Application of RRT for overtaking in a Racing Car Simulation

José Guilherme Freitas Gomes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Miguel de Sousa de Assis Dias
Prof. Carlos António Roque Martinho

November 2019

Acknowledgments

Firstly, I would like to thank both of my supervisors, Prof. João Dias and Prof. Carlos Martinho. They helped me a lot throughout this work, and I regret not asking some questions that sometimes stopped this work development. It was a privilege to work with them, and I would like to thank you for taking me, and helping me become a more capable person.

I would also like to thank Samuel Canada Gomes, the author of the work this thesis is based on. Without any obligation or gain, he helped me with code, some doubts I had about the game and technique, and even gave me some useful advice. Its fair to say he had a great contribution to this work, and I wish him the best of luck in his future endeavours.

Next I would like to thank my father Silvestre and my mother Liliana for continuously supporting me. Even though they were not physically by my side (due to my residence at the time I was studying), they gave me company and constant care, and without them none of this could have been possible. I hope someday the person I become can hope to reciprocate all they did and gave to me. I would also like to thank my grandparents Maria and José, for although not understanding much of what I was doing, always showed that they were extremely proud of me. Through good and bad results, their support never wavered. Finally, I would like to thank my sister Cláudia, for taking care of me all this years, and making me a strong enough person to adventure myself, making this masters far from my zone of confort possible.

I also need to thank Anabela and Ademar Spencer and their kids Guilherme and Gabriel. They took me in their home, barely knowing me, trusting and caring for me, just like a son or a brother. They were my second family throughout these two long years, teaching and helping me overcome my flaws, making me grow in a way I was not able to do alone. They made this journey a reality and I deeply thank them for it.

Finally I would like to thank all my friends especially the ones I made in IST. I came to Lisbon knowing just a long time friend that help me discover this great city, but the people I met in my master's made it a place I could call home. They're all extremely capable and smart, and it is an honour to say I worked with them. To André, Lourenço, Jan, Gonny, Diogo, Bruno, Mariana, and again Samuel, and all the amazing people I met in these two years, I have nothing but gratitude. Without them, this project could have been finished maybe two months earlier, but they made this adventure something I would never regret.

I owe and dedicate the labor of my love, countless caffeine-induced sleepless nights and stress, to all of these people.

Guilherme Gomes

Funchal, 31st October 2019

Abstract

This document describes the application and development of a TORCS *robot* that, on a racing scenario, follows a determined trajectory (referred as racing line) calculated with the K1999 algorithm and, in case of overtaking, one that is traced by a Rapidly-exploring Random Tree based algorithm called Adapt and Overtake-RRT (ADOVER for short), working in two different modes that will later be compared. It is meant to compete against other *robots* and also humans, having as a requirement maintaining an acceptable performance throughout its execution. After some testing with static opponents, the robot was unable to perform the desired task. On the other hand, it showed promising results in terms of speed and efficiency. Possible improvements are discussed in the last segment.

Keywords

Rapidly-Exploring Random Tree (RRT), The Open Racing Car Simulator (TORCS), Overtaking, K1999, Adapt and Overtake Rapidly-Exploring Random Tree (ADOVER-RRT)

Resumo

Este documento descreve a aplicação e desenvolvimento de um robô do jogo TORCS que, num cenário de corrida, segue uma trajetória determinada pelo algoritmo K1999 e, em caso de ultrapassagem, segue uma trajetória determinada pelo algoritmo Rapidly-Exploring Random Tree (RRT), em dois modos de funcionamento diferentes que mais tarde irão ser comparados. Este robô destina-se a competir contra outros robôs e contra humanos, tendo como requisito manter uma performance aceitável durante toda a sua execução. Após alguns testes com oponentes estáticos, o robô não conseguiu desempenhar a tarefa que lhe foi atribuída. Por outro lado, mostrou resultados promissores em termos de velocidade e eficiência. Possíveis melhoramentos são discutidos no último segmento.

Palavras Chave

Rapidly-Exploring Random Tree (RRT), The Open Racing Car Simulator (TORCS), Ultrapassagem, K1999, Adapt and Overtake Rapidly-Exploring Random Tree (ADOVER-RRT)

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

RRT	Rapidly-Exploring Random Tree
TORCS	The Open Racing Car Simulator
CPU	Central Processing Unit
AI	Artificial Intelligence
RSA	Randomized Search Algorithms
LIDAR	Light Detection and Ranging
ANN	Artificial Neural Network
RARS	Robot Auto-Racing Simulator
ABS	Anti-lock Braking System
RPM	Rotations per minute
IPS-RRT	Iterative Parallel Sampling RRT
ADOVER-RRT	Adapt and Overtake RRT

1 Introduction

Contents

Nowadays, Artificial Intelligence (AI) has an increasingly important role in motor racing. “Robots” (“bots” for short), Central Processing Unit (CPU)-driven cars, expanded their presence beyond offline game sessions and can now fill empty multiplayer lobbies or be an important part of a training session of a competitive game. They can also be present in real life races, already existing functional prototypes that can compete against human drivers. Therefore, they must contribute to their good experience, behaving in either a realistic way, with human-like actions and capabilities, or in a customizable way, in which the bot skills can range from amateurish - with slow reactions, some mistakes and easy-to-keep-up-with pace - to perfect in every way - only preferred by drivers who want to test their skills against unfair opponents.

These bots must have some degree of competitiveness, characterised by some key behaviours: following an optimised trajectory, situational awareness, competition awareness, are all part of a quality racing bot. These are all important to achieve an extremely important capability: overtaking. The automation of this manoeuvre is still considered one of the toughest challenges in the development of autonomous vehicles [17], both in road and competitive scenarios, due to the dynamics involved. All things considered, we thought this was a good problem to tackle.

There is also the need to explore the use of Randomized Search Algorithms (RSA) on kinodynamic environments, such as the ones found in a racing game, where the optimal path is key to a highly competitive bot. The one that it is going to be focused is Rapidly-Exploring Random Tree (RRT), a randomised data structure that is specifically designed to handle nonholonomic constraints and high degrees of freedom, capable of solving kinodynamic planning problems, which seems fit to the environment in context.

1.1 Work’s Goal

This work’s goal is to make the robot **perform an overtaking manoeuvre** in a competitive setting, and **design, explore and study the use of RRT algorithm to achieve it**, or to improve upon an already implemented technique. This study can be divided in these main aspects:

- Quality of the solution: Can the robot initially avoid a collision with a static object, adapting his path, to later improve and be able to overtake a dynamic opponent?
- Algorithm performance impact: Can all this be made while maintaining an acceptable game performance, meaning, a smooth framerate is achieved, being possible for a human to compete against it?

A good platform to develop this work is a racing video game that, despite its age (over 20 years old), continues to be updated and applied to a great range of scientific studies and academic research, is

The Open Racing Car Simulator (TORCS) [18]. Its a multi-platform (best compatible with Linux platform) open source game that presents developers with an API conceived for the development of racing robots. Programmers have access to core procedures and already implemented robots that can serve as basis, has a tutorial that details installation and development of a basic car. It still maintains a big community that holds yearly competitions and a forum with people kind enough to answer developers questions.

Considering the benefits, this game was used as a platform for this work.

1.2 Outline

This thesis is organised as follows: In chapter 2 the present work that exists in terms of racing AI, racing line calculation, RRT algorithm with improvements, and a brief discussion about frame rates, will be detailed. In chapter 3 the K1999 will be presented and briefly described (as it is used to calculate the racing line), and the ADOVER-RRT algorithm will also be explained. Chapter 4 details the bot architecture, its search state representation, how ADOVER-RRT is applied to TORCS, how the K1999 is calculating and representing the racing line, and how the path traced by the ADOVER-RRT is being followed. The Chapter 5 firstly details the tests that were made along the development of the robot with a preceding study about the quality of the returned solutions, concluding with a discussion. The document ends with the chapter 6 where the general work is evaluated, and some notes for future work that should improve the results are presented.

2 Related Work

Contents

This section starts with a brief reflection on what makes a racing game robot good, the behaviours that it should display, and some examples that implement said features. Given this work objective focus on overtaking, only the behaviours that influence overtaking (and said behaviour itself) will be described. One real-life and several video game robots are depicted and compared. Only the implementation quality will be compared, and what is to be learned from each one. It is important to note that these examples are commercial, meaning that neither the source code nor extensive documentation is available to the public. Consequently, the behaviour quality assessment will be made through gameplay observation and, for the real-life example, company citation.

Then, approaches to the racing line calculation will be explained, taking in consideration TORCS tracks characteristics and how each algorithm described can be applied to it.

The proposed algorithm RRT will be described, along with some variants that try to improve its performance and/or attempt to parallelize it, with a mention to an application of an RRT variant applied to it. The solution quality/game performance trade-off and how it should be pondered over when deciding the technique will also be explained.

A discussion will close this chapter.

2.1 Racing AI

2.1.1 Capabilities and behaviours

Has previously stated, a robot needs to be competitive. Analysing a track, tracing a trajectory through it and completing it, is a prerequisite for it, but if the robot can only focus and follow the line that it calculated, it is not competing against the player, just against itself. While more complex behaviours can be considered, the ones here detailed (if correctly executed) make for a realistic high difficulty robot. Low level difficulty requires other characteristics that will not be implemented in this work, but will be addressed as a positive feature in one of the racing game examples. The characteristics that will be considered are the following [19] [20]:

A – Advanced car control A good robot uses everything that the car has to offer, and the actions performed by it must not be limited to accelerating , breaking and turning. Correct use of the gear box, ABS, traction control, understeer and oversteer correction etc.. are techniques that not only improve on the quality of the solution but also increases its realism.

B – Collision avoidance The robot should be able to avoid obstacles (mainly opponents and track objects) in order to, not only perform a better time around the track, but also maintain the quality of interaction with the player. In real life racing (e.g. *F1*), pilots who intentionally collide with opponents, causing avoidable accidents, or leave the track for no apparent reason, are faced with time penalties, and in some cases suspensions [21]. This is not always the case, since there are some competitions that collisions are common (Rally cross), so the robot must be aware of the rules of the simulated environment.

C – Overtaking When competing, the agent will be faced with a situation that is extremely common in racing: the opponent on front of him is slower than him, and there's an opportunity to traverse a trajectory around this opponent. A good AI must be able to correctly detect this situation, measuring and analysing the track section that both are currently in, and attempt it if proven worth it, in a way that respects the rules referred above (although light touches are acceptable).

D – Aggressiveness It's a positive factor, but too much can negatively impact player experience. This behaviour will mainly be demonstrated on overtakes. Taking in consideration the situation in which the robot can, or gets overtaken, a passive agent will maintain his race line while carefully avoiding the player. This attitude favours a clean but easy race, only adequate to inexperienced or rookie players, since the robot can be easily ignored and beaten. However, if the AI's aggressive factor increases, it may reach a point where it will protect his position or tries to gain one at all costs, meaning that it can actively try to bump/collide the player making him loose control and going off track. Although it increases competitiveness, it can break some rules and will deny the player a friendly experience, sometimes even inciting him to respond with equal degree of aggression, increasing the probability of not terminating the race due to damage. Given these observations, an ideal agent will protect its position and overtake when deemed practicable, but will never disrupt other opponents in a "dirty", violent way. This is a factor hard to balance, and many times it's the player's choice, represented by a difficulty level or by a separate configuration.

2.1.2 Examples

Considering what was described in the previous section, a real life, and some video game examples that apply such characteristics are presented. It is explained what can be learned in each case and then compared their quality of implementation. As cited in the beginning of this section, the main information source will be observation and gameplay.

2.1.2.A *Roborace* Robocar/Devbot (2016) [1]

Robocar and *Devbot* are the platform cars that will be competing on a race, consisting of only similar cars, the only difference between the competitors being the team-developed algorithms. This car learns how to traverse a circuit thanks to virtual simulations. A bank of computers simulate the environment physical aspects and previously read Light Detection and Ranging (LIDAR) and ultrasound readings are replicated. Then, simulated GPS location and virtual machine vision are interpreted and the equipped *Nvidia PX-2* computer processes said information and calculates the optimal path. Besides already following a close to optimal racing line, overtaking is already being developed, and it was successfully executed in testing environment. Collision avoidance is a top priority, higher than usual including this work, due to vehicle cost. This project shows that the work made in video-games can be executed in real time with physical vehicles, and behaviours that are considered important are the same as this project : collision avoidance and overtaking.

2.1.2.B *rFactor 2* Image Space Incorporated (2013) [2]

A very good example on how to implement racing AI in general, so I'll consider this the baseline of this comparison. The behaviours that I am focusing can be easily observed in multi-class racing and *rFactor 2* gets recognition for implementing them correctly in said situation. When different vehicle classes with very different capabilities compete at the same time, the circuit gets crowded and overtakes occur frequently. Since this type of racing is one of the main focus of this game, agents in this game highly prioritise collision avoidance, and overtaking manoeuvres display great quality. They rarely get delayed by slower opponents. A clean race is very easy to maintain, not because opponents don't protect their position, but because they do it without colliding with the player. Talking about overtakes, it is easily observed the aggressiveness of more powerful cars and the correct behaviour of less powerful ones. In the first case, faster opponents quickly assess the situations and overtake with barely any hesitation, efficiently using all the track width and free space. In the second case, while equally powerful competitors may put up a fight, these just free the needed space and facilitate overtaking. The player can adjust the difficulty, even having the possibility of giving the bots an unfair advantage (giving their cars more traction/power), their aggressiveness and their variety, represented by a "AI Limit" percentage, where a 0 percent means that the bots are not restricted to a small range of values and can display a great range of performances, and a 100 limit makes them more consistent, making them follow similar lines. These lines are precalculated for each available track.

A – *Assetto Corsa* Kunos Simulazioni (2014) [3] Bots display moderate aggression and overall realistic behaviours but they're too passive while overtaking. They respect the line that the player is following but even in situations where overtaking is viable, they slow the pace and do not overtake, only doing it when it is already too easy (the player went off track or slowed down to much).

B – *RaceRoom Racing Experience* Sector3 Studios (2013) [4] With Adaptive AI, the agents racing style and difficulty fit those of the player, improving the experience. Although this adaptation is good, it still needs some improvement. Again, good overall behaviour, but can sometimes get too aggressive, making clean long sessions hard to achieve. Overtakes are well executed but the consideration in them is too optimistic, leading to accidents.

C – *Project CARS 2* Sligthly Mad Studios (2017) [5] Another strong contender to best AI, has a slight defect: not always the agents adapts and gets ready correctly for the conditions of the event (inadequate tires). It is always important to the AI to consider the maximum of factors that may influence vehicle performance. This will not be compared since it depends on simulation realism, not only AI.

D – *Gran Turismo Sport* Polyphony Digital (2017) [6] Adaptable difficulty, with high difficulty bots behaving too perfectly, making no mistakes and maintaining consistency, which is a slight low point. But in lower difficulties, the AI emulates realistically a rookie driver making similar mistakes, like keeping an uneven but slower pace and misjudging some turns.

E – *F1 2018* Codemasters (2018) [7] Adaptable, too aggressive in high difficulty although realistic, given current F1 drivers. It is improved over its previous iteration F1 2017, where the aggression was basically non-existent, again showing that when is to overtake is almost as important as how to do it.

