POLITECNICO DI MILANO
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Open Loop Planning for Formula 1 Race Strategy identification

Supervisor: Prof. Marcello Restelli
Co-supervisor: Amarildo Likmeta

M.Sc. Thesis by:
Diego Piccinotti, mat. 922744

Academic Year 2019-2020

To my parents, who encouraged me to follow my passion
and allowed me to get this far. Thank you.

Contents

List of Figures

List of Tables

Abstract

Formula 1 (F1) is one of the most competitive categories of motorsport racing, in which single-seater, high-performance cars compete around a closed circuit. In F1 events, participating race cars have to complete a defined number of laps around a closed circuit. The goal for a driver is to finish each race in the best possible placement consistently.
In addition to driver skills and car performance, the result in F1 competitions is often also determined by the tire strategy adopted by the team. For this reason, Formula 1 teams invest considerable resources in race outcome simulation and prediction to supply their race engineers with fast predictions of most likely events.
In this work, we aim to provide an automated way of identifying tire strategies for F1 races by considering the problem of deciding when to perform a pit-stop and which compound to use as a sequential decision-making problem. In order to provide recommendations in a reasonable time, we, therefore, investigate the application of anytime online planning algorithms to tackle this problem.
To cope with the stochastic and continuous nature of the problem, we propose an agent based on an open-loop approach combining both Monte Carlo sampling and a Temporal Difference (TD) backup operator.
To evaluate the proposed approach, we design a planning environment able to provide a replica of past F1 races, which we base on a lap time simulator. To this end, we discuss the feasibility of designing and implementing a regression-based lap time simulator, using publicly available data of past races.
Finally, exploiting the availability of a fairly complete racing simulator [13], which we modify to be consistent with a planning application, we conduct a thorough evaluation of the proposed planner, as well as various anytime planners from the MCTS literature on a sample of races belonging to the "Turbo Hybrid era" of F1.

Estratto in lingua italiana

La Formula 1 (F1) è una delle categorie più competitive nel motorsport, in cui veicoli monoposto ad altissima performance competono su circuiti chiusi. Durante gli eventi di F1, le monoposto devono completare un ammontare di giri attorno ad un circuito nel minor tempo possibile. Alla conclusione dell'evento, l'ordine di arrivo viene utilizzato per assegnare punti a ciascuno dei piloti classificati nelle prime dieci posizioni. Pertanto, l'obiettivo per ogni pilota è quello di concludere la gara con il miglior piazzamento possibile.

Il risultato delle gare di F1 non è determinato solo dall'abilità dei piloti e dalla performance delle monoposto, ma anche dalla strategia di gara che viene impiegata dalla squadra. La strategia di gara è costituita dall'ordine delle mescole che vengono montate su una monoposto durante la gara, nonché dal giro a cui vengono montate. La sostituzione delle gomme, chiamata pit-stop, è obbligatoria e deve avvenire almeno una volta per ogni pilota durante la gara.

Montare la mescola giusta al momento giusto può permettere di guadagnare un vantaggio significativo sugli avversari o, viceversa, se viene adottata una strategia sbagliata, il tempo perso rispetto ai piloti avversari può essere considerevole. Per questo motivo, le squadre di F1 investono ingenti risorse per simulare le situazioni di gara e ottenere predizioni dei risultati più probabili, in modo da poterle fornire ai propri strateghi nel più breve tempo possibile. Gli strateghi devono spesso reagire a mosse degli avversari e situazioni di gara inaspettate, pertanto il processo di decisione può rivelarsi frenetico e la necessità di prendere decisioni in breve tempo può impedire al gruppo di strategia di considerare tutte le opzioni, o addirittura indurlo all'errore.

Considerando i fattori appena citati, crediamo che l'utilizzo di uno strumento per il supporto alle decisioni in grado di fornire suggerimenti sulla strategia di gara molto velocemente possa essere utile a migliorare la qualità e la competitività delle strategie di gara utilizzate dalle squadre. L'obiettivo del nostro lavoro è quindi quello di progettare ed implementare un agente autonomo in grado di identificare le strategie di gara per la F1, considerando il problema della decisione se effettuare un pit-stop e quali gomme montare come un problema di decisione sequenziale.

Per offrire raccomandazioni in un tempo ragionevole, concentriamo la nostra ricerca sull'applicazione di algoritmi anytime di planning online per affrontare il problema. Per gestire la natura stocastica e continua del problema, proponiamo un agente basato su un approccio open-loop che combini sia il campionamento Monte Carlo che un operatore di backup di tipo Temporal Difference. Il nostro approccio innovativo si basa sulla modifica dell'algoritmo di MCTS UCT, sfruttando l'operatore di update di Q-Learning [33] al posto dell'update Monte Carlo al fine di ridurre la varianza nella stima della funzione di valore per le coppie stato-azione, che, nel problema di identificazione delle strategie di gara, rappresenta una difficoltà significativa a causa della alta stocasticità del problema e dell'asimmetria nelle ricompense cumulative per le azioni. Inoltre, per ridurre la difficoltà nel modellare lo scenario multi-agente, consideriamo uno scenario a singolo agente in cui i piloti avversari siano controllati dall'ambiente di pianificazione e seguano strategie prefissate, ottenute da articoli di opinione sportiva.

Per effettuare una valutazione del nostro approccio utilizziamo un ambiente di planning di nostra progettazione, basato su un simulatore di tempi sul giro ed in grado di offrire una replica di gare F1 avvenute negli anni passati. Analizziamo pertanto la fattibilità di progettazione e realizzazione di un simulatore dei tempi basato su regressione effettuata su dati pubblicamente disponibili delle gare passate, osservandone le criticità.
Infine, sfruttando la disponibilità di un simulatore di gara probabilistico [13] e modificandone il comportamento per essere consistente con la nostra applicazione, conduciamo un'attenta valutazione dell'algoritmo proposto, insieme a vari altri planner anytime presenti nella letteratura in ambito MCTS, su di un campione di gare appartenenti all'"era turbo-ibrida" della F1, mostrando che il nostro algoritmo è in grado di superare nella maggior parte delle gare considerate la performance di baseline basate su strategie reali e metodi automatici.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Restelli, who has provided me with guidance and insight during all the steps of this work. I especially would like to thank him for giving me the example of a most professional, precise, and complete way of thinking and working, never leaving anything to luck.
Then, I would like to thank my co-supervisor, Amarildo, whose support has been invaluable during this work. He always guided me on this difficult path and provided me with valuable feedback, allowing me to grow both in my technical skills as well as in self-confidence and work autonomy.
I would also like to thank my mentors, Francesco and Giada, who have been following my professional and personal development closely in the last two years. Their help has been immense in making many of my professional choices, as well as becoming the person I am now, and I hope someday to be able to give back to someone else what they gave me.
It would be impossible not to thank my friends, especially Alberto, Alessandro, Francesco, and Jacopo, who have always been there for me and proved to be an invaluable company during this pandemic time. I wouldn't have made it out of this without you.
A special thank goes to all of my family. They have been extra patient with me during all of my path and supported me through my choices, even when this meant switching faculty or moving to another city. I will never forget your effort.
Last, but not least, I would like to thank Anastasia, who, despite being part of my life for a relatively short time, has already had a large impact on it. Thank you for seeing my value and believing in myself more than I would do, I am stronger when you're around.

Chapter 1

Introduction

This chapter offers an overview of what will follow during the reading of this work.

Formula 1 racing

Formula 1 (F1) is one of the most competitive categories of motorsport racing, in which single-seater, high-performance cars compete around a closed circuit. In F1 events, participating race cars have to complete a defined number of laps around a closed circuit. Points are awarded after each race to the top-ten finishing drivers, and the sum of points obtained by each driver during the races of a season defines the championship standings. Therefore, the goal for a driver is to finish each race in the best possible placement consistently.

The final placement for a driver depends both on his skill and the car's performance, as well as the tire strategy employed during the race. In F1, tire replacement (called pit-stop) must happen at least once for each participating car, and the teams choose, among a selection provided by the tire manufacturer for each race, which tire to fit next. The tire strategy is, therefore, the sequence of tires used during a race by a driver.

Motivation and goal

In Formula 1, fitting the right set of tires at the right moment can lead to significant time gains (or losses) over opposing drivers. This effect calls for strategical planning of the pit-stops by the teams in order to maximize the advantage over competitors. Formula 1 teams devote many resources to this end, having groups of people tasked with finding the best tire strategy, both prior to and during the race.

The decision-making process often becomes hectic, as, during the race, the people inside strategy groups need to react to potentially unexpected scenarios that may present during the race. The need for making decisions in a short time may prevent the strategy team from considering all the options or even lead to misjudging the situation and make strategic mistakes. Having considered these factors, we believe that using a decision support tool able to provide tire strategy suggestions in a short time during the race would improve the quality and competitiveness of strategies applied by the teams.

It is, therefore, the goal of this thesis work to devise and implement an autonomous agent able to provide suggestions on whether to perform a pit-stop and which tire compound to fit on the car.

Contributions

We investigate the feasibility of applying online planning to the specific problem of race-strategy identification and propose an innovative open-loop approach that combines Monte Carlo sampling and Temporal Difference (TD) updates to identify whether to perform a pit-stop at each race lap and which tire compound to employ. Our novel approach modifies the MCTS UCT algorithm, leveraging the Q-learning update in place of the MC one to reduce variance in the estimation of the state-action value function, which in the race strategy identification scenario represents a significant issue due to the high stochasticity of the problem and asymmetry in the return of actions. Moreover, we reduce the complexity of modeling a multi-agent planning environment by modeling the problem as a single-agent setting in which opponents are controlled by the environment and follow plausible predefined strategies extracted from sports opinion articles. Lastly, we perform an extensive evaluation of different planning algorithms using a simulator based on [13], which we modify to be consistent with a planning application.

Structure of Thesis

Chapter 2 introduces all the preliminary concepts needed to understand the content of this work. In Section 2.1 we cover an introduction on Reinforcement Learning, starting from MDP fundamentals and presenting several methods to solve MDPs. In Section 2.2, we present the fundamentals of Monte Carlo Tree Search (MCTS) and provide a taxonomy of such methods.

Chapter 3 presents the state-of-the-art planning algorithms that may be applied to our scenario. In Section 3.1 we introduce the most popular MCTS algorithm, Upper Confidence Bound for Trees (UCT), then continue with the following sections describing algorithms for tackling continuous state spaces like Double Progressive Widening, as well as Open Loop Optimistic Planning. We finally present, starting from Section 3.5, methods that exploit variations to the backup operator to overcome limits of UCT.

Chapter 4 provides a more detailed description of F1 racing and focuses on the race strategy identification problem. Section 4.1 introduces the background needed to understand Formula 1 racing and tire strategies, providing intuitions on the relevance of applying carefully designed strategies. In Section 4.2, instead, we present the formalization we adopt to describe the F1 race strategy problem and discuss the possible issues that may arise when modeling such a problem.

Chapter 5 presents the approach we used to tackle the race strategy identification problem and our main contributions. Section 5.1 presents the approaches we took to build a planning environment modeling the F1 race strategy problem. We then present, in Section 5.2, our proposed planner, also giving a quick introduction and providing notation for the open-loop setting. Finally, in Section 5.3 we discuss alternative rollout policies used to improve our planner's performance in the considered setting.

Chapter 6 presents the experimental evaluation results for our proposed planner and compares them to other planners' performance in the same setting.

Chapter 7 contains the conclusions we can draw on our work and highlights areas that can be the object of future work.

Chapter 2

Preliminaries

This chapter introduces all necessary knowledge to ease the reading of this thesis.

2.1 Reinforcement Learning

2.1.1 Introduction

As defined by Sutton and Barto in their book "Reinforcement Learning: An Introduction" [31], the Reinforcement Learning (RL) problem is *"a framing of the problem of learning from interaction to achieve a goal"*.

Following the terminology defined by the authors, we call the decision-maker the **agent** and the entity it interacts with, which is constituted by everything outside the agent, is called the **environment**. The agent's goal is to maximize over time a given function (usually the sum) of a signal received from the environment as a consequence of its actions, which is called **reward**.
Furthermore, actions taken by the agent may change what is called the **state** of the environment.

The interaction between the agent and the environment happens in a discrete-time fashion: at each of a sequence of time steps $t = 0, 1, 2, 3, \dots$ the agent selects and performs an action $A_t \in \mathcal{A}(S_t)$ based on some representation of the state $S_t \in \mathcal{S}$ of the environment, then, one step later, receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and observes the new representation of the state of the environment S_{t+1}. $\mathcal{S}$ is the set of possible states, also called the state space, and $\mathcal{A}(S_t)$ is the set of actions available in state S_t.

Formally, the goal of the action is to maximize its **return**, which is a function of the reward sequence collected during the decision-making. In the simplest case the return is just the sum of the rewards collected, but, if we want to introduce a connotation of how much an agent values taking rewards immediately with respect to taking them in the future, the return can be expressed as the discounted sum of rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $\gamma \in [0, 1]$ is a parameter called **discount factor** or discount rate and regulates how much short-sighted an agent will be: $\gamma = 1$ means that the agent values immediate rewards as much as future rewards, whereas with $\gamma = 0$ would take into account only step-wise rewards.

2.1.2 Markov Decision Processes

Markov Decision Processes, or MDPs, are a way of modeling sequential decision problems, a type of problem in which an agent's return is determined by a sequence of actions or decisions taken in an environment, rather than a single action. Markov Decision Processes are defined for stochastic environments that are fully observable: the state of the environment is known at any step, but the transition from a state to another occurs with probability.

Formally, an MDP is defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- $\mathcal{S}$ is the **state space**, which is the set of all possible states of the problem.
- $\mathcal{A}$ is the **action space**, which is the set of all possible actions in the problem. If the actions number is not fixed for each state, we need an action set $\mathcal{A}(s)$ for each state $s \in \mathcal{S}$.
- $P : \mathcal{S} \times \mathcal{A} \rightarrow \Delta\mathcal{S}$ is the **transition model**, which describes the effect of each action in each state. Given that Markov Decision Processes are non-deterministic, the transition model will be a probability distribution over the state space.
- $R(s, a) = \mathbb{E}[r|s, a]$ is the **reward function**, specifying the expected return obtained by the agent by performing action a in state s.
- $\gamma \in [0, 1]$ is called the **discount factor**, which represents how much an agent considers a reward valuable now with respect to the future.

2.1.3 Properties

A Markov Decision Process is said to be **finite** if the action space and the state space are both finite.

Markov property

In a Markov Decision Process, the **Markov property** must be satisfied for both the transition model and the reward function: given states $s, s' \in S$, the probability of transitioning to state s' from state s and observing a reward r must depend only on the current state s and not on the history of states previously visited by the agent. Recurring to the formal definitions found in [31], in the general case the response of the environment at time $t+1$ might depend on everything happened up to time t and the environment's dynamics can be described by specifying by the complete probability distribution:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}, \tag{2.1}$$

for all r, s' and all possible values of the past events S_0, A_0, R_1, $\ldots$ S_{t-1}, A_{t-1}, R_t, S_t, A_t. The Markov property is satisfied for a state signal if and only if 2.1 is equal to

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\}$$

for all $s' \in \mathcal{S}$ and histories $S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$. If this is the case, the one-step dynamics of the environment allow to predict the next state given only the knowledge of current state, regardless of the environment's history.

General sequential decision problems do not necessarily enjoy the Markov Property, and their reward is assumed to depend also on the history of the environment.

Time horizon

Markov Decision Processes are also characterized by a **time horizon**, which may be finite or infinite. If the time horizon of an MDP is finite, it means that the problem will end after a well-defined number of time steps, and no more rewards can be collected by the agent. More formally, there is a fixed time limit N after which

$$G_h([S_0, S_1, \ldots, S_{N+k}]) = G_h([S_0, S_1, \ldots, S_N])$$

that is, the return of a state history longer than N steps is the same as if the history were truncated at N steps.

2.1.4 Solving MDPs

A solution to an MDP is represented by a **policy**, which specifies for each state a probability distribution over the actions available in such state. Policies are usually denoted by π, and $\pi(s)$ represents the distribution specified by policy π in state s. When a policy is deterministic, $\pi(s)$ denotes a single action to be executed in state s.

Each time a policy is executed starting from the initial state s_0 of the environment, due to the stochastic nature of the problem, the sequence of states visited by the agent may vary. To evaluate a policy, it is therefore necessary to take the expected value over the return of the possible histories of the environment generated by following the policy.

As found in [31], the expected return obtained executing a policy π starting in a state s is given by

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} \,\middle|\, S_t = s \right] \tag{2.2}$$

where the expectation is with respect to the probability distribution over state sequences determined by s and π. The function v_π is called **state-value function**.

Similarly, the **state-action function** q_π is defined as

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} \,\middle|\, S_t = s, A_t = a \right] \tag{2.3}$$

which gives the expected return for taking action a in state s and then following policy π during the following state sequence.

An optimal policy, denoted with π^*, is a policy yielding the maximum expected return. All optimal policies for the same problem share their state-value function, called **optimal state-value function**, denoted $\mathbf{v}_*$, and defined as

$$v_*(s) = \max_\pi v_\pi(s)$$

for all $s \in \mathcal{S}$.
The same holds for the **optimal state-action** function:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The optimal state-action function is denoted with π_*.

2.1.5 Bellman equations

The functions introduced in the previous section are called **value functions**, and they all share one attractive property: they satisfy particular recursive relationships. Equations 2.2 and 2.3 can be rewritten as recursive equations, obtaining:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_\pi(s') \right) \tag{2.4}$$

and

$$q_\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a') \tag{2.5}$$

which are the **Bellman equations** for state-value function and state-action function, respectively. The Bellman optimality equations for the above functions can be obtained by considering optimal state-value function and state-action function in equations 2.4 and 2.5 respectively:

$$v_*(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_*(s') \right)$$

$$q_*(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') q_*(s',a')$$

2.1.6 Finding optimal policies

Once the optimal value functions for v and q are available, finding an optimal policy becomes easy.

Using v_*, it is sufficient to run a greedy one-step search from each state s over available actions and assign nonzero probability only to those actions that obtain the maximum for the Bellman equation, which is exactly $v_*(s)$. If, instead, we use the optimal q function, there is no need to run the one-step search: all the necessary information is already contained in $q_*(s,a)$. To obtain an optimal policy, it is sufficient to greedily select for each state the actions that maximize $q_*(s,a)$.

The interesting property of this approach is that the greedy approach to build the policy generates optimal policies not only with respect to single-step decisions, but also in the long run. As the authors state in [31], this is because the optimal value functions, as defined in equations 2.4 and 2.5 already take into account the reward consequences of all possible future behavior.

The problem with finding optimal policies based on optimal value functions is that their exact computation requires great amounts of time and

memory, which make the approach unfeasible for large enough problems or for time-limited applications.
RL's online approach to MDP solution is to approximate optimal policies in a way that spends most of the effort into learning to make good decisions in frequently encountered states. This means that in less frequent states RL agents might have poor performance, but due to those states seldom occurring, the overall solution to the MDP provided by the RL agent policy will still show good performance.

In the following sections a selection of methods to solve MDPs will be presented.

