal, decreasing the total computation time required to find the optimal path, when compared to Dijkstra's method. The heuristic cost was not weighted; instead, all of the other costs were tuned to the expected heuristic values.

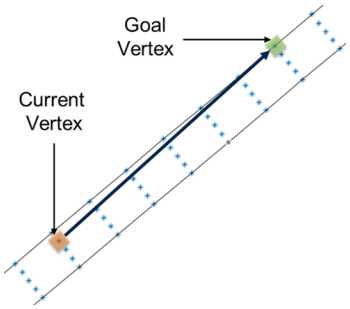


Figure 3.7: Heuristic cost.

3.2.3 Path Smoothing

The path generated from the A* planner results in a rough path with discontinuous curvature, represented by the orange line in Figure 3.8. The non-smoothed path was upsampled between points and a moving mean is applied for smoothness and allows the path to be tracked more accurately by the motion controller.

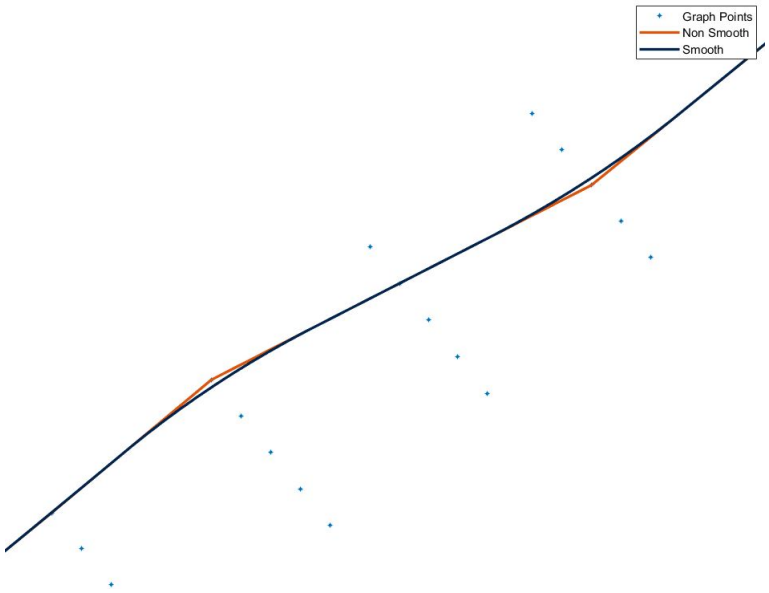


Figure 3.8: Path smoothing.

3.3 Experimental Setup

3.3.1 LGSVL Simulator

Testing was done on the LGSVL Simulator, seen in Figure 3.9, that had custom vehicle models and tracks provided to the IAC teams to allow for replication of the competition environment. The LGSVL Simulator was an open source, high-fidelity simulator designed for autonomous vehicle software testing [88]. The simulator provided integration with various *autonomous driving* (AD) open-source system platforms like Apollo, Autoware.AI and Autoware.Auto. The simulator also allows for plug-ins for any type of run-time framework. Like Autoware.AI and Autoware.Auto, this code stack utilized the open-source ROS2 bridge for communication.

The simulation utilizes Unity's game engine to simulate a highly realistic and customizable environments. The customization can be broken down into four different categories: environment simulation, sensor simulation, vehicle dynamics, and control simulation of the ego vehicle. The environmental simulations included control over aspects of the scenario such as time of day or weather conditions. This was not necessary for this experiment because the IAC competitions would not be held in any type of wet or cold conditions that would greatly impact the vehicle dynamics.

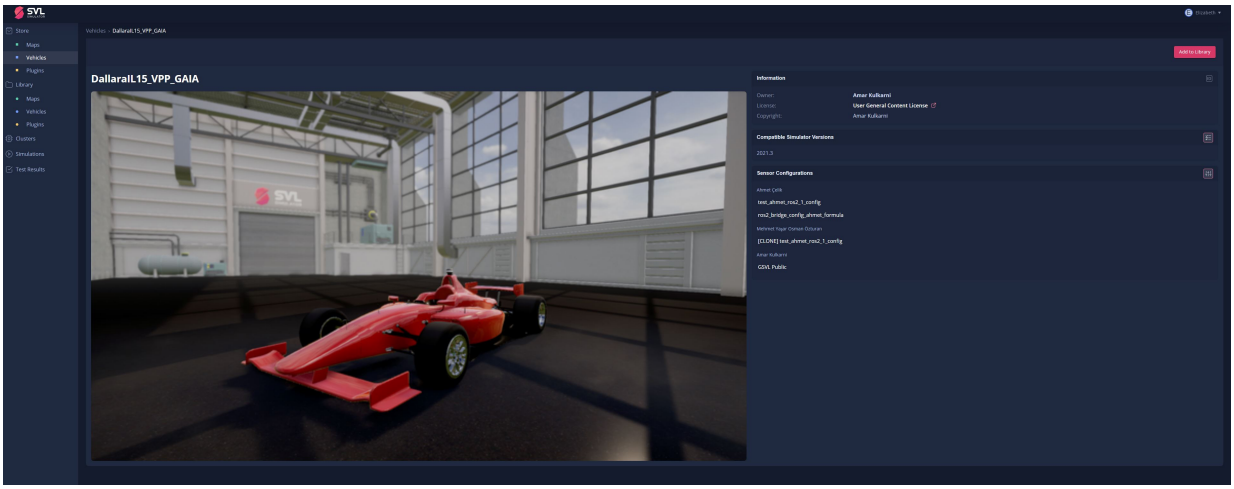


Figure 3.9: LGSVL Simulation.

The simulator came with a set of "out-of-the-box sensor" configurations for sensors such as GPS and IMU, but also allows for real-world sensor models to be used during testing. The sensors can be configured with placement on the vehicle, publishing rate, and topic name among other things. The testing simulation environment used the same sensor configuration as the ones of the AV-21. LGSVL provides a basic vehicle model, but as with most things regarding the simulation, external models can be imported into the simulation using the Functional Mockup Interface (FMI) [89]. However, the main reason this simulation environment was chosen was because of its real-time simulation capabilities and ease of use. The ROS2 integration allows for easy porting of code between the software-in-the-loop (SIL) setup and the actual car.

The planner assumes zero noise and bias on the sensor outputs for the sensor simulation since that was out of the scope of this testing. The built-in vehicle dynamics module was insufficient and was replaced with a custom-built vehicle and tire dynamics model that more accurately replicated the true performance of the AV-21. The control inputs were produced using Auburn's autonomous racing software stack including localization, perception, motion planning, controls, and safety monitoring. The full set of parameters used for testing can be found in Appendix B. A model of the Las Vegas Motor Speedway (LVMS) was used for testing for the 2020 IAC at CES competition. An example of the simulation environment used for testing can be seen in Figure 3.10. The SVL simulator has since been discontinued by LG and

an alternative simulator would be needed in order to continue to have up-to-date simulation [90]

Figure 3.10: SVL simulation for the LVMS.

3.3.2 Object Simulation

To effectively test the motion planning algorithm, a simple object simulator was developed. The density, locations, and trajectories of the objects can be set explicitly or randomized. An example of random object placements can be seen below in Figure 3.11. Object detections had no added noise, and a constant velocity was assumed for the predictions, since object detection and tracking are outside the scope of this planner. For this testing, the detection distance was set to 100 meters which is approximately the detection range with the sensors on the actual vehicle.

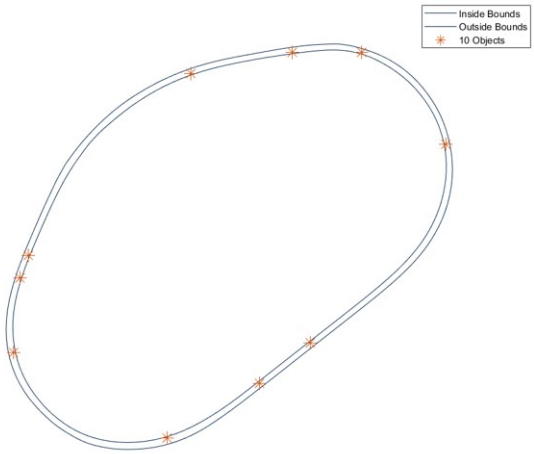


Figure 3.11: Example of randomly placed objects.

3.3.3 Tests

Tests of the planner were performed in both simulation and in the real world at LVMS. In simulation, two types of static scenarios were tested. For the first static test, static objects were placed on the race line randomly. For the second scenario object locations were selected in a way that intentionally forced complex avoidance maneuvers. For the simulated scenarios, the car was given an initial desired velocity of 35 m/s which was increased with every lap completed up to a top speed of 75 m/s. The dynamic testing involved placing another car on the inside lane (to mimic the IAC competition rules) that was set to go between 10 m/s and 45 m/s slower than the ego vehicle.

For the real-world experiments, Auburn's AV-21 was tested at LVMS against both dynamic "ghost" vehicles and real vehicles from other teams. These tests were performed with 5 m/s and 15 m/s differential speeds between the two vehicles. This range of differential speeds was much lower due to the risks involved in real-world testing.

3.4 Results

The planner was able to successfully avoid and pass the randomly dispersed static objects at the full range of velocities with no collisions. An example of a successful pass is shown in Figure 3.12. The blue dot indicates the location of ego vehicle, the green square indicates a

detected object prediction, the blue line represents the desired race line, the orange line represents the planned path, and the star indicates the current goal vertex. The average, minimum, and maximum planner calculation times for each tested speed during the static obstacle tests are shown in Table 3.1. At the average planner calculation time was 5.1 ms, the ego vehicle will travel less than 0.39 meters between iterations when driving 270 kph (75 m/s). These are much lower calculation times than many of the algorithms discussed in the prior chapter.

Another example of a static obstacle avoidance test can be seen in Figure 3.13. In Figure 3.13(a), the ego vehicle approaches four objects. Initially, only the first two objects are within detection range of the ego vehicle. Because of this, the planner outputs a path that would go through the undetected third object, which is indicated by the red box. Once the third object is within detection range, the planner re-plans a safe path that avoids the newly-detected object as shown in Figure 3.13(b)

The results from the simulated dynamic obstacle overtaking tests can be seen in Table 3.2. The average calculation time was 0.0033 seconds, which is likely slightly faster due to there being less objects to avoid. Figure 3.14 shows an example of the ego vehicle approaching and planning an overtake around a dynamic object.

Table 3.1: Static Object Calculation Times

Speeds	**Path Calculation Time (ms)**		
(m/s)	***Average***	***Min***	***Max***
35	65.	8900E-7	186.6
45	5.4	1.1	187.1
55	3.9	9.87E-7	297.6
60	5.3	1.0	173.3
65	3.2	1.2	126.6
70	5.6	8.84E-7	317.1
75	2.1	1.1	7.3
All	5.1	8.89E-7	317.1

Table 3.2: Dynamic Testing Calculation Times

Differential Speeds (m/s)	**Path Calculation Time (ms)**		
	Average	***Min***	***Max***
10	4.8	9.27E-7	17.2
15	3.5	8.20E-7	35.0
20	3.3	8.44E-7	21.1
25	2.4	7.76E-7	06.2
30	3.9	7.04E-7	32.7
35	2.8	7.60E-7	13.5
40	3.6	8.40E-7	16.2
45	3.3	8.78E-7	10.7
All	3.3	7.04E-7	35.

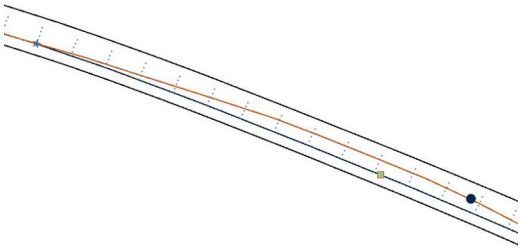


Figure 3.12: Path plan around an object.

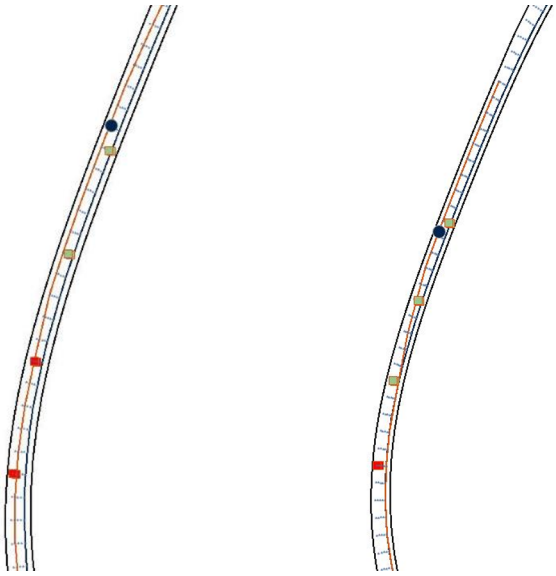


Figure 3.13: (a) Detection of two of objects. (b) Re-planned path to avoid the third and fourth object.

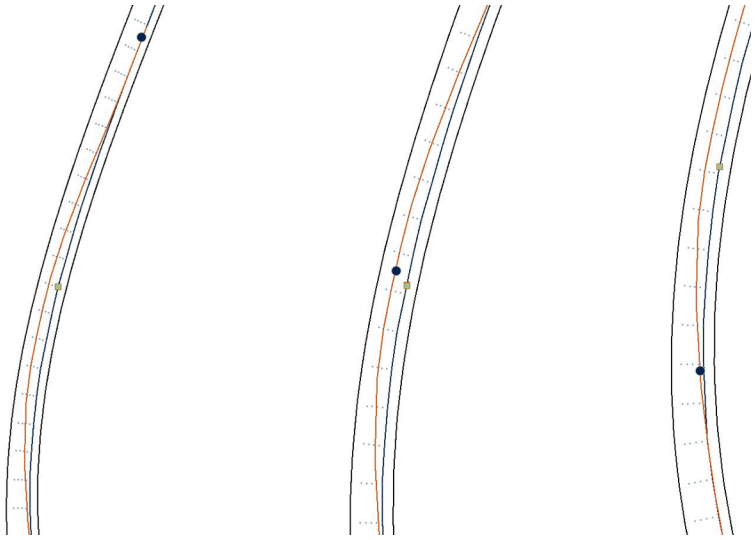


Figure 3.14: (a) Detected moving object and planning path around. (b) Mid-object pass. (c) Moving back to race line.

3.5 Discussion

When this planner was tested on Auburn University's AV-21, the average planning time was .32 ms, or ten times faster than the simulated results. The standard deviation of the calculation times was .13 ms and the minimum and maximum calculation times were .19 ms and 2.0 ms, respectively. The planner was successfully able to avoid and overtake real vehicles as well as simulated "ghost" opponents during testing. There was a couple of reasons the calculation times were so much lower on the actual car. First, the simulation tests were done on a computer that was running both the simulation environment as well as the entire software stack. Second, the computer in the car is far more powerful than the one used for the simulation results

This iteration of the planner works exactly how it needs for the narrow and specific use-case during the IAC. However, translating it to a broader application, or even a slightly more complex situation would not result in ideal path solutions. Types of potential additional complexity includes:

- defending car modulating speed
- defending car performing evasive maneuvers
- defending car racing off a pre-described race line

- multi-car racing with any/all of the above above possibilities
- road course

One of the limitations of this iteration of the planner is its limitations to ovals an tri-ovals More specially, tracks that have large and sweeping turns in the same direction with any range of banking. A road course would bring in the element of both left and right turns that are much tighter than at the IMS or LVMS. An example of a road course at Modena, Italy is shown in Figure 3.15.

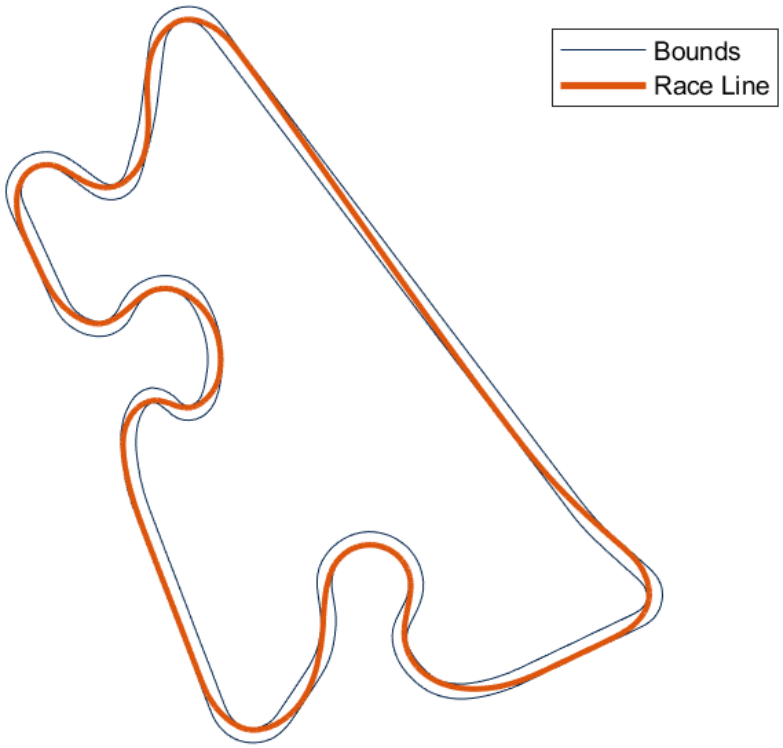


Figure 3.15: Modena Road Course

A closer view of a graph produced using the same planner for the road course is shown in Figure 3.16 The orange line represents a potential ideal racing line for this particular course Ideally, the ego vehicle would follow the line, hugging closer to the inside of this tight curve

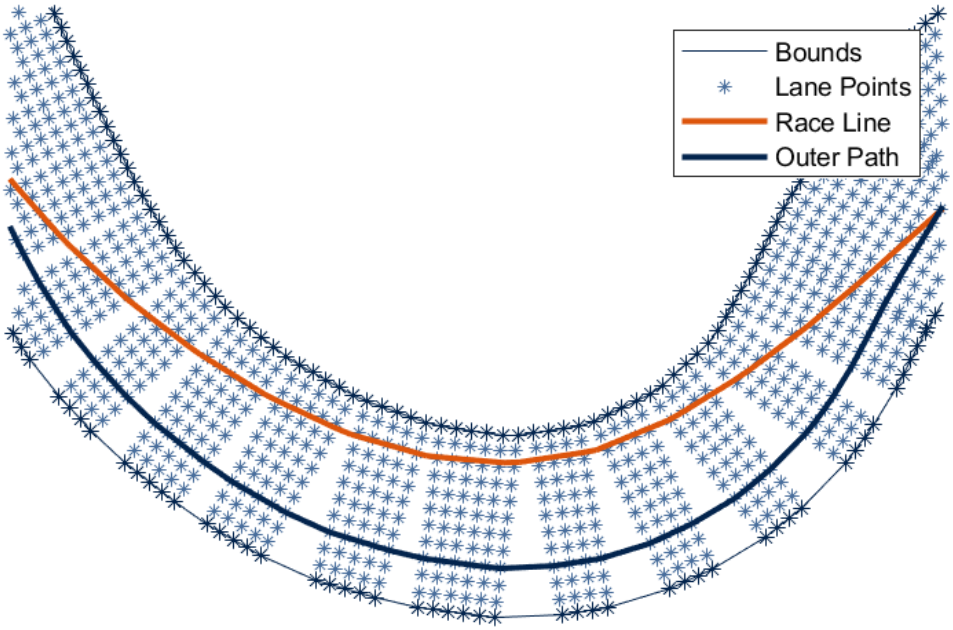


Figure 3.16: Modena Road Course Closeup

However, circumstances of the race might dictate that the ego vehicle travel along the outside of the curve. This would result in a graph search that includes areas with less dense nodes. An approximation of one such outer line is represented by the blue line. In this particular example the problem is not as evident because of the density of the nodes Changing that spacing to 5 meters, which is closer to what was run in this planner testing, results in a path similar to Figure 3.17.

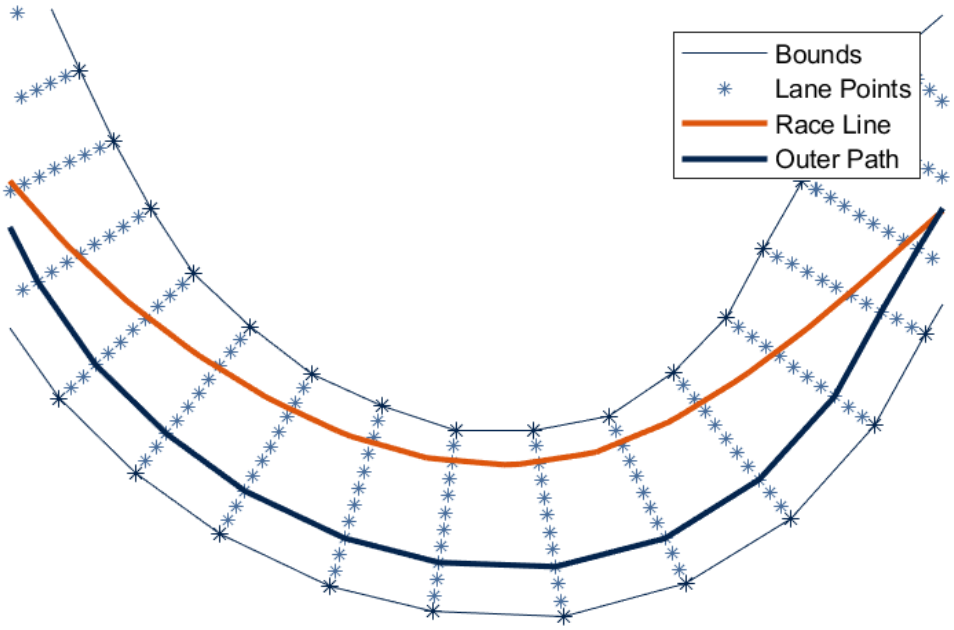


Figure 3.17: Modena Road Course Closeup with Courser Layer Spacing

Since the path between nodes is approximated as a straight line, the relative sparseness of the layers results in a path with clear straight line segments. The path-smoothing method from this planner may have an adequate result. Alternatively, a more elegant solution that integrates the curvature of the road could also be found. Another possible solution is to keep the density of nodes high enough to wave away the problem, but with a higher density of nodes comes a longer computation time.

Another major issue with translating this planner to a road course is that the ideal racing line does not follow any specific lane, rather it follows a line between grid points at different layers. This is more apparent in Figure 3.18. If the planner were to use the graph and desired path as shown, the algorithm would try and approximate the race line to the closest graph point at every layer. The result would be a jagged path such as the one shown in blue in Figure 3.19. The approximation could be adequate enough for the straight sections, it would not hold up for the curved sections where the affect of the approximation is even greatly exaggerated.