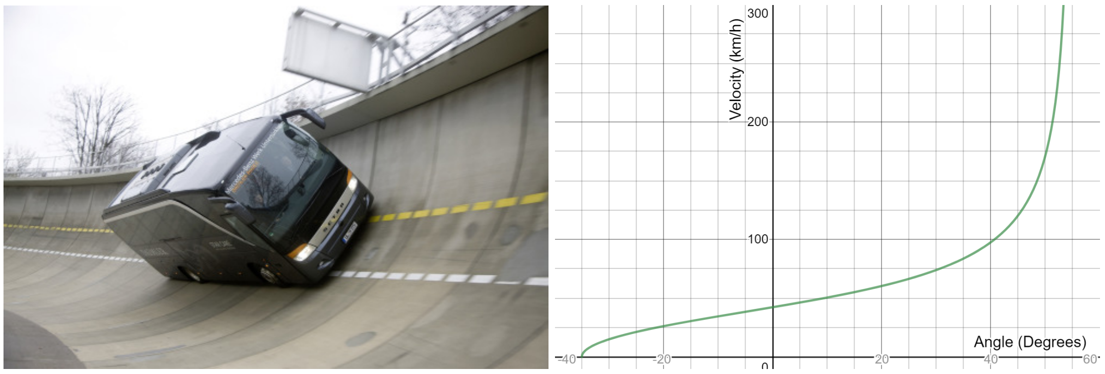
2.1.2.C Comparison table

Title	**Collision Avoidance**	**Overtaking**	**Notes**
rFactor 2	Very good	Very good	Comparison baseline
Roborace	Top priority, well executed	In development	Real life example
AC	Too prioritised	Too passive	Needs aggression
RaceRoom	Line too prioritised	Too aggressive	Risky
PC 2	Very good	Very good	Few factors considered
GT Sport	Very good	Very good	Realistic low diff.
F1 2018	Very good	Too aggressive	Realistic high diff.

2.2 Racing Line

Foremost, the context in which the racing line is drawn and followed is defined as a circuit, or race track. A circuit is generally a closed path, meaning that a racing robot will start, go around it, and end the lap in the same segment. In order to achieve the best results, racing line algorithms must take in consideration the physical characteristics of said track. The more variables and the more precise the information it collects, the realest the virtual track it can model, meaning it can prepare for more situations and further optimise the route the robot must go through. It is also important to consider the physical limitations of the car, most importantly its turn radius (the smallest circle it can describe), torque and top speed, friction with the road, aerodynamics, and so on. TORCS tracks have a fixed-width, and are generally flat, meaning simpler algorithms can trace a decent racing line, although the aforementioned still holds. All the information about the cars are also accessible and taken in consideration.

One of the first characteristics of the track that must be considered is its 2-dimensional shape [22] [8]. Not only the angle of its turns but sequence of its segments must be considered for an optimal result. The width must also be measured, with fixed width segments being the simplest to work with. Important 3-dimensional features include elevation and tilt (also called bank) variation. These are usually present in physical circuits and they affect the racing line radius and speed profile. Steep hills negatively impact acceleration and slopes reduce brakes effectiveness, whereas road banking, i.e. the angle in which the road is inclined about its longitudinal axis with respect to the horizontal, can reduce or increase the effect of inertia, if inclined towards the inside or outside of the turn respectively, making it possible to turn at higher speeds, effect shown in Fig.2.1. The speed in which an automobile can make a banked turn is calculated with formula introduced in the next section.



(a) A bus traversing a sharp banked corner at about 100km/h [23].

(b) Relation between max. speed and turn bank angle, with the data from a 20 meter radius turn, using the formula present in 2.6

Figure 2.1: Effect of road inclination on maximum possible vehicle speed.

This path is important to achieve the maximum possible velocity per segment, making the track traversal quicker. There have been several ways to trace this desired path, ranging from purely geometric to Artificial Neural Network (ANN) based techniques, but all of them follow share a basic principle - the straighter the line, the fastest a car can go through it.

2.2.1 Geometric approach

If we consider the track shape and perform some geometric calculations (considering the formula in 2.5), we can find the maximum radius we can draw around a turn. There are more advanced geometric approaches, (for example Braghin's Approach in [22]), but this one is enough to introduce some basic concepts.

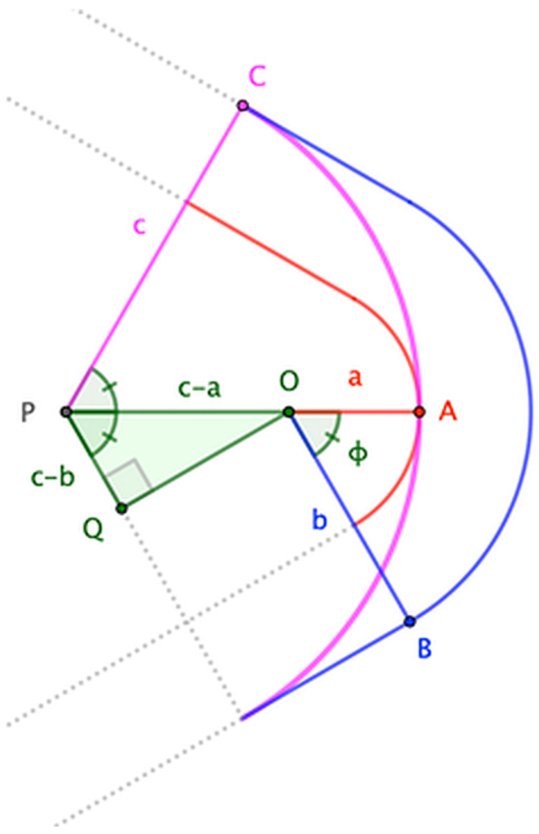


Figure 2.2: Racing line arc scheme. Optimal radius c is unknown. [13]

As shown in Fig.2.2 , we want to find c:

c Is the racing line circumference radius, circumscribed by the track limits, with centre at P.

ϕ Half the turn angle.

a **and** b Inner and outer turn radius, with centre at O.

A **and** C Tangential point between the desired line and the track limits.

$\triangle OPQ$ Right triangle where the hypotenuse is lined by P and O.

$$\cos\phi = \frac{c-b}{c-a} \tag{2.1}$$

$$(c-a)\cos\phi = c-b \tag{2.2}$$

$$b - a\cos\phi = c(1-\cos\phi) \tag{2.3}$$

$$c = \frac{b-a\cos\phi}{1-\cos\phi} \tag{2.4}$$

We can then write :

$$c = \frac{b-a+a-a\cos\phi}{1-\cos\phi} = \frac{b-a}{1-\cos\phi} + \frac{a(1-\cos\phi)}{1-\cos\phi} = a + \frac{b-a}{1-\cos\phi} \tag{2.5}$$

Given the radius of the ideal arch the car has to describe, we can now calculate its desired speed profile, i.e. the target speed the car has to be. Since we want to complete the lap in the least amount of time, will be the maximum allowed velocity. Given:

v_{max} as the maximum velocity an automobile can make the turn.

r the radius of the turn.

g Constant gravitational acceleration.

θ Incline angle, with a positive angle meaning the lowest side is inside the curve.

μ_s Coefficient of static friction.

$$v_{max} = \sqrt{\frac{rg(\tan\theta + \mu_s)}{1 - \mu_s \tan\theta}} \tag{2.6}$$

This formula can also be simplified for flat turns:

$$v_{max} = \sqrt{rg\mu_s} \tag{2.7}$$

It is important to note that aerodynamics can also be introduced in this formula, by previously adding the force that air resistance exerts to the one exerted by gravity. Knowing the radius and maximum speed allowed, we can fill a table with the Fig.2.2 measurements:

μ_s	$g(m/s)$	$\theta(^o)$	$r(m)$	$v_{max}(m/s)$
0.5	9.81	0	30	12.12
			80	19.80
		15	80	26.37

Table 2.1: Comparison between different turn angles with the same radius, and resulting max. velocity, using an example turn of 30 meters, and its ideal radius of 80 meters calculated with formula 2.5, and speed calculated with 2.6.

Analysing the table 2.1 shows that increasing the radius of the arch described by the car allows for an increased maximum velocity, as well as an increased inclination angle towards the centre of the turn. Considering this, it is clearly beneficial to follow the optimised path.

Although this technique allows for the optimisation of single segments, it does not take in consideration how the trajectory followed in a segment might affect the one that is to be followed in the preceding segment. An important thing to consider while taking a corner is the exit speed, the speed that one goes through at the apex of the turn. Due to acceleration and braking time asymmetry, it might be beneficial to brake before the recommended point and go through a tighter route, reaching the point of intersection between the two lines with a higher speed, as demonstrated in Fig.2.3. This leaves room for further improvement.

2.2.2 K1999 Path-Optimisation Algorithm [8]

This is an algorithm that is much more efficient than reinforcement learning techniques [8]. It was implemented in Robot Auto-Racing Simulator (RARS) and TORCS, and in this second one the car follows the target trajectory using a servo-control. The resulting path is approximated by a sequence of points $(\vec{x}_i)_{1\leq i\leq n}$. The curvature c_i of the track at each point $\vec{x}_i$ is computed as the inverse of the radius of the circumscribed circle for points $\vec{x}_{i-1}$, $\vec{x}_i$ and $\vec{x}_{i+1}$. The formula is presented in 2.8.

$$c_i = \frac{2\det((\vec{x}_{i+1}) - (\vec{x}_i), (\vec{x}_{i-1}) - (\vec{x}_i))}{\| \vec{x}_{i+1} - \vec{x}_i \| \| \vec{x}_i - \vec{x}_{i-1} \| \| \vec{x}_{i+1} - \vec{x}_{i-1} \|} \tag{2.8}$$

The $(\vec{x}_i)_{1\leq i\leq n}$ points are initially set to the centre of the track. The resulting curvature after is positive for turns to the left, and negative to the right. By repeating the calculation made in 2.8, the path will be slowly modified, converging to an optimised path. The algorithm is presented in 2.1.

Algorithm 2.1: K1999 basic algorithm

```
begin
    for $i = 1$ to $n$ do
        $c_1 \longleftarrow c_{i-1}$
        $c_2 \longleftarrow c_{i+1}$
        Set $\vec{x}_i$ at equal distance to $\vec{x}_{i+1}$ and $\vec{x}_{i-1}$ so that $c_i = \frac{1}{2}(c_1 + c_2)$
        if $\vec{x}_i$ is out of the track then
            Move $\vec{x}_i$ back onto the track
```

After the path has been traced, the speed profile - in this case the target speed v_i - can be calculated. It is obtained by taking in consideration turn radius and tyre grip, similarly to the geometric approach; if the norm is inferior to a_0, it will not slip. This is done with two procedures: first in 2.2 , s_i being the speed value of the current node, is initialised with the maximum vehicle speed. Then in 2.3 the speed is refined anticipating braking. This second step is iterated, converging to an optimised speed profile.

Algorithm 2.2: Speed Profile Initialisation

```
begin
    for $i = 1$ to $n$ do
        if $|c_i| > \varepsilon$ then                    // $\varepsilon$ is a small positive constant
            $s_i \longleftarrow \sqrt{a_0/c_i}$
        else
            $s_i \longleftarrow \sqrt{a_0/\varepsilon}$
```

There are already ways to refine this algorithm so it can converge faster, consider security margins, the asymmetry between car acceleration and braking capabilities and curves inflection. These are implemented using Gradient Descent and changing the variation of $\vec{x}_i$ upon path calculation in the

Algorithm 2.3: Speed Profile Optimisation

```
begin
    for $i = 1$ to $n$ do
        $N \longleftarrow \frac{1}{2}(c_{i-1} + c_i)v_i^2$                 // Normal Acceleration
        $T \longleftarrow \sqrt{max(0, a_0^2 - N^2)}$            // Tangential Acceleration
        $v \longleftarrow \frac{1}{2}(v_i + v_{i-1})$
        $D \longleftarrow kv^2$                                    // Air drag
        $t \longleftarrow ||\vec{x}_i - \vec{x}_{i-1}||/v$
        $v_{i-1} \longleftarrow min(s_{i_1}, v_i + t(T + D))$
```

basic algorithm. A path generated using this algorithm, comparing both basic and optimises versions is shown in Fig. 2.4. With this technique the robot will achieve excellent lap times with no performance repercussions, but will be rather poor at passing due to it relying to much to the precomputed trajectory. The algorithm can retrace the route while on race, but will prioritise its path and not an aggressive overtaking tactic, failing to avoid collisions with immobile vehicles, or getting delayed by staying behind slower cars that offer clear overtaking possibilities.

2.2.3 *Race Optimal AL* © [9]

This is a genetic algorithm method that converges to an optimal racing line. It uses real life circuits models and calculates the speed profile using client-inputted vehicle data. Its lap simulation first step is similar to the geometric approach, since it only considers the 2D shape of the track, and calculates the fastest possible speed at every point based on curvature radius. Physical constrains are considered in the subsequent adjustments: acceleration does not exceed what the tires can handle and engine can provide, and braking is limited by tire friction and aerodynamic drag. The *Race Optimal AL* ©pseudo-algorithm is described as follows:

1. Set up an initial population of solutions with random variations. Each solution will be a path - a complete trajectory around the track - built with a set of control points (illustrated in Fig.2.5). Typical population is between 40 to 100 paths.

2. Solutions are bred together to create offspring. These children represent combinations of the original parents.

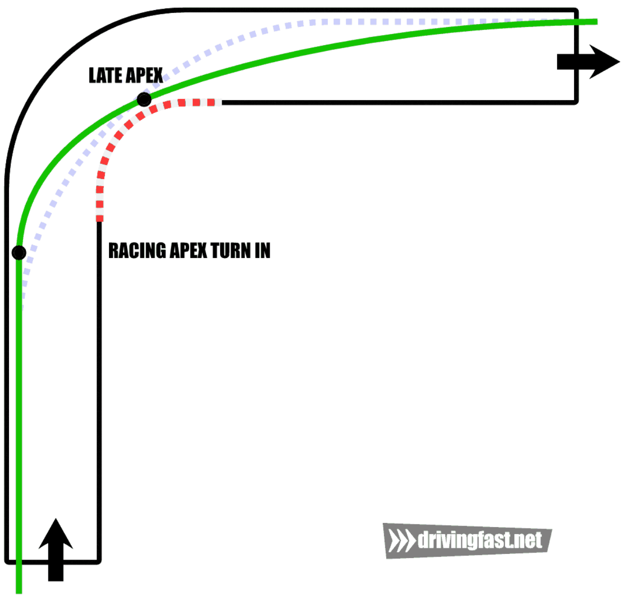
3. Apply mutation to the offspring. This means creating random changes on the child path control points, in order to more thoroughly search all possible racing lines.