2.1.7 Dynamic Programming algorithms

Dynamic Programming (DP) is an optimization method based on Bellman's principle of optimality [2]:

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

This principle can be translated into practice by decomposing a multi-stage problem into a sequence of simpler subproblems, usually easier to handle than the whole problem. The solution obtained by applying DP follows a bottom-up approach, as the simpler subproblems' solutions are brought together to solve (possibly) many different larger problems. [26]

Value iteration algorithm

The value iteration algorithm [31] is a Dynamic Programming iterative algorithm that allows to find an optimal policy for MDPs.

The algorithm is based on the state-value Bellman equation and, starting from an arbitrary initialization of the state-value function for each state, at each iteration computes a new value for $v(s)$ from its past values, as shown in figure 2.1.
By applying the update infinitely many times, value iteration is ensured to converge to the optimal values.
Pseudocode in box 1 describes the value iteraton algorithm.

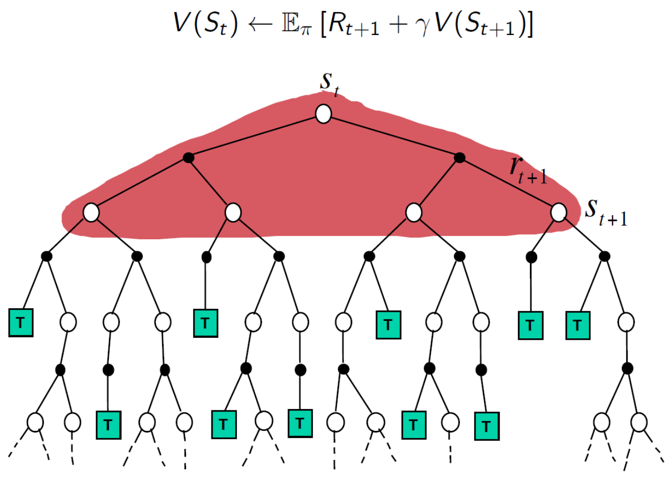

$V(S_t) \leftarrow \mathbb{E}_\pi\left[R_{t+1} + \gamma V(S_{t+1})\right]$
s_t
r_{t+1}
s_{t+1}
T

Figure 2.1: A visualization of the DP backup, taken from [29].

Policy iteration algorithm

Policy iteration algorithm [31] is also a dynamic programming iterative algorithm for finding optimal policies. The underlying idea is to gradually improve from an arbitrary initial policy π_0 towards π_*, through alternating **policy evaluation** and **policy improvement** steps.

Policy evaluation The policy evaluation step iteratively computes the values of $v_\pi(s)$ for all states, in a similar fashion to the value iteration algorithm. The difference is that policy evaluation updates the values of $v_\pi(s)$ only according to actions prescribed by policy π, instead of taking the maximum value obtained by any action.
Pseudocode in box 2 describes the iterative policy evaluation algorithm.

Algorithm 1 Value iteration algorithm

```
1: Initialize $v(s)$ to arbitrary values
2: while $v_{t+1}(s) \neq v_t(s)$ do                                  ▷ Repeat until $v(s)$ converges
3:     for all $s \in \mathcal{S}$ do
4:         for all $a \in \mathcal{A}$ do
5:             $q(s,a) \leftarrow \sum_{s} p(s'|s,a)\,(r(s,a) + \gamma v(s'))$
6:         end for
7:         $v(s) \leftarrow \max_{a} q(s,a)$                            ▷ Select greedily over q functions
8:     end for
9: end while
```

Algorithm 2 Policy evaluation algorithm

```
1: procedure POLICY-EVALUATION($\pi$)
2:     Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}$
3:     do                                        ▷ Repeat until $V(s)$ converges
4:         $\Delta \leftarrow 0$
5:         for all $s \in \mathcal{S}$ do
6:             $v \leftarrow V(s)$
7:             $V(s) \leftarrow \sum_a \pi(a|s)\,(R(s,a) + \gamma \sum_{s'} p(s'|s,a) v_\pi(s'))$
8:             $\Delta \leftarrow max(\Delta, |v - V(s)|)$
9:         end for
10:    while $\Delta < \Theta$                    ▷ $\Theta$ is a small positive number
11:    return $V(s) \approx v_\pi$
12: end procedure
```

The value function also represents the fixed point of the Bellman operator $\mathcal{T}_\pi : \mathbb{R}^n \to \mathbb{R}^n$, defined using vector notation as:

$$\mathcal{T}_\pi V = r_\pi + \gamma P_\pi V$$

The Bellman operator is equivalent to one iteration (also called bootstrap) of the dynamic programming algorithm, and its application infinitely many times causes the estimated value function to converge to the true one for the current policy π.

Policy improvement The policy improvement step, as the name suggests, iteratively improves the policy. This step is based upon the *policy improvement theorem* [31], which states that given a pair of policies π', π if for any state $s \in \mathcal{S}$ holds $q_\pi(s, \pi'(s)) \geq v_\pi(s)$, then policy π' is as good as or better than policy π. That is, it must obtain greater or equal expected return than policy π from all states $s \in \mathcal{S}$. Formally, $v_{\pi'}(s) \geq v_\pi(s)$.
The actual improvement of the policy is made by greedily selecting for each state those actions that achieve maximum values for the state-action function. Formally:

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, q_\pi(s,a).$$

Policy iteration By alternating policy evaluation and policy iteration steps, it is possible to obtain a monotonically improving policy sequence, leading after infinitely many steps to the optimal policy for the MDP:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*}$$

Algorithm 3 Policy iteration algorithm

```
 1: procedure POLICY-ITERATION
 2:     for all $s \in \mathcal{S}$ do                                ▷ Initialization step
 3:         Arbitrarily initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(\mathcal{S})$
 4:     end for
 5:     do                                                  ▷ Repeat until $\pi$ converges
 6:         $unchanged \leftarrow$ true
 7:         $v \leftarrow$ POLICY-EVALUATION($\pi$)                        ▷ Evaluation step
 8:         for all $s \in \mathcal{S}$ do                             ▷ Improvement step
 9:             $a \leftarrow \pi(s)$
10:             for all $a \in \mathcal{A}$ do
11:                 $q(s,a) \leftarrow \sum_{s} p(s'|s,a)\left(r(s,a) + \gamma v(s')\right)$
12:             end for
13:             $\pi(s) \leftarrow \operatorname{argmax}_{a} q(s,a)$          ▷ Select greedily over actions
14:             if $a \neq \pi(s)$ then                        ▷ Convergence evaluation
15:                 $unchanged \leftarrow$ false
16:             end if
17:         end for
18:     while $unchanged$
19:     return $V, \pi$
20: end procedure
```

where $\xrightarrow{E}$ and $\xrightarrow{I}$ denote the policy evaluation and policy improvement steps, respectively.

The complete policy iteration algorithm is described by the pseudocode in box 3. To keep the pseudocode simple, the check to avoid looping between equivalently good policies is omitted.

2.1.8 Monte Carlo Methods

Unlike previously presented methods, Monte Carlo (MC) Methods [31] exploit experience collected by interaction with the environment to estimate value functions, by considering complete episodes' returns to learn value functions. This class of algorithms does not need to have access to a complete model description but only to interact with it to collect sample returns.

The idea underlying Monte Carlo methods is that the state-value and action-value functions can be estimated through averaging returns for each state-action pair from episodes sampled by interacting with the environment.

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$

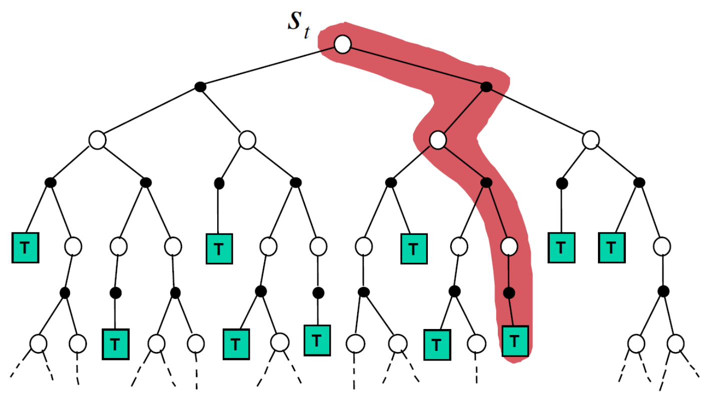


Figure 2.2: A visualization of the Monte Carlo backup, taken from [29]. The red trace represents a complete episode from the root of the search tree to a terminal state, marked with the green square.

To ensure well-defined returns are available, we adopt the definition of **episodic tasks** from [31]. Such definition assumes that experience is divided into episodes (see figure 2.2) and that all episodes eventually terminate, independently from the action sequence that is chosen during the episode. Only after the completion of an episode policies and value estimates may be updated.
Monte Carlo methods can be considered incremental in the sense that they learn after each complete episode but cannot be considered online methods as they cannot learn from partial returns.

Since the return for taking an action in a specific state depends on the sequence of actions that will be performed subsequently in the same episode, and since action selection is undergoing learning in all states, the problem becomes nonstationary from the perspective of the earlier state, as stated in [31].
The following sections will present a selection of MC methods to deal with such nonstationarity.

Learning the state-value function

To estimate $v_\pi(s)$ given a set of episodes obtained by executing policy π and passing through state s, we can use two Monte Carlo techniques: first-visit MC and every-visit MC. Each occurrence of a state s in an episode is called **visit**, and the difference between the two approaches is in how they consider

Algorithm 4 First-visit MC for estimating v_π

```
1: procedure MC-ESTIMATION($\pi$)
2:     Arbitrarily initialize $V$
3:     for all $s \in \mathcal{S}$ do
4:         $Returns(s) \leftarrow$ empty list
5:     end for
6:     while true do                                  ▷ The procedure never stops
7:         Generate an episode using $\pi$
8:         for all states $s$ appearing in the episode do
9:             $G \leftarrow$ return following the first occurrence of s
10:            $Returns(s) \leftarrow$ APPEND($Returns(s), G$)
11:            $V(s) \leftarrow$ AVERAGE($Returns(s)$)
12:        end for
13:    end while
14: end procedure
```

repeatedly visited states in the same episode.
When estimating $v_\pi(s)$, **first-visit MC** averages the returns following only the first visit to s in the episode, whereas **every-visit MC** estimates $v_\pi(s)$ by taking the average over the returns following any occurrence of s. The pseudocode in box 4, adapted from [31], sketches the functioning of first-visit MC method for v_π estimation.

Both first-visit and every-visit MC estimations converge quadratically to the true mean value of $v_\pi(s)$ as the number of visits to s goes to infinity [31].

Learning the state-action function

The same methods can be used to estimate the state-action function, which is useful to enable finding policies if no model of the environment is available. The difference with state-value function estimation lies in the definition of visit: in this case we consider state-action pairs rather than only states. We say that a state-action pair s, a is visited in an episode if state s was visited and action a was taken in it.

To effectively provide a policy to solve the problem, the value of all the actions in any state must be estimated. This is not guaranteed to happen if the policy is deterministic, thus we must assure continual exploration. A possible way to do this is to use the **exploring starts** assumption [31], which states that episodes start in a state-action pair and that every pair has a nonzero probability of being selected as the start. In the limit of infinite

Algorithm 5 Monte Carlo ES algorithm

```
 1: procedure MCES
 2:     $Q(s, a) \leftarrow$ arbitrary
 3:     $\pi(s) \leftarrow$ arbitrary
 4:     while true do
 5:         Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$
 6:         Generate an episode starting from $S_0, A_0$, executing $\pi$
 7:         for all pairs $s, a$ appearing in the episode do      ▷ Policy evaluation
 8:             $G \leftarrow$ return following the first occurrence of $s, a$
 9:             $Returns(s) \leftarrow$ APPEND($Returns(s, a), G$)
10:             $Q(s, a) \leftarrow$ AVERAGE($Returns(s, a)$)
11:         end for
12:         for all $s$ in the episode do                         ▷ Policy improvement
13:             $\pi(s) = \text{argmax}_a Q(s, a)$
14:         end for
15:     end while
16: end procedure
```

episodes, the exploring starts assumption ensures that any state is visited infinite times, leading to convergence of estimated values to their true mean value.

Monte Carlo Control

Monte Carlo control can be seen as a variation of the policy iteration algorithm, in which policy evaluation and policy improvement alternate after each episode. The returns collected from the episode are immediately used to improve the estimates of value functions of the visited states, then the policy is improved at these states, and a new episode is generated. As in policy iteration, the policy improvement step is carried out by greedily constructing the next policy π_{k+1} with respect to q_{π_k}:

$$\pi_{k+1}(s) = \underset{a}{\text{argmax}}\, q_{\pi_k}(s, a)$$

A simple algorithm exploiting this kind of process is shown in box 5 and is called Monte Carlo ES [31], standing for Monte Carlo with Exploring Starts.

The Monte Carlo ES algorithm is an example of **on-policy algorithm**, since it updates its value function estimations based on returns collected in episodes generated by executing the current policy. **Off-policy** methods, on the other hand, exploit information collected running a different policy, called behavior policy, from the one to be evaluated, which is called target policy. As the authors state in [31], an advantage of this separation is that the target policy may be deterministic, while the behavior policy can be explorative (i.e nondeterministic), thus avoiding the need for the exploring starts assumption.

Advantages of Monte Carlo methods

Monte Carlo methods present various advantages, among which three are most relevant.

The first one is that they can learn value functions, and subsequently policies, from experience obtained through direct interaction with the environment. This is particularly useful in those cases in which the model of the environment is not available.

Another advantage is that Monte Carlo methods do not require bootstrap, which means they can compute value functions for a state without building upon other states' estimates. This can make Monte Carlo methods attractive for those problems in which one needs the value of a subset of states: starting from the desired states, one can generate episodes and average only returns coming from those states of interest.

Lastly, Monte Carlo methods can also learn from simulated experience, enabling to provide reasonable policies without any interaction with the environment where the process of learning could be dangerous or even hurtful. This approach obviously requires a model approximating the behavior of the environment, but this is simply charged with generating the state transitions, it does not need to specify the complete probability distribution, which is instead required in dynamic programming methods.

2.1.9 Temporal Difference learning

Temporal difference learning (TD) [31] constitutes a hybrid approach between DP and MC methods: like the latter, it is able to learn from direct experience but, like the former, it can update its estimates before an episode is complete.

The simplest TD method for estimating the state-value function is **TD(0)**, which updates the estimates of $V(s)$ at each time-step following the rule

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right],$$

where α is a constant step-size parameter and $R_{t+1} + \gamma V(S_{t+1})$ is called the target for the TD update. Figure 2.3 depicts the TD backup update.

Algorithm 7 Sarsa on-policy TD control algorithm

```
1: procedure Sarsa
2:     for all $s \in \mathcal{S}, a \in \mathcal{A}$ do                    ▷ Initialize Q
3:         if $s$ is terminal then
4:             $Q(s, a) \leftarrow 0$
5:         else
6:             $Q(s, a) \leftarrow$ arbitrary
7:         end if
8:     end for
9:     for all episodes do
10:        $S \leftarrow$ episode starting state
11:        $\pi \leftarrow$ policy derived from $Q$                          ▷ e.g. $\epsilon$-greedy
12:        $A \leftarrow \pi(S)$
13:        while $S$ is not terminal do
14:            Take action $A$; observe reward, $R$, and next state, $S'$
15:            $A' \leftarrow \pi(S)$
16:            $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$
17:            $S \leftarrow S'; A \leftarrow A'$
18:            $\pi \leftarrow$ policy derived from $Q$
19:        end while
20:    end for
21: end procedure
```

Since it updates its estimate of $V(s)$ based on a previous estimate, TD(0) is a bootstrapping method, like DP methods. The procedural form of the algorithm, adapted from [31], is shown in box 6.

Sarsa

Sarsa [31] is an on-policy TD control method which uses the pattern of generalized policy iteration [31]. Sarsa uses TD learning to learn the state-action function for the current policy, considering transitions from a state-action pair to another and using the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ performs the update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

after each transition. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero.
Box 7 presents the Sarsa control algorithm, adapted from [31].

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$

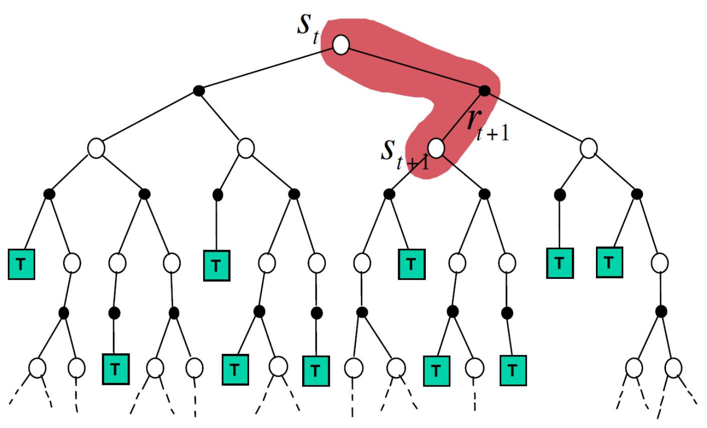


Figure 2.3: A visualization of the TD backup, taken from [29].

Algorithm 6 TD(0) algorithm

```
 1: procedure TD(0)($\pi$)                                ▷ $\pi$ is the policy to be evaluated
 2:     for all $s \in \mathcal{S}$ do
 3:         $V(s) \leftarrow$ arbitrary initialization
 4:     end for
 5:     for all episodes do
 6:         $S \leftarrow$ episode starting state
 7:         $t \leftarrow 0$
 8:         while $S$ is not terminal do
 9:             $A \leftarrow \pi(S)$                         ▷ Get action prescribed by policy
10:             Take action $A$; observe reward, $R$, and next state, $S'$
11:             $V(S) \leftarrow V(S) + \alpha\,[R + \gamma V(S') - V(S)]$
12:             $S \leftarrow S'$
13:         end while
14:     end for
15: end procedure
```

Algorithm 8 Q-learning algorithm (off-policy TD control)

```
1: procedure Q-LEARNING($\alpha, \varepsilon$)    ▷ $\alpha \in (0, 1], \varepsilon$ small and $> 0$
2:     for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ do
3:         Initialize $Q(s, a)$ arbitrarily, except for $Q(terminal, \dot{}) = 0$
4:     end for
5:     for all episodes do
6:         $s \leftarrow$ episode starting state
7:         while $s$ is not terminal do
8:             $a \leftarrow \pi(s)$    ▷ $\pi$ is derived from $Q$ (e.g., $\varepsilon$-greedy)
9:             Take action $a$; observe reward, $R$, and next state, $s'$
10:            $Q(s, s) \leftarrow Q(s, a) + \alpha\,[R + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
11:            $s \leftarrow s'$
12:        end while
13:    end for
14: end procedure
```

The authors of [31] report that *"Sarsa algorithm has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability equal to one if the step-size parameter decreases according to the usual stochastic approximation condition"*.