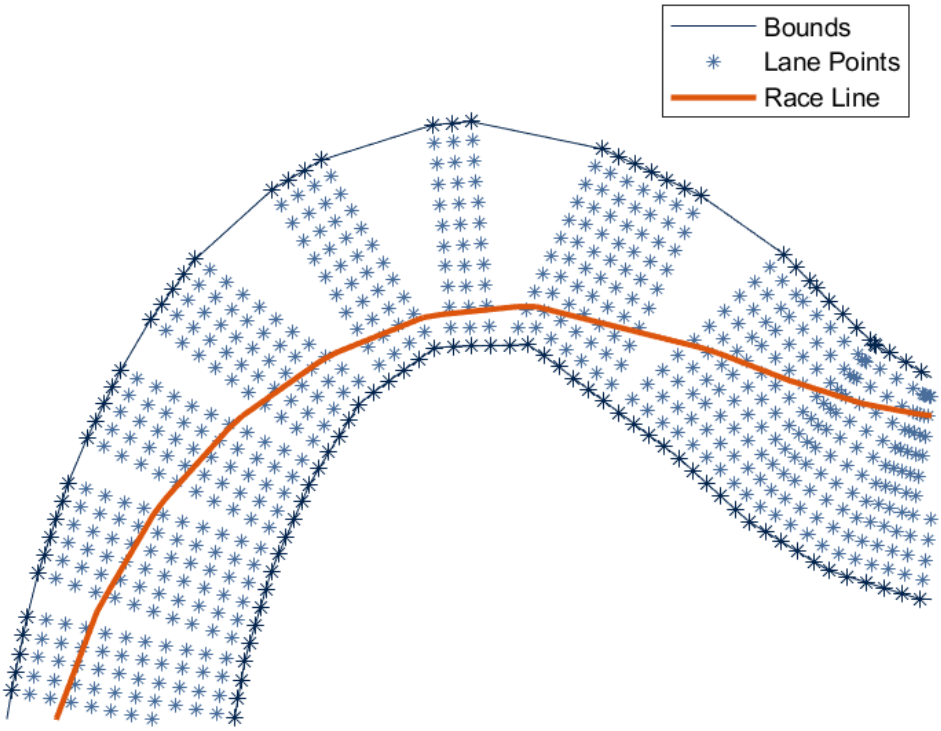


Figure 3.18: Modena Road Course

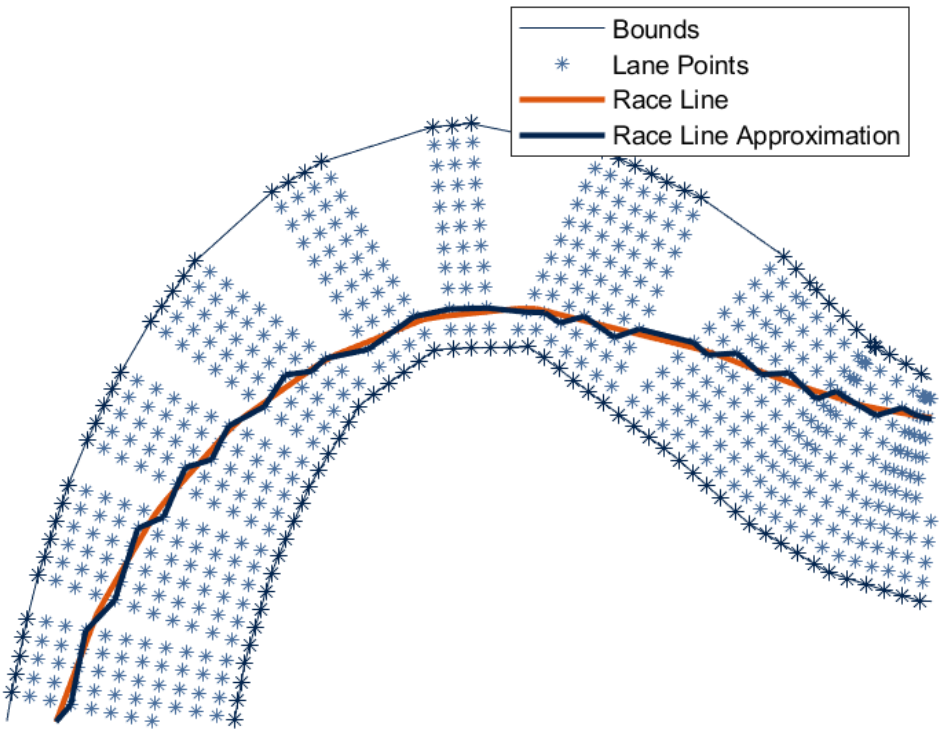


Figure 3.19: Modena Road Course

This problem is not something that can be easilty ignored either a path-smoothing technique or higher density of nodes. The graph would need to integrate the race line into each layer. Although it might not be difficult to do such an integration, is not part of this iteration of the path planner.

Chapter 4

Graph Based Planner for Road Courses

This chapter addresses various concerns raised about the usability of the planner as the complexity of the racing increases such as racing on road courses. These changes include changes in the layer generation, the edge generation, and an addition of a velocity profile generation. For a more detailed description of the planner's architecture, see Appendix A.2.

4.1 Layer Generation

With the previous iteration of the planner having a uniform grid around the tack (i.e. the same number of nodes in each layer, with each layer even spaced around the track) shown to be too simplistic of an approach with more complex tracks, a new layer generation scheme is devised. The first is a more sophisticated way of generating the track bounds using the *Global Race Trajectory Optimization* tool produced by the Technical University of Munich's (TUM) Institute of Automotive Technology [91, 92].

The first step in using this tool is to is obtain data about the track that is being produced. This data in the form of: $[X_m, Y_m, width_{right}, width_{left}]$ where X_m and Y_m are local coordinates of a line around the track. For example, the line could be the right bound of the track. The input file consists of points along that bound with the width of the track populating the $width_{left}$ column and $0m$ populating the $width_{right}$ column. Other viable lines could be points along a driven route around the track, as long as accurate offset data is also acquired. The output of the tool is the array with columns: $[X_{ref}, Y_{ref}, w_{right}, w_{left}, x_{norm}, y_{norm}, \alpha, s, \psi_{raceline}, \kappa, v_x, a_x]$, which are defined below.

- X_{ref}, Y_{ref} (m) - the coordinates (in the same frame as the input file) of the *reference line*
- w_{right}, w_{left} (m) - the distance from the reference line to the right and left bound along the normal vector
- x_{norm}, y_{norm} (m) - the coordinates of the normalized normal vector based on the reference line point
- α (m) - representative of the lateral shift from the reference line to the optimal race line
- s (m) - the curvilinear distance along the race line
- $\psi_{raceline}$ (rad) - the heading of the race line with zero being north in an NED frame
- κ (rad/m) - curvature of the raceline
- v_x, a_x $(m/s, m/s^2)$ - the target velocity and acceleration, respectively, at the given point based off pre-set parameters (*note:* these values will be ignored since many of the input parameters that are used to calculate it are unknown)

Using the values above, a few more details about the track can be extracted using Equations (4.1-4.4) [91][92].

- Right Bound Coordinates, x_{right}, y_{right}:

$$[x_{right}, y_{right}] = [x_{norm}, y_{norm}] * w_{right} + [X_{ref}, Y_{ref}] \tag{4.1}$$

- Left Bound Coordinates, x_{left}, y_{left}:

$$[x_{left}, y_{left}] = -[x_{norm}, y_{norm}] * w_{left} + [X_{ref}, Y_{ref}] \tag{4.2}$$

- Grid Theta, θ_{grid}:

$$\theta_{Grid} = atan(y_{norm}/x_{norm}) \tag{4.3}$$

- Race Line Coordinates, $x_{raceline}, y_{raceline}$

$$[x_{raceline}, y_{raceline}] = [x_{ref}, y_{ref}] + \alpha[x_{norm}, y_{norm}] \tag{4.4}$$

An example of the tool's out put when track information from the road course at Modena can be seen in Figures 4.1 and 4.2.

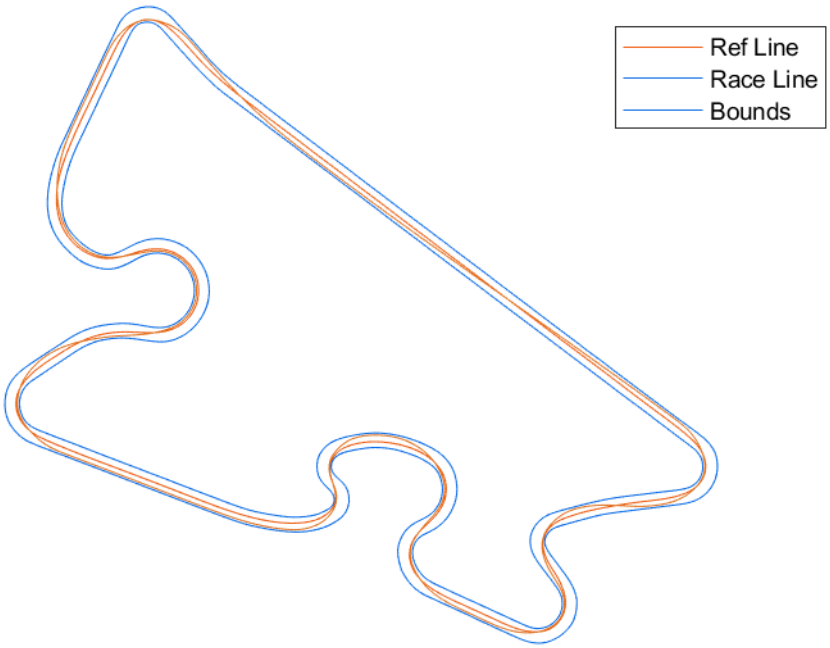


Figure 4.1: Modena Race Track Output from the Global Race Trajectory Optimization Tool

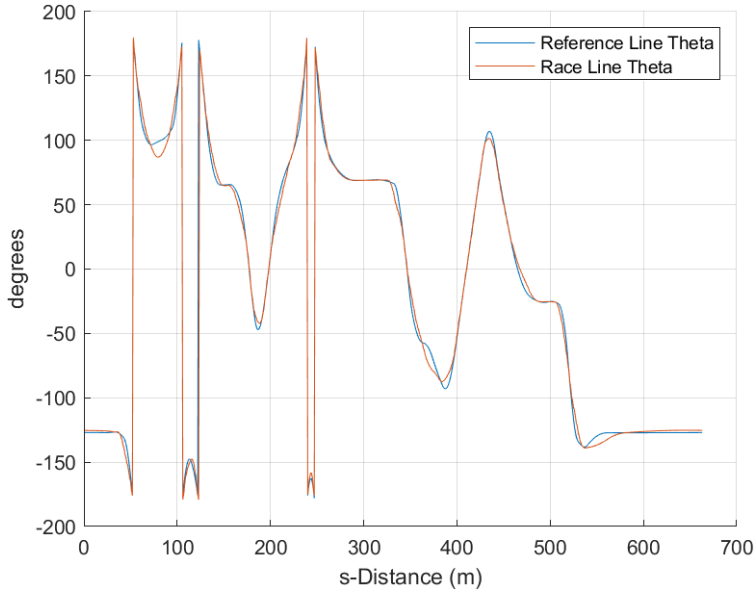


Figure 4.2: Modena Thetas from the Global Race Trajectory Optimization Tool

The next step for the layer generation is to find the points along the track for the layers. Previously, the layer distance is consistent around the entire track. However, this iteration of the planner allows for different spacing based off the curvature of the track or race line. For sections that have a radius of curvature below a predetermined threshold, layer distance can be closer together to allow for a denser area of nodes. For sections that are straighter, the distance between layers can be set farther apart. The threshold for the tighter layer distance can be tuned based off the track, vehicle parameters, or the specific needs of the planner.

A close up example from a road course in Berlin can be seen below in Figure 4.3. The blue lines indicate the straight layer distance, which is chosen to be 10 meters apart for this scenario. The orange lines represent the layers that were designated as tighter curves by having a radius of curvature less than 0.0052 rad/m (0.300 deg/m) and has a separation distance of 5 meters. The spacing distance is relative to the distance along the inside bounds.

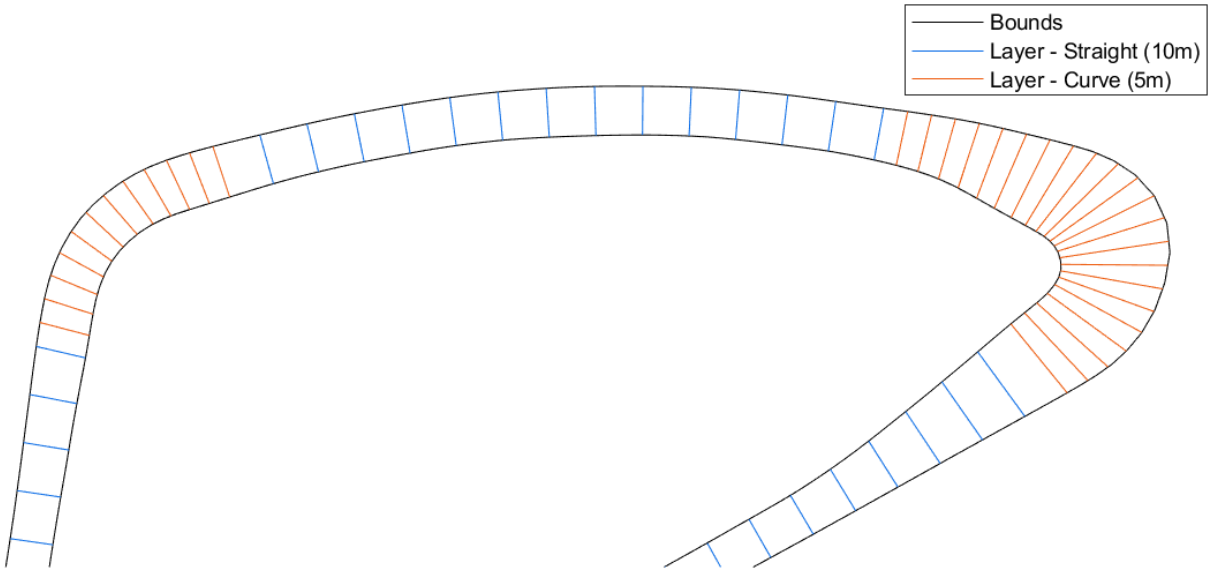


Figure 4.3: Layer Spacing

For each layer, lane points were generated. Previously, the number of lanes in each layer were consistent, however, in tracks such as the Modena Circuit, the width varies from 6.9 meters at its narrowest and 23.2 meters at it widest. If there is a set 5 lanes per layer, the narrowest section would have lane spacing of 1.38 meters and the widest would have 4.65 meter spacing. Instead, this planner uses a lane separation distance parameter to determine how many lanes are in each layer. Also, to correct the issues with estimating the desired path,

which would be the race line in this case, the race line itself is inserted as a node point. As seen in Figure 4.4.

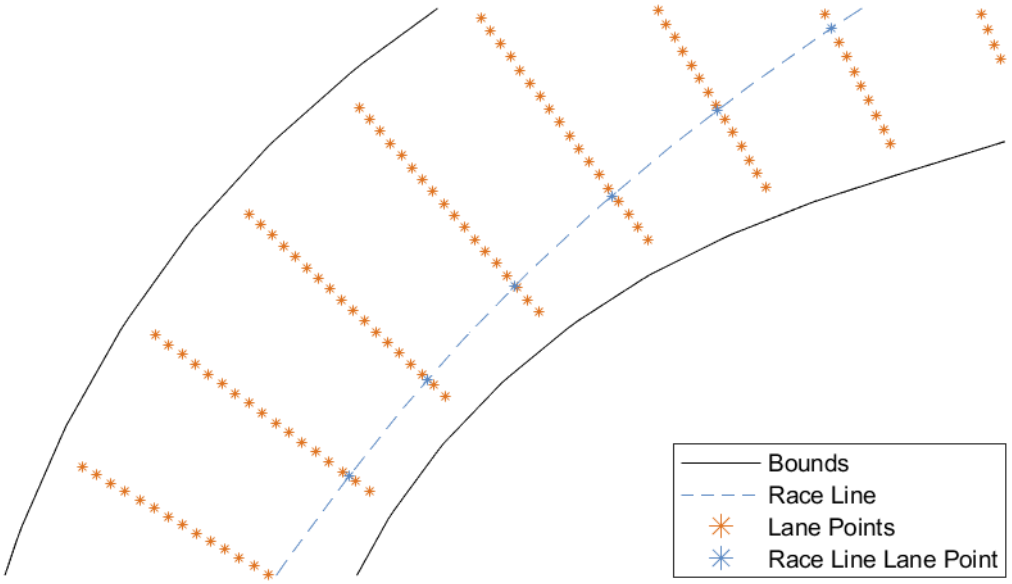


Figure 4.4: Lane Points Generation

Another parameter in this planner is the offset from the inside and outside bounds as shown by the fact the nodes do not quite reach the actual track bounds. This can be tuned based on knowledge of the track. For example, if the edges are exactly at the edge between the banking and the skirt that need to be avoided or an additional layer of safety is needed to keep the car from drifting into a wall.

There is one more calculation made at this step, which the angle of the node. This angle is important for the next step and represents the end heading of the previous edge and the beginning heading of the following edge to that node. It also can greatly affect the splines produced, which will be explored in more detail in the next section.

There are a couple of option in choosing the node angle for any particular node. The first is to use θ_{grid} for the boundary at each node in that layer. An example of doing so can be seen in Figure 4.5. In this example, the vectors for each node are represented by the orange arrows. Also displayed are the vectors corresponding to the grid (in black) and the race line (in blue). However a problem arises upon a closer look at the nodes along the race line. The vector out of a race line node in this example is notable different than the ideal vector from the race line

itself. This would mean that it would be almost impossible to guide the car back on to the ideal path.

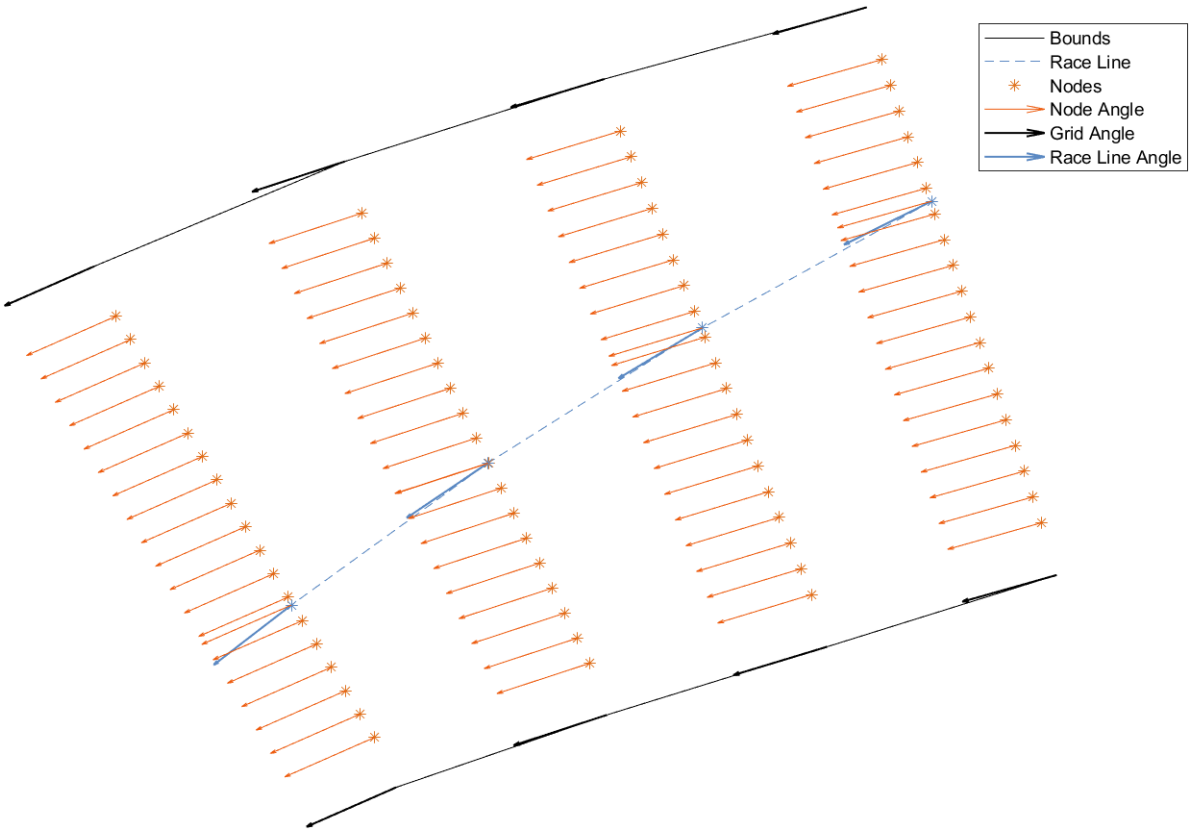


Figure 4.5: Node Theta - Grid Theta

The next option for choosing θ_{node} would be to use $\psi_{raceline}$, or the heading of the race line itself. An example of this can be seen in Figure 4.6. While this might initially appear to be a better solution for nodes nearer the race line, the farther out the node is form the race line, the more extreme the difference between successive nodes are layer to layer. In other words, if the planner needed a path that went along the upper bound, the edges produced would basically be unfeasible due to the heading requirements of that edge.

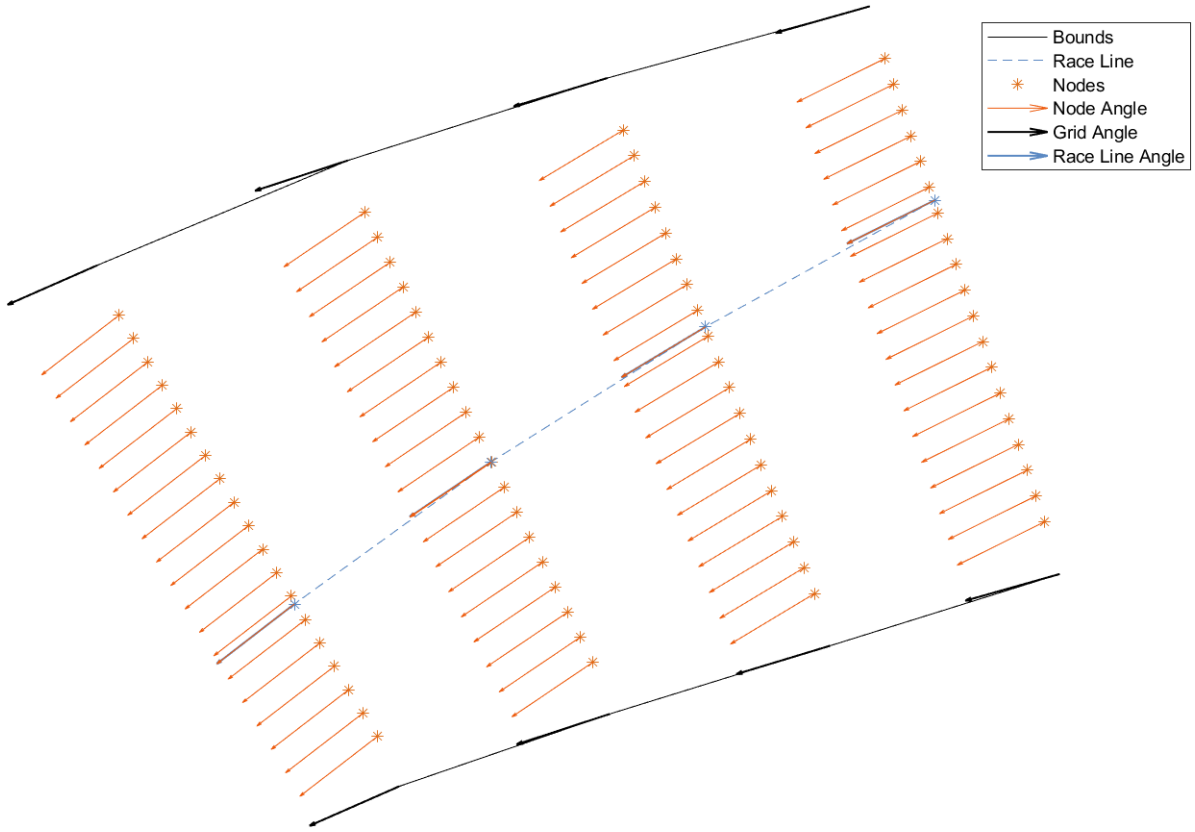


Figure 4.6: Node Theta - Race Theta

In the end, the solution is to use a mix of both θ_{grid} and $\psi_{raceline}$, as seen in Figure 4.7. The angle of the node is linearly interpolated between θ_{grid} and $\psi_{raceline}$ based on the nodes distance along the vector between the right bound to the race line point, with $\psi_{raceline}$ as the theta for the race line. Then it interpolated from $\psi_{raceline}$ to θ_{grid} to the left bound. Overall, using a variable theta for each layer creates smoother transitions at the nodes and allows for more dynamically feasible edges.