4. Considering the fitness criteria for the selection being simulated lap time (said simulation described very broadly in the beginning of this section), the children and parents are selected, and the

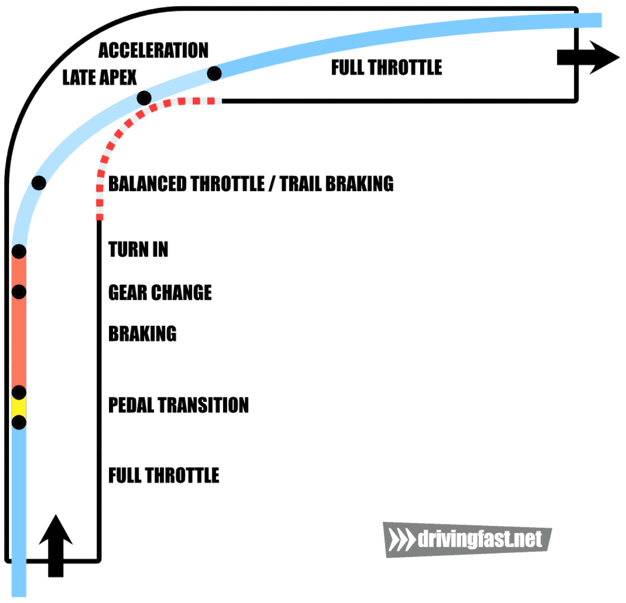
best(shortest lap time) will be retained for the next generation. The others “die off” and the end of this process marks the end of a generation.

5. Repeat from Step 2 until fitness criteria or the minimum number of generations is satisfied.

An example of a full path calculated by the *Race Optimal*©algorithm is shown in Fig. 2.5 and a path segment simulation displaying the calculated speed profile, input, and g-force, is shown in Fig.2.6.



(a) Comparison between the geometrical line dotted in grey, and the late apex trajectory, in green.



(b) Benefits of following the late apex trajectory. Comparing with the geometrical line shown in 2.3(a), the full throttle segment is longer, allowing a greater exit speed.

Figure 2.3: Comparison between trajectories and benefits of following the late apex trajectory. Both images taken from [14].

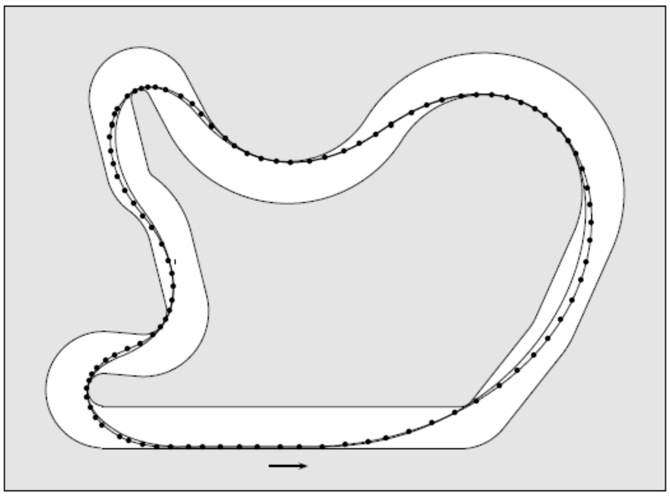


Figure 2.4: Algorithm 2.1 before (dotted) and after gradient descent (not dotted) [8].

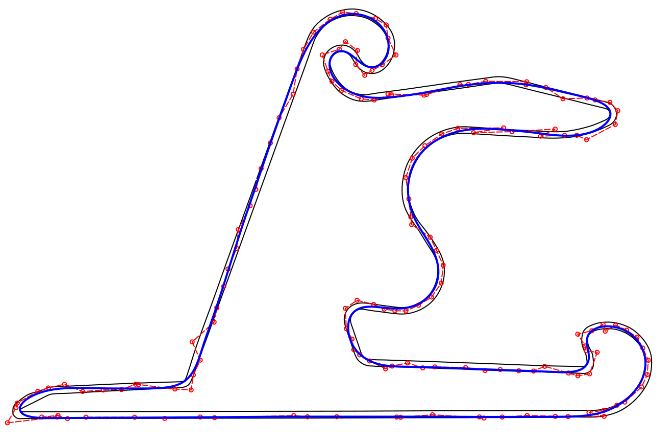


Figure 2.5: Control points (red) and resulting racing line (blue) for the Porsche GT3 RS 4.0 at Shanghai International Circuit [9].

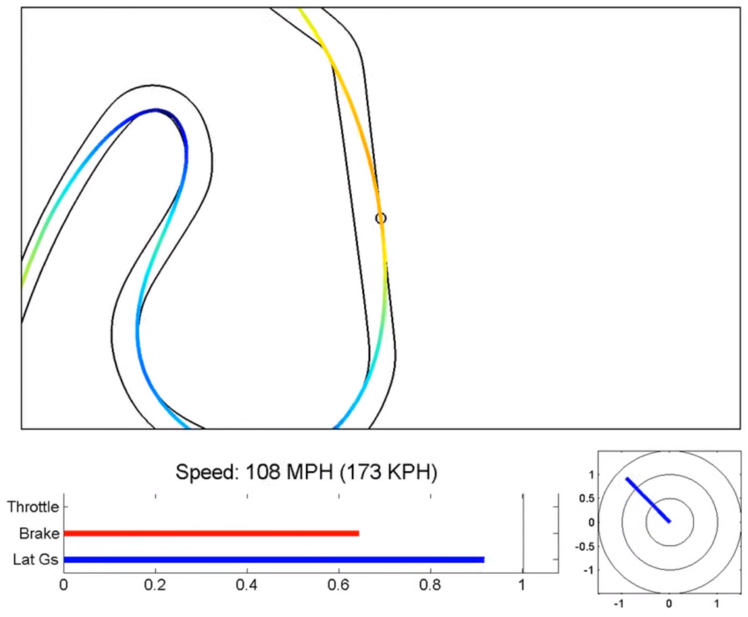


Figure 2.6: TT Circuit Assen - F1 Car - Race Optimal Simulated Lap segment. The higher the colour temperature, the higher the target speed. Throttle, brake and g-force information is also shown [9].

2.3 RRT

In this section will be discussed the relevant works that implemented RRT and variants that improve the performance of the base algorithm.

2.3.1 Base Algorithm [10]

Randomised data structure designed to handle nonholonomic constraints (including dynamics) and high degrees of freedom. It is iteratively expanded by applying control inputs that drive the system slightly toward randomly selected points. Can also handle kinodynamic environments, which is TORCS case. It will search for a continuous path in a metric space X, with and initial point x_{init} to a goal state or region x_{goal}. Its *state space* can be a 2D or 3D world, whereas in kinodynamic planning problems both configuration and velocity can be encoded in each state. Other interpretations of X are also possible.

A new node cannot be present within X_{obs} ,an obstacle region with no explicit representation available. This region may present bounds to configurations, velocity, etc... All states (and therefore the complete tree and paths) must lie in X_{free}, the complement of X_{obs}. They are connected through *state transition equations* of the form $\dot{x} = f(x, u)$, with u being an input vector, that expresses the nonholonomic constraints. This control-theoretic representation is powerful enough to encode virtually any kinematic and dynamical model. By integrating f over a fixed time interval Δt, new state x_{new} can be determined given x and input u, this being the return of $NEW\_STATE(x, u, \Delta t)$. For a given initial state, x_{init}, an RRT T with K vertices is constructed as shown in Algorithm 2.4:

Algorithm 2.4: Basic algorithm

```
begin
    $T$.init($x_{init}$)
    for $k = 1$ to $K$ do
        $x_{rand} \longleftarrow RANDOM\_STATE()$
        $x_{near} \longleftarrow NEAREST\_NEIGHBOUR(x_{rand}, T)$
        $u \longleftarrow SELECT\_INPUT(x_{rand}, x_{near})$
        $x_{new} \longleftarrow NEW\_STATE(x_{near}, u, \Delta t)$
        $T.add\_vertex(x_{new})$
        $T.add\_edge(x_{near}, x_{new}, u)$
    return $T$
```

Let ρ denote a distance metric on the space state. In more detail:

1. The tree starts at x_{init}, a state within X_{free}.
2. In each iteration a random state x_{rand} is picked from X, assumed bounded.
3. x_{near} is found, this being the closest vertex, i.e. already connected state, in terms of ρ.
4. Then a u that minimises the distance from x_{near} to x_{rand} is selected, and ensures that $x_{new} \exists X_{free}$. Collision detection can be performed here.
5. x_{new} is created and added to T as a vertex.
6. An edge between x_{near} and x_{new} is created, with u recorded in said edge, meaning it must be applied in x_{near} to reach x_{new}.

This algorithm has several advantages that make it desirable for this application. As stated, it can handle kinodynamic environments with nonholonomic constraints and high degrees of freedom. It is heavily biased towards unexplored portions of the state space, unlike naive random trees as showing in Fig.2.7. Some steps can be changed and further simplified to best fit certain environments constraints and goals, but in his basis form it is relatively simple to implement and can be further optimised to be light and iterative. Unfortunately, since the tree stops expanding once it reaches K vertices or its goal, and the connections between states never change, it is prone to generating jagged paths and never converges to an optimal solution. We present some improvements that fix these issues.

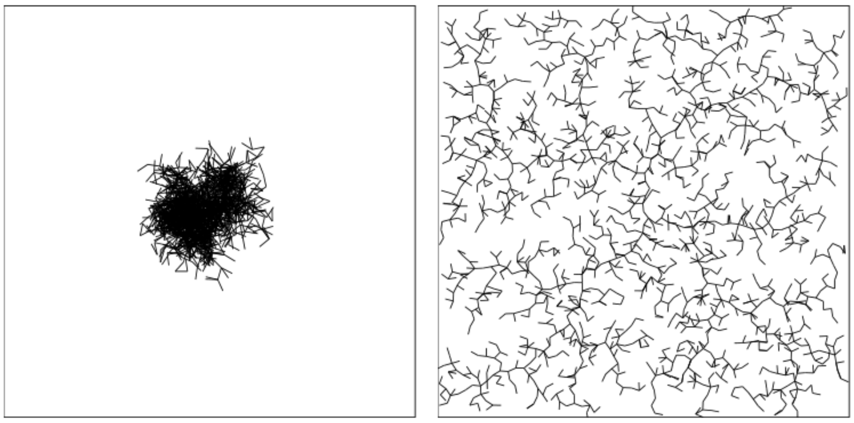


Figure 2.7: A Naive Random Tree vs. an RRT , each with 2000 vertices [10].

2.3.2 RRT* [11]

RRT* is an incremental sampling based algorithm which finds an initial path very quickly, and later improves said path, smoothing and making it shorter, as the number of samples increases throughout its execution. It works very similarly to RRT and inherits all of its properties. However, it introduces two new features that allow for said improvements: **Near Neighbour Search** and **Rewiring Tree** operations. This is shown in algorithm 2.5.

Each vertex now hold the information of its self travel distance relative to its parent, which will be referred as edge cost. Before x_{new} insertion after u is selected, $NEAR$ returns all states (X_{nbr}) within the ball with centre in x_{new}, with radius defined by equation 2.9 , where γ is a planning constant.

$$k = \gamma(\frac{\log(n)}{n}) \tag{2.9}$$

$CHOOSE\_PARENT$ returns x_{min}, the X_{nbr} state that has the least edge cost, and in $INSERT\_NODE$ x_{new} is added to T with it as parent, with the new edge cost added. Finally, $REWIRE$ will rebuild the tree within the radius of area k, changing connections between states to ensure a minimal edge cost

Algorithm 2.5: RRT*

```
begin
    $T$.init($x_{init}$)
    for $k = 1$ to $K$ do
        $x_{rand} \longleftarrow RANDOM\_STATE()$
        $x_{near} \longleftarrow NEAREST\_NEIGHBOUR(x_{rand}, T)$
        $u \longleftarrow SELECT\_INPUT(x_{rand}, x_{near})$
        if $x_{new} \exists X_{free}$ then
            $X_{nbr} \longleftarrow NEAR(T, x_{new})$
            $x_{min} \longleftarrow CHOOSE\_PARENT(X_{nbr}, x_{near}, x_{new})$
            $T \longleftarrow INSERT\_NODE(T, x_{min}, x_{new})$
            $T \longleftarrow REWIRE(T, X_{nbr}, x_{min}, x_{new})$
    return $T$
```

between them, and a smoother path. A comparison between RRT and RRT* between trees can be seen in Fig.2.8.

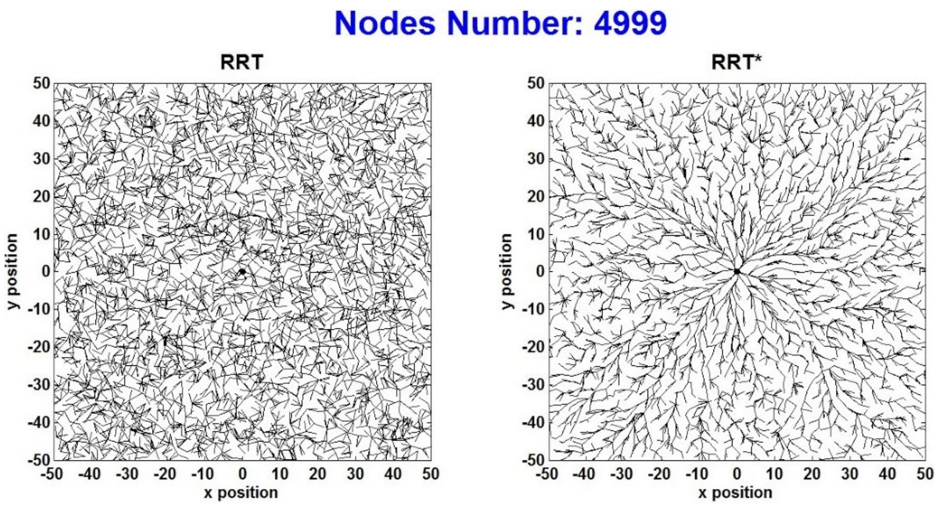


Figure 2.8: RRT vs RRT*. Note the smooth branching on RRT*. Image taken from [15].

It has a larger overhead and development complexity, but it manages to slowly converge to the optimal solution, ensuring asymptotic optimality, although not in a finite time [11]. This is not practical, so, to overcome this limitation, an improvement over this technique is presented.

2.3.3 RRT*-Smart [12]

As stated previously, RRT*-Smart objective is to overcome RRT*'s convergence time, and it does by introducing two new key concepts - **Intelligent sampling** and **Path Optimisation**. The process is outlined in algorithm 2.6.

Initially, RRT*-Smart works in the same way. However, once the first path is found, $InitialPathFound$

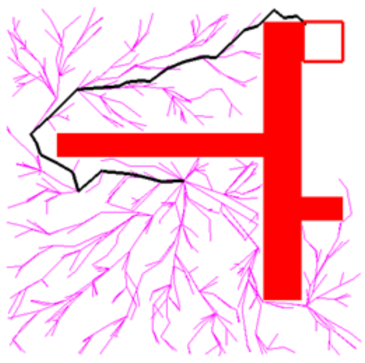
Algorithm 2.6: RRT*-Smart

```
begin
    $T$.init($x_{init}$)
    for $k = 1$ to $K$ do
        if $k = n + b, n + 2b, n + 3b...$ then
            $x_{rand} \longleftarrow RANDOM\_STATE(x_{beacons})$
        else
            $x_{rand} \longleftarrow RANDOM\_STATE()$
            $x_{near} \longleftarrow NEAREST\_NEIGHBOUR(x_{rand}, T)$
            $u \longleftarrow SELECT\_INPUT(x_{rand}, x_{near})$
            if $x_{new} \exists X_{free}$ then
                $X_{nbr} \longleftarrow NEAR(T, x_{new})$
                $x_{min} \longleftarrow CHOOSE\_PARENT(X_{nbr}, x_{near}, x_{new})$
                $T \longleftarrow INSERT\_NODE(T, x_{min}, x_{new})$
                $T \longleftarrow REWIRE(T, X_{nbr}, x_{min}, x_{new})$
                if $InitialPathFound$ then
                    $n \longleftarrow k$
                $(T, directCost) \longleftarrow PATH\_OPTIMISATION(T, x_{init}, x_{goal})$
                if $directCost_{new} < directCost_{old}$ then
                    $x_{beacons} \longleftarrow PATH\_OPTIMISATION(T, x_{init}, x_{goal})$
    return $T$
```

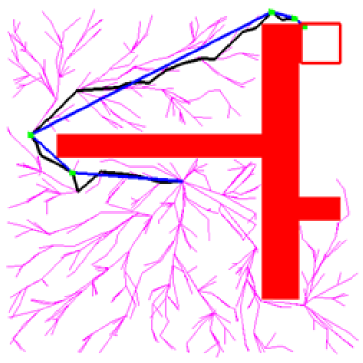
returns the iteration number n at which it was found. Then n and a biasing constant b are used to inform the algorithm on when to start biasing sampling. $PATH\_OPTIMISATION()$ initially updates the path $directCost$, directly connecting the nodes visible to each other, and due to triangle inequality, their distance shortens, reducing total path cost.

