Analysing the Impacts of Dynamically Evolving Selection Policies in Monte Carlo Tree Search Through Evolutionary Algorithms

Fred Valdez Ameneyro
Author

Dr. Edgar Galván-López
Supervisor

A thesis submitted in fulfillment of the requirements for the Ph.D. degree in Data Science

at the
Hamilton Institute
Maynooth University
Maynooth, Co. Kildare, Ireland
Department head: Andrew Parnell

April 2024

Abstract

This thesis presents an innovative exploration into the synergy between Monte Carlo Tree Search (MCTS) and Evolutionary Algorithms (EAs), focusing on the evolution of selection policies within MCTS. MCTS, a powerful and versatile algorithm, has seen widespread adoption in various domains, from strategic gaming to robotics, due to its ability to effectively navigate large and complex decision spaces. However, the adaptability of its selection policy, a critical factor in its performance, remains a challenging aspect that demands further research.

The primary aim of this work is to investigate how evolutionary processes can be harnessed to adaptively evolve MCTS's selection policies online, thus enhancing the algorithm's efficiency and robustness in different problem landscapes, as well as in different stages of the search. By integrating EAs into MCTS, this thesis explores the dynamic and context-aware exploration of the search space, potentially surpassing the performance of traditional approaches.

The thesis lays the groundwork for understanding the fundamentals of MCTS and EA embeddings for online decision-making. It offers a detailed survey on the integration of MCTS and EAs, particularly focusing on enhancing MCTS's selection policy without prior exposure to the domain.

A series of test problems, including the Function Optimisation Problem and proposed simplifications of the board game Carcassonne, provide a platform to evaluate the interaction between MCTS's tree policy and game tree characteristics. Empirical analyses of evolved selection policies are presented, comparing them with traditional MCTS and Minimax approaches and assessing their performance.

The thesis aims to contribute significantly to AI and decision-making algorithms by advancing the integration of evolutionary strategies within MCTS. It focuses on developing adaptable and effective selection policies, examining the role of every aspect of the evolutionary processes, and refining EA integration for enhanced decision-making efficiency in MCTS.

Declaration

I, Fred Valdez Ameneyro, declare that this thesis titled, "**Analysing the Impacts of Dynamically Evolving Selection Policies in Monte Carlo Tree Search Through Evolutionary Algorithms**" and the work presented in it are my own. I confirm that

- This work was done wholly or mainly while in candidature for a Ph.D. degree in Data Science at the Maynooth University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this has been clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, all the text in this thesis is original and of my authorship.
- I used external tools (Grammarly, ChatGPT 4) for assistance strictly limited to grammar, syntax, and spelling checks, as well as for identifying potential ambiguities or difficult-to-read sentences. At no point did these tools contribute to the original research, ideas, analytical conclusions, or text body presented within this thesis.
- All the figures in this thesis were generated by me using Python's Plotly library or Google Drawings. Where any figure has not been generated by me, it is clearly stated.
- I have acknowledged all main sources of help.

Signed:

Date:

Acknowledgements

This thesis is the product of many years of hard work, and I would like to express my gratitude to the people who have supported me throughout this journey. First, I would like to thank my supervisor, Edgar Galvan, for his patience, his time spent, and his willingness to provide all the experience and guidance I could possibly need. I also want to take the time to appreciate Ken, David, Janet, Rosemary and Kate for their disposition to provide me and my colleagues with all the support we needed.

I want to especially thank my family, Susana, Fred, Danae and Yerik, for their invaluable support and love, which surpassed the distance barriers and gave me the strength and motivation to accomplish everything I have set out to do, even if it means being apart from each other for long periods of time.

I am very grateful to my friends too, a list that grew very large over the years, and I cannot possibly name all of them here. First, Maira, for her company and infinite support and affection, which I could never be grateful enough to have. Dáire, Kevin and Anna for the countless experiences together, sharing the same office and the same struggles. Osvaldo, Candela, Margarita, Laura, Nuria, Pablo and Esther, for becoming my new family in this strange country. Kanishka, Neli and Anna for being those friends that you know are always there for you. Amin, Niloufar, Aoife, Cormac and Conor for making the office environment happier and friendlier. Prabhleen for always sharing the best moments of her life with me and always being a great colleague and friend. Fergal for listening to my ideas and supporting me. And of course, Estevito for motivating me to never give up the gym. I also want to thank my friends from Mexico, Manuel, Blanca, Coral, Manuel (the other one), Claudio, Jafet, Ariel and many others, for keeping in touch with me and being there for me even when I did not reply as often as I should have.

Finally, I am very grateful to have had the opportunity to live in this beautiful country (except for the weather), and I am very proud to have been able to contribute to the scientific community in Ireland.

This thesis has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 18/CRT/6049. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission

List of publications

Peer-reviewed Journal articles

- Edgar Galván, Gavin Simpson, and Fred Valdez Ameneyro. "Evolving the MCTS Upper Confidence Bounds for Trees Using a Semantic-inspired Evolutionary Algorithm in the Game of Carcassonne". In: *IEEE Transactions on Games* (2022) *, vol. 15, no. 3, pp. 420-429, Sept. 2023, doi: 10.1109/TG.2022.3203232.* https://ieeexplore.ieee.org/document/9872022
- Edgar Galván, Fred Valdez Ameneyro. "An Analysis on the Effects of Evolving the Monte Carlo Tree Search Upper Confidence for Trees Selection Policy on Unimodal, Multimodal and Deceptive Landscapes". In: *Information Sciences Journal*, 2024. **Under review**

Peer-reviewed Conference papers

- Fred Valdez Ameneyro, Edgar Galván, and Ángel Fernando Kuri Morales. "Playing carcassonne with monte carlo tree search". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2020, pp. 2343–2350
- Fred Valdez Ameneyro and Edgar Galván. "Towards Understanding the Effects of Evolving the MCTS UCT Selection Policy". In: *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2022, pp. 1683–1690

Contents

List of Figures

List of Tables

Part I
Motives, Background and Literature Review

1 Introduction

1.1 Introduction and Motives

A key area of interest within Artificial Intelligence (AI) is the development of efficient decision-making strategies, particularly in complex, uncertain environments. Monte Carlo Tree Search (MCTS) [99], a versatile and powerful algorithm, stands out as a prominent approach in this domain. Its ability to make informed decisions based on statistical sampling has led to its widespread adoption in various applications [27], including but not limited to game playing [45], energy [65], and optimisation tasks [162].

Despite its success, a critical aspect of MCTS, the selection policy, has been a subject of continuous research. The selection policy, which guides the exploration-exploitation balance in the search process, is crucial for the correct performance of MCTS. Traditional approaches, such as the Upper Confidence Bounds (UCB1) [10], have demonstrated effectiveness in several domains. However, the one-size-fits-all nature of these policies limits their adaptability to diverse problems.

This thesis takes a deep dive into the performance of MCTS in various problem domains, focusing on the interaction between the tree policy and the game tree's characteristics. It analyses the impact of dynamically evolving selection policies in MCTS via Evolutionary Algorithms (EAs) [48]. Then, it elaborates on the concept of evolving selection policies in MCTS using Genetic Programming (GP) [103]. By incorporating EAs into MCTS, we aim to adaptively tailor the selection policy to specific problem characteristics. This approach is an initial stepping stone towards the online evolution of dynamic and context-aware tree policies, potentially leading to superior versatility and increased performance in various applications. This thesis is structured as follows.

1.2 Research Goals

The primary objectives of this thesis are to:

(1) Investigate the interaction between MCTS's tree policy and game tree characteristics in model problems.

(2) Develop and analyse EAs that evolve selection policies within MCTS, focusing on their adaptability and effectiveness.

(3) Compare the performance of evolved selection policies with traditional approaches in MCTS.

(4) Explore the role of semantics in the evolutionary process and its impact on the evolution of selection policies in MCTS.

(5) To study in depth the consequences of evolving selection policies in MCTS through EAs in real-world complex problems, such as the game of Carcassonne.

By addressing these objectives, this thesis aims to contribute to the field of artificial intelligence and decision-making algorithms, particularly in the context of MCTS and EAs.

1.3 Scope and limitations

The focus of this thesis is on analysing the integration of EAs into the MCTS algorithm, specifically in evolving selection policies for online decision-making. To this end, we surveyed and identified key problem characteristics that allow effective comparison of different tree policies, spanning from problem definition to decision tree structure and the reward landscape. We first selected the Function Optimisation Problem (FOP), a toy problem that lets us visualise the location and intensity of the search performed by MCTS. FOP has a fixed branching factor and a fixed depth in all the branches of its tree, thus simplifying the behavioural comparison of tree policies while remaining challenging to the algorithm. On the other hand, the game of Carcassonne was chosen as a real-world problem that allows us to evaluate the performance of the evolved tree policies in a more complex and strategic environment that offers distinctive playstyles and scalability. Similar to FOP, the game of Carcassonne also has a fixed depth in all the branches of its tree, but has a variable branching factor and can feature stochastic events that further complicate the decision-making process and make it serve as an excellent benchmark. Furthermore, we provide a methodology for determining the game's maximum score, essential for normalizing tree search method rewards and standardising their usage in decision-making research related to Carcassonne.

The thesis analyses empirical experimentation of the vanilla MCTS algorithm with various exploration-exploitation balances, alongside our two proposed EA-inspired MCTS extensions: Evolutionary Algorithm Monte Carlo Tree Search (EA-MCTS) and Semantically-Inspired Evolutionary Algorithm Monte Carlo Tree Search (SIEA-MCTS). EA-MCTS embeds a GP within MCTS to evolve its selection policy dynamically, without previous exposure to the problem or domain knowledge. SIEA-MCTS extends EA-MCTS with semantics, a concept that describes the behaviour of solutions within their operational contexts, to help guide the evolutionary process and ensure behavioural diversity in its small population size. The presented results are limited to the parameters defined for our algorithms and the selected problems, with an aim to provide a deep and comprehensive analysis of MCTS and the performance of the evolved selection policies in the chosen problems.

1.4 Thesis Structure

- **Chapter 2 Background** - This chapter introduces foundational concepts for understanding MCTS, including decision theory and game theory. It provides a detailed description of MCTS and its components, as well as its usage in adversarial and stochastic domains. To further the description of MCTS, the chapter introduces the Multi-Armed Bandit (MAB) problem, a fundamental concept in decision-making under uncertainty. With it, we discuss the UCB1 policy, and its adaptation for trees, the Upper Confidence Bounds for Trees (UCT), which is the most commonly adopted selection policy in MCTS. The chapter then delves into classical tree search strategies, such as Minimax and its extensions. Lastly, it explores evolutionary algorithms, particularly focusing on GP and Evolution Strategy (ES), to then wrap up with the introduction of semantics, a useful concept that describes the behaviour of solutions within their operational contexts.
- **Chapter 3 Surveying synergies between Monte Carlo Tree Search and Evolutionary Algorithms** - This chapter presents a survey of integrations between MCTS and EAs in the context of game playing and online decision-making, describing the multiple approaches found in the literature and taking a deep dive into how each implementation works. The chapter first delves into various ways EAs were embedded into MCTSs, with a special focus on how the selection and simulation phases of MCTS have been altered. Conversely, the chapter offers an analogous survey of instances where MCTS is used to modify multiple aspects of EAs, like the allocation of fitness resources or offspring generation. The survey is extended to include works where the EAs are modified with notions of the MAB problems, arguing that MABs are a crucial aspect in MCTS and hence, by extension, relevant to our research. The chapter concludes with a list of approaches where EAs are used for online decision-making in games, describing the difficulty of the problem and the potential of EAs to solve it.

- **Chapter 4 Test Problems** - This chapter delves into the attributes of game trees that are pivotal for the efficiency of MCTS, focusing particularly on the structural characteristics and reward distributions of these trees. It conducts an in-depth examination of specific test problems, namely, the FOP [87] and the game of Carcassonne, which serve as benchmarks for evaluating and improving the selection policies of MCTS. We introduce five unique functions within the FOP domain, each exemplifying a distinct reward distribution, and analyse the challenges they might present to MCTS. The chapter then discusses the game of Carcassonne, a game with deep strategic elements and unique playing styles, making it a prime subject for tree search.
- **Chapter 5 Empirical Analysis of Evolving Selection Policies in MCTS** - This chapter, introduces EA-MCTS and SIEA-MCTS, two EA-inspired variants of the MCTS algorithm that use GP to evolve their selection policies on the go. It then presents the results of experimental comparisons between EA-MCTS, SIEA-MCTS, and the traditional MCTS algorithm, for the test problems FOP and Carcassonne. The chapter takes a deep dive into the construction process and structure of the statistical trees produced by each MCTS algorithm, analysing their performance across the different FOP functions, and focusing on the influence of the inclusion of semantics in the evolutionary process. Then, it offers a quantitative comparison of the agents in the single-player variants of Carcassonne, assessing the performance and playing-styles of the MCTS-based agents in contrast to four minimax-based variants. The chapter concludes with an analysis of the selection policies evolved by EA-MCTS and SIEA-MCTS, highlighting the key characteristics of the evolved policies and their impact on the algorithm's performance.
- **Chapter 6 Evolutionary MCTS in the base game of Carcassonne** - This chapter, analyses applying MCTS, as well as our proposed EA-based MCTS agents, EA-MCTS and SIEA-MCTS, in the base game of Carcassonne for two players. The chapter describes the results of a series of matches between MCTS and other state-of-the-art Carcassonne agents. First, it analyses the performance of five vanilla MCTS variants with distinct exploration versus exploitation balances against a random uniform agent, to then determine the best vanilla MCTS variant with a round-robin tournament among them. Next, the chapter confronts the EA-based MCTS variants against the best vanilla MCTS variant, the best Expectimax agents, and a random uniform agent in another round-robin tournament. The chapter concludes with a discussion of the results and the implications of the findings.
- **Chapter 7 Conclusions and Future Work** - The final chapter concludes this thesis, summarising the results, and elaborating on our key findings regarding using EAs in MCTS. It also discusses the problems and potential avenues for improving EA-MCTS and SIEA-MCTS building upon the insights gained from our experiments.

2 Background

2.1 Introduction

Monte Carlo Tree Search (MCTS) [99] is a versatile decision-making algorithm that has had a great impact in the field of Artificial Intelligence (AI), with applications ranging from energy-based problems [65, 67], the design of Deep Neural Network (DNN) architectures [173] to tasks on the track of General Artificial Intelligence (GAI) like the General Video Game AI (GVGAI) competition [130, 131], where MCTS serves as the foundational algorithm for most of the most successful contestant agents. As a testament to its versatility, MCTS has been extended with a very diversified collection of modifications [27, 162].

In this chapter, we begin by introducing concepts that are relevant to the understanding of MCTS, discussing decision theory in Section 2.2 and game theory in Section 2.3. We then provide an overview of MCTS and its components in Section 2.4. Minimax and its extension for stochastic games are discussed in Section 2.5. Finally, an overview of Evolutionary Algorithms (EAs) is offered in Section 2.6, focusing on Genetic Programming (GP) in Section 2.6.1 and Evolution Strategy (ES) in Section 2.6.2.

2.2 Decision theory

Decision theory, under the context of AI and Machine Learning (ML), is a field that studies the optimal or near-optimal decision-making of agents in complex and uncertain environments. The goal of decision theory is to provide a mathematical foundation to understand and develop algorithms that can help agents make decisions that maximise their expected utility. A decision-making process often involves a state space, an action space, a transition model, and a reward function [171]. The state space is the set of all possible states that the agent can be in, with the action space being the set of all possible actions that can be taken from those states. The transition model is a function that maps

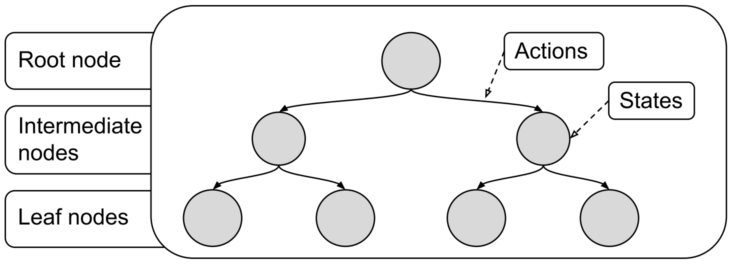

Figure 2.1: Decision tree example.

a state and an action to a new state. Finally, the reward function quantifies the desirability of states and/or outcomes, allowing the system to evaluate the quality of its decisions.

Decision theory-related problems are complex and often involve uncertainty, which can be modelled in different ways. For example, in a stochastic (random) environment, the outcome of an action is not always deterministic, and the agent must consider the probability of each possible outcome. In a partially observable environment, the agent does not have access to the full state of the environment and must make decisions based on limited information. In a multi-agent environment, the agent must consider the actions of other agents, which can be adversarial or cooperative [152]. Existing games are commonly used as benchmarks to test and develop decision-making algorithms on more complex problems.

Decision trees in decision theory

Under the context of decision theory, decision trees are tools for modelling decision-making scenarios. They are directed graphs that display decisions and their possible outcomes, structured in nodes and edges (see Figure 2.1). Each node in a decision tree represents a point of decision or chance, leading to edges that show the actions that can be taken or the events that might occur. This setup allows for a clear representation of the decision process, where the root node indicates the initial state or decision to be made, and leaf nodes represent the final outcomes or states that can be reached following certain decisions.

Decision trees help in breaking down the decision-making process into hierarchical series of choices, making it easier to analyse the consequences of each action [47]. The goal of using decision trees in decision theory within AI and ML is to aid in the development of algorithms that can navigate through these decision spaces effectively, seeking to make choices that lead to the maximisation of expected utility.

Table 2.1: Game characteristics

Characteristic	**Description**
Real-time / Turn-based	Events and decision-making occur in real-time or in turns.
Atomic turns / Complex turns	An atomic turn requires a single decision, whereas a complex turn asks for multiple decisions before the turn ends.
Simultaneous / Sequential	The players make their decisions simultaneously or sequentially, one after the other.
Perfect information / Imperfect information	The players have access to the full state of the game at all times or not.
Complete information / Incomplete information	The players know the possible moves and outcomes available to all players at all times or not.
Stochastic / Deterministic	A game is stochastic if there is randomness involved that can potentially alter its course. Otherwise, it is deterministic.
Zero-sum / Non-zero-sum	In a zero-sum game, each player's wins equal the losses of the other player. In other words, the total amount of any currency in the game from a player's perspective is constantly zero.
Symmetric / Asymmetric	A symmetric has the same rules or conditions for every player, while asymmetric games do not.

2.3 Game theory

A game is an environment where one or more decision-making agents, also called players, compete or collaborate to achieve a goal. Games are attractive in AI research for several reasons. Firstly, their complexity can be scaled, making them excellent benchmarks. Secondly, they are controlled environments that can be shaped by rules to meet specific requirements. Thirdly, games can be played indefinitely under any desired circumstances, providing an endless source of data. Lastly, games can be played by human beings, allowing for measurement and interpretation of the machine's abilities through direct confrontation with the algorithm.

AI players require different skill sets that vary according to the characteristics of the game. Games show different characteristics presented in Table 2.1 according to game theory [76, 171].

In this work, we focus on turn-based games with atomic turns, sequential decision-making, and with perfect and complete information. These games are particularly interesting for AI research, as they are more tractable and can be used to develop and test decision-making algorithms. Carcassonne is a game that fits this description and possesses characteristics that make it particularly appealing for research. A detailed analysis of the game of Carcassonne will be provided in Chapter 4.

2.3.1 Game trees

Game trees model sequential decisions and interactions between players in games of perfect information, where every move made is known to all participants. Game trees are used to analyse possible outcomes and strategies. Although primarily associated with game theory, game trees are conceptually linked to Markov Decision Processes (MDPs) [135], which model decision-making in environments where outcomes can be influenced by both agent actions and stochastic factors.

To navigate the tree, AI agents often have access to a forward model that can simulate actions and generate accurate hypothetical future states. Importantly, the forward model is independent and does not alter the original current state. The game-tree complexity and the state-space complexity are two measurements useful for evaluating the navigational and computational challenges that AI agents face when making decisions within these environments.

- The game-tree complexity measures the number of unique full sequences of events that can be followed. In terms of game theory, it reflects the number of full games that can be played. It quantifies all possible paths from the root to the leaves of the game tree.
- The state-space complexity measures the number of legal states that can be represented with the model and that can be reached from the initial state. It quantifies the number of total nodes in the game tree, ignoring duplicates.

Calculating the exact state-space complexity and game-tree complexity of a domain is a challenging task in itself [78]. Therefore, lower bounds are often approximated for complex domains. For example, the game-tree complexity of Chess is estimated to be around 10^{123} [150], while the state-space complexity of the game of Go is approximately 10^{170} [168]. It is worth noting that both the state-space complexity and the game-tree complexity are correlated to the average branching factor of the tree, which tends to be larger in domains with stochastic events.

2.3.2 Game trees in multi-agent adversarial domains with uncertainty

A stochastic event in a domain is represented in its decision tree with a chance node, also called **node* (pronounced as "star node"). With *nodes, the decision tree becomes an Expectimax tree or *-Minimax tree that distinguishes between *nodes and decision nodes. Stochastic events add complexity to the decision tree by requiring the consideration of their potential outcomes. Non-determinism in games can manifest in various ways, but a common scenario involves alternating between a random event and a decision,

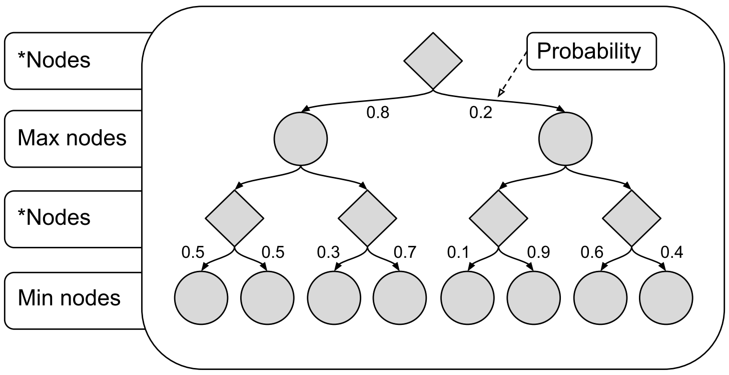

Figure 2.2: Regular *-minimax tree example. *Nodes are diamond-shaped, while decision nodes are circle-shaped.

as seen in games like Backgammon and Carcassonne. For example, in Backgammon, two six-sided dice are rolled at the beginning of each player's turn, and then the player chooses which checkers to move. These types of games exhibit regular *-minimax trees [15], where a layer of chance nodes alternates with a layer of decision nodes throughout the tree. This structure is illustrated in Figure 2.2, where directed edges emanating from *nodes represent unique outcomes of the stochastic events, and edges from decision nodes represent available actions. Note that each layer of *nodes is alternated with a layer of decision nodes, and the sum of event probabilities within each *node is equal to 1.

Figure 2.2 also illustrates the tree of a domain where two agents with opposite objectives alternate actions. In adversarial domains, decision nodes are classified as max or min nodes if the decision-maker seeks to maximise or minimise the reward, respectively. However, there are games where a player can have multiple actions, or multiple stochastic events can occur in the same turn. For example, in the game of Risk, a player's turn consists of several phases. The attack phase allows the player to perform multiple attacks, each consisting of choosing the source and target territories of the attack and then rolling the dice to determine the outcome. The player can repeat this cycle to convenience, only limited by the availability of the troops. In this case, the alternation of random events and player decisions still occurs, however, the game tree is not a regular *-minimax tree as the decision nodes are not necessarily at the same depth.

2.4 Monte Carlo Tree Search

MCTS is a decision-making technique that offers the flexibility to be stopped at any time to obtain the best current estimated action. This attribute extends the applicability of MCTS to both strategic, slow-paced domains, as well as real-time scenarios. MCTS also employs a policy-driven approach for tree expansion, with Monte Carlo simulations used

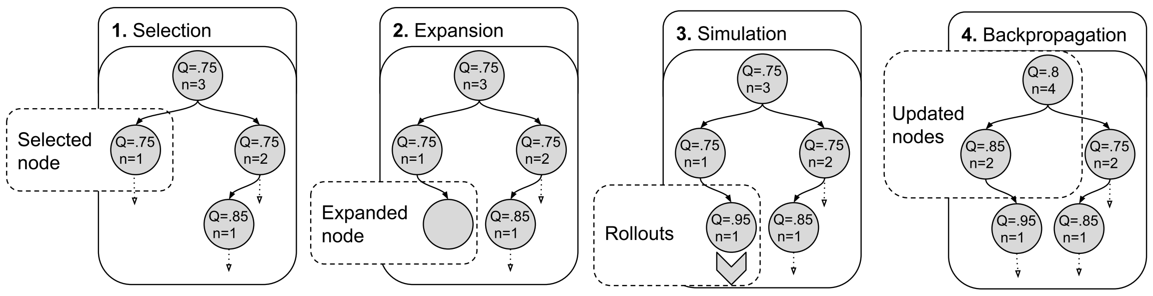

Figure 2.3: Illustration of an iteration of the MCTS algorithm.

to evaluate the nodes and inform decision-making.

The MCTS algorithm takes a current state as input and returns the action it believes is the best available. Given enough time, vanilla MCTS converges to minimax search in adversarial domains. The vanilla version of MCTS consists of four steps or phases, explained next:

- *Selection phase*: Starting from the root node, the tree policy is used to select child nodes iteratively until an expandable node is reached. That node becomes the selected node.
- *Expansion phase*: Chosen by the expansion policy, a new state is simulated from the selected node. The new node is added to the statistical tree as a child of the selected node. That node becomes the expanded node.
- *Simulation phase*: A series of rollouts, also called playouts, are executed from the expanded node to obtain its approximate evaluation. Each rollout simulates actions according to the default policy until a terminal state is reached to collect its outcome. The results are averaged and returned as the evaluation of the expanded node.
- *Backpropagation phase*: The collected information from the current iteration is backpropagated to the tree. Typically, only the nodes that connect the expanded node and the root node are updated. The nodes updated, information, and aggregation methods are determined by the backup operator

A complete run of these steps is called an *iteration*. With each iteration, MCTS builds a statistical tree that stores information in every node. Starting with only the current state as the root node, MCTS uses the forward model to simulate future game states and expand its statistical tree, meaning that the statistical tree is a subset of the game tree that grows as states are discovered. Each node in the statistical tree keeps track of its visit count and reward. The vanilla version of MCTS is illustrated in Fugure 2.3.

The tree policy, expansion policy, and default policy define the behaviour of the selection, expansion and simulation steps in MCTS, respectively. A different policy, called the recommendation policy, determines which action to return when the algorithm terminates.

Vanilla MCTS usually returns the action that has been visited the most, referred to as the robust action, as its recommendation policy. Other recommendation policies include the max action (the action with the highest rewards), the robust-max action (the action with both the largest visits and rewards, requiring additional iterations while none of the actions suffices) [42], and the secure child (the action that maximises a lower confidence bound) [35].

We identify three primary approaches to apply MCTS in games, each offering distinct advantages depending on the context of use: (a) Online isolated decision-making is the most direct form to deploy MCTS to determine the best possible action in response to a specific situation within an unknown environment. This application is typically seen in scenarios where a singular decision is required. (b) Online sequential decision-making, on the other hand, facilitates decision-making throughout the entire duration of a game. This approach benefits from the algorithm's capability to build upon knowledge acquired from previous decisions if the information gathered is relevant for subsequent decisions. For instance, MCTS can reuse parts of its previously generated statistical tree, setting the current state as the new root and discarding the rest of the tree [126]. The search then begins with a statistical tree that has been already partially explored [27]. (c) Offline deployment, or preparatory learning, allows the algorithm to optimise its strategies in a specific domain through self-play and iterative improvement [33]. This approach enables MCTS to learn and refine its decision-making capabilities based on early exposure to the domain, thereby improving its performance in actual game situations by having developed a repertoire of strategies before engaging in live play [154]. In this work, we are interested in the adaptability and potential of MCTS variants, which can be more effectively measured in scenarios where the algorithm is used for online isolated decision-making.

2.4.1 Monte Carlo simulations

Every search algorithm needs a heuristic evaluation of the nodes within the game tree of any complex game. While hand-made evaluations tailored for specific games are commonly used, there is also a widely adopted heuristic that requires no human knowledge and has demonstrated reasonable robustness [85]: Monte Carlo simulations. Monte Carlo simulations, within the context of games, obtain numerical results by repeatedly sampling the outcomes of games using random play. The accuracy of these results improves as more simulations are executed. The Monte Carlo simulations, often referred to as roll-outs, playouts, or simulations, are used in the MCTS algorithm to evaluate any state. This approach allows the algorithm to assess the potential value of a move based on the aggregated outcomes of these simulations, rather than relying on detailed, game-specific knowledge or complex strategic evaluation. By averaging numerous results from a given state to the end of the game, MCTS can estimate the expected utility of making a certain move, guiding the selection of the most promising paths through the game tree. This method's strength lies in its simplicity and versatility, making it applicable to a broad range of games and scenarios where traditional heuristic evaluations might not be feasible or effective.

2.4.2 The tree policy

One of the most impactful, hence most researched aspects of MCTS is its selection step. The selection step in MCTS is centred around the tree policy, which determines which nodes are deserving of resource allocation. The tree policy aims to strike a balance between acquiring new knowledge (exploration) and making decisions based on existing knowledge (exploitation). It is employed to iteratively select the action that leads to the most interesting node based on the information accumulated in the statistical tree. Whenever the tree policy selects an action, it effectively addresses a Multi-Armed Bandit (MAB) problem in which each available action can be viewed as an arm of the bandit.

Multi Armed Bandits

A stochastic MAB is a model that characterises decision-making problems in the face of uncertainty. It serves as an analogy to a gambler who is presented with multiple slot machines and aims to maximise their overall reward. In a MAB problem, an agent is confronted with a set of k arms, each associated with an unknown reward distribution D. The agent is tasked with collecting the reward r_t from one arm at each discrete time step $t \in \{0, 1, ..., n\}$. The primary objective of the agent is to acquire knowledge about the reward distributions of the arms in order to make more informed decisions over time and maximise his cumulative reward. The MAB problem exemplifies the fundamental trade-off between exploration and exploitation.

In the context of MAB, exploration refers to the act of gathering information about the potential rewards of each arm. This typically entails choosing arms that may seem to have lower rewards in order to increase certainty about their reward distribution. On the other hand, exploitation involves selecting the arm that the decision-maker believes, based on the accumulated information, will yield the highest immediate reward.

Several conditions and assumptions are associated with a MAB problem. Within the context of tree search, the reward distributions are assumed to be stochastic and independent of one another [159]. Additionally, in certain domains, the rewards are considered to be non-stationary. The distributions of each arm can change over time in a non-stationary MAB, such as when a search algorithm discovers an optimal strategy in a branch of the tree that significantly alters the rewards of a node.

There are several approaches to addressing MAB problems, and the literature has proposed multiple policies [8, 9, 28]. Each policy presents unique characteristics that influence the behaviour of the agent. These policies are categorised in [122] as follows.

- The *bayesian exploration* approach assumes that the agent holds a prior belief about the reward distributions of the arms. The agent then utilises the information gathered from the arms to update its beliefs. An important example in this category is Thompson Sampling (TS) [163], where the agent samples from the assumed prior distribution of rewards and selects the arm with the best outcome from those samples. The assumed distributions are then updated based on the observed reward obtained from trying the arm. TS has been extensively studied [95], its effectiveness has been empirically demonstrated [34, 74, 149], and its asymptotic optimality has been theoretically proven [2].
- The ϵ*-greedy exploration* chooses the best arm with a 1-ϵ probability, and a random arm otherwise. This approach is simple and effective, but it does not guarantee asymptotic optimality [10].
- The *soft-max exploration* chooses the best arm with a probability proportional to its estimated value, like the EXP3 algorithm [11].
- Last, the *optimistic exploration* refers to policies optimistic in the face of uncertainty, with the main exponent being the Upper Confidence Bounds (UCB1) [10].

In a MAB problem, the regret refers to the total loss incurred by trying suboptimal arms. In other words, regret represents the difference between the reward that could have been obtained by always selecting the best arm and the reward actually achieved by the agent. The UCB1 policy, discussed next, constrains the regret to grow logarithmically, making it asymptotically optimal. UCB1 is the most widely used policy in MCTS, and it is the one employed in the original version of MCTS [99].

Upper Confidence Bounds

Upper Confidence Bounds(UCB1) is a strategy used to address MAB problems, guaranteeing convergence to the best arm by limiting regret. UCB1 adopts an optimistic in the face of uncertainty approach, ensuring that every arm always has a probability of being chosen greater than zero. UCB1 is formally described in Equation 2.1.

$$UCB1_j = Q_j + C\sqrt{\frac{2 \cdot ln(N)}{n_j}} \tag{2.1}$$

where Q_j is the average reward of arm j and n_j is its number of tries. N is the total number of tries among all the alternative arms and C is a constant that balances exploration and exploitation. UCB1 has been modified and adapted to various scenarios. For instance, there are adaptations of the UCB1 policy for non-stationary MAB problems, such as Discounted Upper Confidence Bounds (D-UCB) [100] where the significance of the oldest attempts for each arm diminishes over time, and Sliding Window Upper Confidence Bounds (SW-UCB) [70] which considers only the most recent set of rewards to compute the current distributions of the arms. The UCB1 used as part of the tree policy in MCTS is the Upper Confidence Bounds for Trees (UCT), described in the next section.

Upper Confidence Bounds for Trees

UCT employs the UCB1 strategy to select a node at each level of the tree until a node that can be expanded is encountered. The hypothesis is that the most promising node to allocate resources to is the one with the highest UCB1 value, as this value effectively balances the trade-off between exploration and exploitation potential for each node. UCT represents the edges of the game tree as arms of a bandit to identify the most promising nodes within the tree, beginning from the root node.

In scenarios where there are two adversarial decision-makers, the tree is explored using a minimax approach: Player 1 aims to maximise the outcome, while Player 2 tries to minimise it. In this context, when selecting an action corresponding to the opponent's turn, Q_j in the UCB1 formula is swapped to $-Q_j$. Consequently, UCT works as a minimax algorithm, expressing interest in the optimal action available to the opponent during their turn.

To implement UCT in a statistical tree built by MCTS, it is necessary to store the reward Q_i and the number of visits n_i for each node. The reward Q_i represents the average of the rewards obtained by the agent in all the simulations that traversed the node, while n_i denotes the number of times the node has been selected by the tree policy. Naturally, most adaptations of the UCB1 formula can be extended to UCT at the expense of storing additional information in the statistical tree if needed.

2.5 Classic tree search algorithms: Minimax

Tree search agents expand subsets of the decision tree as part of their decision-making process in order to acquire and store information. One commonly used tree search algorithm in multi-agent domains, especially in adversarial scenarios, is Minimax. This algorithm categorises nodes as either *max nodes* or *min nodes* depending on whether the decision-maker at that node seeks to maximise or minimise the reward from the perspective of the current decision-maker. Minimax exhaustively expands nodes, assuming that the opponent will always make the best available move. Techniques such as alpha-beta pruning [98] and forward pruning [23] can help reduce the number of states that need to be evaluated in the Minimax algorithm. Despite the use of pruning techniques, exhaustive search becomes intractable in complex domains given the exponential growth of the states that need to be searched. To stop the algorithm, the search depth is generally restricted. When a time limit is given instead of a maximum depth, an effective way to manage the uncertainty of the number of states at different depths is to progressively deepen the search using iterative deepening until a time limit is reached, a strategy known as Iterative Deepening Depth-First Search (IDDFS) [101].

2.5.1 Expectimax

Expectimax, also known as Expectiminimax, is an adaptation of the Minimax algorithm designed for non-deterministic games. Similar to Minimax, Expectimax explores the game tree in a width-first manner. However, instead of making deterministic choices at chance nodes, it calculates the values of these nodes as the weighted average evaluation of their successors.

To improve the Expectimax algorithm in regular expectimax trees, the *-minimax family of algorithms, including Star1, Star2, and Star2.5, have been proposed in [15]. These algorithms use bounds to prune the tree, similar to how alpha-beta pruning works for Minimax, but taking into account the probabilities associated with each chance node. Consequently, the *minimax algorithms can efficiently prune the tree, particularly when the actions can be sorted by quality based on prior knowledge of the domain.

2.6 Evolutionary Algorithms

EAs [48] are versatile optimisation techniques that excel in solving problems with large search spaces. They explore the space of possible solutions using evolution-inspired operations, including selection, crossover, and mutation. A selection mechanism gives priority to the fittest individuals in the population, increasing their chances of passing on their genetic material to future generations. This iterative process improves the quality of the population until certain stopping criteria are satisfied. Figure 2.4 illustrates a general EA.

Ideally, an EA should be able to simultaneously explore the search space extensively and exploit the most promising regions. However, controlling the exploration-exploitation dilemma poses a non-trivial challenge in EAs due to the complexity of the interactions of its components. Prioritising exploration can significantly slow down the convergence speed of the algorithm or even prevent it altogether. Conversely, a focus on exploitation may lead to a loss of population diversity or premature convergence, hindering further improvement, a phenomenon known as stagnation [43].

To ensure the exploration of the search space, EAs must maintain a diverse population. This diversity allows the evolutionary process to have options to escape local optima and explore the entire search space, and is typically achieved through mutation, which can potentially produce novel genetic material. On the other hand, crossover operators aim

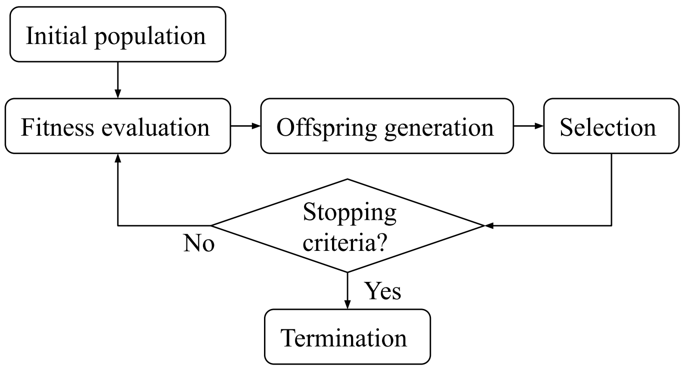

Figure 2.4: Generic steps in an Evolutionary Algorithm.

to preserve and recombine genetic material. The selection mechanism determines how often the best individuals can produce offspring, determining the influence on the genetic material of the next generation.

EAs typically offer a high degree of flexibility in their design and contain multiple adjustable parameters. Besides the definition of the genetic operators, the balance between exploration and exploitation within the search space is also influenced by the representation of individuals, fitness evaluation, population size, population structure (steady state or generational) and the allocated computational resources. The field of EAs encompasses four notable methods [12]: Genetic Algorithm (GA) [72, 80], ES [22, 139], Evolutionary Programming (EP) [56, 117], and GP [103]. In this thesis we focus on GP and ES, described next.

2.6.1 Genetic Programming

GP [103] is a type of automated programming in which individuals are commonly represented as syntax trees. The tree structure is typically used to represent mathematical expressions, but it can also be used to represent other types of solutions, such as computer programs, behavioural trees or decision trees. The GP procedure commences by generating a population of programs using an initialisation method. This population is subsequently assessed, and the most promising individuals are chosen as parents for the next generation. The selection process determines which individuals will contribute to producing offspring. In the *tournament selection*, one of the most common selection approaches in GP, a random subset of the population is chosen and the best individual from the subset is designated as a parent.

The selected parents undergo modifications through genetic operators, such as crossover and mutation, to generate offspring. The fittest individuals are combined with the offspring to complete the population of the next generation. This process continues until specific stopping criteria are met, and the GP eventually returns the best individual from the final population. These steps align with the general stages of the EA illustrated in Figure 2.4.

The design of a GP includes the definition of a terminal set, a function set, a fitness measure, an initialisation method, a selection method, genetic operators, and termination criteria, each with its respective parameters. Additionally, constraints are necessary to regulate the representation, such as a maximum depth restriction, to prevent phenomena like bloat. Bloat is a phenomenon in GP where syntax trees tend to gradually increase in depth throughout generations without necessarily increasing their performance. This growth often produces introns, which are internal nodes that do not influence the output of the individual. Introns are believed to emerge in GP as an evolutionary mechanism to safeguard valuable structures by preserving multiple instances of them, as most genetic operators are thought to possess some degree of destructive nature [114].

To simplify our discourse, we make the distinction between the three types of trees we have defined thus far. First, the **game tree** represents all potential actions (edges) and states (nodes) within a game. Next, the **statistical tree** is a subset of the game tree created and used by the search algorithm to store the information it finds. Lastly, the **syntax tree**, is the tree-like structure representing a mathematical formula, which is also an individual in the GP population, explained next.

Genetic Programming individual representation

A syntax tree is a data structure that represents the flow of information from the leaves to the root of a tree. Various forms of GP have been proposed in the literature, with the tree-like structure being the most common representation [103]. In this structure, functions are represented by internal nodes, while terminals are represented by leaves. Each internal node takes its children as inputs and applies its function to them. The syntax tree is evaluated recursively, starting from the root node and proceeding until the terminal nodes are reached. Figure 2.5 provides an example of a syntax tree.

A syntax tree does not guarantee mathematical correctness when representing mathematical expressions. Therefore, it is common practice to incorporate safety rules in the functions. These rules ensure that the execution does not crash when evaluating a tree. For instance, one such rule is to take the absolute value of the input in any square root operation. Another rule is to return the numerator in a division if the denominator is zero, which can cause several discontinuities. These rules are typically implemented as a set of if-else statements that are evaluated before the function is executed. Under this context, the *genotype* of an individual in GP is the syntax tree, while the *phenotype* is the output of the syntax tree when evaluated.

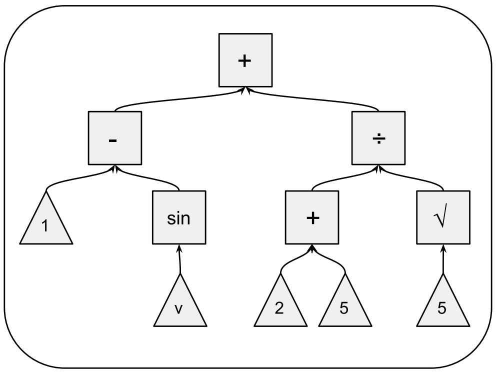

Figure 2.5: Illustration of a syntax tree. Squared nodes are functions and triangular nodes are terminals. In this example, the syntax tree represents $1 - sin(v) + \frac{(2+5)}{\sqrt{5}}$.

The fitness measure, function set, and terminal set are chosen to align with the specific problem being addressed. The fitness measure is typically a function that evaluates the performance of the syntax tree on the problem at hand. For example, in a problem involving the approximation of a given dataset, variables from the dataset could serve as terminals, while mathematical operators can be employed as functions, with the fitness measure being the Mean Squared Error (MSE) between the output of the syntax tree and the target values.

Population initialisation

Regarding the initial population in GP, there are several standard initialisation methods available, including *grow*, *full*, and *half-and-half* as described by Koza [103]. The full method grows a symmetrical tree with the same maximum depth for all branches, with the maximum depth being a user-defined parameter. In this method, functions are randomly selected for each node until the maximum depth is reached and terminals are added instead. The grow method is similar to the full method but terminals can be randomly added at any depth, meaning that the depth is not enforced for every branch of the tree. The half-and-half method mixes both full and grow in a proportion defined by an additional parameter.

