Application of RRT for overtaking in a Racing Car Simulation

Guilherme Gomes
jguilhermefgomes@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 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: ADOVER-RRT, RRT, Overtaking, K1999

1. Introduction

Nowadays, AI has an increasingly important role in motor racing. "Robots" ("bots" for short) are an important part racing games. 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 [4], 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 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 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.

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 two main aspects:(1) 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? (2) 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 TORCS [1]. 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.

2. Proposed solution

This section details the processes that work together to accomplish this study objective. These algorithms rely on the available track information, shown in the official TORCS APIDOC [1]. 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.

2.1. K1999

Considering the K1999 description present in [2] and track description in TORCS APIDOC [1], 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.2. A diagram that shows this algorithm cycle can be seen in Fig.3. The difference between the two traced paths can be seen in Fig.1.

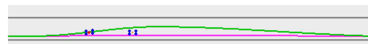

Figure 1: The blue dots represent the cars corners, the dynamic path is represented in green and the static, optimal path in pink. The agent using the K1999 retraces its current path and redraws it around the opponent it finds withing look-ahead distance.

Figure 2: Both lines (dynamic in green and static in pink) traced by K1999 at the start of the race.

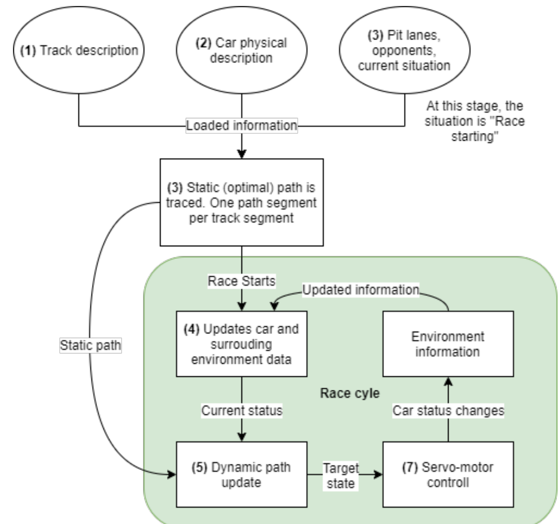

Figure 3: Scheme with the K1999 cycle, as implemented in TORCS. The states are numbered according to the steps enumeration presented at the start of the section.

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

2.2.1 Description

We want to compare on-race and pre-race situations. On-race refers to the tree being built while the car is racing, meaning the expansion process will be part of the car update cycle. Pre-built refers 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. The time it takes can be ignored.

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 1:

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

```
begin
    T.init(x_init);
    for k = 1 to K do
        angle = 0;
        if (!ON_RACE) or (ON_RACE and FRAME%FREQ == 0) then
            repeat
                x_rand = RANDOM_STATE();
                x_near = NEAREST_NEIGHBOUR();
                x_step = STEP();
                if x_step ∄ X_free then
                    continue;
                end
                if x_near.parent ≠ null then
                    angle = BRANCH_ANGLE();
                    if angle ≤ angleLimit then
                        continue
                    end
                end
                T = INSERT_NODE(T, x_near, x_step);
                A state was added;
            until state added;
        end
    end
    return T
end
```

Algorithm 1: ADOVER-RRT

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 5. 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:

$$\arccos\left((\overline{NS}^2 + \overline{NP}^2 + \overline{PS}^2)/(2 * \overline{NS}^2 * \overline{NP}^2)\right) \quad (1)$$

The effect is shown in 4 and its necessity to this project in chapter **??**. 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. 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}.

2.3. 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. 4.

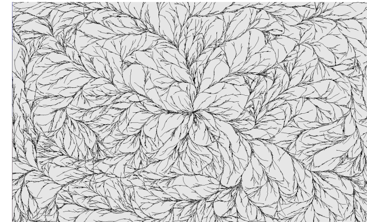

Figure 4: With angle limited to 160 degrees

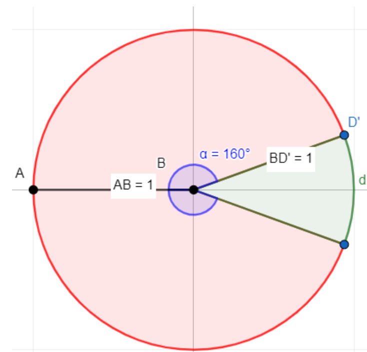

Figure 5: Considering B being x_{near} with parent A, D (x_{step}) can only be extended inside d arch.

3. 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 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 by moving the optimal and dynamic path segments to their respective closest copied vertices. 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.6.

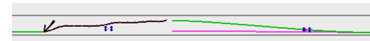

Figure 6: RRT path retracing, with the new trajectory in black, with nodes in red. Notice how the K1999 then creates a return path to the optimal trajectory, only at the end of the retraced line.

4. Methodology

This chapter details the implementation of the developed robot using the ADOVER-RRT and K1999.

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. This architecture is represented in the scheme shown in Fig.7. It also applies to both of the proposed on-race and pre-race modes, since their changes are reflected inside the **RRT** sub-module.