When a shorter path is found by re-iterating the RRT* algorithm, *beacons* - the basis nodes for intelligent sampling - are requested and generated by $PATH_{O}PTIMISATION$; else, the old beacons remain. In $x_{rand} \longleftarrow RANDOM\_STATE(x_{beacons})$ samples are being spawned within the radius with centre in $x_{beacons}$. These are generated in locations where the algorithm needs a bigger node concentration, aiming to decrease edge cost. After these initial beacons are found, intelligent sampling is activated, spawning new samples in the previously defined vicinity, with a biasing probability calculated in 2.10, where B is a programmer dependent constant. This ratio has a trade-off between rate of convergence and search space exploration. The final result is shown in Fig.2.9

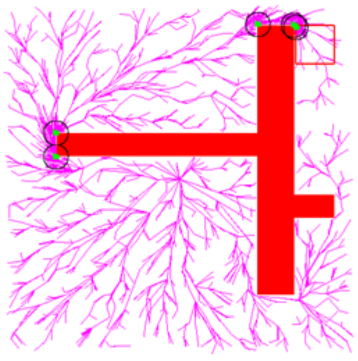
$$BiasingRatio = \frac{n}{x_{free}} * B \tag{2.10}$$



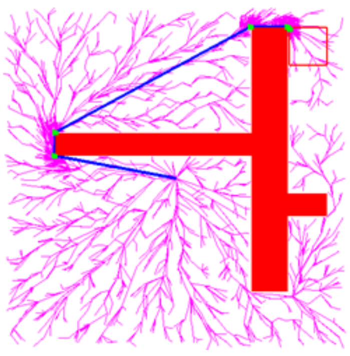
(a) Path given by RRT*



(b) First path optimisation



(c) Biasing towards beacons. Notice the high node concentration



(d) Optimal path

Figure 2.9: RRT*-Smart steps. Images taken from [12].

2.3.4 Parallelization

There have been various efforts to implement parallelization in RRT, so it could run in real time or more efficiently. In [24] the constraint check (collision detection and invalid point culling) is parallelized, this being considered by the author the main application of GPGPU to improve performance, whereas in [25] the tree generation is paralleled with a method called *Sampling-based Roadmap of Trees* (SRT). It takes several samples from the search space, and uses those samples as the initial node of several trees, that can either find a local solution or be joint to find one. Joining trees is possible through simple paths or bi-directional planning.

2.3.4.A IPS-RRT applied to TORCS

In [16] the algorithm Iterative Parallel Sampling RRT (IPS-RRT) is used. This is a variation of RRT that allows parallelization, since it focus on executing several iterations of parallel tree samples. It is very similar to RRT and improves from other algorithms with a parallelization effort. At each iteration, a number of samples are concurrently generated and checked for constraints, and the valid samples are synchronised with the global tree at the end of each iteration. The number of iterations and generated samples can be changed to fit system performance, and if both are the same, the algorithms performs exactly like RRT. The pseudo code is presented in algorithm [2.7]. IPS-RRT threads generate only one sample between synchronisations, and global tree updates are seen in each iteration, making this algorithm suitable for its main purpose, GPGPU. Due to this schedule, IPS-RRT trees are different from the basis RRT. They expand in breadth, rather than in depth, having their new points closer to the initial point.

Algorithm 2.7: IPS-RRT

```
begin
    for $n = 0$ to $nIters$ do
        Start $nParallelSamples$ threads executing:
            $T' \longleftarrow T$
            $x_{rand} \longleftarrow RANDOM\_STATE()$
            $x_{near} \longleftarrow NEAREST\_NEIGHBOUR(x_{rand}, T)$
            $x_{new} \longleftarrow APPLY\_DELTA(x_{rand}, x_{new})$
        if $!validPoint(x_{new})$ then
            $END\_THREAD()$
            $T'.addVertex(x_{new})$
            $T'.addEdge(x_{near}, x_{new})$
            $SYNCHRONISE\_THREADS(), T \longleftarrow all\ T's$
    return $T$
```

Applied to TORCS, this algorithm generated a new point with a predicted position and velocity in each game iteration (Fig. 2.10). Each state's input is not calculated on demand, but inferred afterwards by the control module, that is constituted by the steering and pedals PID controller. In the process of building the tree, threads do not keep copies of the main tree; instead the neighbours are picked from a constant memory tree, updated in each iteration. This is practical and efficient, but the GPU memory slow speed negatively impacts this procedure, limiting tree expansion. This affected the robot performance, that followed a tree that did not have enough depth, this revealed when it failed to prepare for, e.g., tight corners that preceded long straights, a big disadvantage in racing.

2.4 Frame rate and algorithm impact

Should a robot compete against a human player, it must not impact the simulation in such a way that the frame rates goes below what it is deemed acceptable - a value that, although subjective, sits around 24 frames per second, according to [26]. So a frame update cannot take more than $\approx$ 0.0417 seconds to

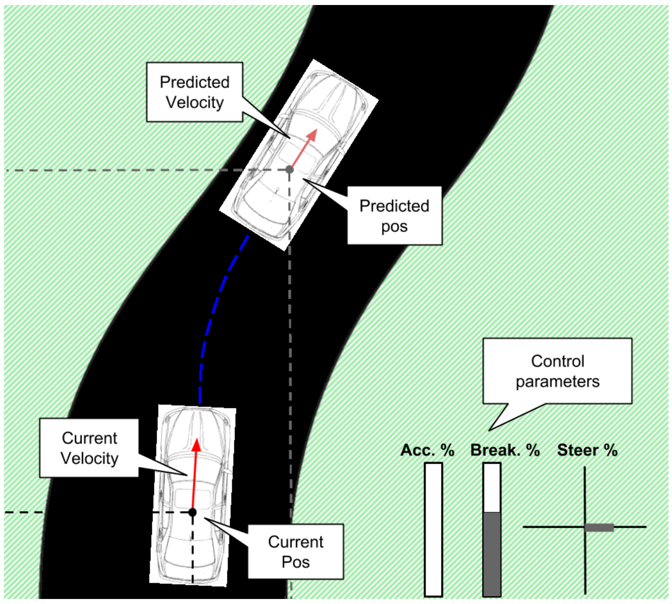


Figure 2.10: Attributes in each IPS-RRT state [16].

update. This is also noteworthy considering TORCS simulation is frame independent - it only depends on the real execution time - meaning that interruptions to the came core procedures will display jumps instead of smooth transitions, negatively impacting game experience.

On the other hand, for the algorithm to be viable, it must cover enough state space, and have a sufficient information saved in each state, in order to best calculate a good solution. It was already stated that, given enough time, the racing line and tree algorithms will output the ideal solution, either being the ideal racing path or the shortest path (we will consider the first the desirable one due to already explained dynamics). Given that a time limit for each update is now imposed, a sufficient amount of states might not be reached, and the robot might suffer from complications as the ones shown in IPS-RRT [16]. The algorithm has enough time before the race starts, but a before-race algorithm can't adapt to on-race situations, reducing solution quality.

This trade-off must be considered, and a possible balance will be presented in the following section.

2.5 Discussion

Due to it being documented, supported by an active community, and presenting already implemented robots that will be used has a basis for this work, TORCS was chosen as the case to study. Its API provide all the information and procedures needed for the creation of various robots, including this work.

As said before, TORCS is open-source game with already implemented robots, constituted by modules that will be of great use for this work. One of them is the *inferno* robot. It implements the K1999 algorithm and it follows its calculated path through an also already implemented servo-motor controller [8].

K1999 displays great results and can be adapted to work collectively with another algorithm, so this work will try improve upon it.

Since we want to find the best overtaking path while racing, we need an algorithm that is light and easy to adapt, but capable enough to handle the constraints and dynamics of the proposed environment and task. Also, given that the overtaking path is not always the shortest, but the smoothest and one that respects the car properties, algorithms like the RRT*-Smart carry too much overhead, are not needed. The basis RRT fits our needs: it has a small overhead, an easy implementation, and it is possible to encode the dynamics of the vehicle within each state [27]. A smooth path can be achieved with bézier lines, or by simply reducing branch jaggedness through angle limitations, as will be shown in chapter 3. Considering the trade-off explained in the previous section, its flexibility is appealing, enough to trace the trajectory on-race or pre-built it before the race starts, with small changes. Both alternatives are tested in this work. This solution can be adapted to fit the states the K1999 algorithm output that the car controller then receives, meaning he can follow said trajectory if its adapted. This will be detailed in chapter 4.

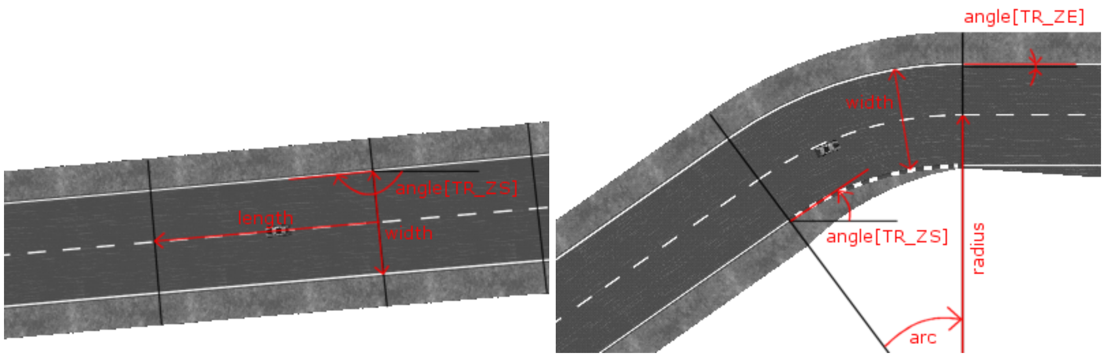
3 Proposed solution

Contents

This section details the processes that work together to accomplish this study objective. These algorithms rely on the available track information, so how the tracks are represented in-game will be shown here. Since it was picked to trace the initial line the car follows, and the implemented car controller receives its states in order to guide the car through the track, the K1999 will be briefly detailed in TORCS context, adding information to what was already detailed.

3.1 TORCS Track Representation

The K1999 and ADOVER-RRT use the track information available through TORCS Back-end, upon loading before the race starts, to calculate the racing line and then adapt it. According to [18], these tracks are composed by a limited number of fixed-width segments that can be classified as turns or straights. Straights have a length, width, and angle between concurrent segments, shown in Fig.3.1(a). Turns are arch segments, with a turn radius, arch size, width and angle between concurrent segments, shown in Fig.3.1(b).



(a) Straight track section **(b)** Turn track section

Figure 3.1: Track segment types in TORCS.

3.2 K1999

Considering the description already presented in chapter 2 and the track structure description previously described, this technique applied to TORCS (together with some supporting functions) works as follows:

1. When the track gets loaded, its description and each segment physical characteristics get stored.
2. An object of the car class is created, it containing the physical information about the car, its current situation, and its plan (path).

3. The opponents, track, pit, and current situation information are passed to the plan, and the initial static route is created, with one path segment per track segment.

4. The race starts and the robot starts updating his status and environment situation.

5. With the new information, the dynamic route is updated. It tries to maintain this on-race route as similar to its static counterpart, since its considered the optimal path.

6. In case it finds an obstacle within a predetermined range, it slowly converges the route to the new best one, maintaining smoothness.

7. The servo-motor controller receives each path segment information and responds accordingly, applying input to match the desired car state (speed and position).

The result for the in-game track *E-Track 1* is displayed in Fig.3.3. A diagram that shows this algorithm cycle can be seen in Fig.3.4. The difference between the two traced paths can be seen in Fig.3.2.

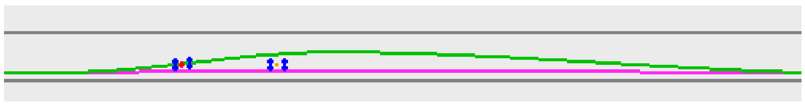


Figure 3.2: The blue dots represent the cars corners, the dynamic path is represented in green and the static, optimal path in pink. The agent using the K1999 retraces its current path and redraws it around the opponent it finds withing look-ahead distance.

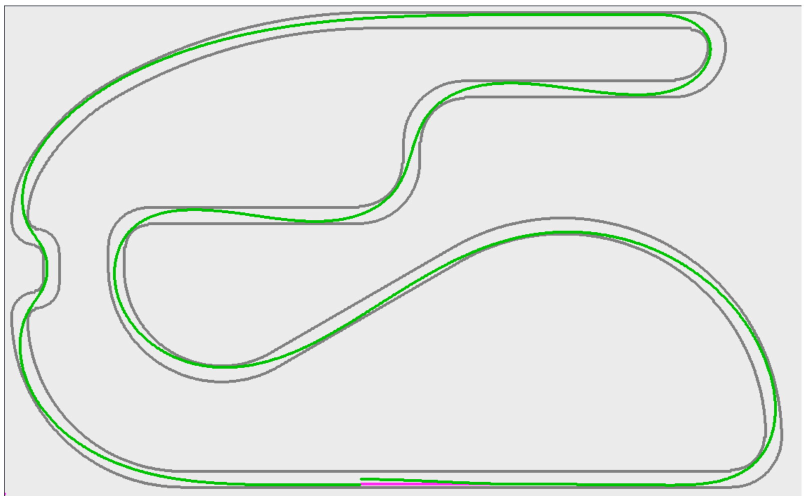
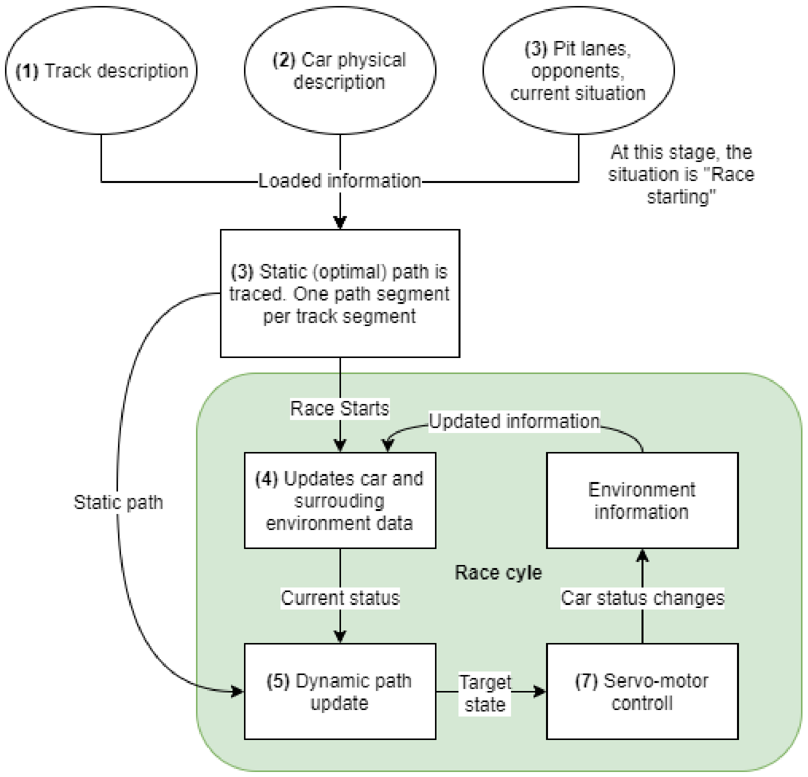


Figure 3.3: Both lines (dynamic in green and static in pink) traced by K1999 at the start of the race.