Genetic operators

There is a vast variety of genetic operators in GP, with the most common ones being crossover and mutation. Crossover involves combining two or more individuals to produce offspring. In GP, the subtree crossover, illustrated in Figure 2.6, swaps subtrees between parents. Other crossover variants have been proposed in the literature, such as the Context

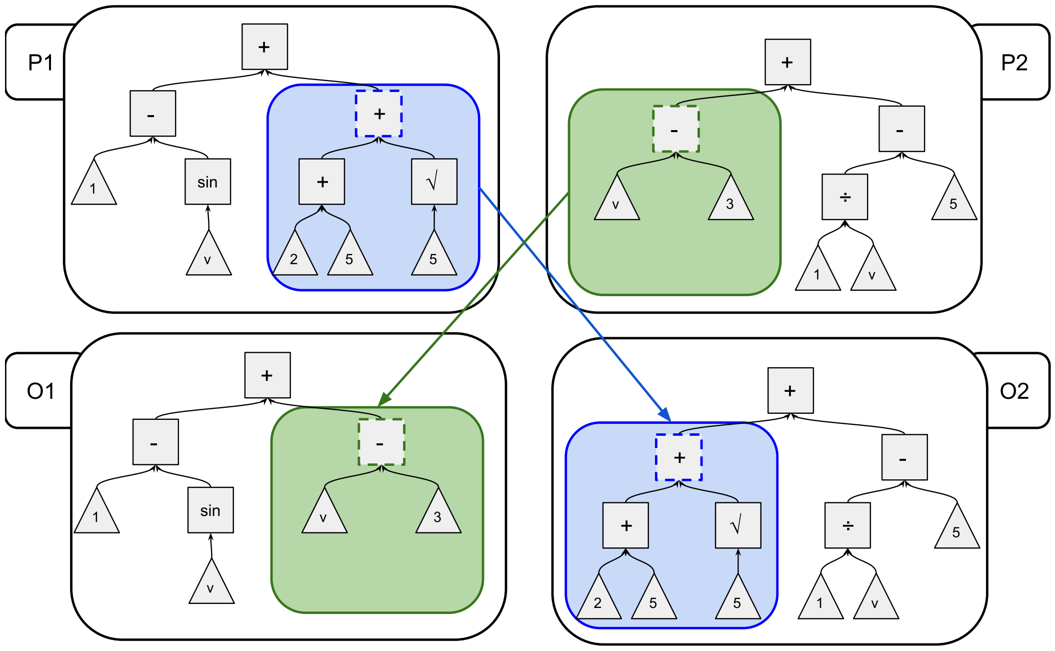

Figure 2.6: Syntax tree subtree crossover. The dashed nodes are the cut-off points for the crossover. The subtrees are swapped between the parents (P1 and P2, at the top) to produce offspring (O1 and O2, at the bottom).

Aware Crossover (CAC) [114], Semantics Aware Crossover (SAC) [170] and Semantic Similarity-based Crossover (SSC) [170]. These variants aim to generate offspring that are more informed and potentially better solutions.

Mutation, on the other hand, entails randomly altering an individual to produce offspring. It serves as a source of novel genetic material for the population. In the case of syntax trees, the mutation operator is typically implemented by replacing a subtree with a newly generated subtree (subtree mutation). Alternatively, it may involve swapping the content of a randomly chosen node with another valid function or terminal (point mutation). The mutation operator plays a crucial role in exploration by introducing potential changes to the genetic makeup of individuals. However, modifications in the genotype can have unpredictable effects on the overall performance of a solution. There are metrics that can help us understand how the evolutionary process navigates the search space and how challenging it may be to find optimal solutions. For instance, locality [68] describes the correlation between small changes in an individual's genotype and small differences in its phenotype. The Fitness Distance Correlation (FDC) [89], on the other hand, measures the correlation of the fitness of the solutions and their distance to the optimum solution. Finally, it is common practice to perform a Fitness Landscape Analysis (FLA) [175] between the problem domain and the proposed solution representation. This analysis helps assess the efficiency of the genetic operators by examining whether they produce solutions that exhibit similar behaviours to their parent solutions. If the genetic operators fail to maintain behavioural similarities, the evolutionary search may perform on par with random search [141].

Semantics

Diversity, which refers to the differences between individuals, can be assessed by examining their genotypes. Several metrics can be used for this purpose, such as the Edit Distance (ED) [49], the Alignment Distance [167] and the Normalised Compression Distance (NCD) [40]. The NCD, for instance, employs the Kolmogorov complexity [107] to compare how differently a program would generate the structure of one individual compared to another. This makes NCD applicable to any type of individual representation. It was initially tested on the syntax trees of GP in [68]. However, these approaches have a key limitation: they solely consider the genotype of individuals and do not take into account their actual behaviour, also referred to as **semantics** in GP.

According to Galvan et al. [60] and inspired by [124], rather than focusing on its genotype, the behaviour of each individual, based on the outputs when provided with the relevant fitness cases as inputs, can be used as a diversity measure. The actual behaviour of a solution exists within the semantic space S. This semantic space represents all possible behaviours of the solution for all considered inputs. Let p and q be programs from a language P. When p is applied to an input $\mathsf{i} \in I$, p produces an output $p(\mathsf{i})$.

Def. 1 *The semantic mapping function $s : P \mapsto S$ maps any program from P and the semantic space S.*

which has the behaviour $s(p) = s(q) \iff \forall \mathsf{i} \in I,\ p(\mathsf{i}) = q(\mathsf{i})$ and posseses three key properties. Firstly, every program and input set has unique semantics. Secondly, multiple programs can have the same semantics for a given input set. Thirdly, programs that generate different outputs for the same input set exhibit distinct semantics. Def. 1 is general and does not specify the representation of semantics. In GP, a popular representation of semantics is Sampling Semantics (SS), defined as the vector of output values computed by an individual program for a given input set. This differs from the notion of fitness cases, which are input-output pairs where the output is the desired one.

Based on Def. 1, we can establish the definition of the Sampling Semantic Distance (SSD) between two programs (p, q). Let $SS(p, I) = \{p_1, p_2, ..., p_k\}$ and $SS(q, I) = \{q_1, q_2, \cdots, q_k\}$ be the SS of p and q, when evaluated on the same set of inputs I. The SSD between p and q is defined as shown in Equation 2.2.

$$SSD(p, q) = \sum_{i \in I} \frac{|p_i - q_i|}{|I|} \tag{2.2}$$

The Semantic Similarity (SSi) [170] compares the SSD of any two programs to determine their similarity. In an EA, the SSi can be employed to manipulate the behavioural difference between a parent solution and its offspring. This means that it serves as a tool to control the level of exploration exhibited by the operators responsible for generating offspring. The objective is to produce offspring that are neither too dissimilar nor identical to their parents, similar to the concept of a learning rate in Reinforcement Learning (RL) models. The SSi determines if programs p and q are *semantically similar*, formally defined in Equation 2.3.

$$SSi(p, q) = (L < |SSD(p) - SSD(q)| < U) \tag{2.3}$$

where L and U are the lower and upper bounds of the SSi indicator respectively, also called the *semantic sensitivity*. L and U are parameters tuned empirically to keep the solution's behaviour similar to the parent's while still allowing for exploration, which has proven beneficial for GP [69, 170] and Multi-Objective Genetic Programming [63].

2.6.2 Evolution Strategies

Evolution Strategies (ES) are generally applied to real-valued representations of optimisation problems. In ES, the mutation is the main operator and crossover is the secondary, optional, operator. There are two basic forms of ES, known as (μ, λ)-ES and $(\mu + \lambda)$-ES. The variable μ refers to the size of the parent population and λ refers to the number of offspring that are produced in the following generation before selection is applied. In the former, the parent is discarded whereas in the latter, the parent is kept as part of the next generation's population.

Because of how ES explores the search space, it is a common practice to seed the initial population with a known solution to the problem at hand with the expectation to improve it. The evolutionary process begins by creating offspring from the known solution and then progresses from there.

The distinguishing characteristics of different EA methods, such as the syntax tree representation from GP and the population structure of ES, can be combined to create new hybrid approaches. For example, it is possible to incorporate a GP tree-like representation into an ES framework, as demonstrated in our IEEE Transaction on Games article [61]. This hybrid method was developed for evolving programs in resource-constrained domains. In this hybrid approach, the ES employs a relatively small population and uses the genetic operators typically used in GP to modify the syntax trees. The search space for potential syntax trees is vast, and a significant portion of the trees generated during evolution may not be useful or viable solutions. This poses a challenge when working with a small population, as making progress becomes difficult. In such scenarios maintaining diversity becomes crucial to avoid stagnation.

3

Surveying synergies: Monte Carlo Tree Search and Evolutionary Algorithms

Monte Carlo Tree Search (MCTS) and Evolutionary Algorithms (EAs) are algorithms that can benefit from each other, and multiple combined approaches have been attempted in literature. Most of the approaches that use EAs in MCTS aim to optimise some aspect of it or to try to make it more generalisable through adaptation. Conversely, when MCTS is embedded within EAs, the former is used to make EAs more methodological and controlled. The usage of EAs under the context of MCTS is primarily focused on offline optimisation through iterative exposure to the domain. However, there have been ingenious proposals to integrate them with MCTS online, that is, as MCTS's decision is being made. In this chapter, we summarise the usage of EAs combined with MCTS. Section 3.1 discusses the usage of EAs in MCTS, while Section 3.2 discusses the usage of MCTS in EAs. The selection phase of MCTS models every choice made to traverse the tree as a Multi-Armed Bandit (MAB) problem. This implies that, by extension, EAs interact with the MAB model when used in combination with MCTS. Thus, Section 3.3 expands our scope to include works that combine MABs with EAs. Finally, Section 3.5 discusses the usage of EAs and MCTS in the context of games, with special focus on how EAs are adapted to be used online despite the constrained computational resources.

3.1 Evolutionary Algorithms in Monte Carlo Tree Search

EAs have been successfully employed to optimise components of the standard MCTS algorithm. For instance, the Self-Adaptive Monte Carlo Tree Search (SA-MCTS) [158] algorithm, models the vanilla MCTS parameters as a Combinatorial Multi-Armed Bandit (CMAB) problem to tune them on-the-go as originally proposed in [157]. In their work, the authors proposed three versions of SA-MCTS, each with a different parameter allocation strategy and two of which use EAs. The three SA-MCTS variants are SA-MCTS with Naïve Monte Carlo (SA-MCTS$_{\text{NMC}}$), SA-MCTS with N-Tuple Bandit Evolutionary Algorithm (SA-MCTS$_{\text{NEA}}$) and SA-MCTS with a simple Evolutionary Algorithm (SA-MCTS$_{\text{EA}}$). In SA-MCTS$_{\text{NEA}}$, the parameters are evolved with the N-Tuple Bandit Evolutionary Algorithm (NTBEA) [104] by representing each parameter as a 1-tuple and the combination of parameters as n-tuples. SA-MCTS$_{\text{EA}}$ represents combinations of parameters as individuals in a Genetic Algorithm (GA), where each parameter is a gene. The population follows a $(\mu + \lambda)$-Evolution Strategy (ES) where $\mu \geq 2$. Each individual is evaluated by letting it control a MCTS iteration and using the reward as its fitness. It uses a uniform random crossover and a uniform random mutation to generate offspring. They tested the SA-MCTS algorithms in 20 games of the General Video Game AI (GVGAI) competition [131]. The MCTS parameters that were tuned included the C of the Upper Confidence Bounds (UCB1) formula (refer to Chapter 2) in the tree policy and the maximum depth of the playouts of the simulation phase. The experimental results on the GVGAI games demonstrated that SA-MCTS agents generally achieved higher win rates than the baseline MCTS agents, with the EA-based versions performing slightly better [156].

3.1.1 Evolutionary Algorithms in Monte Carlo Tree Search's simulation phase

One way to improve the performance of MCTS is to optimise the evaluation of the states found by the tree search. This can be achieved by learning an evaluation function for any game state, or by enhancing MCTS's default policy. The default policy in MCTS is Monte Carlo simulations, which choose actions uniformly at random selecting actions until reaching the end of the game, although additional rules can be incorporated. Playouts that involve more than just a random selection of actions are known as *heavy playouts*. Note that heavy playouts have been proven to not always be beneficial to the overall performance of MCTS and they may even be harmful [71].

The decisions of the default policy can be optimised with ES as in Hivemind [133], which evolves rules based on patterns in the vicinity of the last move played in the game of Hex. The evolved rules are used to bias the choices of MCTS's default policy, leading to improvements in performance.

The Knowledge Based Fast Evolutionary Monte Carlo Tree Search (KB Fast-Evo MCTS) [127] is an extension of Fast Evolutionary Monte Carlo Tree Search (Fast-Evo MCTS) [111] where the rollouts of MCTS are guided by policies derived from an EA. The EA follows a (1+1)-ES (explained in Chapter 2) in which each individual acts as a default policy that selects actions during the simulation step of MCTS. The individuals process features of the game state with a set of weights, represented as genes. These features include Euclidean distances from the playing agent to all interactable objects, and new weights are added to the genome if new objects are encountered.

KB Fast-Evo MCTS improves upon Fast-Evo MCTS by introducing a knowledge base that combines two factors: curiosity and experience. The knowledge base keeps track of statistics for all interactable sprites in the game, with curiosity aiming to discover the consequences of collisions between sprites, while experience rewards collisions that have proven beneficial to the agent. The fitness of each individual is calculated using a score function that considers changes in experience, curiosity, and the game score at the end of the playout. The authors employ games from the GVGAI framework to compare their proposed KB Fast-Evo MCTS with vanilla MCTS, Fast-Evo MCTS, and Knowledge Based Monte Carlo Tree Search (KB MCTS). They find that KB Fast-Evo MCTS performs the best among these approaches. KB MCTS is an MCTS variant proposed as an ablation study to KB Fast-Evo MCTS, where the knowledge base is used to evaluate the results of the playouts during the simulation phase, but no evolution is performed.

In another approach described in [3], a Genetic Programming (GP) paradigm is used to evolve agents that guide the default policy in the game of Ms. Pac-Man. The agents generated through the GP consist of if-then-else decision trees that determine simple actions, such as directing Pac-Man towards the closest pill. These decision trees combine a series of hand-crafted heuristics that provide information about the game, such as the distance to the closest ghost. The authors discovered that MCTS with the evolved decision trees as the default policy performed competitively compared to hand-crafted agents.

The EvoMCTS algorithm [17, 18] proposes the evolution of a board evaluation functions using GP. In this approach, the evolved board evaluation function serves to guide the default policy. During the simulation step, the default policy selects the best available action by considering a one-step look-ahead evaluation of the next available states. To handle large branching factors, the default policy in EvoMCTS randomly samples a certain number of actions, defined as the *playoutBranchingFactor*, from each state. EvoMCTS defines two types of terminal nodes to be considered in the evolved trees:

- Basic terminal nodes: return values based on the current game state. For instance, one possible implementation could tally the number of game pieces owned by the player.
- Game-oriented terminal nodes: return values relevant to the application in particular. For example, a heuristic function that evaluates corner control in Othello, which is considered an important aspect of the game.

Although the basic terminal nodes aim to be game-independent, they are certainly limited in their applicability, as are the evolved trees. After all, heuristics benefit some algorithms more than others, and it is difficult to determine how much of the success of an algorithm can be attributed to its heuristics or the algorithm itself [127]. The authors of EvoMCTS argue that the game-specific terminals proposed use little domain knowledge and are properties that any player quickly realises after playing a few games.

Interestingly, the representation of EvoMCTS allows individuals to have Explicitly

Defined Introns (EDIs), which are theorised to benefit evolution by protecting sections of code with no repercussions in the evaluation of the carriers. EvoMCTS also features coevolution and innovative genetic operators. It was tested in the games of Othello, Hex, and Dodgem, and the results showed that the evolved trees were considerably stronger than vanilla MCTS agents.

Bandit-based Genetic Programming (BGP) [82] is an EA that systematically tests modifications from a predefined set of available modifications that can be applied to an individual in a domain with stochastic rewards. The assumption is that even if two different modifications are individually proven to be beneficial, the combination of both modifications may not necessarily yield a superior individual. To address this issue, confidence bounds are employed, providing strong theoretical justifications. In BGP, each unique modification is initially tested a fixed number of times when applied to the individual, and the resulting outcomes are used to calculate the UCB1 value for each modification. As multiple hypotheses are being tested simultaneously (one for each available modification), the problem is modelled as a racing algorithm. Hoeffding's bounds are used to calculate an upper bound and a lower bound on the efficiency of each modification. BGP is then a modified racing algorithm that uses these upper and lower bounds to accept or reject proposed modifications to the individual. Each modification is then tested again, the same number of times, and ordered based on their UCB1 values. In BGP accepting a modification is equivalent to performing mutation in an EA. The MAB framework is employed when selecting which modification to test next. BGP has shown promising results when applied to enhancing agents for playing the game of Go. In this context, each modification corresponds to a change in the biases of the default policy within a MCTS-based agent.

More recently, a combination of GP and MCTS to play Hearthstone was proposed in [38], where the GP evolves an evaluation function to influence the decisions made by the default policy. They tested their agent, the MCTS-GP, with a variety of deck styles and found it more robust than the vanilla MCTS.

3.1.2 Evolutionary Algorithms in Monte Carlo Tree Search's selection phase

MCTS's selection phase uses a tree policy to select the nodes to explore next, making it a phase of particular research interest as it greatly influences the MCTS's performance [27]. Regarding offline evolution of the tree policy, the work by Cazenave [33] evolves a tree policy formula with GP for the game of Go. This approach successfully finds formulae that consider a wide variety of alternative statistics and heuristics, such as the All-Moves-As-First (AMAF) value of the move, some features of the board, or the best reward among the children of the parent node.

In [25], Bravi et al. propose three versions of a GP that evolves the tree policy for

Table 3.1: Approaches with Evolutionary Algorithms in Monte Carlo Tree Search.

Online/ Offline	**Modified in MCTS**	**Name**	**Game**	**Ref.**
Online	Tree policy, default policy	SA-MCTS$_{\text{NEA}}$	GVGAI	[158]
		SA-MCTS$_{\text{EA}}$	GVGAI	[158]
	Default policy	KB Fast-Evo MCTS	GVGAI	[127]
		Fast-Evo MCTS	GVGAI	[111]
	Tree policy	EA-MCTS, SIEA-MCTS	Carcassonne	[61]
Offline	Default policy	Hivemind	Hex	[133]
		(No given name)	Ms Pac-Man	[3]
		EvoMCTS	Othello, Dodgem, Hex	[17, 18]
		BGP	Go	[82]
		MCTS-GP	Hearthstone	[38]
	Tree policy	(No given name)	Go	[33]
		UCB$_+$, UCB$_{++}$, UCB$_\#$	GVGAI	[25]

MCTS in rogue-like real-time games from the GVGAI competition, categorising GP terminals considered for composing tree policy alternatives into three groups:

- Tree variables: are directly related to the constructed statistical tree, such as the number of visits or rewards of certain nodes.
- Agent variables: are related to a memory of the agent, such as its number of visits to the current location or an action repetition count.
- Game variables: that describe aspects of the game state, such as the number of “portals” or distances from the agent to a Non-Player Character (NPC).

With them, they propose three evolutionary-based tree policies: UCB$_+$ (using only tree variables), UCB$_{++}$ (using both tree and agent variables), and UCB$_\#$ (using tree, agent and game variables), with the latter surpassing the other two in the majority of the games. They experimented with the tree policies they found through evolution on 62 GVGAI games available at the time, finding that most of the evolved equations behaved rather similarly to UCB1 on average. They encourage evolving heuristics for clusters of games with similar design aspects.

Bravi’s approach was later employed to evolve different playstyles in the game Mini Dungeons 2, where 4 unique procedural personas are evolved offline [81]. A big issue of all the approaches that evolve a tree policy is that they demand exposure to the game beforehand to have enough time to perform their evolutionary process, as can be seen in Table 3.1, which summarises the algorithms discussed in this section.

3.2 Monte Carlo Tree Search in Evolutionary Algorithms

The versatility of EAs is achieved thanks to their numerous parameters, which are flexible enough to meet different requirements. The evolutionary process uses meta-heuristics and randomness to conduct the search. However, certain EA aspects can be controlled with ideas present in MCTS.

3.2.1 Monte Carlo Tree Search in Rolling Horizon Evolutionary Algorithms

There are multiple hybrids of Rolling Horizon Evolutionary Algorithm (RHEA) and MCTS [58, 83], where MCTS is used to improve certain aspects of Rolling Horizon Evolutionary Algorithm (RHEA). For instance, in [57], the algorithm called C-MCTS-S allocates half of the computational budget to run MCTS and then uses the best action path in its statistical tree to initialise the RHEA population, which uses the rest of the computational budget. Similarly, the Statistical Tree-based Population Seeding RHEA (STPS-RHEA) technique [64] initialises the population by following the path in the statistical tree that leads to the best reward, and the rest of the individuals are generated by running the Upper Confidence Bounds for Trees (UCT) tree policy. The tree policy has the potential to create a different individual every time as the statistical tree is updated with the evaluation of every previously seeded individual. To prevent stagnation due to the small population size, the technique also injects seeded individuals in every subsequent generation of the evolutionary process.

In [83], the authors propose five approaches to combine RHEA and MCTS in real-time games of the GVGAI framework, and they find these approaches to be more robust than the standard RHEA. In the chosen games, the agent controls a character in a 2D map with multiple interactable items or Non-Playable Characters (NPCs). The games may also include stochastic elements, such as unpredictable NPC movements. Their first proposed algorithm is a variant of RHEA called RHEA with rollouts (EAroll). It runs a small MCTS with a parameterised maximum search depth with its root set at the end of the sequence of actions of each RHEA individual. The rewards obtained from the MCTS are then averaged with those of the individual. The idea is to obtain a more accurate fitness evaluation for each individual by finding a potential plan after the evolved actions. The second variant is called RHEA then MCTS for alternative actions (EAaltActions). It first runs a RHEA to find a solution and then allocates a portion of the computational budget to a MCTS that searches for an alternative independent solution. Finally, the two solutions are compared, and the one with the better fitness is selected as the final decision. The third variant, called EAroll plus sequence planning (EAroll-seqPlan), incorporates a buffer to retain the plan from the previously executed action in EAroll, which is then used to initialise the search when the algorithm is called again. The fourth variant, EAroll plus occlusion detection (EAroll-occ), includes a check to identify inconsequential actions in the EAroll action sequences. If any such actions are identified, they are removed to potentially enhance the overall performance of the controller. The final variant, named EAroll plus NPC attitude check (EAroll-att), introduces awareness of NPCs present in the game, based on the assumption that their behaviours remain consistent throughout the game. The agent learns whether collisions with a specific NPC type have resulted in negative or positive consequences and uses this information to penalise or reward its proximity to each type of NPC.

3.2.2 Generating EA offspring using Monte Carlo Tree Search

In the context of adversarial games with complex turns, Evolutionary Monte Carlo Tree Search (EMCTS) [14] is a hybrid algorithm where the nodes of the statistical tree represent complete action sequences for a single turn. The edges of the tree represent mutations of the action sequences that lead to their offspring. As a result, EMCTS systematically explores the search space of an EA. The search process in EMCTS begins with a single individual, which serves as the root of the tree. The statistical tree is constructed using the MCTS algorithm, employing UCB1 as the tree policy. The action sequence represented by each node in the statistical tree is evaluated by simulating its actions and using a heuristic on the resulting state. The statistical tree of EMCTS exhibits a high branching factor due to the numerous possibilities for mutation for each individual. To tackle this issue, EMCTS incorporates Bridge Burning (BB) [93, 144], a pruning technique that divides the search into an arbitrary number of phases. Once the budget for each phase is exhausted, the best child of the root becomes the new root of the statistical tree. All nodes that are not part of the new statistical tree are discarded, allowing the search to proceed.

The main drawback of EMCTS is its limitation in planning for the current turn, which is addressed in its extension, Flexible Horizon Evolutionary Monte Carlo Tree Search (FH-EMCTS) [13]. In FH-EMCTS, the parameter *search horizon* is introduced, which increases the depth of the evolved solutions obtained by EMCTS. This extended depth includes the consideration of the opponent's atomic actions as well. While the actions from the current turn are evolved, the actions of the opponent's turn are generated using an arbitrary opponent model. The computational cost is optimised by limiting the actions to the top ten most promising ones and using a "lazy" approach for generating mutation actions during the tree search. In the game Hero Academy, FH-EMCTS outperformed Online Evolutionary Planning (OEP) and EMCTS (both explained before).

MCTSPO [113] is a neuroevolution technique that applies the UCB1 policy to select parents in the evolutionary process. It is a modification of the Deep Genetic Algorithm (DeepGA) [161], which can be classified as both an evolutionary algorithm (EA) and a tree

Table 3.2: Approaches with Monte Carlo Tree Search in Evolutionary Algorithms.

Modified in EA	**Base EA**	**Name**	**Ref.**
Offspring generation	Genetic Algorithm	EMCTS	[14]
	Genetic Algorithm	FH-EMCTS	[13]
	Deep Genetic Algorithm	MCTSPO	[113]
	Genetic Programming	Expansion	[86]
Initialisation and offspring generation	Rolling Horizon Evolutionary Algorithm	STPS-RHEA	[64]
Initialisation	Rolling Horizon Evolutionary Algorithm	C-MCTS-S	[57]
Fitness evaluation	Rolling Horizon Evolutionary Algorithm	EAroll	[83]
Final output	Rolling Horizon Evolutionary Algorithm	EAaltActions	[83]

search method. In MCTSPO, the evolutionary process is modelled as a tree, where each node represents an individual Neural Network (NN), and the edges connect parents with their offspring, similar to EMCTS. The only operator used in this process is mutation, meaning that each node will have exactly one parent, and each parent can have multiple offspring. Gaussian noise is added to the parent's weight vector during mutation, using a safe mutation technique [105] to ensure that the resulting individual remains close to the parent in the search space. It uses progressive widening [35] to control the number of mutations to explore for each NN in the tree. Candidate mutations are precalculated and stored as a vector, consisting of a random seed for the Gaussian distribution and a magnitude. By using this approach, the algorithm does not need to store the weights of the NN in the tree nodes but is still capable of reproducing any path in the tree to reach a desired state. In MCTSPO, MCTS is used to generate and explore the evolutionary tree with MaxUCT [96] serving as the tree policy. Instead of rollouts, a single evaluation of the NN is conducted. Experimental results showed that MCTSPO outperforms DeepGA and Trust Region Policy Optimization (TRPO) [148] in classic control environments and high-dimensional tasks in OpenAI Roboschool. However, as the task difficulty decreases, the performance gap between MCTSPO and the baseline algorithms narrows. They explain that this phenomenon occurs because less exploration is required in easier tasks, and gradient-based approaches can quickly identify optimal solutions.

More recently, a new type of mutation in GP, called expansion, was proposed in [86]. Expansion uses MCTS to grow the syntax tree of the individual being mutated. To control bloat, the reduction operator, which decreases the size of the syntax tree was introduced. The expansion and reduction operators achieved good results in symbolic regression problems benchmark problems. The approaches that use MCTS to influence any aspect of an EA are summarised in Table 3.2.

3.3 Multi-Armed Bandits in Evolutionary Algorithms

The selection phase of MCTS models every choice made to traverse the tree as a MAB problem. This implies that, by extension, EAs interact with the MAB model when MCTS is integrated with them. The interaction between EAs and MABs can have many forms and consequences relevant to our exploration of the interactions between MCTS and EAs. To this end, in this section, we extend our scope to include works that combine MABs with EAs, where the MAB is not necessarily employed under the context of the MCTS algorithm.

Interestingly, literature on the combination of EAs and MABs is scarce [108]. In their survey [43], the authors summarise the efforts made by researchers to address the exploration versus exploitation dilemma in EAs. However, they do not mention MABs, despite the intuition that the aforementioned dilemma could benefit from MAB models. Similarly, in a recent survey on practical applications of MABs by Bouneffouf et al. [24], EAs are not mentioned. In a work that discusses trends and challenges in EAs parameter optimisation [94], MABs are only mentioned in the contributions by Fialho et al. [51].

3.3.1 Evaluation of EA individuals using Multi-Armed Bandits

In problems where fitness evaluations are noisy, the evolutionary process may inadvertently discard useful individuals. To overcome this, resources can be distributed among individuals to evaluate them multiple times, allowing for more reliable decision-making. In this scenario, individuals are modelled as arms of a MAB, where a policy determines which individuals deserve to be evaluated again [97, 104, 136].

In [136], the authors apply the mentioned approach to Conversion Rate Optimisation (CRO) problems. The conversion rate represents the rate at which visitors to a website perform a target action, with a binary outcome of either success or failure. In this scenario, each web page configuration can be represented as an individual in an EA. The fitness of each individual is updated each time a user visits the web page. Due to the limited number of visitors, it is crucial to distribute them efficiently among the individuals in the population. It is worth noting that this setting results in individuals having different levels of certainty in their fitness, as their visit counts are often unequal.

The Collaborative Evolutionary Reinforcement Learning (CERL) algorithm proposed by Khadka et al. [97] uses the UCB1 policy to allocate a limited number of training epochs among a population of NNs. The fitness of each NN is based on its improvement after some epochs. The algorithm distributes its resources for further training among the less-trained networks and the most promising ones. By employing the UCB1 policy, the CERL algorithm has demonstrated better sample efficiency and achieved superior results compared to other methods. CERL is evaluated on 5 continuous robotic-control tasks hosted on OpenAI gym.

In [50], the resources used for individual evaluations are allocated based on fitness expectations, which are calculated by analysing the fitness distribution and the smoothness of the fitness landscape. In this case, the UCB1 policy is used to create filters that determine which individuals in an EA should be evaluated.

The UCB Random-restarts with Decreasing Step-size (URDS) [46] is an ES-based Multi-Modal Optimisation (MMO) algorithm that enhances the quality of the Quasi-random Restart with Decreasing Step-size (QRDS) algorithm [147]. URDS performs a local search with a unitary population until an unknown local-optima point is found. The population is then initialised one step away from that point, initiating a new local search. The step size decreases over time to transition from exploration to exploitation as more information about the search space is gathered. URDS improves the restart location of QRDS by partitioning the search space. It separates the search space into m^d areas using a regular grid with m subdivisions along each dimension d. Each area is treated as an arm of a MAB, which tracks the number of individuals initialised in it and the quality of the resulting optima points. When a restart is needed, the UCB1 policy is used to select the most interesting area in the search space. Then, a new individual is initialised in that area to initiate the local search. QRDS has shown competitive results in locally complex functions and functions with unevenly distributed optima points throughout the search space. However, it is limited because the number of subdivisions in the search space increases exponentially as the number of dimensions increases.

3.3.2 Generating EA offspring using Multi-Armed Bandits

The number of genetic operators in the field of EA continues to increase over time. However, selecting the appropriate operator is a challenge that is often arbitrarily determined by the user or tuned through trial and error. Furthermore, the impact of operators may vary at different stages of the evolutionary process. To address this issue, MAB have been proposed as a means to dynamically select genetic operators.

An interesting idea is the Adaptive Operator Selection (AOS) [51] method, which allows for the dynamic selection of genetic operators and their target individuals in each generation based on online information. AOS employs the UCB1 policy to determine the genetic operator to be used at any given moment during the evolutionary process. This decision is based on the historical impact of the operator on the performance of the population. Consequently, promising operators are selected more frequently, while even the least effective operators are not completely disregarded. AOS analyses updates to the UCB1 policy to adapt to the current location of the population in the search space. The rationale behind this is that operators that initially performed poorly might become useful at later stages of the search. Thus, the problem of selecting a genetic operator is modelled as a Dynamic Multi-Armed Bandit (DMAB).

The effectiveness of AOS was evaluated in the context of multi-objective optimisation in a study by Li et al. [106]. Multi-objective paradigms pose challenges in defining fitness improvement because an improvement in one objective may result in a decline in performance for other objectives. To address this issue, the Fitness-Rank-Rate-based Multi-Armed Bandit (FRRMAB) model was proposed, which enables the integration of AOS with the Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D). The FRRMAB scheme employs the UCB1 policy with fitness-rate-rank values as rewards. A variety of extensions to AOS have since been proposed [44, 51, 52, 53, 106, 115].

In a study by Liu et al. [108], a bandit-based mutation operator is introduced and used within a Random-Mutation Hill-Climber (RMHC) algorithm, referred to as Biased-mutation RMHC (B-RMHC). In this approach, the genome of an individual is represented as an array of bandits, with each bandit having as many arms as there are available values for that particular gene. For each gene, an urgency metric is computed based on the UCB1 policy. This urgency metric takes into account the potential improvement in fitness if a mutation is applied to the current state for each gene. The gene with the highest urgency value is then selected for mutation.

The Bandit-based Random-Mutation Hill-Climber (Bandit-based RMHC) is expanded to work with populations of more than one individual and non-binary genes in [58]. The authors compared different RHEA variants using the GVGAI framework [131]. The collection of enhancements to RHEA, mostly discussed in Section 3.2, include using the bandit-based mutation operator from [108], adding information from the statistical tree [132], using Monte Carlo simulations [83], and using a shift buffer to partially reuse the population generated on the next decision within the same game.

The NTBEA [104] is a RMHC variant designed to deal with noisy and expensive discrete optimisation problems. NTBEA generates a set of neighbouring individuals from a current solution using a mutation operator, seen as arms of a MAB. It then calculates the UCB1 value for each offspring using an N-Tuple Fitness Landscape Model, which stores the rewards and visits of each n-tuple of gene values. The iterative selection process picks the neighbouring individual with the highest UCB1 value. By using the N-Tuple model, it becomes possible to estimate the fitness of unsampled solutions, thus saving computational resources. NTBEA provides informative statistics for each parameter choice and their combinations, and also allows explicit control over the balance between exploration and exploitation in the algorithm [112]. We mentioned NTBEA in Section 3.1, demonstrating

Table 3.3: Multi-Armed Bandit approaches in Evolutionary Algorithms.

Modified in EA	**Bandit's arms**	**Name**	**Ref.**
Fitness Evaluation	Individuals	CERL	[97]
		NTBEA	[104]
		(No given name)	[136]
Offspring generation	Genetic operators	AOS	[51, 106]
		Bandit-based RMHC	[108]
		BGP	[82]
	Genes	Bandit-based RMHC	[108]
	States of a gene	Bandit-based RMHC	[108]
	Search space regions	URDS	[46]

the intersection between MABs and EAs when the latter are used to evolve MCTS.

NTBEA has been tested for automatic game improvement [104] and for the optimisation of game agents [110, 112]. In the context of automatic game improvement, the fitness function used is the difference in scores between agents with different skill levels. The hypothesis is that a game will be more enjoyable if the player feels a sense of improvement over time, as reflected in the difference in scores achieved by these agents. NTBEA has proven to be robust when compared to both vanilla RMHC and B-RMHC in [104].

In the aspect of optimising game-playing agents, the fitness function measures the performance of the agent across a set of games. In a recent study [112], NTBEA was compared to Sequential Model-based Algorithm Configuration (SMAC) [84], Grid Search, and a multi-valued version of Sliding Window compact Genetic Algorithm (SWcGA) [110]. The results showed positive outcomes for NTBEA. A mix of a Compact Genetic Algorithm (cGA) with MABs is proposed as future work in [110]. Table 3.3 summarises the different ways in which MABs have been used to influence EAs.

3.4 Online evolutionary-based planning in games

Although EAs have been used to play games and to improve the performance of game-playing agents with offline optimisation as explained in Section 3.1, we want to emphasize that the use of EAs for online decision-making in games is far less common. It is common to find EAs being used to learn specific aspects of a game offline, such as heuristic evaluations of game states or evolution of behaviour trees [128, 123]. The optimised models can then be used with other algorithms that are better suited for decision-making. EAs have been sucessfully used to evolve NNs offline for playing games like 9x9 Go [140], checkers [36, 37], othello [121], and chess [55]. Another approach involves optimising plans by learning from a database of human-explained strategies, which can be used to play real-time games, as in the Stochastic Plan Optimisation (SPO) [169]. On the other hand, it is challenging to incorporate EAs for online decision-making in games, because of the constrained computational resources. The efficiency of decision-making agents is a primary concern, especially in real-time games, but even in turn-based games, resource limitations can pose challenges. However, certain EA variants, such as the RHEA [129] or the RMHC, have shown robustness and can compete with MCTS under similar constraints in some domains. In this section, we summarise some EA approaches used for online decision-making in games.

RMHC [108] is one of the simplest forms of an EA used in games where the genes of a single individual, modelled as a sequence of actions, are mutated on each generation to generate offspring. The best individual is kept as a parent for the next generation, and the process is repeated. The output of the algorithm is the first action of the best individual. Improvements for this algorithm were presented in Section 3.3.

RHEA evolves a population of action sequences and outputs the first action in the best individual when stopped. RHEA has unique characteristics that make it well-suited for certain games. The length of the solutions can be adjusted to address sparse rewards by increasing the likelihood of sampling states that provide useful information for the search. However, like any EA used in real-time games, RHEA has limitations due to small population sizes and a limited number of generations, emphasising the need for an effective search within the solution space. To enhance RHEA's performance, several extensions combine it with MCTS's statistical tree [58]. This combination allows for tracking rewards and producing better-informed solutions.

Other EAs used for online decision-making include OEP [91] and Real-time NeuroEvolution of Augmenting Topologies (rtNEAT) [160]. OEP is an EA designed to construct sequences of atomic actions in games with complex turns, similar to RHEA. It evolves a plan for a full turn, which consists of multiple atomic actions. The full action sequence of the best individual is returned and executed at the end. OEP has been tested with good results in real-time games like StarCraft [172] and strategy card games like Hero Academy [93]. Its extension, Continuous Online Evolutionary Planning (COEP) [92], enhances both OEP and RHEA by continuously improving solutions with newly acquired information during gameplay.

On the other hand, rtNEAT evolves NNs that guide the behaviour of the agent in real-time games. It replaces the current network with a more complex one after a few game ticks, maintaining a small population that is evolved in parallel. On the same track as rtNEAT, it is common to use EAs for neuroevolution [59] and subsequently employ the evolved NN in a game. For instance, in [165], a neural agent is evolved to control a simulated racing car. Another example can be found in [1], where Multi-Objective Genetic

Table 3.4: Historical instances of machines defeating expert humans in games

Year	Game	Aproach	Reference
1979	Backgammon	BKG 9.8	[20]
1994	Checkers	Chinook	[145]
1997	Chess	DeepBlue	[31]
1997	Othello	Logistello	[30]
1998	Scrabble	Maven	[151]
2016	Go	AlphaGo	[154]
2019	DotA 2	OpenAI Five	[21]

Programming (MOGP) is used to evolve racing car controllers that are not based on NNs. Monte Carlo Tree Search Networks (MCTSnets) [75], makes MCTS adaptive with Deep Neural Networks (DNNs) that learn the evaluation of the states, which statistics to track at each node, the backpropagation operator and the expansion policy. Finally, another interesting EA-based planning approach is Fast Random Genetic Search (FRGS), which evolves plans for individual units in Real-Time Strategy (RTS) games.

3.5 Artificial Intelligence-based decision-making in games

Some of the most significant achievements of Artificial Intelligence (AI) agents defeating human experts in their games are listed in Table 3.4. These human-machine confrontations have historically captivated the masses as they serve as milestones reflecting progress in the field of AI [32].

Games with increased complexities take longer to be conquered by AI agents. However, in recent years, AI capabilities have been increasing, as have human expectations of them. When DeepBlue [31] defeated the chess world champion in 1997, doubts arose on whether or not the machine had humans helping it. With time, the strength of chess computers could no longer be questioned, and machines became virtually unbeatable by any human player. Nowadays, the most important chess tournaments broadcasted online feature a “machine evaluation bar”, which shows who has the upper hand in a game with an approximate evaluation of the game state. Thus, the machine’s evaluation is the closest we have to the ground truth, but it is likely still inaccurate due to the complexity of chess. Machines have surpassed humans in chess to the point where these evaluations are, on occasion, incomprehensible to humans. To achieve the machine’s assessment of a position, a player would sometimes need to find a series of incredibly complex moves. This makes machines unrealistic training partners for humans seeking to improve their game, as they will make moves that other humans are unlikely to make. Maia [116], an AI model trained to imitate human playing styles, illustrates the types of positions that are challenging for humans and the types of mistakes they can make.

AlphaGo [154] defeating Lee Sedol in the game of Go in 2016 was a significant breakthrough for AI agents, given the large branching factor of the game of Go and the lack of human understanding of it. AlphaGo trained two DNNs on games of human experts to learn state evaluations (value network) and the likelihood of a move being a good candidate for exploration (policy network). AlphaGo Zero [155] was later developed and managed to beat AlphaGo without any human knowledge, relying solely on self-play. AlphaZero [153], the next generation of Alpha agents, was able to learn chess and shogi, and defeated the best AI agents in these games by training exclusively with self-play. Most recently, in 2019, OpenAI Five [21] defeated the world champion team of Defense of the Ancients (DOTA) 2 players. The human team consisted of 5 players who competed against the AI agent in a best-of-three series. The AI agent, trained through self-play, managed to win two consecutive games. Dota 2 is a real-time 5 versus 5 game, where the AI agent had to control 5 different characters in real-time, each with their own set of skills and items, and each able to move freely in any direction throughout the map.

3.5.1 Games used for research in Artificial Intelligence

Multiple games have been used for AI research and some platforms offer multiple games as training grounds for AI agents. For example, the GVGAI competition [131] offers a new set of real-time videogames in each edition, all built using the Video Game Descriptive Language (VGDL) [146]. The competition aims to have agents compete and determine which one excels at General Video Game Playing (GVGP). Ludii [134] offers a diverse range of abstract turn-based games, and also provides a scheme for defining games with a set of “ludemes” (units of game information) that can be learned and considered by AI agents. OpenAI Gym [26] offers problems in classic control, algorithmic, 2D and 3D robots. It also features board games and Atari games with the integrated Arcade Learning Environment (ALE) [16]. Other notable pages include Zillions of Games [41], General Game Playing Base (GGP-Base) [143], the Mario AI competition [166] and Unity ML agents [90].

Some games have been developed for research, such as Legends Of Code and Magic (LOCM) [102], which is a simplified version of Hearthstone. Hearthstone is a popular Collectible Card Game (CCG) with complex turns, stochastic events, and imperfect information. In comparison, LOCM features a draft-based deck-building phase and simplified gameplay. For research in RTS games, platforms like Extensive, Lightweight and Flexible (ELF) offer games like Mini-RTS, Capture the Flag, and Tower Defence [164]. Mini-RTS, for example, is a miniature version of StarCraft, which involves resource gathering, fog of war, building troops, and commanding them to attack or defend on a 2D map. In a work by Andersen et al. [6], a summary of RTS games and platforms used for Reinforcement Learning (RL) is presented, including their performance-focused game, Deep RTS. Various RTS games, among others, are included in the IEEE Conference on Games (CoG) competition, which is held annually and features a variety of games such as Bot Bowl (a simulation of Blood Bowl, a board game with sparse rewards), Tales of Tribute (a deck-building card game where players acquire cards from a shared pool), the Video Game Championship (VGC) AI competition (a Pokèmon team-builder and battler), Dota 2, StarCraft, microRTS, and the multi-agent Google Research Football competition.

In recent years, there has been an increase in research using newly available euro-style tabletop games such as Settlers of Catan, Ticket to Ride, and Carcassonne. Although most of these games are multiplayer, there is a preference for limiting the player pool to two. Carcassonne, in particluar, remains a challenge where few AI-based approaches have been attempted [5, 19, 78]. It is a game with scalability potential that offers a variety of strategies, making it interesting for AI research and is one of the games used in this work. A detailed introduction of the game of Carcassonnne will be presented in Chapter 4.

Part II
Contributions of this thesis

4 Test problems and their analysis

Related publication: Fred Valdez Ameneyro, Edgar Galván, and Ángel Fernando Kuri Morales. "Playing carcassonne with monte carlo tree search". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2020, pp. 2343–2350

4.1 Introduction

In this chapter we undertake a comprehensive analysis of game tree characteristics that significantly influence the tree policy of Monte Carlo Tree Search (MCTS), focusing on both the reward distribution and structural properties of the game tree. This analysis is critical for selecting and developing appropriate test problems for evaluating MCTS selection policies and for guiding the development of new MCTS variants.