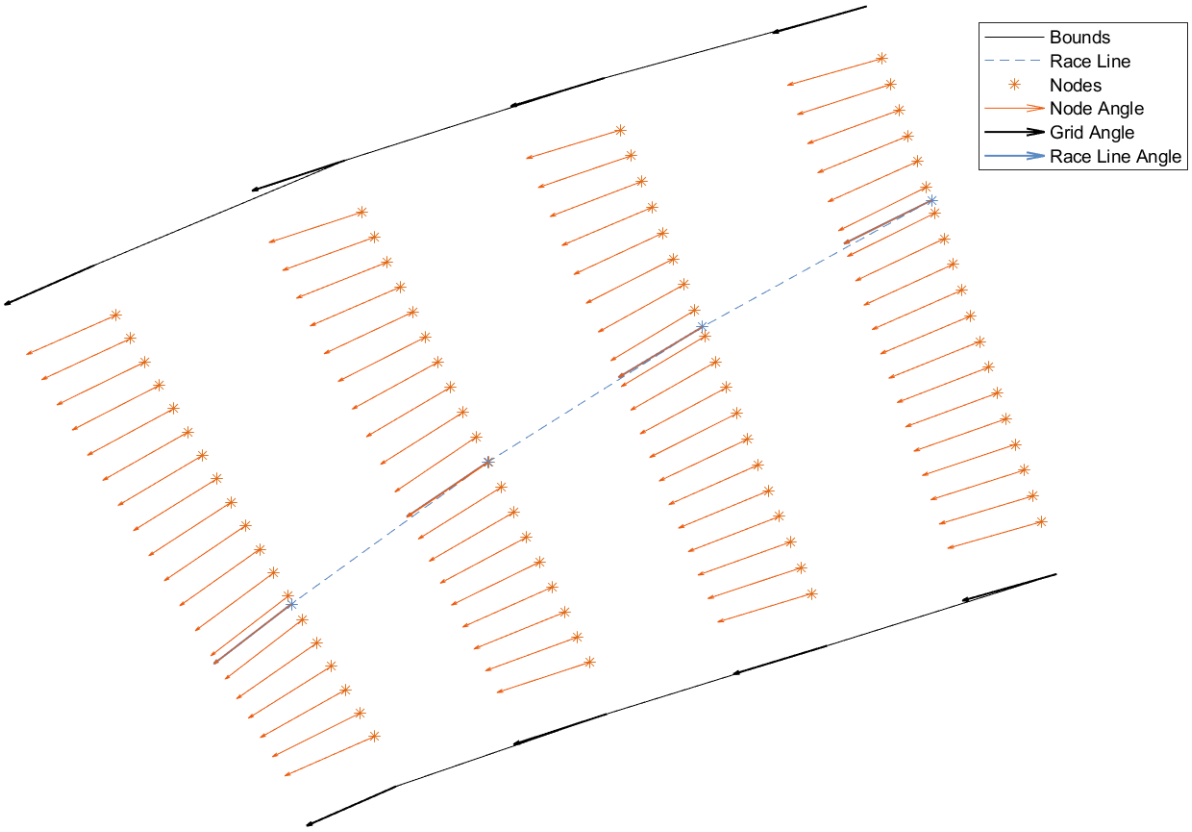


Figure 4.7: Node Theta - Variable Theta

4.2 Spline Generation

For the edge generation in this planner, a cubic polynomial is selected as the basis for the splines as opposed to straight lines between node edges. The cubic spline guarantees C_1 continuity between nodes. The basis for this spline generation is found in [93]. The general formula is described below in Equation (4.5).

$$\begin{cases} x(\mu) = a_{3,x}\mu^3 + a_{2,x}\mu^2 + a_{1,x}\mu + a_{0,x} \\ y(\mu) = a_{3,y}\mu^3 + a_{2,y}\mu^2 + a_{1,y}\mu + a_{0,y} \end{cases} \tag{4.5}$$

where $\mu \in [0, 1]$ and represents points along the path. For example, if two nodes are approximated 10 meters apart and a $\mu = [0, 0.1, 0.2...1.0]$ is used, the spline points produced would be about a meter apart. So if a denser amount of points are need, a more granular μ could be selected (i.e $\mu = [0, 0.01, 0.02...1.0]$ for ten times the amount of points per edge). The coefficients, $a_{3,(x,y)}, a_{2,(x,t)}, a_{1,(x,y)}, a_{0,(x,y)}$, satisfy Equations (4.6-4.9).

$$\begin{cases} x_{start} = a_{0,x} \\ y_{start} = y_{0,y} \end{cases} \tag{4.6}$$

$$\begin{cases} x'_{start} = s_{len}cos(\theta_s) = a_{1,x} \\ y'_{start} = s_{len}sin(\theta_s) = a_{1,x} \end{cases} \tag{4.7}$$

$$\begin{cases} x_{end} = a_{3,x} + a_{2,x} + a_{1,x} + a_{0,x} \\ y_{end} = a_{3,y} + a_{2,y} + a_{1,y} + a_{0,y} \end{cases} \tag{4.8}$$

$$\begin{cases} x'_{end} = s_{len}cos(\theta_e) = 3a_{3,x} + 2a_{2,x} + a_{1,x} \\ y'_{end} = s_{len}sin(\theta_e) = 3a_{3,y} + 2a_{2,y} + a_{1,y} \end{cases} \tag{4.9}$$

The terms $[x_{start}, y_{start}]$ and $[x_{end}, y_{end}]$ represent the coordinate points of the starting node and ending node, respectively. $theta_s$ and $theta_e$ represent the node theta discussed in the previous subsection. s_{len} represents the length of the spline which will differ from the distance between the two nodes. However, the solution to this set of equations has to be iteratively calculated because the actual length of the spline is not known at first and is initialized as the distance between the nodes. So, coefficients are solved for, and the length of the spline is calculated and the coefficients are then re-solved for. In this planner's implementation, the number of iterations continues for a set number of times or until the differences in lengths between iterations is lower than a set threshold.

Using the third-degree polynomial splines allows for edges that closer align to the desired path, as shown below in Figure 4.8. The blue line represents the spline produced along the race line nodes while the orange dashed line is the desired path. While it does not line up exactly, the variations are negligible, even less so in segments of lower curvature which form a seemingly smooth line through the node transitions.

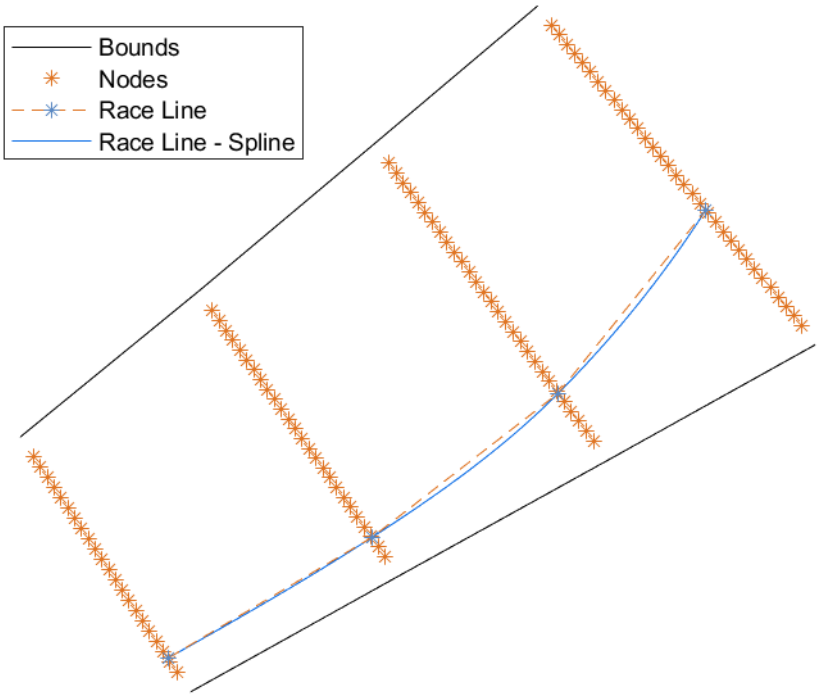


Figure 4.8: Spline along race line

As previously mentioned, the node thetas have strong influences on the behaviors of the splines (seen in Equations (4.7) and (4.9)). In Figure 4.9, the splines produced from using the corresponding $theta_{grid}$ for both θ_s and θ_e. In this particular section of the track, the spline along the desired path shows a notable offset along the third segment and appears to be oscillating. This is due, in part, to a large enough difference between the θ_{grid} and $\psi_{raceline}$.

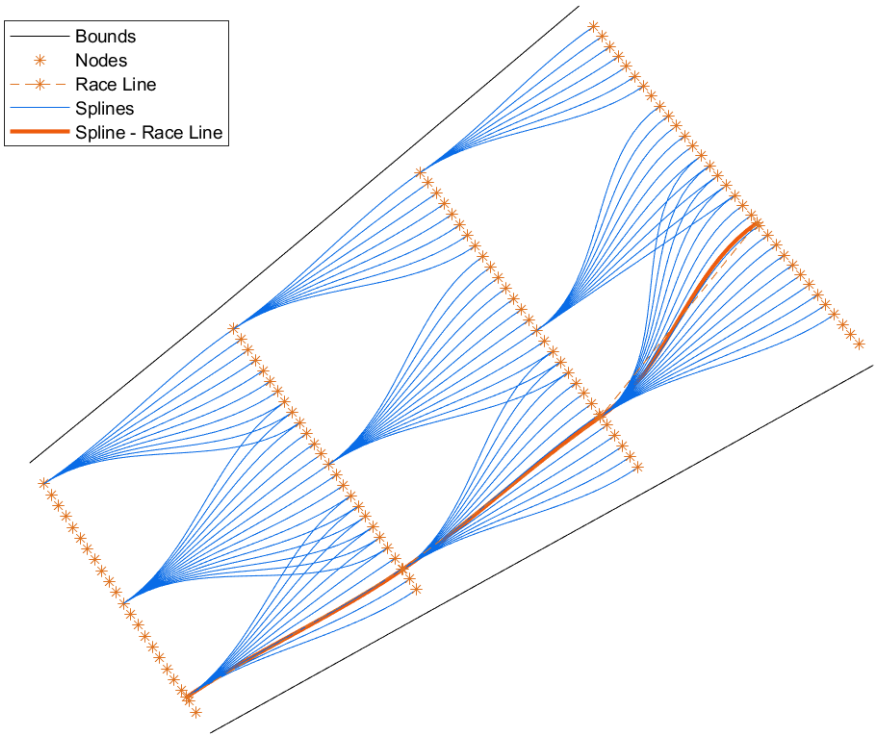


Figure 4.9: Splines with θ_{grid} as θ_{node}

The effects of using just the corresponding $\psi_{raceline}$ for both thetas is shown in Figure 4.10. While the oscillation and offset of the race line are not present along the desired path's splines, the splines farther away from the race line show a noticeable problem. Particularly notable are the splines in the third segment. For both ends of the nodes, the corresponding $\psi_{raceline}$ and θ_{grid} are very different as the race line is about to enter a tight turn and is beginning its transition from the outer line to an inner line. Figure 4.11 shows the results of using the varying theta as the node theta, combining the better parts of the previous two options.

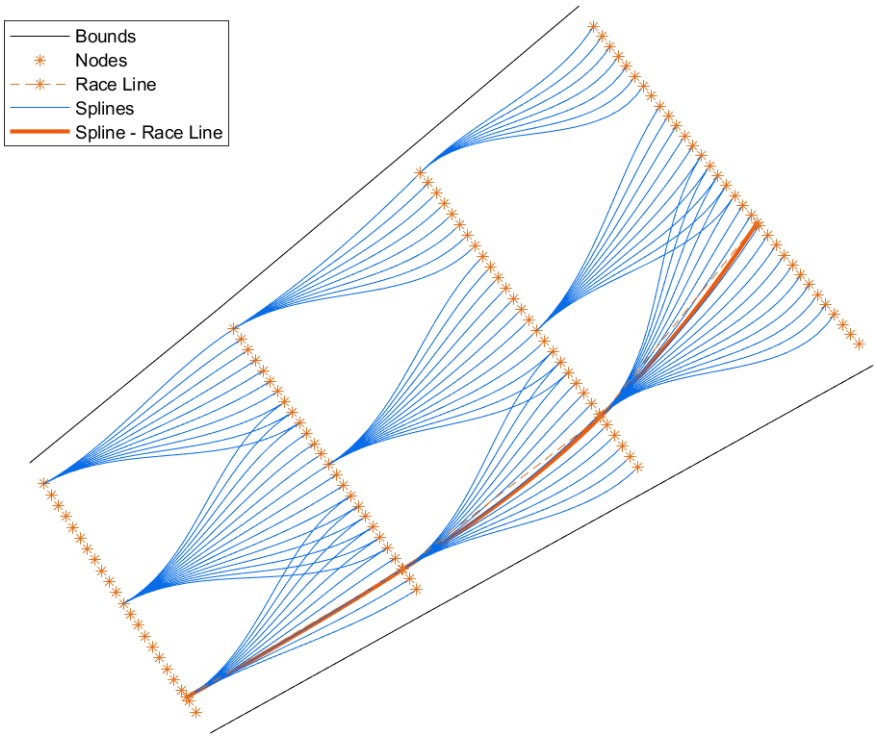


Figure 4.10: Spline with $\theta_{raceline}$ as θ_{grid}

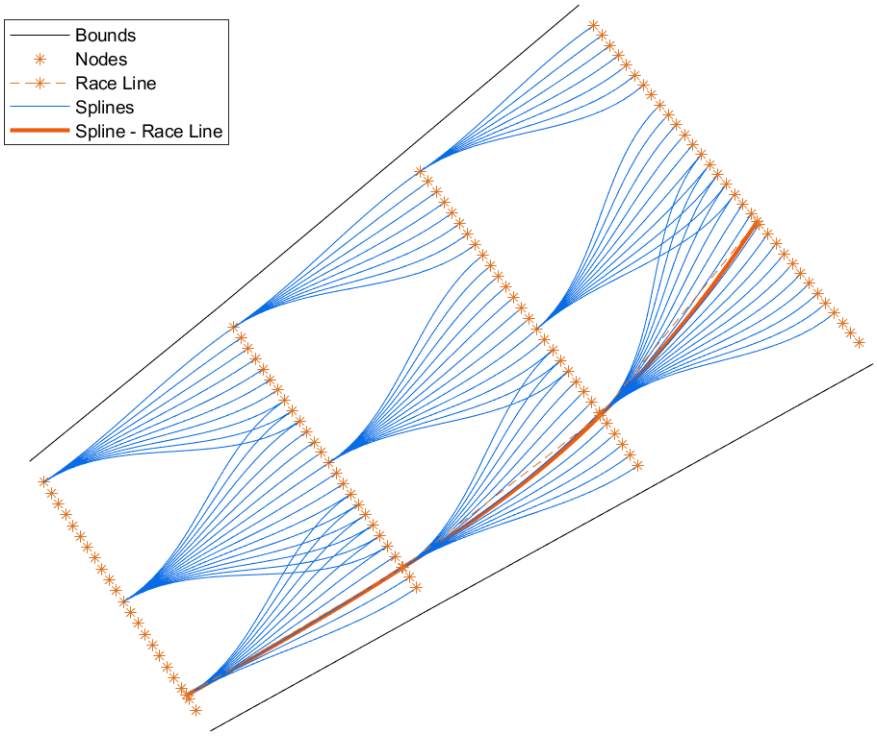


Figure 4.11: Spline with variable θ as θ_{grid}

The last step in the spline generation is to check if the spline is viable, and if so add it to the graph as an edge. In the previous figures, only a few splines were shown as to not crowd

the image. However, in the planner, a spline is produced from every point in each layer to every points in the next layer, as seen in Figures 4.13 and 4.14, but not every spline is kept. Since the coefficients describing the spline are known, the radius of curvature can be accurately calculated with Equations (4.10) and (4.11).

$$r_{curve} = |\frac{1}{\kappa}| \tag{4.10}$$

$$\kappa_{spline} = \frac{x' * y'' - y' * x''}{\sqrt{(x'^2 + y'^2)^3}} \tag{4.11}$$

The coeffiecients x', y', x'', y'' are calculated in Equations (4.12) and (4.13).

$$\begin{cases} x' = 3a_{3,x}\mu^2 + 2a_{2,x}\mu + a_{1,x} \\ y' = 3a_{3,y}\mu^2 + 2a_{2,y}\mu + a_{1,y} \end{cases} \tag{4.12}$$

$$\begin{cases} x'' = 6a_{3,x}\mu + 2a_{2,x} \\ y'' = 6a_{3,y}\mu + 2a_{2,y} \end{cases} \tag{4.13}$$

The radius of curvature, is defined as the reciprocal of the *curvature* where curvature is a value that describes how much a line deviates from being "straight" [94]. It can be seen as the radius of the circle that *comfortably* fits to that point, as seen in Figure 4.12. This value is important because one of a vehicle's characteristic is the minimal radius of curvature that it is able to feasibly drive. For each spline produced, the minimal radius of curvature (or maximum kappa) is found and compared to the vehicle's minimal radius of curvature. If the r_{curve} is less than the vehicle's turning radius then the spline is deemed dynamically infeasible to traverse and is not included in the graph.

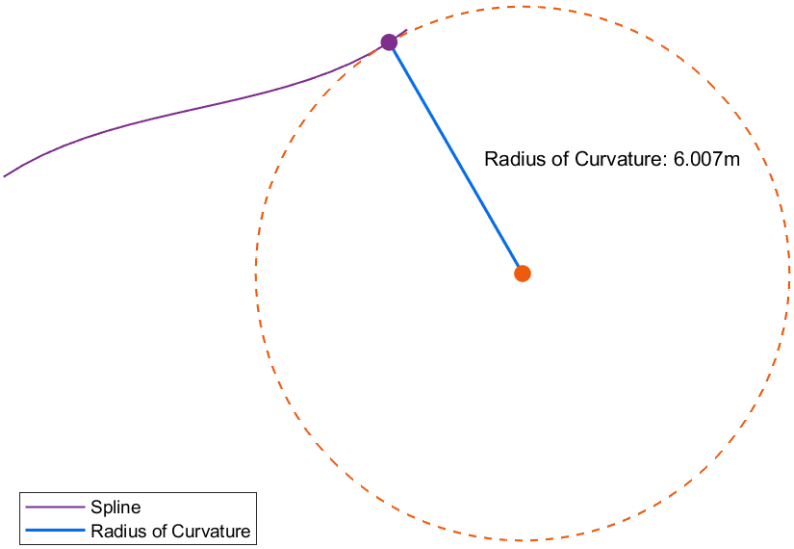


Figure 4.12: Radius of curvature

In Figures 4.13 and 4.14, each spline that is produced and tested are shown along with their maximum κ and minimum r_{curve}. In this example, the vehicle's radius of curvature is set to 5 meters. The splines that are kept are in orange, and the ones that are not traversible are plotted in blue.