Q-learning

Q-learning is an off-policy TD control algorithm [33], defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The goal of the algorithm is to directly approximate the optimal action-value function q_*, regardless of the policy followed by the agent. The meaning of the policy in Q-learning is to determine which state-action pairs are to be visited by the agent, but the only requirement for convergence is that all pairs continue to be visited over time [31].
Box 8, adapted from [31], presents the pseudocode for the implementation of Q-learning.

Advantages of TD learning

TD methods present advantages over both DP and MC methods, analyzed by the authors in [31].
Being able to learn from direct experience, the first and foremost advantage of TD methods over DP methods is that they do not require a model of the

environment.
Another obvious advantage over MC methods is that Temporal Difference methods are naturally suited to online and fully incremental implementations since they do not need to wait until the end of episodes as Monte Carlo methods do. This is especially useful in applications with very long episodes, or whose episodes may never terminate (continuing tasks).
Moreover, TD, unlike some MC methods, does not need discounting episodes in which experimental actions are taken, since it learns from single transitions disregarding subsequent actions, leading to increased robustness against slowed learning coming from the discounting.

2.1.10 Eligibility traces based methods

Eligibility traces [31] offer a simple method for credit assignment in sequential decision making. The intuition behind them is to hold a memory of the last visited states eligible for the decision-making effects. This approach is called the **backward view** of eligibility traces.

TD(λ)

Eligibility traces can be combined with Temporal Difference methods to obtain the **TD(λ)** algorithm [31], which is used for predicting the state-value function. In this section, we focus on the backward view variant of this algorithm.

In this version of the algorithm, an additional variable called **eligibility trace** is associated with each state. The eligibility trace E, in its simplest form, is defined as

$$\begin{cases} E_t(s) = \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \\ E_t(s) = \gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t \end{cases}.$$

This kind of eligibility trace is called accumulating trace because it accumulates every time the state is visited, then decays with a factor of λ, which is called the **trace-decay parameter**. The eligibility traces act as a memory and indicate for each state its level of eligibility for undergoing learning changes due to the occurrence of reinforcing events.

The algorithm uses the TD error, defined for the state-value function as

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t),$$

to trigger updates to the state-value functions of all the states that are eligible following the update rule:

$$V_{t+1}(s) \leftarrow V_t(s) + \alpha\delta_t E_t(s), \text{ for all } s \in \mathcal{S}$$

The algorithm in box 9 presents the complete algorithm for backward view TD(λ), adapted from [31].

Algorithm 9 TD(λ) algorithm with accumulating traces

```
 1: procedure TD-$\lambda$
 2:     for all $s \in \mathcal{S}$ do                                      ▷ Initialize V
 3:         if $s$ is terminal then
 4:             $V(s) \leftarrow 0$
 5:         else
 6:             $V(s) \leftarrow$ arbitrary
 7:         end if
 8:     end for
 9:     for all episodes do
10:         $E(s) \leftarrow 0$, for all $s \in \mathcal{S}$          ▷ Initialize eligibility traces
11:         $S \leftarrow$ episode starting state
12:         while $S$ is not terminal do
13:             $A \leftarrow \pi(S)$
14:             Take action $A$; observe reward, $R$, and next state, $S'$
15:             $\delta = R + \gamma V(S') - V(S)$
16:             $E(S) \leftarrow E(S) + 1$                              ▷ Update traces
17:             for all $s \in \mathcal{S}$ do
18:                 $V(s) \leftarrow V(s) + \alpha\delta E(s)$
19:                 $E(s) \leftarrow \gamma\lambda E(s)$                 ▷ Decay traces
20:             end for
21:         end while
22:     end for
23: end procedure
```

It is interesting to notice that the parameter λ transforms on its extreme values the behavior of TD(λ) into that of TD(0) and Monte Carlo prediction. When $\lambda = 0$, all the eligibility traces go to zero except the one for the current state, reducing to TD(0) behavior. On the other hand, if $\lambda = 1$, eligibility traces never decay, leading to a full episode evaluation like in Monte Carlo methods.

TD(λ) acts, therefore, as a blend of Monte Carlo and TD methods, with the λ parameter controlling how much the behavior of the algorithm is oriented towards complete returns or one-step returns.

Algorithm 10 Sarsa(λ) control algorithm with accumulating traces

```
1: procedure SARSA-$\lambda$
2:     for all $s \in \mathcal{S}, a \in \mathcal{A}$ do                          ▷ Initialize Q
3:         if $s$ is terminal then
4:             $Q(s,a) \leftarrow 0$
5:         else
6:             $Q(s,a) \leftarrow$ arbitrary
7:         end if
8:     end for
9:     for all episodes do
10:        $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
11:        $S \leftarrow$ episode starting state
12:        $\pi \leftarrow$ policy derived from $Q$                          ▷ e.g. $\epsilon$-greedy
13:        $A \leftarrow \pi(S)$
14:        while $S$ is not terminal do
15:            Take action $A$; observe reward, $R$, and next state, $S'$
16:            $A' \leftarrow \pi(S)$
17:            $\delta = R + \gamma Q(S',A') - Q(S,A)$
18:            $E(S,A) \leftarrow E(S,A) + 1$                          ▷ Update traces
19:            for all $s \in \mathcal{S}, a \in \mathcal{A}$ do
20:                $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
21:                $E(s,a) \leftarrow \gamma\lambda E(s,a)$                          ▷ Decay traces
22:            end for
23:            $S \leftarrow S'; A \leftarrow A'$
24:        end while
25:    end for
26: end procedure
```

Sarsa(λ)

As in previous sections, we will move to the control algorithm once we defined the state-value function estimation method.

Sarsa(λ) is an on-policy control algorithm based on the generalized policy iteration pattern which uses the TD(λ) method to predict state-action value functions. Unlike TD(λ), though, it requires an eligibility trace for each state-action pair.
The algorithm, adapted from [31], is shown in box 10.

2.2 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) [5] is a technique aimed at finding optimal decisions for MDPs. The method works by taking samples from the decision space and using the results to build a search tree, whose estimates are progressively updated through time.
The MCTS process can be divided into four phases:

1. **Selection**: from the root of the search tree, a selection policy is recursively applied up to reaching an unexpanded leaf node. A node is expandable if it represents a nonterminal state and has unvisited children.
2. **Expansion**: one or more successors to the previously found unexpanded node are generated according to the actions available in the node.
3. **Simulation**: from the newly generated node(s), a simulation (also called rollout) is run up to a terminal state.
4. **Backpropagation**: a "back-up" of the rewards collected during the simulation is performed at the nodes which have been visited in the trajectory from the root, updating their statistics.

A visualization of the phases of MCTS is provided by figure 2.4.

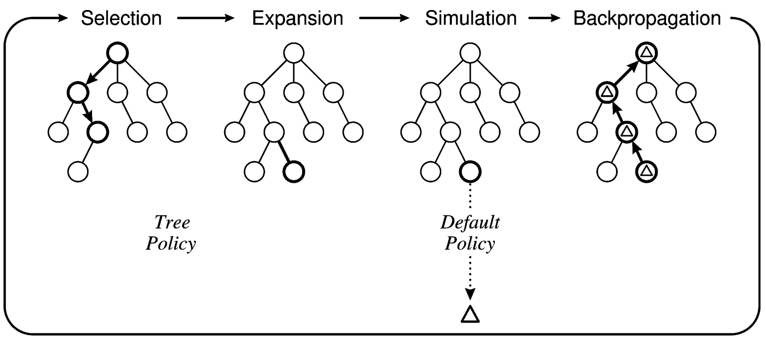


Figure 2.4: A visualization of the general MCTS algorithm phases, taken from [5].

The result of the overall search of MCTS is the action that taken at the search tree's root node, corresponding to the current MDP state, will lead to its best child. The concept of best child is defined by the specific implementation, but among the most popular criteria are, as described by Schadd in [28], based on the work of Chaslot *et al.* [7]:

- *Max child*: the root child with highest mean reward is selected.

- *Robust child*: the most visited root child is selected.
- *Max-robust child*: the root child with both highest mean reward and visit count is selected; if none of the root children satisfy such condition, the search is resumed and continues until the condition is satisfied. [10]
- *Secure child*: the child maximizing a lower confidence bound is selected.

The general MCTS algorithm is presented in box 11, adapted from [5]. The **tree policy** is the selection policy in force while visiting already explored portions of the tree, while the **default policy** is the rollout policy used from a leaf state to produce a value estimate. They are represented by the TreePolicy and DefaultPolicy functions, respectively, in the algorithm.

2.2.1 Taxonomy

Many dichotomies may be considered when analyzing MCTS algorithms, based on different properties that each of them shows.

As MCTS is a class of decision-time planning algorithms [31], a first intuitive classification can be done by considering their ability to be interrupted and return an action to be executed at any time during their computation. Such algorithms are said to be **anytime**.
Consider a time-limited planning setting, in which the planner must return its decision at the end of the computation time budget. Anytime algorithms, like the standard UCT algorithm [18] [17], are able to interrupt the search at any moment and return the current search tree's root node statistics.
This is not the case for non-anytime algorithms (see, for instance, OLOP [6]), which, instead, cannot return any meaningful result before their computation is complete.

Considering, instead, the representation and dynamics of MDPs we can observe two more classification criteria for MCTS algorithms.
The first classification is connected to the state transition function's properties, and we can distinguish between algorithms for **deterministic** transitions and **non-deterministic** ones. If transitions in the MDP are stochastic, the algorithm must account for the variability in successor states, given the same action performed at the parent state node. This can usually be done either by adding multiple successor nodes (see, for instance, Double Progressive Widening [9]), or devising a representation such that a single node can handle the uncertainty on the current state.

Algorithm 11 General MCTS approach

```
1: procedure MctsSearch($s_0$)
2:     Create root node $v_0$ from state $s_0$
3:     while within computational budget do
4:         $v_l \leftarrow$ TreePolicy($v_0$)
5:         $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
6:         Backup($v_l, \Delta$)
7:     end while
8:     return $a(BestChild(v_0))$
9: end procedure
```

The latter approach, by adopting the automation terminology, is said to be **open-loop**, because the representation of current states used for the planning is based on the action sequence instead of the actual MDP's state representation. This approach contrasts with the so-called **closed-loop** approach, which instead considers a planning representation based on sequences of state realizations, or trajectories.
Usually, non-deterministic algorithms are able to handle also deterministic transition settings.

Focusing on the **state-space** and **action-space** in the representation of an MDP, we find that these sets may be either **continuous** or **discrete**. Respectively, algorithms have to deal with an infinite set of states, actions, or even both at the same time.
The naïve approach to this kind of spaces is to store in the tree all possibilities that are encountered while interacting with the environment. For long enough episodes, this quickly leads to the intractability of the problem, requiring, for instance, discretization techniques like Double Progressive Widening [9] to cope with the continuous nature of the problem.

On the implementation side, some algorithms need to be able to set a specific state in the environment, either to resume the search or to take more samples from a specific state represented by a node in the tree. This is a relevant point, as not every environment implementation offers this possibility (see, for instance, the Atari games Gym environments[1]) and might limit the applicability of some algorithms.

[1]`https://gym.openai.com/envs/#atari`

Chapter 3

State of the art

This chapter introduces relevant contributions from the existing body of literature. It presents some popular and well-known algorithms along with more problem-oriented approaches, specifically in the areas of MCTS, open-loop planning, and action-value function estimators.

3.1 Upper Confidence Bound for Trees (UCT)

Among the most popular MCTS algorithms, UCT [19] is based on the simple, yet effective, bandit algorithm **UCB-1** [1]. The goal of MCTS is to estimate the action-value function for actions that are available in the current state of the MDP by iteratively building a partial search tree. The intuition behind UCT is to consider action selection at each node of the tree as a bandit problem. By applying the UCB-1 algorithm to such a problem, it is possible to tackle the exploration-exploitation dilemma and focus on using the search computational budget towards the most promising options, avoiding building the full search tree for problems in which this may not be feasible in a reasonable time.

An **upper confidence bound** is associated with each action:

$$u_t(s,a,d) = Q_t(s,a,d) + 2C_p\sqrt{\frac{2\ln n}{n_a}}, \tag{3.1}$$

where $Q_t(s,a,d)$ is the estimation for the action-value function at time t for executing action a in state s, when at depth d; n is the number of times the node corresponding to state s at depth d was visited, whereas n_a is the number of times action a was selected; C_p is the **exploration constant** and the larger the value, the more uniform the exploration of each level of the tree will be.

Algorithm 12 Upper Confidence Bound for Trees

```
 1: procedure UctSearch($s_0$)
 2:     Create root node $v_0$ from state $s_0$
 3:     while within computational budget do
 4:         $v_l \leftarrow$ TreePolicy($v_0$)
 5:         $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
 6:         Backup($v_l, \Delta$)
 7:     end while
 8:     return $a$(BestChild($v_0$))
 9: end procedure

10: procedure TreePolicy($v$)
11:     while $v$ not terminal do
12:         if $v$ not fully expanded then
13:             return Expand($v$)
14:         else
15:             $v \leftarrow$ BestChild($v, C_p$)
16:         end if
17:     end while
18:     return $v$
19: end procedure

20: procedure DefaultPolicy($s$)
21:     while s is non-terminal do
22:         choose $a \in A(s)$ uniformly at random
23:         Generate next state $s'$
24:         $s \leftarrow s'$
25:     end while
26:     return $R(s, a)$
27: end procedure

28: procedure Expand($v$)
29:     Choose $a \in$ untried actions from $A(s(v))$
30:     Add a new child $v'$ to $v$ generated by executing $a$ in $s(v)$
31:     return $v'$
32: end procedure
```

```
33: procedure BestChild($v, c$)                ▷ $C(v)$ denotes child states of $v$
34:     return $\text{argmax}_{v' \in C(v)} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
35: end procedure

36: procedure Backup($v, \Delta$)
37:     while $v$ is not null do
38:         $N(v) \leftarrow N(v) + 1$
39:         $Q(v) \leftarrow Q(v) + \Delta(v, p)$
40:         $v \leftarrow$ parent of $v$
41:     end while
42: end procedure
```

During the selection phase in UCT, the action maximizing the bound in equation 3.1 is selected. The counts for action and node visits are updated, together with the Q-function estimates, during the backup phase.
Box 12 presents the pseudocode for UCT algorithm, adapted from [5].

UCT has some relevant properties, both theoretical and practical, which, combined with the ease of implementation of this algorithm, contribute to the algorithm's popularity in the MCTS class.
First, the algorithm's estimator for the action-value function is **consistent**, which means the empirical mean converges to the true value of the mean for the value function. To this regard, theorem 7 from [20] states

> *Consider algorithm UCT running on a game tree of depth D, branching factor K with stochastic payoffs at the leaves. Assume that the payoffs lie in the interval* $[0, 1]$*. Then the bias of the estimated expected payoff,* X_n*, is* $O((KD\log(n)+KD)/n)$*. Further, the failure probability at the root converges to zero as the number of samples grows to infinity.*

Moreover, UCT is a **trajectory-based** algorithm; this means that at each search iteration, it considers complete sequences of actions built incrementally before performing a backup. Lastly, the UCT algorithm is an **anytime** algorithm, meaning that it does not need to know its planning budget and can be interrupted at any point, returning an action suggestion based on the current action-value function estimations held at the root.

3.2 Double Progressive Widening

For problems with high stochasticity in action outcomes, plain MCTS performance is heavily limited because low probability outcomes might be added to the tree and never get visited again, leading to highly biased estimates of their statistics. Moreover, in continuous state (and possibly action) spaces, it is impossible to represent the problem without some discretization technique. Therefore, as shown by Bjarnason *et al.* in [3], it is beneficial to adopt some tree width limitation technique to reduce the number of child nodes in the search tree to accumulate meaningful statistics for them.

Couëtoux and Doghmen propose in their work [9] to apply the **Double Progressive Widening (DPW) technique** to Monte Carlo Tree Search. DPW provides a method to progressively add and limit successor nodes as a node gets explored more and more. This is done through the definition of two functions of the number of times an action a has been taken in state s

$$k = \lceil Cn^{\alpha} \rceil \tag{3.2}$$

and

$$k' = \lceil C{n_a}^{\beta} \rceil, \tag{3.3}$$

where C, α and β are constants specific to the algorithm, with $C > 0, \alpha \in (0, 1), \beta \in (0, 1)$; n is the number of times a state s has been visited, whereas n_a is the number of times action a has been taken in such state.

Equation 3.2 represents a limit for the number of actions to be considered when the action space is continuous. Equation 3.3, on the other hand, represents the limit to the number of successors a node can have depending on the number of visits it received.
Whenever the current number of actions or successors to a node is lesser than their respective bound, the algorithm is allowed to add another action or successor, respectively.

When applied to UCT, DPW retains UCT's theoretical properties and represents a **consistent** estimator for the action-value function. Moreover, like UCT, DPW is an **anytime** and **trajectory-based** algorithm. Finally, as this work focuses on a MDP characterized by continuous state space and discrete actions, we consider an experimental comparison implementation of DPW leveraging only the bound of equation 3.3 over successor states.

3.3 Open Loop Optimistic Planning

Whereas previously described algorithms were based on a closed-loop approach, it is possible to devise search-tree-based algorithms exploiting an

Algorithm 13 Open Loop Optimistic Planning

```
1: procedure OLOP(n)
2:     Let $M$ be the largest integer such that $M\left\lceil log\frac{M}{2log\frac{1}{\gamma}}\right\rceil \leq n$
3:     $L \leftarrow \left\lceil log\frac{M}{2log\frac{1}{\gamma}}\right\rceil$
4:     for all episodes $m = 1, 2, \dots, M$ do
5:         Compute $B$-values at time $m-1$ for sequences of actions $\in A^L$
6:         Choose the action sequence $a^m \in \text{argmax}_{a\in A^L} B_a(m-1)$
7:         Execute the action sequence and observe the rewards sequence
8:     end for
9:     return most played action $a(n) = \text{argmax}_{a\in A} T_a(M)$
10: end procedure
```

open-loop approach. This is the case for Bubeck and Munos' Open Loop Optimistic Planning (OLOP) [6], which was designed to work in a stochastic and discounted environment, jointly with a limited numerical budget n.

The intuition behind OLOP is to consider the exploration phase of the planning as a bandit problem having as arms all the possible action sequences from the current state to the end of the MDP.

In a UCB-1 fashion, the algorithm assigns to each sequence an **upper confidence bound**, defined as:

$$U_a(m) = \sum_{t=1}^{h}\left(\gamma^t \hat{\mu}_{a_{1:t}}(m) + \gamma^t\sqrt{\frac{2\log M}{T_{a_{1:t}}(m)}}\right) + \frac{\gamma^{h+1}}{1-\gamma}$$

where $1 < m < M$ is the index of the current episode, $T_a(m)$ is the number of times the algorithm has played a sequence of actions beginning with a, and $\hat{\mu}_a(m)$ is the empirical average of the rewards for the sequence a.
Then, a further sharpening of the confidence bounds is applied, obtaining the B**-values**:

$$B_a(m) = \inf_{1\leq h\leq L} U_{a_{1:h}}$$

The algorithm proceeds at each round by selecting the sequence with the highest B-value, observes the reward, and updates the B-values. Finally, when the budget is completely spent, the algorithm returns the most played action at the root of the search tree.