3.3 ADOVER-RRT

The RRT was the proposed algorithm to work on, so it was developed and adapted to best fit the needs of this work. As a result a variant of the RRT was created, called Adapt and Overtake RRT (ADOVER-RRT).



(1) Track description
(2) Car physical description
(3) Pit lanes, opponents, current situation
At this stage, the situation is "Race starting"
Loaded information
(3) Static (optimal) path is traced. One path segment per track segment
Static path
Race Starts
Updated information
(4) Updates car and surrouding environment data
Environment information
Race cyle
Current status
Car status changes
(5) Dynamic path update
Target state
(7) Servo-motor controll

Figure 3.4: Scheme with the K1999 cycle, as implemented in TORCS. The states are numbered according to the steps enumeration presented at the start of the section.

The adaptations will be described in this section, with the implementation and further detailing of certain characteristics in chapter 4.

3.3.1 Description

We want to compare on-race and pre-race situations. On-race refers to the tree being built while the car (and player, if present) race, meaning the expansion process will be part of the car update cycle. Pre-race, pre-built, pre-computed and so on, all refer to when the tree is completely expanded after the track is loaded, before the race starts, meaning the expansion process will not be concurrent to the car update process. For both, the base algorithm might remain the same, but some parameters change, as the path adaptation process.

If we want to expand the tree to preferably completely fill the track - connecting two points in either segment of it - the tree goal (and stopping condition), becomes the tree having a target number of vertices. Another condition might be acceptable, but we can always ignore the number of iterations it takes, since it does not impact in race performance. The overtaking trajectory will be bounded between two vertices that are found while on race.

On the other hand, if we want the algorithm to construct its tree *while* the robot is racing, we have to look at the other side of the already stated trade-off, and now the number of iterations per state (or how long it takes to add a new vertex) matters, meaning we have to limit how many executions per frame or how long it takes to add a new state. The pseudo code is described in algorithm 3.1:

Algorithm 3.1: ADOVER-RRT

```
begin
    $T$.init($x_{init}$)
    for $k = 1$ to $K$ do
        $angle \longleftarrow 0$
        if ($!ON\_RACE$) or ($ON\_RACE$ and $FRAME\%FREQ == 0$) then
            repeat
                $x_{rand} \longleftarrow RANDOM\_STATE()$
                $x_{near} \longleftarrow NEAREST\_NEIGHBOUR(x_{rand}, T)$
                $x_{step} \longleftarrow STEP(x_{near}, x_{rand}, stepsize)$
                if $x_{step} \nexists X_{free}$ then
                    continue
                if $x_{near}.parent \neq$ null then
                    $angle \longleftarrow BRANCH\_ANGLE(x_{near}, x_{step}, x_{near}.parent)$
                    if $angle \leq angleLimit$ then
                        continue
                $T \longleftarrow INSERT\_NODE(T, x_{near}, x_{step})$
                *A state was added*
            until state added
    return $T$
```

This pseudo algorithm works according to the two "modes" presented as possibilities: with $!ON\_RACE$ or ON_RACE and once every $FREQ$ $FRAMES$. The first option makes the tree expand completely offline, with x_{init} in the middle of a predetermined track segment, and growing until its size reaches at least K nodes. The second option is to expand the tree while the robot is racing, starting only when we want to overtake. x_{init} and goal will be offsets of the closest opponent position. When the distance between the last node added and the goal is less than a programmer-set distance, it stops expanding.

Due to ADOVER-RRT being similar to RRT - the tree is never retraced, with distances and connections between already added vertices never changing - paths traced by this algorithm can possible be jagged. Since we wanted to avoid this issue without introducing too much overhead, an angle limitation was added : x_{new} would have to make an angle, with x_{near} being the angle vertex, and its parent the third point, of **at least** a programmer set angle, visualised in 3.6. If we consider P as the parent of x_{near}, N as x_{near} and S as a proposed x_{new} , $\measuredangle PNS$ angle is calculated with:

$$\arccos\left((\overline{NS}^2 + \overline{NP}^2 + \overline{PS}^2)/(2 * \overline{NS}^2 * \overline{NP}^2)\right) \tag{3.1}$$

The effect is shown in 3.5 and its necessity to this project in chapter 5. This angle value was decided through trial and observation of the subsequent drawn trajectories. It always improves path smoothness in a pre-race and on-race scenario, but due to it reducing the probability a node is added to the tree, it takes a toll on the expansion speed, that affect primarily on-race generation. This method of angle calculation was chosen, simply due to it being the first proposed, implemented and tested, existing other viable calculation, for example vector dot product.

Although their working frequency and stop conditions are different, both on-race and pre-race trees share their expansion process:

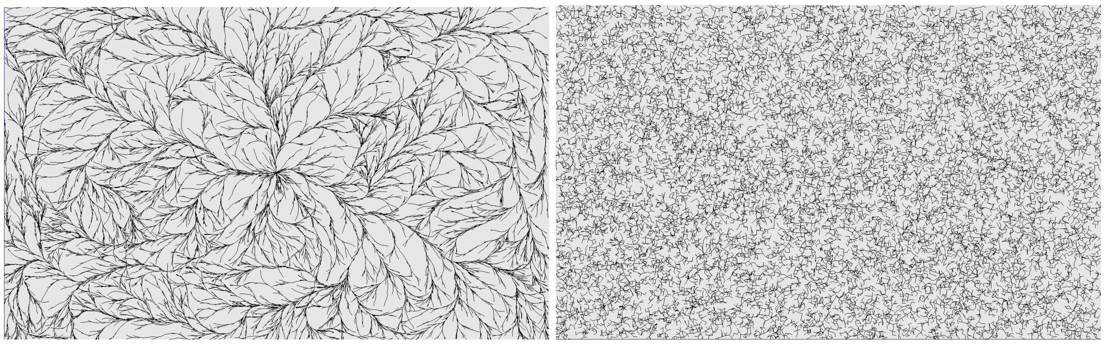
1. x_{rand} is generated as a random position within map space (inside and outside the racing track).
2. x_{near} is the tree node that has the least euclidean distance.
3. x_{step} is a point, $stepsize$ far from x_{near}, collinear to x_{near} and x_{rand}.
4. If x_{step} sits outside X_{free}, (X_{obs} detailed in chap. chapter 4), x_{step} is rejected and the process restarts.
5. If x_{near} has a parent, to measure the angle $\measuredangle PNS$. If this angle is less than the steer lock or a programmer defined angle, x_{step} is rejected and the process restarts.
6. x_{step} is added to T, becoming x_{new}.

3.3.2 Characteristics

The presented algorithm is flexible, light and fast enough to be run on-race, and with enough time it can completely fill the track with nodes, allowing for a trajectory between any track segments, which fits an offline process. The algorithm speed may also be attributed to its simple collision detection. There are some wasted resources due to nodes getting rejected, but the tracing of trajectories too close to the borders is avoided, as well as some of the tree jaggedness due to the branch angle bounds, shown in Fig. 3.5.

3.4 Path Adjustment

This process is what ties the racing line calculation algorithm K1999 and the tree building algorithm ADOVER-RRT. The main reason of its existence is due to the fact that the robot that serves as the basis for this project, *inferno*, follows the path K1999 generates with a servo-motor controller, that receives as target K1999 path segments, each encoded with a desired car position and velocity. This means the tree algorithm only has to alter the position of these target nodes, and does not have to encode velocity



(a) With angle limited to 160 degrees **(b)** With no angle limitation

Figure 3.5: Effects of branch angle limitation, both with 25000 vertices and same step size.

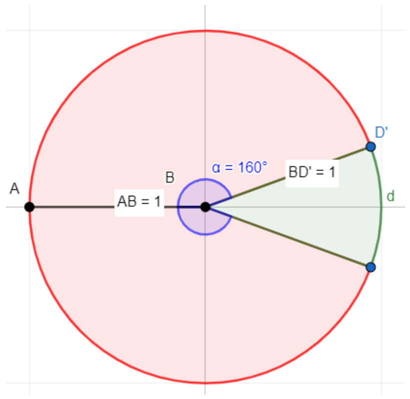


Figure 3.6: Considering B being x_{near} with parent A, D (x_{step}) can only be extended inside d arch.

in its own states. The controller complies with the modified positions, targets them, and adapts the car speed to properly follow them. This eased development, increased code readability, and reduced tree expansion overhead. It is also a simple process by itself, with a very low execution time.

The path adjustment is only needed when the car detects it needs to overtake an opponent, and it does that by checking if the closest opponent is currently at look-ahead distance (a K1999 defined constant), is slower than him and leaves room for this manoeuvre to be executed (by measuring the lateral distance between him and the closest track border). If the opponent verifies all these three conditions, its position is used to find the goal index (n track segments forward) and the start index (n track segments backwards). With these indexes it is possible to find the respective K1999 optimal path segment, and the overtaking trajectory will connect these two positions.

It slightly differs between pre-built and on-race trees. If the tree is fully expanded before the race, the adaptation process starts by the time the car detects it needs to overtake an opponent. The first node that is copied to the new trajectory vector is the desired goal, and from that vertex until the last node added sits close enough to the desired start, it will check if the current vertex has a parent and copy said parent to the new vector. This backtracking process is fast but not very reliable, since it does not consider current opponent position, and that could result in a collision. A possible improvement is presented in chapter 6.

With on-race expansion, the first node on the new trajectory, which is also x_{init}, will be the desired start. The tree will begin to expand at a defined rate and it will only stop when the last node added sits close enough to the desired goal. When the tree stops expanding, x_{init} is already connected to x_{goal}, so the backtracking that is done with the pre-race tree is also done here, copying every node from x_{goal} to x_{init}. During the expansion, the opponent position is considered, with this tree building impacting update performance, but better adapting to a dynamic situation.

After either one of these processes are complete, the copied vertices are used as references to the new path. The K1999 path is retraced with the following algorithm 3.2:

Algorithm 3.2: Path adaptation

```
begin
    foreach vertex element in Path vector do
        Find the closest optimal path segment to this element
        Change the optimal path segment to position of this element
        Change the dynamic path segment to position of this element
```

These updated segments are then sent to the control module, allowing for the car to follow the new trajectory. Notice that no velocity information is altered in this process, but the updated path needs to be smooth enough, else it will not follow these altered states, or will fail to follow any one, loosing control. An example of path retracing can be seen in Fig.3.7.

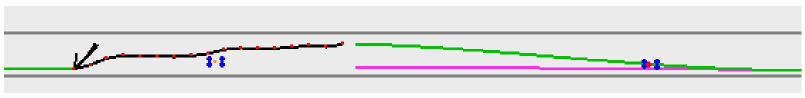


Figure 3.7: RRT path retracing, with the new trajectory in black, with nodes in red. Notice how the K1999 then creates a return path to the optimal trajectory, only at the end of the retraced line.

4 TORCS Robot Implementation

Contents

This chapter details the implementation of the developed robot using the ADOVER-RRT algorithm. It has its architecture presented, with each module elaborated, preceded by an explanation of the state representation, path creation and particularities of the algorithm.

4.1 Bot Architecture

The TORCS bot is composed by several modules, with the developed algorithm functioning in the *Retracing Module*. These consist of the following:

- **TORCS back-end** provides all the necessary procedures and information to control the car and it fully understand its environment.
- **Racing line module** is where the *K1999 Path Optimisation* algorithm calculates the racing line.
- **Retracing module** where the ADOVER-RRT receives information from the racing line, builds a tree, and retraces its path when needed.
- **Control Module** uses a servo-motor controller, destined to control every aspect of the car, so to best match the target state sent by the racing line module to the current state of the car.

The robot is also composed by an information module, a test-only module that displays visual and text data, collected from the other modules. It will not be detailed in this chapter since it is not essential. This architecture is represented in the scheme shown in Fig.4.1. It also applies to both of the proposed on-race and pre-race modes, since their changes are reflected inside the **RRT** sub-module.

4.1.1 TORCS Back-end

It it composed by 3 main sub-modules; **Track, Car and Opponents**. The first one sends to the *Retracing* and *Racing Line* modules, after the track is chosen and loaded into the race, a raw description of the track, containing its size, segment list and description, and others. The Car sub module outputs the vehicle current information and its constraints, and constantly receives commands, driving the car. These commands are the gas pedal, the brake pedal, the gear and the steering angle. They are car dependant, varying on number of gears and steer lock. The Opponents sub-module is very similar to the Car module (given that they are all cars) but it is restricted to showing information about the current adversaries in race, most importantly their speed and current position.

4.1.2 Racing Line

Upon receiving the complete track, it traces a static and an initial dynamic path, built as detailed in chapter 3. In race, it continuously communicates with both the back-end and control module, updating

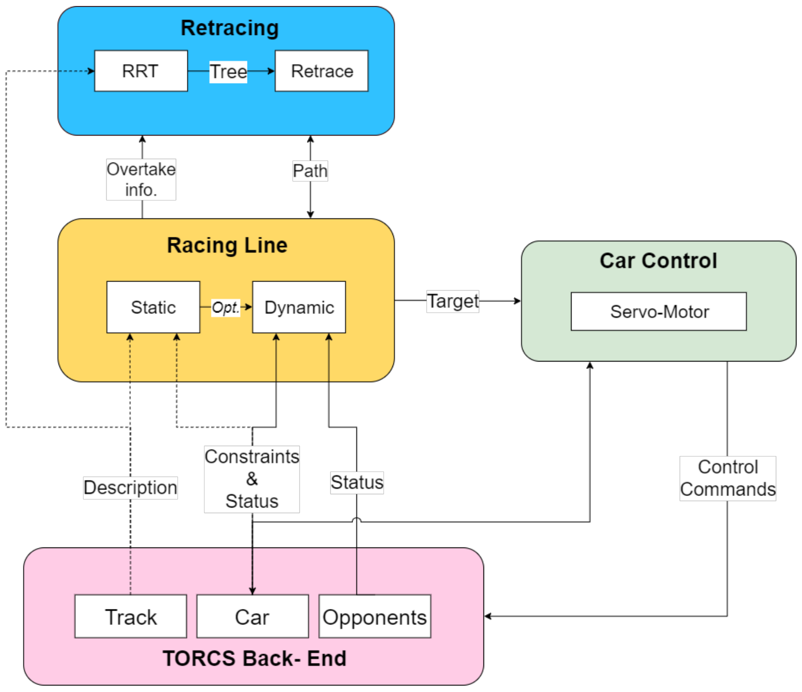


Figure 4.1: Solid lines represent continuous transmission, with dashed lines being information sent once, when the sub module starts.

the dynamic path information and car situation, outputting the position and velocity the car must reach in each segment. It actively tries to maintain the dynamic path as close to its static path, as the second one is considered the optimal, but it is only possible if there are no obstacles in it. To better handle situations it might face regarding its adversaries, it has a structure that uses the data it gets from the Opponent sub module. It is filled with useful information, such as which car to overtake, when to do it, where to do it, and when it is expected to complete it, among others, that are used in the *Retracing* module.