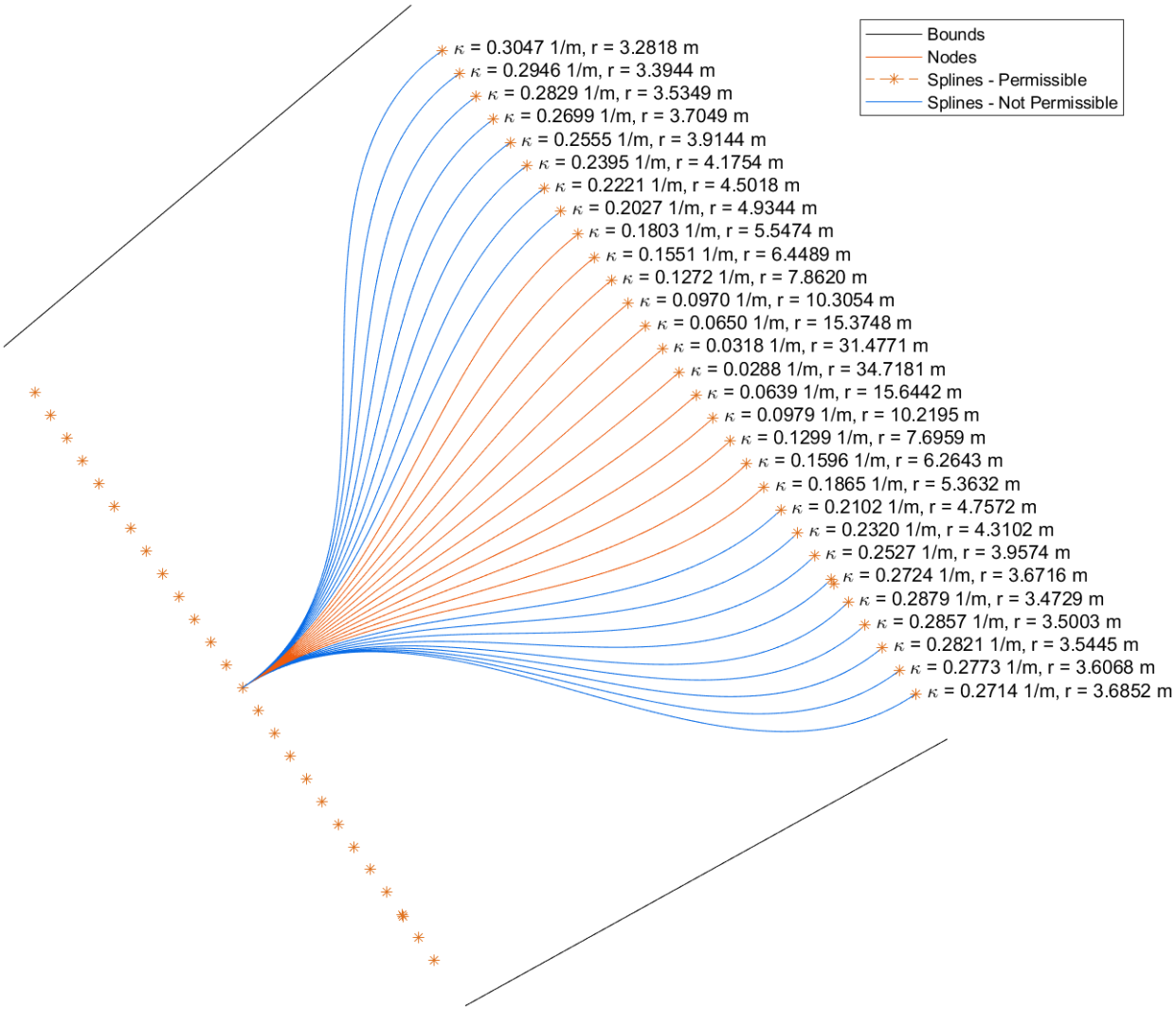


Figure 4.13: Splines from node at the middle of the graph

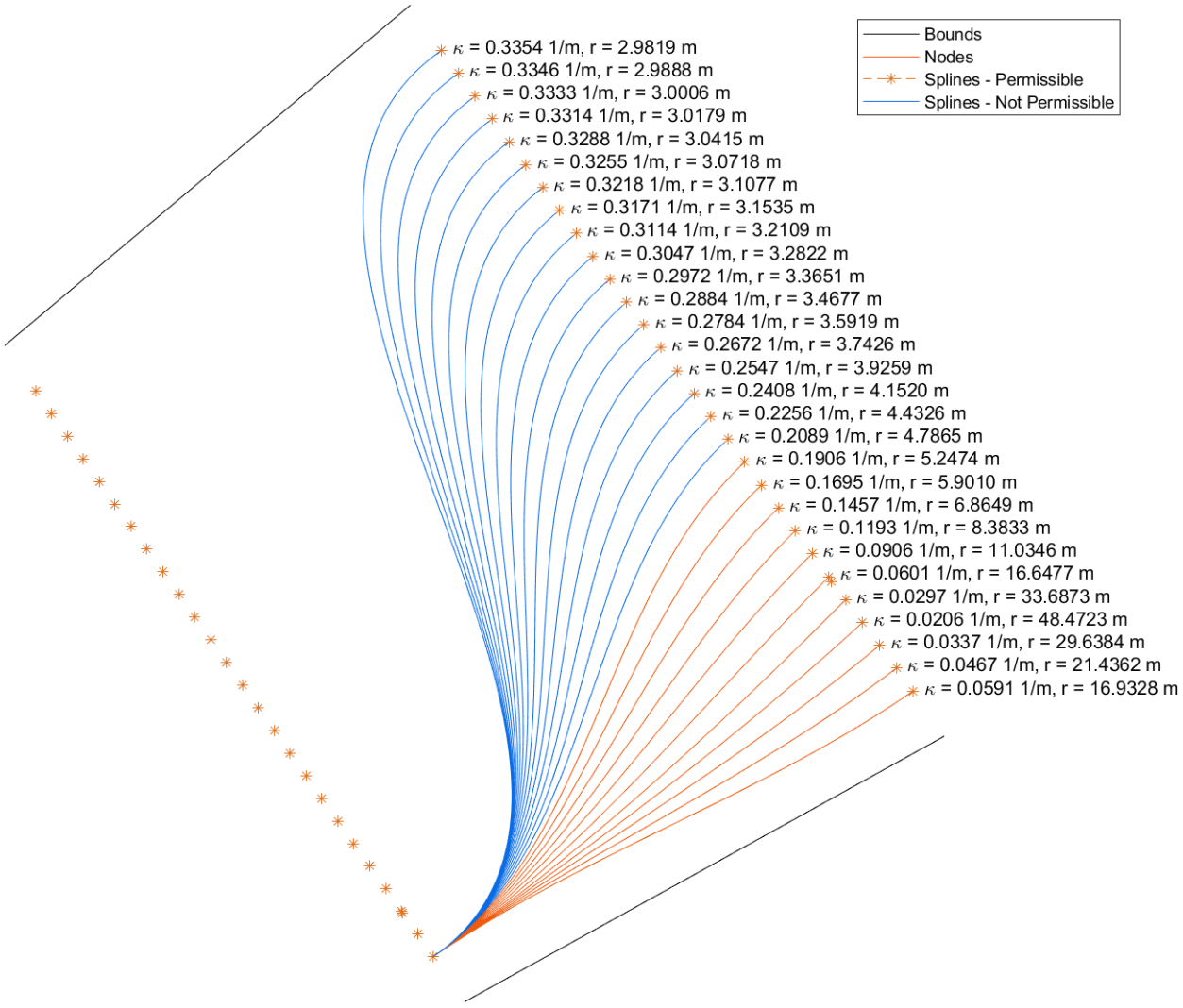


Figure 4.14: Splines from node at the end of the graph

Finally, because not all splines are considered admissible, there may be potential paths that lead to ”dead ends”. Without pruning the splines, the graph would look like Figure 4.15a. To prune the dead ends, the graph is iterated through to first check if there are any edges that have no layers leading to it and if so are removed, resulting in Figure 4.15b. Then edges that have no edges going from it are removed. In the end, the graph edges look like those in Figure 4.15c.

In the example shown in Figure 4.15, while Figure 4.15a has 1,296 edges, Figure 4.15b has 904, and Figure 4.15c has 653. Therefore, half of the splines have been pruned for this 20-layer section for the graph. The high rate of pruning is more prevalent in areas of the graph with higher curvature. In the end, the pruning will lead to a quicker A* search because it will

never explore a dead end and guarantees a path from the start to the goal as long as the nodes with no edges are removed as options for the goal node.

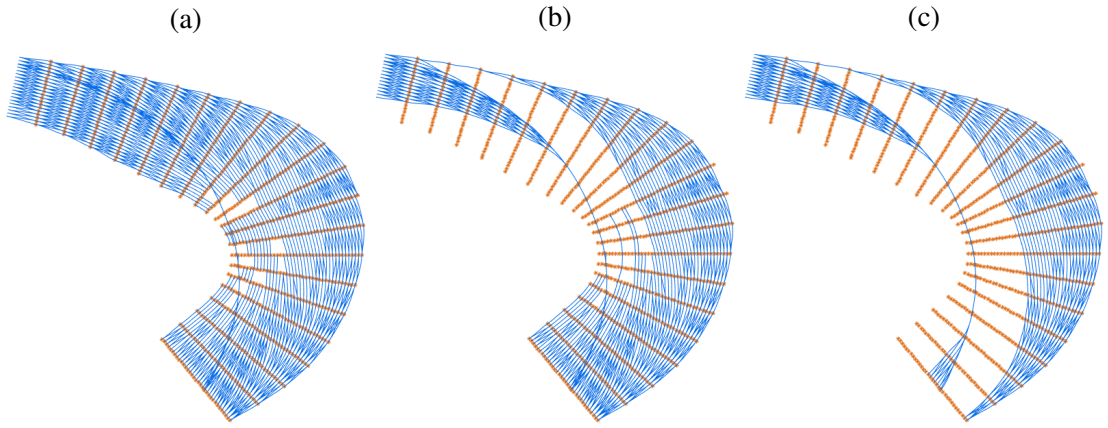


Figure 4.15: (a) unpruned, (b) starts pruned, (c) ends pruned

4.3 Velocity Profile Generation

4.3.1 Graph Generation - Offline

The final update to this planner is the velocity profile generation. This is done in two parts, during the graph generation and during run-time. During the graph generation, the goal is to characterize the potential velocity profiles of each edge such that once the path is produced, desired velocities along the path can be selected, as seen in Figure 4.16. In this example, the start and goal node are set to the start/finish line and the path produced is a race line free of objects around the track. The desired velocity profile of the path is plotted as a gradient of the value of the velocity at that point according to various parameters that will be discussed later in this section.

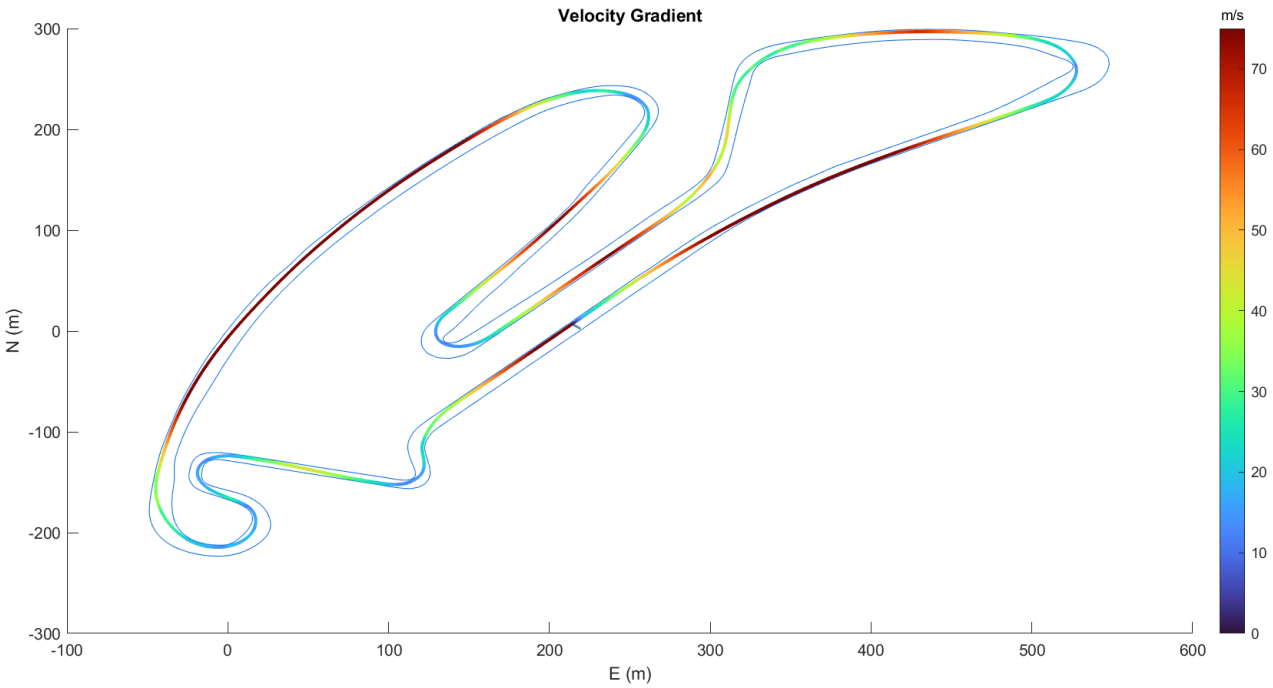


Figure 4.16: Velocity Profile of Starting Lap

However, before the velocities can be selected the velocity profiles of each edge needs to be produced. This part is done along with the spline generation. The first step is finding the maximum velocity allowed along the spline, $vel_{max,spline}$, by using Equation (4.14).

$$vel_{max,spline} = min\left(vel_{max,car}, \sqrt{\alpha_{max,car}/\kappa}\right) \tag{4.14}$$

which is the minimum between the maximum velocity of the car, a parameter of the particular vehicle, and the max acceleration of the can be divided by the curvature. A vector of potential velocities along that spline is created based off the vel_{step} parameter. For example, if $vel_{step} = 1m/s$ and $vel_{max,spline} = 20m/s$, the vector would be $vels_{spline} = [0, 1, 2, ...vel_{max,spline}]$. Next, the maximum lateral acceleration, $\alpha_{max,lat}$, along the spline is found using Equation (4.15).

$$\alpha_{max,lat} = v^2\kappa \tag{4.15}$$

where $\alpha_{max,lat}$ is an array of maximum lateral accelerations that correspond with each velocity in vel_{spline}.

The longitudinal acceleration can be derived next. Typically when considering the limits of a vehicle, understanding the tire friction circle is key. In short, the friction circle represented the "force-producing limit of the tire" under certain conditions [95]. The vertical axis represents what that paper calls the *lateral force* while the horizontal axis represents the *longitudinal force* as shown in Figure 4.17, where the orange circle is the friction circle, or the theoretical tire limits of a vehicle of the combined forces (or accelerations). However, finding the limits usually requires extensive testing and knowledge of the tire beforehand, and can quickly change due to various circumstances like weather or tire degradation.

Therefore, a simpler tire friction model was utilized, akin to the blue lines on Figure 4.17, where the friction limits are the additive values of the lateral and longitudinal forces. By choosing a reasonable max force/acceleration for the vehicle, a margin of safety can be established for the tire limits. The maximum longitudinal acceleration is found by using Equation (4.16).

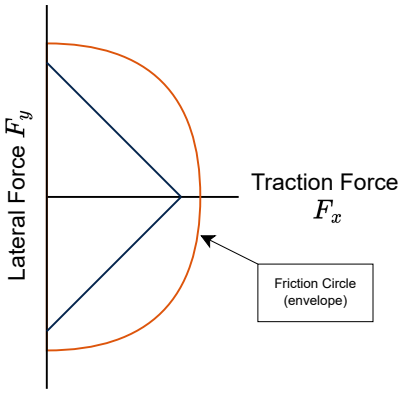


Figure 4.17: Simplified Friction Circle

$$\alpha_{max,long} = \alpha_{max,car} - \alpha_{max,lat} \tag{4.16}$$

Then the maximum total change in velocity over that spline is found using Equation (4.17).

$$\Delta vel = \sqrt{2 * \alpha_{max,lat} * s_{len}} \tag{4.17}$$

The velocity profile range can then be constructed, similar to Equation (4.18). The first column would represented the calculated range of velocities for the previous edge. The second

column represents the lowest velocity possible with maximum deceleration and the third column represents the highest velocity possible with the maximum acceleration. The deceleration is the maximum between the 0 m/s and the starting velocity minus the maximum total change in velocity. The acceleration maximum is the minimum between the velocity limit of the spline and the starting velocity plus Δvel.

$$vel_{range} = [vels_{spl}, max\left(0, vels_{spl} - \Delta vel\right), min\left(vel_{max,spl}, vels_{spl} + \Delta vel\right)] \tag{4.18}$$

These calculations are done for every admissible spline, and a graphical representation of what was calculated can be seen in Figure 4.19. The blue stars represent the $vels_{spline}$ for each of the nodes of the race line. As the line begins to curve, the number of feasible velocities decrease. The purple lines represent the maximum positive change in velocity between nodes based off the previous parameters discussed. The green line represents the maximum negative change in v.

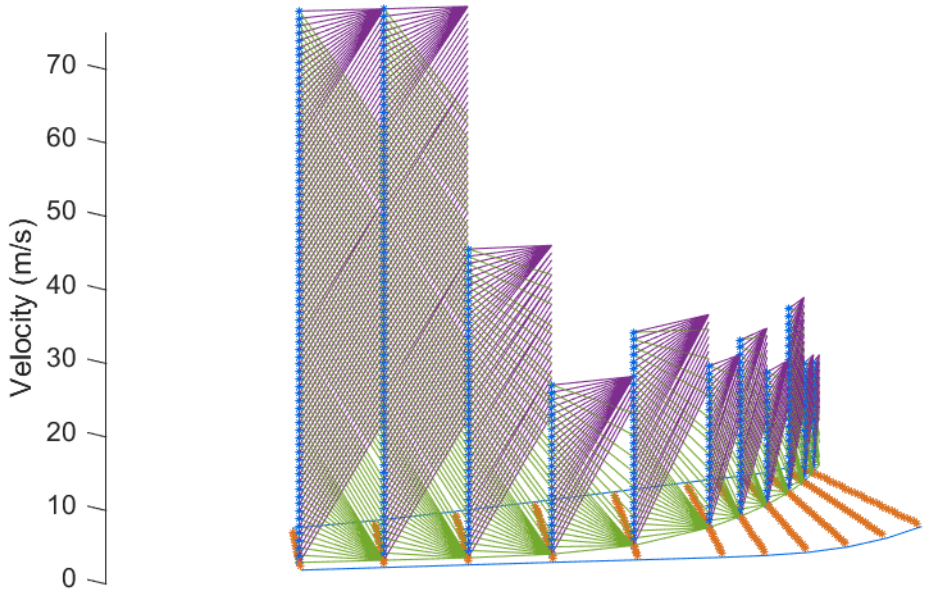


Figure 4.18: Velocity Profiles Along the Race Line

From initial inspection of the profiles shown, an issue becomes apparent once the planner begins selecting desired velocities. Looking at the second and third segments of the profile in Figure 4.19, the second segment is able to span the entire range of vehicle velocity capabilities, which in this case is 75 m/s. However, the third segment has a much smaller maximum radius of curvature and so has a maximum velocity limit of about 45 m/s, much less than previous. If the planner were to select a velocity along the second spline that is above the maximum of the third spline, then once the vehicle gets to the latter it would not be cable of safely decelerating

in time and would likely increase the chances of a crash. In order to keep this from happening, the velocity profiles need to be pruned. In this thesis, the pruning is done during the online portion of the planner.

4.3.2 Profile Selection - Online

Once the planner is initialized and the starting and goal information is passed on, the planner determines if the velocity profile is a *ramp up* or *ramp down* profile. This means that if the starting velocity is lower than the goal velocity, the profile is considered a ramp up profile. On the other hand, if the goal velocity is less than the starting velocity, the profile is considered a ramp down. Each profile has three sub-profiles that were developed for this thesis:

- Ramp Up Profiles
  - *Aggressive* - ramps up to the goal speed as fast as possible
  - *Percent**X*** - increases by X% of the maximum acceleration for each node until the max is reached, scaled by the edge length and layer distance parameter
  - *Steady**X*** - increases speed by X m/s per node until the goal is reached, scaled by the edge length and layer distance parameter
- Ramp Down Profiles
  - *Linear* - reduces the speed linearly from the starting velocity to the goal velocity
  - *Logarithmic* - reduces the speed logarithmically from the starting velocity to the goal velocity
  - *StepTo**X*** - reduces the speed quickly to X m/s, and steps down to 0 m/s during the last couple of nodes, this profile is meant specifically for vehicles coming to a complete stop

After the path is reconstructed from the A* algorithm, the velocities are pruned. This pruning is done such that no ending velocity along a spline is greater than the maximum velocity of the next spline. The result would look similar to Figure 4.19. With this pruning, the planner

would not be able to select a velocity that would be infeasible to decelerate in order to feasibly drive alone the next spline. An example of the profiles along the entire race line can be seen in Figure 4.20.

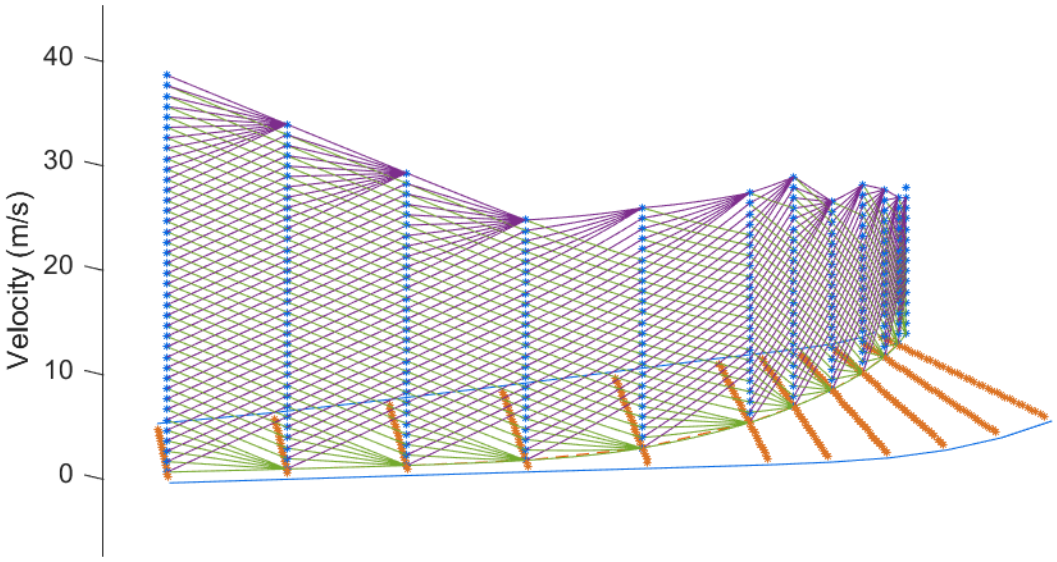


Figure 4.19: Pruned Velocity Profiles Along the Race Line

Figure 4.20: Full Race Profile

The final step in selecting the velocity profile is to choose a velocity associated with each node. The first velocity is the starting velocity or whatever the closest is without being greater. The planner then goes through each node in the path and selects a velocity according to the velocity profile determined in an earlier step in the code. For example, if the velocity profile

is *Aggressive*, then the next velocity selected will be the greatest possible change to the next velocity (or whatever velocity is at the end of the purple lines in the figures). For *Percent##* or *Steady##*, the maximum within the limitations is chosen. So for the percent, the total acceleration and the change in velocity across the edge is calculated and the final velocity is selected for the next node. For *Steady##*, the next nodes velocity will be the minimum between the maximum change in velocity or the velocity step size.

For the ramp down profiles, the *Linear* and *Logarithmic* profiles are pre-computed based off the length of the path and each node is a assigned a desired velocity. A check is run to make sure the velocities are feasible for that node, and if not they are adjusted accordingly. The *StepTo##* acts similar to *Aggressive* profile, except decreasing in velocity. However, it continues to decelerate until it reaches the indicated velocity, in which it holds until the end of the path where it linearly decreases until coming to a stop at the goal node.

After the profile is selected, the velocities associated with each point in the edge are interpolated between the velocities of each node at either end.

Chapter 5

Results

5.1 MKZ Testing

First, two sets of experimental tests were done using a Lincoln MKZ, shown in Figure 5.1, with fully autonomous drive-by-wire capabilities at the Auburn University's National Center For Asphalt Testing (NCAT) . While using the Indy Light's car would have been ideal for testing, due to it being tied up for a future IAC competition, lack of hardware needed, and ideal testing location, the MKZ was the best alternative for this particular setup.

Figure 5.1: Lincoln MKZ used for testing

The MKZ is equipped with various sensors to help with the data collection including a high-grade INS unit that outputs a highly accurate INS/GNSS solution with little error run to run. The vehicle is also equipped with wheel speed sensors were also utilized for an accurate real-time measurement of the vehicle's speed.

5.1.1 Software Setup

Three different desired path outcomes were tested, shown in Figures 5.2-5.4. All three scenarios were set to be an ”emergency stop” type of situation due to the limited space of the skid pad. Due to the space constraint, along with the limitations of the vehicle controller, the maximum speed tested was about 5 m/s (11 mph).

Figure 5.2: MKZ Test #1 Path

Figure 5.3: MKZ Test #2 Path

Figure 5.4: MKZ Test #3 Path

A way-point manager was set up to track where the vehicle was along the desired path. These way-points were taken directly from the output of the path planner and included both coordinates and desired velocities. A simple PID controller was used to control the throttle and breaking. The lateral controller was a discrete heading lead controller, which essentially took the desired way-points and calculated a heading error and then pointed the vehicle towards it [96].

The planner itself was modified slightly to run this particular experiment. The first modification was to the graph generation itself. In the previous chapter, the planner would be a graph based on previously tested and validated track bounds. However, since this particular experimental setup did not have well known bounds, a general graph was generated at the start. This graph had the properties seen in Table 5.1. The longitudinal length was the distance between the first layer and the last layer, forward from the body of the car. The lateral length was the distance to the left and the right of the center lane. Additional parameters for this testing can be found in Appendix C: Table C.1.