By considering complete action sequences, OLOP can perform the search in an open-loop fashion by disregarding the actual states visited by the agent

Algorithm 14 General structure for Open-Loop Optimistic Planning

```
1: for all episodes $m = 1, 2, \dots, M$ do
2:     Compute $U_a(m-1)$ for all $a \in \mathcal{T}$
3:     Compute $B_a(m-1)$ for all $a \in A^L$
4:     Sample a sequence with highest B-value: $a^m \in \mathrm{argmax}_{a\ inA^L} \mathcal{T}_a(M)$
5: end for
6: return the most played sequence $a(n) \in \mathrm{argmax}_{a \in A^L} T_a(M)$
```

while executing the action sequence in the search environment. Interestingly, only the first action of the sequence that the planner deems optimal at the current time-step is actually executed in the real environment, and the next state becomes the root for the following search.
Box 13 presents the pseudocode for OLOP, adapted from [6].

Unlike the previously presented algorithms, UCT and DPW, OLOP offers theoretical guarantees over simple regret [6]. Moreover, the upper bounds built by OLOP hold with high probability, whereas UCT bounds do not hold in practice until the algorithm is in an advanced phase. This makes OLOP both **consistent** and **provably efficient**, whereas UCT is only consistent.
Unfortunately, since the algorithm considers all the possible action sequences in the selection phase, its exponential computational complexity makes it **impractical** to use for problems with long planning sequences.
Another difference between OLOP and UCT-like algorithms is that OLOP chooses a full action sequence at each planning iteration, whereas UCT incrementally builds the action sequence by taking decisions at each node in the tree.
A final difference between the algorithms is that OLOP is **not anytime** since it needs to know its computational budget to compute the maximum planning depth, and it cannot return an action suggestion based on intermediate data.

3.4 KL-OLOP

The algorithm devised by Leurent and Maillard, KL-OLOP [23], extends OLOP by providing tighter upper bounds exploiting the Kullback-Leibler divergence [21]. The authors argue that the structure of OLOP can be generalized as shown in box 14, taken from [23]. Notation is the same used in section 3.3, with the addition of $\mathcal{T} = \sum_{h=0}^{L} A^h$, which stands for the look-ahead tree of depth L.

Based on such generalization, the authors also provide a more general form for the upper and lower confidence bounds on empirical means of intermediate rewards collected by the agent:

$$U_a^\mu(m) = \max\left\{q \in I : T_a(m) d\left(\frac{S_a(m)}{T_a(m)}, q\right) \leq f(m)\right\}$$

$$L_a^\mu(m) = \min\left\{q \in I : T_a(m) d\left(\frac{S_a(m)}{T_a(m)}, q\right) \leq f(m)\right\}$$

where I is an interval, d is a divergence on $I \times I \to \mathbb{R}^+$ and f is a non-decreasing function.

Seen under this framework, the original OLOP algorithm uses a quadratic divergence $d_{\texttt{QUAD}}$ on $I = \mathbb{R}$ and a constant function f_4, whose definition is reported in section 2.3 of [23].
KL-OLOP, instead, uses a different constant function f_2, as defined in section 2.4 of [23], and the **Bernoulli Kullback-Leibler divergence** $d_{\texttt{ber}}$ defined on the interval $I = [0, 1]$ as

$$d_{\texttt{BER}}(p, q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

The constant function f_2 is lower than f_4 and, combined with $d_{\texttt{BER}}$, provides tighter bounds while maintaining the regret bound properties of OLOP in high probability [23].

The authors motivate their work, particularly the use of a different divergence and constant function, to overcome the practical unfeasibility of OLOP. In section 2.3 of [23], Leurent and Maillard show that OLOP suffers from a uniform exploration behavior while the early depths of the tree have not been explored sufficiently yet. Once they get explored, the algorithm resumes its intended behavior but the problem persists, getting shifted to deeper areas of the tree. In practice, this means that OLOP needs to consume large amounts of the computational budget before starting to reduce its exploratory behavior in favor of exploiting the collected information. The authors connect this undesired behavior with the Chernoff-Hoeffding bounds used in OLOP, which, as they state, *"start in the $U_a^\mu(m) > 1$ regime, and can remain in this regime for a long time, especially in the near-optimal branches where $\hat{\mu}_a(m)$ is close to one".* Their proposed bound, instead, guarantees by construction that $U_a^\mu(m) \in I = [0, 1]$, leading to faster convergence of the algorithm.

3.5 Power UCT

The work of Dam *et al.* argues about the learning speed-up obtainable from the application of the power mean operator applied to MCTS backup [11]. The authors propose a straightforward modification to the classic UCT-1 algorithm leveraging a backup operator based on **power mean** [24], which they call Power UCT. The goal of such methodology is to bridge the mean and maximum estimators, as the authors write, "balancing between the negatively biased estimate of the average reward and the positively biased estimate of the maximum reward".

The authors show that Power UCT satisfies the original UCT theorems [11] and argue that their work can represent a generalization of such algorithm, exploiting the more general power mean operator. The power mean backup operator of order p is defined for rewards $X \in [0,1]$ as:

$$\bar{X}_n(p) = \left(\sum_{i=1}^{K} \left(\frac{T_i(n)}{n} \right) \bar{X}^p_{i,T_i(n)} \right)^{\frac{1}{p}}, \tag{3.4}$$

where p is an extended real number, n is the total number of visits to a state node, $T_i(n)$ is the number of times action $i \in K$ has been played and $\bar{X}_{i,T_i(n)}$ is the estimator of the reward obtained playing action i.
In particular, for specific values of p, expression 3.4 yields the arithmetic mean ($p = 1$), the geometric mean ($p \to 0$) or the harmonic mean ($p = -1$) [24]. The larger p, the more the backup operator weighs optimistic samples.

The authors prove that by adding the power mean backup operator, the modified algorithm still retains the theoretical properties of UCT. Moreover, the authors show on a selection of RL problems that Power UCT can obtain better performance at the cost of tuning an extra hyperparameter, p. Such performance is shown to be obtained in the experiments for high values of p, which shift the backup operator further from the arithmetic mean and closer to the max operator.

3.6 Q-learning with UCB-Hoeffding

Jin *et al.* discuss in their work [16] the efficiency of Q-Learning methods and propose a Q-Learning algorithm with an **upper-confidence bound** exploration strategy. The main difference with ε-greedy Q-Learning is in the update rule, which includes the upper confidence bound in the Q-value:

$$Q_h(s,a) \leftarrow (1-\alpha_t)Q(s,a) + \alpha_t[r_h(s,a) + V_{h+1}(s') + b_t]$$

Algorithm 15 Q-Learning with UCB-Hoeffding

```
1: procedure QL-UCB-H($\mathcal{S}, \mathcal{A}, H$)
2:     for all $(s, a, h) \in \mathcal{S} \times \mathcal{A} \times H$ do
3:         initialize $Q_h(s, a) \leftarrow H$ and $N_h(s, a) \leftarrow 0$
4:     end for
5:     get initial state $s_1$
6:     for all steps $h = 1, \ldots, H$ do
7:         take action $a_h \leftarrow \text{argmax}_{a'} Q_h(s_h, a')$, observe $s_{h+1}$
8:         $t = N_h(s_h, a_h) \leftarrow N_h(s_h, a_h) + 1$
9:         compute $b_t$ as in eq. 3.5
10:        $\alpha_t = \frac{H+1}{H+t}$
11:        $Q_h(s, a) \leftarrow (1 - \alpha_t)Q(s, a) + \alpha_t[r_h(s, a) + V_{h+1}(s') + b_t]$
12:        $V_h(s_h) \min H, \max_{a' \in \mathcal{A}} Q_h(s_h, a')$
13:    end for
14: end procedure
```

where b_t is the upper-confidence bound at t-th time the state-action pair was visited. The authors define it as:

$$b_t = c\sqrt{\frac{H^3 log(SAT/p)}{t}} \tag{3.5}$$

where c is a constant, H is the time horizon of the problem, S and A are respectively the cardinalities of the state space and action space, and T is the total number of steps.

Defining the action-value function estimator this way, an ε-greedy selection is no more necessary, as the confidence bound of an action that has been selected few times increases as the total number of visits to the parent state node increases. In fact, the algorithm (detailed in box 15) uses a greedy selection policy. Moreover, confidence bounds promote exploration at the start, while they move to progressive exploitation once enough information on the reward distributions has been collected.

Even if the algorithm proposed by Jin *et al.* is a Q-learning variation, their proposed backup operator can be used in the fourth phase of a MCTS algorithm, as it has been done in this work's experimental comparison.

Chapter 4

Race strategy identification problem

This chapter introduces the necessary background to understand the complexity of determining Formula 1 race strategies, then discusses the difficulties in modeling such a problem as a Markov Decision Process and possible approaches.

4.1 Formula 1 race strategy

In circuit motorsport events, participating race cars have to complete a defined amount of laps around a closed circuit. The final order in which drivers cross the finish line is used to award points, usually to both teams and drivers.

Formula 1 (F1) is a world championship of motor racing, in which open-wheel, single-seater race cars compete. In Formula 1, points are currently awarded after each race to the top-ten finishing drivers and their respective teams; therefore, each driver's goal is consistently achieving the best final placement possible. This final placement does not depend solely on drivers' skill or car performance but can be significantly influenced by their tire strategy. In Formula 1, tire replacement is allowed during the race, and at least two different tire types (also called **compounds**) must be fitted across the event, under penalty of disqualification from the race. We call **tire strategy** the sequence of compounds fitted on each car, joint with the number of laps each tire set was used.

The possibility of fitting fresh tires, also called **pit-stop**, represents one of the most strategic opportunities in Formula 1, allowing either to become faster than direct competitors or to overtake them through time gap gains

Season	A1	A2	A3	A4	A5	A6	A7
2014	Hard	Medium	Soft	Supersoft	-	-	-
2015	Hard	Medium	Soft	Supersoft	-	-	-
2016	Hard	Medium	Soft	Supersoft	Ultrasoft	-	-
2017	Hard	Medium	Soft	Supersoft	Ultrasoft	-	-
2018	Superhard	Hard	Medium	Soft	Supersoft	Ultrasoft	Hypersoft
2019	-	C1	C2	C3	-	C4	C5

Table 4.1: Overview of available tire compounds in the seasons 2014 to 2019. Taken from [15]

instead of passing them on track. The strength of such moves varies from track to track, with relative compound performance, and with the cars' performance difference.

As F1 regulations often change quickly, this work focuses on the so-called "turbo-hybrid era" (2014 - present) cycle. Such a period enjoyed relative tire regulation stability, with a single change in the approach to tire availability in 2016.

In 2014 and 2015 only two compounds were available during the race, one with higher lap-time performance but lower durability (also called "soft" or "prime" compound) and the other presenting higher durability at the expense of lap-time performance (also called "hard" or "option"). Since 2016, three compounds have been available at each race. In both periods, the manufacturer has been choosing and providing to the teams, from a hardness spectrum, the most suitable compounds for the track, mainly in relation to the asphalt's parameters (e.g., temperature and abrasion) and the subsequent expected life of the tire.

F1 tires vary in their constituent rubber's hardness, determining different asphalt grip levels and tire degradation speed. In the same conditions, harder tires tend to offer less grip but usually have greater durability, whereas softer tires offer more grip to the car at the price of reduced durability. Table 4.1 reports the relative hardness and denomination of Pirelli tire sets across the years, as seen in [15]. As speed is limited to 80 Km/h in the pit-lane, depending on the track layout, the cars lose about 20 to 30 seconds on their lap time to perform a pit-stop. This is due to both the difference of speed between the pit-lane and the main straightaway and the actual time the car has to be still while the tires are being fitted.

The tradeoff between grip level and tire durability is crucial in race strategy determination, as higher grip levels make the car faster through corners as well as in braking and acceleration, whereas increased durability reduces the

number of pit-stops needed to complete the race distance. In order to devise a good tire strategy, it is therefore essential to balance the benefit of fitting fresh tires with the cost of stopping the car to fit them.
Since the driver needs to sacrifice immediate lap-time by performing a pit-stop to gain performance in the remainder of the race, tire strategy identification presents an interesting sequential decision-making problem. Moreover, finding a good tire strategy represents one of the most important strategical options available during the race event.

There are four typical tire-related strategic options in formula 1, whose viability depends on car performance and time gaps between the considered driver and his direct competitors.
The first one is to **differentiate** the base tire strategy with respect to competitors. Each team, before the race, plans several alternative strategies corresponding to as many race scenarios. These scenarios account for different factors, including the actual tire degradation rate, closeness with competitors and their compound choices, the possibility of rain, and even accidents on the track. The best-fitting strategy plan is chosen during the race considering decision support systems' predictions and guesses on other teams' strategies, sometimes trying to bet on a particular scenario (unlikely as it may be) that other teams are not covering. If these extreme scenarios come true, the team usually finds itself in an advantageous position that might make up for the lack of car performance in that race event.
When considering teams with similar performance, they are expected to make similar tire strategy choices. When this does not happen, it is usually mainly related to the joint tire-car performance: with a specific compound, a team might have a better speed-durability tradeoff than others. For instance, this was the case from 2017 on for Mercedes, whose car had strong race pace performance with medium-hardness compounds, which, coupled with good stint durability, allowed them to be (almost) as fast as competitors on softer tires and to suffer less from tire degradation issues.
Another difference in strategy choices may be found in the order of compounds used, whose motivation will be explained through an example. Imagine, for instance, a one-stop strategy based only on two compounds: a hard and a soft one. Teams might decide either to have a faster getaway at the start of the race, opting to use the softer compound first, then switching to the harder compound to complete the race. Considering the fact that there is no refueling in F1, tires are most stressed by the weight of the car at the start of the race, when almost all the embarked fuel mass is still unburned. The aforementioned strategy would then allow a fast start, but the softer compound would degrade faster than if it was fitted later in the race, leading

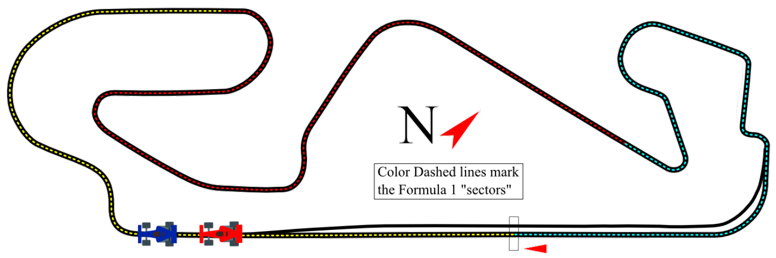
to a shorter stint duration. The opposite strategy (harder compound at the start, then pitting for the softer compound) would allow to resist the fuel-induced tire degradation better but would yield slow lap times in the first stint, especially at the start of the race. In this scenario, the team would hope to make up for the lost time in the final stint, being able to rely on a lighter car and on using the faster compound for longer due to the lower fuel mass remaining.
In this example, there is no clear winner between the two strategies: it largely depends on the track's features and the specific car's performance whether it would be better to have a fast start or a charging final part of the race.

The other two strategic options - undercut and overcut - are the opposite of each other. These are played when following a driver that is difficult to overtake on track due to the track's narrowness or to their sheer performance.
To perform an **undercut**, the pursuing driver attempts to stop before the driver that precedes him, taking advantage of the fresh tires to gain a significant amount of time in the out-lap (the lap that follows the pit-stop) due to the pace difference between the worn tire and the fresh one. When the driver in front stops, he will find himself behind the driver who performed the undercut tactic if he did not have a sufficient gap to defend from the undercut. Figure 4.1 shows the phases of the undercut tactic.
Figure 4.2, instead, reports an analysis of race pace for some of 2010, 2016, and 2017 compounds on the Barcelona-Catalunya track. The crossover point in the graph shows when a harder but fresh compound will become faster than a worn, softer one, which is the key to performing undercut.

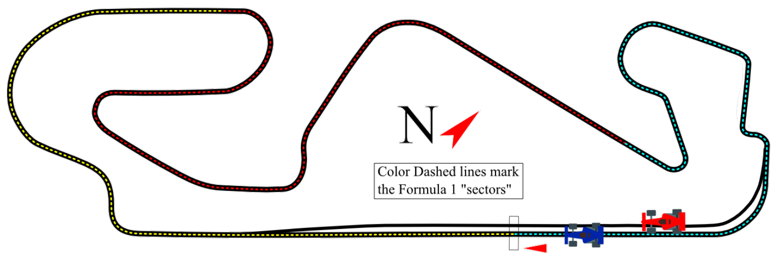
On the other hand, **overcut** acts on the opposite: the driver waits for his rival to perform his pit-stop, then pits a lap or more later. This tactic is seldom seen in modern Formula 1 as Pirelli tires are characterized by a high degradation rate, and the faster driver is usually the pitting first to fit fresh tires. However, on circuits with low degradation and cool temperatures, which makes the warm-up of fresh tires more complicated, the overcut can be used. The overcut tactic works when the driver manages to put in a really good lap just before stopping, while the rival struggles to get up to speed on his out-lap, meaning that the driver can emerge in front after the pit-stop.

A final strategy to be mentioned is **going long**: when the leading driver stops, the pursuer will extend his own stint on the current compound, knowing (or sometimes hoping) that performance is not expected to drop soon for the current tires and the other driver will not be matching his pace, allowing

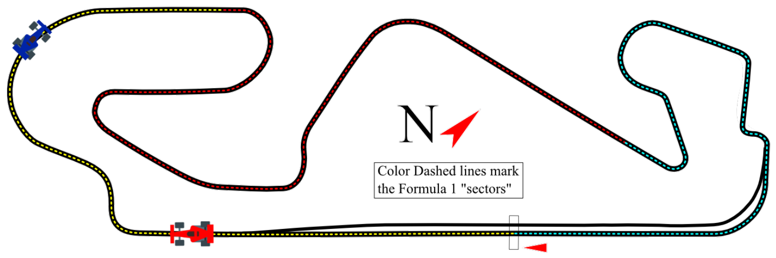
(a)



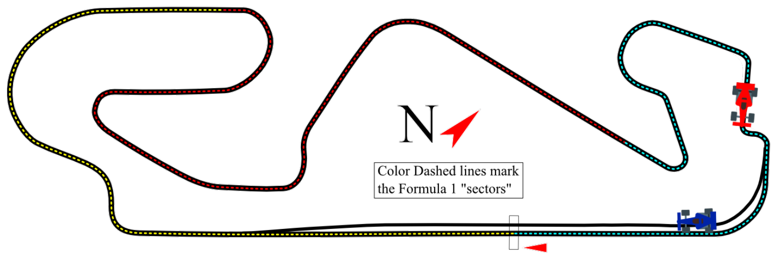
(b)



(c)



(d)



(e)

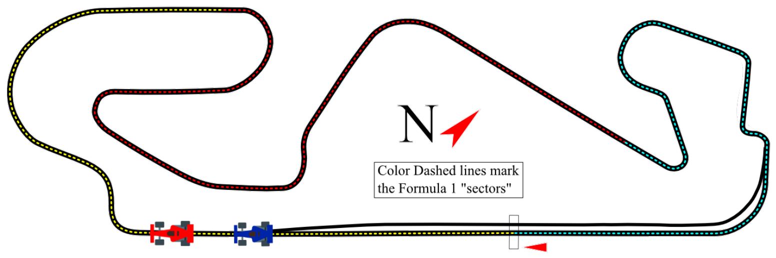


Figure 4.1: An undercut maneuver example. In subfigure (a) the red car is close enough to the blue car to try an undercut maneuver, which starts with pitting early, as shown in subfigure (b). Subfigure (c) shows the situation when the red car emerges from the pit-lane. The red car can exploit fresh tires to gain time against the blue car, which will eventually perform a pit-stop as depicted in figure (d). Finally, if the red car has gained enough time on the blue car, it will stay in front when the blue car emerges from the pit-lane.