To effectively measure and analyse the rewards likely to be encountered in these problems by MCTS algorithms, we introduce the initial belief value. This metric evaluates the potential outcomes of the default policy within a domain, providing insights into how these outcomes can steer the decision-making process of the MCTS. We propose the use of the Function Optimisation Problem (FOP), a synthetic domain that models continuous search spaces as decision trees, and Carcassonne, a game with intricate strategic elements and a unique feature set.

FOP is a toy problem where the rewards can be arbitrarily set. Furthermore, its game tree has a fixed branching factor and fixed tree depth, which allows for a focused analysis of the impact of the tree policy on the search. FOP has been previously used to analyse the behaviour of decision-making algorithms like the Hierarchical Optimistic Optimisation (HOO) in [29] and MCTS in [87]. In this chapter, we use FOP to analyse the impact of the tree policy on the search and to compare the performance of MCTS variants.

Despite being relatively underexplored in academic research, Carcassonne offers rich opportunities for a detailed evaluation of MCTS algorithms due to its complex nature and strategic depth. Indeed, we have used Carcassonne in our recent paper [5], including an

Table 4.1: Game tree characteristics that influence MCTS's tree policy behaviour

Classification	Characteristics
Tree structure	Abrupt termination states
	Tree depth
	Branching factor
	Game state equivalences
	Progression
	Transpositions
Reward distribution	Bias in suboptimal moves
	Optimistic moves
	Shallow search traps
	Smoothness
	Sparse rewards

IEEE Transactions on Games article [61], demonstrating its relevance. Besides the base game, we propose simplified Carcassonne single-player variants designed to retain the core strategic elements of Carcassonne while reducing complexity, allowing for a focused analysis of their reward landscapes in the context of MCTS research.

4.2 The tree policy and its interaction with the game tree properties

The success of MCTS in different areas mainly depends on how well its tree policy works. This interaction has been a major focus in research to make MCTS better, as discussed in many studies [27, 162]. Table 4.1 shows a list of game tree features that affect how well MCTS performs. These features are grouped based on whether they describe the structure of the tree or its reward distribution.

It should be noted that all the characteristics listed in Table 4.1 are specific to games with atomic turns. As explained in Chapter 2, a turn is considered atomic if a single decision is required from the decision-maker before the turn (or frame) ends. A detailed discussion of each of these properties is presented next.

- **Abrupt Termination States**. According to [120], abrupt termination states refer to pronounced asymmetries in the depth of the game tree, and their existence can impact the growth of the statistical tree as well as lead to variations in reward certainty. When abrupt termination states are added to the statistical tree, they influence the comparisons made between nodes. While the evaluation of any terminal state is certain, the vanilla version of MCTS treats them similarly to any other node, iteratively sampling them again. In this way, they waste resources when selected multiple times and hinder information gain. As abrupt termination states get closer to the root, the amount of

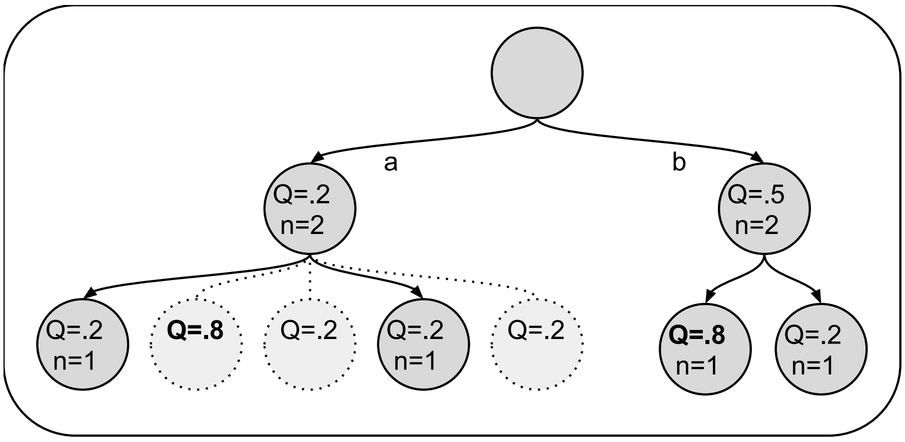

Figure 4.1: Sample statistical tree generated by an MCTS algorithm, where each node has a reward Q and a visit count n. The dotted nodes represent states from the game tree that have not been added to the statistical tree yet.

wasted resources is also likely to increases. One solution to this issue is to propagate the reward certainty to the parent node, as the MCTS-Solver [174] does.

- **Tree Depth**. The number of turns a game has impacts the rollout resources and the volatility of its rewards. The reward certainty of a node is lower as the rollout simulates more moves because the sampled outcomes become less likely. Thus, uneven depth among branches of the tree causes certainty disparities among nodes, a phenomenon rarely addressed by MCTS research. Moreover, in cases where a game terminal state cannot be reached a heuristic evaluation may be necessary to predict the result instead of full playouts. Sometimes, this heuristic is used to evaluate states that are found after a certain number of turns from the state being evaluated. This is known as early termination, which needs a cut-off depth parameter that can be adjusted to enhance the performance of MCTS [109].
- **Branching Factor**. In MCTS, the classical tree policy stops the selection process when it encounters a node that can be expanded. As the branching factor increases, every node has more edges to explore and the exploitation decreases; because most of the iterations are unable to reach deeper levels of the game tree [125]. Additionally, the presence of an uneven number of edges throughout the game tree can result in unfair comparisons between nodes. The following example will illustrate the impact of the branching factor on tree search algorithms.

  Let us consider the statistical tree from Figure 4.1, where the states represented by nodes a and b have different branching factors. Moreover, both nodes a and b have a child node with a similar maximum reward, $Q_{max} = 0.8$. In this scenario, node b is more likely to yield the rewards from its best child upon expansion, making Q_b larger than Q_a on each subsequent iteration. In other words, $P(Q_a < Q_b) > \frac{1}{2}$ will most likely remain true at all times. This implies that, despite their shared maximum reward, node b will be preferred over node a by tree policies like Upper Confidence Bounds (UCB1) that do not account for the branching factor.

- **Progression**. Some game domains might have loops in their game trees. This means that actions do not always move the game closer to an end. A game is called *progressive* [54] if it always naturally reaches an end. On the other hand, a game is *non-progressive* if it can get stuck in a loop. Such a loop happens when a game state s is reached more than once during a single playthrough. In simpler terms, the same state s appears again in the subtree that also starts from s.
  Sometimes, progressive games have rules to limit repeating the same state or move. For example, in Chess, repeating the same game position three times results in a draw. If a game can end early due to a limit on repeating states, its game tree might have "abrupt termination states", which are explained later in this section.

According to [39], while the following game characteristics do not directly affect MCTS performance negatively, accounting for them can be beneficial:

- **Game State Equivalences**. Consider two game states, s_i and s_j. They are considered equivalent if their corresponding subtrees share the same cardinality and rewards. This often happens due to symmetries in the game, like in Go-Moku, Othello, or Go. Removing such equivalent states from the statistical tree can simplify the task of tree search algorithms for these games by reducing its complexity.
- **Transpositions**. Are repeated game states reached through different sequences of actions. Essentially, a state s shows up in several parts of the game tree, excluding each of its corresponding subtrees.

Game state equivalences and transpositions are normally only considered when the domain is known to have them. Addressing them requires sharing information across different parts of the tree during the search for identification and the addition of rules to the tree search algorithms. It is not necessary to avoid them in test problems for MCTS variants, as they do not hinder the performance of the algorithm. We consider them in some instances of the test problems proposed in this chapter to demonstrate their impact on the search.

Other features impacting MCTS performance are based on how rewards are given to game states. These include:

- **Bias in suboptimal moves**. Describes the comparison of the rewards of the less effective moves between players. It is easier to identify an optimal move when there is a big difference between its rewards and that of the suboptimal moves. Moreover, when suboptimal moves have relatively larger rewards, they become harder to ignore for the search algorithm. The disparity of rewards between the suboptimal moves of the players produces a bias. Such bias can make finding good moves harder for one player, creating an unfair situation in which one player is at a greater risk than the other [85].

- **Smoothness of the underlying reward distribution**. In MCTS, the reward of any node is approximated by averaging the results of all the rollouts executed from the subtree with that node as the root. Therefore, a good action from a parent node can be undersampled if its alternatives perform poorly because the reward of that parent node averages them out, making the parent node less appealing to the search. Thus, a node with volatile rewards might have a lesser average reward as a consequence than a node with more consistent rewards. Rewards are expected to be consistent if the underlying value function is smooth [87]. Otherwise, the rewards will have a high variance.
- **Shallow search traps**. A state s_{trap} is considered a k-*level* shallow search trap for player p if its opponent has a winning strategy that ends the game in at most k plies. In other words, player p cannot prevent a loss from s_{trap} if their opponent follows the winning strategy. If a state s_{risk} has an available move (or multiple moves) that leads to a shallow search trap s_{trap}, then player p is said to be *at risk* at state s_{risk}.
  The word “shallow” refers to the number of actions needed to complete the game starting from the search trap. If the opponent can secure a favourable result after k plies, it is called a k-*level* shallow search trap. Shallow search traps are states in adversarial domains that are particularly challenging for MCTS algorithms to handle, as stated by [137, 138].
  In games like Chess, shallow search traps are present in almost every stage of the game in the form of *checkmate in k moves*, which also lead to abrupt termination states. Although there is no defined shallow search trap density threshold, games are deemed **tactical** if their game trees consistently contain identifiable k-level shallow search traps, (i.e. with $k \in 3, 5$).
- **Optimistic moves** are moves that seem favourable initially but can be refuted by the opponent, leading to undesirable game states. Optimistic moves mislead the search until their refutation is discovered, thus having detrimental effects on MCTS by making the algorithm take longer to converge to the minimax value [54]. In MCTS, the evaluation of a node s_i tends to converge towards the value of its most promising child, as that child is more frequently sampled during the search process than its siblings. As a result, optimistic moves can adversely affect the tree search by propagating inaccurate evaluations and hindering the efficient use of computational resources. The search will be increasingly impacted by the optimistic move as the difficulty in identifying the refutation increases. This demonstrates that the average reward may not always be a suitable evaluation for a state, which has led to the exploration of alternatives in literature [33, 25].
- **Sparse rewards**. Standard MCTS rollouts operate under the assumption that every

path in the search tree can ultimately yield a reward. In domains with sparse rewards, like non-progressive games, standard MCTS rollouts might struggle to consistently retrieve a reward, leading to an ineffective tree search. Common approaches to address sparse rewards are heuristic evaluations or sub-goals [66].

These reward-based game tree characteristics are largely influenced by how nodes are evaluated, and the information distributed via the backup operator.

We now proceed to define which tree structure characteristics are desired in domains used for testing the behaviour of the MCTS's tree policy and their variants, based on our previous discussion. We want to be able to compare the exploration versus exploitation balance, a key component of MCTS, which requires tracking the construction of the statistical tree. To this end, we chose domains with a constant tree depth, implying that the domains should not have abrupt termination states which could drastically influence how the statistical tree is built. Equally, the game must be progressive and without sparse rewards, to ensure good quality rewards, as we are interested to see how MCTS makes use of the information acquired. For interpretability purposes, our MCTS variants do not address transpositions and game state equivalences. Finally, we want to be able to compare the performance of the MCTS variants with a combination of different reward distribution characteristics (bias in suboptimal moves, optimistic moves, shallow search traps, and smoothness). Therefore, we chose domains that exhibit these characteristics to different degrees and allow us to manipulate them.

The following section presents the FOP, the first problem used in this work, which is a synthetic domain that can be used to model continuous search spaces as decision trees. The FOP is used to analyse the performance of MCTS across various reward structures while providing insights into the underlying fitness landscape.

4.3 Test problem: Function Optimisation Problem

The Function Optimisation Problem (FOP) has a game tree representation with standard characteristics. These include a small and consistent branching factor, a constant tree depth, a progressive nature, a fully symmetrical tree, and no abrupt termination states. It was inspired by an analysis on tree search in [29], first tested with MCTS in [87] and further explored in our recent work in [4].

The FOP provides a means of modelling domains with continuous search spaces as decision trees, which can be solved using planning algorithms. It also enables the visualisation and interpretation of the underlying reward landscape, which facilitates the analysis of the distribution of locally-expanded nodes by any tree search algorithm. Furthermore, the FOP is a simple problem with known rewards that enables comprehension of the interplay between the game tree and the tree search agents. A definition of the FOP is

Table 4.2: Function Optimisation Problem definition

Aspect	**Description**
Objective	Find the global optima (maximum) of the function $f(x)$, where $x \in \mathbb{R},\ a_0 \leq x \leq b_0$.
States	Each state s_i is a subdomain of the initial domain, with $s_i = [a_i, b_i]$. A state is considered terminal when its domain width is smaller than a given threshold, i.e., when $b_i - a_i < t$. The reward r of any terminal state is determined by a sample from a Bernoulli distribution, with $r_i \sim Bern(f(c_i))$, where c_i is the state's central point, i.e. $c_i = \frac{(b_i - a_i)}{2}$. In order to ensure that $f(x)$ can be used as a probability, $0 <= f(x) <= 1 \vert x \in [a_0, b_0]$ must hold.
Actions	From any state s_i, each available action leads to one of k possible partitions of the current domain, where k is the branching factor. For instance, if $s_i = [a_i, b_i]$ and the branching factor is $k = 2$, each available action will lead to one of two possible next states: $s_{j1} = [a_i, \frac{(a_i + b_i)}{2}]$ or $s_{j2} = [\frac{(a_i + b_i)}{2}, b_i]$. Any state s_j that is available from s_i will have a size of $\frac{a_i - b_i}{k}$.

presented in Table 4.2.

In the FOP, each state represents a subdomain for variable x. The initial state in our case starts at $[a_0, b_0]$, where $a_0 = 0$ and $b_0 = 1$. The available actions are always a set of k evenly spaced partitions of the domain of the current state, where k is the branching factor. Each partition's size is $(b_i - a_i)/k$. The objective is to find the state in which the global maximum of a given function $f(x)$ occurs, with $x \in \mathbb{R}$ and $a_0 \leq x \leq b_0$. A state is considered terminal if its domain is smaller than the threshold t, which is set to $t = 10^{-6}$.

The MCTS rollouts use a random uniform default policy to choose actions. Once a terminal state s_t is reached, f is evaluated at the central point of the state c_t. The reward r_s can either be 1 or 0 and is sampled from a Bernoulli distribution, with $r_s \sim Bern(f(c_s))$. Therefore, since $f(x)$ is used as a probability, it is ensured that for the 5 functions used in this chapter, the function $f(x)$ satisfies $0 \leq f(x) \leq 1$ for all x in the domain $[0, 1]$.

Figure 4.2 illustrates a portion of the game tree for the FOP. When the branching factor is $k = 2$, the game tree becomes a binary tree.

4.3.1 Test Functions

One peculiarity of the FOP is that slopes in the function do not influence the search. Instead, the search is directed by the average of the function in each node's subdomain.

To refine our discourse, we introduce a new term, which we will refer to as the **initial belief value**. The initial belief value of node n_i, denoted as μ_i, represents the average of the initial reward assigned by an MCTS agent when node n_i is first expanded and

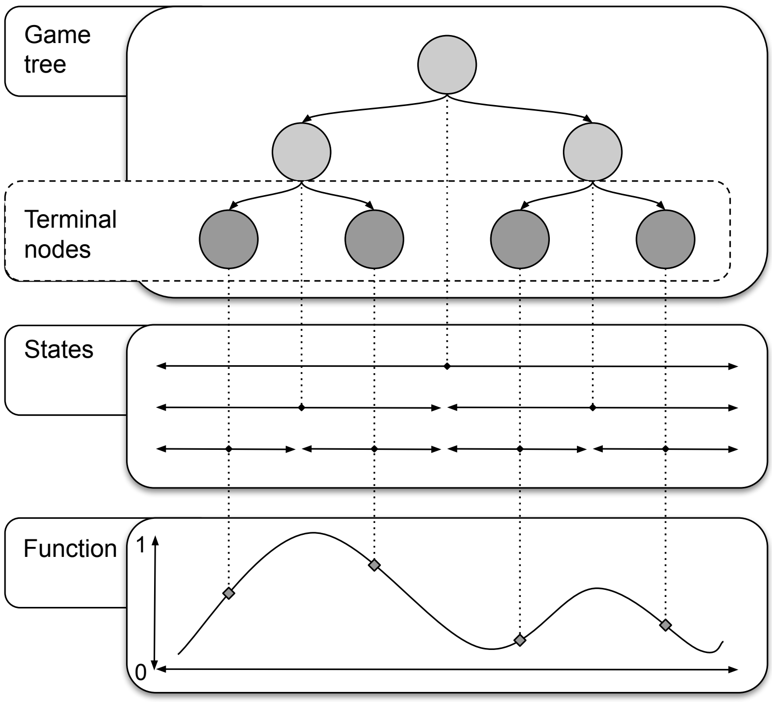

Figure 4.2: Game tree of the Function Optimisation Problem with a maximum depth of two and a branching factor of two. Each node represents a domain of the function, illustrated in "States". Actions lead to evenly distributed subdomains of the current state's domain. When a domain is smaller than a threshold, the state is said to be terminal. The evaluation of any terminal state is made at its central point, as shown in the "Function" box at the bottom of the figure.

included in the statistical tree. Therefore, μ_i depends on the evaluation method of each node, which has an unknown distribution $\hat{D}_i$. Note that when more nodes are expanded from node n_i and its reward is updated, the new reward will no longer be a sample from $\hat{D}_i$ given the tree policy's influence.

The initial belief value has a great impact on how the statistical tree is built by MCTS, as it shows how each node is likely to bias the search when it is first incorporated into the statistical tree. It can be said that when a node has a greater initial belief value than its siblings it will attract the search initially, as its exploitation term is likely to be greater than the exploitation term of its siblings, at least on the first few iterations after they are expanded. Hence, by knowing the initial belief value, we can predict in which order the statistical tree will be built. That knowledge gives us the arguments to claim whether a problem is deceptive or not for MCTS, as we will see in the next section.

The initial belief value of a node can be very different from the game-theoretical value. The FOP problem allows us to compare the initial belief values with the game-theoretical values of each node in the game tree, given that both values can be predicted. In FOP, the

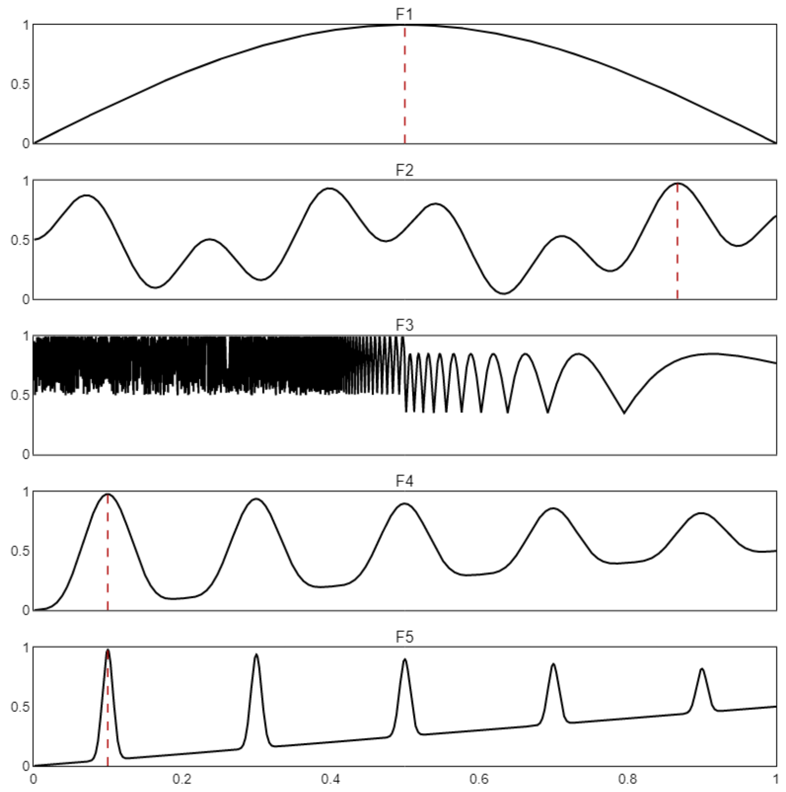

Figure 4.3: Plots of the functions used in the Function Optimisation Problem. The vertical dashed red line illustrates their global maxima, except for f$_3$, as it has multiple optima on the first half of its domain. The domain (x-axis) and range (y-axis) for each function extend from 0 to 1.

game-theoretical value of a node is the value of the maximum available in the subdomain that it represents, and its initial belief value is calculated as illustrated in Figure 4.4.

When MCTS uses a Monte Carlo simulations as its default policy, we calculate the initial belief value of node n_i as the average evaluation of every terminal state reachable from n_i. In the FOP, the initial belief value of a node n_i, called μ_i, converges to Equation 4.1 as the size of the terminal states decreases.

$$\mu_i = \frac{\int_{a_i}^{b_i} f(x)dx}{b_i - a_i} \tag{4.1}$$

where a_i and b_i are the beginning and end of the domain represented by node n_i respectively, and $f(x)$ is the function of the FOP. Figure 4.5 presents an analysis of the initial belief values of the game tree of f$_1$, formally expressed in Equation 4.2. The same analysis is done for the rest of the functions in Figure 4.3. These analyses illustrate the expected rewards that MCTS algorithms will find when traversing the game tree.

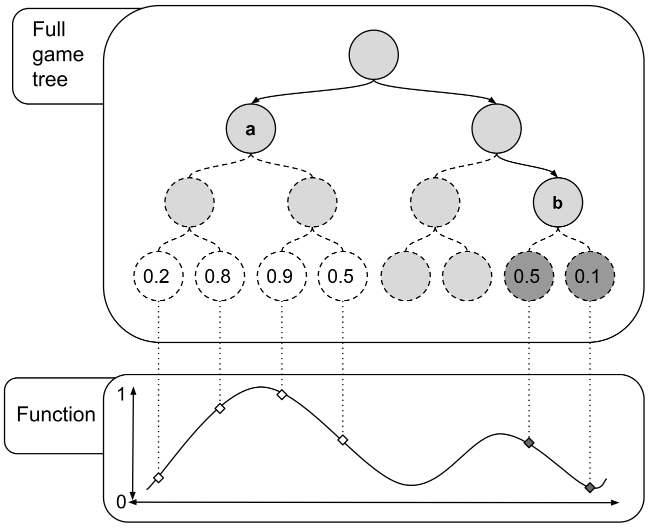

Figure 4.4: The initial belief value of a node in FOP when MCTS uses Monte Carlo simulations as its evaluation method is approximated by averaging the outcomes of all the reachable terminal states from that node, as every terminal is equally likely to be sampled. The dashed nodes have not been added to the statistical tree. Node *a* has 4 unique reachable terminal states, while node *b* has 2.

4.4 Definition of the Functions and their analysis

f_1 shown at the top of Figure 4.3 and plotted again at the top of Figure 4.5 for convenience, depicts a unimodal function. The global optimum is represented by a vertical dashed red line. The function is defined by Equation 4.2.

$$f_1(x) = \sin(\pi x) \tag{4.2}$$

There is only one global optimum in f_1, plotted at the top of Figure 4.5, and it can be accessed from both sides of the search when the branching factor is equal to 2. This function can aid in comprehending how MCTS behaves when multiple branches of the tree have the same rewards, how biased the search can become owing to the randomness of the rewards, and whether both sides of the search converge to an even ground.

f_2, proposed in [29], depicted at the second plot from top to bottom in Figure 4.3 and plotted again at the top of Figure 4.6 for convenience, is a multimodal function with a single global optimum and multiple local optima. The function is defined by Equation 4.3.

$$f_2(x) = 0.5\sin(13x)\sin(27x) + 0.5 \tag{4.3}$$

This function is characterised by a smooth profile across its entire domain, with distinct gradients that aid in identifying optimal regions. It helps to exemplify resource

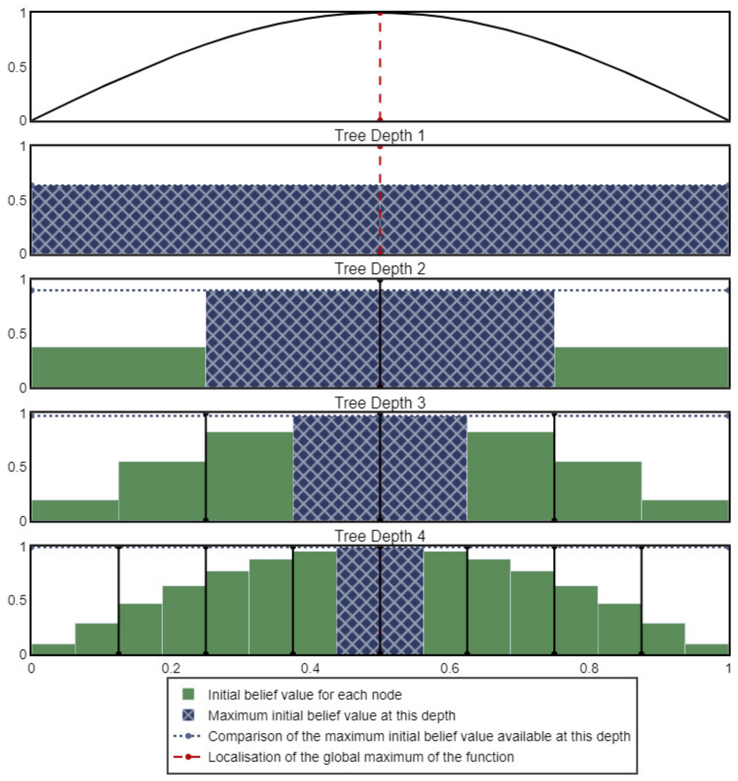

Figure 4.5: Initial belief values of the nodes in the game tree of f_1, plotted at the top. Each bar in the graph represents the initial belief value for each node within the tree. Sibling nodes are enclosed in the same black box. The horizontal dotted line represents the maximum initial belief value at each depth, facilitating comparison with other bars.

allocation when multiple easily found optima show similar levels of significance.

Figure 4.6 indicates that the rewards around the global optima are not the most favourable for the search up until a tree depth of 5. Up until tree depth 2, the best rewards come from the right-hand side of the domain, but in tree depths 3 and 4 the highest rewards are attainable on the left-hand side, potentially attracting the search's focus. This means that the tree search will invest resources on the left-hand side of the domain until the rewards on the right-hand side become more significant by sampling the best node at tree depth 5.

f_3, illustrated at the third plot from top to bottom in Figure 4.3 and plotted again at the top of Figure 4.7 for convenience, is a rugged function inspired by [87] characterised by several global optima scattered throughout the first half of the domain from left to right. As the domain ranges from 0 to 0.5, the optimal points become progressively less dense. Conversely, there is a smoother but less rewarding region in the range of 0.5 to 1. Equation 4.4 defines this function.

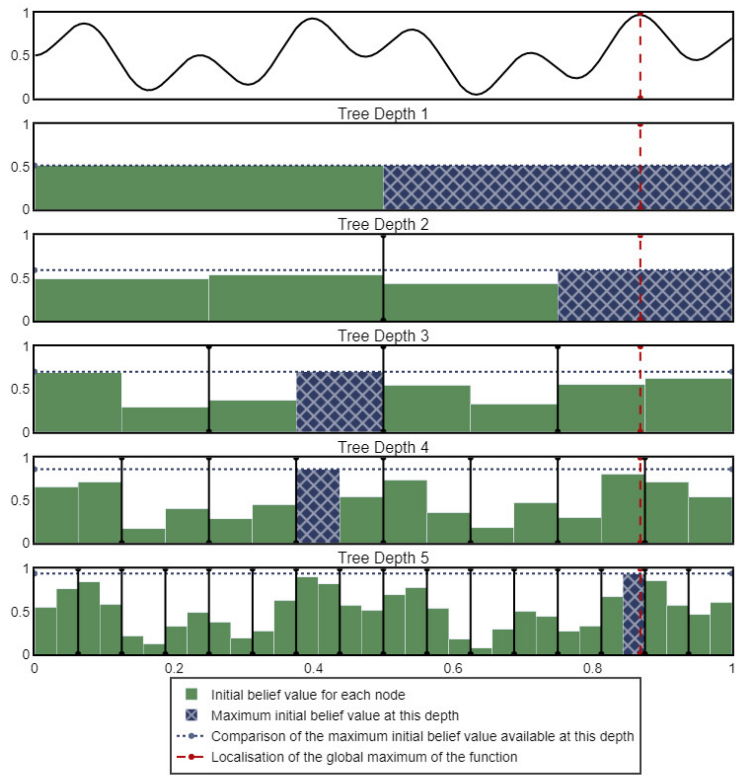

Figure 4.6: Initial belief values of the nodes in the game tree of f_2, plotted at the top. Each bar in the graph represents the initial belief value for each node within the tree. Sibling nodes are enclosed in the same black box. The horizontal dotted line represents the maximum initial belief value at each depth, facilitating comparison with other bars. The vertical dashed red line shows f_2's global maximum.

$$f_3(x) = \begin{cases} 0.5 + 0.5|\sin(\frac{1}{x^5})| & \text{when } x < 0.5, \\ \frac{7}{20} + 0.5|\sin(\frac{1}{x^5})| & \text{when } x \geq 0.5. \end{cases} \tag{4.4}$$

The hypothesis in [87] was that MCTS will be inclined to explore the smooth region of the function rather than the rugged region in order to minimise risk. This notion arises from the assumption that the search might encounter unfavourable rewards near the optima at the rugged portion of the function, potentially discouraging the search from further exploration in those areas.

Figure 4.7 shows that the search will initially focus on the rugged area because the maximum initial belief values are present on the left-hand side of the function for tree depths 1 and 2. At Depths 3 to 6, the right-hand side of the function becomes more attractive and will get more attention once they are reached, as it contains the nodes with the largest initial belief values.

The search will not be able to identify any global optima on the rugged side of the

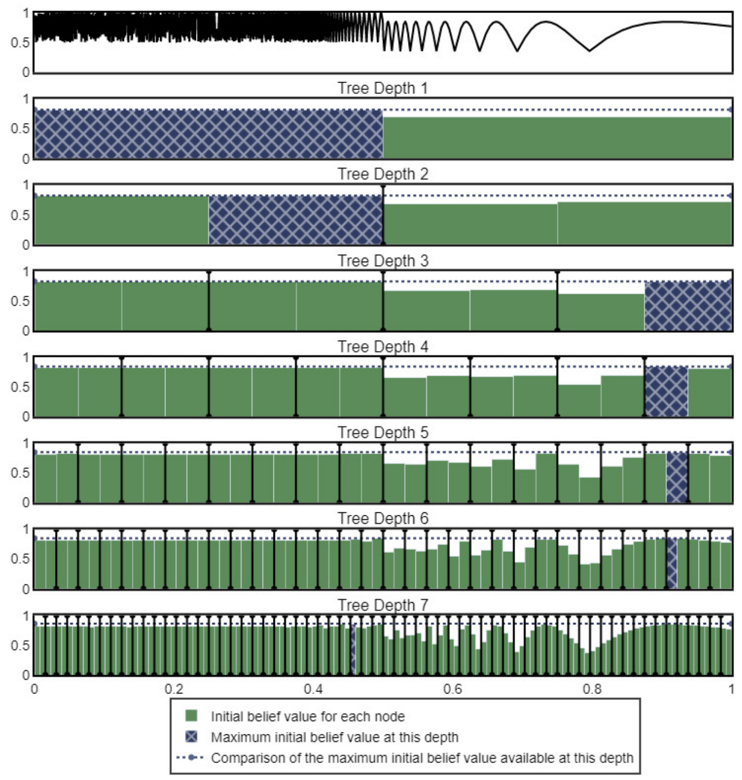

Figure 4.7: Initial belief values of the nodes in the game tree of f_3, plotted at the top. Each bar in the graph represents the initial belief value for each node within the tree. Sibling nodes are enclosed in the same black box. The horizontal dotted line represents the maximum initial belief value at each depth, facilitating comparison with other bars.

domain until it reaches a tree depth of 7. Even at this depth, the maximum values of the smooth region remain competitive, making it hard for the algorithm to focus on approximating more to the global optima.

f_4, is one of our proposed functions, denoted as f_4 in the fourth row from top to bottom in Figure 4.3 and plotted again at the top of Figure 4.8 for convenience, is a deceptive function that contains multiple local optima. Equation 4.5 defines this function.

$$f_4(x) = 0.5x + (-0.7x + 1)\sin(5\pi x)^4 \tag{4.5}$$

Analysing Figure 4.8, we can see that the maximum initial belief values are present on the right-hand side of the domain for tree depths 1,2 and 3, whereas the global optimum is on the left-hand side. This means that the global optimum of f_4 will remain concealed within an initially less rewarding region of the domain until it is eventually sampled, making f_4 a deceptive function designed to guide the search away from the global optimum.

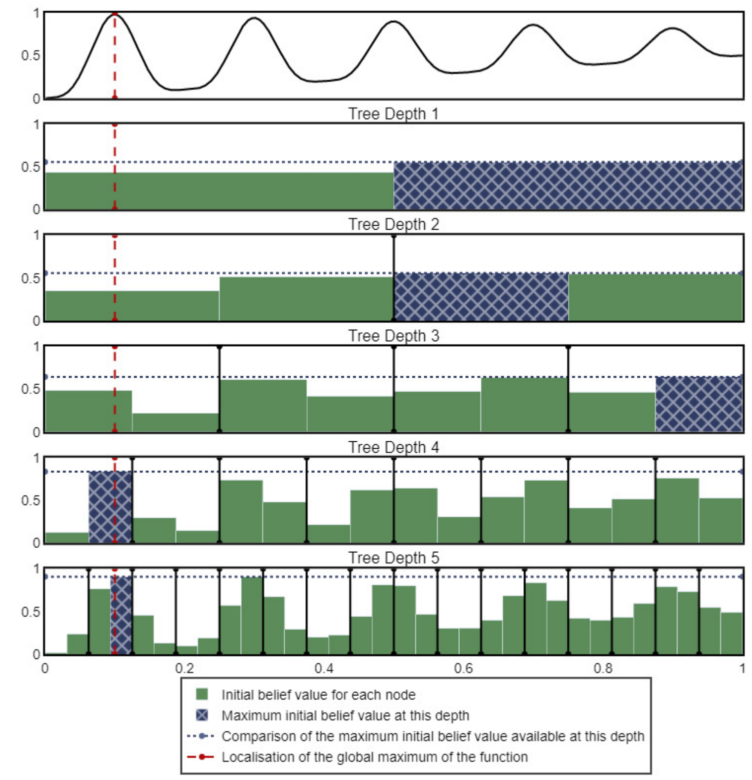

Figure 4.8: Initial belief values of the nodes in the game tree of f_4, plotted at the top. Each bar in the graph represents the initial belief value for each node within the tree. Sibling nodes are enclosed in the same black box. The horizontal dotted line represents the maximum initial belief value at each depth, facilitating comparison with other bars. The vertical dashed red line shows f_4's global maximum.

This is achieved by surrounding the global and local optima with narrow regions of high reward, meaning that FOP states will not return high rewards on those regions until the state is small enough, which happens at tree depth 4. Hence, reaching a certain depth is necessary to observe a substantial increase in rewards near the local or global optima.

We can summarise the difficulty of finding the f_4's global optimum in three factors. First, the global optimum is located within a narrow region of the domain. Second, the global optimum is located within an initially less rewarding region of the domain. Third, the local optima have comparable values to the global optimum.

f_5, also proposed in this work, is shown at the bottom of Figure 4.3 as f_5 and is plotted again at the top of Figure 4.9 for convenience, is also a deceptive function containing multiple local optima. The highly rewarding region around the global optimum is narrower than in f_4 and is expected to be harder to find. Equation 4.6 defines this function.

$$f_5(x) = 0.5x + (-0.7x + 1)\sin(5\pi x)^{80} \tag{4.6}$$

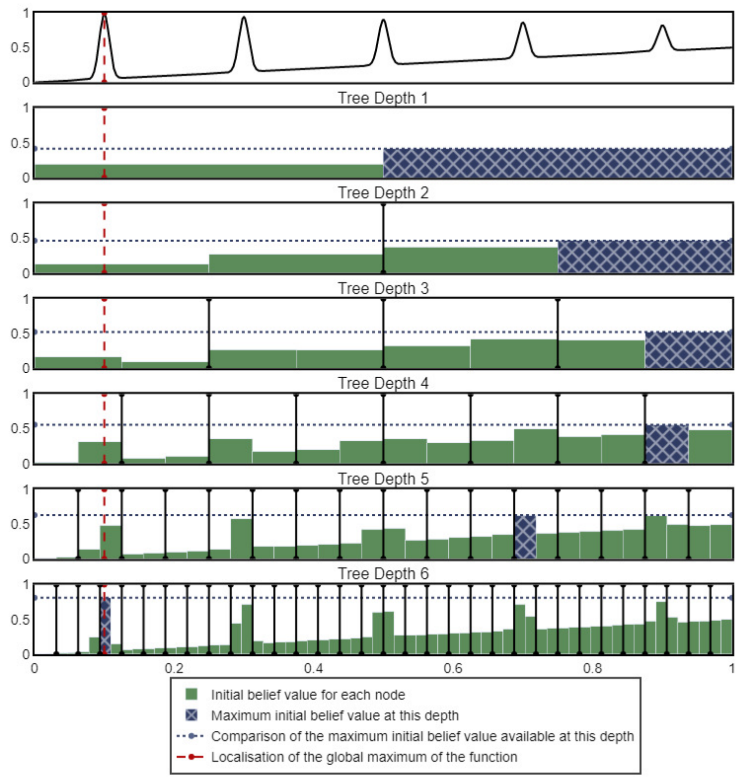

Figure 4.9: Initial belief values of the nodes in the game tree of f_5, plotted at the top. Each bar in the graph represents the initial belief value for each node within the tree. Sibling nodes are enclosed in the same black box. The horizontal dotted line represents the maximum initial belief value at each depth, facilitating comparison with other bars. The vertical dashed red line shows f_5's global maximum.

f_5 exhibits similar characteristics to f_4. The former function, however, features thinner regions of high rewards around both local and global optima. This results in the best rewards being concealed deeper within the tree and making it more challenging for search algorithms to find them. As depicted in Figure 4.9, the largest initial belief values are concentrated in the right-hand side region in tree depths 1 through 5. The maximum initial belief value is found near the global optimum only until tree depth 6. It is expected that the significant disparity in rewards between the left-hand and right-hand sides will deter exploration around the global optimum for a relatively prolonged search period.

4.5 Test problem: The Game of Carcassonne

Carcassonne is a board game designed by Klaus-Jürgen Wrede, released in 2000 and winner of the "Spiel des Jahres" (game of the year) award in 2001. The game takes its name from the town of Carcassonne in southern France, known for its fortified city walls.

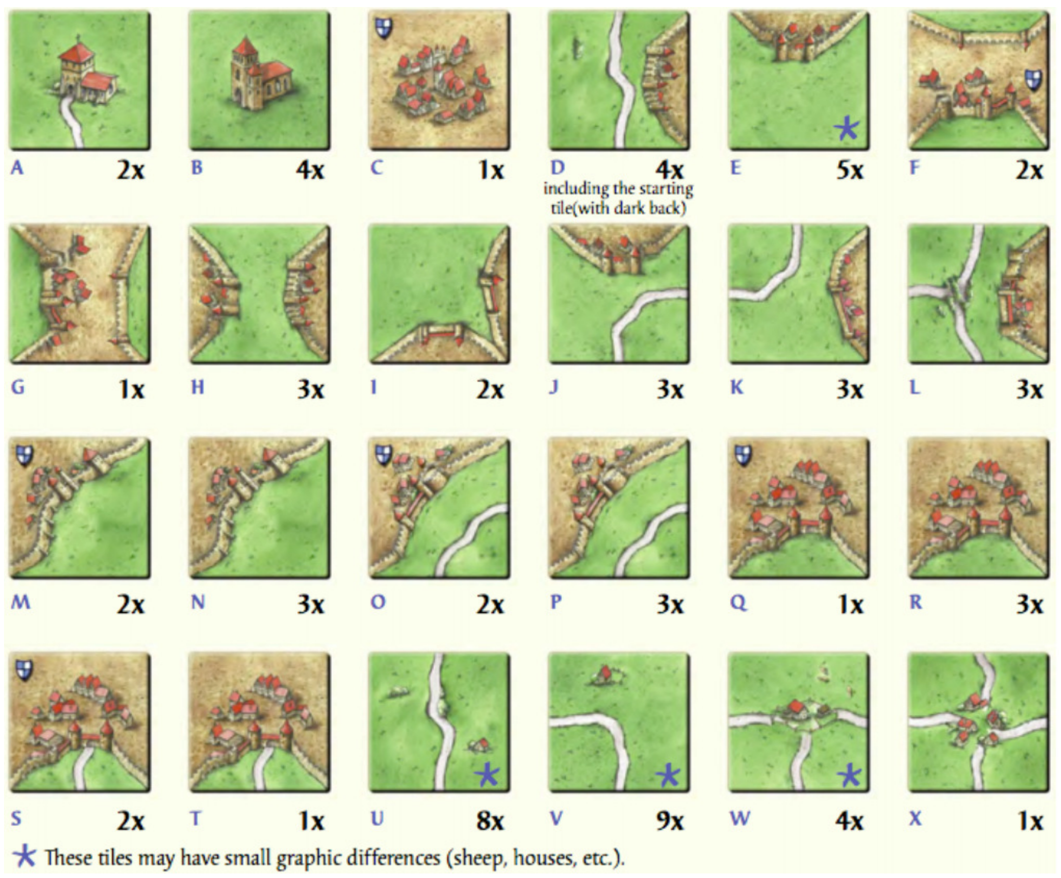
A 2x
B 4x
C 1x
D 4x
including the starting tile(with dark back)
E 5x
F 2x
G 1x
H 3x
I 2x
J 3x
K 3x
L 3x
M 2x
N 3x
O 2x
P 3x
Q 1x
R 3x
S 2x
T 1x
U 8x
V 9x
W 4x
X 1x
★ These tiles may have small graphic differences (sheep, houses, etc.).

Figure 4.10: Carcassonne base game tiles with their duplicate count. Taken directly from the original rulebook.

Of particular interest for our research is the base version of Carcassonne. This version is used for the Carcassonne World Championship, which has been held annually since 2006. Although the game has simple rules, it involves complex player interactions and deep strategic opportunities, making it a challenging and interesting subject for Artificial Intelligence (AI) research.

4.5.1 Carcassonne base game description

In Carcassonne, players take alternate turns trying to score more points than their opponent. To score points, players use figurines called "meeples" (short for "my people"). The game is played with a stack of 71 tiles, composed of tiles with 24 unique configurations, each with a different duplicate count, as shown in Figure 4.10.

The game begins with the "starting tile" (tile D in Figure 4.10) on the board, separate from the main stack. The remaining 71 tiles are shuffled and placed in a face-down stack to be randomly drawn throughout the game. Players then take alternate turns adding

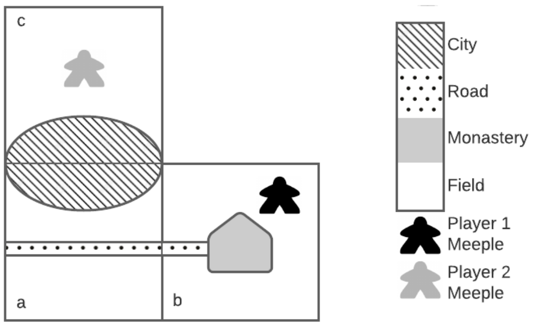

Figure 4.11: Sample state of the game of Carcassonne after 2 turns of play. The starting tile (a) is the same for each game and is on the board when the game begins. Player 1 played tile b and placed a meeple on the field. Player 2 played tile c with a meeple also on the field. The city in tile a was "completed" when tile c was played, while the road is still incomplete.

tiles next to the ones on the board with the only limitation being that the edges of the tiles must match the edges of the tiles they are placed next to. In very unusual cases, it is possible that a tile cannot be legally placed on the board. In this case, the tile is discarded out of the game and a new tile is drawn from the stack.