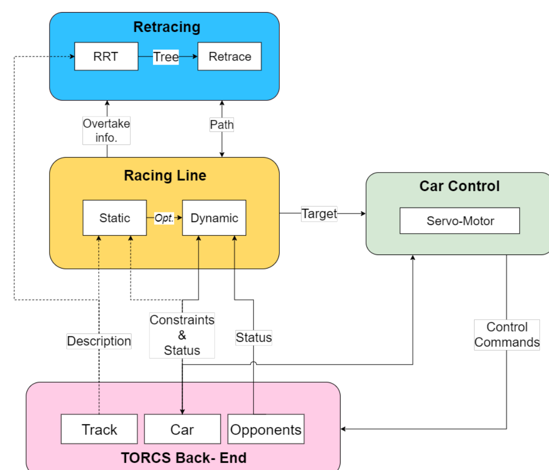

Figure 7: Solid lines represent continuous transmission, with dashed lines being information sent once, when the sub module starts.

4.2. 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. 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. The Opponents sub-module is restricted to showing information about the current adversaries in race.

4.3. Racing Line

It traces a static and an initial dynamic path, built as detailed in chapter 2. In race, it continuously communicates with both the back-end and control module, updating 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 when possible, as the second one is considered the optimal. It has a structure that uses the data it gets from the Opponent sub module filled with 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 it can hold other search algorithms or racing line calculation techniques that perform the same function. An example is the architecture used in [3], where the planning is made by the IPS-RRT algorithm.

4.4. Retracing

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.6. If off-race, it only retraces when needed, using the vertices already created.

4.4.1 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. Acceleration command $car._accelCmd$ will be at 1 when possible. $car._brakeCmd$, has its value regulated by an ABS system, preventing the wheels from blocking and counteracting tire slippage, and it will use tyre trackiont and path segment top speed.

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

$$trgAngle = \arctan(trg.y - car.y, trg.x - car.x); \tag{2}$$

$$trgAngle = car.yaw; \tag{3}$$

$$trgAngle = trgtAngle \in [-\pi, \pi]; \tag{4}$$

$$steer = trgAngle/car.steerLock; \tag{5}$$

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.

4.5. Search State Representation

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.

4.6. ADOVER-RRT applied to TORCS

This is the algorithm that has the function of retracing the path initially calculated by the *Racing Line* module. Considering its previously described workings, it will be explained within TORCS context. 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.8. 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.8 are negligible. These track margins can be visualised in Fig.9.

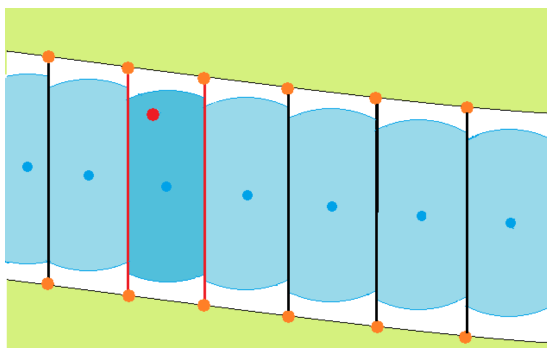

Figure 8: The red dot is x_{rand}. The closest segment to it is outlined in red, with each segment mid position marked by a blue dot. Since this position is close enough to the middle of its segment, it sits in a valid position.

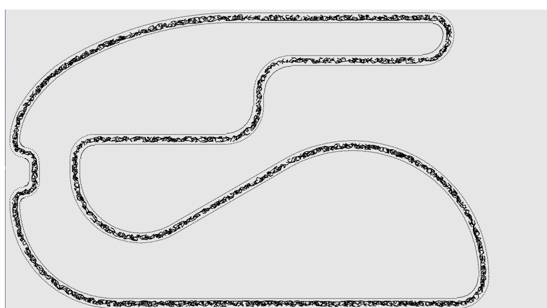

Figure 9: Tree built pre-race, with no angle limitation and 10000 nodes, and step size 6. Notice how no state is outside of the track and it within track safe margins.

The second condition the state must verify is the angle limitation introduced in chapter 2 and visualised in Fig.5. This limits the new branch angle, and results in smoother tree, and consequently smoother paths. Taking in consideration what can be seen in Fig.9, we can compare it to Fig.10 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

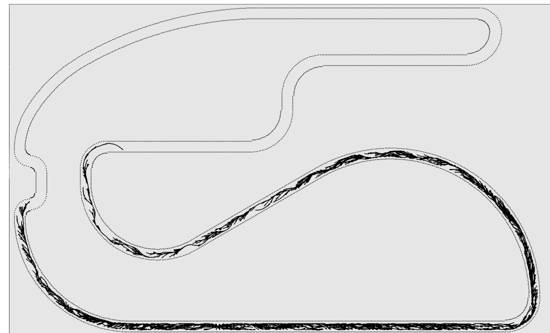

Figure 10: Although having the same step size and number of nodes has the tree in 9, it only covers around half the track and takes much longer to build, due to it having an angle limitation of 165.