This module currently operates with the *K1999 Path Optimisation* algorithm, but since its output are states that are required to be understood by the *Car Control* module, it can hold other search algorithms or racing line calculation techniques that perform the same function. An example is the architecture used in [16], where the planning is made by the IPS-RRT algorithm.

4.1.3 Retracing

As detailed in chapter 3.4, upon receiving the confirmation that the path needs to be retraced, it outputs the adapted trajectory. If its working while racing, it constantly receives information about its environment and competitors, so to know when and where to act. After tree expansion, the trajectory will be traced between the detected start and goal nodes, and the K1999 algorithm update will be interrupted while the car is following it. A path adjustment can be seen in Fig.3.7. If off-race, it only retraces when needed, using the vertices already created.

4.1.4 Control Module

The module that controls the car is divided in three sub modules: pedals, steering, and gears. It drives the car using the functions the *TORCS Back-end* provides, and has as a target the information it receives from the *Racing line* module. It outputs the commands and their values to the Back-end.This module was not developed in this work, but it will be described so to better understand some decisions made relating to the other modules.

For the pedals, the actions regulated are acceleration and braking. The car will get up to the maximum speed allowed by the segment, every time it has a clear path, meaning the acceleration command $car.\_accelCmd$ will be at 1. The braking is regulated by a braking coefficient, that considers both track and tyre friction, a look ahead factor, and the path segment target speed. Its objective is stopping the car from exceeding the target speed, go out of the path/turn due to his angular velocity, and avoid flying caused by track pitch. It will use the $car.\_brakeCmd$, with its value regulated by an Anti-lock Braking System (ABS) system, preventing the wheels from blocking and counteracting tire slippage.

With the updated target position, the steer angle will be obtained with the following formula:

$$\begin{aligned} targetAngle = \arctan(target.y - car.pos.y, target.x - car.pos.x); \\ targetAngle = car.yaw; \\ targetAngle = targetAngle \in [-\pi, \pi]; \\ steer = targetAngle/car.steerLock; \end{aligned}$$

So it is constantly checking for the next path segment position, its angle in relation to it, and steering the car, maintaining course. It is worth nothing that the vehicle has a limited steering angle, represented by $car.\_steerLock$, meaning that its turn radius is physically limited, with smoother transitions between segments being favoured. The *Retracing* and *Racing* modules both respect this limitation, the first updating its path segments positions with smoothing operations, the second limiting the tree new branch angle, ensuring there are no abrupt steering changes in either situation.

Finally, the $car.\_gear$ changes its gears when needed. It initially stores the car engine red line, which are the maximum recommended engine Rotations per minute (RPM), setting this value as an upper bound. In race, it constantly updates on the current RPM, and, if it reaches a percentage limit (usually around 95%), the gear is changed to a higher existing one. It also considers the car current speed; if its either too fast or too slow for the current gear, it respectively ups or downs a gear. This maintains the engine RPM in an acceptable value between an upper and lower bound where the acceleration capabilities are maximised and the engine is safe from damage.

4.2 Search State Representation

As already described, when tracing the optimal static path, the K1999 algorithm takes into account the track segment description and slowly converges the target velocity to an optimal value for each path segment. The dynamic path also updates each path segment velocity, re-calculating it by updating the radius of the offsetted segments. This means it can also adapt to the trajectory changes made by the *Retracing* module. This eases development and removes the overhead from velocity calculations, with the information encoded in each state coming down to just global coordinates. They will always exist within track boundaries defined by *TORCS Back-end*, with the later path adjustment process being viable anywhere in the track.

4.3 ADOVER-RRT applied to TORCS

As stated in section 4.1.3, this is the algorithm that has the function of retracing the path initially calculated by the *Racing Line* module. Considering its workings description in section 3, the parameters, processes and output will be explained within TORCS context and depicted. The complete state space X shares its limits with the map boundaries, with x_{rand} being created within this region. x_{near} will be the closest already existing node, and since each state only has the position representation, this vertex will be the closest in terms of euclidean distance. A collinear point to both of these points is created, with a fixed step size distance from x_{near}. This point is then validated, i.e., the algorithm checks if this point does not lie in X_{obs}. The invalid state space X_{obs} is defined by three spaces/conditions that a new state must not lie on or verify in order to be accepted and added to the tree. The first space is defined by the distance between the proposed position and the two dimensional coordinates of the middle of the closest-to-position segment. This is shown in Fig.4.2. While this is not as precise has a boundary check using segment corner positions, its is simpler to implement and establish a safety margin. Also, due to segment size, the arches seen around the centre position in Fig.4.2 are negligible. These track margins can be visualised in Fig.4.3.

The second condition the state must verify is the angle limitation introduced in chapter chapter 3 and visualised in Fig.3.6. This limits the new branch angle, and results in smoother tree, and consequently smoother paths. Taking in consideration what can be seen in Fig.4.3, we can compare it to Fig.4.4 to visualise the impact.

Finally, to better adapt to the situation faced when the manoeuvre starts a third condition applies to the on-race tree. While expanding, a new state cannot be closer than a programmer-defined constant to the opponent it wants to overtake. This is to avoid colliding with him and promote a safer trajectory. Any state that sits too close is discarded.

Having successfully checked all of these conditions, this new state is added to the tree and the

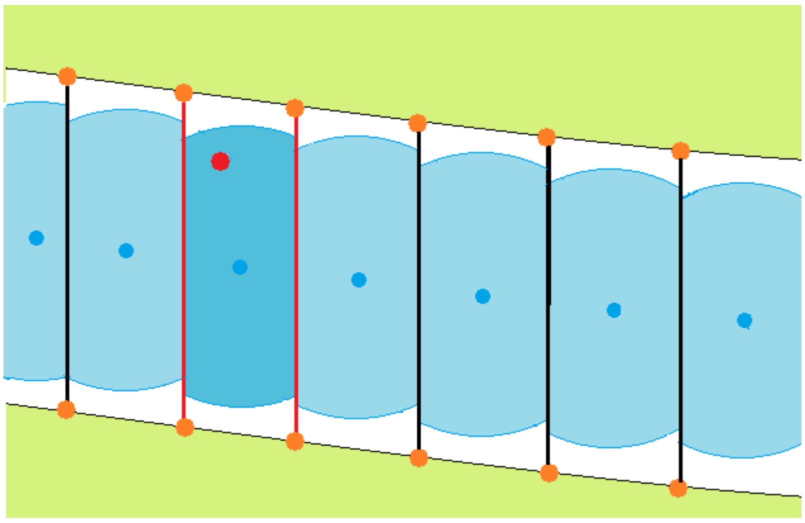


Figure 4.2: The red dot is x_{rand}. The closest segment to it is outlined in red, with each segment mid position marked by a blue dot. Since this position is close enough to the middle of its segment, it sits in a valid position.

process gets repeated until a stopping condition is reached. If the tree is being built while on race, it will stop being built once the last state added reaches a minimum programmable distance to the found goal. Said goal is found similarly to the start of this tree - both are positions that lie a certain number of segments ahead and behind the opponent we want to overtake. If the tree is built before the race starts, the root sits in the middle of a pre-determined track segment, and the stopping condition is it having a minimum K vertices. A better way to determine K could have been developed, but for now it is a simple integer, so the programmer needs to check manually, using the test module, if the tree reaches every segment of the track.

The overtaking behaviour starts by detecting a valid candidate. As detailed in chapter 3, this opponent will be closest than a defined distance, will be in a position where is possible to complete the manoeuvre (meaning its lateral distance to the track borders are enough to fit my car), and its current speed will be lower than ours. The closest opponent that respects these conditions will be flagged as an *overtakee*, as it has been referred until now. When an opponent gets flagged, the tree starts building. When this process finished, both start and goal positions are found by simply adding and subtracting to the track segment index the opponent currently is, and since there is one K1999 path segment per track segment, they both get set to static path segment positions.

The path adjustment then begins being, as stated, a very simple process. While on-race, it first saves the tree states that directly connect the goal to the root, by copying the current state to a new vector, changing the current state to its parent, and repeat until it is *null*, a condition that only root satisfies. Then the distances between the elements of this new vector, and the static path segments are measured. The closest path segment of both dynamic and static path is then set to the position of the closest vector,

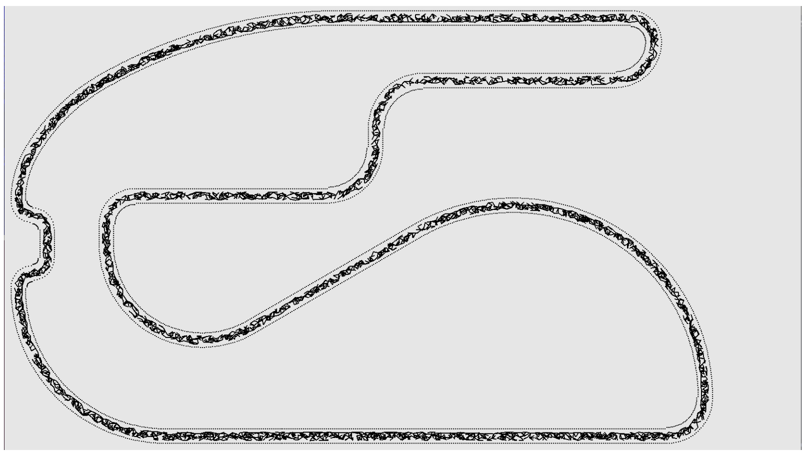


Figure 4.3: Tree built pre-race, with no angle limitation and 10000 nodes, and step size 6. Notice how no state is outside of the track and it within track safe margins.

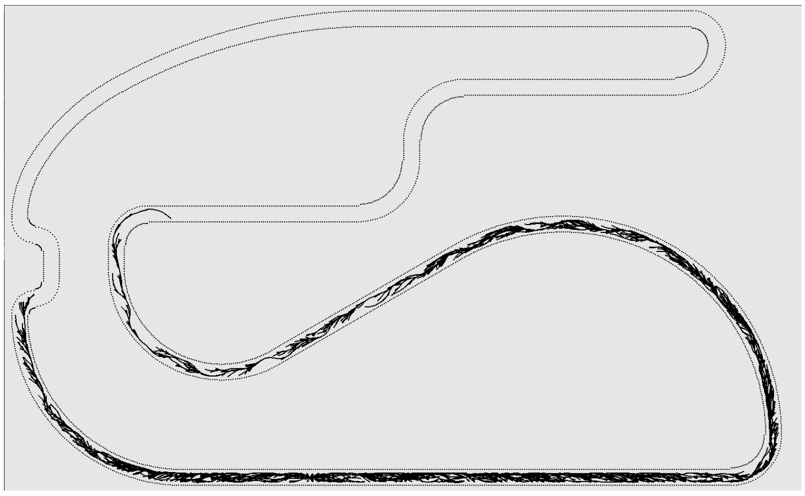


Figure 4.4: Although having the same step size and number of nodes has the tree in 4.3, it only covers around half the track and takes much longer to build, due to it having an angle limitation of 165.

copying it. This information is sent to the control module, making the car follow this new route. This can be seen in Fig.3.7, and the pseudo-code in 4.1.

If off-race is selected, the tree expansion is first completed and then the path is adapted. This process is very similar to the previous one. The pseudo-code is shown in 4.2.

Algorithm 4.1: On-race Path Adaptation

```
begin
    if overtakee detected and tree has not started then
        $startIndex \longleftarrow overtakeeIndex - offset$
        $startSegment \longleftarrow Optimal\_Path\_Segment[startIndex]$
        $x_{new} \longleftarrow startSegment$
        $T \longleftarrow INSERT\_NODE(x_{new})$
        $goalIndex \longleftarrow overtakeeIndex + offset$
        $goalSegment \longleftarrow Optimal\_Path\_Segment[goalIndex]$
    if tree started and goal not reached then
        $EXPAND\_TREE()$
        $distToGoal \longleftarrow EUCL\_DIST(lastTreeNode, goalSegment)$
        if $distToG < constant$ then
            goal reached
            adjust path
    if adjust path then
        Find the closest tree node to $goalSegment$
        $BACKTRACK()$
        $PATH\_ADAPTATION$
```

Algorithm 4.2: Off-race Path Adaptation

```
begin
    if overtakee detected and tree has not started then
        $startIndex \longleftarrow overtakeeIndex - offset$
        $startSegment \longleftarrow Optimal\_Path\_Segment[startIndex]$
        $goalIndex \longleftarrow overtakeeIndex + offset$
        $goalSegment \longleftarrow Optimal\_Path\_Segment[goalIndex]$
        adjust path
    if adjust path then
        Find the closest tree node to $goalSegment$
        repeat
            $BACKTRACK()$
            $distToStart \longleftarrow EUCL\_DIST(lastPathNode, startSegment)$
        until $distToStart < constant$
        $PATH\_ADAPTATION$
```

5 Testing & Evaluation

Contents

This sections describes and discusses the tests made to study the technique speed, efficiency and quality of results. The tests were made on a modified Toshiba SATELLITE L850-16Q with an Intel® Core™ i5-3210M @ 2.5GHz, AMD Radeon™ HD 7670M with 2GB VRAM and 8GB RAM running Lubuntu 18.10. It was programmed in C++ using Microsoft Visual Code and compiled with GNU Make 4.2.1. This setup was chosen due to TORCS having compatibility issues with the latest versions of Microsoft Visual Studio (as per the date of this document).

Some proof-of-concept tests are presented first, aimed to confirm some expectations about the algorithm performance. Then, taking into account the work's goals present in section 1.1, these tests will try to assert if the algorithm is fast enough to trace a trajectory, if its light enough to be run while on-race, and if the trajectory traced by it can be followed to overtake an opponent. Following a requirement order, the tests will check if it is possible to:

1. Build the tree while racing, with no restrictions applied;
2. Do the same but limiting the expansion region to the inside of the track, with a safety margin;
3. Cover the entire track with a high enough K;
4. Adapt the path with a previously built tree and a built-on-race tree;
5. Follow this path (no angle limitation introduced);
6. Apply an angle limitation without slowing the algorithm too much;
7. Smooth the path with said limitation, and improve the followed trajectory;
8. Avoid a collision with a static opponent;
9. Overtake the opponent, now while both are moving;

The impact that tree expansion parameters and some optimisations had on execution/task time will also be evaluated. The quality of the solution will be discussed along with the tests.