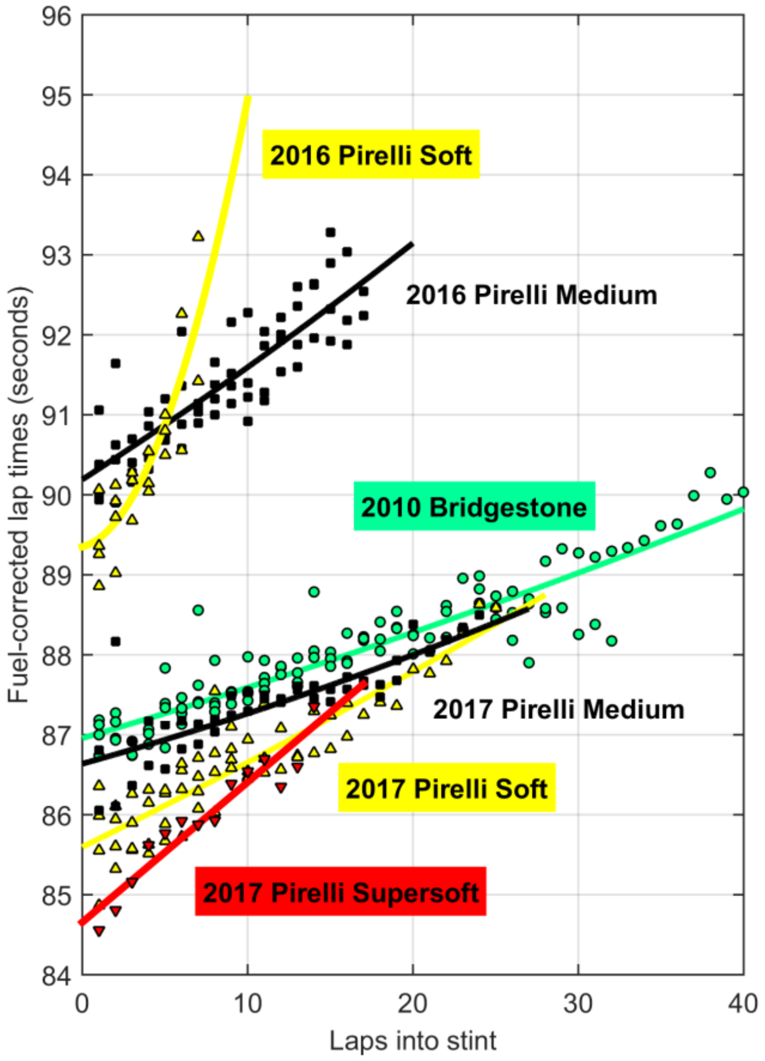


Figure 4.2: Comparison between different compounds' performance over time at Barcelona-Catalunya track. Taken from f1metrics' 2017 pre-season analysis [25]

to close the gap and re-join the track after his own pit-stop in front of the opponent.

Further strategic options may arise during a **Full-Course Yellow flag** (FCY). When exposed by the marshals, a simple yellow flag forbids overtaking, and drivers are required to lift the throttle pedal only in the affected track sector, whereas FCYs affect the whole track. An FCY is usually triggered by accidents on the track that force the race stewards to order the drivers to reduce their speed to maintain safety on the racecourse.
There are two types of FCY, **Safety Car** (SC) and **Virtual Safety Car** (VSC), which both correspond to a delta with respect to a target lap, set by the stewards before each race, that drivers must respect.
VSC is a digital safety measure, which requires drivers to autonomously increase their lap times to 140% of a reference lap. SC, instead, is a physical car that is sent on track to be followed by the drivers. Usually, the lap times of drivers behind the Safety Car increase to around 160% of the reference lap time.
Because all drivers are slowed down, a pit-stop may become particularly appealing as the time lost to change tires reduces during an FCY. This happens because the FCY deployment lowers the speed difference between the main straight and the pit-lane, thus reducing the amount of time lost with respect to competitors when performing the pit-stop.

In order to attain meaningful behavior in this setting, an autonomous agent for race strategy identification needs to take into account the factors discussed above, most notably tire compounds, their degradation, and FCYs. Moreover, taking into account these variables is also crucial to race time simulators designed for this setting in order to be able to provide a most realistic race simulation.

4.2 Problem representation

In Formula 1, pit-stops can be performed once per lap when the driver is about to complete his lap. To model the problem, we adopt in this work an MDP representation.
One issue in identifying a suitable time-discrete representation for the MDP lies in the fact that the race strategy problem is time-continuous: time gaps form between each driver pair as the race unfolds, and, as they can enter the pit-lane only at the end of the lap, they will perform the pit-stops in different time moments even if they are in the same lap. For instance, two drivers separated by a 15s gap performing a pit-stop in the same lap will start their pit-stop action 15s apart.

By considering the problem as a Sequential Decision Process, the time-continuous nature of the problem becomes tractable. Specifically, in this work, each time-step represents a single lap, and at each lap, the teams need to decide if they want to stop their drivers and which compound to fit.

For the race strategy determination problem, the first and easiest element of the MDP to define is the **action space**: we consider as many pit-stop actions as the compounds available to the driver, with each of these actions representing a pit-stop to fit the respective compound, plus a “stay on track” action.

Instead, the **state space** is harder to model for this problem, as many factors have to be taken into account to identify racing situations uniquely. The following features have been selected to describe the state of the race in the most compact way:

- **remaining laps**: the state records how many laps in the race remain to be completed.
- **driver ranking**: this feature represents the position that the driver holds at the start of the lap. It can be replaced by the cumulative race time: the driver with the lowest cumulative time is the leading driver, whereas the driver with the highest cumulative time is in the last position.
- **current tire-set**: the compound each driver is currently using.
- **current tire-set age**: in this work, degradation is modeled as a function of the number of laps a tire set has been used. Such modeling is beneficial because it is challenging to retrieve from the publicly available data [12] a precise tire degradation metric without knowing the asphalt abrasion level, the remaining fuel mass at each lap, and whether a driver is pushing or trying to save the tires (figure 4.3).
- **exposed flags**: the state stores if an FCY (be it a Safety Car or VSC) is currently deployed. Optionally also red flags (which interrupt the race) could be considered.
- **tire change rule already satisfied**: the state stores if each driver has already changed at least one compound. As previously stated, this constraint is prescribed by F1 regulations and results in disqualification from the race if not satisfied before the event ends.
- **remaining tire-sets**: the state includes information on how many tire-sets for each remaining compound a driver has left. By combin-

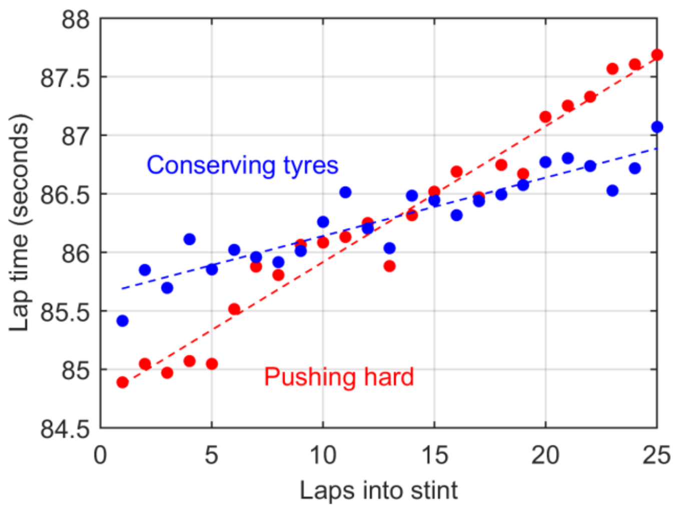


Figure 4.3: Compared lap times on the same compound when a driver is pushing or conserving tires. Taken from f1metrics' 2017 pre-season analysis [25]

ing this feature with the previous one, there is no need to store past compounds to identify a state uniquely.

- **retired driver**: the state contains a boolean feature representing if a driver is no longer participating in the race.

This MDP representation's **reward** can be constituted by lap times obtained by the controlled drivers, with a negative sign. As RL agents act as cumulative reward maximizers, a negative lap time reward will reinforce behavior that aims to minimize the cumulative race time.
Alternatively, the reward can be scaled in the range $[0, 1]$ by applying the formula:

$$r_t = 1 - \frac{\min(t_{lap}, t_{max})}{t_{max}} \tag{4.1}$$

where t_{max} is a reference lap time, which in this work is assumed to be 300 seconds. This assumption aims to obtain a reference time larger than the record lap time on the longest circuit in the F1 calendar, Spa-Francorchamps, and large enough to avoid clipping slow or FCY lap times. This circuit's record lap time is 1:46.286 (106.286s), which, corrected by the Safety Car delta, yields 170.058s as a possible FCY target lap and leaves a large margin for slower laps.
With this scaling, the reward is larger when the lap time is lower and vice-versa: maximizing the cumulative sum of the reward defined in equation 4.1 leads to the minimization of the cumulative race time, as higher rewards correspond to shorter lap times.

A good policy for the MDP should balance the cost of performing a pit-stop with the time gained by fitting fresh tires, obtaining a minimal total race time. On the other hand, this kind of behavior may lead to effects that are not modeled by the reward: an agent may perform pit-stop actions to minimize their race time but by doing so might either get stuck behind other drivers or allow competitors to overtake it when he is in the pit-lane, which otherwise might not have happened if it decided to stay on track. This kind of effect would correspond to the agent losing positions (and therefore championship points) in the final ranking, even if it did successfully minimize its total race time. A more principled approach would be to employ a reward function that weighs two different objectives, minimizing time and gaining positions, but accurately weighing these two contributions would not be straightforward and is beyond the scope of this work.

As an additional challenge, policies for the race strategy problem have a peculiar structure: most of the time, the agent should take the “stay on track action” and act differently only on few laps. This is because the time lost in a pit-stop is significant with respect to the time gained in single laps by using the fresh tires: typical human-designed strategies from 2017 on have just one or two pit-stops, whereas in 2015 and 2016, three or even four-stop races were possible due to a larger gap in performance between fresh and used tires, as can be seen by the degradation curves in figure 4.2.

An additional difficulty of this problem is that, at each lap, 20 drivers take actions almost simultaneously and with different goals. In the real-world setting, the race strategy engineers can clearly identify their competitors and devise or adapt strategies against their adversaries.
To allow an agent to do the same in an MDP framework, it is necessary to model the behavior of adversaries. There are two typical approaches to such modeling: either the other drivers are considered part of the environment, or they are considered autonomous agents. In the first case, other drivers will follow pre-defined behaviors, responding to or even ignoring actions from the agent representing the driver of interest. With the second approach, instead, other drivers are allowed to perform planning to determine their strategies, possibly using different planning algorithms than the one used by the agent of interest. Once the environment receives actions for all the drivers, it will transition to the next lap executing these actions.
When using the MDP representation, the agent needs to be able to provide a decision within some time margin before the driver passes the pit-lane entry; otherwise, he would not have time to react and enter the pit-lane for a pit-stop.

On the computational side, the setting presents a final difficulty: depending on the length of the track, F1 lap times are typically between one and two minutes, which means that the agent must be able to output a suggestion with a small computational budget. The use of anytime algorithms, like UCT, is particularly suited to this class of problems, ensuring to be able to receive a suggestion from the algorithm in the desired time frame.

Chapter 5

Open Loop Planning for Race Strategy

This chapter presents all the components and techniques constituting the solution devised to tackle the race strategy identification problem.
First, the simulation environment in which the planning occurs is presented; then, in the following section, the planning algorithm and the intuition behind it are explained. In the final section, the rollout policies used to provide a meaningful simulation of the race during the planning phase are described.

5.1 Simulation environment

In this work, we build a race simulation environment, starting from the simulator described in [13] and adopting some modifications to make it compatible with our planning setting.

5.1.1 Regression model for lap time prediction

To build a model for lap times prediction, we first investigate a supervised learning approach. We construct a dataset starting from publicly available data collected from multiple sources, on which we fit multiple regression models. We use the Ergast API[1] to collect most of the data, while data on the tires used during the race is collected from the RaceFans[2] website through an automated web scraper. Finally, since the data is not publicly available from the previously mentioned sources, we estimate the presence of an SC or VSC from data.

[1]`https://ergast.com/mrd/`
[2]`https://www.racefans.net/`

Dataset

The dataset contains 131527 rows, each one representing a lap of a driver during a specific race. The data ranges from 2014 to 2019 full championships, representing the so-called "Turbo-Hybrid" era of F1.

The dataset contains the following features:

- **race id**: represents the unique identifier of the race
- **circuit id**: a unique identifier for the circuit.
- **year**: the year the considered race happened.
- **round**: the race's ordinal identifier in the championship's race events calendar.
- **race length**: the total number of laps to be completed in the race.
- **driver id**: represents the unique identifier of the driver.
- **lap**: is the current lap for the considered driver.
- **position**: is the position held by the considered driver at the beginning of the considered lap.
- **milliseconds**: at lap l represents the time spent to complete the previous lap $l - 1$, expressed in milliseconds. For the first lap, the feature has a value of 0.
- **pit-stop count**: represents how many pit-stops the driver has performed.
- **pit-stop milliseconds**: if a pit-stop has been performed in this lap, this feature contains the time spent in the pit-lane, expressed in milliseconds
- **pit-stop**: this feature represents if the driver is entering or exiting the pit-lane in the current lap. It takes the value of -1 if the driver is entering the pit-lane, 1 if he is exiting it, and 0 if no pit-stop is being performed in the current lap.
- **R**: this feature represents a performance rating for the driver-car pair, computed on the previous and current season data, up to the previous race.

$$R = R_{team} + R_{driver}$$

For both the driver and the team, the formula for computing the partial contribution is:

$$R_{team/driver} = \frac{\sum_{i=0}^{N-1} e^{-i} \cdot x_i}{\sum_{j=1}^{N} e^{-i}} + \frac{\sum_{j=1}^{M} y_j}{M} e^{-N},$$

where N is the number of races from the beginning of the season up to the race previous to the current one (so $N = 0$ for the first race of the season), M is the number of races in the previous season, x_i is the number of points earned in the i-th race of the current season, and y_i is the amount of points earned in the i-th race of the past season. The variables x and y contain the points scored by either the driver or the team, depending on if R is computed for the driver or the team, respectively. In the first term, e^{-i} is used to weigh the race results, giving more importance to the more recent races with respect to less recent ones. The second term, instead, is the product of the mean of scored points in the past season with a weight, e^{-N}, which reduces the previous season's contribution to the rating as more races of the current season are completed. Notice that the first summation is computed by going backward starting from the current race, so x_0 is the race just before the current one, x_1 is the second race before the current one, until the first race of the season. In the worst case, the feature has a value of 0 with neither the driver nor the team having scored any point in the two seasons. On the other hand, the maximum value of R is achieved when in all the considered races the driver wins (scoring 25 points per race) and his teammate finishes second (scoring 20 points per race), yielding a maximum value of $(25+45) \cdot (1+e^{-N})$.

- **FCY**: this feature expresses if the current lap time is affected by a Safety Car or Virtual Safety Car. It has been estimated from data by flagging as FCY-affected those laps (excluding the starting lap) whose average lap time over all drivers was at least 120% of the average lap time computed over all drivers and the entire race.
- **rainy race**: this feature is a flag expressing if the race has been impacted by rain. This feature has been extracted from a wet race list found on a Reddit post[3], then checked by hand.
- **battle**: is a flag representing if a driver is currently involved in a battle with another driver. We considered a pair of drivers in a battle if their gap was less than two seconds.

[3]`https://www.reddit.com/r/formula1/comments/g0plkr/list_of_wet_weather_races_and_wins_by_driver/`

- **next lap time**: at lap l is the time spent by the driver to complete it, expressed in milliseconds. It represents the **target value** for the regression model.
- **fitted tire**: the tire compound currently fitted on the car.
- **tire age**: the number of laps a tire-set has been fitted in the race. The feature acts as proxy for tire degradation measurement, which is otherwise impossible to estimate directly from the publicly available data. We assume all tire-sets are fitted fresh, as no public information is available about the tires' previous usage during qualifying sessions.
- **starting position**: is the position of the grid in which the driver starts the race. This feature is useful mainly for the first lap's time prediction.
- **qualifying time**: is the best lap time obtained by the driver through the qualifying sessions.
- **pole position lap time**: the lap time of the driver who qualified in pole position.
- **cumulative race time**: the driver's total race time at the beginning of the current lap.
- **gap from the car in front**: as the name suggests, this feature tracks the time gap from the driver preceding the considered driver. It is set to 0 if the driver is leading the race
- **gap from the following car**: as the previous feature, this feature tracks the time gap between the considered driver and the car that follows him. If the driver is in the last position, this feature is set to 0.
- **lap time of the car in front**: at lap l represents the time spent by the preceding driver to complete the previous lap $l-1$, expressed in milliseconds. It is set to 0 if the considered driver is leading the race.
- **lap time of the following car**: at lap l represents the time spent by the following driver to complete the previous lap $l-1$, expressed in milliseconds. It is set to 0 if the considered driver is last in the race.
- **delta gap car in front**: at lap l is computed as the difference between the gap with the car in front at lap $l-1$ and lap $l-2$, expressed in

milliseconds. It is set to 0 if the considered driver is leading the race. The feature captures if the considered driver is closing or losing ground to the driver in front.

- **delta gap following car**: at lap l is computed as the difference between the gap with the following car at lap $l-1$ and lap $l-2$, expressed in milliseconds. It is set to 0 if the considered driver is last in the race. The feature captures if the chasing driver is closing or losing ground to the considered driver.
- **DRS**: after the second lap of the race, if the driver is within a one-second gap from the car in front and no FCY is currently active, he can use DRS and the feature assumes the true value, otherwise false.
- **previous milliseconds**: at lap l represents the lap time spent by the driver to complete lap $l-2$ the previous lap's time, expressed in milliseconds.
- **previous milliseconds delta**: at lap l represents the difference between the lap time at time $l-1$ and $l-2$. the goal of this feature is to capture a possible improvement or worsening trend for lap times.
- **tires gentleness**: is a constant feature, specific to each driver, representing how much a driver can extend his stint on the same compound with respect to the average stint on the same compound in the same race. For driver d, the feature is computed from data, excluding safety car laps and rainy races, as the ratio of the mean stint length for the driver and the mean stint length of all drivers.
- **lap constance**: is a constant feature, specific for each driver, which captures how much a driver can maintain lap times similar to each other, with respect to the average case. To compute it, for each driver, we first compute the standard deviation for the lap times in each race, then average it.
- **wet qualifying**: this feature represents if the qualifying event was held in wet conditions. The feature has been estimated from data, comparing race lap times with the pole-position time. As qualifying times represent (not considering mistakes) the fastest times each car can set on the track, if the race lap times are lower than the pole-position time, the qualifying was certainly held on a wet track.