This graph moved with vehicle, where the center lane and first layer was always the location of the vehicle. This is shown in Figure 5.5 where each colored set represents the graph as the MKZ moves along the path (black arrow). The green dot represents the ego vehicle at that point in time. Since the general shape of the graph stays the same, the edges and the

Table 5.1: MKZ Testing Parameters

Parameter	**Value**
Longitudinal Length	40 meters
Lateral Length	14 meters
Lane Separation	5 meters
Layer Spacing	.5 meters

nodes were generated at the graph initialization. This allows for algorithm to be run in the body frame. Outside objects ar then rotated from their detected frame into the body frame. Once the desired path is calculated, the path is then rotated from the body frame into ENU coordinates.

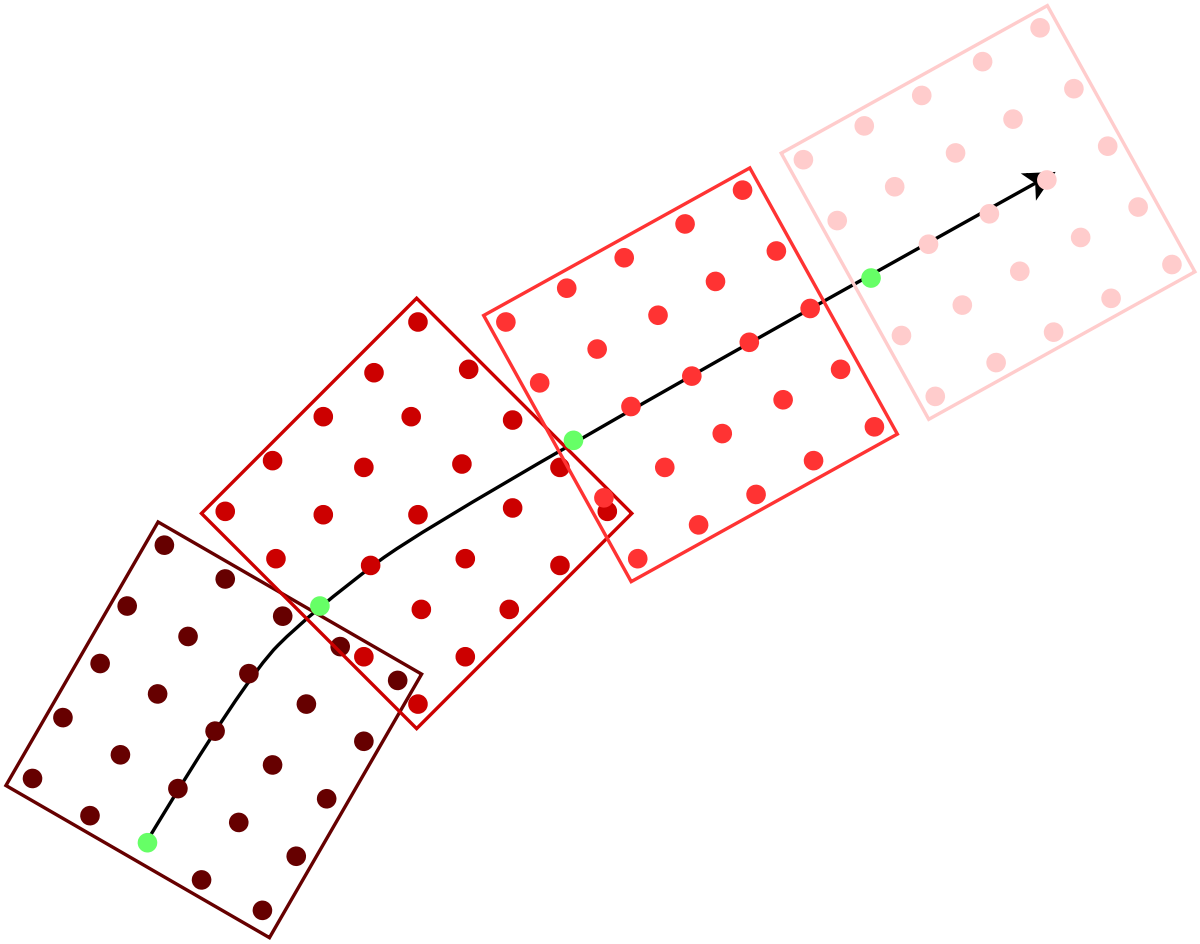


Figure 5.5: Graph Generation of the MKZ

Once an object is detected, the path and the graph become locked in place. In other words, the planner stops updating the path unless new data reflects new previously undetected objects. This path is then sent to the way-point manager and controllers. Freezing the path keeps the planner from continuing to re-plan paths that differ from the original path from the

updated positioning as the ego vehicle continues forward. Since both localization and object detection/prediction are outside of the scope of this thesis, the commercial INS/GNSS solution is taken as truth and objects are spoofed onto the graph with perfect detections.

5.1.2 Results

The results of the three scenarios are shown in Figures 5.6-5.11. The first test is a straight path with no objects spoofed, the second has objects placed such that a "lane change" is forced, and the last scenario has object placements such that a "double-lane change" is forced. The spoofed objects are represented by the purple dots while the light blue line is the planner's outputted path. The red star represents the starting node and the green star represents the goal node. On the left side, the colored line represents the actual path the experimental vehicle actually took along with a gradient of the vehicle's speed at that point. The right side represents the outputted path along with the associated desired trajectory for that point.

In all three scenarios the vehicle was set to have a goal speed of 0 m/s after detecting an object. In all scenarios, the vehicle begins reducing its speed immediately and came to a stop, as expected. However, the vehicle never actually comes to a complete stop at the goal. This was most likely due to rounding errors in the longitudinal controller not allowing a commanded velocity of 0 m/s. Another note, the recorded path, when compared to the planner's path, differs slightly in the second and third tests. There appears to be some over shoot, and the controller does not follow precisely. This error can potentially be attributed to poor tuning of the look-ahead distance in the way-point manager and of the lateral controller. The look-ahead distance refers to how far ahead on the path that the desired way-point is selected. So if the distance is set to 5 meters, the waypoint that is sent to the controllers is the point along the curve that is 5 meters in front of where the vehicle is at that moment.

Despite this, the planner showed that it is able to generate paths. In scenarios where a sudden stop due to a obstacle is needed, the planner is able to react quickly and correctly for both the safety of the ego vehicle and the vehicles around it.

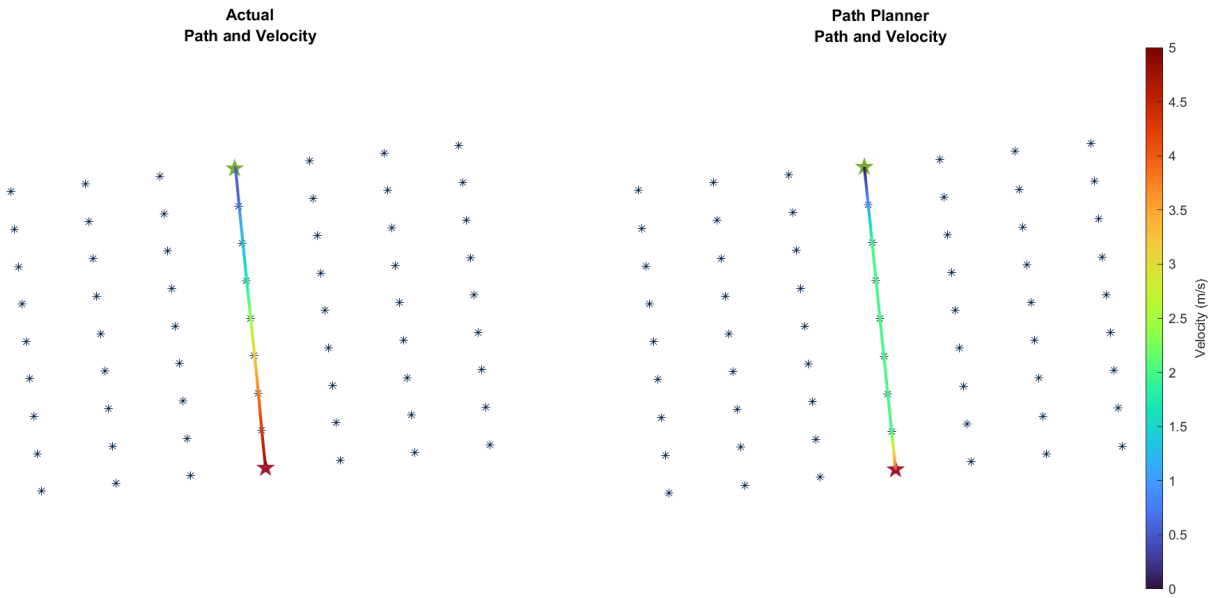


Figure 5.6: MKZ Test #1 Velocity Results

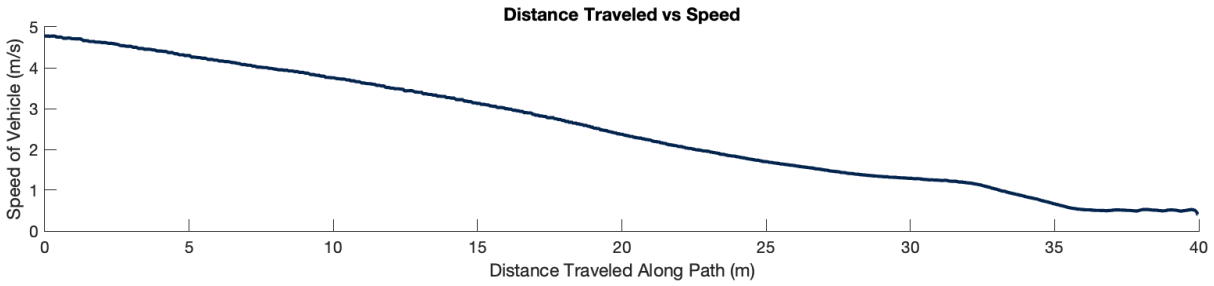


Figure 5.7: MKZ Test #1 Distance Traveled vs Speed

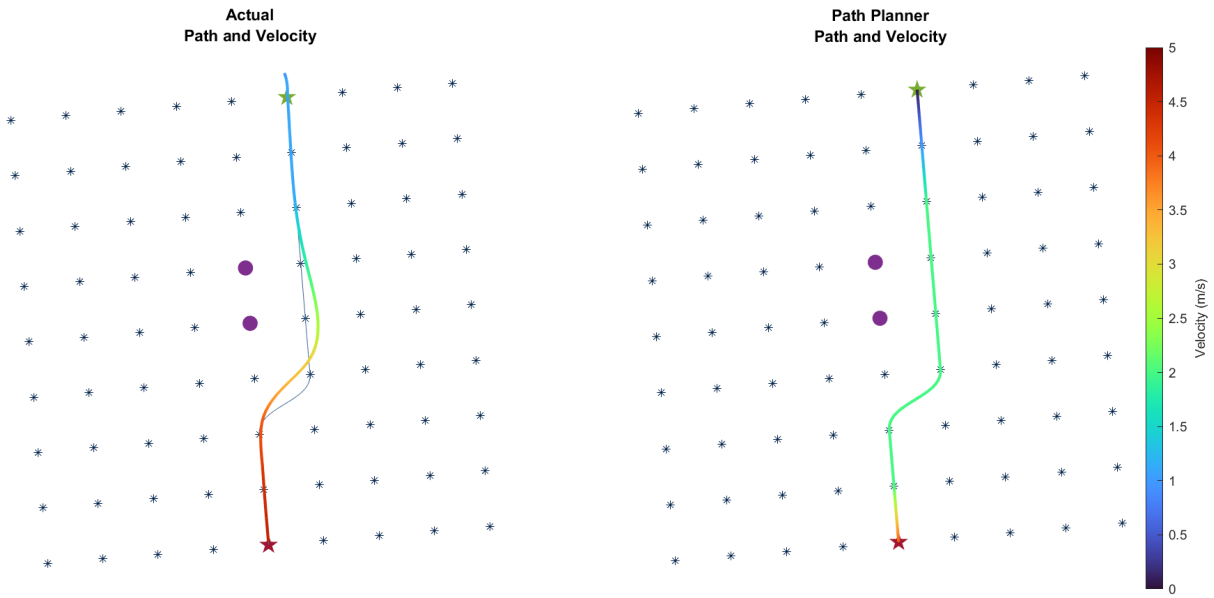


Figure 5.8: MKZ Test #2 Velocity Results

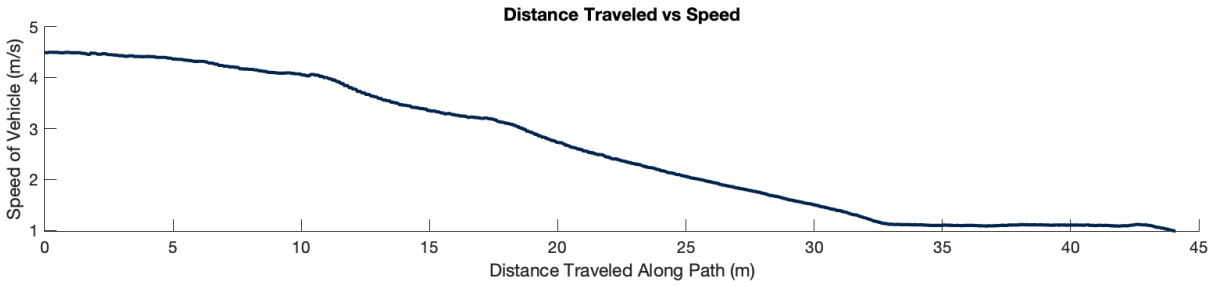


Figure 5.9: MKZ Test #2 Distance Traveled vs Speed

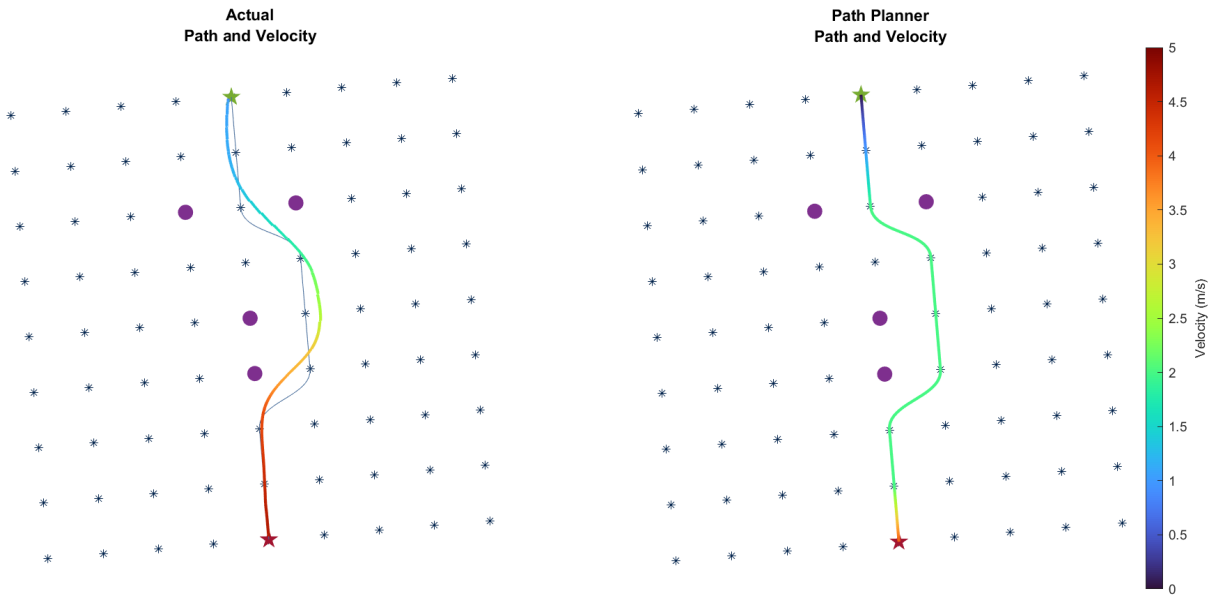


Figure 5.10: MKZ Test #3 Velocity Results

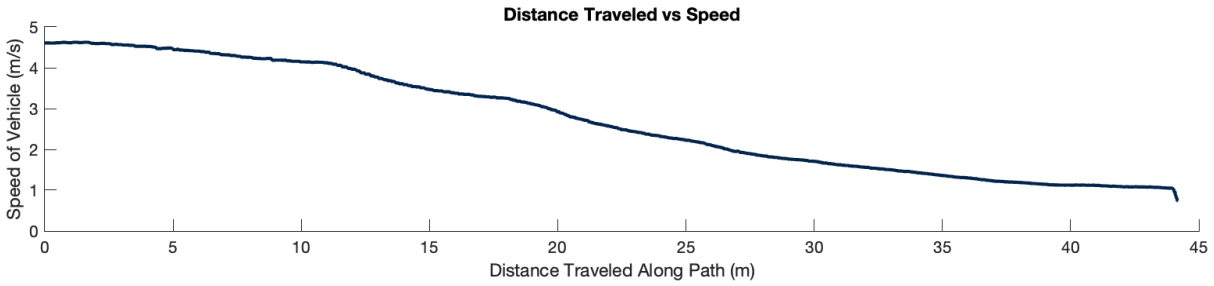


Figure 5.11: MKZ Test #3 Distance Traveled vs Speed

5.2 Simulation Testing

The second set of experimental tests were conducted entirely in simulation because of the practicality of testing the planner on the race car. Instead, testing similar to that done in the first planner was used for this testing. This included both a simple simulation used for rapid tuning of parameters and a full high-fidelity race car simulation, Autonoma's AutoVerse.

The simple simulation was designed for quickly fine tuning the parameters by selecting a position along the race line to spoof the car's location. It also allowed for simulated objects to be placed either randomly or in specified locations along the race line to test the planner's ability to produce a viable path. Another key feature of the simple simulation was printed outputs from the planner that can be seen in the top-left corner of the tmux terminal in Figure 5.12. In

this particular example, the output is displaying the cost calculation of different nodes while A* was running.

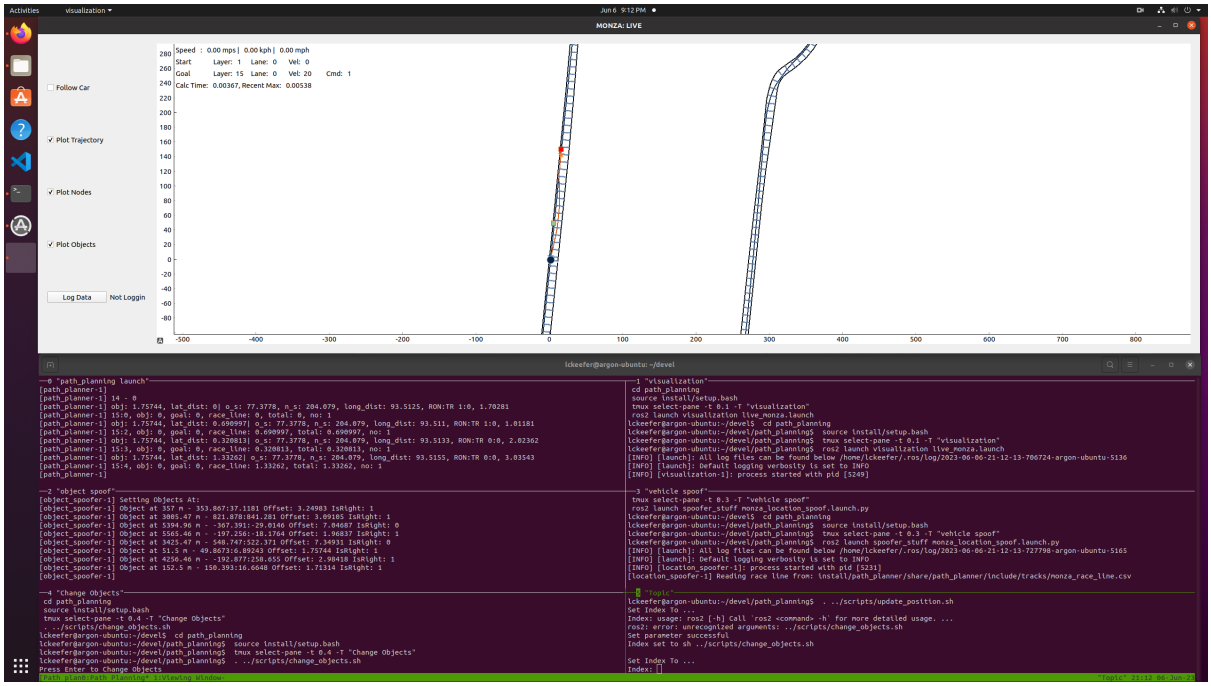
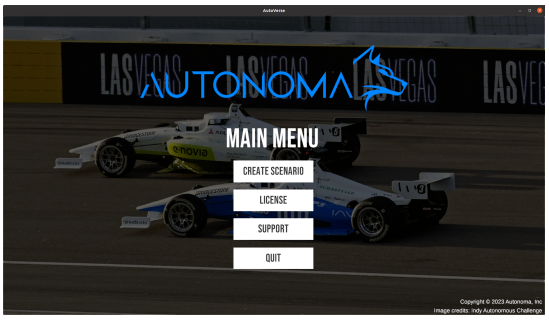


Figure 5.12: Simple simulation setup

Once the path planner passed this initial evaluation in the simple simulation, testing was moved to more realistic simulation testing. Previously, testing was done on LGSVL, but that simulator has since been discontinued and no longer has updates to keep up with the developments of the IAC. In its place is the Unity based simulator from Autonoma, called AutoVerse [97]. This simulation provides a high-fidelity real-world simulator complete with a realistic sensor suite, real world vehicle models, and software-in-the-loop capabilities that allow for easy plug-and-play for the path planning testing. This simulator provides an accurate replica of various race tracks for testing such as the Texas Motor Speedway and the Monza Circuit, seen in Figure 5.13b and 5.13c, respectively. The specific parameters that were used in this testing are list in Appendix C: Table C.3 and Table C.2.



(a) Autonoma AutoVerse Home Screen

(b) Texas Motor Speedway Simulation

(c) Monza Simulation

Figure 5.13: Simple Simulation Testing for TMS

The path planner made certain assumptions about scenario because of the scope of the testing. This includes using the truth location of the car in the planner as well as assuming perfect object detection, Therefore, if an object is detected it is assumed that it really ”exists” and must be avoided. Previously the ATR code stack was used for the lateral and longitudinal controls as well as some of the safety options. For path planner integration for this set of testing, an in-development full-code stack from Autonoma was used because it was designed to interface with the simulation. However, it is still in development, some of the testing was limited because of the need for better tuning of different parameters.

Many of the cost parameter configuration that were used in the previous iteration of the planner were used again, such as the soft, medium, and hard weights. Additional parameters were added such as right/left of object zone cost, it was safe to assume that any objects encountered would be in the inner lane, or the left side of the track, but with the expanding scope of the competition that assumption will not always hold true. Instead, objects could be found in any lane of the track, left, right or center. Therefore, in this testing is that any object that needs to be avoided would lie somewhere along the desired race lines, which are shown in Figure 5.14.

Here it is evident that the desired race line does not just stay in the center of the track. However, as the planner is written, two nodes of equal cost will default to the node explored first which will be the node farthest to the right. This was not a problem when all the passing was to be done on the right, but now the objects could also exist further to the right on the track. This led to an occasional undesirable path that hugged too close to the track bounds.