Since the algorithm has some parameters that, although possible to be automated, at the moment are manually inserted to best fit a testing situation, only a track is tested, the *"E-Track 1"*. Despite this, it can theoretically work in any track, since none of them present limitations that could possibly hinder its working. This track is 15 (fifteen) meters wide and has a long straight section that eases the trajectory tracing. Its layout can be seen in Fig.5.1.

5.1 Proof of concept testing

In this subsection the initial tests that meant to show that the algorithm was a valid choice are analysed. These will answer the first five points previously enumerated.

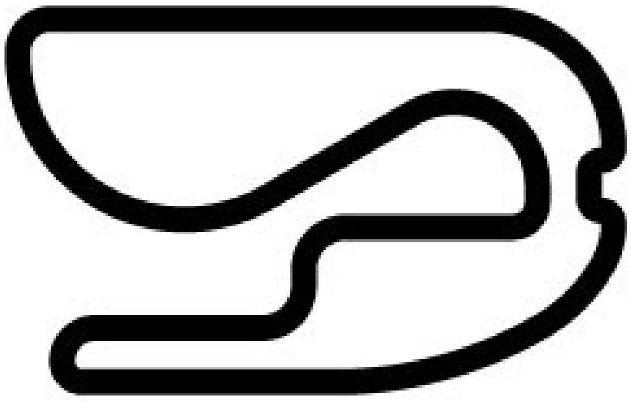


Figure 5.1: ”E-Track 1” track layout

As stated in section 2.4, for this algorithm to be viable a state, it cannot slow the frame update so it takes more than $\approx$ 0.0417 seconds (41.7 milliseconds), maintaining it at at least 24 FPS. To assert this, the average time it took to expand the tree by one node while the car was racing was measured. Different step sizes, the distance between the nearest already connected nodes and the new nodes, were compared. The values 1, 6 and 12 will be common, since they cover the range between a close-to-excessive number of nodes per path, to a close to the limit distance the car can go with no target. The average time increase can be seen in Fig.5.2. The test showed that a tree expansion with no limit can be made while racing. Please note the time measured is the average time it takes to add nodes to the tree, and this only a portion of the complete car cycle, meaning that the limit for an expansion must be (estimated after some tests and frame rate observation) $\leq$ 11 milliseconds. The cycle time is variable due to ever-changing conditions, so the upper bound must include a safety margin.

Seeing that the time it took to update was acceptable, even with a high number of nodes, the first expansion restriction - track limits - was introduced to the test, and the results measured again, shown in Fig. 5.3.

Due to only allowing states inside the track, generation times increases. Also due to the same fact, it becomes harder to expand with increased step sizes, explaining the higher average times between the three tests. Although taking close to 8 milliseconds, to complete the expansions, it still does not impact the performance enough so that the frame rate drops below the acceptable limit.

For the pre-generated tree, we compared different trees with the same number of nodes but different step sizes, aiming to find a good combination, and a result to later compare and assert the angle limit impact on tree reach.

Since each expansion has more reach, it is shown in Fig.5.4 that, has expected, with an increased step size comes a higher coverage rate. For the pre-built tree, generation time is not an issue (it is not executed during play time), so the number of nodes generated can be high, with a short step size; but

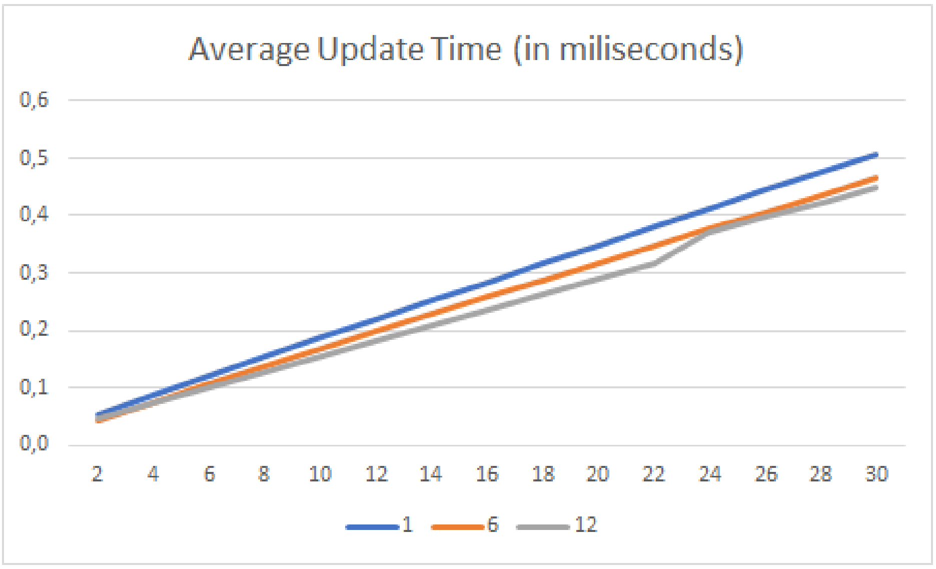


Figure 5.2: Chart comparing mean update times between different step sizes. In Y axis, the average time a tree expansion takes is in milliseconds. The X axis represent the tree size in thousands.

the path adaptation depend heavily on these parameters, so they can only be asserted after this final process test. For the on-racing generation, the chart in 5.3 shows that the step size has an impact, with values of above 8 being avoided, expected to take a longer time to update than step size 6 with no considerable gain.

This final preliminary test will confirm if the car can follow a path, with only the restrictions already considered. Different step sizes will be tested, with both modes compared. The test situation will be the following: a new race will be started with the car that implements this algorithm starting in first, with a player-controlled car starting in second. This second car will remain immobile, acting has a static obstacle. The first car will have to leave its optimal path and trace a trajectory around it. The average speed will be measured, with a higher value being favoured, and each step size will be tested three times. The chart is present in Fig.5.5.

With this chart we can conclude that adapting the trajectory with an on-race is possible and the step size matters. Too small (Step 1) and the car will have to break too much due too constant small changes. Too large (Step 20) and the target will not be updated enough, resulting in a collision. Three different trajectories can be seen in Fig. 5.6.

For the pre-built counterpart, a higher stability was verified. High speeds were achieved with step sizes from 0.5 to 15. The same issue was encountered with a step size larger than 20: the nodes were too far apart, not allowing an effective update, resulting in a collision.

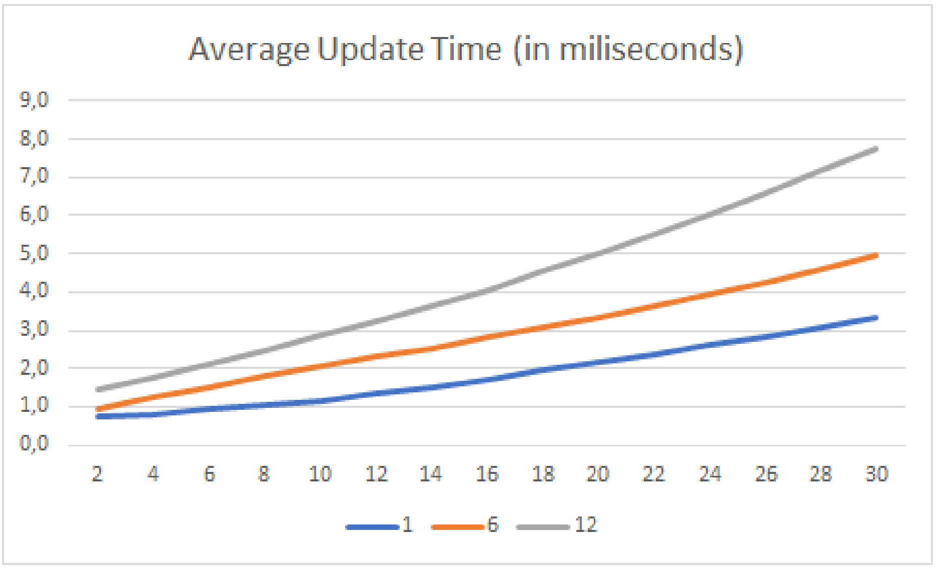
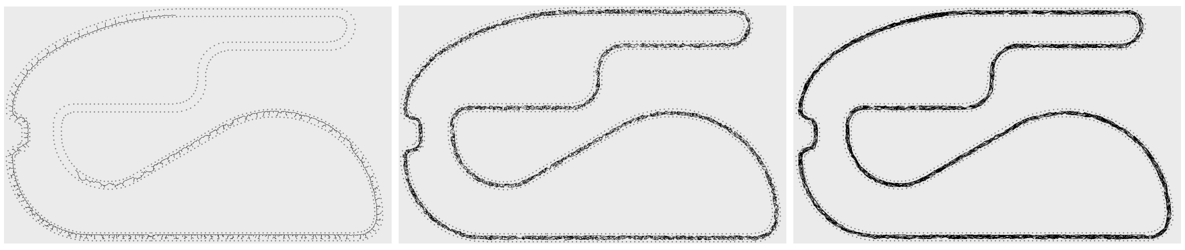


Figure 5.3: Chart comparing mean update times, with track limits. In Y axis, the average time a tree expansion takes is in milliseconds. The X axis represent the tree size in thousands



(a) Step size 1 **(b)** Step size 6 **(c)** Step size 12

Figure 5.4: Step size comparison, all trees having 5000 nodes, with no angle limitation.

5.2 Tests

We first tested if the angle limitation was really needed. The previous tests show that for short paths (100 segments), path jaggedness was not an issue. So we increased the path size to 200 segments and retested the same situation. In pre-race, although a slight decrease in average speed, the car was still able to follow the complete trajectory. In on-race, the path took longer too long too build with step sizes shorter than 6, and the car went further than the start segment, before the path could be adapted. But due to the fact it finished adaptation before it collided, the car was still able to avoid the opponent by following the new trajectory midway.

So, instead of increasing the path size again, it got reduced to 100 segments, and the obstacle was moved to a different location - the apex of the first corner. It blocks the optimal path and the trajectory the algorithm will need to trace will differ from the close-to-straight lines outputted until now. We proceeded

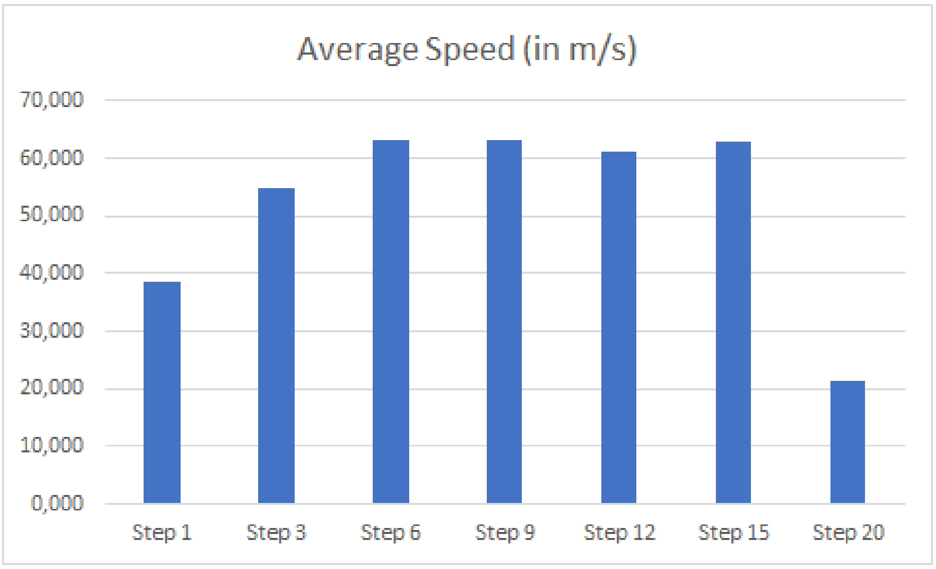
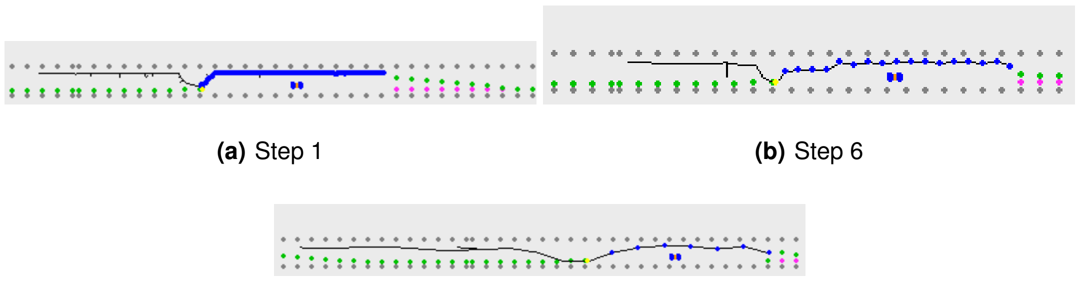


Figure 5.5: Chart comparing average speed while following the adapted trajectory. For each step size 3 runs were made with their average speed crossing the adapted path being recorded. Collisions still counted, with them heavily reducing the average speed, consequently the average of averages (the values presented in this chart).



(a) Step 1

(b) Step 6

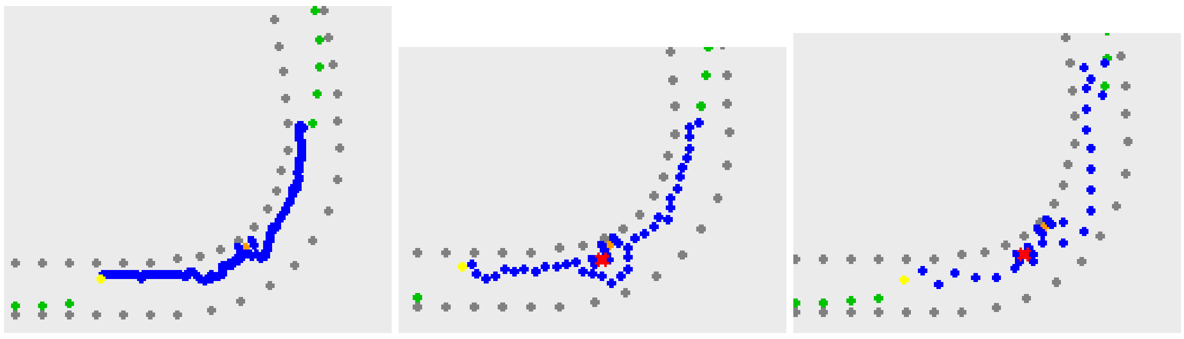
(c) Step 15

Figure 5.6: Step size effect in on-race trajectory tracing. Only initial restrictions applied.

to test this new, increased difficulty situation. The tests started with the pre-race generation with no angle limitation and increasing step sizes. Several tries were made with each step size. Some trajectories can be seen in Fig.5.7.

As expected, due to the path adaptation ignoring opponent location, trajectory completion was not guaranteed, and the car failed to overtake its opponent several times. As seen in Fig.5.7(c), the trajectory jaggedness and nodes location made the car stop. Although applying an angle limitation of 165 degrees smoothed the path, as shown in Fig. 5.8 it was still too unreliable.

The pre-race method displayed excellent results in terms of performance, with negligible performance impact. It built the tree and adapted a trajectory with it that the car could follow. But unfortunately, due to it lacking updated opponent information, it needs to be improved in order to be considered for future



(a) Step size 1 **(b)** Step size 3 **(c)** Step size 6

Figure 5.7: Trajectories adapted with different step sizes and no angle limitation. The player car is represented with a yellow dot, and this car with a red dot.