As the game progresses, the board expands as players add tiles to it. The game concludes when the tiles are exhausted. An example of a Carcassonne board is shown in Figure 4.11.

Each player's turn consists of four phases:

- **Drawing a tile:** A tile is drawn from the stack of tiles at random. If there are no tiles in the stack, the game is over.
- **Placing a tile:** The player chooses any valid spot on the board to play the drawn tile. If it is not possible to legally place the tile, it is discarded and a new tile is drawn. Tile placement follows two rules: (i) the tile must be played in contact with at least one tile that is already on the board, and (ii) all the features on the edges of the played tile that are in contact with other tiles must match the features on the edges of those tiles.
- **Placing a meeple:** The player can choose to place a meeple on a feature of the played tile. A meeple cannot be placed in a feature that, once connected with the board, already has any meeples on it.
- **Scoring a feature:** All the features with meeples that were completed with the tile placement are scored and those meeples are returned to their owners.

The base game of Carcassonne has four features: roads, cities, monasteries, and fields, each one with its unique scoring and completion rules, described in Table 4.3.

In Carcassonne, points can only be scored using meeples, which are limited to 7 per

Table 4.3: Carcassonne features completion and scoring rules.

Feature	**Completion condition**	**On completion**	**End of the game**
Road	No more tiles can be added to the road	Return meeples. 1 point per tile	1 point per tile
City	No more tiles can be added to the city	Return meeples. 2 points per tile + 2 points per shield	1 point per tile + 1 point per shield
Cloister	Cloister's sides and diagonals have tiles	Return meeple. 9 points	1 point + 1 point per surrounding tile
Field	Never completed	N/A	3 points per adjacent completed city

player. If a meeple is played, it remains on the board and cannot be used again until it is released. When a feature is completed, the meeples in that feature are released and the player or players with the majority of meeples in that feature receive its points. When a meeple is released, it is returned to its owner to be used again.

Throughout the game, players diligently monitor their scores. Upon reaching the game's conclusion, all unfinished features that are occupied by meeples are scored following the criteria outlined in the final column of Table 4.3. The player who has the highest total score is proclaimed the victor. To predict the potential outcome of the game at any given non-terminal turn, one can compute its final score as though the game had concluded at that specific moment. This predictive heuristic is well-known in competitive Carcassonne and is called the "virtual score".

Carcassonne's game tree

Research in Carcassonne is scarce. In her master thesis, [78] analysed its complexity, concluding that it has a game-state complexity of approximately 10^{40} and a game-tree complexity of approximately 10^{194}, which are comparable to those of Chess and Go as can be seen in Figure 4.12.

Although the analysis by [78] indicates an average branching factor of 55 for Carcassonne, the number of possible resulting states after an action is significantly higher if we factor in the tile drawn by the next player. The 71 tiles from the stack have 24 possible tile designs. Some of those unique 24 tiles run out as the game progresses, therefore reducing the potential outcomes from the random event. Regardless, the average branching factor remains relatively consistent throughout the game given the proportional increase of the board size.

Regarding Carcassonne's random events, as the unique tiles vary in quantity, there is always an uneven probability of drawing each of them throughout the game. Despite its unpredictability, the game remains fully observable. Both players have access to informa-

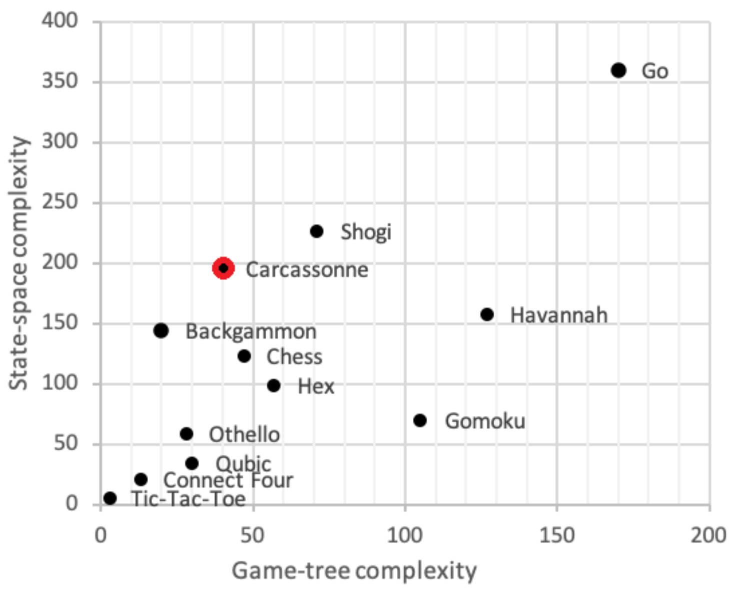

Figure 4.12: State-space complexity and game-tree complexities in base $log(10)$ of games commonly used for research, including Carcassonne.

tion about the remaining tiles in the stack and possess complete knowledge of the game state at all times.

Carcassonne is a game with atomic actions, despite them having four phases as we previously described. The first phase (drawing a tile) corresponds to a random event. The following two phases (placing the tile and placing a meeple) are modelled as single actions where both the tile and the meeple are played simultaneously. This makes all possible actions atomic, with or without a meeple. The final phase of the turn, scoring, is automatic and does not require a decision from the player.

Thus, given its atomic turns and the random event at the beginning of each turn, Carcassonne has a regular *-minimax tree (refer to Chapter 2). Moreover, because the game's length depends on the number of tiles only, it has a constant depth, meaning that we do not have to worry about abrupt termination states in the game tree.

4.5.2 Carcassonne fitness landscape analysis

Conducting an analysis based on depth, similar to the one we did for FOP in Section 4.3, is not feasible for Carcassonne due to its considerable branching factor and the indeterminacy of optimal moves. Unlike in FOP, where we possess clear insights into the optima's location and can thus identify the best action, Carcassonne's intricate fitness landscape demands a different methodological approach.

The gameplay mechanics of Carcassonne incorporate a resource management dimension, given that players are restricted by the finite number of meeples they can play throughout the game. This limitation injects the scoring system in Carcassonne with a potentially deceptive quality. The following observations are made to support this claim:

(1) Players have a limited number of meeples, fewer than the total number of turns, making it impossible to play a meeple on every turn. Therefore, employing a strategy that uses meeples indiscriminately will likely lead to a shortage, preventing the player from capitalising on scoring opportunities later in the game.

(2) The completion of features is uncertain due to the randomness of tile draws. It may require several turns to complete a feature, and opponents have the potential to block it. Consequently, the reclamation of meeples from completed features demands strategic foresight and is not always assured.

(3) Situations may arise during the game where a player can claim, complete, and score an unclaimed feature, all while retrieving the meeple used within the same turn. These moments increase the strategic advantage of maintaining meeple availability.

In other words, adopting greedy strategies to maximise score gains on every turn does not necessarily lead to optimal outcomes, as such approaches may lead to meeples shortages and eventually lower scores. Greedy strategies are typically employed by inexperienced players who soon recognise them as suboptimal. This tendency also presents challenges for algorithms based on the minimax principle, as they struggle to forecast strategies that account for future meeple usage. This suggests that Carcassonne has a deceptive fitness landscape with respect to its scoring system.

Scores in Carcassonne are a state evaluation function embedded within the game and available to the players at all times. The virtual score (discussed in section 4.5.1) is even more precise and is easy to calculate with the information available to the players. The difference of final scores was proven superior by [78] when used to calculate the rewards in MCTS upon the conclusion of a game, instead of the classic win-loss-draw type of reward, and is the one adopted in this work.

Previous works that involve the game of Carcassonne and MCTS use the difference of final scores as is [78] or divided by a large arbitrary number [19]. We remind the reader that the original C proposed for UCB1 is $\sqrt{2}$ [10], meant for rewards in the range [0,1] [99] (or occasionally [-1,1] for adversarial games) as it satisfies the Hoeffding inequality [79, 19]. In other words, very large rewards require equally large C values to be properly explored. Similarly, dividing the scores by large numbers makes the rewards comparably small, requiring smaller C values to strike a healthy balance. Thus, it is important to define an upper bound for Carcassonne scores to find an adequate and standardised way to normalise the score.

Carcassonne maximum theoretical score

To the best of our knowledge, there is no formally estimated maximum score achievable with the tiles of the base game of Carcassonne (71 tiles plus the starting tile), besides empiric calculations. We now define a maximum theoretical bound for the score in Carcassonne that ignores the limitations of the number of players or the game version and is easy to implement. In this way, our score bound is generalisable across different variants of the game and for any amount of players with minimum overhead, enabling its indistinct usage for tree search algorithms in Carcassonne. Our calculation of the maximum score is shown in Equation 4.7.

$$\text{Max Score} = 2 * p_c + p_r + p_m + 3 * n_c * m \tag{4.7}$$

where m is the number of initial meeples. p_r and p_m are the sum of the scorable potential of the roads and cloisters in all the tiles, respectively. Similarly, p_c is the sum of the potential value of every city in every tile. To calculate n_c, which is the maximum count of completed cities achievable with the available tiles, we first introduce the concept of a **city opening**. A city opening refers to a space on the board adjacent to an existing city, requiring a city segment on the newly placed tile to match it. As an example, the initial state of the game shown in Figure 4.11 has a city with one city opening. This also means that any tile placed adjacent to the top of the initial tile needs to have a city in it. Tiles that contribute to city completion are tiles that never increase the number of city openings of cities already on the board when played on a city opening. With this knowledge, we now define the tiles with city configurations that can contribute to city completion, shown as A and B in Figure 4.13.

City configurations A and B in Figure 4.13 are the only ones that contribute to city completion in the base game of Carcassonne. A and B include any tiles with that city configuration regardless of the non-city features on the other sides of the tile. A and B can be found more than once in a single tile if the cities are not connected in between them. We can prove that any other tiles with cities, e.g. tiles with a single connected city on three of its sides (see, for example, tile Q in Figure 4.10) cannot complete cities by themselves by arguing that there is no possible arrangement of such tiles on the board, where placing another of them decreases the number of city openings. In other words, if a tile has the same city on three or more of its sides, that tile can never decrease other cities' opening count, hence not contributing to their completion.

Figure 4.13 defines the three basic city completion configurations: AA, AB, and BB. These configurations represent the smallest possible completed cities and are the only ones considered in the score calculation. Thus, we can calculate the maximum completed cities n_c in Equation 4.8 as follows:

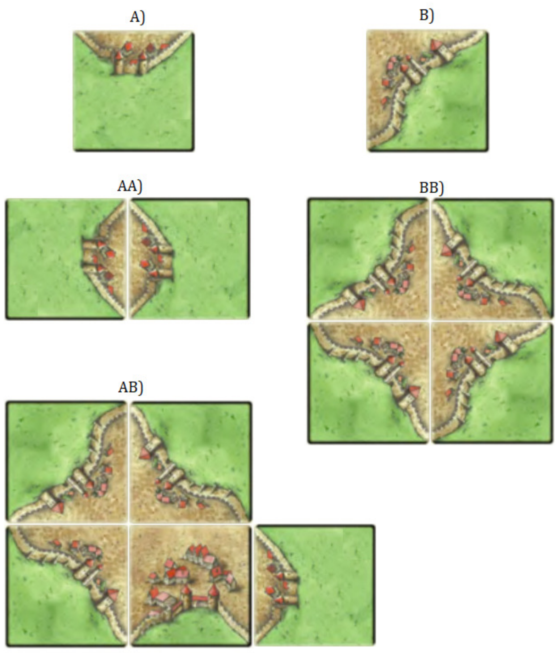

Figure 4.13: Carcassonne city completion tiles (A, B) and their respective basic completion configurations (AA, AB, BB).

$$n_c = \text{count}(AA) + \text{count}(BB) + \text{count}(AB) \tag{4.8}$$

where $\text{count}(AA) = \frac{\text{count}(A)}{2}$, $\text{count}(BB) = \frac{\text{count}(B)}{4}$ and $\text{count}(AB)$ is typically zero. It equals 1 only in the specific scenario where there is exactly 1 leftover A city configuration and 3 leftover B city configurations remaining. The rest of the city tiles in the stack are assumed to fit in between the completed city configurations from Figure 4.13 for simplicity. For this reason, the maximum score calculation counts every city in the stack as if they were part of completed cities by multiplying their values by 2, to match the rules described in Table 4.3. The assumptions done to calculate our maximum score in Carcassonne are listed below:

- All scored points are attained by the same player.
- There are meeples available to score all the features throughout the game.
- There are enough turns to place and collect all the meeples required to score throughout the game.
- There are as many completed cities as possible.
- There are at least as many farms as meeples.
- Every city is completed, and all cities on the tiles are part of those completed cities.
- Every road is scored, and all cities on the tiles are part of those completed cities.
- Every cloister is scored with the maximum score possible.

- Every farm is in contact with all the completed cities.

It should be noted that some of these assumptions are not always realistic in actual gameplay. For example, it is not always possible to have every city in contact with every farm. The listed assumptions are aimed at simplicity, as most of their limitations are far from trivial and may greatly vary from one version of the game to another. Using our calculation, we find that the base game of Carcassonne has a maximum score of 593 points, with the following breakdown:

- 120 points from cities.
- 62 points from roads.
- 54 points from cloisters.
- 357 points from farms (7 farms, each with 17 completed cities).

The aforementioned score calculation method has been implemented in the Carcassonne game simulation environment used in this research to automatically normalise MCTS rewards based on scores in any variant of the game.

Monte Carlo simulations in Carcassonne

Monte Carlo methods use random sampling to compute numerical results. Specifically, MCTS applies this approach to approximate heuristic evaluations of states using a random uniform default policy. During the simulation phase, MCTS employs this policy to accumulate rewards by simulating complete games through random move selection. Although this policy is considered mostly domain-independent, its effectiveness can greatly vary. For example, in Chess, randomly simulated games often result in draws, leading to inconsistent reward retrieval. Additionally, research indicates that biases can still be present in random simulations. For instance, players with more move alternatives might also be more prone to blunders compared to players with fewer options. To back this claim, we present Figure 4.14, where a Chess position with a decisive material advantage for white is shown. However, Monte Carlo simulations are more likely to return a favourable evaluation for black. Further analysis by [7] indicates that this is due to the random policy being more likely to blunder with white, as the white queen has multiple actions from where it can be easily captured. Furthermore, the move $Qd7+$ forces black to capture the white queen, making that move an unavoidable loss of material for white.

For this reason, we now analyse how insightful Monte Carlo simulations are in the game of Carcassonne for MCTS. To do so, we determine if they are informative, prone to biases and how detrimental those biases are to the search. We also use them to observe how the simulation rewards correlate to the game features among different turns.

A major difference between Carcassonne and Chess is Carcassonne's scoring system. In Carcassonne, draws are far less common than in Chess and the scoring system provides

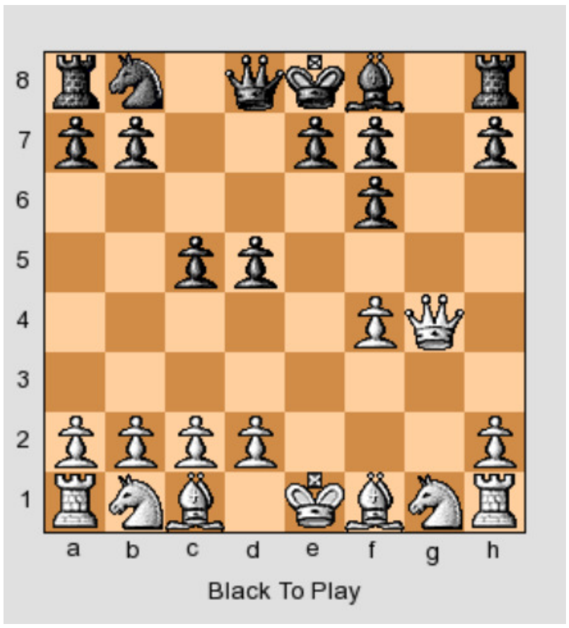

Figure 4.14: Chess position where Monte Carlo simulations indicate that black is slightly better, although white's material advantage should be decisive. Taken from [7].

additional information about the game state.

The reward used in the MCTS variants presented in this work for the game of Carcassonne is the normalised difference of the final scores at the end of the game, from the perspective of the player using the algorithm. In the case of single-player Carcassonne variants, the reward is the normalised final score. Throughout this section, we present the scores used as rewards before normalisation, for discourse simplicity.

To measure how deceptive rewards are for tree search algorithms in Carcassonne when using Monte Carlo simulations as the default policies, we compared the scores for every type of action, grouped by meeple usage. We first focused on the actions available to Player 1 on Turn 1 to visualise how the initial belief value varies for every type of feature claimed with a meeple. To do so, we ran $1,000$ Monte Carlo simulations from each state possible after Turn 1. For their initial action, Player 1 could potentially draw any one of the 24 distinct tile configurations available in the game. Each of these configurations allows for an average of approximately 20.95 ± 13.15 alternative actions, leading to a total of 482 distinct states after Turn 1. The results of the total $482,000$ random games are shown in Figure 4.15.

Figure 4.15 shows that using a meeple in a cloister gives the largest positive difference of final scores overall, as well as the largest average score for Player 1. The rewards of the cities are the second best and are better than those of the roads. Interestingly, actions

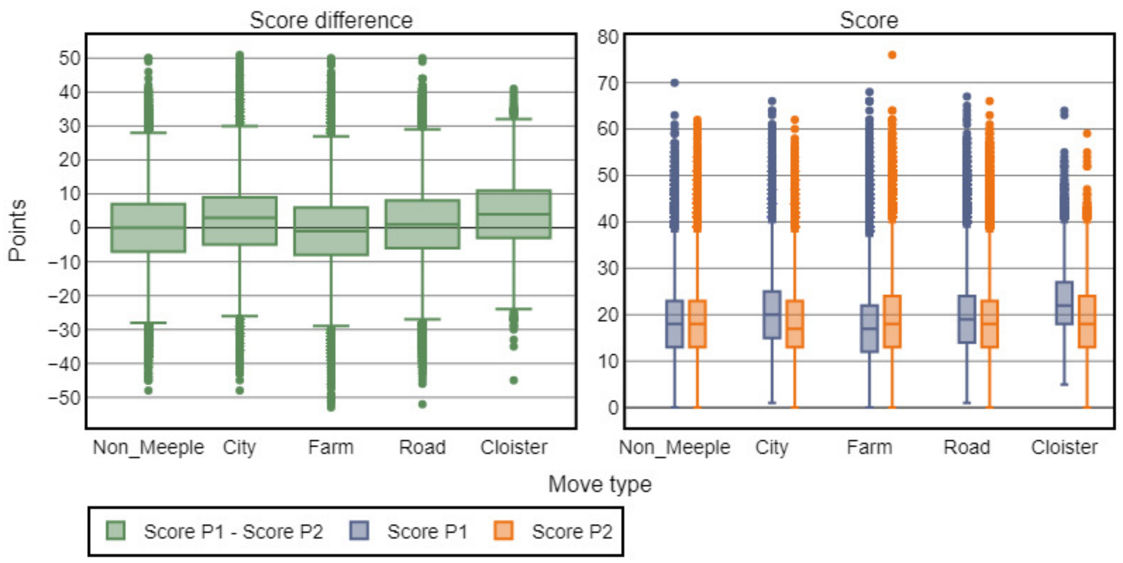

Figure 4.15: Scores of the base game of Carcassonne averaged from 1, 000 Monte Carlo simulations from every possible state available after Turn 1, grouped by how the meeple was used on Turn 1.

that play a meeple on a farm, referred to as **farm-actions** from now on, are the least rewarding type of action on average, giving negative rewards. Furthermore, farm-actions have worse rewards than not using a meeple at all. It is worth noting that farms are the most commonly available feature in Carcassonne, present in 23 out of the 24 unique tile configurations (refer to Figure 4.10) and accounting for 98.59% of the tiles in the base game tile stack. Moreover, most of the tiles have multiple farms. For instance, tile X in Figure 4.10 has four unique farms, isolated by the roads.

The low rewards of the farm-actions shown in Figure 4.15 are a consequence of random play being unlikely to complete cities, implying a bias in random play. To demonstrate how completed cities on the board influences the rewards of the farm-actions, we ran the same experiment for all the states possible after Turn 2, taking into consideration two contrasting states for Turn 1 shown in Figure 4.16.

In the board state $B1$, depicted in Figure 4.16, Player 1 has already completed a city on Turn 1. In the board state $B2$, Player 1 placed a tile that makes the city require more tiles to be completed, from an original minimum of 1 to a minimum of 3. For each board in Figure 4.16, we ran 1, 000 Monte Carlo simulations from each state available after every action possible from each of the boards in Fugure 4.16. In the case of board state $B1$, Player 2 can potentially draw any one of the remaining 23 tile configurations (not 24 configurations, because one specific tile configuration is unplayable in the current board context) each with an average of 28 ± 16.49 alternative actions, resulting in 644 total possible game states after Turn 2. For board state $B2$, only 23 tile configurations

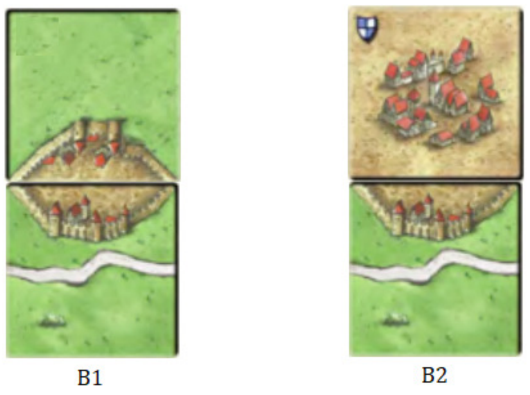

Figure 4.16: Two different Carcassonne game states after Turn 1. In $B1$, Player 1 completed a city in its turn. In $B2$, Player 1 placed a tile that makes the city require additional tiles to be completed.

remain available for Player 2, as the tile used in Turn 1 had only one copy in the tile stack (Tile C in Figure 4.10). Each of those tile configurations has an average of 31.91 ± 17.23 alternative actions, resulting in 734 total possible game states after Turn 2. The results of this experiment are shown in Figure 4.17.

Figure 4.17 shows the difference of final scores from the perspective of the player choosing the action, which in this case is Player 2. The rewards of the farm-actions are influenced by the presence of a city already completed, illustrated by the difference in farm rewards with and without a completed city (boards $B1$ and $B2$ from Figure 4.16, respectively). Interestingly, given a board with a single completed city ($B1$), farm-actions are already as rewarding as any action with no meeples. Thus, 1 completed city is enough to make farm-actions more appealing to players based on Monte Carlo simulations than actions that do not use a meeple. Note that in the base game of Carcassonne, 22 tiles out of the initial 71 (that is, a likelihood of 31% of the initial tile stack) can complete a city on Turn 1, which is the most rewarding meeple usage alternative for Player 1 according to Figure 4.15. Additionally, claiming and completing cities is arguably almost always the best action for any player whenever possible, ensuring an increasing number of completed cities as the game progresses. This indicates that farm-actions are expected to be increasingly more rewarding as the game progresses when played between agents with some degree of optimal play.

Among all the moves available to a player in Carcassonne, farm-actions have the peculiarity of losing the played meeple permanently. As discussed before, meeples are a limited resource in the game of Carcassonne and when used in farm-actions they remain on the board for the rest of the game, thus they are not available for reuse. While farm-actions may be optimal in certain situations, they tend to be suboptimal in the early

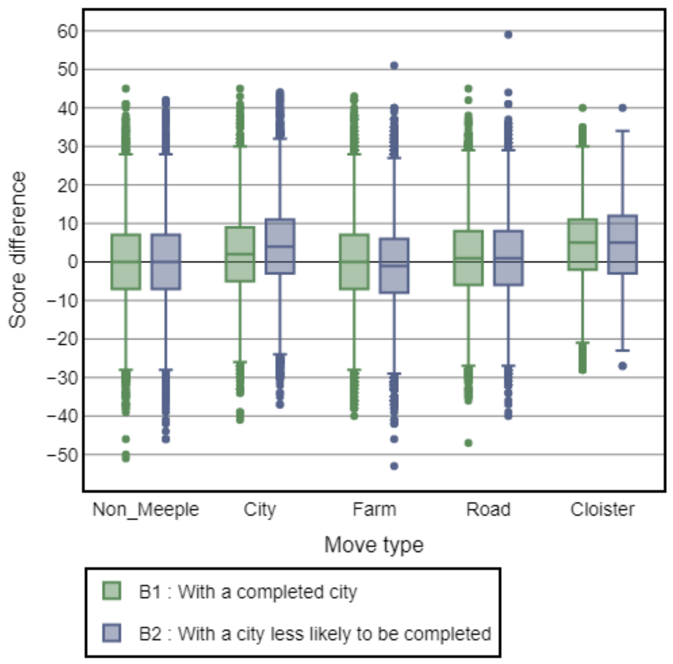

Figure 4.17: Difference of final scores (Score P2 - Score P1) in the base game of Carcassonne, averaged from 1,000 Monte Carlo simulations from every possible state available after Turn 2, grouped by meeple usage in Turn 2 and previous board state $B1$ or $B2$ introduced in Figure 4.16.

stages of the game, thus becoming a source of deception if the Monte Carlo simulations favour them early. To demonstrate that fact, Figure 4.18 shows the average difference of final scores from Player 1 perspective, grouped by when and how the meeples were used, with data from 482,000 Monte Carlo simulations (1,000 from each of the 482 possible states after Turn 1).

Figure 4.18 differentiates only between farms and the rest of the features, as farms are the only feature with the particularity of losing the played meeple permanently, whereas the rest of the features share completion and meeple release conditions, described in Table 4.3.

From the plot on the left-hand side of Figure 4.18, we observe that the difference of final scores decreases as more meeples are invested in farms, irrespective of the turn. This trend is visually represented by progressively darker shades moving towards the right-hand side of the plot. Additionally, this plot highlights that playing meeples in farms at the early stages of the game is less optimal, as evidenced by the darker shades in the plot's bottom, particularly on its left-hand side.

The middle plot in Figure 4.18 shows the number of played meeples on non-farm features. It reveals that the count of meeples on the x-axis can exceed the initial allocation

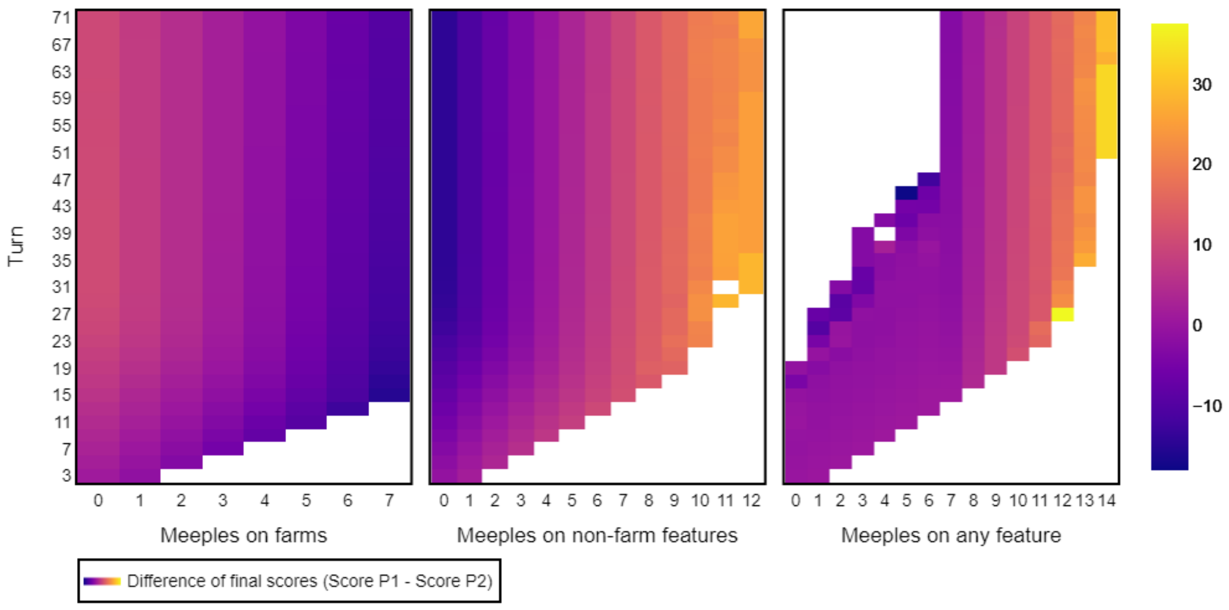

Figure 4.18: Impact of meeple usage on the difference of final scores (Score P1 - Score P2) in the base game of Carcassonne. Data is averaged from 1,000 Monte Carlo simulations from every possible state after Turn 1. Each plot point aggregates the mean difference of final scores for game instances where Player 1's total played meeples match the x-axis value by the game turn specified on the y-axis. The same dataset is segmented differently by meeple usage in each plot: on farms in the left plot, on non-farm features in the middle plot, and overall usage in the right plot.

of 7, indicating that some meeples were recycled. Importantly, when 8 or more meeples are played, there is a noticeable increase in the difference of final scores. This pattern suggests that recycling meeples by completing features enhances overall performance. Moreover, the middle plot demonstrates that the difference of final scores grows with the increased use of meeples in non-farm features, contrasting with the findings for farms depicted in the left plot. This further indicates the advantage of quickly playing meeples to claim non-farm features, underlining the distinct scoring dynamics between farm and other game features. It is crucial to note that Monte Carlo simulations have a low probability of completing cities from random choices. As a result, farms are shown to be less advantageous in Figure 4.18 under random play conditions, compared to strategic play.

When turning our attention towards right-hand side plot in Figure 4.18, we can observe that the maximum meeples used increased from 12 in the middle plot to 14. This implies that every game with 12+ meeples in the right-hand side plot includes at least 1 meeples on a farm, showcasing that high and positive differences of final scores are achievable with meeples on farms. Furthermore, the games with 12+ played meeples are the most rewarding overall, shown by the larger scores when nearing the right-hand side of the

Table 4.4: Carcassonne variants.

Variant	Players	Initial meeples	Tiles	Nature
$Carc_{base}$	2	7	72	stochastic
$Carc_{1,s}$	1	1	24	stochastic
$Carc_{1,d}$	1	1	24	deterministic
$Carc_{3,s}$	1	3	24	stochastic
$Carc_{3,d}$	1	3	24	deterministic

plot.

As an additional note, from the 488, 000 games used to generate Figure 4.18, there was no single instance of a game where Player 1 did not play any meeple by Turn 21, indicated by the empty spaces at the top of the right-hand side plot. This illustrates that Monte Carlo simulation have a bias towards using meeples.

4.5.3 Carcassonne proposed variants

We use Carcassonne as a benchmark to compare and analyse the behaviour of the MCTS agents. To do so, we propose simplified versions of the game. We propose to simplify the base game of Carcassonne by altering the following aspects of the game:

- Making the game deterministic.
- Removing tiles from the game.
- Decreasing the number of players.
- Decreasing the number of initial meeples.

We include a deterministic version to reduce the branching factor of the game, to allow our agents to explore the game tree more deeply. The *search horizon* of a tree search algorithm is the limit of the tree that it evaluates. Vanilla MCTS's search grows asymmetrically but is also width-first, as it prioritises unexplored nodes in the selected branch before going any deeper. This characteristic is very effective in general, as actions that are closer to the root are more likely to be relevant for decision-making. It also implies that games with large branching factors limit the depth MCTS's can search. Carcassonne has a relatively large branching factor, becoming a game where MCTS struggles to evaluate the game tree deeply, especially because Carcassonne incorporates long-term plans and strategies that are rewarding and require a deep search to be identified.

When proposing Carcassonne variants, it needs to be taken into consideration that there should be fewer meeples than turns to play for each player, to keep the resource management aspect of the game. We want to decrease the complexity, to make easier conclusions about both the game and the agents that play it. Table 4.4 summarises our proposed Carcassonne variants with their assigned names.

Besides the base game for two players, we introduce a single-player version of the game with a decreased number of tiles. Our proposed single-player variants follow the original rules of Carcassonne, but are limited to one player. We also offer two different sets of initial meeple counts. Further description for each variant is given next.

Single-player Carcassonne with 3 initial meeples

In the single-player variant of Carcassonne, the player sequentially takes turns until all tiles have been used, aiming to achieve the highest possible score. Beyond the adjustment to the number of players, we suggest varying the initial number of meeples to accentuate distinct aspects of the game. Furthermore, we investigate the potential for playing the game in a deterministic manner.

To make the game deterministic, we let the order of the tiles be randomised at first but remain constant and known to the player throughout the game, removing the recurrent stochastic nature of drawing a random tile. Regarding the tiles, we use one single tile of each unique configuration to keep the properties of the game. This means that the game will have 24 tiles in total, one for each unique tile configuration in Figure 4.10. The single-player variants with 3 initial meeples are referred to as $Carc_{3,s}$ and $Carc_{3,d}$, for stochastic and deterministic, respectively.

We performed an analysis of the initial belief values of each action for every stochastic single-player version of the game. We ran $1,000$ Monte Carlo simulations from each state and available after every possible state after Turn 1. The results are presented in Figure 4.19.

According to Figure 4.19, actions that claim a cloister on the first turn are the most valuable ones. The rewards of the cities are the second best and are better than those of the roads. Farm-actions are the least rewarding type of action on average, even worse than not playing a meeple at all. The relative values of every feature from Figure 4.19 for $Carc_{3,s}$ and $Carc_{3,d}$ are consistent with those from Figure 4.15 for the base game of Carcassonne. This indicates that the single-player Carcassonne variants with 3 initial meeples effectively simplify the game without altering its essence. However, the maximum achievable score is significantly reduced, despite removing the interaction between players. Based on our calculations from Section 4.5.2, the maximum theoretical score for our single-player Carcassonne variants with three meeples is 135 points, detailed as follows:

- 52 points from cities.
- 20 points from roads.
- 18 points from cloisters.
- 45 points from farms (3 farms, each with 5 completed cities).

To continue comparing the proposed variants, we show the average score at the end

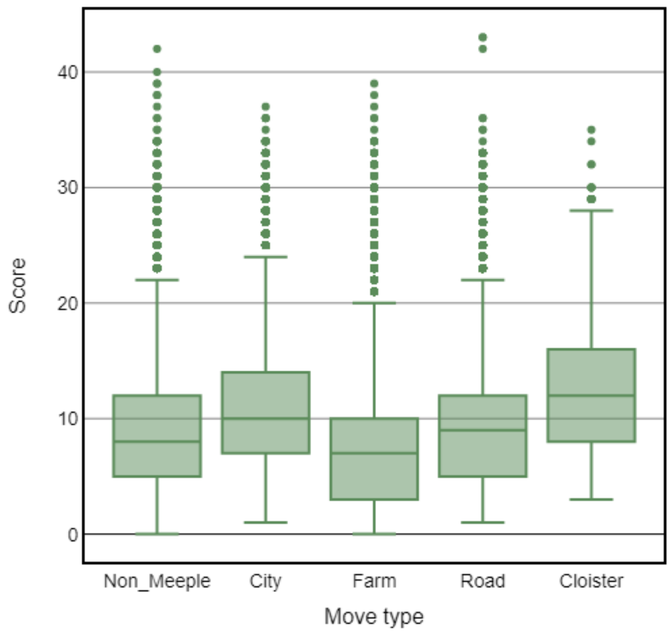

Figure 4.19: Final scores of the single-player Carcassonne variant with 3 initial meeples, averaged from $1,000$ Monte Carlo simulations from every possible state available after Turn 1, grouped by how the meeple was used on Turn 1.

of $Carc_{3,s}$ and $Carc_{3,d}$ according to when and where the meeples are used, as we did for the base game of Carcassonne. The results are presented in Figure 4.20.

.

The left-most plot in Figure 4.20 is equivalent to the left-most plot in Figure 4.18, showing that allocating more meeples to farms and doing so earlier in the game, negatively impacts final scores under random play, as evidenced by the darker shades on the bottom and right-hand side of the plot. The middle plot in Figure 4.20 indicates that the number of meeples played on non-farm features can exceed the initial allocation of 3, going up to 8. It also demonstrates that final scores increase with the greater use of meeples on non-farm features, as the plot progressively brightens towards the right-hand side.

The middle plot in Figure 4.20 has an evident low final score data point for 6 total played meeples by Turn 7, depicted by the darker isolated box at the bottom-right, which contrasts with the final scores of the rest of the games for the same column. This is an isolated unlikely game where 6 meeples were played in non-farm features within the first 7 turns and led to a relatively low final score with random play.

The right-most plot in Figure 4.20 also shows a maximum of 8 played meeples, consistent with the middle plot. The major difference between the middle and right-most plots is the increased contrast in final scores for games with 7 and 8 total played meeples,

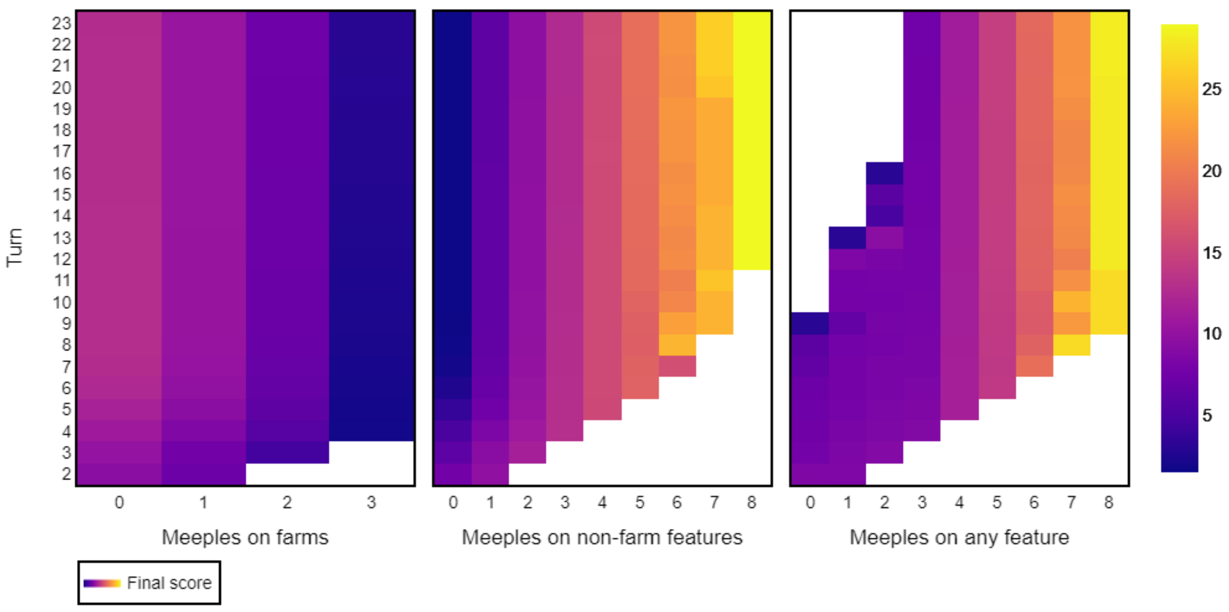

Figure 4.20: Impact of meeple usage on the final scores in the single-player Carcassonne variant with 3 initial meeples. Data is averaged from $1,000$ Monte Carlo simulations from every possible state after Turn 1. Each plot point averages the final scores for game instances where Player 1's total played meeples match the x-axis value by the game turn specified on the y-axis. The same dataset is segmented differently by meeple usage in each plot: on farms in the left plot, on non-farm features in the middle plot, and overall usage in the right plot.

more pronounced in the right-most plot. This suggests that including farm meeples in the segmentation criteria for the right-most plot reveals more games with lower final scores in the column for 7 total played meeples, indicating that farms may be less rewarding in the $Carc_{3,s}$ and $Carc_{3,d}$ variants than in the base game of Carcassonne, likely due to the reduced number of cities that can be completed.

Lastly, the right-most plot in Figure 4.20 shows that final scores are highest as more meeples are used across all features, aligning with the observations from the right-most plot in Figure 4.18 for the base game of Carcassonne.

Single-player Carcassonne with 1 initial meeple

We now introduce our single-player Carcassonne variant with 1 initial meeple. This version is intended as a base case for resource management, since a single meeple demands strategic usage to achieve high scores. In a Carcassonne game with 1 meeple for the full game, farm-actions are never the best option unless it is the last turn (or a couple of turns) of the game. A meeple, while available, has the potential to score multiple instances of each of the other features, while a meeple on a farm can only score that farm.

Additionally, due to the high density of tiles with farms, a player has a high likelihood of having the opportunity to claim a farm towards the end of the game, if it is the case that it is the most rewarding feature on the board.

According to our analysis from Section 4.5.2, the maximum number of cities that can be completed in our single-player Carcassonne variants equals 5 with the 24 tiles stack. That means the maximum value a farm can have is $5 \times 3 = 15$ points in the game variants $Carc_{1,s}$ and $Carc_{1,d}$. So, for Turn 1, a meeple on a farm will score a maximum of 15 points, as the meeple cannot be collected to be used again. On the other hand, the meeple can be recycled multiple times by claiming and completing other features, such as cities and roads, easily surpassing those 15 points of the farm. Thus, we can declare that farm-actions are never the best option on Turn 1 for our Carcassonne variants with 1 initial meeple.

Using our calculation from Section 4.5.2, we find that our single-player Carcassonne variants with 1 initial meeple have a maximum score of 105 points, with the following breakdown:

- 52 points from cities.
- 20 points from roads.
- 18 points from cloisters.
- 15 points from farms (1 farm with 5 completed cities).

We ran $1,000$ Monte Carlo simulations from each state available after Turn 1. The results are presented in Figure 4.21.

Figure 4.21 further proves that random play is biased to not completing cities, as the mean of the scores when the first and only meeple is used in a farm is zero. It is worth noting that using the meeple in a city is now the most rewarding first action, in contrast with the other Carcassonne versions. This is because the meeple in the city can be collected and used again with a higher probability than from the cloister, making the city the most rewarding feature to claim on the first turn of play.

Figure 4.22 shows the average final score in the game variants $Carc_{1,s}$ and $Carc_{1,d}$ according to when and how the meeple is used, collected from $1,000$ Monte Carlo simulations from each of the 452 possible states after Turn 1.

From the left-most plot in Figure 4.22, we observe that final scores are generally lower when meeples are used on farms, regardless of the turn. By the end of the game, any other feature proves to be more rewarding than farms, as they allow for the completion and recycling of meeples. The observations from Figures 4.18 and 4.20 are consistent with those in Figure 4.22.