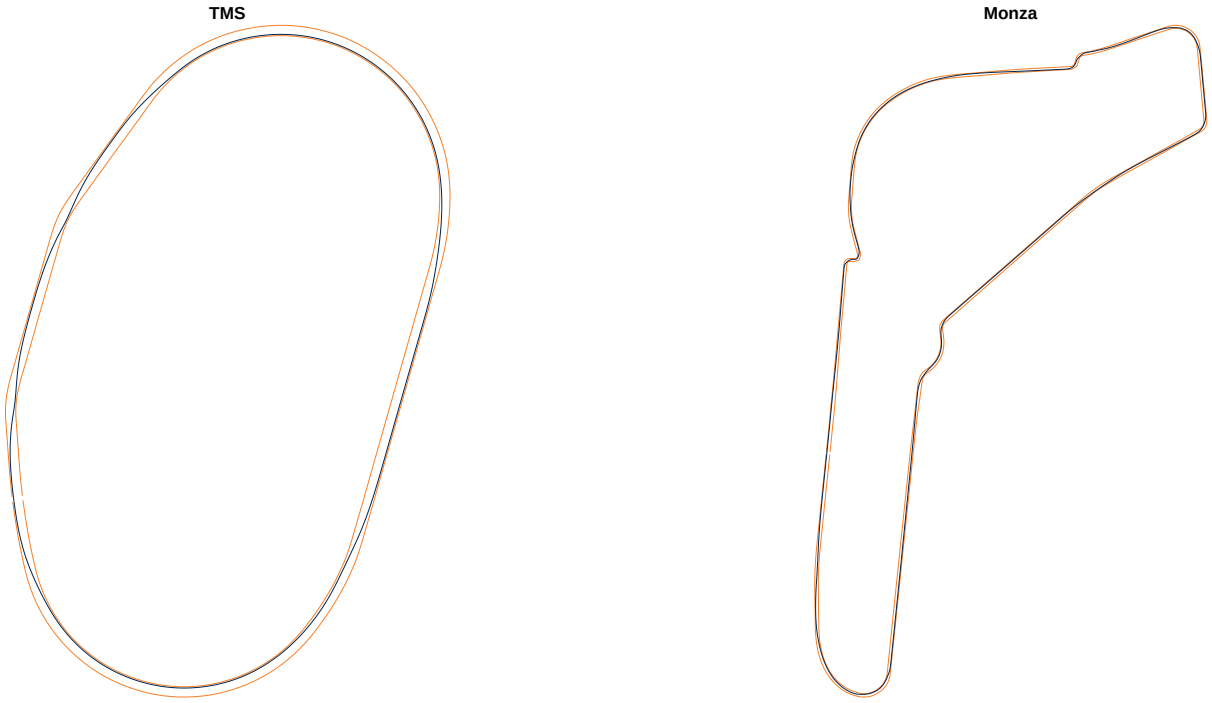


Figure 5.14: Simulation Race Tracks

An exaggerated version of how the object weight parameters are configured can be seen in Figure 5.15. In this example, the detected object is represented by the green square and lies in the right side of track as the vehicle travels from the top of the graph to the bottom. All the nodes that are covered by the purple section are considered nodes that are right of the object. The nodes covered by the red, yellow, and green sections have the hard, medium, and soft weights applied to them, respectively. The nodes that are not covered by any color do not have any object weight applied to them.

Since the number of nodes per layer are not uniform, figuring out which zone a node belongs to was tweaked for this iteration. Essentially, each objects path was projected forward according to its distance from the right bound. The distance between this projected path and the node being explored was calculated and the appropriate object weight applied. Another

option that was considered was projecting the object forward on the race line, but in that case the object weight would have similar affect as the race line cost.

Tuning these parameters can be tricky and require a deeper examination of the costs at each node while the A* algorithm was being run. For example, in Figure 5.15b, an exaggerated version of the final weight parameters are shown. In this case, the hard and medium weight zones are expanded and most of the nodes on the graph fall under one of the weight zones. The hard zone weight was set to 20, the medium to 16, the soft to 10, and the right of object to 20. Initially a designer may set high weights and wide widths for the hard zone would easily force the calculated path away from object as quickly as possible. However, looking at Figure 5.15b shows that the opposite is true. Instead, the path that was output of A* runs right through the object, the exact opposite of what was desirable.

This effect is caused by a number of reasons. First is the fact is that the relative weight of the hard zone and the medium zone are similar. When calculating the costs, a node in the hard weight zone would be more attractive because it has a lower race line distance cost. The width of the hard zone is also wide enough, such that the object cost stays the same, but the race line distance cost continues to increase the farther away from the the lane. A node that lies in the medium or soft weight zone might be too far away from the race line that the reduced costs do not matter in the final calculation.

A more reasonable tuning of the weights is shown in Figure 5.15c. In this one, the three weight zones cover a much more narrow portion of the track. The weights are also 12, 8, and 4 for the hard, medium, and soft zones, respectively. The combination of changes results in a much more reasonable gradient of costs away from the object. The resulting paths will be explored in next sections of this chapter.

(a) Bad tuning example (b) Weight Area - Bad Tuning (c) Weighted Area - Acceptable Tuning

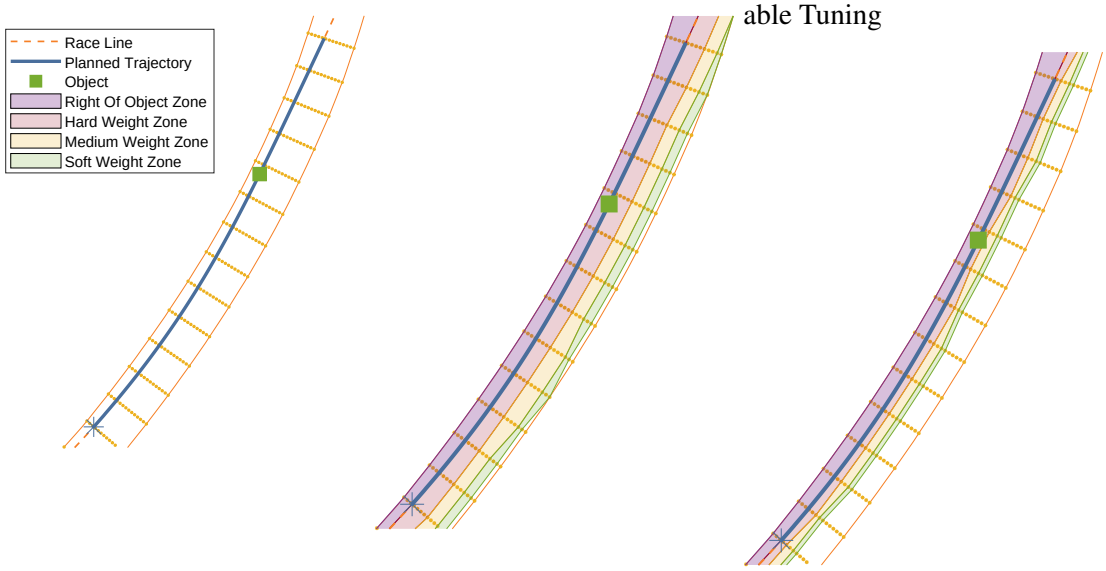


Figure 5.15: How tuning can affect the output path

5.2.1 Results

Both the TMS and Monza tracks were set up in the planner to be tested. For the TMS track, three tests were performed

1. Commanded speeds - checking that the addition of the velocity profiler works as expected, free of objects
2. Simple simulation - initial check that the planner outputs the desired path with an object in the desired path
3. Object avoidance - check that the planner performs as expected and as well as the previous iteration on a track similar to what was previously tested

Similarly, Tests 2 and 3 were performed on the Monza track to test the planner's capabilities on a road course.

TMS Simulation

The first test done on the TMS track was commanding various speeds around the track to see how well the velocity profile addition to the planner worked. In this scenario, the constraints

to the planner were set artificially low to best showcase the velocity profiles. This means the maximum lateral acceleration was set much lower than the actual vehicle's capibility. In theory, this would mean that the maximum velocity around curves would be lower than the maximum velocity set for the track.

The results of this test can be seen in Figure 5.16. The dotted purple line reflects the constraints put on this test such that during the turns, the maximum velocity allowed is lower than the 80 m/s maximum velocity set for the track as a whole. The blue represents the goal speed that was set manually during the tests in 5-10 m/s steps. The commanded speed, represented by the orange line, was the speed sent to the longitudinal controller. This speed was calculated from the velocity profile as a speed that corresponded with a certain distance into the calculated path.

In this scenario, the ramp-up type was set to "linear_1", or linear increase in speed at 1 m/s per layer. This should result in the commanded speeds not being instantaneous, but rather having a ramp up to the next goal speed. This is best exemplified in the first two goal speed change around 15 seconds and 30 seconds.

At higher speeds, the constraints of the velocity profile begin to become apparent. The general velocity profile trend follows the general trend of the maximum speed allowed at that point on the track, but is offset and did not reach or sustain the maximum speeds. This could be due to limitations in the longitudinal controller that did not allow for the type of responses modeled in the velocity profiler, which caused the vehicle to not reach the speeds it was expected, which in turn affected the overall velocity profile results. Despite this, the planner does show that the velocity profiler is able to vary the commanded speed of the vehicle in parts of the track that require a lower speed than the maximum velocity.

Figure 5.16: Commanded Speeds at TMS

Figure 5.17 shows a different view of the commanded speeds. In this graph, the goal, commanded, and actual speeds are shown plotted along a turn in the graph. The goal speed is set higher than the actual speed at that point, but the commanded speed was trending faster with the actual speed matching behind it.

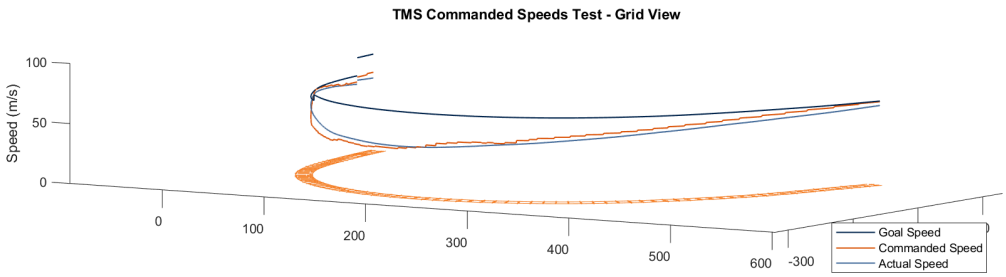


Figure 5.17: Commanded Speed at TMS - Grid Points View

The next test was done in the simple simulation and was used to test the viability of the computed path when encountering an object at different points on the graph. The results of one set of objects is shown in Figure 5.18. In some of the cases, the results are not an accurate reflection of how the planner might react in a live simulation because the car was placed in the race line well within the detection range of 100 meters. This is seen in Figure 5.18c, where the car was placed too close to the object. The planner outputs a path that avoid the object but does go much closer to the object then the chosen parameters would otherwise suggest. However, it

is able to route around the objects, even under non-normal operating conditions which confirms that the planner is able to output paths that can reliably guide the car around a detected object.

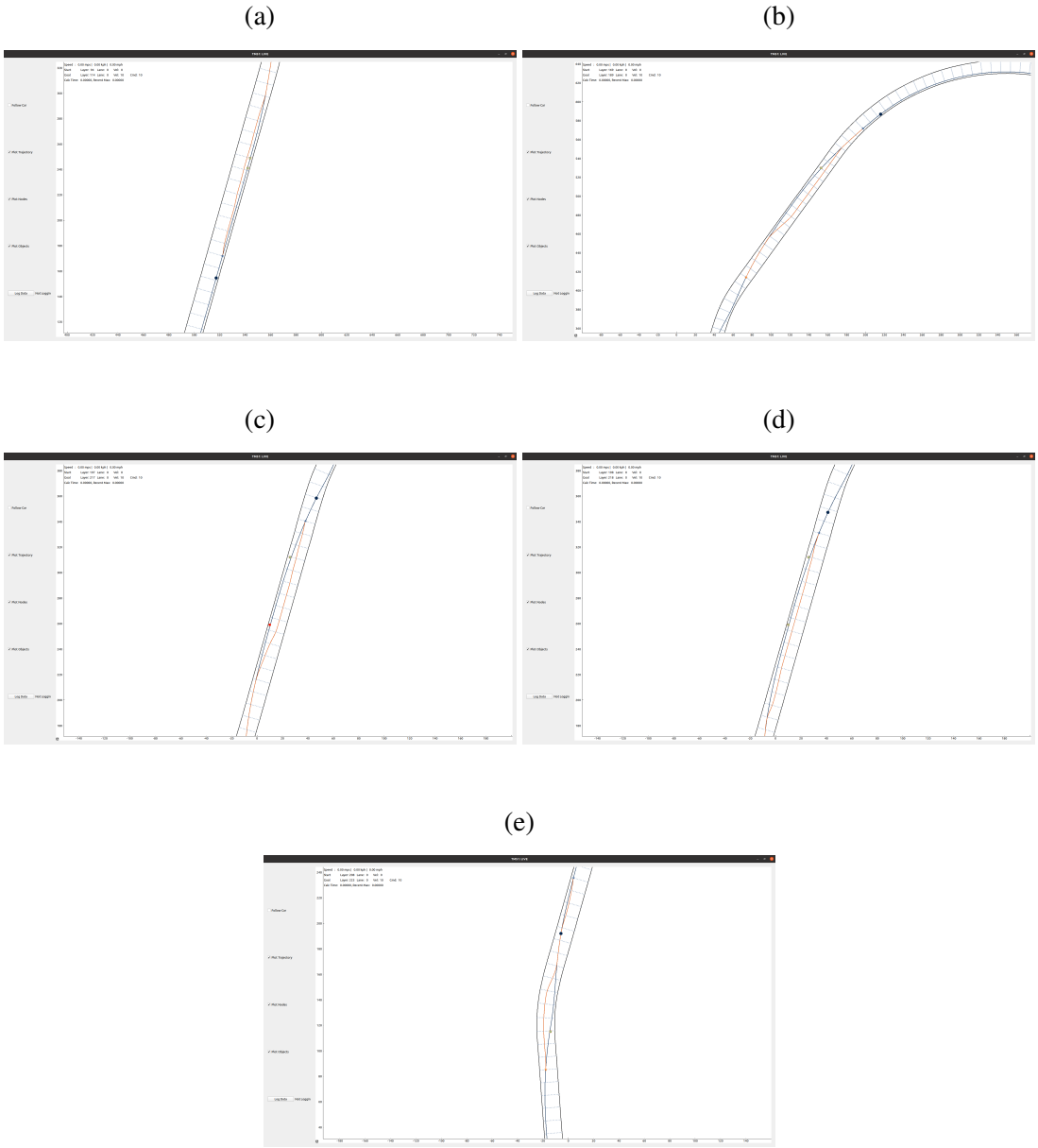


Figure 5.18: Simple Simulation Testing for TMS

For the next test, similar random object placement was used. Since objects can not be simulated in the simulation at the moment, the objects were spoofed similar to the previous tests. The vehicle location was sent to the object spoofer, and if the location was within the viewing distance, the object location was sent to the planner. Figure 5.19 shows an example of the vehicle passing an object, traveling from the bottom to the top of path. In this snapshot,

the vehicle had previously detected the object and the A* path was essentially frozen until the vehicle passes the object and returns to the race line. The planner is still running in the background, but the starting and goal nodes are kept the same which means the path outputted by the planner stays the same unless a new object is detected. The path freeze is necessary so that the path does not continually get replanned as the vehicle approaches the object and the starting node is continually shifted forward. This allows the lateral controller enough time to guide the car away from the object(s).

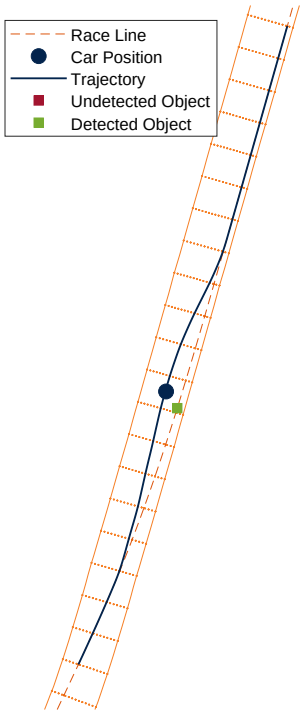


Figure 5.19: Passing an Object in Simulation

In Figure 5.20, the vehicle is seen approaching a series of three objects, having already detected the first two. If the vehicle would continue on this path, it would come quite close to the third object. However, the vehicle does eventually detect the third object and adjusts its path accordingly, seen in Figure 5.21. In this scenario, the planner continues to freeze the starting and ending node, but is able to extend the avoidance path to safely go around all three objects. By keeping the starting node the same, the behavior of the path stays mostly the same towards the beginning and does not cause any large jumps in where the vehicle is supposed to be versus where is actually is, while also allowing the rerouting to seamlessly blend into the previously planned trajectory.

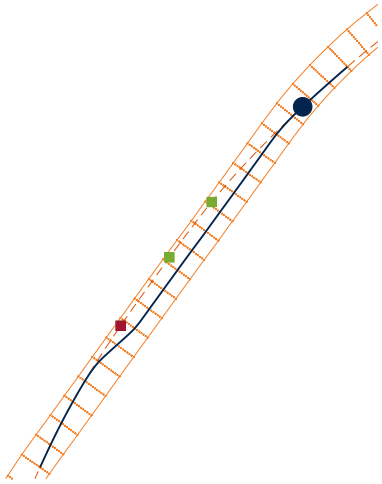


Figure 5.20: Passing objects in simulation with 2 detected objects and 1 undetected object

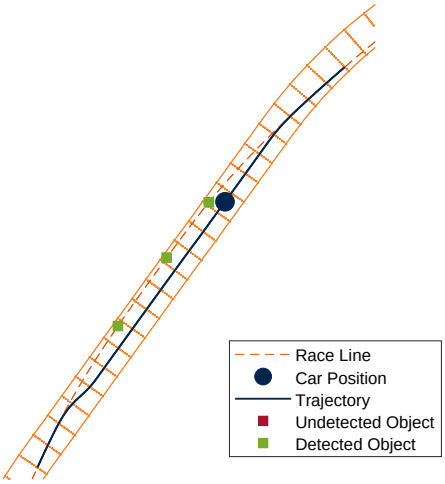


Figure 5.21: Passing objects in simulation with all 3 detected objects

Monza Simulation

Similarly, testing for the Monza was done first in the simple simulation. Figure 5.22 shows a set of results. Results from the AutoVerse simulation are also shown in Figure 5.23. The central subplot shows the location of the randomly generated objects along the track. Certain areas of the track did not have any objects such as the chicanes and tighter turns because in a realistic racing scenario, the vehicle would not be passing in those zones.