We discard from our dataset (allegedly) suspended races by using a lap-time threshold approach. Furthermore, we also discard rainy races and races in which the qualifying was held in wet conditions, using the related dataset features. The final laps of each race are discarded as well since there is no need to predict any time when the race ends. Finally, we discard rows corresponding to each race's first lap since, having a high lap-time value caused by several battles among nearby drivers, they represent outliers to the dataset. Moreover, performing pit-stops in the first laps of the race is not relevant unless accidents or weather changes happen, so we can afford to start the simulation later.

After eliminating the rows listed before, we obtain a reduced dataset composed of 109865 rows. The dataset has then been split into three different sub-datasets, each one identified by the racing situation:

- **Regular laps**: no safety car, the driver is not entering nor exiting the pits. It contains 98989 rows.

- **Safety laps**: safety car has been deployed, but the driver is not entering nor exiting the pits. It contains 4965 rows.

- **Pit laps**: the driver is either entering or exiting the pits, regardless of whether the safety car is on track or not. It contains 5811 rows.

Regression models

To tackle the lap time prediction problem, we evaluate multiple regression models. We describe the training and testing datasets, provide a quick overview of the algorithms, and finally present the best-performing architecture and results.

As a simple baseline, **linear regression** is evaluated first but, as expected, does not provide accurate results, with a mean average error of more than 0.8s over the regular laps test set. To try and obtain more accurate predictions, we evaluate both **random forest** [4] and **XGBoost**-based [8] regression models. As a bonus, the former is also useful to provide a feature ranking during the feature engineering phase for the dataset. Finally, we evaluate a regressor based on Neural Networks, but it does not provide a performance improvement with respect to the other models.

When placed into an environment requiring thousands of samples, the model prediction time is critical. Considering this factor, we decide to deploy the XGBoost model into the environment, which ranked second-best (by a small margin) in prediction performance but provided an advantage of an order of magnitude in prediction time.

Training procedure and results

We train three XGBoost models, one for each of the datasets we described before. We split each dataset into a training and testing set, following a canonical 80-20 split approach. The model's regression target is the difference between the current lap time and the pole position qualifying time. We then normalize both the input and target data using a Min-Max scaling procedure:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

To allow quick prototyping of the race strategy environment, we train the models with the default parameters provided by the Python module XGBoost[4].

The results we obtained on the training and testing set are reported in figure 5.1. The left-hand side figures show the target-prediction scattering for the test sets of each dataset, with points closer to the bisector line indicating a better prediction performance. Note that the axis scale is different among the graphs, and, especially in the second and third row, the absolute error of the model is significant even if the distance from the line appears small in the plot. The right-hand figures, instead, compare the fitted distribution of the target data (labeled *gt*) against the predictions' distribution.

Table 5.1 reports the performance of the models on both training and testing sets. Considering data from both this table and figure 5.1, it is clear that the "safety" model exhibits a larger prediction error due to the variance of predicted data, whereas the "pit" model seems to suffer from outlier predicted data. Moreover, both the "pit" and "safety" models show a significant increase in prediction error when comparing the train and test set, as opposed to the "regular" model. The more considerable step in prediction error can be attributed to two main factors: first, the number of samples for the training datasets is scarce, inducing the model to overfit them, and secondly, the target data is bimodal, as opposed to the unimodal "regular" dataset.

Considerations

Despite the results obtained by the "regular" model in the test set, we observed that the model prediction quality rapidly degrades when fed sequentially with its own predicted data, as is the case for our race strategy environment. We analyzed the lap time difference between the fastest and

[4]https://xgboost.readthedocs.io/en/latest/

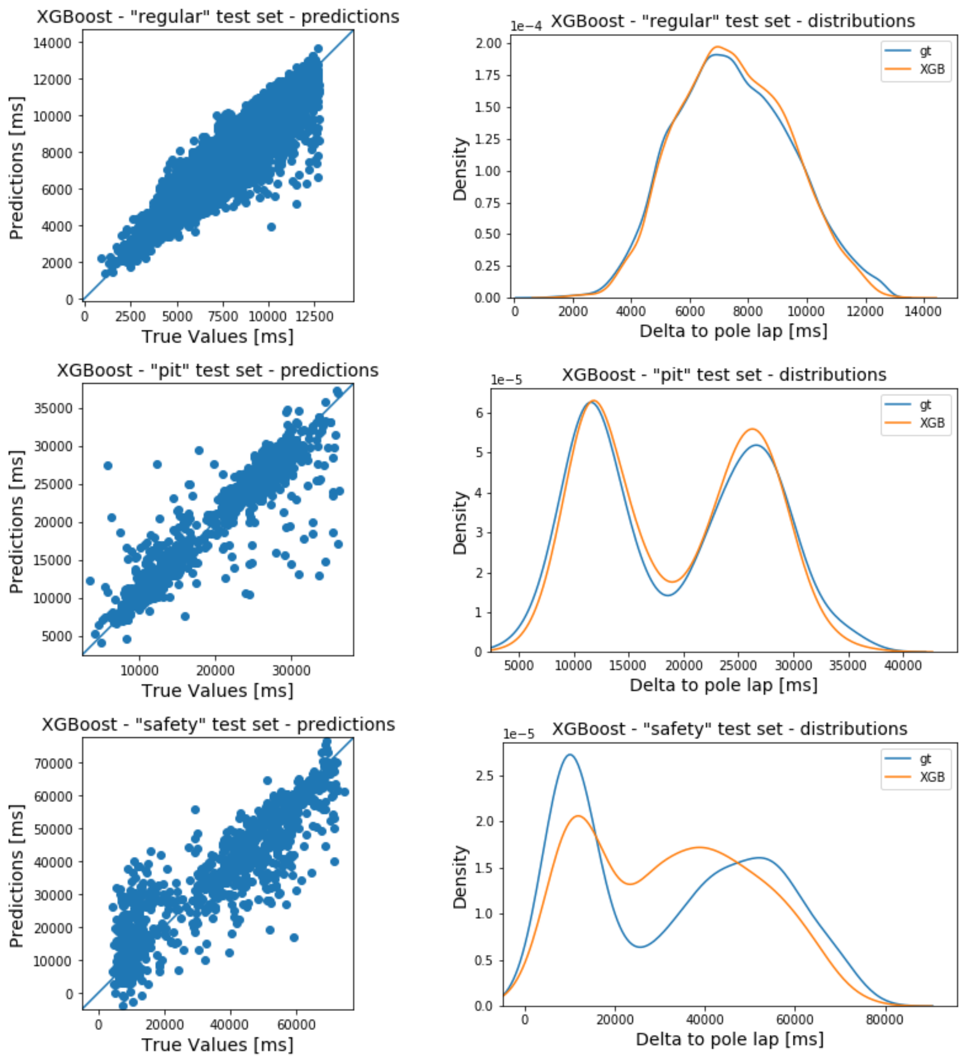


Figure 5.1: Scatter plots and distributions for the ground truth and predicted values of the XGBoost regression models.

Dataset	Train Error [ms]	Test Error [ms]
regular	393.08	449.34
pit	483.81	1635.06
safety	1691.17	6421.07

Table 5.1: Mean Absolute Error (MAE), expressed in milliseconds, for the three XG-Boost regression models.

slowest cars on track at each lap and observed that the delta for "regular" laps in some races went over 30s. In contrast, consulting `F1.com`'s race pace analysis articles for the affected races and further analyzing our dataset, we discovered that a typical delta between such cars should be approximately between 2 and 5 seconds, depending on the relative tire compounds fitted on the cars.
Moreover, we observed from the simulation logs that the cars could not keep somewhat constant lap times throughout the race, spanning over ten seconds in some cases. In real F1 races, during a stint, consecutive lap-times for the same driver tend to be almost constant and differ (depending on his skill) by few tenths of seconds, under the assumption of excluding errors, pit-stops, and FCY situations. Therefore, the resulting environment does not penalize correctly unwise strategies (for instance, performing a pit-stop more than 5 times) due to the randomness of lap time predictions happening among all participants. Furthermore, in some cases, this causes (as an extreme example) drivers starting last to win the race and drivers starting first to get somewhere around 15th place.

These unsatisfactory results led us to consider a different approach to the environment's transition model. In order to obtain a consistent lap time simulation and ease the interpretation of results, we resort to a probabilistic lap-time description with clearly defined time contributes.

5.1.2 Race time simulator

The simulator from [13] provides a parameterized and probabilistic description of the race allowing control over probabilistic race events such as accidents, Safety Cars (SC), or Virtual Safety Cars (VSC). Each of the simulator's internal events relies on probabilistic models, whose parameters have been optimized to provide a plausible replica of past races, using publicly available data [13] on tire strategy and events. Since the decision space is whether to perform a pit-stop on each lap, a lap-by-lap simulator allows us to model temporal transitions between states in an MDP setting.

Lap time computation

The race is modeled using a lap-by-lap approach, as shown in figure 5.2, in which the race time of each driver in the simulation is computed at each lap. The single-lap time t_{lap} is computed by the simulator by summing time

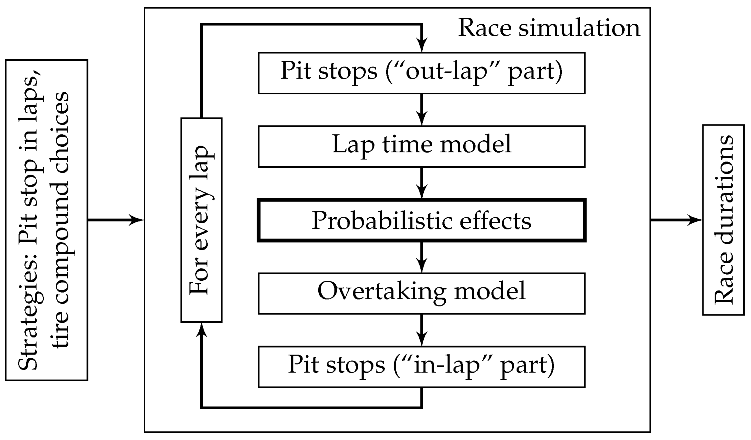

Strategies: Pit stop in laps, tire compound choices
Race simulation
For every lap
Pit stops ("out-lap" part)
Lap time model
Probabilistic effects
Overtaking model
Pit stops ("in-lap" part)
Race durations

Figure 5.2: The racing simulator's original workflow, taken from [13]

malus contributions to a fast reference lap

$$\begin{aligned} t_{lap}(l) = t_{base} + t_{tire}(a_{tire}, c_{tire}) + t_{fuel}(l) + t_{car} \\ + t_{driver} + t_{grid}(l) + t_{pit,inlap/outlap}(l), \end{aligned}$$

where:

- t_{base} is a reference lap time representing a perfect lap set by the fastest car on ideal fuel and tire conditions. In the simulator, such lap is modeled by taking the fastest qualifying lap time, as, during the qualifying session, the cars run on fresh tires and embark on just enough fuel to complete the flying lap and return to the pit-lane.

- t_{tire} models lap time loss due to tire wear. Racing tires behavior, as the authors report in [13], goes through different performance phases. First, the tire needs to warm-up, then reaches a short peak performance phase, followed by an almost constant decay in lap time performance before the tire completely degrades at the end of its life span. In the simulator, this behavior is simplified using a parameterized linear model, which is a function of the age a_{tire} and the tire compound c_{tire}:

  $$t_{tire,lin}(a_{tire}, c_{tire}) \cdot k_{2,lin}(c_{tire}) + k_3(c_{tire}),$$

  where $k_{2,lin}$ and k_3 are constants estimated for each driver and each race, as they vary significantly with track features, driving style, and vehicle balance. Quadratic and cubic models are also provided with the simulator but proved to be unreliable when applied to strategies diverging from the real ones.

- t_{fuel} represents lap time loss due to the embarked fuel weight. With respect to the ideal qualifying conditions, a Formula 1 car performs significantly worse at the start of the race due to the embarked fuel mass required to cover the race distance. As the race unfolds, the fuel gets burned, and the car becomes progressively lighter and, therefore, faster. The simulator models this effect with a simple linear model, disregarding uneven fuel consumption between different laps:

  $$t_{fuel}(l) = (m_{fuel,tot} - m_{fuel,consumed}(l)) \cdot s_{t_{lap},mass},$$

  where $s_{t_{lap},mass}$ is the lap time loss per kg of fuel embarked.

- t_{car} and t_{grid} are the lap time losses due to car performance and drivers' different abilities. The simulation treats them as constant offsets, estimated from qualifying sessions or previous races.

- t_{grid} models the time lost by drivers in different grid positions since cars at the end of the grid are further away from the starting line and need to cover more distance. Furthermore, this time contribution also models the fact that, since drivers are starting from a standstill, the first lap is slower than other laps.

  $$t_{grid} = p_{grid} \cdot t_{perGridPos} + t_{firstlap}$$

  Obviously, this contribution reduces to zero for laps other than the first one.

- $t_{pit,inlap/outlap}$ is the time loss due to pit stops, in-laps, and out-laps. An **in-lap** is the lap before a pit-stop is performed, with the driver entering the pit-lane to fit fresh tires. The **out-lap** is the lap in which the driver exits the pit-lane and needs to warm up the newly fitted tires. As the time is taken at the finish line, which is located slightly after the pit-lane entrance, the cars need to have already slowed down to 80 km/h (pit-lane speed limit), and the in-lap time is affected by a few seconds. The out-lap time, instead, is more significantly affected because the driver needs to wait in a standstill while the tires are fitted and drive out of a longer pit-lane section before rejoining the track. These contributions vary with the track layout, and the pit-stop time is computed for each team from previous races' data.

Apart from t_{base}, all the contributions can be considered stochastic, sampling them from probabilistic models fitted by the authors on previous race data.

The cumulative race time for lap k is computed by the simulator as the sum of single-lap times plus the time lost by the driver in overtaking maneuvers:

$$t_{race,l} = \sum_{l=1}^{k} t_{lap}(l) + t_{overtaking}(l) \tag{5.1}$$

, with

$$t_{overtaking} = t_{DRSeffect} + t_{overtake,win/lose} \tag{5.2}$$

In equation 5.2, $t_{DRSeffect}$ is a bonus on the lap time given by the Drag Reduction System (DRS), which can be activated in Formula 1 only when the pursuing driver has a gap of less than 1s from the driver in front. The system makes the car faster on the straight, thus reducing the overall lap time. $t_{overtake,win/lose}$, instead, is a malus applied to both the winner ($t_{overtake,win}$) and the loser ($t_{overtake,lose}$) in the overtaking maneuver. Both drivers lose time while fighting for an overtake since their racing lines differ from the time-optimal racing line when they are in a position fight.
The overtaking happens when the comparison of the cumulative race time of two drivers, provided by equation 5.1, provides a gap smaller than a time $t_{gap,overtake}$, defined for each track.

Further details on the various time contributions are reported in [14], whereas a detailed description of the probabilistic models and the dataset used to fit them can be found in [13].

Racing events

The simulator also allows considering driver retirements and Full-Course Yellow (FCY) flag events. As we do not model retirements during this thesis work, the simulator's functioning in such cases is not presented here but can be found in [13]. It is still useful, in order to enable a better comprehension of the race simulation, to present how Safety Car and Virtual Safety Car events are handled in the simulator.

The simulator accepts as input a list of events, whose start and ending moments can be represented either by race progress expressed in laps (for instance, from lap 35 to lap 37) or by a specific time moment expressed in seconds from the start of the race (for instance, the safety car enters the track after 45 minutes of racing). Since events need to happen at the same time for each driver, indicating their starting moment through lap progress may potentially cause conflicts for **lapped** drivers. Lapped drivers are those drivers who get overtaken by the race leader, falling one or more laps behind. Considering a specific moment of the race, lapped drivers have completed fewer laps but have been participating in the event for the same time as other drivers. Therefore, indicating the starting and ending race time for each event represents a sounder approach: the simulator converts events from race progress to race time by using a pre-simulation of the race time, then at each lap, after computing each driver's race time, checks if the driver needs to be slowed down due to FCYs.

5.1.3 Racing environment for MDP modeling

On top of the racing simulator described in the previous section, we design an environment representing the complete MDP for the race strategy identification problem. Like the simulator, the environment represents the race with a lap-by-lap approach, accepting, at each time-step, actions for one or more controlled drivers and returning the lap time achieved by following the action. In the environment, actions represent either performing a pit-stop to fit one of the compounds available in the race or a "stay on track action". Moreover, pit-stop actions are distinct for each compound.
On top of this simple framework, the environment ensures, mostly by modifying actions available at each time-step, that constraints related to the F1 race strategy problem are satisfied. This approach allows keeping the implementation of the agent simple, as long as the environment can inform the agent of the actions available in each state. The constraints that were considered in the implementation are as follows:

- The planning starts at a defined race lap, whereas before that lap, the agent-controlled drivers take actions according to the strategy used in the real race. This behavior models the fact that, unless accidents or sudden weather changes, a pit-stop is not required in the first laps of a race.

- When controlled by an agent, a driver cannot perform more than two pit-stops in a single race. One extra pit-stop is allowed each time an FCY is deployed, but only while the FCY stays active on track. This constraint is reasonable as most of the considered period's races could have been completed with at most two pit stops unless accidents or weather changes.

- After performing a pit-stop, a driver cannot perform another pit-stop before five more laps have passed. For instance, if a driver pits at lap 18, he will be unable to pit again until lap 24. Since this is a reasonable behavior even in real racing (all compounds typically last more than ten laps), it is enforced in our environment to reduce the planning complexity and variance in the returns for the pit-stop actions.

- Each driver can choose fresh tires from a finite number of tire sets for each compound. In real races, each team reserves a specific number of tire-sets for each compound to the race. As such data is not easily obtainable for historic races, we assumed all drivers in the environment have tire availabilities as reported in table 5.2.
- As per F1 regulations, drivers must change at least one compound type during the race. If this does not happen, the environment forces the offending driver(s) to perform a pit-stop at the race's penultimate lap to satisfy such rule and avoid awarding penalties.

Available compounds	**Soft**	**Medium**	**Hard**
3	2	2	1
2	3	-	2

Table 5.2: Default tire availabilities for the race simulation environment. Note that the names of the columns refer only to the relative hardness of the available compounds and do not stand for the actual compounds with that name.

After actions for all controlled drivers are received, the computation of lap times is tasked to the race simulator [13].