It is noteworthy that in the right-most plot of Figure 4.22, there are diminished final scores for games involving 5 meeples used early, as indicated by the darker shaded group of games at the bottom of the right-most column. This trend is absent in the middle

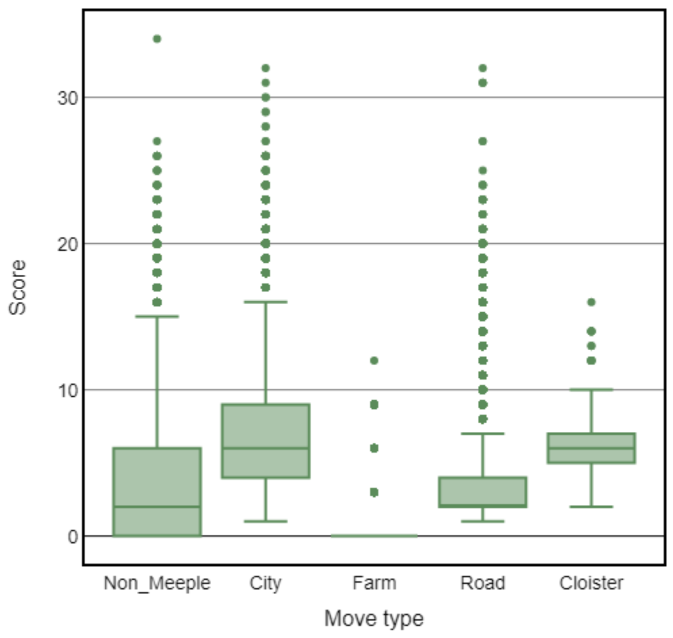

Figure 4.21: Final scores of the single-player Carcassonne variant with 1 initial meeple, averaged from 1,000 Monte Carlo simulations from every possible state available after Turn 1, grouped by how the meeple was used on Turn 1

plot, suggesting it results from scenarios where the meeple was played on a farm as its fifth placement. A similar pattern is observed in the right-most plot of Figure 4.20, where the final scores are lower for games with early meeple usage in the right-most column, but less pronounced. Conversely, Figure 4.18 does not exhibit this pattern, likely due to the ability to complete a larger number of cities with the original set of 71 tiles and the availability of more meeples, which increases the potential value of the farms.

The Carcassonne variants $Carc_{1,s}$, $Carc_{1,d}$, $Carc_{3,s}$, and $Carc_{3,d}$ preserve the game tree characteristics and fitness landscape that are of interest. Furthermore, we observed that Monte Carlo simulations, whether using the difference of final scores in the base game of Carcassonne, or using the final score for single-player variants, are sufficiently informative to direct MCTS algorithms, albeit not without biases. Our findings indicate that random play tends to undervalue farms due to their reduced likelihood of completing cities and exhibits a propensity for rapidly playing meeples, given the greater density of meeple placement options within the set of available actions. This tendency, combined with the misleading nature of score-driven greedy play, establishes Carcassonne as a valuable benchmark for tree search algorithms. The single-player variants not only facilitate the ranking of agents based on final scores but also provide more interpretable insights into their strategic depth and preferred tactics.

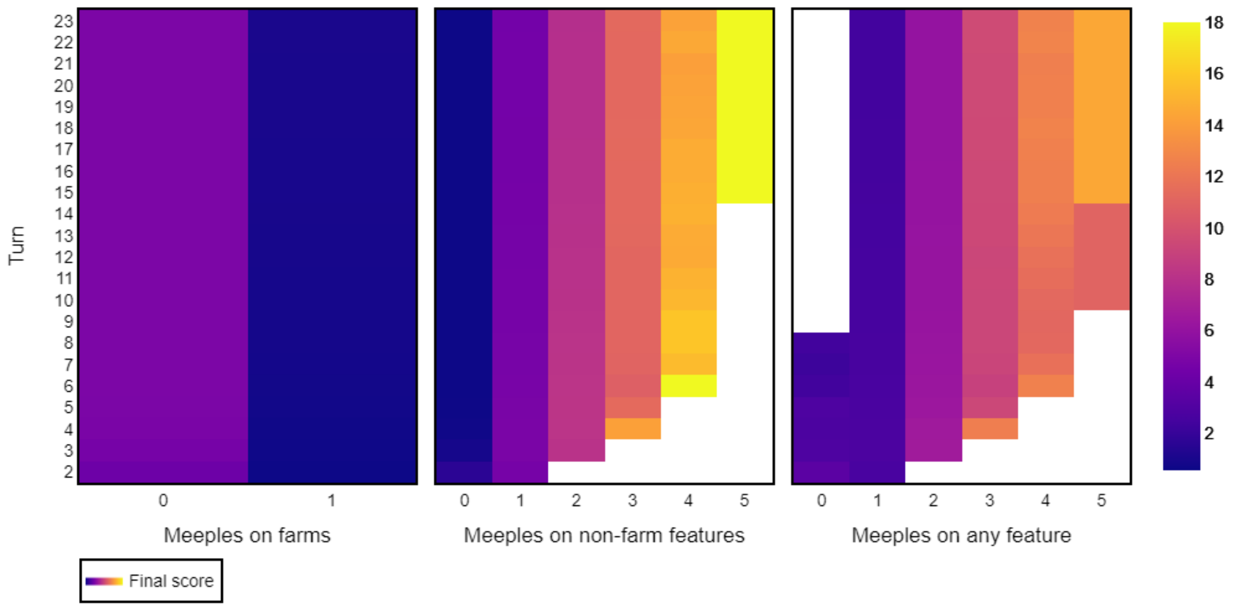

Figure 4.22: Impact of meeple usage on the final scores in the single-player Carcassonne variant with 1 initial meeple. Data is averaged from 1,000 Monte Carlo simulations from every possible state after Turn 1. Each plot point aggregates the mean final scores for game instances where Player 1's total played meeples match the x-axis value by the game turn specified on the y-axis. The same dataset is segmented differently by meeple usage in each plot: on farms in the left plot, on non-farm features in the middle plot, and overall usage in the right plot.

5 Empirical Analysis of Evolving Selection Policies in MCTS

Related publication to this chapter: Fred Valdez Ameneyro and Edgar Galván. "Towards Understanding the Effects of Evolving the MCTS UCT Selection Policy". In: *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2022, pp. 1683–1690.

5.1 Introduction

In this chapter, we introduce Evolutionary Algorithm Monte Carlo Tree Search (EA-MCTS) and Semantically-Inspired Evolutionary Algorithm Monte Carlo Tree Search (SIEA-MCTS), two variants of the Monte Carlo Tree Search (MCTS) algorithm that use Genetic Programming (GP) to evolve their selection policies. EA-MCTS, outlined in Section 5.2, is a novel algorithm aimed at optimising the selection policy of MCTS for navigating the search tree without prior knowledge of the problem domain. SIEA-MCTS builds on the evolutionary process of EA-MCTS by incorporating semantics, a technique shown to improve GP and is elaborated on in Section 5.3. We then present the results of experimental comparisons between EA-MCTS, SIEA-MCTS, and the traditional MCTS algorithm, focusing on the test problems described in Chapter 4. In Section 5.5 we examine the construction process and structure of the statistical trees produced by each MCTS algorithm, analysing their performance across different Function Optimisation Problem (FOP) functions. We also conduct a quantitative comparison using single-player variants of Carcassonne in Section 5.7, to assess the performance of MCTS-based agents versus those using the minimax principle in scenarios requiring complex and long-term strategic planning. The chapter concludes with an analysis of the selection policies evolved by EA-MCTS and SIEA-MCTS in Section 5.8.

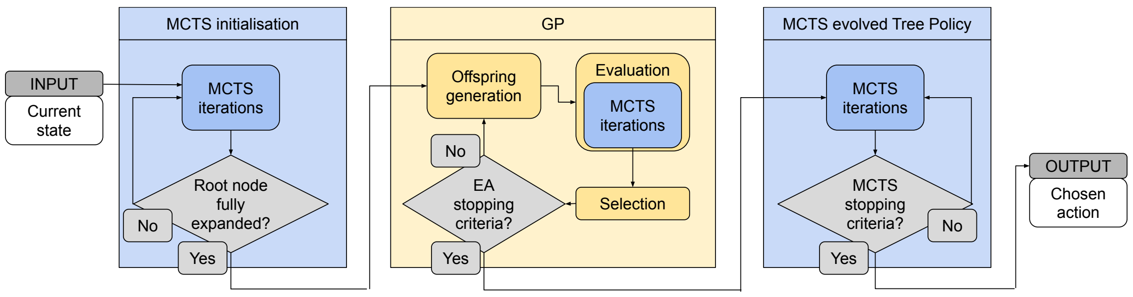

Figure 5.1: EA-MCTS algorithm. The agent receives the current state as an input and outputs the action to be executed.

5.2 Evolving selection policies in MCTS using EAs

To enhance the adaptability of MCTS, we recently introduced the Evolutionary Algorithm Monte Carlo Tree Search (EA-MCTS) [61], inspired by the existence of a wide range of tree policies found in the literature. This MCTS variant seeks to evolve the Upper Confidence Bounds (UCB1) formula within the tree policy dynamically. Other works attempt to evolve the tree policy offline to then use the evolved formula in subsequent decisions [18, 25]. EA-MCTS differs from them as it attempts to evolve the formula from scratch for every single decision throughout a game, including additional steps between iterations but only effectively modifying the selection step of the MCTS algorithm.

EA-MCTS is a first step towards the development of a more general MCTS algorithm that can adapt to different domains and situations. We will now describe the algorithm and discuss its properties. We can separate the functionality of EA-MCTS into the following steps, also illustrated in Figure 5.1.

- **MCTS Initialisation**:The EA-MCTS algorithm starts similarly to the conventional MCTS, described in Chapter 2. It does so by executing the selection, expansion, simulation, and backpropagation phases iteratively. Typically, vanilla MCTS employs the tree policy for node selection from the root until an expandable node is found, meaning the selection policy returns the root node initially and until it can no longer be expanded. UCB1 begins to apply only after the expansion of all root node children, marking the start of GP within the EA-MCTS algorithm.
- **GP Evolution**: From this step, the vanilla MCTS algorithm starts being modified. First, an initial population is formed using the UCB1 formula as the parent. The initial population consists of the parent and λ offspring, generated with subtree mutation. The GP evolves this formula population via a $(\mu+\lambda)$-Evolution Strategy (ES), evaluating each formula's fitness over S fitness iterations within the vanilla MCTS selection policy, with the goal of maximising the average rewards from these iterations. Following the evaluation of all population members, the fittest formula is selected as the subse-

Algorithm 1 EA-MCTS

```
1: Input: Number of gen. $G$, lambda $\lambda$, fitness iterations $S$, statistical tree $T$
2: Output: Evolved tree policy
3: procedure EVOLVING_UCB1_EA
4:     P ← UCB1 formula
5:     for $g \leftarrow 0, \cdots, G$ do
6:         for $i \leftarrow 0, \cdots, \lambda$ do
7:             $O_i$ ← subtree_mutation(P)
8:             a_fitness ← 0
9:             for $s \leftarrow 0, \cdots, S$ do
10:                temp_fit($O_i$) ← select $T(S)$ and rollout($O_i$)
11:                update $T$
12:                a_fitness($O_i$) ← a_fitness($O_i$) + temp_fit($O_i$)
13:            end for
14:            fitness($O_i$) ← a_fitness($O_i$) / $S$
15:        end for
16:        P ← best_individual(O)
17:    end for
18:    return P
19: end procedure
```

quent generation's parent, discarding the rest. This cycle repeats for G generations, culminating in the selection of the optimal formula.

- **MCTS with evolved Tree Policy**: The evolutionary process's resultant formula substitutes UCB1 as the tree policy in all future MCTS iterations, which then proceed as standard using the same updated statistical tree.

An overview of the evolutionary process in EA-MCTS is depicted in Algorithm 1. EA-MCTS is designed to optimise the selection policy on the go while making a single decision. Within EA-MCTS, the MCTS algorithm undergoes modifications through an evolutionary process. This process consists of a GP that executes MCTS iterations, labelled as **fitness iterations**, to assess the fitness of individual solutions. The individuals within the GP are represented as syntax trees modeling the UCB1 formula. In this context, any formula $\hat{f}$ participating in the evolutionary process is termed an *individual* or a *solution*. The GP begins with the UCB1 formula and generates a population of λ offspring. It takes the statistical tree, which has been developed thus far by the MCTS, as input. This statistical tree is then updated based on the outcomes of the fitness iterations until the GP concludes and produces a new formula. Upon completion of the evolutionary process, this newly evolved formula substitutes UCB1 as the tree policy for all subsequent MCTS iterations, which proceed as usual. Notably, EA-MCTS maintains the any-time stopping property inherent to the traditional MCTS algorithm, allowing it to be stopped at any point during the evolutionary process and still yield a valid solution.

In EA-MCTS, the fitness of each individual, $\hat{f}$, is determined with S fitness iterations that use $\hat{f}$ as their tree policy, with the objective of maximising their average reward.

The fitness iterations adhere to the same four phases as the standard vanilla MCTS iterations: selection, expansion, simulation, and backpropagation. However, during each fitness iteration, the selection phase employs different formulae $\hat{f}$ as the tree policy. While fitness iterations primarily serve the purpose of fitness evaluation, they also contribute to the updating of the statistical tree. The precision of this fitness assessment improves with an increased number of fitness iterations. However, allocating more iterations for fitness evaluation implies fewer resources available for the tree search, presenting a trade-off. Moreover, the number of individuals to be generated and evaluated throughout the GP is limited, leading to the need for a small population size.

5.3 Evolving selection policies in MCTS using EAs and semantics

Given the limited population size in the GP of EA-MCTS, promoting diversity is crucial for avoiding stagnation. Specifically, Semantically-Inspired Evolutionary Algorithm Monte Carlo Tree Search (SIEA-MCTS) improves EA-MCTS by making use of **semantics**, a well-known concept in the field of GP [60], with previous success when applied to GP operators like the Semantic Similarity-based Crossover (SCC) [170], used as a similarity measure like the Semantic-based Crowding Distance (SCD) [62] and as an additional Multi-Objective Genetic Programming (MOGP) objective like in the Semantic-based distance as an additional criteriOn (SDO) [63], to mention a few. The core principle of SIEA-MCTS is to use semantic information to discern the most suitable offspring for progression to the next generation when multiple candidates exhibit the same fitness. Algorithm 2 describes the SIEA-MCTS algorithm.

Algorithm 2 differs from Algorithm 1 by introducing the procedure *Sem_Sel*, which uses a distance measure based on semantics, the Sampling Semantic Distance (SSD) [69]. SSD is calculated between the offspring with the best fitness and their parent. Any offspring with an SSD in the window $[\alpha, \beta]$ is preferred as the new parent. If no offspring falls within this range, one of them is returned at random. If both offspring fall in the window, the one with the smallest SSD is returned. This approach is based on the assumption that a semantically different individual, yet sufficiently close to its parent, tends to yield superior results [170].

5.3.1 Extending semantics to work with selection policies in MCTS

The concept of semantics, including the Sampling Semantics (SS), can be extrapolated to a program p that serves as a tree policy in MCTS. We denote $MCTS_p$ to the vanilla MCTS algorithm using p as the formula for the tree policy. In MCTS, the input to the tree policy is a statistical tree τ, and its output is the selected leaf node $n \in \tau$. Since the relevant attribute of n for the tree policy is its numerical evaluation $v(n)$, we can infer

Algorithm 2 SIEA-MCTS

```
1: Input: Number of gen. $G$, lambda $\lambda$, fitness iterations $S$, statistical tree $T$
2: Output: Evolved tree policy
3: procedure EVOLVING_UCB1_EA_SIEA
4:     P ← UCB1 formula
5:     for $g \leftarrow 0, \cdots, G$ do
6:         for $i \leftarrow 0, \cdots, \lambda$ do
7:             $O_i$ ← subtree_mutation(P)
8:             a_fitness ← 0
9:             for $s \leftarrow 0, \cdots, S$ do
10:                temp_fit($O_i$) ← select $T(S)$ and rollout($O_i$)
11:                update $T$
12:                a_fitness($O_i$) ← a_fitness($O_i$) + temp_fit($O_i$)
13:            end for
14:            fitness($O_i$) ← a_fitness($O_i$) / S
15:        end for
16:        P ← Sem_Sel(O,P) procedure
17:    end for
18:    return P
19: end procedure
20: ________________________________________________
21: Input: Population offspring $O$, Parent $P$
22: Output: Best program based on fitness and semantics
23: procedure SEM_SEL(O,P)
24:    $H_f$ ← max(fitness($O$))
25:    if More than one offspring from $O$ equals $H_f$ then
26:        SSD ←Sampling_sem_dist($O, P$)
27:        if one individual within SSD range ($\alpha$, $\beta$) then
28:            New_P ← individual within range
29:        else if more than one individual within SSD range ($\alpha$, $\beta$) then
30:            New_P ← individual closest to lower-bound $\alpha$
31:        else
32:            New_P ← random($O$)
33:        end if
34:    end if
35:    return New_P
36: end procedure
```

that the output of $p(\tau)$ is equal to $v(n)$. Therefore, assuming a consecutive and finite set of $MCTS_p$ iterations are executed as part of a fitness evaluation for program p, we can now define the SS of the tree policy p as shown in Def. 2.

Def. 2 *The Sampling Semantics $SS(p)$ of a program p used as the tree policy of an MCTS algorithm ($MCTS_p$) is the vector of evaluations $v(n)$ of each node $n \in N$. Given k iterations of $MCTS_p$, N is the collection of leaf nodes from the statistical tree selected by p.*

The SS of a tree policy is represented as $SS(p) = [p(\tau_{t=0}), p(\tau_{t=1}), \cdots, p(\tau_{t=k})]$, where k denotes the number of $MCTS_p$ iterations carried out using policy p. In this case, the input set for any tree policy p consists of a collection of versions of the same statistical tree, denoted as $I = \{\tau_{t=0}, \tau_{t=1}, \cdots, \tau t = k\}$, where τ_t is updated according to p whenever $p(\tau_t)$ is computed. Therefore, even if the initial statistical tree $\tau_{t=0}$ is identical when provided to two tree policies p and q, the subsequent input set $\{\tau_{t=1}, \tau_{t=2}, \cdots, \tau_{t=k}\}$ will differ. This occurs because policies p and q update $\tau_{t=j}$ differently whenever $p \neq q$. Consequently, the SSD between two tree policies p and q is calculated using Equation 5.1

$$SSD(p,q) = \sum_{l=0}^{k} \frac{|p(\mathsf{i}_l) - q(\mathsf{j}_l)|}{k} \tag{5.1}$$

Where l is the index of the elements in each ordered input set (in descending order), and $\mathsf{i} \in I$ and $\mathsf{j} \in J$ are elements in the input sets for p and q respectively. The input sets I and J share the first element $\tau_{t=0}$, which corresponds to the initial statistical tree. However, the remaining elements may not be the same. A significant difference between Equation 5.1 and the previously defined Equation 2.2 is that $SS(p)$ and $SS(q)$ are now sorted before the element-wise comparison is performed. The rationale behind ordering the collection of rewards is that the tree policy aims to identify the best leaf node in the statistical tree at each iteration, considering both exploration and exploitation aspects. Thus, the ordering makes SSD compare the maximum rewards of p with those of q, and the minimum rewards of p with those of q, respectively. In simpler terms, the SSD effectively measures the difference in the exploration-exploitation balance between the tree policies.

Semantics under the context of MCTS tree policies rely on the aggregation of rollout outcomes. For this reason, its impact may vary across various applications. For example, the available rewards in the Function Optimisation Problem (FOP) that will be discussed in Section 5.4 are binary, which increases the probability of equal fitness among individuals. However, in applications that involve a wider range of rewards like Carcassonne, ties become less likely. In summary, SIEA-MCTS and EA-MCTS are unique algorithms with an interesting proposal. However, it comes with the downside of requiring addi-

Table 5.1: Vanilla MCTS parameters

Parameter	Value
Tree policy	UCT with UCB1, where $C \in \{0.5, 1, \sqrt{2}, 2, 3\}$
Default and expansion policy	Random uniform
Rollouts, Iterations	1, 5000

Table 5.2: SIEA-MCTS and EA-MCTS parameters

Parameter	Value
Initial UCB1	$C = \sqrt{2}$
Rollouts	1
(μ+λ)-ES	$\mu = 1, \lambda = 4$
Generations g, Fitness iterations S	$g = 20$, $S = 30$
Genetic operator	Subtree mutation (90% − 10% policy)
Mutation subtree generation method	Full (depth $\sim$ Uniform$(1, 3)$)
Initialisation Method	UCB1 formula + λ mutations
Maximum syntax tree depth	8
SSi	$L = 0.1, U = 0.5$
Total fitness iterations	$\lambda * g * S + S = 2,430$
Total MCTS iterations	$5,000$

tional parameters for the evolutionary process. Both EA-MCTS and SIEA-MCTS are a first-of-a-kind type of approach that evolves MCTS's selection policy on the fly.

5.4 FOP experimental setup

The algorithms were allocated a number of iterations that allowed them to explore the bottom of the tree given enough exploitation. This can happen if the search focuses on a branch of the tree for a time long enough to expand it to the end. The parameters for every vanilla MCTS used in this experiment are shown in Table 5.1.

Table 5.2 presents the parameters that were used in the SIEA-MCTS algorithm. SIEA-MCTS employs more parameters than MCTS, and are also utilised in the EA-MCTS algorithm. The parameters were hand-tuned and are consistent with those reported in our IEEE Transactions on Games article [61].

The subtree mutation in EA-MCTS has a 90% − 10% policy, meaning that it has a 90% of choosing an internal node and a 10% probability of selecting a leaf node in the parent's syntax tree to be mutated. The selected node is swapped with a subtree randomly generated using the *full* method [103] previously described in Chapter 2. This method generates syntax trees with a random uniform depth for all of its branches, set between 1 and 3. The mutation operator is constrained to produce offspring that are

Table 5.3: Function Optimisation Problem parameters

Parameter	**Value**
Initial state domain $[a_0, b_0]$	$a_0 = 0, b_0 = 1$
Branching factor K	$k = 2$
Terminal state size threshold	$t = 10^{-6}$
Function $f(x)$	$f_1(x) = \sin(\pi x)$
	$f_2(x) = 0.5\sin(13x)\sin(27x) + 0.5$
	$f_3(x) = \begin{cases} 0.5 + 0.5 \cdot \lvert\sin(\frac{1}{x^5})\rvert & \text{when } x < 0.5, \\ \frac{7}{20} + 0.5 \cdot \lvert\sin(\frac{1}{x^5})\rvert & \text{when } x \geq 0.5. \end{cases}$
	$f_4(x) = 0.5x + (-0.7x + 1)\sin(5\pi x)^4$
	$f_5(x) = 0.5x + (-0.7x + 1)\sin(5\pi x)^{80}$

different from their parent and have a syntax tree's maximum depth smaller than 8. It does so by iteratively generating performing mutation until a valid offspring is found. The terminal set T and function set F used in the construction of the evolved formulae in EA-MCTS and SIEA-MCTS are $T = \{Q_i, N, n_i, C\}, F = \{+, -, \times, \div, \log, \surd\}$, where Q_i is the reward and n_i the number of visits of state i. N is the number of visits of the parent of i, and C is the exploration-exploitation constant, which is randomly sampled from the set $\{0.25, 0.5, 1, 2, 3, 5, 7, 10\}$. Regarding the function set, the division operator returns 1 for any divisor with an absolute value lower than 0.001. The natural logarithm and square root operators take the absolute value of their inputs. Given that the evolution is performed online, the total population is set to 5, which is the sum of the parent and offspring, and the number of generations is limited to 20.

We allocated $5,000$ iterations for each algorithm, both for the vanilla and the evolutionary-based MCTS variants. In both SIEA-MCTS and EA-MCTS, evolution involves performing a set of fitness iterations to evaluate each of the the evolved formulae. In our implementation, the evaluation of a single formula requires $S = 30$ fitness iterations. SIEA-MCTS conducts S iterations for λ individuals and g generations as fitness evaluations, resulting in $\lambda * g * S + S = 2430$ total fitness iterations invested in the evolutionary process.

Every MCTS variant was asked to make a decision from the initial state in FOP using the functions and parameters described in Table 5.3 for 100 runs. The experiments share random seeds within equivalent runs, meaning that each MCTS variant was tested under equal random rewards. It also implies that EA-MCTS evolved formulae differ from those evolved by SIEA-MCTS only whenever semantics are involved in the evolutionary process. The results are presented in Section 5.5.

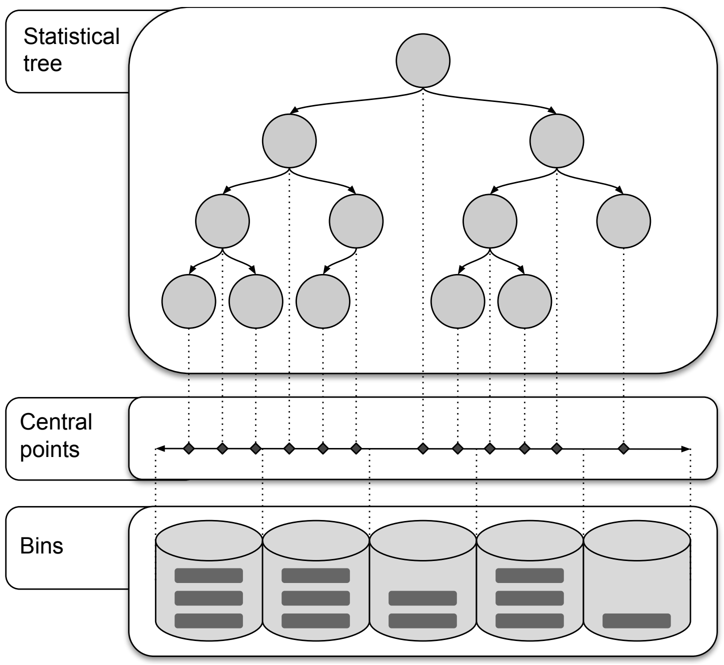

Figure 5.2: The nodes from the statistical tree are allocated in the histogram's bins according to the central point of the state they represent.

5.5 FOP results

We will examine how MCTS explores the game tree of each FOP function when tasked with determining the optimal move from the initial state. To do so, we will analyse the statistical trees produced by each MCTS variant, by showing a histogram and a table with the results for every function, to finally wrap up with an integral analysis of all the functions in Section 5.5.6. The histograms aim to demonstrate the distribution and search intensity across the domain for each function, and are constructed with the nodes in the statistical trees. The histograms are averaged over all the runs, where the nodes of the statistical tree are allocated in 128 bins based on the central point of the states they represent. An illustration of their construction is shown in Figure 5.2

The resulting histograms will be presented in Figures 5.4 through 5.8, corresponding to functions f_1 to f_5, in that order. Tables 5.4 through 5.8 complement said histograms by describing the characteristics of the statistical trees. The calculation of those characteristics is explained next using the statistical tree example given in Figure 5.3.

- **Node expansion rate per iteration**: Ratio of expanded nodes within the final statistical tree to the total number of iterations executed by the algorithm. MCTS usually adds a node to the statistical tree in each iteration achieving a rate of 1. However,

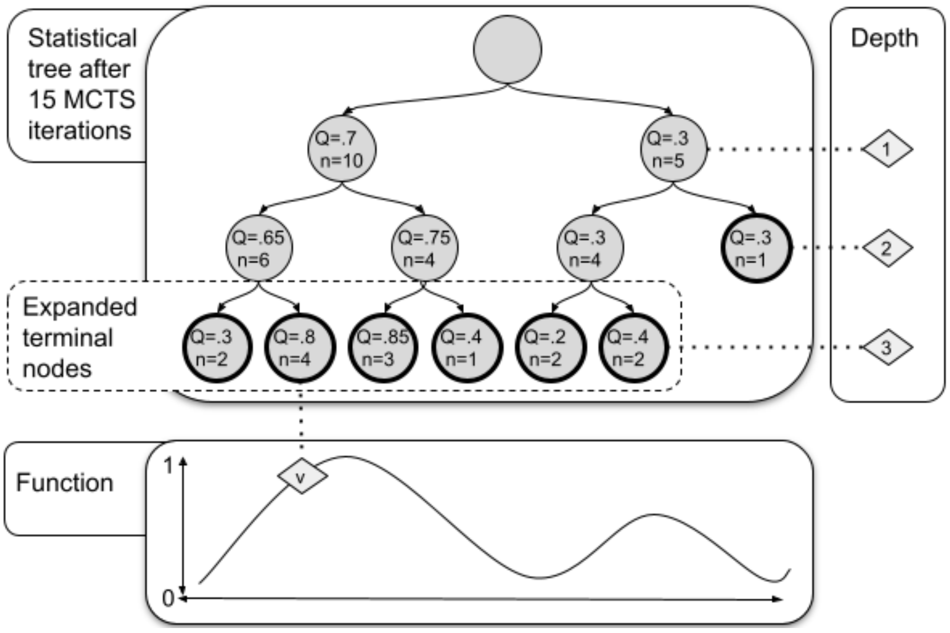

Figure 5.3: Sample statistical tree generated after 15 MCTS iterations, where each node has a reward Q and a visit count n. The game tree has a maximum depth of 3. Terminal nodes are enclosed in the dashed box, and leaf nodes are highlighted with darker contours. The reward v is MCTS's belief of the outcome with optimal play.

when MCTS selects a terminal node during its selection phase, it cannot expand the statistical tree further on that iteration's expansion phase, hence reducing the node expansion rate per iteration. Selecting a terminal node is more likely for agents favouring exploitation. As an example, the statistical tree shown in Figure 5.3 expanded only 12 nodes after 15 iterations, resulting in a node expansion rate per iteration of $\frac{12}{15} = 0.8$.

- **Number of terminal states reached**: Number of unique terminal states present in the final statistical tree. This metric illustrates the diversity of terminal states MCTS explored, providing insights into the algorithm's exploration capabilities. As an example, the statistical tree in Figure 5.3 has a number of terminal states reached of 6.
- **Leaf nodes' average depth**: Average depth of the leaf nodes in the final statistical tree. Generally, a deeper statistical tree depicts more exploitation, whereas a shallower tree depicts more exploration. As an example, the value of this column equals 2.85 for the statistical tree in Figure 5.3.
- **Most visited node-based result**: Indicates MCTS's predicted optimal play outcome, reflecting the algorithm's ability to identify optimal moves. It is the evaluation of the leaf node reached by traversing the statistical tree from the root, selecting the most visited child at each step. This is illustrated as v in the function section of Figure 5.3.

5.5.1 FOP f_1 results

Figure 5.4 depicts the search conducted on each experiment when the MCTS variants try to find the optimal action from the initial state of the FOP with f_1 as its reward function. This function, plotted at the top of Figure 5.4, is selected to illustrate how each particular algorithm behaves when symmetrical rewards exist in different areas of the domain and there is a single global optimum, surrounded by the most rewarding region of the domain.

The bars are shaded based on the percentage of total iterations executed so far when each node was added to the statistical tree, depicting the behaviour of each search stage. The green bars represent the nodes expanded on the first third of the iterations, the blue bars the second third, and the brown bars the last third. From this information, it can be seen that as more iterations are performed, the search of each MCTS variant becomes more localised. Nodes corresponding to the latter portion of iterations (66.7 to 100% of the total, shown with the brown bars) are denser around the global optima when compared to those from the initial stages of the search. This tendency is the outcome of the bias towards exploiting the best nodes as more information is gathered. Note that the total number of nodes varies significantly among algorithms since not every statistical tree has the same number of nodes. In some instances, an MCTS algorithm may not add a node to the statistical tree if the node selected by the tree policy is a terminal state.

As anticipated, the C parameter of every version of the vanilla MCTS has a significant impact on the exploration/exploitation trade-off. An increase in C leads to more exploration and less exploitation. Therefore, the region of the search that receives attention widens as C grows. Furthermore, the average number of nodes allocated to agents with lower C near the global optimum is higher than that for agents with higher C.

Note that the height of the bars is not always smooth among the bars next to each other. These variations arise where a low-depth node's domain ends and another node's domain begins, reflecting the different intensities of the search for nodes closer to the root.

The Evolutionary Algorithm (EA)-based MCTS variants (SIEA-MCTS and EA-MCTS, presented in the last two rows of Figure 5.4) show a considerable preference for exploitation. The nodes are predominantly localised around the global optimum and in a larger count than those of the MCTS with $C = 0.5$ (i.e. the vanilla agent with less exploration).

The EA-based MCTS variants demonstrate success in sampling the region near the global optimum. However, very small bars can be observed at the left and right edges of their plots. These regions correspond to the global minima, and the presence of nodes in that area indicates undesired behaviour from the agents. Note that the average number of nodes allocated to that region is almost negligible because they reflect the impact of only a few evolved formulae out of the hundred runs. Those evolved formulae preferred regions with the lowest rewards instead of the highest rewards on rare occasions.

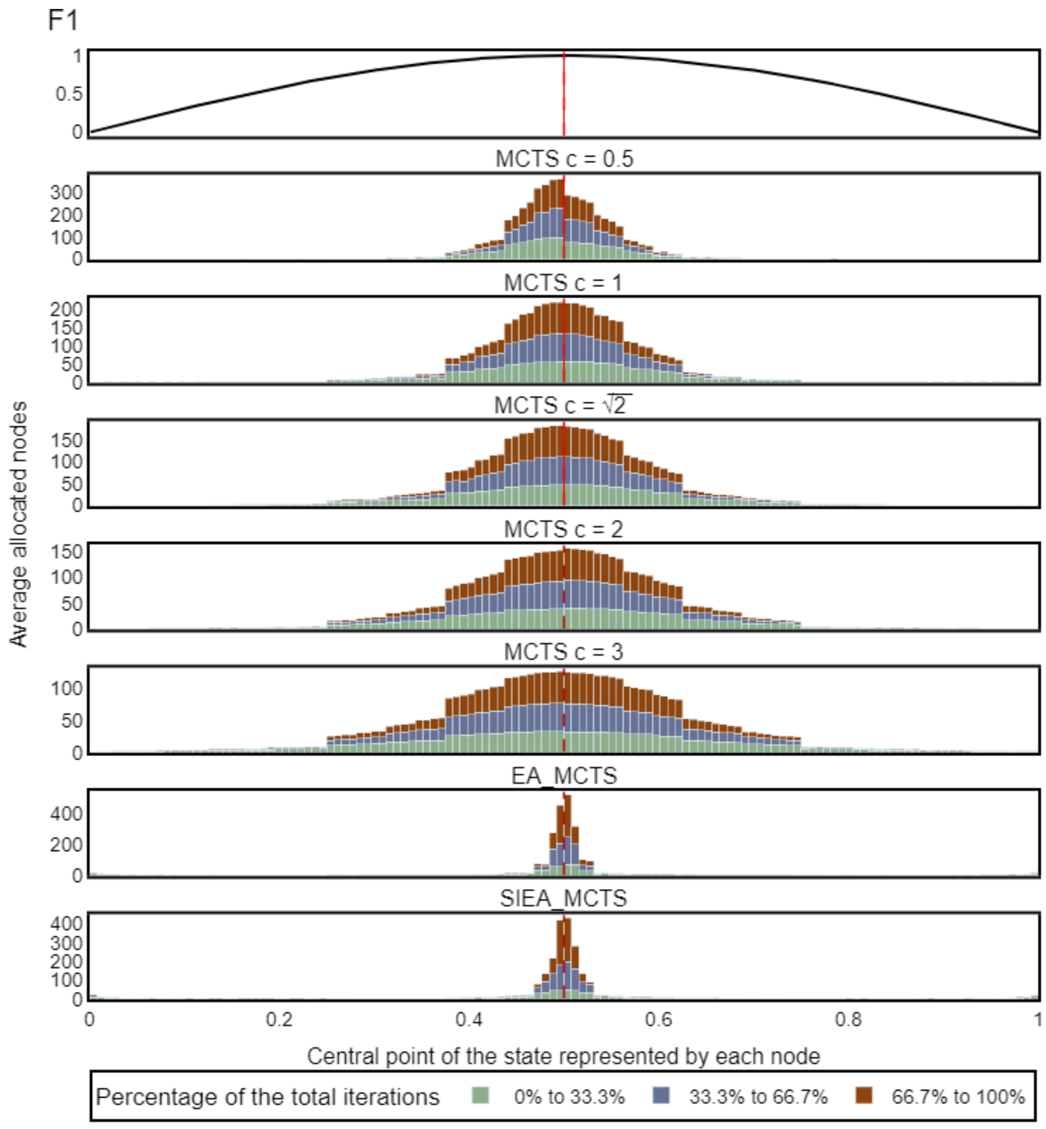

Figure 5.4: Statistical tree nodes sorted by the location of their respective states for f_1, plotted at the top and defined in Table 5.3. For the rest of the plots, the x-axis is the domain of the function and the y-axis is the average of allocated nodes. The dashed vertical red line shows f_1's global optimum.

Table 5.4: Average of 100 statistical trees grown by the agents for f_1.

MCTS variant	Node expansion rate per iteration	Number of terminal states reached	Leaf nodes' average depth	Most visited node-based result
MCTS C=0.5	1 ± 0	0 ± 0	14.04 ± 1.38	0.999 ± 0
MCTS C=1	1 ± 0	0 ± 0	13.14 ± 1.37	0.999 ± 0
MCTS C=$\sqrt{2}$	1 ± 0	0 ± 0	12.85 ± 1.39	0.999 ± 0
MCTS C=2	1 ± 0	0 ± 0	12.58 ± 1.29	0.999 ± 0
MCTS C=3	1 ± 0	0 ± 0	12.33 ± 1.18	0.999 ± 0
EA-MCTS	0.56 ± 0.25	544.52 ± 469.55	13.53 ± 2.82	0.999 ± 0
SIEA-MCTS	0.54 ± 0.24	417.62 ± 376.36	13.39 ± 2.85	0.999 ± 0

Table 5.4 describes the agents' resulting statistical trees for f_1. The data represents an average of 100 runs. This table shows that all the MCTS variants successfully identified the global optimum in every run, as all of them have a most visited node-based result close to 1, which is this function's maximum value. None of the vanilla MCTS variants have expanded terminal nodes in their respective statistical trees, reflected by 0 number of terminal states reached, which means that they did not reach the bottom of the tree. This is a consequence of the combination of their high exploration and the wide region of interesting rewards spreading the search. In contrast, every EA-based MCTS variant added terminal nodes, with EA-MCTS adding more than SIEA-MCTS.

In the sixth row of Table 5.4, EA-MCTS achieved a node expansion rate per iteration of 0.56 on average. This means that in 44% of the cases, equivalent to $2,200$ iterations, it repeatedly selected up to 544.42 unique terminal nodes (as shown in the third column), thus not being able to expand them. A similar pattern is observed with SIEA-MCTS, as noted in the last column, with a node expansion rate per iteration of 0.54 and a number of terminal states reached of 417.62. The number of terminal states reached for both EA-based MCTS variants is relatively low compared to the number of iterations that did not result in node expansion. This suggests that both agents have evolved selection policies that repeatedly select a limited number of tree branches, indicating a strong preference for exploitation over exploration. A detailed analysis of the evolved selection policies will be provided in Section 5.8.

The leaf nodes' average depth column illustrates that for vanilla MCTS variants, the depth of the leaf nodes in the statistical tree decreases with an increase in the C parameter, ranging from 14.04 at $C = 0.5$ to 12.33 at $C = 3$. For EA-based MCTS variants, the trees are relatively deeper, with leaf nodes' average depth values of 13.53 for EA-MCTS and 13.39 for SIEA-MCTS. The standard deviation in leaf nodes' average depth for the EA-based variants, at 2.82 for EA-MCTS and 2.85 for SIEA-MCTS, is considerably higher compared to the vanilla MCTS variants, which range between 1.18 and 1.39. This finding highlights the variability in the statistical trees generated by the evolved formulae.

The differences between the EA-MCTS and SIEA-MCTS variants are minimal across most metrics, except for number of terminal states reached. This observation suggests that incorporating semantics influenced the construction of the statistical trees to a certain degree, but did not significantly alter the most visited node-based result in f_1.

5.5.2 FOP f_2 results

Figure 5.5 depicts the search conducted by the agents in f_2, the multimodal function. As can be observed in Figure 5.5, all the variants expanded the most nodes around the global optimum, with some agents also investing resources in local optima. Notably, the MCTS $C = \sqrt{2}$ and MCTS $C = 2$ variants showed minimal investment at any local optima. On the other hand, MCTS $C = 0.5$ and MCTS $C = 1$ explored both global and local optima. Moreover, they invested resources around the local optimum on the left-hand side of the plot at all stages of the search, illustrated by the multiple shades of the bars on that region. MCTS $C = 3$ also invested resources on that local optimum, but it managed to shift its focus towards the global optimum on its last portion of iterations.

The EA-based MCTS variants exhibit a consistently low number of nodes allocated (y-axis), even around the global optimum, contrasting sharply with the node allocation observed in vanilla MCTS variants. This discrepancy comes from averaging the results from effective formulae that explore extensive areas of the search space with formulae that get stuck and expand fewer nodes in the statistical tree. While it might appear that EA-based MCTS variants distribute their search more evenly across the search domain, it is a secondary product of the reduced node count at the tallest bars. This reduction diminishes the visual contrast, making even minor bars more prominent.

Interestingly, the EA-based MCTS variants have allocated some resources around local and global minima, similar to what we observed in f_1. Moreover, those nodes were expanded in the final stages of the search (66% to 100% of the total iterations), which confirms that the EA-based MCTS variants evolved formulae that preferred low-reward regions of the domain. As a final observation, the allocation of nodes along the search space is distributed similarly for both EA-based MCTS variants, with a slightly higher number of nodes allocated by EA-MCTS on the global optimum.

The results for f_2 are presented in Table 5.5. High exploitation proves unsuitable in this function, as evidenced by the MCTS $C = 0.5$ having the lowest most visited node-based result among the vanilla variants. The best most visited node-based result was achieved by the MCTS $C = 2$, closely followed by MCTS $C = 3$ and MCTS $C = \sqrt{2}$, in that order. On the other hand, the EA-based MCTS variants achieved a maximum value of 0.922,

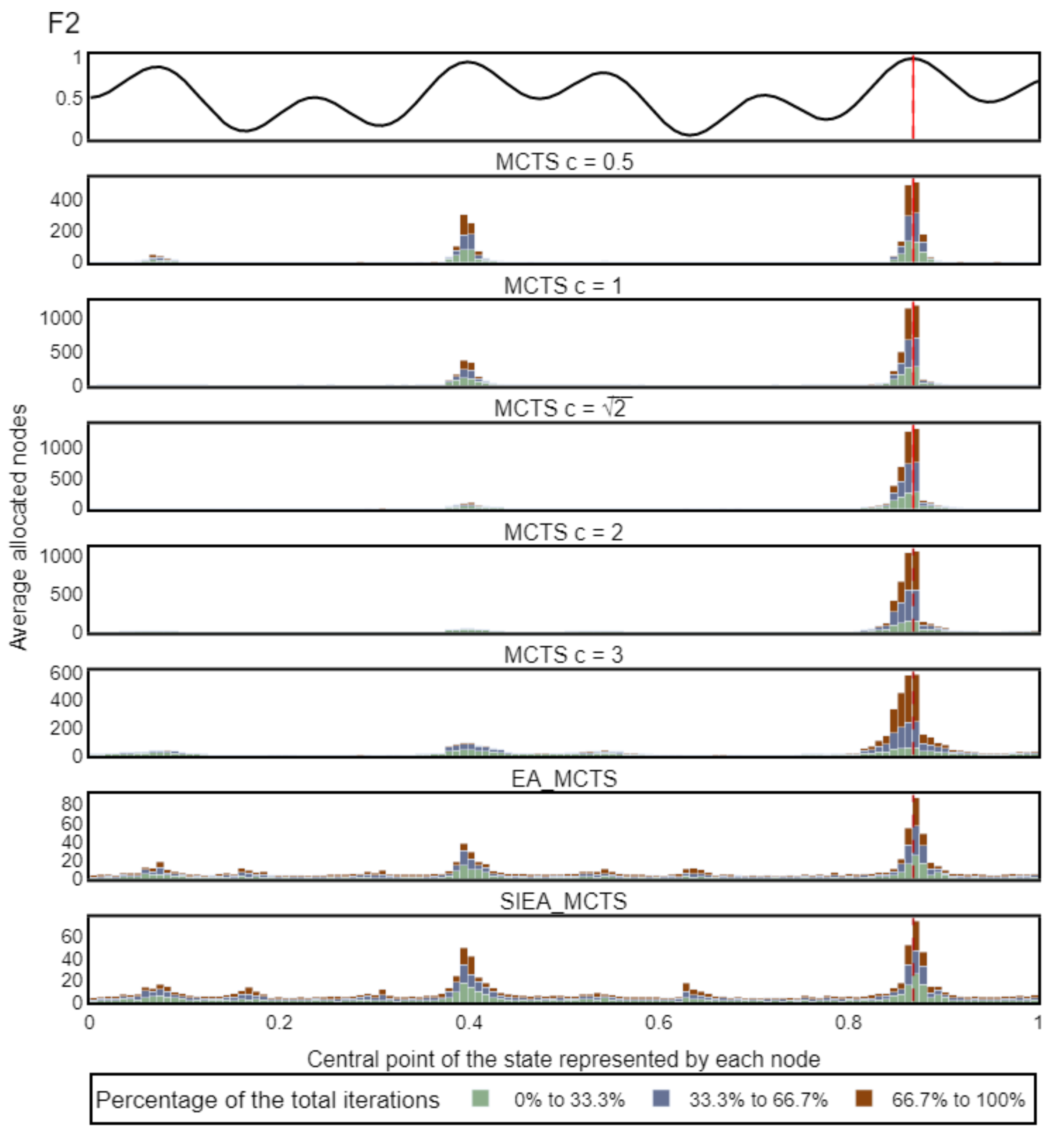

Figure 5.5: Statistical tree nodes sorted by the location of their respective states for f_2, plotted at the top and defined in Table 5.3. For the rest of the plots, the x-axis is the domain of the function and the y-axis is the average of allocated nodes. The dashed vertical red line shows f_2's global optimum.