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 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 predetermined 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. 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. 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 el-

ements 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, copying it. This information is sent to the control module, making the car follow this new route. This can be seen in Fig.6, and the pseudo-code in 2.

```
begin
  if overtakee detected and tree has not started then
    startIndex = overtakeeIndex − offset;
    startSegment = Optimal_Path_Segment[startIndex];

    x_new = startSegment;
    T = INSERT_NODE(x_new);
    goalIndex = overtakeeIndex + offset;
    goalSegment = Optimal_Path_Segment[goalIndex];
  end
  if tree started and goal not reached then
    EXPAND_TREE();
    distToGoal = EUCL_DIST(lastTreeNode, goalSegment);

    if distToG < constant then
      goal reached;
      adjust path;
    end
  end
  if adjust path then
    Find the closest tree node to goalSegment;
    BACKTRACK();
    PATH_ADAPTATION;
  end
end
```

Algorithm 2: On-race Path Adaptation

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 3.

```
begin
  if overtakee detected and tree has not started then
    startIndex = overtakeeIndex − offset
    startSegment = Optimal_Path_Segment[startIndex]
    goalIndex = overtakeeIndex+offset
    goalSegment = Optimal_Path_Segment[goalIndex]
    adjust path
  end
  if adjust path then
    Find the closest tree node to goalSegment
    repeat
      BACKTRACK()
      distToStart = EUCL_DIST(lastPathNode, startSegment)

    until distToStart < constant
    PATH_ADAPTATION
  end
end
```

Algorithm 3: Off-race Path Adaptation

5. Results & discussion

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, 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 test-

ing 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.

5.1. Proof of concept testing

As stated in section **??**, for this algorithm to be viable a state, it cannot slow the frame update so it takes more than ≈ 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.11. The test showed that a tree expansion with no limit can be made while racing. 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) ≤ 11 milliseconds.

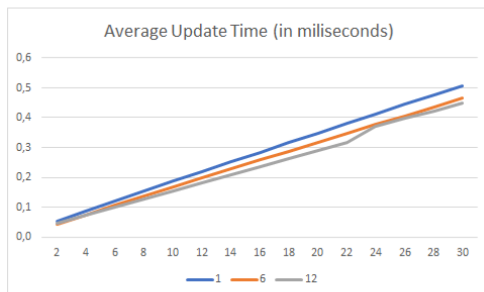

Figure 11: Chart comparing mean update times between different step sizes. In Y axis, the average time a tree expansion takes is in milliseconds. The X axis represent the tree size in thousands.

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. 12.

Due to only allowing states inside the track, generation times increases. 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

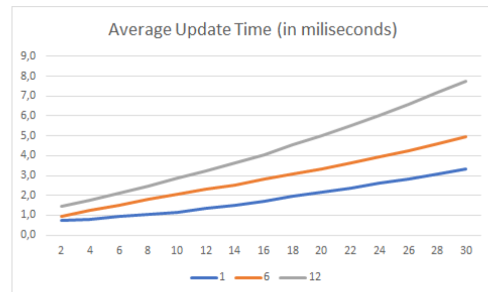

Figure 12: Chart comparing mean update times, with track limits. In Y axis, the average time a tree expansion takes is in milliseconds. The X axis represent the tree size in thousands

combination, and a result to later compare and assert the angle limit impact on tree reach. Since each expansion has more reach, 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 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 12 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. 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.13.

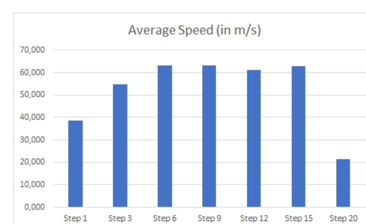

Figure 13: Chart comparing average speed while following the adapted trajectory. For each step size 3 runs were made with their average speed crossing the adapted path being recorded. Collisions still counted, with them heavily reducing the average speed, consequently the average of averages (the values presented in this chart).

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. 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.

5.2. Tests

We first tested if the angle limitation was really needed. We increased the path size to 200 segments and retested the same situation as before. 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.The path then got again reduced to 100 segments, and the obstacle was moved to 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 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. A trajectory can be seen in Fig.14.

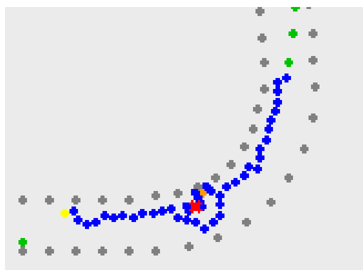

Figure 14: Step size 3

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. The trajectory jaggedness and nodes location made the car stop sometimes. Although applying an angle limitation of 165 degrees smoothed the path 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 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. 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. With step size 6 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). With step size 12 the car was able to complete the trajectory, but only sometimes. 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. 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). The lack of bias still meant some nodes were built in the wrong direction, representing wasted resources. 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 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. 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, was to overtake a moving opponent, which unfortunately was not possible. 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 compet-

itive. 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. The off-race tree expansion had no impact in race performance.

6. Conclusions

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.

References

[1] TORCS - The Open Racing Car Simulator - Browse /api-docs/1.3.7 at SourceForge.net.

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

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

[4] 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*, 9(3):438–450, Sept. 2008.