5.1.4 Race simulator modifications

Slight modifications have been applied to the simulator from [13] to allow a high-level control of the drivers' strategy. Ordinarily, the race simulator would load strategies for all drivers from a configuration file and then run a simulation of the full event with fixed strategies. In order to comply with our planning simulation requirements, this behavior needs to be extended to support both manual lap-by-lap simulation and lap-by-lap strategy specification for controlled drivers.
In order to reach this goal, we first add a `step` function to the simulator, allowing the simulation environment to progress the simulation in the race simulator by one lap. Then, considering that the planning process starts from a specific lap and proceeds incrementally, we implement a function to clear future strategy data for the desired driver and added the possibility to specify a pit-stop for the next lap in the `step` function.

Secondly, since it is clear that the simulation of FCY events is crucial to obtain useful recommendations from the agent within this context, we modify how the Full Course Yellow (FCY) events are generated and handled. In its original implementation, the simulator handles the generation of random events according to real race data, which are stored inside the simulator itself and loaded at the beginning of each simulation. From the planning agent's point of view, even though this information is not included in the state, in different simulations of the same race performed during planning the events would always occur at the same time, consistently. This would mean that the planner would be able to "predict the future" and prepare for the events beforehand. Therefore, we modify the behavior either to generate events during the race either with a stochastic model or to add the real race's events only at runtime. In particular, we allow the simulator to accept new FCY events to be added at runtime one lap after they have been requested. This way, we ensure that FCYs in Monte-Carlo simulations appear only when they would be stochastic events in the race. The duration of the FCY event can either be pre-specified, assuming a human operator evaluates the race situation and enters the expected duration into the system, or it can be generated at random, to simulate multiple scenarios during the search.

5.2 Open-Loop Planning for Race Strategy

In this work, we employ an open-loop approach to tackle the race strategy problem, searching at every lap of the race for the best race strategy to use for the rest of the race. We favor the open-loop strategy over the alternative progressive-widening (PW) [9] because of the additional memory load required to employ PW, stemming from the copy of the internal state of the simulator to each node of the search tree. As the simulator tracks numerous variables for each driver and additional ones for the race, if a full copy of the simulator state is performed at each node, the memory requirements quickly grow for deep enough trees. An approach to tackle this problem would be building a compacted representation of the simulator state, but that proved to be cumbersome in practice and is outside the scope of this work.

5.2.1 Open Loop Planning

In the open-loop setting, the problem is shifted from finding the optimal policy (mapping from states to actions) to finding the optimal sequence of actions to perform starting from the current state, regardless of the intermediate states visited, and instead averaging between them. Clearly, when the

MDP transitions are deterministic, the open-loop and closed-loop settings are equivalent.

More specifically, given a starting state $s \in \mathcal{S}$ and a sequence of actions $\tau = (a_0, a_1, \ldots, a_m), a_i \in \mathcal{A}$, the value of the sequence τ starting from the state s is defined as:

$$V_{OL}(s, \tau) = \mathbb{E}\left[\sum_{t=0}^{m} \gamma^t r_t \;\middle|\; s_0 = s, a_t \in \tau\right].$$

We note that τ can be an infinite sequence if $\gamma < 1$. Accordingly, the open-loop optimal value is given by maximizing over the sequences of actions:

$$V^*_{OL}(s) = \max_{\tau} V_{OL}(s, \tau).$$

We also define the open-loop action-value of a state-action pair, (s, a), as the maximizer over the possible action sequences τ that start with a, denoted with τ_a:

$$Q^*_{OL}(s, a) = \max_{\tau_a} V_{OL}(s, \tau_a).$$

Since $Q^*_{OL}(s, a) < Q^*(s, a)$, planning in an open-loop setting leads to a loss of performance, but it simplifies the planning problem by limiting the size of the search-tree and may be beneficial in applications with small search budgets.

5.2.2 Proposed algorithm

Our algorithm formalization starts from an open-loop setting of UCT. In this work, $\mathcal{T}$ denotes a planning tree and $\mathcal{N}_{d,i}$ denotes the i-th node at depth $d \geq 0$ for $i \in \mathbb{N}$. $\mathcal{N}_{0,0}$ contains a single state, $s_0 \in \mathcal{S}$, from which the agents start performing the planning. Nodes $\mathcal{N}_{d,i}$, with $d > 0$, at deeper levels of the tree, represent the distribution of states given the sequence of actions from the root of the tree to $\mathcal{N}_{d,i}$. More specifically, given a sequence of d actions, $\tau_{d,i} = (a_1, a_2, \ldots, a_d)$, this sequence identifies exactly the node $\mathcal{N}_{d,i}$ in the tree $\mathcal{T}$, representing the distribution of states $s \in \mathcal{S}$ reachable by executing the sequence of actions $\tau_{d,i}$ starting from the root state s_0. The value of a node $\mathcal{N}_{d,i}$ is defined as

$$\mathcal{V}\left(\mathcal{N}_{d,i}\right) = \mathbb{E}_{s \sim \mathcal{P}(\cdot|s_0, \tau_{d,i})}\left[V^*_{OL}(s)\right],$$

and the value of an action $a \in \mathcal{A}$ in a node as

$$\begin{aligned}\mathcal{Q}\left(\mathcal{N}_{d,i}, a\right) &= \mathbb{E}_{s \sim \mathcal{P}(\cdot|s_0, \tau_{d,i})}\left[Q^*_{OL}(s, a)\right] \\ &= \mathbb{E}_{s \sim \mathcal{P}(\cdot|s_0, \tau_{d,i})}\left[r(s, a)\right] + \gamma \mathcal{V}\left(\mathcal{N}_{d+1,j}\right),\end{aligned}$$

where $\tau_{d+1,j} = (\tau_{d,i}|a)$ is the sequence of action derived from concatenating $\tau_{d,i}$ with a and $\mathcal{N}_{d+1,j}$ is the corresponding node in the tree.

The goal of the proposed planner is to estimate the open-loop values of the actions in the root of the tree by employing a UCT-like tree policy, which considers action selection at each node as a separate bandit problem and selects the action that optimizes an upper bound for $\mathcal{Q}$ values of the actions in the node.

The second modification that is added to the standard UCT scheme is the tree back-up strategy. UCT employs MC updates recursively up the tree after receiving the leaf-value estimates from the rollout policy. Since the rollout policy is suboptimal by definition (if an optimal policy for rollout were available, planning would not be necessary) and the selection policy in the tree includes the exploration and exploitation of intermediate value estimates, the back-up values include the evaluation of suboptimal policies, which change with each search iteration. This usually makes the value estimates in the tree very noisy, which is a problem in the race strategy problem as the margins for selecting a good pit-stop strategy are small compared to the race duration. For this reason, a TD operator is employed, namely the Q-learning operator [34] as follows:

$$\mathcal{Q}_t\left(\mathcal{N}_{d,i}, a\right) = (1-\alpha_t)\mathcal{Q}_t\left(\mathcal{N}_{d,i}, a\right) + \alpha_t \left(r_t + \gamma \max_{a'} \mathcal{Q}_t\left(\mathcal{N}_{d+1,j}, a'\right)\right), \quad (5.3)$$

where r_t is the reward observed in the current search pass at node $\mathcal{N}_{d,i}$ and α_t is the learning rate. In our algorithm, we apply a modified max operator: when all the children of a node have not been explored yet, for the Q-learning update of Equation 5.3, the max operator is applied considering only the visited nodes, disregarding the unexplored actions' $\mathcal{Q}$ value initialization.

As the application of TD updates in a MCTS setting has been studied in the literature, we include a comparison with other TD update strategies [32, 11] in the experimental setting.

Box 16 shows the pseudocode of the planner employed. At each lap in the race, multiple iterations of planning are performed until the planning budget is reached. At each search iteration, the UCB selection at each node of the tree is performed until a leaf-node is reached, then a rollout is performed using one of the rollout policies discussed in section 5.3. The rollout allows building an initial (noisy) estimation of the node value. Next, QL updates are recursively employed up the tree, updating the node and action values and counts. This means that the initial noisy back-up value given by the rollout, even though it is stored in the leaf node, might not make its way up to the root, since at each node the max operator is employed to define the

Algorithm 16 Q-Learning Open Loop Planning

```
 1: procedure OLSEARCH($s_0$)
 2:     Create root node $\mathcal{N}_{0,0}$ from state $s_0$
 3:     while within computational budget do
 4:         $\mathcal{N}_{d,i}, s \leftarrow$ TREEPOLICY($\mathcal{N}_{0,0}$)
 5:         $\mathcal{V}(\mathcal{N}_{d,i}) \leftarrow$ ROLLOUT($\mathcal{N}_{d,i}, s$)
 6:         BACKUP($\mathcal{N}_{d,i}$)
 7:     end while
 8:     return BESTCHILD($\mathcal{N}_{0,0}$)
 9: end procedure

10: procedure TREEPOLICY($\mathcal{N}$)
11:     while $\mathcal{N}$ not terminal do
12:         if $\mathcal{N}$ not fully expanded then
13:             return EXPAND($\mathcal{N}$)
14:         else
15:             $\mathcal{N} \leftarrow$ BESTCHILD($\mathcal{N}, C_p$)
16:         end if
17:     end while
18:     return $\mathcal{N}$
19: end procedure

20: procedure ROLLOUT($\mathcal{N}$, s)
21:     $\Delta \leftarrow 0$
22:     while s is non-terminal do
23:         Choose $a \in A(s)$ according to rollout strategy
24:         Generate next state $s'$ and reward $r$
25:         $\Delta \leftarrow \gamma\Delta + r$
26:         $s \leftarrow s'$
27:     end while
28:     return $\Delta$
29: end procedure
```

Algorithm 17 Q-Learning Open Loop Planning

```
30: procedure EXPAND($\mathcal{N}$)
31:     Choose $a \in$ untried actions from $\mathcal{N}$
32:     $s \leftarrow$ SIMULATEUNTIL($\mathcal{N}$)
33:     Execute $a$ in $s$ generating $s'$ and $r$
34:     Add a new child $\mathcal{N}'$ to $\mathcal{N}$
35:     $\mathcal{N}'.n \leftarrow 0$
36:     $\mathcal{N}'.r = r$                ▷ Store the reward obtained during first visit
37:     return $\mathcal{N}', s'$
38: end procedure

39: procedure BESTCHILD($\mathcal{N}, c$)
40:     $C(\mathcal{N})$ denotes children nodes of $\mathcal{N}$
41:     $C(\mathcal{N}, a)$ denotes the child of $\mathcal{N}$ corresponding to action $a$
42:     return $\text{argmax}_a\ \mathcal{Q}(\mathcal{N}, a) + c\sqrt{\frac{2 \ln \mathcal{N}.n}{C(\mathcal{N},a).n}}$
43: end procedure

44: procedure BACKUP($\mathcal{N}, V$)
45:     $C'(\mathcal{N})$ denotes explored children nodes of $\mathcal{N}$
46:     $\mathcal{N}' \leftarrow$ parent of $\mathcal{N}$
47:     $\mathcal{N}.n \leftarrow \mathcal{N}.n + 1$
48:     while $\mathcal{N}'$ is not null do
49:         if $\mathcal{N}$ is leaf then
50:             $\Delta \leftarrow V$
51:         else
52:             $\Delta \leftarrow \max_{a' \in C'(\mathcal{N})} Q(\mathcal{N}, a')$
53:         end if
54:          $Q(\mathcal{N}', a) \leftarrow Q(\mathcal{N}', a) \quad +$
                        $\alpha(\mathcal{N}'.r + \gamma\Delta - Q(\mathcal{N}', a))$
55:         $\mathcal{N}'.n \leftarrow \mathcal{N}'.n + 1$
56:         $\mathcal{N} \leftarrow \mathcal{N}'$
57:         $\mathcal{N}' \leftarrow$ parent of $\mathcal{N}$
58:     end while
59: end procedure
```

target value, as shown in the BACKUP procedure.

We omit the SIMULATEUNTIL procedure's implementation because it is trivial, but we shortly describe it. This procedure takes as input a node of the search tree and simulates all the transitions from the root state to the input node by executing the sequence of actions that identify that node. In the end, it returns the state of the environment resulting from the sequence execution, which is used later as a starting point for the rollout.

5.3 Rollout Policies

In the race strategy problem, during the planning phase, it is not desirable for an agent to be "clairvoyant" over the adversaries' strategies. Therefore, the possibility of specifying a planning configuration for the simulator to use when performing the tree search is needed to satisfy such constraint. In the experimental setting of this work, the goal was implemented by considering all drivers as controlled by the planner and applying a rollout policy (also called **default strategy**) for the opposing drivers. The rollout policy is a crucial component of MCTS-like algorithms, as it provides an initial estimate for the value of leaf nodes, which is then used during the back-up phase of such algorithms. Using a random policy would generate extremely noisy value estimates in the tree nodes; therefore, careful consideration of the rollout is needed.

The classic rollout policy used in MCTS algorithms is a random policy, choosing with an equal probability between actions at each time step. Since good policies for the race strategy problem have the structure discussed at the end of section 4.2, the random policy yields poor performance and doesn't provide meaningful information for the nodes' value initialization. To obtain more reasonable rollout estimates, we adopt an informed approach, building two rollout policies that better approximate human strategies.

The first approach we consider is a simple stochastic strategy: the driver, at each lap, has a 0.9 probability of staying on the track and a 0.1 probability of making a pit stop by randomly choosing one of the compounds available. As the strategies provided by this simple baseline are, for the most part, unreasonable for both the track situation and the tire degradation status in the simulator, a more realistic strategy definition is needed.

The second approach we explore focuses on leveraging the predicted race strategies publicly available in sports articles (see table 5.3) taken from the

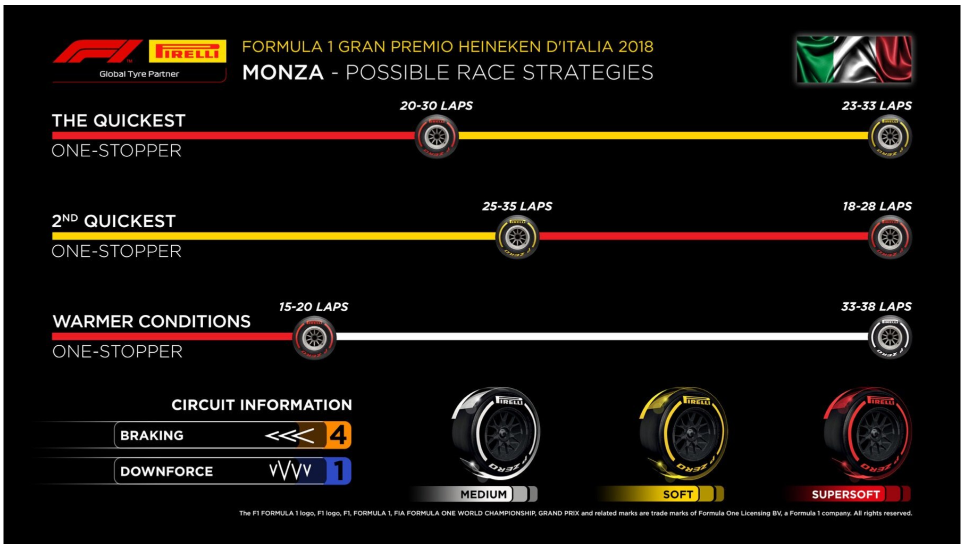

Global Tyre Partner
FORMULA 1 GRAN PREMIO HEINEKEN D'ITALIA 2018
MONZA - POSSIBLE RACE STRATEGIES
THE QUICKEST
ONE-STOPPER
20-30 LAPS
23-33 LAPS
2ND QUICKEST
ONE-STOPPER
25-35 LAPS
18-28 LAPS
WARMER CONDITIONS
ONE-STOPPER
15-20 LAPS
33-38 LAPS
CIRCUIT INFORMATION
BRAKING 4
DOWNFORCE 1
MEDIUM
SOFT
SUPERSOFT
The F1 FORMULA 1 logo, F1 logo, F1, FORMULA 1, FIA FORMULA ONE WORLD CHAMPIONSHIP, GRAND PRIX and related marks are trade marks of Formula One Licensing BV, a Formula 1 company. All rights reserved.

Figure 5.3: A sample of strategies predicted by Pirelli for the 2018 Italian GP, taken from `https://twitter.com/pirellisport/status/1036169705634054144`

Season	Track	Source
2015	Japan	`https://www.espn.co.uk/f1/story/_/id/13768566/japanese-grand-prix-strategy-briefing`
2016	Japan	`https://www.espn.co.uk/f1/story/_/id/17750221/japanese-grand-prix-strategy-guide`
2017	Australia	`https://www.espn.com/f1/story/_/id/19005027/australian-grand-prix-race-strategy-guide`
2017	Spain	`https://www.espn.co.uk/f1/story/_/id/19379342/spanish-grand-prix-strategy-guide`
2017	Austria	`https://www.espn.com/f1/story/_/id/27087391/austrian-grand-prix-strategy-guide`
2017	Belgium	`https://www.espn.com/f1/story/_/id/20473131/belgian-grand-prix-strategy-guide`
2017	Russia	`https://www.espn.com/f1/story/_/id/19273668/russian-grand-prix-strategy-guide`
2018	China	`https://www.espn.com.au/f1/story/_/id/23177650/chinese-grand-prix-strategy-guide`
2018	Italy	`https://www.espn.com/f1/story/_/id/24553859/italian-grand-prix-strategy-guide`
2018	Brazil	`https://www.espn.com/f1/story/_/id/25242197/brazilian-grand-prix-strategy-guide-race-pace`

Table 5.3: ESPN source articles used for building fictitious tire durabilities.

motorsport opinion website ESPN.[5]

Since the Monte-Carlo agents explore different and sometimes unreasonable strategic options, it would have been hard to automatically identify and re-map the agent's strategy to one of the predicted ones. To be independent of such mapping, we compute a reasonable amount of laps that each compound would last in a specific race, named **tire durability**, for each compound allowed in the race. Figure 5.3 shows an example of the manufacturer's predicted strategies, proposing different options to cover the race distance. Note that the same compound can be used in stints of different lengths, depending on the sequence of tires included in the strategy and the position of the stint in the sequence. This may be due to the fact that, in the

[5]`https://www.espn.com/f1`

real world, tire degradation is higher in the early stages of the race since less rubber has been deposited on the asphalt, and the cars have more weight due to the almost entirely unburned fuel mass. In practice, this means that the same compound will last longer if fitted towards the end of the race, being less stressed during its working period. Furthermore, a compound usually wears out faster if the driver pushes for faster lap times, leading to a shorter stint duration.

To compute tire durability, we first average the duration for stints with uncertain duration, then we average the duration of stints on the same compound (but belonging to different strategies). However, after applying the second averaging, some of the original strategies are no longer obtainable: one or more compounds would reduce their maximum durability and therefore leave a part of the race uncovered, requiring a further pit stop. To address this problem, we extend the durability of the hardest compounds to cover the missing laps, relying on the empirical observation that they show a slower lap-time degradation than the softer ones and, therefore, extending a stint for a few laps would have less impact than doing so on a softer compound.

The rollout policy that takes advantage of these durabilities works as follows. When the current tire set has reached its expected duration, the pit stop is performed with a probability of 0.9, or the decision is deferred by one lap with a probability of 0.1. The choice of the next compound to use is deterministic: if there are any compounds whose durability would cover the remainder of the race, the policy suggests fitting the softest one among them. If there is no such compound, the softest compound available is fitted instead. Finally, to meet F1 regulations, the policy ensures that each driver switches to a different compound at their first pit stop.