Table 5.5: Average of 100 statistical trees grown by the agents for f_2.

MCTS variant	**Node expansion rate per iteration**	**Number of terminal states reached**	**Leaf nodes' average depth**	**Most visited node-based result**
MCTS C=0.5	0.49 ± 0.2	621.06 ± 383.92	15.65 ± 2.14	0.936 ± 0.04
MCTS C=1	0.96 ± 0.05	1063.9 ± 381.89	15.61 ± 2.21	0.964 ± 0.02
MCTS C=$\sqrt{2}$	1 ± 0	739.44 ± 260.14	15.05 ± 2.54	0.971 ± 0.01
MCTS C=2	1 ± 0	196.51 ± 146.44	14.25 ± 2.55	0.973 ± 0
MCTS C=3	1 ± 0	0 ± 0	12.96 ± 2.12	0.972 ± 0.01
EA-MCTS	0.21 ± 0.08	68.96 ± 49.39	11.35 ± 3.27	0.922 ± 0.05
SIEA-MCTS	0.23 ± 0.11	73.65 ± 66.67	11.37 ± 3.21	0.922 ± 0.05

which is lower than any vanilla variant. This is a consequence of the EA-based MCTS variants' propensity to get stuck in local minima. This phenomenon is more evident in this function, given the comparatively low node expansion rate per iteration and number of terminal states reached of the EA-based MCTS variants. Both EA-MCTS and SIEA-MCTS report low node expansion rate per iteration (0.21 and 0.23, respectively), even lower than in f_1 (0.56 and 0.54, respectively) meaning that more terminal nodes were reached in f_2 than in f_1. However, the number of terminal states reached for those agents is lower in f_2 (69.96 for EA-MCTS and 73.65 for SIEA-MCTS) than in f_1 (544.52 for EA-MCTS and 417.62 for SIEA-MCTS. These observations suggest that a greater proportion of evolved formulae got stuck selecting the same branches of the tree multiple times in f_2, compared to f_1.

Note that the vanilla MCTS variants have more number of terminal states reached than any of the EA-based MCTS variants. This is an indication that the vanilla MCTS variants never get stuck, which is a characteristic of the UCB1. The UCB1 formula is designed to eventually select any node with optimism, regardless of the rewards obtained. This property is then proven to be occasionally lost when the formula is evolved.

The vanilla MCTS variants with $C = \sqrt{2}$ and $C = 2$ expanded nodes on every iteration (node expansion rate per iteration = 1), but some of them were terminal states (number of terminal states reached > 0). This means that terminal nodes were expanded, but not selected again, otherwise the node expansion rate per iteration would be lower than 1. In contrast, the vanilla MCTS variant with $C = 3$ did not reach any terminal state, as reflected by the 0 number of terminal states reached.

5.5.3 FOP f_3 results

Figure 5.6 displays the search conducted by the agents in f_3, the rugged function. f_3 is remarkable for having multiple global optima, which are unevenly distributed on the left portion of the domain. The concentration of these global optima gradually decreases as the domain of the function goes from 0 to 0.5. The agents in our study preferred the region with global optima near the centre of the domain, where the global optima are surrounded by larger regions with higher values, and the fitness landscape is less rugged. This circumstance facilitates the agents' ability to locate and consistently exploit the optima. Conversely, if the region were more volatile, small variations in the input would yield significant fluctuations in the output, making it more difficult for the agents to identify the global optima. Note that the preference for the global optima with the smoothest surroundings becomes less apparent as C increases.

Even though the domain's right-hand side lacks a global optimum, it is still explored to some extent by the agents, though not as much as the left-hand side. This is because the right-hand side features local optima surrounded by consistent rewards, making it initially appealing to the search due to the high average value of the region. The reward structure, as depicted in Figure 4.5, reveals that from a tree depth of 3 up to a tree depth of 6, the highest rewards available to the search are situated on the domain's right-hand side. However, the agents will not exploit the right-hand side as heavily since the globally optimal points of the left-hand side are likely to be eventually discovered.

With MCTS $C = 0.5$ and $C = 1$, a couple of global optima are discovered and thoroughly sampled. The lower count of nodes (y-axis) for MCTS $C = 0.5$ in contrast to MCTS $C = 1$ suggests that the lack of exploration of the former made it expand fewer nodes, as it likely reached the bottom of the tree. As the value of C increases, vanilla MCTS variants exhibit a more dispersed node distribution on the domain's left-hand side, while also intensifying their search efforts around the local optima on the right-hand side.

The EA-based MCTS variants exhibit erratic preferences, with scattered nodes on the whole domain. This is a consequence of the EA-based MCTS variants' tendency to get stuck in the first local or global optimum found, which varies on each run. Remarkably, the EA-based variants exhibit a significant number of nodes around the right-most global minimum in f_3 found on the right-hand side of the domain. This global minimum is the one with the most consistent rewards around it, making it the most appealing to the formulae that occasionally evolved to prefer low-reward regions.

Table 5.6 highlights that the EA-based MCTS variants have considerably fewer nodes in their statistical trees than the vanilla MCTS variants, indicating a lower exploration intensity and explaining the low node counts found in the last couple of plots in Figure 5.6.

Table 5.6 also shows that MCTS $C = 1$ achieved the highest most visited node-based result for f_3, with a value of 0.993. The vanilla agent with the worst performance was the MCTS with a value of $C = 3$, with a value of 0.903 which is significantly lower than the rest of the vanilla MCTS variants. This can be explained by the combination of high exploration and the large number of global optima present in f_3, making MCTS $C = 3$ not

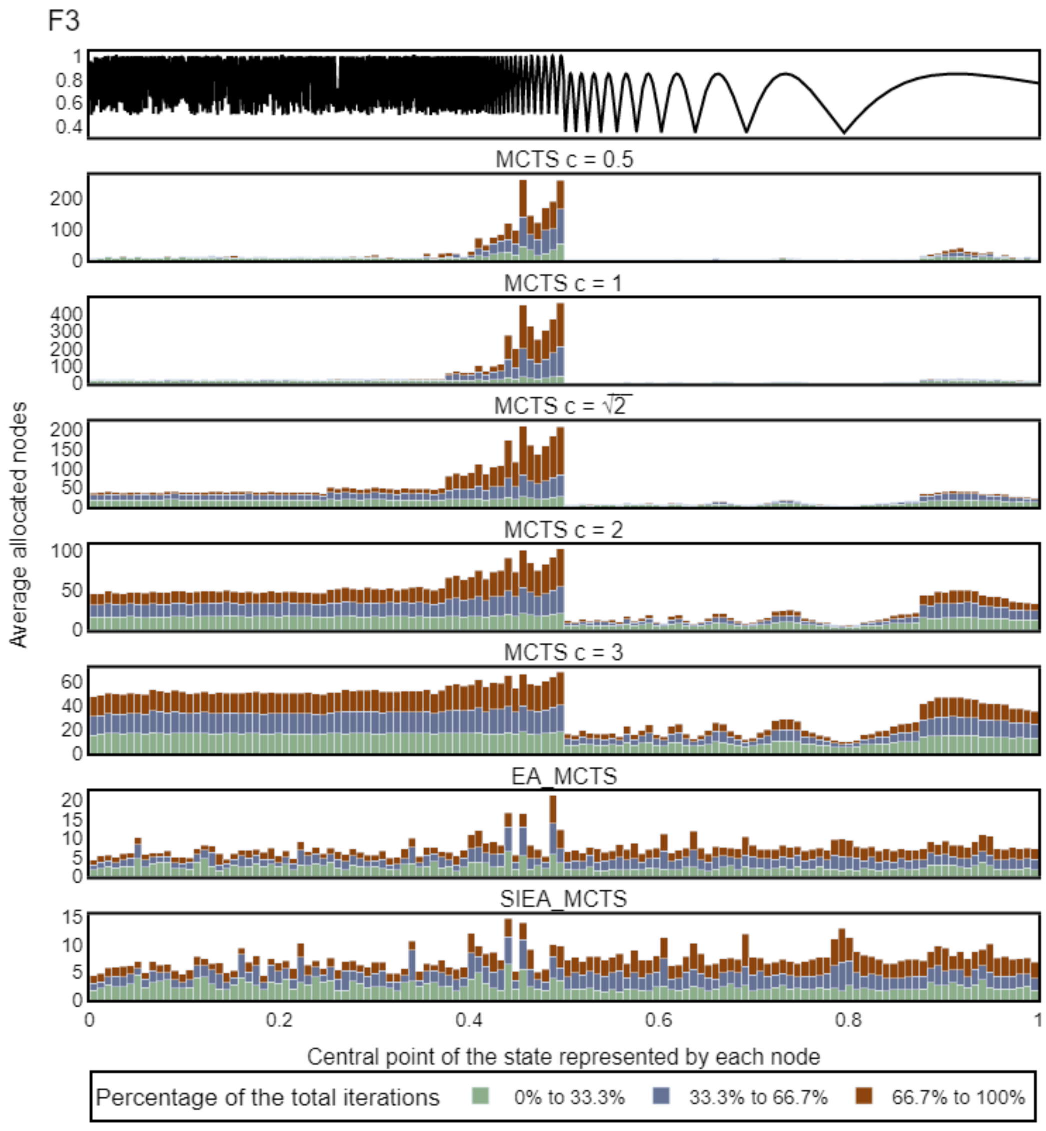

Figure 5.6: Statistical tree nodes sorted by the location of their respective states for f_3, plotted at the top and defined in Table 5.3. For the rest of the plots, the x-axis is the domain of the function and the y-axis is the average of allocated nodes.

Table 5.6: Average of 100 statistical trees grown by the agents for f$_3$.

MCTS variant	**Node expansion rate per iteration**	**Number of terminal states reached**	**Leaf nodes' average depth**	**Most visited node-based result**
MCTS C=0.5	0.53 ± 0.13	492.54 ± 241.58	14.62 ± 2.77	0.970 ± 0.06
MCTS C=1	1 ± 0	117.18 ± 113.42	13.46 ± 2.34	0.993 ± 0.02
MCTS C=$\sqrt{2}$	1 ± 0	0 ± 0	12.28 ± 1.49	0.986 ± 0.02
MCTS C=2	1 ± 0	0 ± 0	11.82 ± 0.98	0.964 ± 0.07
MCTS C=3	1 ± 0	0 ± 0	11.66 ± 0.74	0.903 ± 0.13
EA-MCTS	0.19 ± 0.1	38.59 ± 30.76	10.47 ± 2.89	0.970 ± 0.07
SIEA-MCTS	0.19 ± 0.08	34.39 ± 22.53	10.42 ± 2.84	0.971 ± 0.06

able to sufficiently sample any of them, investing its resources in all of them instead. This can be further confirmed with Figure 5.6, where we can see that MCTS $C = 3$ expanded nodes in multiple regions of the domain, resulting in decreased precision for each local or global optimum.

For f$_3$, the only vanilla MCTS variants that expanded terminal states were the MCTS $C = 0.5$ and MCTS $C = 1$. The former also has a node expansion rate per iteration lower than 0.53, meaning that it reached the bottom of the tree repeatedly, whereas MCTS $C = 1$ did not revisit its expanded terminal nodes. Since f$_3$ has multiple global and local optima, it is likely for MCTS $C = 0.5$, which is the vanilla variant with the lowest exploration, to solely focus on the first few global or local optima that it finds. This is reinforced by the most visited node-based result of MCTS $C = 0.5$ being lower than some other vanilla variants because in some runs MCTS $C = 0.5$ failed to sample global optima from the left-hand side of the domain and got stuck on local optima on the right-hand side. On the other hand, the MCTS $C = \sqrt{2}$, MCTS $C = 2$, and MCTS $C = 3$ did not expand any terminal nodes, and have less leaf nodes' average depth as C increases.

The EA-based MCTS variants exhibited no significant differences among themselves. They faced difficulty in generating explorative formulae, as demonstrated by their low node expansion rate per iteration and number of terminal states reached, which imply that a large portion of the evolved formulae got stuck. We observed that the function's nature harmed the evolutionary process, as it offered good rewards in several regions, implying that the fitness evaluation of the formulae did not provide any incentive to explore the tree. However, the most visited node-based result of the EA-based MCTS variants are comparable to that of the vanilla MCTS variants.

5.5.4 FOP f_4 results

Figure 5.7 illustrates the search conducted by the agents in f_4, a function designed to exhibit a deceiving fitness landscape that could potentially pose a challenge to the agents. However, even with f_4's deceptive reward landscape, the vanilla MCTS variants were not significantly misled. Most vanilla MCTS variants had their nodes concentrated around the global optimum, except for MCTS $c = 0.5$, which allocated some resources around multiple local optima. Conversely, the EA-based MCTS variants consistently invested resources in every optima present in the function. Moreover, the EA-based variants were the only ones to explore the right-most local optimum. We can also observe some expanded nodes in the regions with the lowest rewards, especially on the left-hand side of the domain.

In Table 5.7, we observe that the maximum most visited node-based result is achieved by MCTS $C = \sqrt{2}$ with a value of 0.978, closely followed by MCTS $C = 2$, and MCTS $C = 3$. The lowest most visited node-based result among the vanilla variants is from MCTS $C = 0.5$, which was sometimes deceived towards local optima as shown in Figure 5.7. The lack of exploration from the latter is also reflected in the low node expansion rate per iteration and number of terminal states reached, meaning that the statistical trees it generated had fewer and more localised nodes. In contrast, the next vanilla agent with the lowest exploration, MCTS $C = 1$, has a number of terminal states reached larger than any other agent, suggesting that it found more regions of interest than MCTS $C = 0.5$. This is consistent with MCTS $C = 1$ also having the largest leaf nodes' average depth, meaning that many of the expanded nodes were sampled deeper in the tree. Furthermore, MCTS $C = 1$ has a number of terminal states reached of 0.91, which is lower than 1 but still larger than that of MCTS $C = 0.5$, which was 0.36. In summary, the slightly higher exploration of MCTS $C = 1$ was sufficient to avoid getting stuck in local or global optima as often as MCTS $C = 0.5$ while also expanding more nodes on the statistical tree, highlighting the importance of an adequate C value in deceptive functions like f_4.

The EA-based MCTS variants achieved a maximum most visited node-based result of 0.87, which is lower than any of the vanilla variants. In f_4, the number of terminal states reached by the EA-based MCTS variants (53.82 for EA-MCTS and 52.14 for SIEA-MCTS) paired with the low node expansion rate per iteration (0.21 for EA-MCTS and 0.23 for SIEA-MCTS) suggest that a large proportion of evolved formulae got stuck when searching the tree in f_4. Interestingly, MCTS $C = 0.5$ exhibited the most similar behaviour to the EA-based variants, with the lowest number of terminal states reached and node expansion rate per iteration among the vanilla variants, suggesting that it also got stuck in some regions of the tree. This showcases how a decrease in exploration in MCTS leads to the behaviour exhibited by the EA-based variants, at least in domains with small rewarding regions and deceptive fitness landscapes. This behaviour is not necessarily

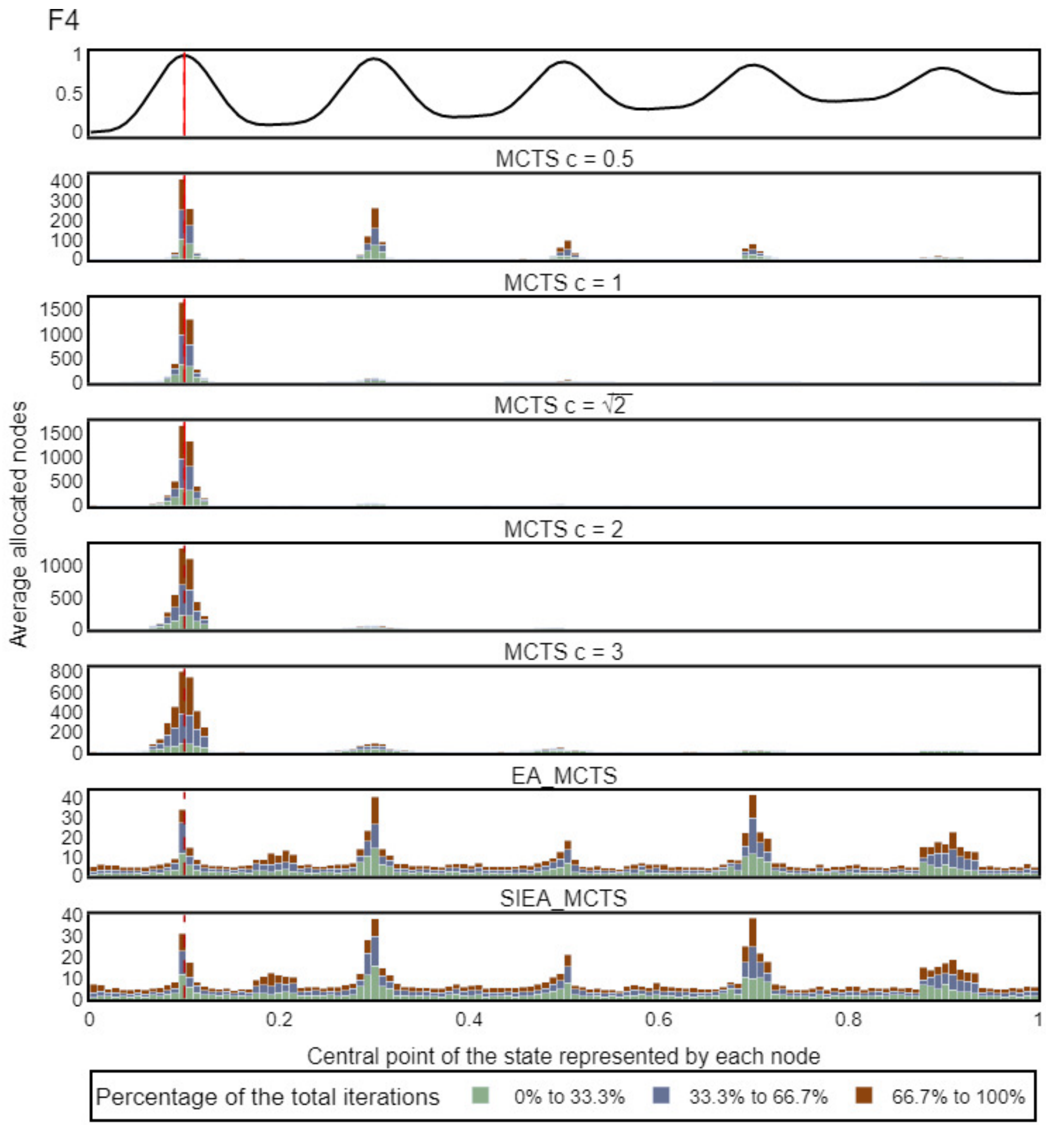

Figure 5.7: Statistical tree nodes sorted by the location of their respective states for f_4, plotted at the top and defined in Table 5.3. For the rest of the plots, the x-axis is the domain of the function and the y-axis is the average of allocated nodes. The dashed vertical red line shows f_4's global optimum.

Table 5.7: Average of 100 statistical trees grown by the agents for f_4.

MCTS variant	Node expansion rate per iteration	Number of terminal states reached	Leaf nodes' average depth	Most visited node-based result
MCTS C=0.5	0.36 ± 0.2	446.54 ± 372.17	15.3 ± 2.46	0.912 ± 0.05
MCTS C=1	0.91 ± 0.09	1212.1 ± 274.43	15.73 ± 2.27	0.974 ± 0.02
MCTS C=$\sqrt{2}$	1 ± 0.01	1029.2 ± 192.22	15.35 ± 2.49	0.979 ± 0
MCTS C=2	1 ± 0	415.14 ± 154.97	14.58 ± 2.59	0.978 ± 0
MCTS C=3	1 ± 0	0 ± 0	13.46 ± 2.42	0.977 ± 0
EA-MCTS	0.21 ± 0.1	53.82 ± 57.15	11.09 ± 3.15	0.874 ± 0.06
SIEA-MCTS	0.21 ± 0.1	52.14 ± 57.84	10.99 ± 3.08	0.873 ± 0.06

undesirable, as it can lead to the discovery of new regions of interest in the domain.

5.5.5 FOP f_5 results

Figure 5.8 displays the search conducted by the agents in f_5 (shown at the top of Figure 5.8), a function built to be more challenging to MCTS variants than f_4 due to its increased deceiving nature. To find the global optimum, a significant amount of exploration is necessary, which is only consistently achieved by vanilla MCTS variants with $C = 2$ and $C = 3$. Moreover, they only find and invest their resources around the global optimum after the first 33% of their iterations, as is reflected by the shades of the bars in that region (left-hand side of the fifth and sixth plots in Figure 5.8). On the other hand, although the rest of the vanilla variants did exhibit some interest around the global optimum, this region was not compelling enough for them to explore it further.

Interestingly, the maximum count of allocated nodes (y-axis) increases as C increases for the vanilla MCTS variants. The low count of allocated nodes in the plot for MCTS $C = 0.5$ is a consequence of the low count of total nodes in the statistical tree, with a node expansion rate per iteration of 0.11 according to Table 5.8. This means that the MCTS $C = 0.5$ reached the bottom of the tree and got stuck in the same terminal nodes repeatedly. In contrast, MCTS $C = 3$ has the highest maximum count of allocated nodes, with a low count around other local optima. Furthermore, the shades of the bars around the global optimum indicate that once found, the rewards of that optimum were large enough to attract the subsequent search.

The behaviour of the EA-based MCTS variants on f_5 mirrors their behaviour on f_4. They tend to sample local optima on the right-hand side of the domain and struggle to consistently find the global optimum. This is evident in Table 5.8, where the agents' most visited node-based result have the lowest values compared to the previous functions.

The best most visited node-based result is 0.957 achieved by MCTS $C = 3$, which

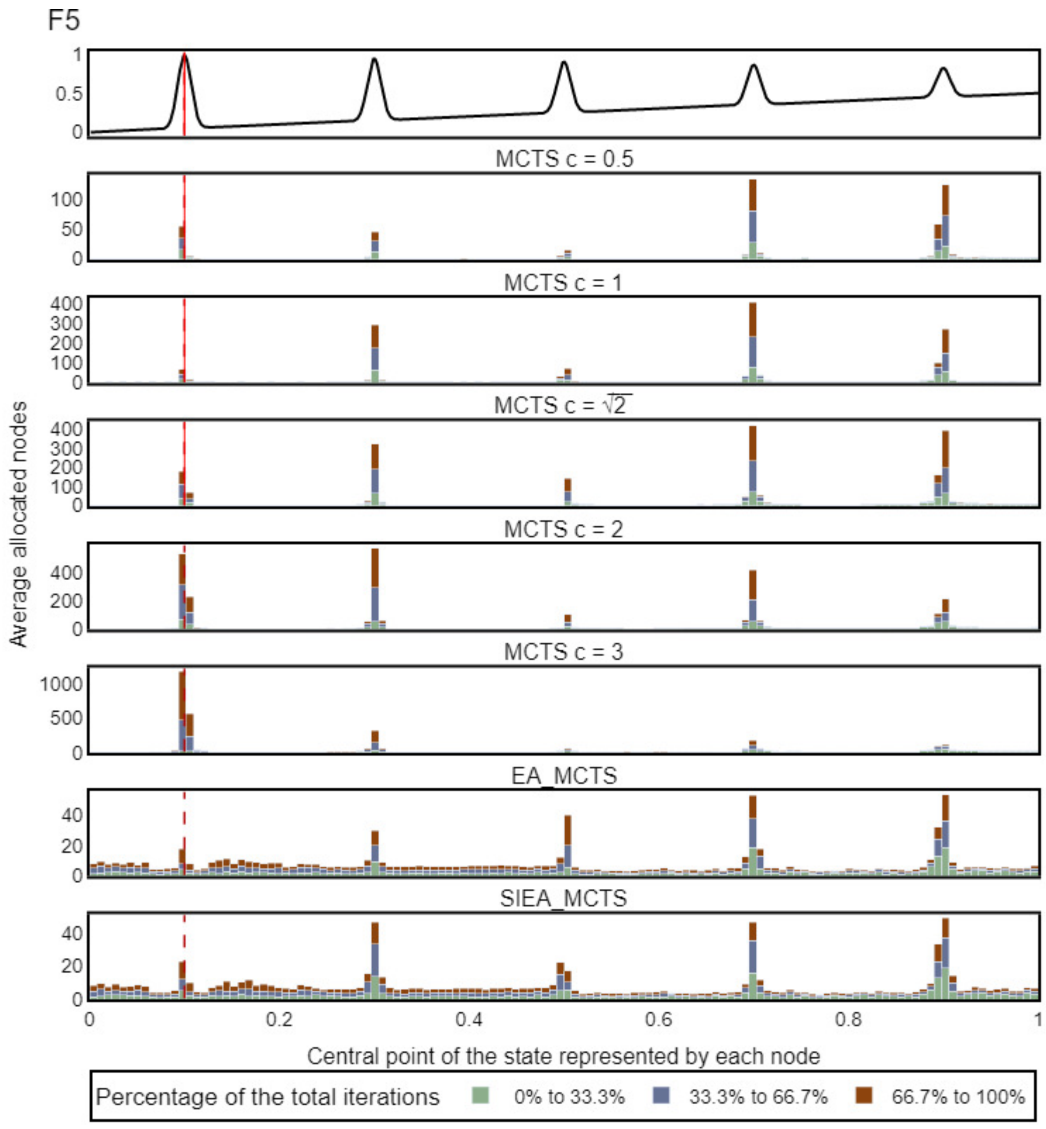

Figure 5.8: Statistical tree nodes sorted by the location of their respective states for f_5, plotted at the top and defined in Table 5.3. For the rest of the plots, the x-axis is the domain of the function and the y-axis is the average of allocated nodes. The dashed vertical red line shows f_5's global optimum.

Table 5.8: Average of 100 statistical trees grown by the agents for f_5.

MCTS variant	Node expansion rate per iteration	Number of terminal states reached	Leaf nodes' average depth	Most visited node-based result
MCTS C=0.5	0.11 ± 0.04	120.91 ± 98.29	14.35 ± 3.44	0.822 ± 0.08
MCTS C=1	0.31 ± 0.05	393.43 ± 152.67	15.01 ± 3.11	0.865 ± 0.05
MCTS C=$\sqrt{2}$	0.45 ± 0.06	567.67 ± 158.25	14.85 ± 3.16	0.875 ± 0.06
MCTS C=2	0.64 ± 0.07	771.57 ± 133.4	14.66 ± 3.17	0.915 ± 0.06
MCTS C=3	0.9 ± 0.06	704.82 ± 211.08	13.92 ± 3.01	0.957 ± 0.04
EA-MCTS	0.19 ± 0.1	53.26 ± 69.47	10.85 ± 3.08	0.809 ± 0.1
SIEA-MCTS	0.2 ± 0.09	46.98 ± 45.15	10.9 ± 3.06	0.812 ± 0.1

decreases as C decreases for the rest of the vanilla variants. As the rewards in f_5 are very deceptive, exploration proves to be crucial in finding the global optimum. The vanilla MCTS variants with $C = 0.5$ and $C = 1$ failed to find the global optima, as they did not explore the domain sufficiently, and performed particularly poorly in this function, with a most visited node-based result of 0.822 and 0.865, respectively. On the other hand, the EA-based variants achieved a maximum most visited node-based result of 0.81, which is lower than the lowest achieved by any vanilla variant (0.82) only by a small margin. Moreover, the EA-based MCTS variants in f_5 have the lowest most visited node-based result compared to all other functions but also have the largest standard deviation, suggesting greater volatility in the search.

We can observe that no MCTS variant had a node expansion rate per iteration equal to 1, meaning that all of them failed to expand nodes on some iterations, hence implying that every variant reached the bottom of the tree. The third column of Table 5.8 shows that the number of terminal states reached increases as C increases for the vanilla MCTS variants, except for MCTS $C = 3$, which also has the most volatile number of terminal states reached. This is likely a consequence of the variability in the timing of finding the region near the global optimum, a region only consistently found and exploited by MCTS $C = 3$ according to Figure 5.8.

Note that the standard deviation of the leaf nodes' average depth of every MCTS variant in f_5, which ranges from 3.01 to 3.44, is greater overall than in any of the previous functions, which can go as low as 0.74 for MCTS $C = 3$ in f_3 (shown in Table 5.6), illustrating the variability of the statistical tree construction for f_5 given the difficulty to find the best rewards in it.

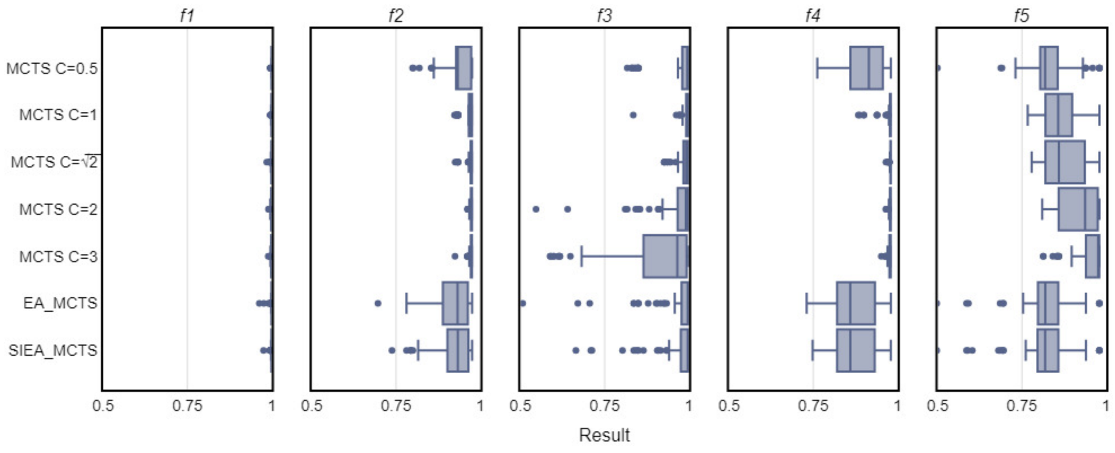

Figure 5.9: Distribution of the most visited node-based result by MCTS variant for all the functions in the FOP.

5.5.6 Integrated analysis of FOP results

We now compare the results of all the experiments of the FOP functions presented in this chapter. Figure 5.9 shows the distribution of the most visited node-based result metric, with a plot for each function and a row for every MCTS variant.

Figure 5.9 makes evident that different C parameters of the vanilla MCTS variants are required to perform better on each function according to its characteristics. The best mean most visited node-based result in the most deceptive function, f_5 (found in the right-most plot in Figure 5.9), is achieved by MCTS $C = 3$ values, whereas lower C values performed better in f_3, the function with a rugged reward landscape and multiple global optima (found in the middle plot in Figure 5.9). Interestingly, the EA-based MCTS variants have a most visited node-based result distribution similar between them and to MCTS $C = 0.5$, for each FOP function.

We performed a D'Agostino and Pearson's normality test on the most visited node-based result of every MCTS variant for every function. The null hypothesis of the test is that the data is normally distributed with a significance level of 0.05. The test rejected the null hypothesis for almost every MCTS variant in every function, meaning that the data is not normally distributed. In f_3, the variants EA-MCTS and MCTS had p-values of 0.069 and 0.103 respectively, meaning that there is not enough evidence to reject the null hypothesis for them. Overall, the normal distribution is not expected, as the most visited node-based result is a metric bounded between the minimum and the maximum of each function, making its distribution skewed towards the maximum, as shown in Figure 5.9.

Given that we learned that the distribution of the most visited node-based result cannot be assumed to be normal, we performed a non-parametric test, the two-sided Mann-Whitney U test, to compare the performances of each MCTS variant on every

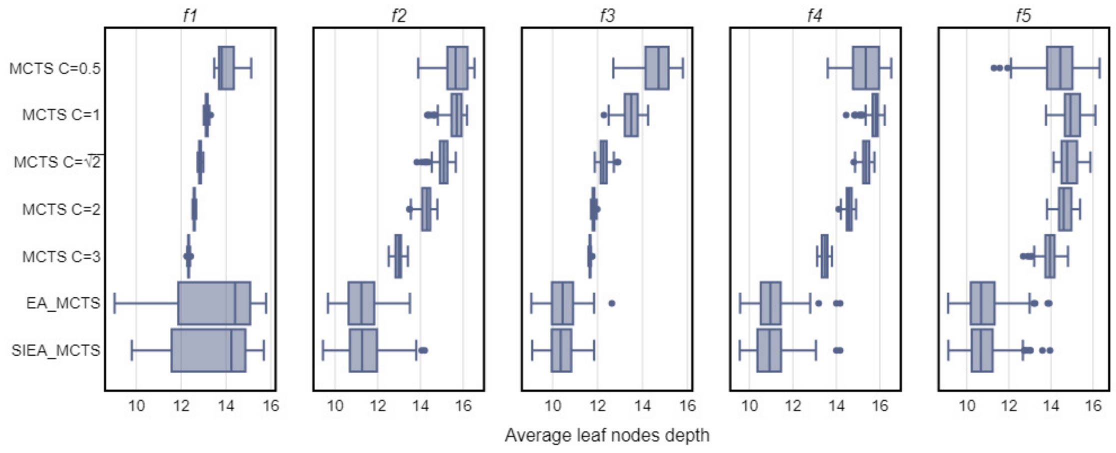

Figure 5.10: Average leaf node depth by MCTS variant and FOP function, extracted from the final statistical tree generated by each MCTS variant.

function independently with an alpha level of 0.05. The null hypothesis of the test is that the samples of the results for each variant come from the same distribution. We found that, when comparing different versions of the vanilla MCTS variants, those with similar C values showed statistically similar results. This suggests that minor adjustments to the MCTS settings do not drastically change how well the method works in most situations. For example, MCTS $C = 1$ and $C = \sqrt{2}$ have no statistically significant difference between them in most of the functions, whereas MCTS $C = 1$ and $C = 3$ do. However, function f_5 is the exception, as most of the vanilla variants perform significantly differently from each other regardless of their C. In f_5, only the vanilla MCTS variants with $C = 1$ and $C = \sqrt{2}$ did not have a significantly different most visited node-based result between them, with p-value $= 0.283$, p-value > 0.05.

This statistical test also indicates that there is no significant difference between the performances of both EA-based MCTS variants for any function. However, note that this test took into consideration only the most visited node-based result, ignoring how the statistical tree was constructed. For instance, the performance of EA-MCTS is not significantly different from that of MCTS $C = 0.5$ for functions f_1, f_2, f_3, and f_5, but their statistical trees are structured very differently. This difference can be observed in Figure 5.10 where the leaf nodes' average depth for each variant and function is shown.

From Figure 5.10, we can observe that the vanilla MCTS variants with more exploitation (lower C values) have deeper leaf nodes overall, as they can exploit the interesting regions more deeply. However, this does not hold for the deceptive functions (f_4 and f_5 at the fourth and fifth plots of Figure 5.10, respectively), where the vanilla MCTS variants with $C = 1$, $C = \sqrt{2}$ and $C = 2$ have deeper nodes in average than the vanilla MCTS with $C = 0.5$. The decrease in depth of the leaf nodes of MCTS $C = 0.5$ in f_4 and f_5 is a consequence of its lack of exploration, as it is prone to exploit any optima found with a lesser incentive to explore different regions than in the other functions. As f_4 and f_5 are deceptive functions, the optima found by MCTS $C = 0.5$ easily become the only regions of interest for the agent, resulting in a deep statistical tree around the optima but shallower everywhere else. On the other hand, the EA-based MCTS variants have shallower leaf nodes on every function, except f_1. They also feature a larger variance in the depth of their leaf nodes than the vanilla variants, which is a consequence of the volatility of the evolutionary process.

Based on the Central Limit Theorem (CLT), the leaf nodes' average depth metric can be assumed to follow a normal distribution for each function and MCTS variant. This assumption is grounded in the fact that leaf nodes' average depth represents an aggregation of averages obtained from independent distributions across 100 runs. We performed a two-sample t-test with unequal variances (Welch's t-test) on the leaf nodes' average depth for every MCTS variant pair and each function. The null hypothesis of the test is that the distributions of the leaf nodes' average depth for each variant are the same with an alpha value of 0.05. In other words, we are verifying if two variants constructed statistical trees with similar depths. We found that the EA-based variants had no statistically significant difference between them for any of the functions, allowing us to conclude that the inclusion of semantics is also not influencing the construction of the statistical tree, for the specific case of FOP. On the other hand, there is a significant difference between the leaf nodes' average depth of every other MCTS variant pair for every function except for MCTS $C = 0.5$ and MCTS $C = 1$ in f_2, and MCTS $C = 0.5$ and MCTS $C = \sqrt{2}$ in f_4. These observations show how even if the most visited node-based result is similar between two MCTS variants (Figure 5.9), their statistical trees can be significantly different (Figure 5.10).

5.6 Single-player Carcassonne experimental setup

The FOP experiments were designed to assess the effectiveness of EA-based MCTS variants in a simplified and controlled setting. While informative, these experiments do not fully capture the strategies available in more complex environments. To address this, we explored single-player Carcassonne variants, characterised by their variable branching factors and the unpredictability of optimal actions and rewards. The variants $Carc_{1,d}$ and $Carc_{3,d}$, which are the deterministic single-player Carcassonne variants with 1 and 3 initial meeples respectively, have long-term strategies based on their resource management aspect, while also having a constant tree depth and higher branching factors than FOP. On the other hand, the variants $Carc_{1,s}$ and $Carc_{3,s}$, which are the stochastic single-player Carcassonne variants with 1 and 3 initial meeples respectively, incorporate chance nodes,

Table 5.9: Expectimax parameters, named according to their maximum depth and heuristic function.

Name	**Description**
$Expectimax - 1 - s$	Maximum depth=1, Heuristic=score (s)
$Expectimax - 1 - vs$	Maximum depth=1, Heuristic=virtual score (vs)
$Expectimax - 2 - s$	Maximum depth=2, Heuristic=score (s)
$Expectimax - 2 - vs$	Maximum depth=2, Heuristic=virtual score (vs)

thus forming regular minimax game trees and elevating their complexity. We remind the reader that our deterministic variants of Carcassonne have a fixed sequence of tiles in the tile stack, which is shuffled at the beginning of each game, while the stochastic variants draw tiles at random from the tile stack on every turn.

This section outlines the experimental setup for the single-player Carcassonne variants. We included each variant previously detailed in Table 4.4. For consistency, the MCTS variants, including EA-MCTS and SIEA-MCTS, were configured with equal parameters to those used in the FOP experiments (see Tables 5.1 and 5.2). However, for this series of experiments we focused exclusively on vanilla MCTS variants using exploration constants $C = 0.5, \sqrt{2}$, and 3, to illustrate the highest contrast in strategies according to the exploration and exploitation balance. We also increased the iteration count for each MCTS variant to $10,000$ to potentially accentuate any particularity in their behaviour and strategies. To provide a comparative baseline, the MCTS variants were compared to the Expectimax algorithm, which is a well-known decision-making algorithm that strategically minimises the maximum possible loss by evaluating all potential outcomes, and the random agent, which selects actions uniformly at random. These algorithms have been tested with the base game of Carcassonne previously in [78], also more recently in our IEEE SSCI paper [5] and our IEEE Transactions in Games article [61].

5.6.1 Expectimax in Carcassonne

In our research, we use both Minimax and Expectimax (explained in Chapter 2) as benchmarks for the single-player Carcassonne variants, with the agents having access to game scores. The specifics of the agents' configurations are detailed in Table 5.9. Note that when applying Expectimax to the stochastic Carcassonne variants, the maximum depth parameter does not consider layers of chance nodes, and the algorithm behaves as Minimax.

Since the Expectimax agents, as detailed in Table 5.9, employ heuristic functions for state evaluation rather than the random uniform simulations used by MCTS, and possess differing stopping conditions, a direct comparison is not feasible. Nevertheless, they offer

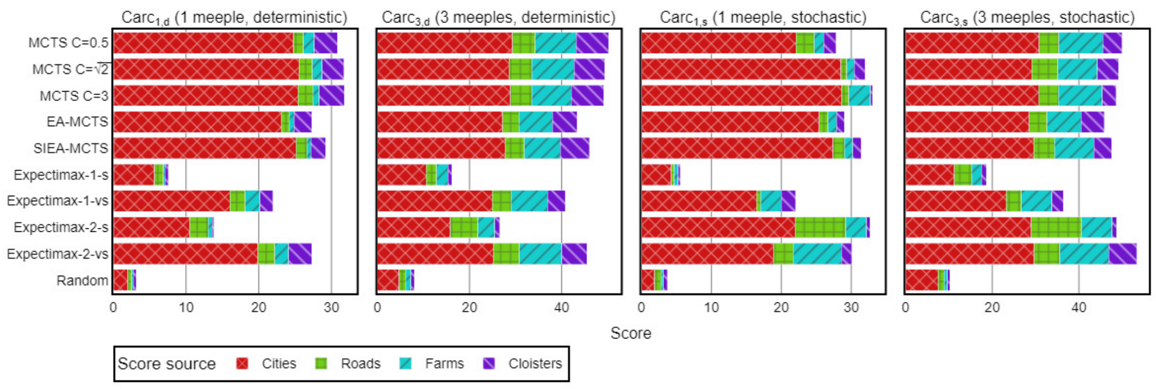

Figure 5.11: Breakdown of scores for every agent in each single-player Carcassonne variant.

a behavioural baseline, demonstrating the performance of agents that can foresee game scores one or two turns ahead in the single-player Carcassonne variants. We used both the scores and virtual scores from the game as heuristics for Expectimax to observe variations in agent behaviour when granted differing levels of game state information. The scores, integral to the game state, are always accessible to the player, reflecting the current state of play. Conversely, virtual scores, while also accessible at any moment, are not inherent to the game state and function as a “best-guess” heuristic.

5.7 Single-player Carcassonne Results

Figure 5.11 displays the final scores of each agent over 30 independent games for the single-player Carcassonne variants, with scores broken down by source: city, road, cloister, and farm. This breakdown aims to highlight strategic differences among the agents. The figure’s plots are arranged as follows, from left to right: deterministic with 1 meeple, deterministic with 3 meeples, stochastic with 1 meeple, and stochastic with 3 meeples.

In terms of total scores, no single agent consistently outperforms the others across all variants. The large score gap between the random agent and the rest underscores the strategic depth of Carcassonne. Regarding the vanilla MCTS variants, the one with the highest exploration factor, MCTS $C = 3$, excelled in the game variants with 1 initial meeple (first and third plots, from left to right, in Figure 5.11), whereas the less exploratory vanilla variant, MCTS $C = 0.5$, had the best scores in the game variants with 3 initial meeples (second and fourth plots, from left to right, in Figure 5.11).