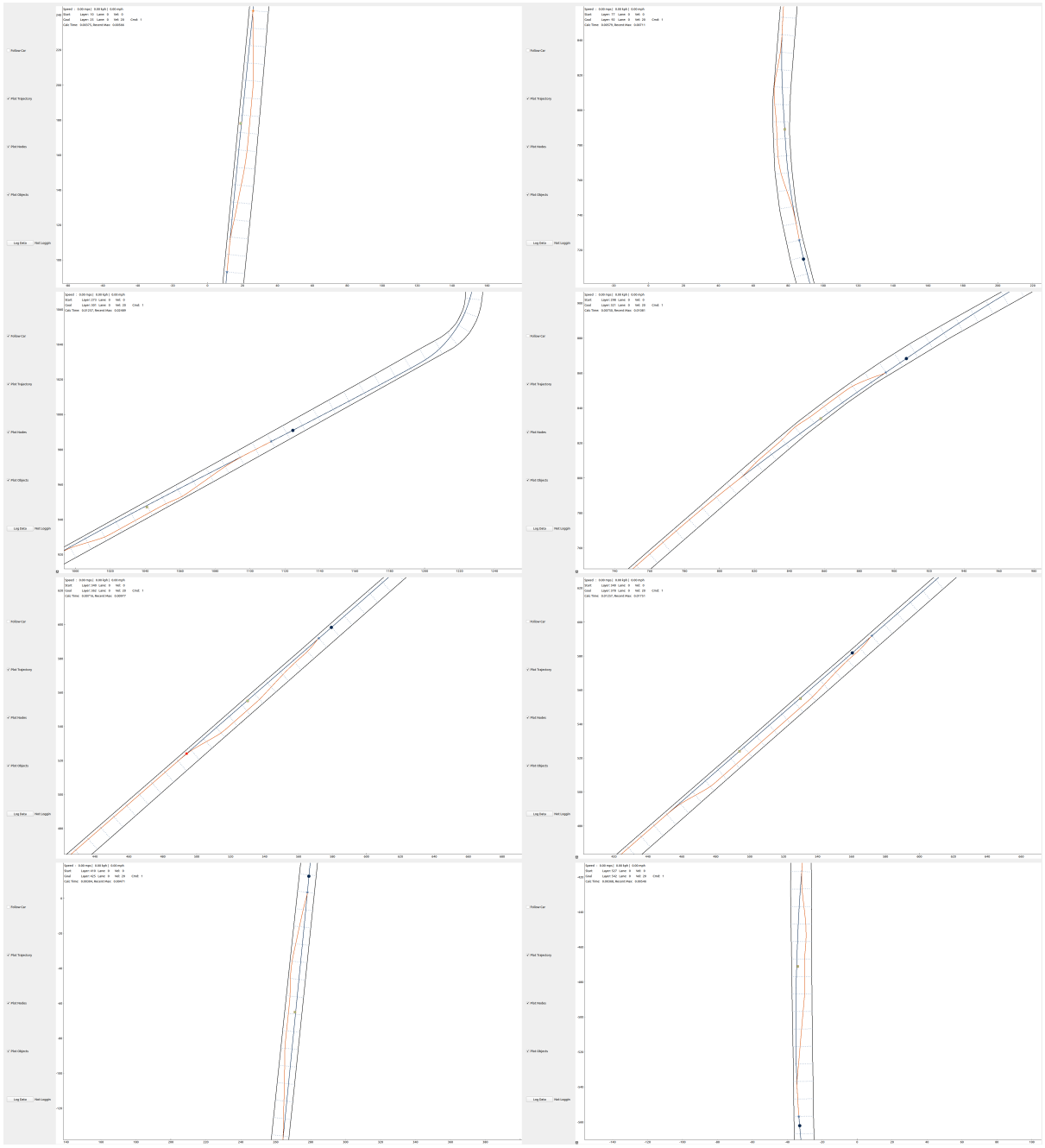


Figure 5.22: Simple Simulation Testing for Monza

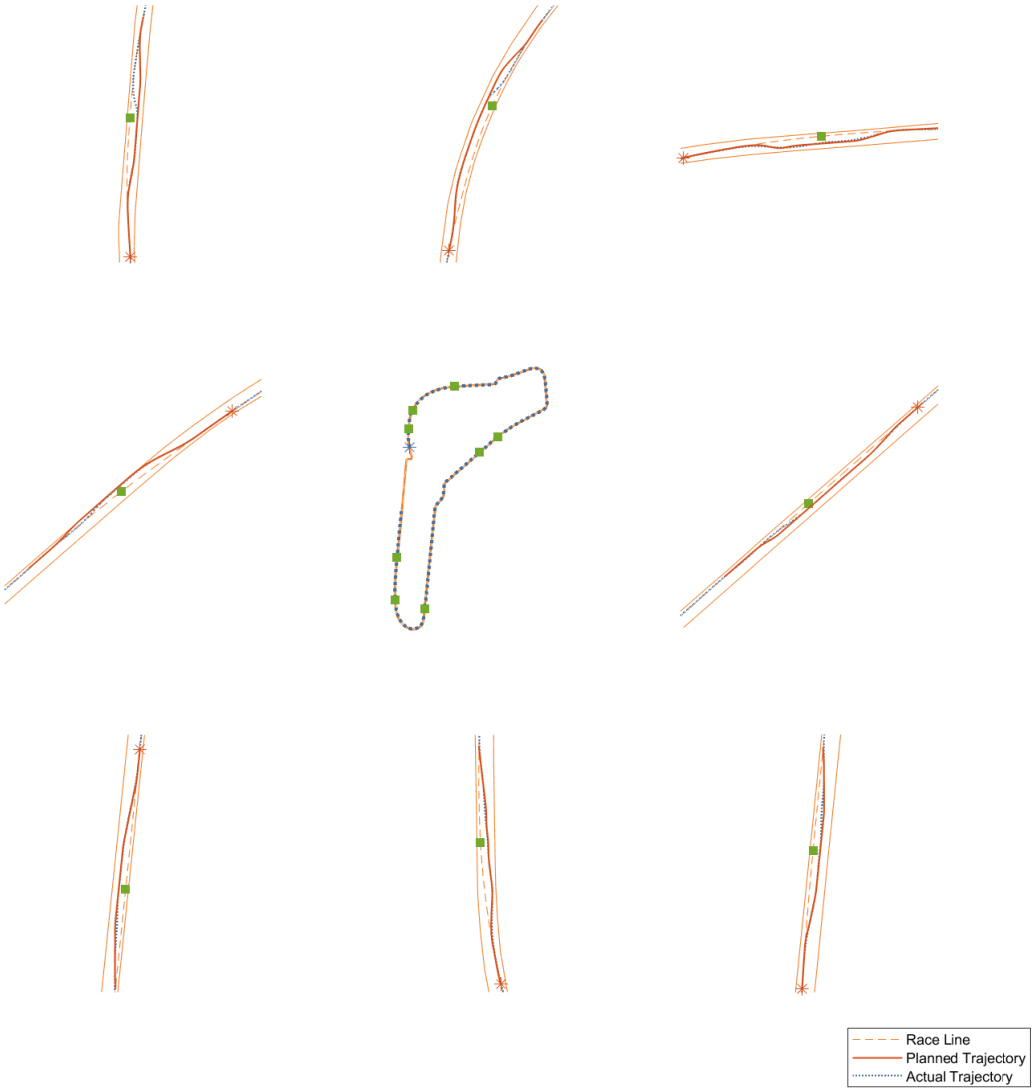


Figure 5.23: Full lap of random objects at Monza

Chapter 6

Conclusions

Overall, the planner acted as expected, consistently planning viable paths around detected objects. The main goal of this path planner iteration was to build upon the initial planner for the IAC by providing a more robust spline generation technique that would account for more complex track configurations as well as add a velocity profile that could aid in assuring the produced paths fell within the vehicle constraints. Additionally, the planner needed to increase in complexity while keeping a low computation time to make it viable in a racing situation. Unfortunately, due to a change in testing hardware, the time complexity of the planner was not as rigorously tested for comparison. However, even with older hardware the planner performed with maximum calculation time, of about 20 ms per iteration. This computation time's is on par with the maximum values seen in Chapter 3.

Many of the path planning algorithms introduced in Chapter 2 were designed with robustness in mind. Specifically they were designed to handle a wide range of situations with a wide range of unknowns. In return, many of these planners either take extensive tuning with a multitude of parameters or take too long to find a solution to be viable for high-speed racing. This thesis presents and alternative to these types of planners. In exchange for giving up robustness to handle a range of unknowns, this planner utilizes a simple, efficient, and reliable planning algorithm to navigate a well-defined environment.

6.1 Summary

This thesis presented an introduction to autonomous vehicles and racing in Chapter 1. Also presented in the first chapter are the contributions to the field of autonomous racing. Chapter 2 presents some background information relevant to different aspects of the path planner including path planning algorithms, path generation techniques, and how the problem of path planning has been addressed already. Chapters 3 and 4 present two path planners. The first planner was an earlier in-development planner that was designed for a competition set on an oval course. The second planner built upon the first planner but was expanded and augmented so that it could be used on road course and could handle more complex maneuvers. Finally, Chapter 6 presents two experimental setups and results that tested the usability of the second planner.

6.2 Future Work

While the planner developed in this thesis is certainly not the final solution to place into just any autonomous racing software stack, it can serve as the basis for more complex needs and scenarios. There were many ideas and unexplored planner additions and areas of improvement that were discovered during the duration of the research and implementation for this planner. The following are some of these ideas that could be avenues for future investigation.

Post Planner Path Smoothing In the first planner iteration, path smoothing was done with a moving mean in order to soften the corners of the path. In the second iteration, the moving mean was also implemented as a path smoothing technique. However, with the spline edges, this could result in wavy paths such as the ones shown in Figure 6.1. For the most part, the paths appeared smoother and any waviness present did not effect the overall driveability of the path at the speeds tested, but it could effect the controlability at higher speeds. Also, the spine generation techniques only guarantees a C^1 continuity, but C^2 would be more preferred at higher speeds. Future iterations of the planner could explore more advanced path smoothing techniques.

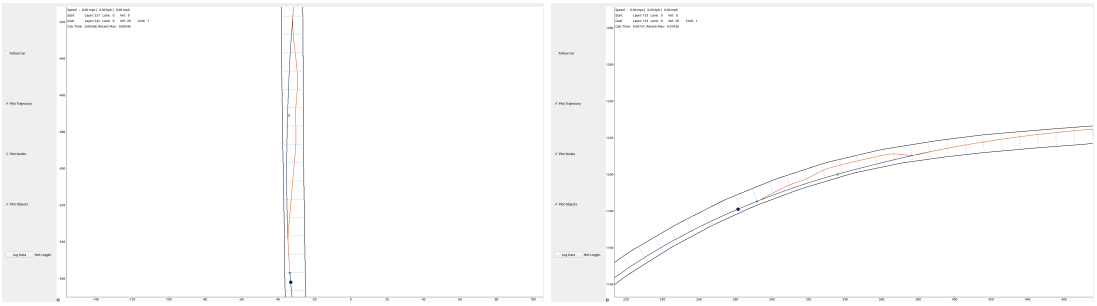


Figure 6.1: Wavy Path Smoothing

Integration with Object Prediction This planner assumes that any object it detects is a static object and propagates its cost zones forward along the race line. The assumption made is that the detected object has a similar calculation for what race line it *should* be on without taking in to account it prior behaviors. This might not always be true in more advanced autonomous racing scenarios and the opponent vehicle could deviate from the expected race line in order to perform a defensive maneuver. While the planner as is could respond with a shift in object position by adjusting the node the object is associated with, ultimately a safer option would be to add an object prediction method to more accurately produce paths that avoid collisions. An even more advanced object prediction method could take into account the objects previous moves in order to predict how they will defend certain attacks. These options combined with multiple dynamic objects are all areas of further research to integrate with a planner.

Cost Configuration With more complex edge generation, such as the one implemented in this thesis, more characteristics can be extracted from the spline such as curvature, range of curvature, or start/end node offset. Using these and other spline characteristics, a more sophisticated cost calculation could be implemented. This could allow for more complex maneuvers when racing against more than one opponent at a time.

Vehicle Model With the second iteration of the planner introduced in Chpater 4, some vehicle dynamic parameters were incorporated in the spline and velocity generation. However, further integration of the model could be used in order to continually produce dynamically feasible

paths and profiles while also reducing the safety margins in order to better emulate racing maneuvers seen with human drivers.

Code Optimization As more complexity is added to the planner, a stronger focus in how the code is written will need to be taken. This includes optimizing various pieces of the run-time code. Including:

1. More intelligent way of identifying the nearest nodes for both the car and the detected objects

2. Optimizing sections of code in a loop and utilizing functions in C++'s Standard Template Library (STL)

3. Cleaner data structures that are not as computationally burdensome with potentially needless complexity

4. Reduce the number of steps required for the velocity generation such as removing the need to prune the velocities by just having an improved velocity profile selection.

More optimal code could also allow for a denser node field. This would in turn allow for more control over trickier maneuvers as the scope of the competition rules continue to widen, allowing for much more complex and dynamic scenarios.

Testing For overall testing improvements, work to get a finer tuned set of controllers in order to increase the speed and handling of the vehicle during the testing. Additionally, an improved simulation environment that allows multiple simulated vehicles in order to more accurately reflect the racing scenario. This would require an increase in computing power in order to run two or more instances of the simulated vehicle as well as each vehicles' code stacks. Real-world validation on a road course could also validate the potential issues that could arise from the planner that do not come through in the simulation. For this type of testing, a road set-up on the car is necessary as well as a road course that is accurately mapped and validated.

Appendix A

Path Planner Implementation

A.1 Path Planner for Oval Courses

The following is a more detailed description of the how the planner for oval courses is implemented in code. The offline portions can be done either offline or upon planner initialization. The online portion is done repeatedly until the process is terminated. It can be tied to either a timer-in-a-loop or a callback from the localization. Regardless, the most up-to-date position estimation and speed and object detections are used.

Offline

1. Get inner track points - Using known points of the inner bounds, the inside bound is resampled to be an X meters distance apart from each other, where X is the layer distance parameter.

2. Get outer track points - An associated outer bound point is calculated so that a perpendicular line from the inner bound point can be drawn to the outer bound point.

3. Get lane points - The lane points are calculated by interpolating along the line between the inner and out bound point for each layer. The number of lanes is determined by a parameter.

Online

1. Get start and goal node - The closest vertex to the vehicle's current location is used to determine the starting node. The goal node is found by using the vertex on the race line

that is some distance, X, forward on the race line, where X is some parameter. The race line is determined by a parameter that indicates which layer should be the goal lane.

2. Apply A* - Using the start node, goal node, and detected objects location, Algorithm 3 is applied. For line 16, the description for how each cost is calculated can be found below.
   (a) Object cost - The distance laterally and longitudinally of the node to the object is determined and the appropriate weight is added.
   (b) Race line cost - The absolute distance from the node to the closest point on the race line is calculated and used for the race line cost.
   (c) Dynamic cost - The cost is applied if the edge causes a change in lanes from one layer to the next.
   (d) Heuristic - The absolute distance between the node being explored and the goal node is used for this value.
3. Smooth path - The path is smoothed using a moving mean.
4. Publish path and relevant information.

A.2 Path Planner for Road Courses

The following is a more detailed description of the how the planner for road courses is implemented in code. The offline portions can be done either offline or upon planner initialization. The online portion is done repeatedly until the process is terminated. It can be tied to either a timer-in-a-loop or a callback from the localization. Regardless, the most up-to-date position estimation and speed and object detections are used.

Offline

1. Get track bounds - Input known bounds of the desired track into *Global Race Trajectory Optimization* to get the output CSV file. [91][92]
2. Re-sample the track data - The graph layers will be selected from these new points.

3. Generate layers - Using the curvature of the track to determine if the layer is on a curve or a straight portion, the layer information is extracted from the re-sampled data in Step 2 at the appropriate distances apart [93].

4. Generate lanes - Using the width of the track and the lane distance, the lane points at each layer can be generated along the line between each layers left and right bounds.

5. Generate splines - For each lane point, or vertex, in the graph, a spline between it and every vertex in the following layer is generated using Equations (4.6-4.9) [93]. Only splines whose minimum radius of curvature is greater than the vehicle turn radius are kept.

6. Spline pruning - Since not all splines are kept, there may be orphaned edges or edges that lead to nothing or have no start. These edges are pruned from the graph which guarantees that a path between any two vertices on the graph can be found.

7. Velocity profile generation - The velocity profile for each remaining left is generated. This process is outlined in more detail in Section 4.3.1.

8. Velocity profile pruning - Similar to the spline pruning, the velocities are also pruned to ensure that only velocities that can connect to neighboring edges remain.

Online

1. Get start and goal node - The closest vertex to the vehicle's current location is used to determine the starting node. The goal node is found by using the vertex on the race line that is some distance, X, forward on the race line, where X is some parameter.

2. Apply A* - Using the start node, goal node, and detected objects location, Algorithm 3 is applied. For line 16, the description for how each cost is calculated can be found below.

   (a) Object cost - the distance laterally and longitudinally of the node to the object is determined and the appropriate weight is added.

(b) Race line cost - the absolute distance from the node to the closest point on the race line is calculated and used for the race line cost.

(c) Heuristic - The curvilinear distance associated with the layers race line is used as the heuristic cost.

3. Generate the velocity profile - Using the selected edge, the velocity profile is extracted from the graph.

4. Smooth path - The path is smoothed using a moving mean.

5. Determine the goal velocity - The controller used in this implementation can only take in a single value for goal velocity. Using a parameter, the velocity associated with a point X meters along the path is found.

6. Publish path and relevant information.

Appendix B

Parameters used for testing oval course planner

The following parameters were used for the experiments that tested the oval course planner.

Table B.1: MKZ Testing

Parameter	**Value**
Layer Distance (m)	10
Number of Lanes	5
Distance to Goal (m)	100
Soft Object Weight	6
Medium Object Weight	8
Hard Object Weight	12
Soft Zone Distance (m)	1
Medium Zone Distance (m)	1
Hard Zone Distance (m)	1
Front Object Distance (m)	50
Back Object Distance (m)	65
Dynamic Cost	2.25
Distance Cost	.75
Passing Lane	3
Defending Lane	0

The planner represents the following:

- **Layer Distance** - The distance between layers.
- **Number of Lanes** - The number of lanes in each layer, same around the entire track
- **Distance to Goal (m)** - The distance from the starting node to set the goal node.
- **Hard Object Weight** - The weight associated with the closest zone around an object (see Figure‘3.3).
- **Medium Object Weight** - The weight associated with the middle zone around an object (see Figure 3.3)
- **Soft Object Weight** - The weight associated with the farthest zone around an object (see Figure 3.3).
- **Hard Zone Distance (m)** - The distance past the object where the Hard Object Weight is in effect.
- **Medium Zone Distance (m)** - The distance past the Hard Zone Distance where the Medium Object Weight is in effect.
- **Soft Zone Distance (m)** - The distance past the Medium Zone Distance where the Soft Object weight is in effect.
- **Front Object Distance (m)** - The distance past an object to apply the three-zone weighting.
- **Back Object Distance (m)** - The distance behind an object to apply the three-zone weighting.
- **Dynamic Cost** - The cost associated with the lateral movement between lanes.
- **Distance Cost** - The valued multiplied to distance from the race line.
- **Passing Lane** - The goal lane when the vehicle encounters an object.
- **Defending Lane** - The goal lane of the vehicle.

Appendix C

Parameters used for testing road course planner

The following parameters were used for the experiments that tested the road course planner.

Table C.1: MKZ Testing

Parameter	**Value**
Layer Distance - Straight (m)	5
Mu Spacing	0.2
Velocity Step Size (m/s)	1
Vehicle Turn Radius (m)	5
Maximum Car Velocity (m/s)	10
Maximum Allowable Acceleration (m/s^2)	19.62
Soft Object Weight	2
Medium Object Weight	4
Soft Zone Distance	2
Medium Zone Distance	2
Hard Zone Distance	2

Table C.2: Monza Circuit Testing

Parameter	**Value**
Layer Distance - Straight (m)	10
Layer Distance - Curve (m)	5
Curve Omega Max (deg/m)	0.3
Mu Spacing	0.05
Velocity Step Size (m/s)	1
Vehicle Turn Radius (m)	5
Max Car Velocity (m/s)	90
Max Allowable Acceleration (m/s^2)	10
Send Velocity Distance (m)	20
Distance to Goal (m)	200
Soft Object Weight	4
Medium Object Weight	8
Hard Object Weight	12
Soft Zone Distance (m)	1
Medium Zone Distance (m)	2
Hard Zone Distance (m)	2
Front Object Distance (m)	30
Back Object Distance (m)	30

Table C.3: Texas Motor Speedway Testing

Parameter	**Value**
Layer Distance - Straight (m)	10
Layer Distance - Curve (m)	10
Curve Omega Max (rad/m)	0.3
Mu Spacing	0.05
Velocity Step Size (m/s)	1
Vehicle Turn Radius (m)	5
Max Car Velocity (m/s)	80
Max Allowable Acceleration (m/s^2)	19.62
Send Velocity Distance (m)	25
Soft Object Weight	4
Medium Object Weight	8
Hard Object Weight	12
Soft Zone Distance (m)	1
Medium Zone Distance (m)	2
Hard Zone Distance (m)	2
Front Object Distance (m)	50
Back Object Distance (m)	50

The parameters represent the following:

- **Layer Distance - Straight (m)** - The distance between layers for the straight section.
- **Layer Distance - Curve (m)** - The distance between layers for the curved sections.
- **Curve Omega Max (deg/m)** - The radius of curvature where, if less than, the part of the track is considered a curve and the layer distance is set appropriately.
- **Mu Spacing** - Where $1/\mu$ represent the number of points in each generated spline .
- **Velocity Step Size (m/s)** - For the velocity profile generation, the step size between the calculated velocities.
- **Vehicle Turn Radius (m)** - The turn radius of the vehicle.
- **Max Car Velocity (m/s)** - The maximum velocity of the vehicle. This parameter limits the maximum possible velocities generated by the velocity profile generator.
- **Max Allowable Acceleration** (m/s^2) - maximum allowable acceleration of the vehicle.
- **Send Velocity Distance** - The velocity associated with the distance along the path of this parameter that is sent to the controller as the goal velocity.
- **Distance to Goal (m)** - The distance from the starting node to set the goal node.
- **Hard Object Weight** - The weight associated with the closest zone around an object (see Figure‘3.3).
- **Medium Object Weight** - The weight associated with the middle zone around an object (see Figure 3.3)
- **Soft Object Weight** - The weight associated with the farthest zone around an object (see Figure 3.3).
- **Hard Zone Distance (m)** - The distance past the object where the Hard Object Weight is in effect.
- **Medium Zone Distance (m)** - The distance past the Hard Zone Distance where the Medium Object Weight is in effect.
- **Soft Zone Distance (m)** - The distance past the Medium Zone Distance where the Soft Object weight is in effect.
- **Front Object Distance (m)** - The distance past an object to apply the three-zone weighting.
- **Back Object Distance (m)** - The distance behind an object to apply the three-zone weighting.

Bibliography

[1] Lan Yang et al. “A Systematic Review of Autonomous Emergency Braking System: Impact Factor, Technology, and Performance Evaluation”. en. In: *Journal of Advanced Transportation* 2022 (Apr. 2022). Ed. by Francesco Galante, pp. 1–13. ISSN: 2042-3195, 0197-6729. DOI: 10.1155/2022/1188089. URL: https://www.hindawi.com/journals/jat/2022/1188089/ (visited on 03/19/2023).

[2] *Blind Spot Information System (BLIS) Overview*. URL: https://volvo.custhelp.com/app/answers/detail/a_id/9874/~/blind-spot-information-system-%28blis%29-overview (visited on 03/19/2023).

[3] Lingyun Xiao and Feng Gao. “A comprehensive review of the development of adaptive cruise control systems”. en. In: *Vehicle System Dynamics* 48.10 (Oct. 2010), pp. 1167–1192. ISSN: 0042-3114, 1744-5159. DOI: 10.1080/00423110903365910. URL: http://www.tandfonline.com/doi/abs/10.1080/00423110903365910 (visited on 03/19/2023).

[4] *Lane Departure vs. Lane Keeping vs. Lane Centering Tech*. en. URL: https://www.jdpower.com/cars/shopping-guides/lane-departure-vs-lane-keeping-vs-lane-centering-tech (visited on 03/19/2023).

[5] *SAE Levels of Driving Automation™ Refined for Clarity and International Audience*. en. URL: https://www.sae.org/site/blog/sae-j3016-update (visited on 03/19/2023).

[6] *Honda Legend Sedan with Level 3 Autonomy Available for Lease in Japan*. en-us. Section: News. Mar. 2021. URL: https://www.caranddriver.com/news/a35729591/honda-legend-level-3-autonomy-leases-japan/ (visited on 03/19/2023).

[7] *What Full Autonomy Means for the Waymo Driver - IEEE Spectrum*. en. URL: https://spectrum.ieee.org/full-autonomy-waymo-driver (visited on 03/19/2023).

[8] *Autonomous Vehicle Technology — Driverless Cars — Cruise*. en. URL: https://www.getcruise.com/technology/ (visited on 03/19/2023).

[9] *From Darpa Grand Challenge 2004DARPA's Debacle in the Desert*. en-US. June 2004. URL: https://www.popsci.com/scitech/article/2004-06/darpa-grand-challenge-2004darpas-debacle-desert/ (visited on 03/18/2023).

[10] Sebastian Thrun et al. "Stanley: The robot that won the DARPA Grand Challenge". en. In: *Journal of Field Robotics* 23.9 (Sept. 2006), pp. 661–692. ISSN: 15564959, 15564967. DOI: 10.1002/rob.20147. URL: https://onlinelibrary.wiley.com/doi/10.1002/rob.20147 (visited on 03/18/2023).

[11] *Roborace*. en. URL: https://roborace.com (visited on 12/26/2022).

[12] *Indy Autonomous Challenge - Official Website*. en-US. URL: https://www.indyautonomouscha
com (visited on 12/26/2022).

[13] Craig Scarborough. *Autonomous racing is here! How Roborace is helping develop technology for the future*. en-US. Nov. 2021. URL: https://motorsport.tech/roborace/how-roborace-is-helping-develop-technology-for-the-future (visited on 03/21/2023).

[14] John H. Reif. "Complexity of the mover's problem and generalizations". en. In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. San Juan, Puerto Rico: IEEE, Oct. 1979, pp. 421–427. DOI: 10.1109/SFCS.1979.10. URL: http://ieeexplore.ieee.org/document/4568037/ (visited on 01/29/2023).

[15] Steven LaVelle. *Planning Algorithms*. Cambridge University Press, 2014.

[16] S.K. Gupta et al. "Automated process planning for sheet metal bending operations". en. In: *Journal of Manufacturing Systems* 17.5 (Jan. 1998), pp. 338–360. ISSN: 02786125. DOI: 10.1016/S0278-6125(98)80002-2. URL: https://linkinghub.elsevier.com/retrieve/pii/S0278612598800022 (visited on 01/29/2023).

[17] Alexander Shkolnik and Russ Tedrake. "Sample-Based Planning with Volumes in Configuration Space". en. In: ().

[18] Stefania Pellegrinelli et al. "Multi-robot spot-welding cells for car-body assembly: Design and motion planning". en. In: *Robotics and Computer-Integrated Manufacturing* 44 (Apr. 2017), pp. 97–116. ISSN: 07365845. DOI: 10.1016/j.rcim.2016.08.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S0736584515300302 (visited on 01/29/2023).