Chapter 6

Experimental setting and results

Our experiments focus on a specific driver, Sebastian Vettel, and on a specific time frame, from 2015 to 2018. This decision is backed up by the fact that Vettel, who was driving for Scuderia Ferrari at the time, was a close contender for the championship title in 2017 and 2018 and had a good performance in the remaining years. Furthermore, we can recall that Scuderia Ferrari made some strategic mistakes in some races during this period, so the goal of our experiments for these races is to check if an online planning agent would avoid them. As the race simulator presents some deviations from the historical data, we simulate each race 1000 times and average each driver's final race time to generate the baseline performance of the true strategies employed by the drivers.

In our work, only dry races are considered due to missing data, which would be necessary to simulate wet conditions correctly. For instance, track surface wetness and the related grip level [13] at each lap are crucial to predict meaningful lap times in a simulated race.
For our experiments, we select a sample of 9 races among those where the average gap between Vettel and the driver in front was less than 10 seconds (see Table 6.1). This selection criterion is aimed at finding races in which, with better strategic decisions, it would have been possible to bring the driver to the front. As the authors are Ferrari fans, we add the 2017 Spanish GP race to the race list, as it hosted a spectacular battle between Hamilton and Vettel, and we feel that the Scuderia's strategy could have been stronger on that event, bringing the number of considered competitions to 10.

For each race, we perform 100 experiments with each of the following planners: Sarsa UCT [32], Open-Loop UCT [22], PowerUCT [11], and our

Season	**Track**	**Laps**	**SC**	**VSC**
2015	Japan	53	No	No
2016	Japan	53	No	No
2017	Australia	58	No	No
2017	Spain	66	No	Yes
2017	Austria	71	No	No
2017	Belgium	44	Yes	No
2017	Russia	53	No	Yes
2018	China	56	Yes	No
2018	Italy	53	Yes	No
2018	Brazil	71	No	No

Table 6.1: List of the races used for evaluating planners' performance in the experiments. The laps column reports the real number of laps for each race

agent Q-learning OL UCT. In fact, we also considered an evaluation of a PW-UCT algorithm for the evaluation campaign, but the implementation quickly ran out of memory during the tree search, even for small search budgets. Furthermore, we perform the same amount of experiments using VSE [15], a neural-network-based agent designed specifically for automated F1 pit-stop decisions.

For our QL OL-UCT, we employ an exponentially decaying learning rate, computed as

$$\alpha = \frac{1}{b + N^a},$$

where a and b are constant parameters controlling the schedule curve shape, and N is the number of visits to the node which is currently updated.
We start the strategy planning from lap 8 and set the discount factor γ to 1 to make the agent focus on the cumulative reward, as, in F1 racing, points are awarded based on the final standing. The computational budget of our experimental setting represents the maximum number of samples that the agent can take from the environment. We set the computational budget to 10,000 samples. The time horizon for the problem is variable between each race, as it corresponds to the total number of laps prescribed for each racing event minus the starting lap. Table 6.1 reports the selected races and the respective number of prescribed laps.

We tune each planner's hyperparameters on the 2017 Australian GP race, running the Bayesian optimization framework Mango [27] for 30 optimization iterations and employed the same hyper-parameters in each race.

Sarsa UCT	Power UCT	OL UCT	QL-OL UCT
$c = 30$	$c = 4.7$	$c = 30$	$c = 50$
$\lambda = 0.1$	$p = 150$		$a = 0.2$
			$b = 1$

Table 6.2: Hyperparameters used for each planner.

The hyper-parameters assignments for each planner are reported in table 6.2

We consider the 2017 Australian GP a suitable choice for hyperparameter tuning because of two main factors. First, the real race was a close fight between Vettel and Hamilton, which ultimately Vettel won. Secondly, we select this race because it had no accidents and no subsequent FCYs so that the optimizer would not look for parameters correlated to specific racing situations.

Table 6.3 reports the results for our experiments, comparing the undiscounted cumulative return for both the planners mentioned at the beginning of this section and two baselines. The ESPN baseline value is obtained by applying the *default* strategy prescribed by the environment, whereas the *true* baseline corresponds to the result obtained by applying the strategy used by the driver in the real race. We show in bold the best planner (between the UCT variants) in each race, but only when the difference between the best and second-best satisfies a statistical significance test. We do not have a best-performing planner only for the China 2018 race since the performance of QL-OL UCT and SARSA UCT are close when compared to their confidence interval. Moreover, with the symbol $*$, we denote the best strategy between planners and baselines (with statistical significance).

The results we obtained show that our proposed planning algorithm per-

Season	Track	ESPN	True	VSE	Sarsa UCT	Power UCT	OL UCT	QL-OL UCT	Ranking Gain
2015	Japan	4576.01±1.0	4577.52±1.0	4575.34±1.3	4575.36±1.2	4583.25±2.4	4577.99±1.1	**4570.35±1.0***	0.4
2016	Japan	4507.85±1.0	4507.54±0.7	4549.01±1.3	4508.90±0.8	4524.45±1.2	4519.03±1.3	**4505.35±0.9***	0.1
2017	Australia	4470.39±1.7	4466.22±1.9	4477.29±1.3	4466.56±2.9	4474.12±2.2	4479.90±2.2	**4459.71±2.4***	-1.3
2017	Spain	5202.42±1.4	5209.89±1.3	5207.94±1.1	5196.38±2.1	5200.83±2.0	5211.05±1.1	**5188.05±1.3***	0.1
2017	Austria	4525.88±1.2	4430.84±1.7*	4491.66±1.9	4476.43±2.8	**4444.38±2.4**	4484.14±1.8	4465.85±2.9	-2.4
2017	Belgium	4265.4±0.7	4256.44±1.0	4236.0±0.6*	4255.98±0.7	4259.52±0.7	4260.24±1.0	**4246.09±0.7**	2.9
2017	Russia	4419.98±1.3	4412.87±1.2*	4428.62±2.4	4425.10±2.1	4437.00±1.3	4430.93±1.7	**4421.54±1.3**	0.0
2018	China	5140.7±0.9	5134.01±1.0	5095.34±2.0*	5099.33±1.5	5128.63±0.9	5113.31±2.6	5098.93±1.5	4.2
2018	Italy	3909.37±1.9	3898.38±1.9*	3943.95±1.9	3907.22±1.4	3918.42±1.3	3911.24±1.3	**3903.67±1.5**	-0.3
2018	Brazil	4678.24±2.1*	4700.36±2.1	4711.61±1.7	4692.7±3.1	4699.25±1.7	4706.94±1.5	**4686.32±2.9**	2.0

Table 6.3: Cumulative return comparison for the experimental setting, lower is better. Bold stands for best performance among planners, star stands for best overall race time.

forms better in most races than other planners. Furthermore, the planner is able to improve the average race times on most occasions with respect to the real strategy performed by the drivers. All planners fail to do so in three races: Italy 2018, Russia 2017, and Austria 2017. In particular, the Austria 2017 race shows the larger gap between real strategy and planner performance. Our a-posteriori analysis found that the predicted ESPN strategy was highly inconsistent with the real strategies applied during the race: the article considered more tire wear than in the race, almost halving some of the durability of the compounds. Therefore, we attribute this performance loss to the suboptimality of the rollout policy, which can be addressed by increasing the computational budget: in this specific race, the performance of our proposed algorithm improved by around 10 seconds by increasing the computational budget to 100,000, whereas a parameter tuning for the race did not provide significant performance gains.

Since Austria 2017 represents a particularly difficult problem for the evaluated planners, we also deploy provably efficient algorithms, specifically OLOP and KL-OLOP, with a budget of 10,000 to assess their performance. The results obtained by these planners are in line with those of QL-OL UCT but the computational time is significantly higher. We, therefore, do not extend the evaluation campaign to other races or to higher budget, since these algorithms are not anytime and wouldn't allow any practical application with meaningful budgets.

In figure 6.1, we show return as a function of the employed search budget in the races in which our planner did not obtain the best performance. It is evident from the plots that the planner benefits from the increasing budget but is still not able to match the performance of the real strategy. This is especially evident for the Austria 2017 race, where the gap to the real strategy remains around 25s even with a budget of 100.000. Higher budget amounts than those reported in the figure could have allowed further improvement of the performance, but have not been considered because the computation time would have become unreasonable on our infrastructure.

A final remark is that, in most of the considered races, autonomous agents are able to improve or maintain the final position of the real driver. The last column, labeled *Ranking Gain* shows the average ranking improvement achieved by our planner compared to the position achieved by the true race strategies of the race. An interesting case where this does not happen is represented by the 2017 Australian GP, in which our proposed algorithm is able to improve the cumulative race time but loses positions in the final placement. This is most likely an artifact generated by the reward structure: since the reward promotes behaviors that minimize the cumula-

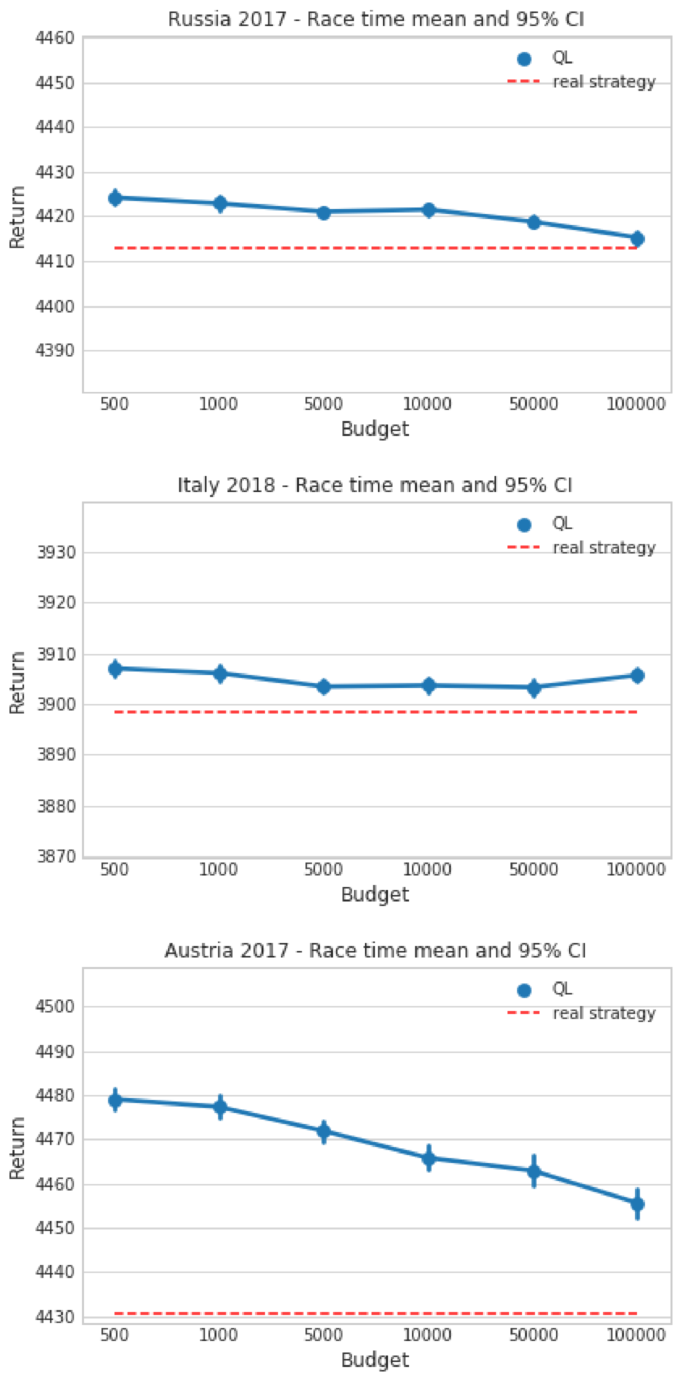


Figure 6.1: Race time as a function of computational budget for Austria 2017, Russia 2017, and Italy 2018 races, lower is better. The plot shows mean and 95% confidence interval for different budget values. The red dashed line represents the baseline for the return obtained by applying in the environment the real strategy of Vettel.

tive race time, the agent does not consider penalizing to take actions that allow its competitors to overtake it, as long as this allows to maximize the reward, suggesting the need for a carefully thought-out reward function that incentivizes fast laps and position gains, while still providing a dense reward signal to the agent.

Chapter 7

Conclusions

Summary

In this work, we investigated how MCTS algorithms can be used to design an automatic race strategy identification system. We employed an open-loop search strategy to tackle the large, continuous, stochastic state transition model and employed TD updates to address the high variance of the returns observed in the search tree. We empirically demonstrated, adapting a racing simulator to our needs, that open-loop planning can be used to improve the performance of hand-crafted race strategies, represented by the ESPN rollout policies employed during the search. We believe that online-planning algorithms can be a resourceful tool, able to provide race strategy recommendations to the strategists of Formula 1 teams during the race, especially when race situations differ from the predictions made before the race.
Nevertheless, we observe that the performance of planners strongly depends on the rollout policies employed, as they are used for initial evaluations of the tree nodes, and therefore affect the regions of the tree explored during the search. This can, in turn, be tackled with higher search budgets.

Limitations

Despite the encouraging results we obtained in the experimental settings, our work still presents some limitations. First, to be useful in a real application scenario, our system needs to be able to return good recommendations in a given time frame. With the actual implementation, we consider as budget a maximum number of interactions with the planning environment, but a straightforward modification could be made to implement a time constraint.

Second, our system heavily relies on the racing simulator presented in [13] which, in turn, is currently able to simulate with reasonable accuracy only dry races included in the 2014-2019 time span, whereas new races' parameters need to be estimated from scratch.
Third, the reward function we employ in our MDP representation does not account for ranking gains or losses with each action performed by the planner, thus leading to, in specific races, the agent losing positions by allowing opponents to overtake it when performing a pit-stop.
Fourth, many assumptions were made in our work to simplify the search space, which may not hold true in all races. For instance, the constraint on the maximum number of pit stops may not hold in races in which the weather suddenly changes or the driver experiences tire punctures or accidents.
Finally, many fundamental racing interactions are coarsely modeled by the simulator or not modeled at all. Traffic, overtaking, and retirements are some examples of partially modeled elements in the simulator. An example of an unmodeled element is, instead, tire degradation variability when pushing lap times or saving tires.

Future Work

Several future extensions of this work are possible. First, to decrease search times and to allow generalization across races, we can employ an AlphaZero [30] planning strategy, where function approximators are used to give an initial bias to the actions to explore in the tree and to evaluate the leaf nodes. The extension of this algorithm to an open-loop setting is not straightforward and can present an interesting research problem.
Second, a multi-agent framework can be employed to model situations where the controlled driver is "dueling" with other drivers. In this case, the dueling driver would not be considered as part of the environment but as a separate agent. In this setting, multi-player (double in the simple case of one rival) MCTS could be employed, allowing the agent to consider the adversarial behavior of the opponent. Finally, Inverse Reinforcement Learning (IRL) can be employed to produce better reward functions that also consider the ranking gains, apart from the lap-times as a reward signal. This way, we could obtain a dense reward signal that also captures the true objective of gaining positions in the final placement.

Bibliography

[1] P. Auer, Nicolò Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2004.

[2] Richard Bellman. *Dynamic Programming.* Dover Publications, 1957.

[3] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding klondike solitaire with monte-carlo planning, 2009.

[4] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[6] Sébastien Bubeck and Remi Munos. Open loop optimistic planning. pages 477–489, 01 2010.

[7] Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04:343–357, 11 2008.

[8] Tianqi Chen and Carlos Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016.

[9] Adrien Couëtoux and Hassen Doghmen. Adding double progressive widening to upper confidence trees to cope with uncertainty in planning problems. In *EWRL 2011*, 2011.

[10] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. volume 4630, 05 2006.

[11] Tuan Dam, Pascal Klink, Carlo D'Eramo, Jan Peters, and Joni Pajarinen. Generalized mean estimation in monte-carlo tree search, 2020.

[12] A. Heilmeier, M. Graf, and M. Lienkamp. A race simulation for strategy decisions in circuit motorsports. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2986–2993, 2018.

[13] Alexander Heilmeier, Michael Graf, Johannes Betz, and Markus Lienkamp. Application of monte carlo methods to consider probabilistic effects in a race simulation for circuit motorsport. *Applied Sciences*, 10(12), 2020.

[14] Alexander Heilmeier, Michael Graf, and Markus Lienkamp. A race simulation for strategy decisions in circuit motorsports. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, November 2018.

[15] Alexander Heilmeier, André Thomaser, Michael Graf, and Johannes Betz. Virtual strategy engineer: Using artificial neural networks for making race strategy decisions in circuit motorsport. *Applied Sciences*, 10(21), 2020.

[16] Chi Jin, Zeyuan Allen-Zhu, Sebastien Bubeck, and Michael I. Jordan. Is q-learning provably efficient?, 2018.

[17] L. Kocsis, Csaba Szepesvari, and J. Willemson. Improved monte-carlo search. 2006.

[18] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[19] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[20] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search, 2006.

[21] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[22] Erwan Lecarpentier, Guillaume Infantes, Charles Lesire, and Emmanuel Rachelson. Open loop execution of tree-search algorithms. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2362–2368. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[23] Edouard Leurent and Odalric-Ambrym Maillard. Practical open-loop optimistic planning, 2019.

[24] Dragoslav Mitrinovic and Petar Vasic. Analytic inequalities / by d.s. mitrinovic in cooperation with p.m. vasic. 1970.

[25] Andrew Phillips. 2017 preseason analysis. `https://f1metrics.wordpress.com/2017/03/14/2017-preseason-analysis/`.

[26] Jean-Michel Réveillac. 4 - dynamic programming. In Jean-Michel Réveillac, editor, *Optimization Tools for Logistics*, pages 55 – 75. Elsevier, 2015.

[27] Sandeep Singh Sandha, Mohit Aggarwal, Igor Fedorov, and Mani Srivastava. Mango: A python library for parallel hyperparameter tuning. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3987–3991. IEEE, 2020.

[28] Frederik Schadd. Monte-carlo search techniques in the modern board game thurn and taxis. 2009.

[29] David Silver. Lectures on reinforcement learning. URL: `https://www.davidsilver.uk/teaching/`, 2015.

[30] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[31] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[32] Tom Vodopivec, Spyridon Samothrakis, and Branko Šter. On monte carlo tree search and reinforcement learning. *J. Artif. Int. Res.*, 60(1):881–936, September 2017.

[33] Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992.

[34] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, 1989.

Acronyms

DP Dynamic Programming.

DPW Double Progressive Widening.

DRS Drag Reduction System.

F1 Formula 1.

FCY Full-Course Yellow.

MC Monte Carlo.

MCTS Monte Carlo Tree Search.

MDP Markov Decision Process.

OLOP Open Loop Optimistic Planning.

RL Reinforcement Learning.

SC Safety Car.

TD Temporal Difference.

UCB-1 Upper Confidence Bound.

UCT Upper Confidence Bound for Trees.

VSC Virtual Safety Car.