The EA-based MCTS variants performed marginally worse than the vanilla MCTS on almost every game, with the exception of $Carc_{1,s}$, where MCTS $C = 0.5$ scored lower. Notably, SIEA-MCTS consistently achieved higher scores than EA-MCTS, though the differences were not statistically significant in any game variant when assessed using a student's t-test with an alpha value of 0.05.

Focusing on the deterministic Carcassonne variants (two left-most plots of Figure 5.11), it is apparent that the Expectimax agents scored lower than any vanilla MCTS agents. Among them, $Expectimax - 2 - s$, using the game score as its heuristic and a maximum depth of 2, outperformed EA-MCTS. Interestingly, $Expectimax - 2 - s$, despite having a deeper search, scored less than $Expectimax - 1 - vs$, which operates at a maximum depth of 1 but employs the virtual scores heuristic. This highlights the effectiveness of the virtual score in enabling agents to more accurately anticipate game outcomes. For instance, $Expectimax - 1 - s$ and $Expectimax - 2 - s$ cannot always predict the value of the farms, as farms are only scored at the end of the game. This is further confirmed by $Expectimax - 2 - s$'s score percentage coming from farms being 3.6% and 13.5% in $Carc_{1,d}$ and $Carc_{3,d}$, respectively, whereas $Expectimax - 2 - vs$'s score percentage coming from farms are 6.95% and 20.21% in the same game variants.

In contrast, in the stochastic variants (two right-most plots of Figure 5.11), the Expectimax agents with a maximum depth of 2 consistently outperformed those with a maximum depth of 1. $Expectimax - 2 - s$ scored higher than $Expectimax - 2 - vs$ in almost every game variant except for $Carc_{1,s}$. A difference in their playing styles is evident in the scores coming from roads. $Expectimax - 2 - s$ has 22.32% and 23.9% of its points coming from roads, which is the highest among all agents. $Expectimax - 2 - vs$, with a greater focus on farms, outscored all agents in $Carc_{3,s}$.

A key takeaway from Figure 5.11 is the dominance of cities as a scoring source for all agents across all game variants. Even the random agent's score is mostly composed of city points, ranging from 49.1% to 73% of its total score across all the game variants. This indicates cities' efficiency and reliability for point accumulation are superior to the rest of the features in the game of Carcassonne.

The increased availability of meeples in variants with 3 initial meeples ($Carc_{3,d}$ and $Carc_{3,d}$ at the second and fourth plots from left to right, respectively) facilitated a broader distribution of points across different features. This is shown by the clear increase in score percentage coming from features that are not cities in those plots. For instance, all the MCTS variants had 1.71% to 9.37% of their scores coming from farms in the variants with 1 initial meeples ($Carc_{1,d}$ and $Carc_{1,s}$ at the first and third plots from left to right, respectively), whereas in the variants with 3 initial meeples, the same agents had 16.59% to 20.24% of their scores coming from farms. Figure 5.12 shows the total number of meeples played on farms by each agent on average for every turn of the game. Note that the total meeples played can exceed the initial allocation of meeples because meeples can be recycled when features are completed.

Figure 5.12 highlights a surprising trend: $Expectimax - 1 - s$, which employs the scores as its heuristic function, consistently played the highest number of meeples on

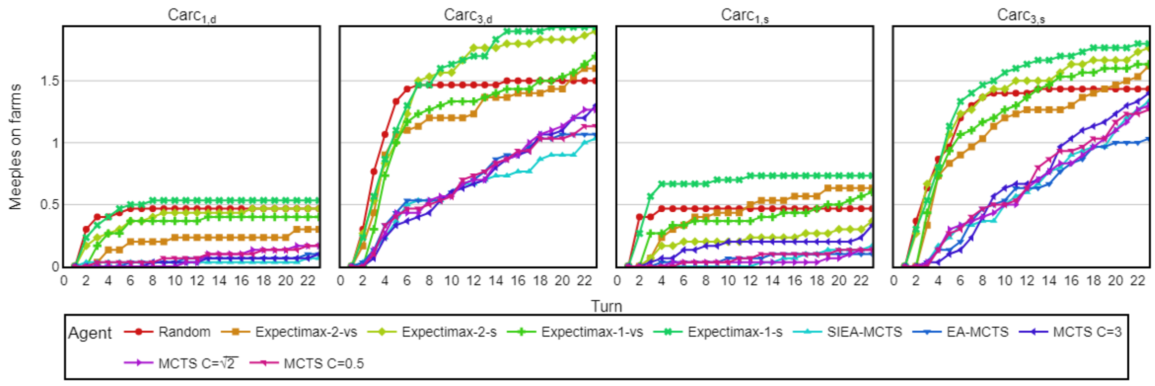

Figure 5.12: Total number of meeples placed on farms by each MCTS variant up until the turn number indicated on the x-axis.

farms, despite lacking a mechanism to evaluate farm value. This outcome is attributed to its tendency to select a random action when presented with a tile that cannot complete a feature, as these tiles do not offer immediate rewards. In Carcassonne, actions that involve placing meeples on farms are more widely available than any other feature, making it likely for a random choice to make them. The pattern exhibited by $Expectimax{-}1{-}s$ diminishes significantly when its maximum depth is increased to 2 as rewards can be found more easily, evidenced by the lower average of meeples on farms for $Expectimax - 2 - s$.

Figure 5.12 also shows that the MCTS variants generally used fewer meeples on farms compared to other algorithms. This suggests that MCTS, by using full random playouts as evaluations, can find different insights into state value compared to Expectimax, and yield a different strategy. The EA-based MCTS variants mirrored the vanilla MCTS variants in terms of meeple placement on farms. Remarkably, despite placing fewer meeples on farms, the MCTS variants achieved comparable farm scores to the Expectimax algorithms. In other words, the MCTS variants efficiency for meeples on farms was greater. The difference in meeple placement on farms between MCTS variants and the Expectimax algorithm is especially notorious in $Carc_{3,d}$ and $Carc_{3,s}$, the game versions with 3 initial meeples (second and fourth plots, from left to right in Figure 5.12).

Figure 5.13 reveals each agent's total meeple usage per game. It shows that while MCTS variants exhibited distinct farming patterns, their overall meeple usage across all features was similar to the Expectimax algorithms. The agents $Expectimax - 2 - s$ and random stand out for their unique meeple usage. The random agent's limited ability to complete features led to a consistent meeple count near its initial allocation (1 for $Carc_{1,d}$ and $Carc_{1,s}$, and 3 for $Carc_{3,d}$ and $Carc_{3,s}$). In contrast, $Expectimax - 2 - s$ displayed the most meeple reuse, caused by its heuristic evaluation (scores) which can only increase on feature completion. Surprisingly, $Expectimax - 1 - s$, with the same heuristic as

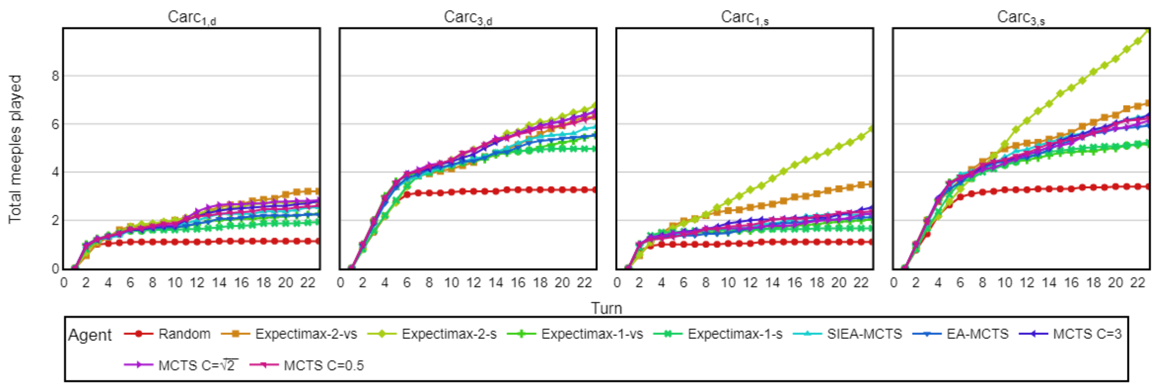

Figure 5.13: Total number of meeples played by each MCTS variant up until the turn number indicated on the x-axis.

$Expectimax - 2 - s$, was far less effective in reusing meeples. This suggests the existence of valuable two-turn strategies in the game of Carcassonne that $Expectimax - 1 - s$ failed to anticipate and $Expectimax - 2 - s$ is able to exploit.

Figure 5.14 illustrates the visit count for the most visited node by each MCTS variant per game turn. The vanilla MCTS variants have a lower visit count than any of the EA-based agents in general. Among the vanilla MCTS variants, the visit count seems to increase as the C parameter increases. For instance, the MCTS $C = 3$ shows the lowest visit count on average whereas MCTS $C = 0.5$ shows the highest. This is expected, as the higher the C parameter, the more the agent will explore alternative options.

The EA-based MCTS variants allocated a significant portion of their iterations to repeatedly exploring the same node, with an average visit count per turn of 77.68% to 93.34% across all game variants from turns 1 to 21. In contrast, vanilla MCTS variants concentrated only 7.07% to 40.42% of their iterations on the most visited node. Despite their narrower exploration, the EA-based MCTS variants achieved scores comparable to the vanilla ones (refer to Figure 5.11). This suggests that the EA-based MCTS variants were able to find a strategy that was as effective as the vanilla MCTS variants, despite their limited exploration.

Table 5.10 presents a summary of the outcomes from the single-player Carcassonne experiments. The first column identifies the specific Carcassonne variant being examined, and the second column lists the agent used in that variant. The third column, labelled "score", displays the agent's average score over 30 game runs. The fourth column, titled "meeple efficiency" describes the average points each played meeple scored for the player. Note that a high meeple availability is not necessarily always desirable. The fifth column is the "meeple availability" illustrates the proportion of turns during which the agent had at least one meeple available for play. Meeple availability reflects the agent's ability

Table 5.10: Scores and meeple usage statistics for each Carcassone single-player variant. The best MCTS and non-MCTS agent for each game variant are highlighted.

Game	Agent	Score	Meeple efficiency	Meeple availability
$Carc_{1,d}$	MCTS C=0.5	30.83 ± 9.38	11.86	0.122
	MCTS C=$\sqrt{2}$	31.7 ± 7.21	11.19	**0.128**
	MCTS C=3	$\mathbf{31.8 \pm 10.17}$	11.36	**0.128**
	EA-MCTS	27.27 ± 8.53	**12.21**	0.107
	SIEA-MCTS	29.23 ± 8.81	11.54	0.114
	$Expectimax - 1 - s$	7.53 ± 8.94	3.83	0.101
	$Expectimax - 1 - vs$	21.93 ± 11.72	**9.4**	0.101
	$Expectimax - 2 - s$	13.83 ± 11.78	4.94	0.171
	$Expectimax - 2 - vs$	$\mathbf{27.33 \pm 12.88}$	8.54	**0.226**
	Random	3.13 ± 3.79	2.76	0.059
$Carc_{3,d}$	MCTS C=0.5	$\mathbf{50.2 \pm 5.65}$	7.76	0.307
	MCTS C=$\sqrt{2}$	49.4 ± 7.85	7.48	0.310
	MCTS C=3	49.13 ± 6.48	7.48	**0.314**
	EA-MCTS	43.4 ± 5.46	**7.8**	0.271
	SIEA-MCTS	46.07 ± 8.18	7.72	0.296
	$Expectimax - 1 - s$	16.2 ± 7.32	3.26	0.288
	$Expectimax - 1 - vs$	40.77 ± 11.48	**7.24**	0.245
	$Expectimax - 2 - s$	26.53 ± 12.29	3.83	**0.438**
	$Expectimax - 2 - vs$	$\mathbf{45.5 \pm 10.51}$	7.04	0.358
	Random	8.07 ± 5	2.47	0.19
$Carc_{1,s}$	MCTS C=0.5	27.83 ± 7.5	11.6	0.106
	MCTS C=$\sqrt{2}$	31.97 ± 10.52	**13.9**	0.100
	MCTS C=3	$\mathbf{33.07 \pm 11.02}$	12.88	**0.114**
	EA-MCTS	29.03 ± 8.19	13.61	0.094
	SIEA-MCTS	31.43 ± 9.28	13.47	0.103
	$Expectimax - 1 - s$	5.53 ± 5.59	3.32	0.081
	$Expectimax - 1 - vs$	22.03 ± 13.5	**10.84**	0.088
	$Expectimax - 2 - s$	$\mathbf{32.7 \pm 18.47}$	5.25	**0.552**
	$Expectimax - 2 - vs$	30.07 ± 14.22	8.28	0.242
	Random	3.73 ± 4.26	3.39	0.064
$Carc_{3,s}$	MCTS C=0.5	$\mathbf{49.9 \pm 7.29}$	**7.84**	0.294
	MCTS C=$\sqrt{2}$	49.13 ± 5.75	7.8	0.288
	MCTS C=3	48.57 ± 6.92	7.59	0.294
	EA-MCTS	45.87 ± 5.73	7.56	0.286
	SIEA-MCTS	47.6 ± 7.5	7.32	**0.316**
	$Expectimax - 1 - s$	18.7 ± 7.76	3.53	0.290
	$Expectimax - 1 - vs$	36.43 ± 9.31	7.01	0.239
	$Expectimax - 2 - s$	48.67 ± 8.68	4.66	**0.812**
	$Expectimax - 2 - vs$	$\mathbf{53.33 \pm 11.2}$	**7.62**	0.404
	Random	10.27 ± 5.25	3.02	0.209

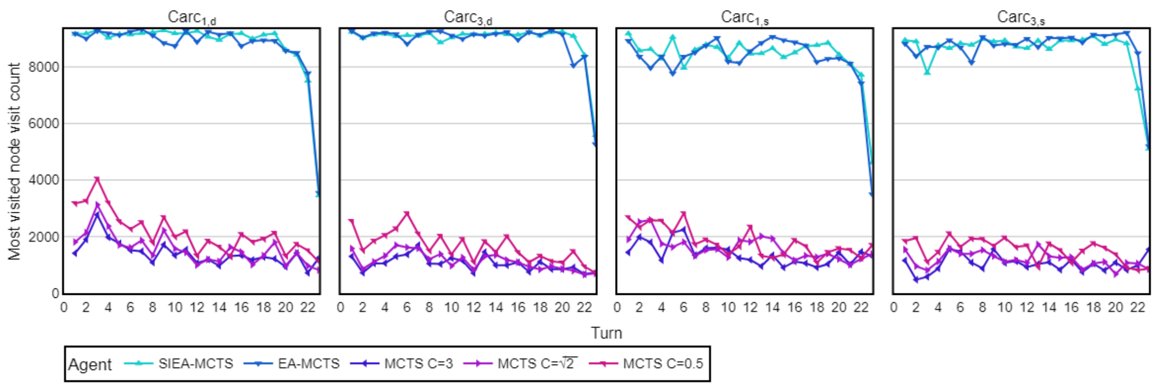

Figure 5.14: Visit count of the most visited node for each MCTS variant on the turn indicated by the x-axis.

to manage its resources effectively throughout the game, as having at least one meeple available allows the agent to capitalise on opportunities as they arise.

The highest scores are achieved by MCTS $C = 3$ in the game variants with 1 initial meeple, while MCTS $C = 0.5$ achieves the best scores in the game variants with 3 initial meeples. This suggests that exploration is crucial for the game variants with 1 initial meeple, whereas exploitation becomes more important for the game variants with 3 initial meeples. Table 5.10 clearly shows that a high availability of meeples is correlated with high scores for the game variants with 1 initial meeple ($Carc_{1,d}$ and $Carc_{1,s}$), as the algorithms with the highest scores are the ones with the highest meeple availability. MCTS $C = 3$ has the highest meeple availability in every game variant, demonstrating that it is the most efficient at managing its resources, especially regarding the last meeple it has available. MCTS $C = 3$'s high meeple availability comes from its likelihood to invest resources in actions that appear suboptimal at first, enabling it to uncover multi-turn opportunities that other vanilla MCTS variants overlook. On the other hand, EA-MCTS has the highest meeple efficiency among the MCTS agents in 3 out of the 4 game variants, implying it is the most efficient in making the most points out of the meeples it uses, without focusing on recycling them. Despite this efficiency, EA-MCTS's strategy leads to the lowest scores among the MCTS variants. SIEA-MCTS has better scores than EA-MCTS in all game variants, also with a greater standard deviation, implying a positive impact of the use of semantics. However, it is only in the stochastic game variant with 1 initial meeple that both EA-MCTS and SIEA-MCTS have success with their strategies comparable to MCTS $C = 0.5$ and MCTS $C = \sqrt{2}$.

5.8 Analysis of the evolved selection policies

UCB1, shown again Equation 5.2 for convenience, is widely used due to its desirable properties. It guarantees that all actions will eventually be explored, bounds the regret at any given time, and ensures convergence to the optimal action [10]. Exploitation (on the right-hand side of the equation) and exploration (on the left-hand side of the equation) are both crucial in maintaining these properties.

$$UCB1_i = Q_i + C\sqrt{\frac{2 \cdot \ln(N)}{n_i}} \tag{5.2}$$

Equation 5.2, shows the UCB1 formula for arm i. In it, Q_i is the average reward of arm i and n_i is its number of tries. N is the total number of tries among all the arms. C is the exploration and exploitation balance constant. The desirable properties of the UCB1 formula can be easily modified by the mutation operator of the GP used in EA-MCTS and SIEA-MCTS, explained previously in Chapter 2. The search space is extensive, and the chance of producing useful formulae with random alterations, such as the ones performed by the mutation operator, is low. Hence, the evolutionary process in EA-MCTS and SIEA-MCTS as proposed in this chapter struggled to consistently discover formulae that balance exploration and exploitation.

Any tree policy requires the presence of at least n_i and Q_i to make informed choices, as stated by [33]. If either of them is absent, or if one is so dominant that it can account for most of the variance in the formula thereby rendering the other terminal almost inconsequential, the formula might not exhibit balanced exploration and exploitation. When exploration is entirely lost, the tree policy becomes stuck when searching the tree, repeatedly trying the same action and focusing on the same branch of the tree without any reason to switch to a different one.

The appearance rate of each terminal in the syntax trees of the evolved formulae across all the experiments in this chapter is provided in Table 5.11. The column "rate of n" indicates that only 49% to 73% of the final evolved formulae contained at least one n as a terminal, which is a critical piece of data to perform exploration, highlighting the difficulty of evolving a balanced tree policy formulae. On the other hand, Q is the most frequently used terminal overall, with an 88% to 99% presence among all the MCTS variants and domains, well above those of N (49% to 67%) and n (49% to 73%). This is expected since the fitness evaluation of the GP used in the EA-based MCTS variants favours greedy formulae that exploit rather than those that explore. Additionally, the number of fitness iterations used to evaluate the fitness of each formula is not enough to effectively illustrate the benefits of exploratory formulae. Notably, none of the terminals were present in the totality of the formulae, which was a key factor contributing to

Table 5.11: Analysis of the formulae evolved by EA-MCTS and SIEA-MCTS for all the functions in FOP and all the single-player Carcassonne variants. The columns with a rate show the percentage of final evolved formulae that contained at least one terminal of each n, N, and Q respectively.

Problem	**Rate of n**		**Rate of N**		**Rate of Q**		**Syntax tree nodes**	
	EA	**SIEA**	**EA**	**SIEA**	**EA**	**SIEA**	**EA**	**SIEA**
FOP f_1	0.49	0.5	0.52	0.52	0.95	0.95	12 ± 5.7	11.92 ± 5.5
FOP f_2	0.55	0.63	0.53	0.49	0.93	0.93	13.2 ± 7.4	12.81 ± 8.2
FOP f_3	0.73	0.72	0.6	0.59	0.91	0.88	13.69 ± 6.8	14.69 ± 7.5
FOP f_4	0.61	0.65	0.6	0.52	0.93	0.94	14.24 ± 7.1	13.07 ± 7.3
FOP f_5	0.58	0.59	0.67	0.61	0.99	0.99	14.32 ± 7.5	13.86 ± 6.9
$Carc_{1,d}$	0.59	0.53	0.62	0.63	0.94	0.93	14.26 ± 8.41	14.05 ± 7.96
$Carc_{1,s}$	0.58	0.57	0.64	0.64	0.88	0.89	13.23 ± 8.11	13.74 ± 8.06
$Carc_{3,d}$	0.57	0.55	0.63	0.61	0.94	0.94	14.38 ± 8.12	14.15 ± 8.13
$Carc_{3,s}$	0.61	0.55	0.61	0.61	0.89	0.89	13.59 ± 8.15	13.26 ± 7.89

evolution failure.

The last column shows the number of nodes of the formula when represented as a syntax tree by the GP. Its average is comparable to that of the original UCB1 formula (13 nodes). This indicates that the proposed GP does not encourage bloating. On the other hand, its high variance implies that the evolutionary process has considerably varied results in terms of the size of the evolved formulae.

There are significant differences in the columns "rate of Q" and "Syntax tree nodes" between Carcassonne's deterministic variants ($Carc_{1,d}$ and $Carc_{3,d}$) and its stochastic variants ($Carc_{1,s}$ and $Carc_{3,s}$). Both EA-MCTS and SIEA-MCTS had a rate of Q of 0.93 0.94 for the deterministic Carcassonne variants, whereas the rate of Q for the stochastic variants was 0.88 0.89. This suggests that the impact of dropping Q is smaller in the stochastic Carcassonne variants, likely due to their decreased reward certainty. This is further supported by f_3, which is the FOP function with the most volatile rewards, having the smallest rate of Q (0.88 0.91) among all the FOP functions which range from 0.91 to 0.99. Similarly, the average number of syntax tree nodes for the deterministic variants ranged from 14.05 to 14.38, whereas the average number of syntax tree nodes for the stochastic variants ranged from 13.23 to 13.74.

5.8.1 Summary of findings

Function $f1$ had a single global optimum, where all MCTS variants consistently achieved maximum scores in every run. This function helped us demonstrate that the EA-based MCTS variants can evolve tree policies as effective as the vanilla MCTS variants when the reward landscape is simple and smooth, with the major difference being that the

EA-based MCTS variants showed less exploration than any of the vanilla variants.

Function $f2$ features multiple local optima and a single global optimum. The EA-based MCTS variants performed similarly to the vanilla MCTS variants with low C, getting stuck in local optima and failing to consistently find the global optimum. In this function we observe that, in sharp contrast to vanilla MCTS, the EA-based MCTS variants sometimes explore the regions of the tree with the lowest rewards. This hints at the evolutionary process promoting exploration of the tree by simply evaluating the formulae in the population. This phenomenon, particular of our algorithms, is not necessarily undesired and has potential to be manipulated with further research.

Function $f3$ offered multiple global optima and a rugged fitness landscape. In this function, EA-MCTS and SIEA-MCTS consistently sampled different optima on each run, searching deeper than the vanilla MCTS variants. This deep exploitation was proven desirable to a certain extent in this type of landscape, as the EA-based MCTS variants achieved higher scores in this function than some of the vanilla MCTS variants.

The functions $f4$ and $f5$ are deceptive to different extents. Specifically, the branches of their trees initially offering the highest rewards at first do not lead to optimal results. Our analysis demonstrated how vanilla MCTS agents require more exploration as the deception in the reward landscape of the problem increases because exploration is needed to avoid getting stuck in local optima. The EA-based MCTS variants demonstrated a likelihood of focusing on the first few local or global optima they found because their evolutionary process had no incentive to explore other branches of the tree once a branch was explored sufficiently to yield good rewards. This indicates that the exploration offered by the evolutionary process was insufficient to find the global optima in these functions, and requires further development. However, the EA-based MCTS variants were able to find the global optimum in some of the runs.

In the single-player Carcassonne experiments, the EA-based MCTS variants achieved scores comparable to the vanilla MCTS variants. Moreover, in the case with the most limited resources, EA-MCTS and SIEA-MCTS score higher than the vanilla MCTS variant with the most suboptimal C constant. The EA-based MCTS variants were able to find long-term strategies by methodically employing their limited game resources, in sharp contrast to greedy algorithms like the Expectimax agents. Although the strategic behaviour was similar between the vanilla and the EA-based MCTS variants, their statistical trees presented major structural differences, indicating that the EA-based MCTS variants had a distinctive approach to searching the tree.

Overall, EA-MCTS and SIEA-MCTS are volatile, evolving primarily exploitative tree policies, with exploration primarily driven by the evolutionary process itself. They offer adaptable solutions that vary from attempt to attempt and have the potential to outperform the vanilla MCTS algorithm with a suboptimal C constant. Additionally, the resulting evolved tree policy serves as an additional output of the search process. The tree policies evolved by their GP occasionally omitted informative terminals, revealing the significance of these terminals in the problem, despite this being an undesirable phenomenon. The number of nodes in the syntax tree of the evolved formulae is comparable to that of the original UCB1 formula, suggesting that the proposed GP does not promote excessive growth. The high variance in the number of nodes of the evolved formulae suggests that the evolutionary process has considerably varied results in terms of the size of the evolved formulae.

6 Evolutionary MCTS in the base game of Carcassonne

Related publication to this chapter: Edgar Galván, Gavin Simpson, and Fred Valdez Ameneyro. “Evolving the MCTS Upper Confidence Bounds for Trees Using a Semantic-inspired Evolutionary Algorithm in the Game of Carcassonne”. In: *IEEE Transactions on Games* (2022).

6.1 Introduction

In this chapter, we analyse applying MCTS, as well as our proposed Evolutionary Algorithm (EA)-based Monte Carlo Tree Search (MCTS) agents in the game of Carcassonne for two players, jumping from the synthetic Function Optimisation Problem (FOP) to more of a real-world problem. To do so, we executed a series of matches between them and other state-of-the-art Carcassonne agents. First, we analyse the performance of vanilla MCTS in the base game of Carcassonne in Section 6.2. We face multiple vanilla MCTS variants with different C constants against a random uniform agent to compare their performance. We then determine the best vanilla MCTS variant with a round-robin tournament discussed in Section 6.3. Next, we proceed to confront our EA-based MCTS variants, EA-MCTS and SIEA-MCTS, against two of the best vanilla MCTS variants, the best Expectimax agents, and a random uniform agent in a final round-robin tournament presented in Section 6.4.

6.2 Performance of vanilla Monte Carlo Tree Search in the base game of Carcassonne

In two-player games, the reward Q used for the Upper Confidence Bounds (UCB1) formula is normally defined as 1 for a win, 0 for a draw, and -1 for a loss. We will refer to this

Table 6.1: MCTS parameters for reward comparison

Parameter	**Value**
Iterations, Rollouts	$5,000$, 1
UCB1's C constant	Random float in $[0.5, 3]$
Reward Q (R1)	Win:1, Draw:0, Loss:-1
Reward Q (R2)	Normalised difference of final scores
Normalisation factor	593

Table 6.2: Results of 100 games between two vanilla MCTS agents with $R1$ and $R2$ as their reward types.

Reward type	**As Player 1**			**As Player 2**		
	Wins	**Draws**	**Losses**	**Wins**	**Draws**	**Losses**
$R1$	2	0	48	4	1	45
$R2$	45	1	4	48	2	0

reward as reward type 1 ($R1$). However, the normalised difference of final scores, referred to as reward type 2 ($R2$), is more informative as a reward for MCTS in the game of Carcassonne as discussed in Chapter 4. To back this claim, we performed a series of games between two vanilla MCTS agents, each using the reward types $R1$ and $R2$, respectively. The difference of final scores in $R2$ is normalised with our maximum score presented in Chapter 4, which equals 593 for the base game of Carcassonne. The parameters for those games are presented in Table 6.1.

We compared the performance of MCTS with $R1$ and $R2$ in 100 games, where each agent played 50 games as Player 1 and then 50 as Player 2. The constant C for every pair of games with swapped player order was chosen at random from the range $[0.5, 3]$, equal for both agents. The results of this experiment are displayed in Table 6.2. The MCTS agent with $R2$ won 93 of the games (45 as Player 1 and 48 as Player 2), while MCTS with $R1$ won 6 (2 as Player 1, 4 as Player 2), with only 1 drawn game. The results of this experiment reveal that there is a clear advantage in using the normalised difference of final scores as the reward for MCTS in the game of Carcassonne, and is the one adopted for the rest of the experiments presented in this chapter. The superiority of $R2$ over $R1$ is because $R2$ gives additional information that helps MCTS to increase its advantage when winning, or decrease the gap when losing, where $R1$ would give similar rewards to multiple alternatives.

The performance of MCTS is also greatly affected by the constant C from the UCB1 formula. In Chapter 5, we learned that the constant $C = 0.5$ achieved the highest scores among the vanilla MCTS variants for the stochastic single-player versions of Carcassonne, $Carc_{1,s}$ and $Carc_{3,s}$, which are direct simplifications of the base game used in this chapter. Here, we explore the impact of the C constant in the full base game of Carcassonne for

Table 6.3: MCTS's average scores against a random uniform agent. The difference of final scores (DFS) and proportion of final scores (PFS) are calculated from MCTS's perspective

Agent	DFS	PFS	Score	Meeples played
MCTS as Player 1				
$C = 0.5$	97.05 ± 4.8	7.07 ± 2.9	113.05 ± 17.2	16.2 ± 2.5
$C = 1$	85.75 ± 4.1	8.9 ± 4.1	96.6 ± 11.9	14.3 ± 2.2
$C = \sqrt{2}$	84.05 ± 4.4	7.07 ± 3.3	97.9 ± 13.1	14.75 ± 2.4
$C = 2$	75.15 ± 5	5.83 ± 2.7	90.7 ± 18.3	13.35 ± 2.3
$C = 3$	71.35 ± 4.3	5.66 ± 1.8	86.65 ± 14.6	12.95 ± 1.8
MCTS as Player 2				
$C = 0.5$	90.8 ± 4.8	6.99 ± 3.2	105.95 ± 16.1	14.05 ± 1.8
$C = 1$	86.4 ± 4.7	7.31 ± 3.1	100.1 ± 16.5	14.5 ± 2.5
$C = \sqrt{2}$	79.9 ± 4.5	6.85 ± 2.5	93.55 ± 15.5	14 ± 2.4
$C = 2$	71.9 ± 4.8	5.57 ± 2.8	87.65 ± 15.2	13 ± 2.5
$C = 3$	67.15 ± 4.9	5.9 ± 3.6	80.85 ± 16.6	12.45 ± 1.7

two players. The parameters used for this experiment are the same as $MCTS_{R2}$, as seen in Table 6.1, but with $C \in \{0.5, 1, \sqrt{2}, 2, 3\}$. We compared the performance of vanilla MCTS when facing a random uniform agent in matches of 20 games as Player 1 and 20 games as Player 2 each. The results of this experiment are presented in Table 6.3.

Table 6.3's difference of final scores (DFS) column shows the difference of final scores between MCTS and the random agent, from MCTS's perspective. As can be seen, the largest positive difference of final scores is attained by MCTS $C = 0.5$, both when playing as Player 1 or as Player 2. This difference of final scores decreases as C increases, a trend persistent regardless of the player order. Similarly, MCTS's final scores, found in the fourth column from left to right in Table 6.3, are larger for smaller C values, regardless of the player order. These results are consistent with the results from the single-player versions of Carcassonne in Chapters 4 and 5. The column proportion of final scores (PFS) shows the proportion of MCTS's final score to the random agent's final score, relevant in the game of Carcassonne given that a plausible strategy is to block scoring opportunities for the opposing player instead of trying to score the most points for oneself. The column meeples played shows the total count of meeples played by MCTS, reflecting the ability of MCTS to recycle the limited resources available in the game of Carcassonne. The highest average meeples played are achieved by MCTS $C = 0.5$ when playing as Player 1, and the lowest by MCTS $C = 3$. However, the MCTS variant with the most meeples played as Player 2 is MCTS $C = 1$. Note that while the total count of meeples played is correlated to the final score, it is not a direct measure of the agent's performance in this version of the game. While MCTS $C = 0.5$ has the largest scores overall, MCTS $C = 1$ has the best proportion of final scores, regardless of the player order. Moreover, MCTS $C = 1$ outstands as the only MCTS variant that achieves larger scores, difference of final scores, and total played meeples as Player 2 than it does as Player 1. This suggests that MCTS $C = 1$ has a playing style that can take better advantage of playing as Player 2, but not enough to outperform MCTS $C = 0.5$.

By using a two-sided Mann-Whitney U test with $alpha = 0.05$, MCTS's final scores as Player 1 were found significantly different to its final scores as Player 2, except for MCTS $C = 2$. This confirms that the game of Carcassonne gives an advantage to Player 1. Table 6.3 also shows that MCTS $C = 0.5$ is the best variant when playing as Player 1 and Player 2, proving 0.5 to be the best C constant from the ones we tested, for the base game of Carcassonne and versus a random uniform player.

6.3 Round-robin tournament between vanilla MCTS variants in the base game of Carcassonne

The games versus the random uniform agent, although informative, are not enough to claim that MCTS $C = 0.5$ is the best among the vanilla MCTS variants. To complement the results from the previous section, we performed a round-robin tournament between them. Given that Player 1 has a slight advantage in the game of Carcassonne, when we compare two agents in a direct confrontation, each must get an equal number of games as Player 1 and as Player 2. For the experiments in this chapter, we define a match as a fixed number of games between two players, meaning that we require 2 matches to evenly confront a pair of agents switching the player order. This is analogous to most major sports leagues, with players/teams playing each other twice.

The parameters used for the round-robin tournament are the same as in Table 6.1, using the reward type $R2$, but with $C \in \{0.5, 1, \sqrt{2}, 2, 3\}$. Each match was composed of 10 games for each agent as Player 1 and as Player 2. The same 10 random seeds are used in each game of every match, to compare the agents under equivalent circumstances. The results of each individual match in the round-robin tournament are presented in Table 6.4, where the player on the left is Player 1 in the match against the player on the top as Player 2. Each cell of Table 6.4 displays the count of wins, draws, and loses, as well as the average difference of final scores (final score of Player 1 - final score of Player 2) with its standard deviation in each individual match.

We can observe that MCTS $C = 0.5$ wins all of its matches when playing as Player 1. Furthermore, it wins every single game in those matches except for a single drawn game against MCTS $C = 3$. On the other hand, MCTS $C = 0.5$ did not win every match as Player 2, as it was defeated by MCTS $C = 1$ in their match. That match, however, was very close, with 5 wins for MCTS $C = 1$, 1 draw, and 4 wins for MCTS $C = 0.5$. The

Table 6.4: Carcassonne's vanilla MCTS round-robin matches. Each cell displays Player 1's wins/draws/losses, as well as the average difference of final scores from Player 1's perspective.

		Player 2				
		$C = 0.5$	$C = 1$	$C = \sqrt{2}$	$C = 2$	$C = 3$
Player 1	$C = 0.5$	-	10/0/0 20.1 ± 15.8	10/0/0 24.8 ± 19.8	10/0/0 29.6 ± 24.9	9/1/0 25.5 ± 16.2
	$C = 1$	5/1/4 2.2 ± 18.0	-	9/0/1 13.9 ± 18.1	5/0/5 4.8 ± 15.3	8/1/1 13.3 ± 14.3
	$C = \sqrt{2}$	2/0/8 -13.2 ± 15.0	3/0/7 -6.0 ± 15.5	-	5/0/5 5.9 ± 21.1	7/0/3 8.6 ± 17.2
	$C = 2$	0/0/10 -24.6 ± 20.1	5/0/5 -1.0 ± 11.2	4/0/6 -2.2 ± 16.5	-	7/1/2 9.8 ± 12.3
	$C = 3$	1/0/9 -22.7 ± 19.1	3/0/7 -5.3 ± 23.3	2/0/8 -13.7 ± 21.2	4/1/5 -0.3 ± 14.4	-

Table 6.5: League points (LP) awarded to each agent after a Carcassonne match.

Name	**Description**	**LP**
Bonus win points (BWP)	Games won to games lost ratio ≥ 3	1
Bonus loss points (BLP)	If lost, games lost to games won ratio ≤ 1.5	1
Win(W), Draw(D), Loss(L)	Match won, drawn, or lost	4, 2, 0

evenness of the match can also be appreciated in their relatively low difference of final scores, which was 2.2 ± 18 on average. To perform a quantitative analysis of how each agent compared to each other in their matches and in the tournament overall, we propose to use a cumulative scoring system across all their matches. Table 6.5 describes the league points (LP) we are awarding each agent after a Carcassonne match.

We awarded 4 points for a win, 2 for a draw, and 0 for a loss, as shown in Table 6.5. Additionally, we awarded bonus win points (BWP) to the winning player when it clearly outperforms its opponent, and bonus lost points (BWP) to the losing agent when the match is near to even. The summary of the results of the round-robin tournament is presented in Table 6.6, where rankings are determined according to each agent's league points (column "LP"), using the cumulative difference of final scores across all the games (column "DFS") as a tie-breaker. The league points are calculated from the results of the matches in Table 6.4 using the scoring system in Table 6.5. The columns W, D, and L present the total number of matches won, drawn, and lost, respectively.

Table 6.6 shows that MCTS $C = 0.5$ is the best vanilla MCTS variant for the base game of Carcassonne, with a clear advantage over the other variants. MCTS $C = 1$ is the second best, with a significant difference in league points from the other variants. MCTS $C = \sqrt{2}$ and MCTS $C = 2$ are tied in the third position, with the same amount of league points but with MCTS $C = \sqrt{2}$ having a better difference of final scores. MCTS $C = 3$ is the worst variant, with the lowest league points and the worst difference of final scores.

Table 6.6: Carcassonne's vanilla MCTS round-robin tournament results.

Rank	Agent	LP	DFS	W	D	L	BWP	BLP
1	MCTS $C = 0.5$	38	158.3 ± 13.8	7	0	1	7	1
2	MCTS $C = 1$	28	26.4 ± 15.2	5	2	1	2	0
3	MCTS $C = \sqrt{2}$	15	-27.5 ± 22.7	3	1	4	1	0
4	MCTS $C = 2$	18	-58.0 ± 16.8	2	3	3	1	1
5	MCTS $C = 3$	5	-99.2 ± 9.8	0	0	8	0	1

We found that the value of Q of the UCB1 formula in the base game of Carcassonne for two players is relatively small compared to the single-player versions, given that in the multi-player setting the difference of final scores is used instead of the full score. This makes the reward Q comparatively small after normalisation, requiring a small C to compensate and find a good exploration versus exploitation balance.

6.4 EA-MCTS and SIEA-MCTS in the base game of Carcassonne

Vanilla MCTS's tree search alternates the priority of the moves to explore next according to the turn of the player making the choice, following the minimax principle. During the selection step of MCTS, if the move to be chosen is on the agent's turn, it chooses the action that gives the highest rewards. Conversely, if the move is on the opponent's turn, it chooses the action that gives the lowest rewards. This is typically done by alternating the sign of the exploitation part in the UCB1 formula, or its equivalent for any of the MCTS variants found in the literature [162]. However, the tree policy formulae evolved by EA-based MCTS and Semantically-Inspired Evolutionary Algorithm Monte Carlo Tree Search (SIEA-MCTS)-based MCTS agents are not ensured to have distinguishable exploration and exploitation parts. To alternate the priority of the actions according to the player taking the turn, Evolutionary Algorithm Monte Carlo Tree Search (EA-MCTS) and SIEA-MCTS alternate the sign of every instance of the Q terminal in the evolved formula accordingly.

Regarding handling stochastic events like the ones in the base game of Carcassonne, when vanilla MCTS traverses trees with chance nodes, it typically samples any random events it encounters and continues the search. When a random event is sampled it creates a chance node, which is not considered a new expansion. MCTS then adds a random node to the outcome of the random event in its expansion phase. When the reward is backpropagated through a chance node, it is weighted with the probability of the related random event. Because of this, in regular minimax trees, the selection step of the vanilla MCTS chooses actions according to the pondered rewards it has collected from its chance

Table 6.7: Parameters of the agents for the base game of Carcassonne

Parameter	**Value**
All MCTS variants	
Iterations, Rollouts	$5,000$, 1
Reward Q	Normalised difference of final scores
UCB1's C constant	0.5
Normalisation factor	593
EA-based MCTS variants	
(μ+λ)-ES	$\mu = 1, \lambda = 4$
Generations g, Fitness iterations S	$g = 20$, $S = 30$
Genetic operator	Subtree mutation (90% – 10% policy)
Mutation subtree generation method	Full (depth $\sim$ Uniform$(1, 3)$)
Initialisation Method	UCB1 formula + λ mutations
Maximum syntax tree depth	8
Total fitness iterations	$\lambda * g * S + S = 2,430$
SSi (SIEA-MCTS)	$L = 0.1, U = 0.5$
Expectimax variants	
Maximum depth	2
Heuristic ($Expectimax - 2 - s$)	Difference of scores
Heuristic ($Expectimax - 2 - vs$)	Difference of virtual scores

nodes. EA-MCTS and SIEA-MCTS behave analogously: the Q terminal is the pondered reward of the chance nodes. Thus, the evolved formulae are expected to have the same behaviour as vanilla MCTS when traversing chance nodes.

We now proceed to compare the EA-based MCTS variants with the rest of the agents. The parameters used for the round-robin tournament with the best agents of each type in the base game of Carcassonne are presented in Table 6.7. The base game of Carcassonne has a stochastic event at the beginning of every turn, meaning that it has a regular minimax tree [15] with a layer of chance nodes alternated with a layer of decision nodes (refer to Chapter 2). The maximum depth of the Expectimax agents, $Expectimax-2-s$ and $Expectimax-2-vs$, does not consider chance nodes for its depth calculation. In other words, Expectimax samples every potential random outcome in a chance node without increasing its internal depth counter. Under those circumstances, a depth of 2 is the maximum depth feasible for an exhaustive search within a reasonable time frame, even considering pruning techniques like the *-minimax group of algorithms. The *-minimax group of algorithms [15] with the virtual scores as their heuristic state evaluation are reported to be the previous state of the art for Carcassonne [78]. They use a heuristic move order and reward bounds to prune the tree, while a probing factor parameter determines the name of the Star algorithm. In this chapter, we use Expectimax without pruning in order to perform an exhaustive search within the established maximum depth, avoiding

Table 6.8: Carcassonne's round-robin matches. Each cell displays Player 1's wins/draws/losses, as well as the average difference of final scores from Player 1's perspective.

			Player 2						
			Vanilla MCTS		**EA-based MCTS**		**Minimax**		
			MCTS C=0.5	**MCTS C=1**	**EA-MCTS**	**SIEA-MCTS**	**Exp-2-s**	**Exp-2-vs**	**Random**
Player 1	**Vanilla**	**MCTS C=0.5**	-	13/0/7 6.75 ± 17.2	15/1/4 11.95 ± 23.4	15/0/5 15.05 ± 21.0	20/0/0 62.6 ± 19.7	17/0/3 27.65 ± 27.9	20/0/0 97.05 ± 18.2
		MCTS C=1	12/0/8 1.1 ± 22.3	-	12/0/8 6.7 ± 21.9	12/0/8 4.4 ± 23.0	20/0/0 52.4 ± 16.9	19/0/1 26.4 ± 22.1	20/0/0 85.75 ± 12.9
	EA-based	**EA-MCTS**	11/0/9 -3.1 ± 22.7	10/1/9 5.0 ± 18.2	-	12/1/7 -1.3 ± 22.5	20/0/0 64.55 ± 19.3	17/1/2 31.0 ± 21.4	20/0/0 86.95 ± 17.0
		SIEA-MCTS	7/1/12 -2.6 ± 20.1	6/0/14 -6.8 ± 20.0	11/1/8 3.35 ± 24.4	-	20/0/0 57.95 ± 21.6	16/0/4 16.15 ± 24.4	20/0/0 98.65 ± 17.7
	Minimax	**Exp-2-s**	0/0/20 -54.2 ± 18.1	0/0/20 -59.4 ± 22.1	0/0/20 -52.7 ± 19.8	0/0/20 -56.5 ± 25.7	-	0/0/20 -38.2 ± 25.3	20/0/0 33.5 ± 17.8
		Exp-2-vs	3/0/17 -29.3 ± 23.1	5/0/15 -20.7 ± 24.9	6/0/14 -15.6 ± 24.9	7/1/12 -12.3 ± 25.4	20/0/0 48.7 ± 23.8	-	20/0/0 71.8 ± 23.7
	Random		0/0/20 -90.8 ± 17.4	0/0/20 -86.4 ± 17.3	0/0/20 -85.0 ± 17.5	0/0/20 -84.1 ± 16.3	4/0/16 -23.1 ± 19.1	0/0/20 -64.2 ± 17.4	-

any dependency on the quality of the pruning parameters of the *-minimax algorithms.