[19] K. Sugihara and J. Smith. "Genetic algorithms for adaptive motion planning of an autonomous mobile robot". en. In: *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*. Monterey, CA, USA: IEEE Comput. Soc. Press, 1997, pp. 138–143. ISBN: 978-0-8186-8138-7. DOI: 10.1109/CIRA.1997.613850. URL: http://ieeexplore.ieee.org/document/613850/ (visited on 01/29/2023).

[20] Laurene Claussmann et al. "A Review of Motion Planning for Highway Autonomous Driving". en. In: *IEEE Transactions on Intelligent Transportation Systems* 21.5 (May 2020), pp. 1826–1848. ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2019.2913998. URL: https://ieeexplore.ieee.org/document/8715479/ (visited on 01/29/2023).

[21] Lun Quan et al. "Survey of UAV motion planning". en. In: *IET Cyber-Systems and Robotics* 2.1 (Mar. 2020), pp. 14–21. ISSN: 2631-6315, 2631-6315. DOI: 10.1049/iet-csr.2020.0004. URL: https://onlinelibrary.wiley.com/doi/10.1049/iet-csr.2020.0004 (visited on 01/29/2023).

[22] Torin Adamson et al. “Optimizing Low Energy Pathways in Receptor-Ligand Binding with Motion Planning”. en. In: *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. San Diego, CA, USA: IEEE, Nov. 2019, pp. 2041–2048. ISBN: 978-1-72811-867-3. DOI: 10.1109/BIBM47256.2019.8983169. URL: https://ieeexplore.ieee.org/document/8983169/ (visited on 01/29/2023).

[23] David Gonzalez et al. “A Review of Motion Planning Techniques for Automated Vehicles”. en. In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (Apr. 2016), pp. 1135–1145. ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2015.2498841. URL: http://ieeexplore.ieee.org/document/7339478/ (visited on 01/29/2023).

[24] Chengmin Zhou, Bingding Huang, and Pasi Fränti. “A review of motion planning algorithms for intelligent robotics”. en. In: ().

[25] Lynch Choset. *Principles of Robot Motion*. The MIT Press, 2005.

[26] Jørgen Bang-Jenson and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. 1st. Springer-Verlag, 2000.

[27] Scott Beamer, Krste Asanovic, and David Patterson. “Direction-optimizing Breadth-First Search”. en. In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–10. ISBN: 978-1-4673-0805-2. DOI: 10.1109/SC.2012.50. URL: http://ieeexplore.ieee.org/document/6468458/ (visited on 01/12/2023).

[28] E W Dijkstra. “A Note on Two Problemsin Connexion with Graphs”. en. In: *Numerische Mathematlk* (1959).

[29] *greedy algorithm*. URL: https://xlinux.nist.gov/dads//HTML/greedyalgo.html (visited on 01/16/2023).

[30] Thomas Cormen et al. *Introduction to Algorithms*. Vol. Third Edition. MIT Press, 2007.

[31] Huijuan Wang, Yuan Yu, and Quanbo Yuan. “Application of Dijkstra algorithm in robot path-planning”. en. In: *2011 Second International Conference on Mechanic Automation and Control Engineering*. Inner Mongolia, China: IEEE, July 2011, pp. 1067–1069. ISBN: 978-1-4244-9436-1. DOI: 10.1109/MACE.2011.5987118. URL: http://ieeexplore.ieee.org/document/5987118/ (visited on 01/13/2023).

[32] Guan-zheng Tan, Huan He, and Sloman Aaron. “Global optimal path planning for mobile robot based on improved Dijkstra algorithm and ant system algorithm”. en. In: *Journal of Central South University of Technology* 13.1 (Feb. 2006), pp. 80–86. ISSN: 1005-9784, 1993-0666. DOI: 10.1007/s11771-006-0111-8. URL: http://link.springer.com/10.1007/s11771-006-0111-8 (visited on 01/13/2023).

[33] Yong Deng et al. “Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment”. en. In: *Applied Soft Computing* 12.3 (Mar. 2012), pp. 1231–1237. ISSN: 15684946. DOI: 10.1016/j.asoc.2011.11.011. URL: https://linkinghub.elsevier.com/retrieve/pii/S1568494611004376 (visited on 01/13/2023).

[34] M. Noto and H. Sato. “A method for the shortest path search by extended Dijkstra algorithm”. en. In: *SMC 2000 Conference Proceedings. 2000 IEEE International Conference on Systems, Man and Cybernetics. 'Cybernetics Evolving to Systems, Humans, Organizations, and their Complex Interactions' (Cat. No.00CH37166)*. Vol. 3. Nashville, TN, USA: IEEE, 2000, pp. 2316–2320. ISBN: 978-0-7803-6583-4. DOI: 10.1109/ICSMC.2000.886462. URL: http://ieeexplore.ieee.org/document/886462/ (visited on 01/13/2023).

[35] Peter Hart, Nils Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. en. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136. URL: http://ieeexplore.ieee.org/document/4082128/ (visited on 12/26/2022).

[36] Eric W. Weisstein. *von Neumann Neighborhood*. en. Text. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/ (visited on 01/21/2023).

[37] Ira Pohl. "The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 12–17.

[38] Abhinav Bhatia, Justin Svegliato, and Shlomo Zilberstein. "On the Benefits of Randomly Adjusting Anytime Weighted A*". en. In: *Proceedings of the International Symposium on Combinatorial Search* 12.1 (July 2021), pp. 116–120. ISSN: 2832-9163, 2832-9171. DOI: 10.1609/socs.v12i1.18558. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18558 (visited on 01/25/2023).

[39] Eric A. Hansen, Shlomo Zilberstein, and Victor A. Danilchenko. *Anytime Heuristic Search: First Results*. Tech. rep. 97-50. Computer Science Department, University of Massachussetts Amherst, 1997. URL: http://rbr.cs.umass.edu/shlomo/papers/HZDtr9750.html.

[40] Maxim Likhachev, Geo Gordon, and Sebastian Thrun. "ARA*: Formal Analysis". In: (2003).

[41] Sven Koenig, Maxim Likhachev, and David Furcy. "Lifelong Planning A". en. In: *Artificial Intelligence* 155.1-2 (May 2004), pp. 93–146. ISSN: 00043702. DOI: 10.1016/j.artint.2003.12.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S000437020300225X (visited on 01/28/2023).

[42] Szilárd Aradi. *Survey of Deep Reinforcement Learning for Motion Planning of Autonomous Vehicles*. en. arXiv:2001.11231 [cs, eess, stat]. Jan. 2020. URL: http://arxiv.org/abs/2001.11231 (visited on 12/26/2022).

[43] A. Stentz. "Optimal and efficient path planning for partially-known environments". en. In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. San Diego, CA, USA: IEEE Comput. Soc. Press, 1994, pp. 3310–3317. ISBN:

978-0-8186-5330-8. DOI: 10.1109/ROBOT.1994.351061. URL: http://ieeexplore.ieee.org/document/351061/ (visited on 01/29/2023).

[44] Anthony Stentz. "The Focussed D* Algorithm for Real-Time Replanning". en. In: ().

[45] Sven Koenig and Maxim Likhachev. "D* Lite". In: *Aaai/iaai* 15 (July 2002), pp. 476–483.

[46] *MotionPlanningHigherDimensions*. URL: http://motion.cs.illinois.edu/RoboticSystems/MotionPlanningHigherDimensions.html (visited on 02/25/2023).

[47] J. Barraquand and J.-C. Latombe. "A Monte-Carlo algorithm for path planning with many degrees of freedom". en. In: *Proceedings., IEEE International Conference on Robotics and Automation*. Cincinnati, OH, USA: IEEE Comput. Soc. Press, 1990, pp. 1712–1717. ISBN: 978-0-8186-9061-7. DOI: 10.1109/ROBOT.1990.126256. URL: http://ieeexplore.ieee.org/document/126256/ (visited on 02/25/2023).

[48] L.E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". en. In: *IEEE Transactions on Robotics and Automation* 12.4 (Aug. 1996), pp. 566–580. ISSN: 1042296X. DOI: 10.1109/70.508439. URL: http://ieeexplore.ieee.org/document/508439/ (visited on 02/25/2023).

[49] Roland Geraerts and M.H. Overmars. "A comparative study of probabilistic roadmap planners". In: *Algorithmic Foundations of Robotics V* (2004).

[50] Gildardo Sanchez-Ante and Jean-Claude Latombe. "A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking". In: *International Journal of Robotic Research - IJRR*. 2001, pp. 403–417.

[51] M.S. Branicky et al. "Quasi-randomized path planning". en. In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*. Vol. 2. Seoul, South Korea: IEEE, 2001, pp. 1481–1487. ISBN: 978-0-7803-6576-6. DOI: 10.1109/ROBOT.2001.932820. URL: http://ieeexplore.ieee.org/document/932820/ (visited on 02/27/2023).

[52] Nancy M Amato et al. “OBPRM: An Obstacle-Based PRM for 3D Workspaces”. en. In: ().

[53] T. Siméon, J.-P. Laumond, and C. Nissoux. “Visibility-based probabilistic roadmaps for motion planning”. en. In: *Advanced Robotics* 14.6 (Jan. 2000), pp. 477–493. ISSN: 0169-1864, 1568-5535. DOI: 10.1163/156855300741960. URL: https://www.tandfonline.com/doi/full/10.1163/156855300741960 (visited on 02/27/2023).

[54] Steven M LaValle et al. “Rapidly-exploring random trees: A new tool for path planning”. In: (1998). Publisher: Ames, IA, USA.

[55] Sertac Karaman and Emilio Frazzoli. *Incremental Sampling-based Algorithms for Optimal Motion Planning*. en. arXiv:1005.0416 [cs]. May 2010. URL: http://arxiv.org/abs/1005.0416 (visited on 12/26/2022).

[56] Iram Noreen, Amna Khan, and Zulfiqar Habib. “A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms”. en. In: (2016).

[57] Fahad Islam et al. “RRT*-Smart: Rapid convergence implementation of RRT* towards optimal solution”. en. In: *2012 IEEE International Conference on Mechatronics and Automation*. Chengdu, China: IEEE, Aug. 2012, pp. 1651–1656. ISBN: 978-1-4673-1278-3. DOI: 10.1109/ICMA.2012.6284384. URL: http://ieeexplore.ieee.org/document/6284384/ (visited on 03/03/2023).

[58] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. “Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic”. en. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. arXiv:1404.2334 [cs]. Sept. 2014, pp. 2997–3004. DOI: 10.1109/IROS.2014.6942976. URL: http://arxiv.org/abs/1404.2334 (visited on 03/04/2023).

[59] Abhijeet Ravankar et al. “Path Smoothing Techniques in Robot Navigation: State-of-the-Art, Current and Future Challenges”. en. In: *Sensors* 18.9 (Sept. 2018), p. 3170. ISSN:

1424-8220. DOI: 10.3390/s18093170. URL: http://www.mdpi.com/1424-8220/18/9/3170 (visited on 03/04/2023).

[60] L. E. Dubins. "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents". en. In: *American Journal of Mathematics* 79.3 (July 1957), p. 497. ISSN: 00029327. DOI: 10.2307/2372560. URL: https://www.jstor.org/stable/2372560?origin=crossref (visited on 03/04/2023).

[61] Yucong Lin and Srikanth Saripalli. "Path planning using 3D Dubins Curve for Unmanned Aerial Vehicles". en. In: *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*. Orlando, FL, USA: IEEE, May 2014, pp. 296–304. ISBN: 978-1-4799-2376-2. DOI: 10.1109/ICUAS.2014.6842268. URL: http://ieeexplore.ieee.org/document/6842268/ (visited on 03/04/2023).

[62] Wenyu Cai, Meiyan Zhang, and Yahong Zheng. "Task Assignment and Path Planning for Multiple Autonomous Underwater Vehicles Using 3D Dubins Curves †". en. In: *Sensors* 17.7 (July 2017), p. 1607. ISSN: 1424-8220. DOI: 10.3390/s17071607. URL: http://www.mdpi.com/1424-8220/17/7/1607 (visited on 03/04/2023).

[63] Doran K. Wilde. "Computing clothoid segments for trajectory generation". en. In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. St. Louis, MO, USA: IEEE, Oct. 2009, pp. 2440–2445. ISBN: 978-1-4244-3803-7. DOI: 10.1109/IROS.2009.5354700. URL: http://ieeexplore.ieee.org/document/5354700/ (visited on 03/05/2023).

[64] Junior A. R. Silva and Valdir Grassi. "Clothoid-Based Global Path Planning for Autonomous Vehicles in Urban Scenarios". en. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. Brisbane, QLD: IEEE, May 2018, pp. 4312–4318. ISBN: 978-1-5386-3081-5. DOI: 10.1109/ICRA.2018.8461201. URL: https://ieeexplore.ieee.org/document/8461201/ (visited on 03/05/2023).

[65] Misel Brezak and Ivan Petrovic. "Real-time Approximation of Clothoids With Bounded Error for Path Planning Applications". en. In: *IEEE Transactions on Robotics* 30.2 (Apr.

2014), pp. 507–515. ISSN: 1552-3098, 1941-0468. DOI: 10.1109/TRO.2013.2283928. URL: http://ieeexplore.ieee.org/document/6646277/ (visited on 03/05/2023).

[66] Ji-wung Choi, Renwick Curry, and Gabriel Elkaim. "Path Planning Based on Bezier Curve for Autonomous Ground Vehicles". en. In: *Advances in Electrical and Electronics Engineering - IAENG Special Edition of the World Congress on Engineering and Computer Science 2008*. San Francisco, California, USA: IEEE, Oct. 2008, pp. 158–166. ISBN: 978-0-7695-3555-5. DOI: 10.1109/WCECS.2008.27. URL: http://ieeexplore.ieee.org/document/5233184/ (visited on 03/05/2023).

[67] Baoye Song, Guohui Tian, and Fengyu Zhou. "A Comparison Study on Path Smoothing Algorithms for Laser Robot Navigated Mobile Robot Path Planning in Intelligent Space ". en. In: *Journal of Information* (2010).

[68] Long Han et al. "Bezier curve based path planning for autonomous vehicle in urban environment". en. In: *2010 IEEE Intelligent Vehicles Symposium*. La Jolla, CA, USA: IEEE, June 2010, pp. 1036–1042. ISBN: 978-1-4244-7866-8. DOI: 10.1109/IVS.2010.5548085. URL: http://ieeexplore.ieee.org/document/5548085/ (visited on 03/07/2023).

[69] Jung Leng Foo et al. "Path Planning of Unmanned Aerial Vehicles using B-Splines and Particle Swarm Optimization". en. In: *Journal of Aerospace Computing, Information, and Communication* 6.4 (Apr. 2009), pp. 271–290. ISSN: 1542-9423. DOI: 10.2514/1.36917. URL: https://arc.aiaa.org/doi/10.2514/1.36917 (visited on 03/07/2023).

[70] Robbin vanHoek, Jeroen Ploeg, and Henk Nijmeijer. "Cooperative Driving of Automated Vehicles Using B-Splines for Trajectory Planning". en. In: *IEEE Transactions on Intelligent Vehicles* 6.3 (Sept. 2021), pp. 594–604. ISSN: 2379-8904, 2379-8858. DOI: 10.1109/TIV.2021.3072679. URL: https://ieeexplore.ieee.org/document/9415170/ (visited on 12/26/2022).

[71] Timothy Arney. “Dynamic path planning and execution using B-Splines”. en. In: *2007 Third International Conference on Information and Automation for Sustainability*. Melbourne, Australia: IEEE, Dec. 2007, pp. 1–6. ISBN: 978-1-4244-1899-2. DOI: 10.1109/ICIAFS.2007.4544771. URL: http://ieeexplore.ieee.org/document/4544771/ (visited on 03/07/2023).

[72] Eric W. Weisstein. *NURBS Curve*. en. Text. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/ (visited on 03/08/2023).

[73] Fusheng Liang, Chengwei Kang, and Fengzhou Fang. “A smooth tool path planning method on NURBS surface based on the shortest boundary geodesic map”. en. In: *Journal of Manufacturing Processes* 58 (Oct. 2020), pp. 646–658. ISSN: 15266125. DOI: 10.1016/j.jmapro.2020.08.047. URL: https://linkinghub.elsevier.com/retrieve/pii/S1526612520305533 (visited on 03/08/2023).

[74] A.J. Schmid and H. Woern. “Path planning for a humanoid using NURBS curves”. en. In: *IEEE International Conference on Automation Science and Engineering, 2005.* Edmonton, AB, Canada: IEEE, 2005, pp. 351–356. ISBN: 978-0-7803-9425-4. DOI: 10.1109/COASE.2005.1506794. URL: http://ieeexplore.ieee.org/document/1506794/ (visited on 03/08/2023).

[75] Sawssen Jalel, Philippe Marthon, and Atef Hamouda. “NURBS Based Multi-objective Path Planning”. en. In: *Pattern Recognition*. Ed. by Jesús Ariel Carrasco-Ochoa et al. Vol. 9116. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 190–199. ISBN: 978-3-319-19263-5. DOI: 10.1007/978-3-319-19264-2_19. URL: http://link.springer.com/10.1007/978-3-319-19264-2_19 (visited on 03/08/2023).

[76] Eric W. Weisstein. *Cubic Spline*. en. Text. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/ (visited on 03/09/2023).

[77] John Connors and Gabriel Elkaim. “Analysis of a Spline Based, Obstacle Avoiding Path Planning Algorithm”. en. In: *2007 IEEE 65th Vehicular Technology Conference - VTC2007-Spring*. ISSN: 1550-2252. Dublin, Ireland: IEEE, Apr. 2007, pp. 2565–2569.

ISBN: 978-1-4244-0266-3. DOI: 10.1109/VETECS.2007.528. URL: http://ieeexplore.ieee.org/document/4212956/ (visited on 03/09/2023).

[78] Martin Saska et al. "Robot Path Planning using Particle Swarm Optimization of Ferguson Splines". en. In: *2006 IEEE Conference on Emerging Technologies and Factory Automation*. Prague, Czech Republic: IEEE, Sept. 2006, pp. 833–839. ISBN: 978-0-7803-9758-3. DOI: 10.1109/ETFA.2006.355416. URL: http://ieeexplore.ieee.org/document/4178249/ (visited on 03/09/2023).

[79] Jianfang Lian et al. "Cubic Spline Interpolation-Based Robot Path Planning Using a Chaotic Adaptive Particle Swarm Optimization Algorithm". en. In: *Mathematical Problems in Engineering* 2020 (Feb. 2020), pp. 1–20. ISSN: 1024-123X, 1563-5147. DOI: 10.1155/2020/1849240. URL: https://www.hindawi.com/journals/mpe/2020/1849240/ (visited on 03/09/2023).

[80] Tizar Rizano et al. "Global path planning for competitive robotic cars". en. In: *52nd IEEE Conference on Decision and Control*. Firenze: IEEE, Dec. 2013, pp. 4510–4516. ISBN: 978-1-4673-5717-3. DOI: 10.1109/CDC.2013.6760584. URL: http://ieeexplore.ieee.org/document/6760584/ (visited on 03/12/2023).

[81] Keonyup Chu, Minchae Lee, and Myoungho Sunwoo. "Local Path Planning for Off-Road Autonomous Driving With Avoidance of Static Obstacles". en. In: *IEEE Transactions on Intelligent Transportation Systems* 13.4 (Dec. 2012), pp. 1599–1616. ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2012.2198214. URL: http://ieeexplore.ieee.org/document/6203588/ (visited on 03/12/2023).

[82] Saeed Amirfarhangi Bonab and Ali Emadi. "Optimization-based Path Planning for an Autonomous Vehicle in a Racing Track". en. In: *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*. Lisbon, Portugal: IEEE, Oct. 2019, pp. 3823–3828. ISBN: 978-1-72814-878-6. DOI: 10.1109/IECON.2019.8926856. URL: https://ieeexplore.ieee.org/document/8926856/ (visited on 03/12/2023).

[83] Ayoub Raji et al. “Motion Planning and Control for Multi Vehicle Autonomous Racing at High Speeds”. en. In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*. Macau, China: IEEE, Oct. 2022, pp. 2775–2782. ISBN: 978-1-66546-880-0. DOI: 10.1109/ITSC55140.2022.9922239. URL: https://ieeexplore.ieee.org/document/9922239/ (visited on 03/16/2023).

[84] Jose L. Vazquez et al. “Optimization-Based Hierarchical Motion Planning for Autonomous Racing”. en. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Las Vegas, NV, USA: IEEE, Oct. 2020, pp. 2397–2403. ISBN: 978-1-72816-212-6. DOI: 10.1109/IROS45743.2020.9341731. URL: https://ieeexplore.ieee.org/document/9341731/ (visited on 03/16/2023).

[85] Daegyu Lee et al. *A Resilient Navigation and Path Planning System for High-speed Autonomous Race Car*. en. arXiv:2207.12232 [cs, eess]. Sept. 2022. URL: http://arxiv.org/abs/2207.12232 (visited on 03/12/2023).

[86] Johannes Betz et al. “A Software Architecture for the Dynamic Path Planning of an Autonomous Racecar at the Limits of Handling”. en. In: *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*. Graz, Austria: IEEE, Nov. 2019, pp. 1–8. ISBN: 978-1-72810-142-2. DOI: 10.1109/ICCVE45908.2019.8965238. URL: https://ieeexplore.ieee.org/document/8965238/ (visited on 03/12/2023).

[87] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. “A Guide to Heuristic-based Path Planning”. en. In: ().

[88] Guodong Rong et al. *LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving*. en. arXiv:2005.03778 [cs, eess]. June 2020. URL: http://arxiv.org/abs/2005.03778 (visited on 12/26/2022).

[89] *Functional Mockup Interface*. URL: https://fmi-standard.org/ (visited on 01/11/2023).

[90] *SVL Simulator Sunset*. en. URL: https://www.svlsimulator.com/news/2022-01-20-svl-simulator-sunset/ (visited on 01/11/2023).

[91] TUM-Institute of Automotive Technology. *Global Race Trajectory Optimization*. original-date: 2019-05-21T08:29:33Z. Jan. 2023. URL: https://github.com/TUMFTM/global_racetrajectory_optimization (visited on 02/04/2023).

[92] Alexander Heilmeier et al. "Minimum curvature trajectory planning and control for an autonomous race car". en. In: *Vehicle System Dynamics* 58.10 (Oct. 2020), pp. 1497–1527. ISSN: 0042-3114, 1744-5159. DOI: 10.1080/00423114.2019.1631455. URL: https://www.tandfonline.com/doi/full/10.1080/00423114.2019.1631455 (visited on 02/04/2023).

[93] Tim Stahl et al. "Multilayer Graph-Based Trajectory Planning for Race Vehicles in Dynamic Scenarios". en. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. Auckland, New Zealand: IEEE, Oct. 2019, pp. 3149–3154. ISBN: 978-1-5386-7024-8. DOI: 10.1109/ITSC.2019.8917032. URL: https://ieeexplore.ieee.org/document/8917032/ (visited on 12/26/2022).

[94] *Curvature (article)*. en. URL: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/curvature (visited on 02/11/2023).

[95] William F. Milliken and Douglas L. Milliken. *Race Car Vehicle Dynamics*. Fifth. Society of Automotive Engineers, Inc, 195. ISBN: 1-56091-526-9.

[96] William Kennedy. "Steering Actuator Delay Compensation for a Ground Vehicle Lateral Control System". Thesis. 2023.

[97] Autonoma. *Autonoma*. en-US. URL: https://autonomalabs.com/ (visited on 06/07/2023).