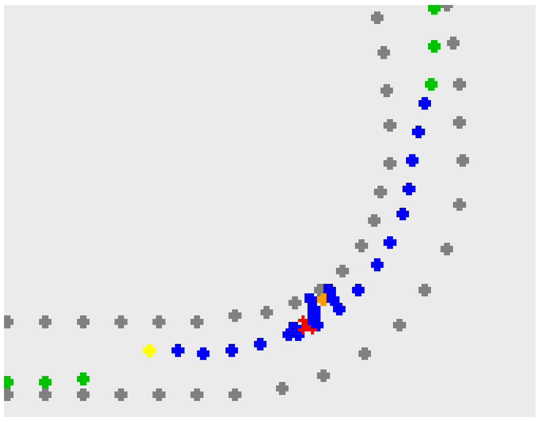


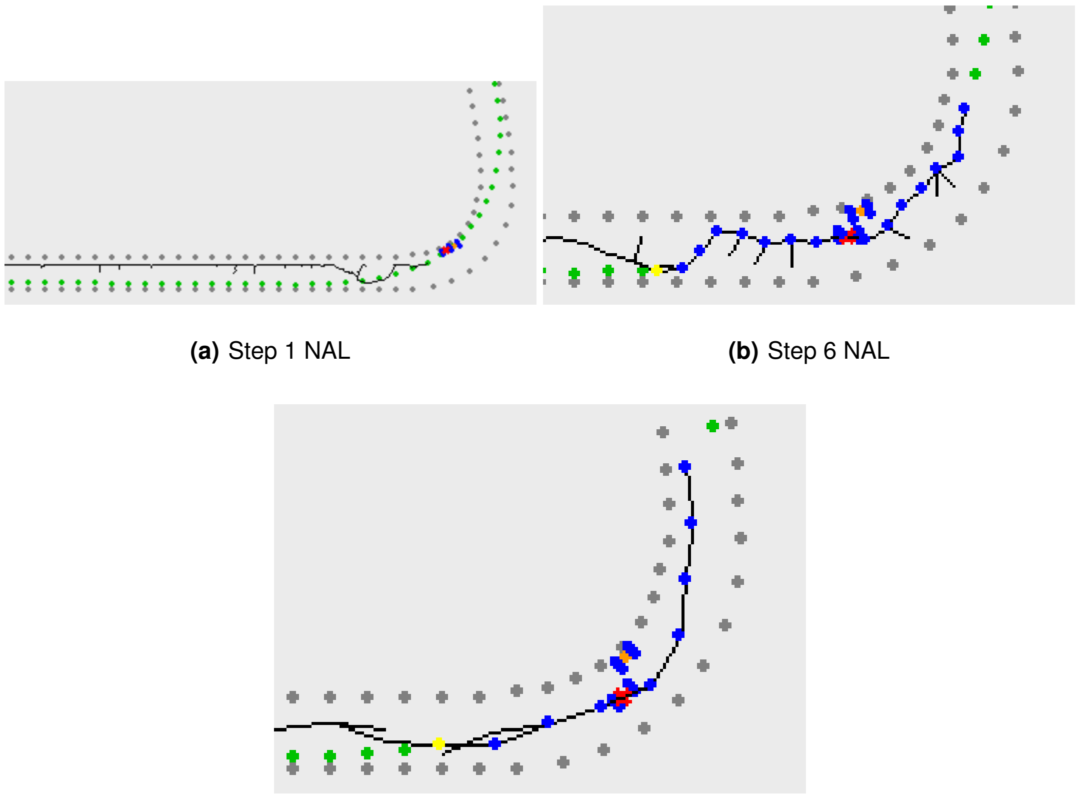
Figure 5.8: Adapted path in a turn. Although smooth, the opponent blocks this path, meaning it cannot be followed successfully.

testing. With these results it was predicted that it would not be possible to overtake a dynamic opponent with this technique.

Then the on-race method was tested. As stated in the beginning of this section, it avoided a collision with the static opponent, even with a longer trajectory. So it was tested in the same scenario as the pre-race method. NAL in the captions means no angle limit applied.

As shown in Fig. 5.9(a), due to no existing bias, the tree did not finish building before the car collided with the opponent. The tree would only finish in some tests, this being too unreliable to be considered. In Fig. 5.9(b), it did, but due to path jaggedness, it had to brake too much on turn entrance and could not complete the trajectory (due to it being a slight hill). In Fig. 5.9(c) the car was able to complete the trajectory, but only sometimes. In the situation shown, the velocity was not adequate and it almost left track.

We then tested with a 160 angle limitation. Higher degree limitations were also tested, but due to low frame rates on tree building, were considered invalid and ignored. The trajectories are represented


(a) Step 1 NAL **(b)** Step 6 NAL

(c) Step 12 NAL

Figure 5.9: Step size effect in on-race trajectory tracing. Only initial restrictions applied.

in Fig. 5.10.

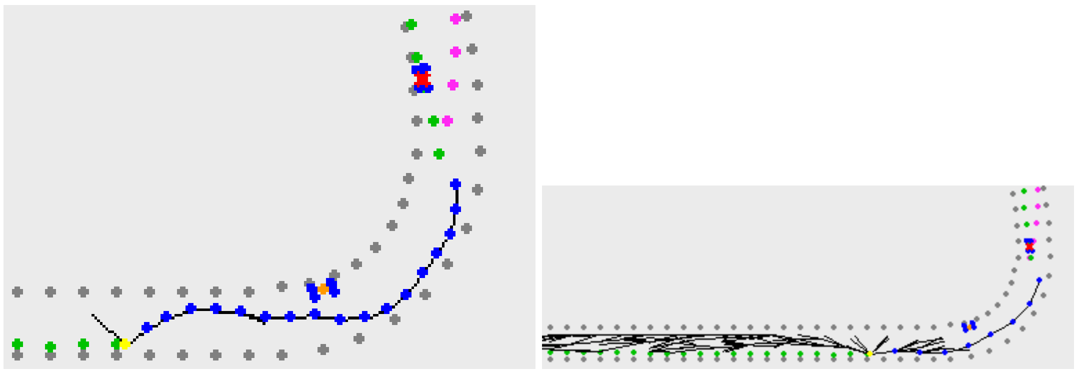
This resulted in smoother paths that were more reliably followed, still with small performance impact (frame rate drops were noticeable, but only for a very short moment). Notice that in 5.10(b), the lack of bias allowed for a tree expansion in the opposite direction of the needed one, meaning a high percentage of nodes that would not be used were created.

Finally, the primary task was tested : overtaking an opponent. Unfortunately, even with parameter optimisation, the robot was not able to overtake a moving opponent. The reasons will be discussed in the final section of this chapter.

5.3 Discussion

In chapter chapter 1 two questions were asked. In this chapter they were tested and in this section they will be answered according to the test results.

The first question, an objective regarding the quality of the solution, was divided into two objectives.



(a) Step 6, angle limitation of 160 **(b)** Step 12, angle limitation of 160

Figure 5.10: Step size and angle limitation effect in on-race trajectory.

The first objective was to avoid a collision with a static object. The test results show that, as long the parameters take reasonable values, with step sizes between 1 and 12, and the path adaptation process considers the position of the obstacle it has to avoid, it could complete said objective. But this could only be made with a on-race tree; an off-race tree was not stable enough to be considered in future tests.

The second objective in this question, the main task of this work, was to overtake a moving opponent, which unfortunately was not possible. The main causes observed are:

- The overtaking detection occurred only once, with the desired trajectory start and goal not changing until the path was completely traced by the vehicle. This meant that, in a race, were the overtakee position constantly changes, this is not a viable practice. By the time the robot followed the trajectory, its opponent was already ahead of the goal.
- The robot was able to follow only some trajectories, with many it being either too slow to complete them, too fast with it ending off track, or too jagged meaning the average speed was too low to be considered competitive.
- There was no path re-adaptation while the first adapted path was being followed, meaning the robot could be easily blocked if the overtakee stood in the traced path.

Some possible solutions are discussed in the last chapter.

The algorithm performance impact was also queried, this time showing promising results. With a light collision detection technique, and even with an angle limitation that smoothed the path, the on-race tree was built without affecting player gameplay. Some improvements can be made to avoid wasted resources on nodes built in the wrong direction, presented in the next chapter. As predicted, the off-race tree expansion had no impact in race performance, neither its path adaptation process, although this last one needing further improvement.

6 Conclusion

Contents

The goal for this work was to explore RRT and its uses, and try to execute an overtaking manoeuvre with it. ADOVER-RRT was designed, an algorithm that, implemented in the base robot present in the game TORCS *inferno*, hopefully could achieve this task. Unfortunately it was not able to overtake a moving opponent. But its performance was promising, with the game being playable while this robot was running. It is a flexible and light algorithm, able to be run off and on-race, with some flaws that need to be improved.

Analysing its implementation and test results, the main problem of this technique is its incompatibility with its task environment: a racing situation, with highly dynamic opponents and a track with a diversity of segments, needs an equally dynamic algorithm. Robot position and speed, target status, opponent status, next opponent to overtake, when to overtake, start and goal of the overtaking manoeuvre, are all variables that are constantly changing while the robot is racing, so the number of constants that are considered while the tree is being built and the path adapted need to, preferably, be non existent. But is also important to say that this algorithm approach contributed with an interesting idea, that in our opinion, should be explored. The combination of the tree building process with a preceding optimal path tracing, in this works case thanks to K1999 algorithm, meant that the robot built a new path only when needed, saving a great amount of processing power, and achieving good results when racing alone. It was proven to be highly beneficial to start the race with an already built optimal path, that although not always followed, still encoded important information.

We predict that, with the improvements presented in the next section, this algorithm can complete the objective that was initially given.

6.1 Notes for Future Work

Considering the main flaw pointed in the previous section, the fact that this algorithm uses information that should not be constant throughout its process, some possible solutions are presented that mainly tackle this issue, including others.

With the off-race method, its main flaw was clearly not adapting to the situation the car was currently facing. This can be solved by continuously updating and using the opponent position in the path adaptation process, ignoring the path if blocked, possibly retracing it, or simply searching for another available branch.

The on-race method needed its generation speed increased. The angle limitation took a too heavy toll on its performance, and without bias, the tree was not able to connect the desired start and goal before the robot collided with its opponent. Therefore, an expansion bias could be implemented to increase its speed, with this also reducing wasted resources. Considering the path adaptation method, the start and goal that it detects need to better fit the dynamic environment. A better prediction on when the opponent will be completely overtook needs to be made to find a better goal. It also needs to be changed and the path retraced if needed.

ADOVER-RRT could also improve if, after some overhead tests, improve the base RRT section to include the RRT* improvements. Paths would be smoother and easier to follow. Path smoothness can also be achieved with bezier curves or clothoids, but further overhead tests need to be made, considering that the tree expansion would need more calculations.

Finally, a more advanced car control module could be implemented. The current model present in *inferno*, is easy to work with and flexible, but it does not consider behaviours that would give him a competitive edge, for example, engine breaking.

Bibliography

[1] "Global championship of driverless cars." [Online]. Available: https://roborace.com/

[2] "Studio-397 – Racing Simulation." [Online]. Available: https://www.studio-397.com/

[3] "Assetto Corsa your racing simulator." [Online]. Available: https://www.assettocorsa.net/home-ac/

[4] "RaceRoom Racing Experience." [Online]. Available: http://game.raceroom.com/

[5] "Project CARS 2 - The Cars." [Online]. Available: https://www.projectcarsgame.com/the-cars/

[6] "gran-turismo.com." [Online]. Available: https://www.gran-turismo.com/us/products/gtsport/

[7] "F1 2018." [Online]. Available: http://www.codemasters.com/game/f1-2018/

[8] R. Coulom, "Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur," p. 169.

[9] R. Vesel, "Race Optimal." [Online]. Available: https://www.raceoptimal.com

[10] S. M. Lavalle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," Tech. Rep., 1998.

[11] S. Karaman and E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning," *arXiv:1105.1186 [cs]*, May 2011, arXiv: 1105.1186. [Online]. Available: http://arxiv.org/abs/1105.1186

[12] F. Islam, J. Nasir, U. Malik, Y. Ayaz, and O. Hasan, "Smart: Rapid convergence implementation of RRTtowards optimal solution," in *2012 IEEE International Conference on Mechatronics and Automation*. Chengdu, China: IEEE, Aug. 2012, pp. 1651–1656. [Online]. Available: http://ieeexplore.ieee.org/document/6284384/

[13] "geometry - Car racing: How to calculate the radius of the racing line through a turn of varying length." [Online]. Available: https://math.stackexchange.com/questions/289575/car-racing-how-to-calculate-the-radius-of-the-racing-line-through-a-turn-of-var

[14] "Home." [Online]. Available: https://drivingfast.net/

[15] “What’s the difference between RRT and RRT* and which one should we use.” [Online]. Available: https://www.youtube.com/watch?v=JeEk_CWcRFI

[16] S. Gomes, J. Dias, and C. Martinho, “Iterative Parallel Sampling RRT for Racing Car Simulation,” in *Progress in Artificial Intelligence*, ser. Lecture Notes in Computer Science, E. Oliveira, J. Gama, Z. Vale, and H. Lopes Cardoso, Eds. Springer International Publishing, 2017, pp. 111–122.

[17] J. E. Naranjo, C. Gonzalez, R. Garcia, and T. d. Pedro, “Lane-Change Fuzzy Control in Autonomous Vehicles for the Overtaking Maneuver,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 9, no. 3, pp. 438–450, Sep. 2008.

[18] “TORCS - The Open Racing Car Simulator - Browse /api-docs/1.3.7 at SourceForge.net.” [Online]. Available: https://sourceforge.net/projects/torcs/files/api-docs/1.3.7/

[19] “Easy Tips for Better AI in rFactor 2 / rFactor 1 / Most Any Racing Game Ever.” [Online]. Available: https://www.youtube.com/watch?v=3X34Y3s6pSQ

[20] J. Togelius and S. Lucas, “Evolving robust and specialized car racing skills,” in *2006 IEEE International Conference on Evolutionary Computation*. Vancouver, BC, Canada: IEEE, 2006, pp. 1187–1194. [Online]. Available: http://ieeexplore.ieee.org/document/1688444/

[21] “The official home of Formula 1® | F1.com.” [Online]. Available: https://www.formula1.com/

[22] L. Lanzi, I. L. Cardamone, I. D. Loiacono, A. Pietro, B. Matr, A. Pietro, and B. Introduzione, *Automatic Racing Lines Generation For High-End Car Games*.

[23] “Banked turns.” [Online]. Available: http://dynref.engr.illinois.edu/avb.html

[24] J. Bialkowski, S. Karaman, and E. Frazzoli, “Massively parallelizing the RRT and the RRT,” in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2011, pp. 3513–3518.

[25] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, “Sampling-based roadmap of trees for parallel motion planning,” *IEEE Transactions on Robotics*, vol. 21, no. 4, pp. 597–608, Aug. 2005.

[26] M. Claypool, K. Claypool, and F. Damaa, “The effects of frame rate and resolution on users playing first person shooter games,” S. Chandra and C. Griwodz, Eds., San Jose, CA, Jan. 2006, p. 607101. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.648609

[27] S. M. LaValle and J. J. Kuffner, “Randomized Kinodynamic Planning,” *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001. [Online]. Available: https://doi.org/10.1177/02783640122067453