The results of each individual match in the final round-robin tournament which includes MCTS $C = 0.5$, MCTS $C = 1$, EA-MCTS, SIEA-MCTS, $Expectimax - 2 - s$, $Expectimax - 2 - vs$ and a random agent are presented in Table 6.8. In its right-most column, corresponding to matches where the random agent is Player 2, we can observe that SIEA-MCTS is the agent with the largest positive final score difference. On the other hand, in the bottom-most column which corresponds to matches where the random agent is Player 1, MCTS C=0.5 is the agent with the largest positive final score difference. All the agents won all their games against the random agent, except for $Expectimax - 2 - s$, who lost 4 games as Player 2. This was unexpected, given that $Expectimax - 2 - s$ was the agent with the second highest average final score in $Carc_{1,s}$, the single-player Carcassonne variant with 1 initial meeple, and had superior results than the EA-based MCTS variants in $Carc_{3,s}$, the single-player Carcassonne variant with 3 initial meeples (refer to Chapter 5). We found that the greedy nature of $Expectimax - 2 - s$, which tries to maximise its immediate score within its maximum depth of 2, is particularly harmful in the base game of Carcassonne. $Expectimax - 2 - s$ plays its meeples on almost every move from the beginning of the game until it runs out of them, making the agent miss more scoring opportunities than in the single-player versions of the game because the base game has more turns and has an opposing player that can potentially complicate meeple recollection. Note that greedy gameplay is not necessarily always a bad strategy in Carcassonne. However, $Expectimax - 2 - s$'s extremely greedy nature is enough to make it lose against the random agent, showcasing the deceptive nature of the base game of Carcassonne.

MCTS $C = 0.5$ managed to win all of its matches as Player 1 and as Player 2, except for the match where it plays as Player 2 against EA-MCTS and MCTS $C = 1$. However,

Table 6.9: Carcassonne's round-robin tournament results.

Rank	Agent	LP	DFS	W	D	L	BWP	BLP
1	MCTS $C = 0.5$	50	400.0 ± 43.1	10	0	2	8	2
2	MCTS $C = 1$	47	338.35 ± 42.8	10	0	2	6	1
3	EA-MCTS	43	314.45 ± 69.8	9	0	3	5	2
4	SIEA-MCTS	34	301.55 ± 53	7	0	5	5	1
5	$Expectimax - 2 - vs$	20	43.85 ± 54.1	4	0	8	4	0
6	$Expectimax - 2 - s$	10	-490.8 ± 37.7	2	0	10	2	0
7	Random	0	-907.4 ± 45.4	0	0	12	0	0

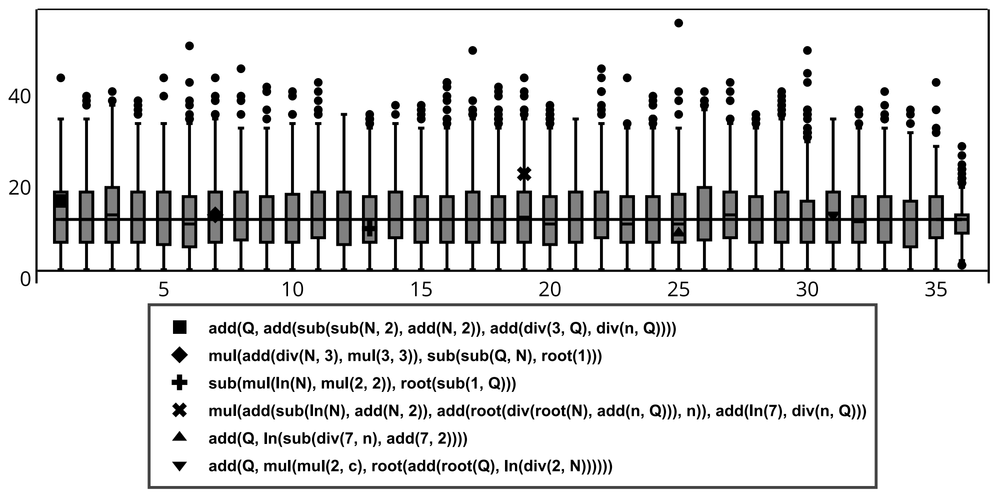

Figure 6.1: EA-MCTS's average number of nodes in the syntax tree (x-axis) of the evolved formulae per turn (y-axis). The horizontal solid line denotes the size of UCB1. Each marker denotes a random evolved formula taken during the game.

the matches were very close. For instance, the match with EA-MCTS had 11 wins for EA-MCTS and 9 wins for MCTS $C = 0.5$, with a difference of final scores of -3.1 ± 22.7 on average.

Note that there is no single instance of a match where an Expectimax variant wins against any of the MCTS variants, suggesting that MCTS is particularly well suited for the base game of Carcassonne. To quantitatively compare the agents in this round-robin tournament, Table 6.9 shows the rankings of the agents determined according to their league points (column "LP"), using the cumulative difference of final scores across all the games (column "DFS") as a tie-breaker. The league points are calculated from the results of the matches in Table 6.8 using the scoring system in Table 6.5.

Table 6.9 ranks MCTS $C = 0.5$ as the best agent for the base game of Carcassonne with 50 LP, followed by MCTS $C = 1$ with 47, which was then followed by the EA-based MCTS variants. EA-MCTS, with 43 LP, only won 2 more matches than SIEA-MCTS, who got 34

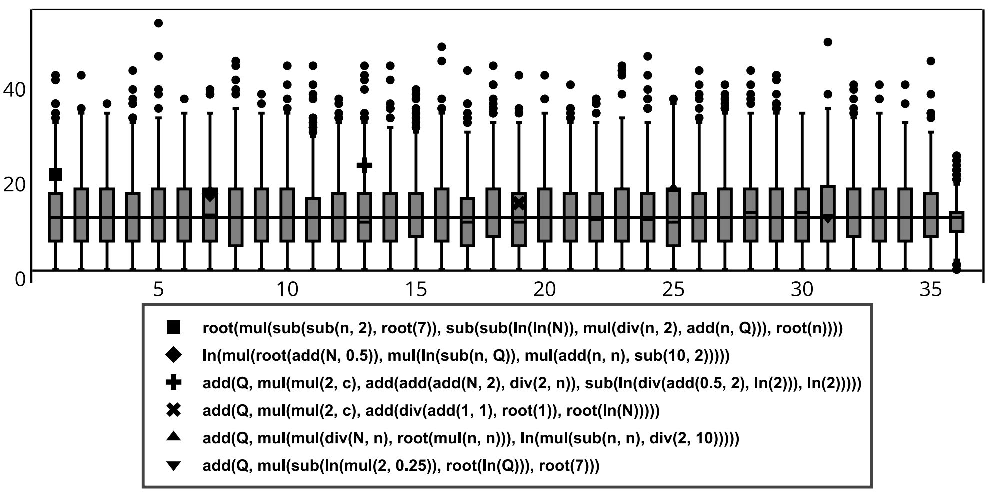

Figure 6.2: SIEA-MCTS's average number of nodes in the syntax tree (x-axis) of the evolved formulae per turn (y-axis). The horizontal solid line denotes the size of UCB1. Each marker denotes a random evolved formula taken during the game.

LP. Their DFS, however, is very similar, with 314.45 ± 69.8 for EA-MCTS and 301.55 ± 53 for SIEA-MCTS. Note that the DFS's standard deviation of EA-MCTS is the largest overall, setting it apart as the agent with the most volatile results. $Expectimax - 2 - vs$ was the only agent other than any MCTS variant to get a positive DFS, however, its LP are significantly lower than the EA-based MCTS variants. $Expectimax - 2 - s$ and the random agent are the worst agents in the round-robin tournament, with negative DFS.

We kept track of all the evolved formulae generated by our EA-based MCTS variants in the round-robin. Figures 6.1 and 6.1 show the number of nodes ($y - axis$) of these evolved formulae throughout each of the 36 turns ($x - axis$) in the game of Carcassonne for EA-MCTS and SIEA-MCTS, respectively. In them, the solid horizontal line denotes the size of the UCB1 formula, which is the initial formula of the evolutionary process. We can observe that 50% of the central data is around the size of the UCB1 expression (see first and third quartiles of the boxplots in both figures), regardless of the EA method used. This can go as small as 10 nodes up to around 20 nodes. Note how this varies as the game progresses. In particular, it is interesting to observe how at the end of the game (Turn 36, right-hand side in each figure), SIEA-MCTS and EA-MCTS tend to produce expressions of similar length compared to UCB1 or slightly shorter when the game is about to finish. These fluctuations in the sizes of the evolved UCB1 formula, along with the performance discussed in the previous section, indicate how our proposed EA-based MCTS variants adapt to different time frames within the game.

6.4.1 Summary of findings

This chapter begins by examining two reward systems for the full game of Carcassonne: R1 and R2. R1 assigns 1 for a win, 0 for a draw, and -1 for a loss, while R2 represents the normalised difference of final scores. We found that the latter (R2) provided significantly more information for MCTS, enabling the agent to pursue higher scores in winning or losing positions. As a result, it was utilized as the reward system for all MCTS variants in this chapter. In matches against a random agent, as well as in a round-robin tournament, we found that MCTS with lower C performed the best in the game of Carcassonne. A potential reason for requiring low C constants is the normalisation of the rewards which makes them relatively small.

Subsequently, we conducted a comparative analysis involving the top-performing MCTS variants, the EA-based MCTS variants, Expectimax variants, and a random agent in another round-robin tournament within the game of Carcassonne. Each agent competed against all others in matches where half were played as the first player and the remaining half as the second player. Our experiments resulted in MCTS with $C = 0.5$ as the top-performing agent, followed by MCTS with $C = 1$, EA-MCTS, and SIEA-MCTS. The EA-based MCTS variants achieved average scores nearly comparable to the best vanilla MCTS variants, with most games won by the vanilla MCTS variants, albeit by narrow margins. Moreover, the EA-based MCTS variants consistently defeated the expectimax-based agents. Our findings suggest that EA-MCTS and SIEA-MCTS show promise in Carcassonne gameplay but may require further refinement. Additionally, Carcassonne demonstrates promise as a valuable benchmark for the development and evaluation of tree search algorithms. Notably, EA-MCTS demonstrated slightly superior performance compared to SIEA-MCTS, implying that enhancements to the inclusion of semantics in the evolutionary process may be beneficial, requiring further investigation.

7 Conclusions

Our research contributes to the field of artificial intelligence and decision-making by developing and evaluating two novel variants that integrate EAs with MCTS, namely EA-MCTS and SIEA-MCTS. These approaches dynamically optimise the MCTS selection policy using Genetic Programming (GP). Our EA-based MCTS variants, through an online evolutionary approach, aim to adapt the selection policy to various problem characteristics. Through a series of experiments in domains ranging from multiple reward distributions in the Function Optimisation Problem (FOP) to different Carcassonne variants, we found that EA-MCTS and SIEA-MCTS have promising capabilities, working as stepping stones towards the online evolution of dynamic and context-aware tree policies.

7.1 Evolutionary Algorithms-inspired Monte Carlo Tree Search: Strengths

EA-MCTS and SIEA-MCTS have been shown to offer potential benefits in terms of adaptability and performance, with competitive results in the FOP and the game of Carcassonne. We identified the following strengths:

- EA-MCTS and SIEA-MCTS can be stopped at any time to yield a result, without the need of a fixed number of iterations to complete the evolutionary process.
- Dynamic evolution allows for adaptability that is sensitive to both the problem at hand and to the stages of the search process.
- The exploration of the tree can be orchestrated by the evolutionary process itself, rather than the exploration component of the evolved formulae. In other words, by evaluating suboptimal evolved formulae, the evolutionary process can potentially positively contribute to exploring regions of the tree that no robust tree policy formulae would.
- EA-MCTS and SIEA-MCTS offer the potential to discover alternative selection policies that are not based on the UCB1 formula.

7.2 Evolutionary Algorithms-inspired Monte Carlo Tree Search: Challenges

These strengths are a significant advantage over other evolutionary algorithms, as they allow the method to be used in real-time decision-making applications without major drawbacks. We found that the following challenges need to be addressed for effective incorporation of EAs, and specifically GP, into MCTS for online decision-making

- The search space is vast and flooded with trivial solutions, which makes it difficult to find useful formulae through random mutations.
- Comparing selection policies is far from trivial, even more so with limited resources.
- There is a trade-off between adaptability and robustness. If more resources are invested in the evaluation of each evolved formula, the evolutionary process will be more robust, but it will also be less adaptable.
- The inclusion of EAs incorporates additional parameters that need to be optimised.

The solutions offered by our EA implementation in MCTS can be further refined to address the identified challenges in our research.

7.3 Evolutionary Algorithms-inspired Monte Carlo Tree Search: Conclusions

Our experiments with the FOP confirm that the UCB1's C values required by MCTS vary across different reward landscapes, as no parameter was universally superior. Specifically, deceptive reward landscapes require more exploration (greater C values) to find rewards that are hidden deeper in the tree. EA-MCTS and SIEA-MCTS performed inferiorly compared to the vanilla MCTS variants with optimal C values in the FOP domains and the game of Carcassonne. However, EA-MCTS and SIEA-MCTS were superior to the worst-performing vanilla MCTS variants in some cases, like the deceptive functions in FOP and the single-player Carcassonne variants, demonstrating their potential to adapt to different problem characteristics.

EA-MCTS and SIEA-MCTS showed a preference for exploitation over exploration, often exploiting the same few branches of the tree exclusively. However, their behaviour is greatly influenced by their evolutionary process, making them behave very differently from the vanilla MCTS variants with low C values. For instance, on occasion, some iterations of the EA-based MCTS variants were spent in branches of the tree with the lowest rewards. Although not necessarily undesirable, this finding suggests that the evolutionary process can potentially positively contribute to exploring regions of the tree that no robust tree policy formulae would.

EA-MCTS and SIEA-MCTS were found to be very volatile and sensitive to local optima. During the evolutionary process, it is hard for any newly evolved formula with a preference for exploration to compete with other formulae that focus on exploiting the best rewards. This is caused by a combination of many factors, but primarily by the fitness evaluation. The fitness evaluation in EA-MCTS and SIEA-MCTS measures the cumulative reward of the fitness iterations in which each evolved formula is used. This implies that formulae exploiting rewarding regions of the statistical tree in all of their fitness iterations are more likely to achieve better fitness than those investing resources in exploring less rewarding branches, biasing the search towards exploitation. Furthermore, greedy formulae that primarily exploit the same rewarding branches were found to survive over multiple generations due to the unlikelihood of producing offspring that can compete with them because the search space for tree policies is vast and flooded with undesirable or trivial solutions. For instance, we found that tree policies evolved by our EA-based MCTS variants can lack crucial terminals like the reward or the number of visits of each node on occasion, which are the primary source of information for the selection policy. Thus, by the time a new competitive formula is produced, the overall behaviour of EA-MCTS and SIEA-MCTS is heavily impacted and the statistical tree has already been biased. While that bias is not necessarily harmful, it can make exploration less rewarding for upcoming generations. This observation describes a feedback loop [119] in the evolutionary process in EA-MCTS and SIEA-MCTS, where each individual updates the statistical tree when evaluated, which is then used as input for the fitness evaluation of the next individual. This creates a *runaway sexual selection* [77] phenomenon where the preferred individuals are those who find the best rewards, which are then available to the next individuals who have no incentive to explore.

While the inclusion of semantics can be beneficial to guide the search and ensure a diverse population, we found that it did not yield significant benefits as implemented in SIEA-MCTS. SIEA-MCTS uses semantics when two evolved formulae share the same fitness evaluation, which has a variable likelihood of happening that depends on the domain. FOP and Carcassonne have continuous rewards, which made the likelihood of two formulae sharing the same fitness evaluation very low. We believe that semantics have potential in this domain, but the current implementation is not effective at exploiting it. We discuss a way to solve this issue in Section 7.5.

Overall, EA-MCTS and SIEA-MCTS offer a competitive alternative to the UCB1 formula, which is the default selection policy in MCTS. They are a promising first step toward the development of a more general and adaptable MCTS based on evolution. Our results indicate that, although competitive, EA-MCTS and SIEA-MCTS are not universally superior to vanilla MCTS variants and require further improvements to be competitive in all domains.

7.4 Taxonomy and transferability

MCTS has been expanded to address problems with various characteristics, such as MCTS-Solver [174], designed for games with abrupt termination states. However, many extensions are tailored to specific features of the game tree and lack general applicability. Moreover, it is generally unclear where each extension can be successfully applied and how the impact is altered when the attributes of the problem differ. We proposed Evolutionary Algorithm Monte Carlo Tree Search (EA-MCTS) and Semantically-Inspired Evolutionary Algorithm Monte Carlo Tree Search (SIEA-MCTS) to address the adaptability of the selection policy in MCTS, which is a general problem that can be applied to any domain. This thesis focuses on the domains we tested, which may not encompass all possible game tree characteristics and require further research. However, we can offer building blocks for a taxonomy based on the problem characteristics we have discussed in this thesis and how they relate to our results. This taxonomy is important to group problems [73] and to determine the transferability of our results to domains with similar characteristics and, in this case, to also identify the most suitable domains for evaluating MCTS variants.

Table 7.1 is compiled from Tables 2.1 and 4.1 in Chapters 2 and 4, respectively. It summarises the problem characteristics that we deemed ideal for testing various tree policies. These characteristics represent the simplest form of decision-making while yielding the most interpretable results.

The categories for branching factor and tree depth depend on several factors, including the time constraints of the algorithms, available computational resources, the cost of the forward model, and the problem's stages. We offer three categories: small, moderate, and large, as rough estimates based on the following criteria.

A small tree depth enables the algorithm to reach the bottom of the tree and expand terminal nodes in a few iterations, rendering exploitation less impactful on those branches. Conversely, a large tree depth is deep enough to render reward retrieval with Monte Carlo simulations inconsistent or unfeasible. A moderate tree depth allows the algorithm to retrieve rewards from terminal states while also requiring a balance between exploration and exploitation.

Regarding the branching factor, a large branching factor prevents MCTS from expanding beyond a few levels of depth, effectively functioning as a breadth-first algorithm. Conversely, a small branching factor demands a deeper tree to prevent the algorithm from reaching its bottom and incurring associated consequences. A moderate branching factor enables the algorithm to explore the tree in a balanced manner while maintaining a reasonable depth.

Additionally, we offer a taxonomy focused on the underlying reward landscapes of the problems that influence the performance of MCTS and its selection policy. Table 7.2

Table 7.1: Problem characteristics

Characteristic	**FOP**	**Carcassonne**	**Suggested**
Problem definition			
Pace	Turn-based	Turn-based	Turn-based
Turns	Atomic	Atomic	Atomic
Order	Sequential	Sequential	Sequential
Information	Perfect	Perfect	Perfect
	Complete	Incomplete	Complete
Randomness	Deterministic	Stochastic	Deterministic
Zero-sum	N/A	Non zero-sum	Any
Symmetry	N/A	Asymmetric	Any
Tree structure			
Abrupt termination states	No	No	No
Tree depth	Small	Moderate	Moderate
Branching factor	Small	Large	Moderate
Game state equivalences	Yes	Yes	Any
Progression	Progressive	Progressive	Progressive
Transpositions	No	Yes	Any

elaborates on Table 4.1 from Chapter 4 and summarises the reward landscapes of the problems we tested. The last column reflects the characteristics we believe offer the algorithms proposed in this thesis the most potential with their current design, but they are not limited to them.

Some of the characteristics presented in Table 7.2 are quantitative but lack defined thresholds. For instance, quantifying the smoothness or ruggedness of a function often involves calculating the Lipschitz constant, which is challenging in practice and requires knowledge of the function [88]. However, reward landscapes and their characteristics are typically unknown and difficult to evaluate, requiring estimations based on problem knowledge. We speculate that EA-MCTS and SIEA-MCTS could be enhanced to address the challenges posed by the reward landscapes of the problems under examination. For example, the deceptive functions in the FOP problem require enhanced exploration to find rewards hidden deeper in the tree. A summary of potential improvements to our proposed EA-based MCTS variants is provided in Section 7.5.

7.5 Future Work

We identified the following aspects of EA-MCTS and SIEA-MCTS that can potentially be improved.

- Although undesired or trivial formulae can be arguably beneficial as explained before,

Table 7.2: Reward characteristics

Characteristic	$f1$	$f2$	$f3$	$f4$	$f5$	**Carcassonne**	**Potential**
Modality	Uni	Multi	Uni	Uni	Uni	Multi	Uni
Deceptive	No	No	No	Yes	Yes	Yes	No
Bias in suboptimal moves	No	No	No	No	No	Yes	No
Shallow search traps	No	No	No	No	No	No	No
Smoothness	Yes	Yes	No	Yes	No	No	Any
Sparse rewards	No	No	No	No	No	No	No

the evolutionary process could benefit from a constrained offspring generation method that ensures that the offspring are valid. For instance, they can be tested on independent artificial statistical trees to ensure that a) they prefer the nodes with higher rewards when tied in visits, and b) they prefer the nodes with fewer visits when tied in rewards.

- A fitness evaluation based on rewards might not be the most effective approach. The current fitness evaluation is inherently biased toward exploitation. This problem can be addressed with a multi-objective optimisation approach, where the fitness evaluation is based on the trade-off between exploration and exploitation. Moreover, the co-existence of multiple solutions that do not dominate each other, as often occurs in populations of Multi-Objective Genetic Programming approaches, can be particularly beneficial under the context of evolving tree policies.
- While the fitness in EA-MCTS and SIEA-MCTS is permanent, the utility of each evolved formula can change as the statistical tree grows and updates. This implies that evolved formulae might become suboptimal as more MCTS iterations are executed. This can be addressed by re-evaluating the formulae in every generation. Under the context of selection policies, the additional resources investment used for re-evaluation are not a waste, since the game tree continues being explored and the statistical tree continues growing.
- Semantics can be re-defined for tree policies. To this end, semantics could focus on the traversed tree branches instead of the collected rewards. In this way, the algorithm would have access to valuable information about how the tree policy behaves, which can be used to guide the search and ensure a diverse population.
- Semantics could be employed in the offspring generation process rather than just when two individuals share a fitness. This solution could: a) prevent repetitive exploitation of the same tree branches, especially in consecutive iterations, and b) help mitigate the runaway sexual selection phenomenon.
- The evolutionary process can be extended beyond a fixed number of generations to effectively evolve the selection policy for as long as the MCTS is used. This would allow

the evolutionary process to adapt to the changes in the game tree as it grows and is updated, especially when used for an increasing number of iterations.

- The terminal set in the GP implementation for EA-MCTS and SIEA-MCTS could be expanded to include additional statistical tree information, such as the maximum reward of node children, the branching factor, or the standard deviation of the rewards. This approach could potentially increase the effectiveness ceiling of the evolved selection policy, assuming the evolutionary process can handle the expanded search space. Additionally, the implementation could be tailored to incorporate available domain-specific heuristic information, if so desired.
- The representation of the individuals in the population can be greatly improved through the use of a more expressive language, like grammatical evolution [142] or Cartesian Genetic Programming (CGP) [118]. This would allow for a more diverse population, a better shape of the solution search space and potentially more effective evolved selection policies. Furthermore, it is of interest to maintain mathematical expressions in the formula that serve a specific purpose, like the regret bound term in Upper Confidence Bounds (UCB1). To this end, the representation could be tailored to protect mathematical structures that provide specific benefits to the selection policy or ensure the inclusion of crucial terminals and functions that are known to be beneficial for the selection policy. For example, the evolutionary process might be focused on evolving the exploitation term of the selection policy, while the exploration term is kept constant. This would allow the evolutionary process to focus on optimising specific parts of the selection policy while ensuring that crucial information is not lost.

Bibliography

[1] Alexandros Agapitos, Julian Togelius, and Simon M Lucas. "Multiobjective techniques for the use of state in genetic programming applied to simulated car racing". In: *2007 IEEE Congress on Evolutionary Computation*. IEEE. 2007, pp. 1562–1569.

[2] Shipra Agrawal and Navin Goyal. "Analysis of thompson sampling for the multi-armed bandit problem". In: *Conference on learning theory*. JMLR Workshop and Conference Proceedings. 2012, pp. 39–1.

[3] Atif M Alhejali and Simon M Lucas. "Using genetic programming to evolve heuristics for a Monte Carlo Tree Search Ms Pac-Man agent". In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE. 2013, pp. 1–8.

[4] Fred Valdez Ameneyro and Edgar Galván. "Towards Understanding the Effects of Evolving the MCTS UCT Selection Policy". In: *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2022, pp. 1683–1690.

[5] Fred Valdez Ameneyro, Edgar Galván, and Ángel Fernando Kuri Morales. "Playing carcassonne with monte carlo tree search". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2020, pp. 2343–2350.

[6] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. "Deep RTS: a game environment for deep reinforcement learning in real-time strategy games". In: *2018 IEEE conference on computational intelligence and games (CIG)*. IEEE. 2018, pp. 1–8.

[7] Oleg Arenz. "Monte carlo chess". B.S. thesis. Technische Universität Darmstadt, 2022.

[8] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. "Best arm identification in multi-armed bandits." In: *COLT*. 2010, pp. 41–53.

[9] Peter Auer. "Using confidence bounds for exploitation-exploration trade-offs". In: *Journal of Machine Learning Research* 3.Nov (2002), pp. 397–422.

[10] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[11] Peter Auer et al. "The nonstochastic multiarmed bandit problem". In: *SIAM journal on computing* 32.1 (2002), pp. 48–77.

[12] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[13] Hendrik Baier and Peter Cowling. "Evolutionary MCTS with flexible search horizon". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 14. 1. 2018, pp. 2–8.

[14] Hendrik Baier and Peter I Cowling. "Evolutionary MCTS for multi-action adversarial games". In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2018, pp. 1–8.

[15] Bruce W Ballard. "The*-minimax search procedure for trees containing chance nodes". In: *Artificial Intelligence* 21.3 (1983), pp. 327–350.

[16] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[17] Amit Benbassat and Moshe Sipper. "EvoMCTS: A scalable approach for general game learning". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.4 (2014), pp. 382–394.

[18] Amit Benbassat and Moshe Sipper. "EvoMCTS: Enhancing MCTS-based players through genetic programming". In: *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE. 2013, pp. 1–8.

[19] Peter Bergh. *Domain independent enhancements to Monte Carlo tree search for eurogames*. 2020.

[20] Hans J Berliner. "Backgammon computer program beats world champion". In: *Artificial Intelligence* 14.2 (1980), pp. 205–220.

[21] Christopher Berner et al. "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680* (2019).

[22] Hans-Georg Beyer. *The theory of evolution strategies*. Springer Science & Business Media, 2001.

[23] Yngvi Björnsson and T. Anthony Marsland. "Risk management in game-tree pruning". In: *Information Sciences* 122.1 (2000), pp. 23–41.

[24] Djallel Bouneffouf and Irina Rish. "A survey on practical applications of multi-armed and contextual bandits". In: *arXiv preprint arXiv:1904.10040* (2019).

[25] Ivan Bravi et al. "Evolving game-specific UCB alternatives for general video game playing". In: *Applications of Evolutionary Computation: 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part I 20*. Springer. 2017, pp. 393–406.

[26] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[27] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[28] Sébastien Bubeck and Nicolo Cesa-Bianchi. “Regret analysis of stochastic and nonstochastic multi-armed bandit problems”. In: *arXiv preprint arXiv:1204.5721* (2012).

[29] Sebastien Bubeck et al. “X-Armed Bandits”. In: *Journal of Machine Learning Research* 12 (2011), pp. 1655–1695. ISSN: 1532-4435.

[30] Michael Buro. “The Othello match of the year: Takeshi Murakami vs. Logistello”. In: *ICGA Journal* 20.3 (1997), pp. 189–193.

[31] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. “Deep blue”. In: *Artificial intelligence* 134.1-2 (2002), pp. 57–83.

[32] Rodrigo Canaan et al. “Leveling the playing field: Fairness in AI versus human game benchmarks”. In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. 2019, pp. 1–8.

[33] Tristan Cazenave. “Evolving Monte Carlo tree search algorithms”. In: *Dept. Inf., Univ. Paris* 8 (2007).

[34] Olivier Chapelle and Lihong Li. “An empirical evaluation of thompson sampling”. In: *Advances in neural information processing systems* 24 (2011).

[35] Guillaume M JB Chaslot et al. “Progressive strategies for Monte-Carlo tree search”. In: *New Mathematics and Natural Computation* 4.03 (2008), pp. 343–357.

[36] Kumar Chellapilla and David B. Fogel. “Evolving an expert checkers playing program without using human expertise”. In: *IEEE Transactions on Evolutionary Computation* 5.4 (2001), pp. 422–428.

[37] Kumar Chellapilla and David B Fogel. “Evolving neural networks to play checkers without relying on expert knowledge”. In: *IEEE transactions on neural networks* 10.6 (1999), pp. 1382–1391.

[38] Hao-Cheng Chia, Tsung-Su Yeh, and Tsung-Che Chiang. “Designing card game strategies with genetic programming and monte-carlo tree search: A case study of hearthstone”. In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2020, pp. 2351–2358.

[39] Benjamin E Childs, James H Brodeur, and Levente Kocsis. “Transpositions and move groups in Monte Carlo tree search”. In: *2008 IEEE Symposium On Computational Intelligence and Games*. IEEE. 2008, pp. 389–395.

[40] Rudi Cilibrasi and Paul MB Vitányi. “Clustering by compression”. In: *IEEE Transactions on Information theory* 51.4 (2005), pp. 1523–1545.

[41] Zillions Development Corporation. *Zillions of Games*. `http://www.zillions-of-games.com`. Accessed: yyyy-mm-dd. 1998.

[42] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[43] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. "Exploration and exploitation in evolutionary algorithms: A survey". In: *ACM computing surveys (CSUR)* 45.3 (2013), pp. 1–33.

[44] Luis DaCosta et al. "Adaptive operator selection with dynamic multi-armed bandits". In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. 2008, pp. 913–920.

[45] Fernando Fradique Duarte et al. "A Survey of Planning and Learning in Games". In: *Applied Sciences* 10.13 (2020), p. 4529.

[46] Amaury Dubois, Julien Dehos, and Fabien Teytaud. "Improving Multi-modal Optimization Restart Strategy Through Multi-armed Bandit". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2018, pp. 338–343.

[47] Prajit K Dutta. *Strategies and games: theory and practice*. MIT press, 1999.

[48] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[49] Anikó Ekárt and Sándor Zoltán Németh. "A metric for genetic programs and fitness sharing". In: *Genetic Programming: European Conference, EuroGP 2000, Edinburgh, Scotland, UK, April 15-16, 2000. Proceedings 3*. Springer. 2000, pp. 259–270.

[50] Michael Emmerich. "Single-and multi-objective evolutionary design optimization assisted by gaussian random field metamodels". In: *PhD diss., University of Dortmund* (2005).

[51] Alvaro Fialho. "Adaptive operator selection for optimization". PhD thesis. Université Paris Sud-Paris XI, 2010.

[52] Álvaro Fialho et al. "Analyzing bandit-based adaptive operator selection mechanisms". In: *Annals of Mathematics and Artificial Intelligence* 60.1-2 (2010), pp. 25–64.

[53] Álvaro Fialho et al. "Dynamic multi-armed bandits and extreme value-based rewards for adaptive operator selection in evolutionary algorithms". In: *International Conference on Learning and Intelligent Optimization*. Springer. 2009, pp. 176–190.

[54] Hilmar Finnsson and Yngvi Björnsson. "Game-tree properties and MCTS performance". In: *IJCAI*. Vol. 11. 2011, pp. 23–30.

[55] David B Fogel et al. “A self-learning evolutionary chess program”. In: *Proceedings of the IEEE* 92.12 (2004), pp. 1947–1954.

[56] Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. “Artificial intelligence through simulated evolution”. In: (1966).

[57] Raluca D Gaina, Simon M Lucas, and Diego Pérez-Liébana. “Population seeding techniques for rolling horizon evolution in general video game playing”. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 1956–1963.

[58] Raluca D Gaina, Simon M Lucas, and Diego Perez-Liebana. “Rolling horizon evolution enhancements in general video game playing”. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2017, pp. 88–95.

[59] Edgar Galván and Peter Mooney. “Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges”. In: *IEEE Trans. in Artificial Intelligence* (2021).

[60] Edgar Galván and Marc Schoenauer. “Promoting semantic diversity in multi-objective genetic programming”. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*. 2019, pp. 1021–1029. DOI: `10.1145/3321707.3321854`. URL: `https://doi.org/10.1145/3321707.3321854`.

[61] Edgar Galván, Gavin Simpson, and Fred Valdez Ameneyro. “Evolving the MCTS Upper Confidence Bounds for Trees Using a Semantic-inspired Evolutionary Algorithm in the Game of Carcassonne”. In: *IEEE Transactions on Games* (2022).

[62] Edgar Galván, Leonardo Trujillo, and Fergal Stapleton. “Highlights of semantics in multi-objective genetic programming”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2022, pp. 19–20.

[63] Edgar Galván, Leonardo Trujillo, and Fergal Stapleton. “Semantics in multi-objective genetic programming”. In: *Applied Soft Computing* 115 (2022), p. 108143.

[64] Edgar Galván et al. “Statistical tree-based population seeding for rolling horizon eas in general video game playing”. In: *arXiv preprint arXiv:2008.13253* (2020).

[65] Edgar Galván-López et al. “Autonomous Demand-Side Management System Based on Monte Carlo Tree Search”. In: *IEEE International Energy Conference (EnergyCon)*. Dubrovnik, Croatia: IEEE Press, 2014, pp. 1325 –1332.

[66] Edgar Galván-López et al. “Evolving a ms. pacman controller using grammatical evolution”. In: *Applications of Evolutionary Computation: EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM,*

and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I. Springer. 2010, pp. 161–170.

[67] Edgar Galván-López et al. “Heuristic-Based Multi-Agent Monte Carlo Tree Search”. In: *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications.* IEEE. 2014, pp. 177–182.

[68] Edgar Galván-López et al. “Towards an understanding of locality in genetic programming”. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation.* 2010, pp. 901–908.

[69] Edgar Galvan-Lopez et al. “Using semantics in the selection mechanism in genetic programming: a simple method for promoting semantic diversity”. In: *2013 IEEE Congress on Evolutionary Computation.* IEEE. 2013, pp. 2972–2979.

[70] Aurélien Garivier and Eric Moulines. “On upper-confidence bound policies for switching bandit problems”. In: *International Conference on Algorithmic Learning Theory.* Springer. 2011, pp. 174–188.

[71] Sylvain Gelly et al. “Modification of UCT with patterns in Monte-Carlo Go”. PhD thesis. INRIA, 2006.

[72] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley, 1989. ISBN: 0-201-15767-5.

[73] Mario Graff and Riccardo Poli. “Automatic creation of taxonomies of genetic programming systems”. In: *European Conference on Genetic Programming.* Springer. 2009, pp. 145–158.

[74] Ole-Christoffer Granmo. “Solving two-armed Bernoulli bandit problems using a Bayesian learning automaton”. In: *International Journal of Intelligent Computing and Cybernetics* 3.2 (2010), pp. 207–234.

[75] Arthur Guez et al. “Learning to search with MCTSnets”. In: *International conference on machine learning.* PMLR. 2018, pp. 1822–1831.

[76] Shaun Hargreaves Heap and Yanis Varoufakis. *Game theory: a critical text.* Psychology Press, 2004.

[77] Jonathan M Henshaw and Adam G Jones. “Fisher’s lost model of runaway sexual selection”. In: *Evolution* 74.2 (2020), pp. 487–494.

[78] C. Heyden and Master Thesis Dke. “IMPLEMENTING A COMPUTER PLAYER FOR CARCASSONNE”. In: 2009.

[79] Wassily Hoeffding. “Probability inequalities for sums of bounded random variables”. In: *The collected works of Wassily Hoeffding* (1994), pp. 409–426.

[80] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[81] Christoffer Holmgård et al. “Automated playtesting with procedural personas through MCTS with evolved heuristics”. In: *IEEE Transactions on Games* 11.4 (2018), pp. 352–362.

[82] Jean-Baptiste Hoock and Olivier Teytaud. “Bandit-based genetic programming”. In: *European Conference on Genetic Programming.* Springer. 2010, pp. 268–277.

[83] Hendrik Horn et al. “MCTS/EA hybrid GVGAI players and game difficulty estimation”. In: *2016 IEEE Conference on Computational Intelligence and Games (CIG).* IEEE. 2016, pp. 1–8.

[84] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “Sequential model-based optimization for general algorithm configuration”. In: *International conference on learning and intelligent optimization.* Springer. 2011, pp. 507–523.

[85] Takahisa Imagawa and Tomoyuki Kaneko. “Enhancements in monte carlo tree search algorithms for biased game trees”. In: *2015 IEEE Conference on Computational Intelligence and Games (CIG).* IEEE. 2015, pp. 43–50.

[86] Mohiul Islam, Nawwaf Kharma, and Peter Grogono. “Mutation operators for genetic programming using Monte Carlo tree search”. In: *Applied Soft Computing* 97 (2020), p. 106717.

[87] Steven James, George Konidaris, and Benjamin Rosman. “An analysis of monte carlo tree search”. In: *Thirty-First AAAI Conference on Artificial Intelligence.* 2017.

[88] Donald R Jones, Cary D Perttunen, and Bruce E Stuckman. “Lipschitzian optimization without the Lipschitz constant”. In: *Journal of optimization Theory and Applications* 79 (1993), pp. 157–181.

[89] Terry Jones, Stephanie Forrest, et al. “Fitness distance correlation as a measure of problem difficulty for genetic algorithms.” In: *ICGA*. Vol. 95. 1995, pp. 184–192.

[90] Arthur Juliani et al. “Unity: A general platform for intelligent agents”. In: *arXiv preprint arXiv:1809.02627* (2018).

[91] Niels Justesen, Tobias Mahlmann, and Julian Togelius. “Online evolution for multi-action adversarial games”. In: *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30–April 1, 2016, Proceedings, Part I 19.* Springer. 2016, pp. 590–603.

[92] Niels Justesen and Sebastian Risi. “Continual online evolutionary planning for in-game build order adaptation in StarCraft”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2017, pp. 187–194.

[93] Niels Justesen et al. “Playing multiaction adversarial games: Online evolutionary planning versus tree search”. In: *IEEE Transactions on Games* 10.3 (2017), pp. 281–291.

[94] Giorgos Karafotias, Mark Hoogendoorn, and Ágoston E Eiben. “Parameter control in evolutionary algorithms: Trends and challenges”. In: *IEEE Transactions on Evolutionary Computation* 19.2 (2014), pp. 167–187.

[95] Emilie Kaufmann, Nathaniel Korda, and Rémi Munos. “Thompson sampling: An asymptotically optimal finite-time analysis”. In: *Algorithmic Learning Theory: 23rd International Conference, ALT 2012, Lyon, France, October 29-31, 2012. Proceedings 23*. Springer. 2012, pp. 199–213.

[96] Thomas Keller and Malte Helmert. “Trial-based heuristic tree search for finite horizon MDPs”. In: *Twenty-Third International Conference on Automated Planning and Scheduling*. 2013.

[97] Shauharda Khadka et al. “Collaborative evolutionary reinforcement learning”. In: *arXiv preprint arXiv:1905.00976* (2019).

[98] Donald E Knuth and Ronald W Moore. “An analysis of alpha-beta pruning”. In: *Artificial intelligence* 6.4 (1975), pp. 293–326.

[99] Levente Kocsis and Csaba Szepesvári. “Bandit based monte-carlo planning”. In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[100] Levente Kocsis and Csaba Szepesvári. “Discounted ucb”. In: *2nd PASCAL Challenges Workshop*. Vol. 2. 2006.

[101] Richard E Korf. “Depth-first iterative-deepening: An optimal admissible tree search”. In: *Artificial intelligence* 27.1 (1985), pp. 97–109.

[102] Jakub Kowalski and Radosław Miernik. *Legends of Code and Magic*. CodinGame. GitHub Repository. 2018. URL: `https://github.com/CodinGame/LegendsOfCodeAndMagic`.

[103] John R Koza. “Genetic programming II: Automatic discovery of reusable subprograms”. In: *Cambridge, MA, USA* 13.8 (1994), p. 32.

[104] Kamolwan Kunanusont et al. “The n-tuple bandit evolutionary algorithm for automatic game improvement”. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 2201–2208.

[105] Joel Lehman et al. “Safe mutations for deep and recurrent neural networks through output gradients”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2018, pp. 117–124.

[106] Ke Li et al. “Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition”. In: *IEEE Transactions on Evolutionary Computation* 18.1 (2013), pp. 114–130.

[107] Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer, 2008.

[108] Jialin Liu, Diego Pérez-Liébana, and Simon M Lucas. “Bandit-based random mutation hill-climbing”. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 2145–2151.

[109] Richard Lorentz. “Using evaluation functions in Monte-Carlo tree search”. In: *Theoretical computer science* 644 (2016), pp. 106–113.

[110] Simon M Lucas, Jialin Liu, and Diego Perez-Liebana. “The n-tuple bandit evolutionary algorithm for game agent optimisation”. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2018, pp. 1–9.

[111] Simon M Lucas, Spyridon Samothrakis, and Diego Perez. “Fast evolutionary adaptation for monte carlo tree search”. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2014, pp. 349–360.

[112] Simon M Lucas et al. “Efficient evolutionary methods for game agent optimisation: Model-based is best”. In: *arXiv preprint arXiv:1901.00723* (2019).

[113] Xiaobai Ma et al. “Monte-Carlo tree search for policy optimization”. In: *arXiv preprint arXiv:1912.10648* (2019).

[114] Hammad Majeed and Conor Ryan. “A less destructive, context-aware crossover operator for GP”. In: *Genetic Programming: 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10-12, 2006. Proceedings 9*. Springer. 2006, pp. 36–48.

[115] Jorge Maturana et al. “Extreme compass and dynamic multi-armed bandits for adaptive operator selection”. In: *2009 IEEE Congress on Evolutionary Computation*. IEEE. 2009, pp. 365–372.

[116] Reid McIlroy-Young et al. “Aligning superhuman ai with human behavior: Chess as a model system”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 1677–1687.

[117] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[118] Julian Francis Miller and Simon L Harding. “Cartesian genetic programming”. In: *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*. 2008, pp. 2701–2726.

[119] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[120] Thomas M Moerland et al. “Monte carlo tree search for asymmetric trees”. In: *arXiv preprint arXiv:1805.09218* (2018).

[121] David E Moriarty, Risto Miikkulainen, et al. “Discovering complex Othello strategies through evolutionary neural networks”. In: *Connection Science* 7.3 (1995), pp. 195–210.

[122] Rémi Munos et al. “From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning”. In: *Foundations and Trends® in Machine Learning* 7.1 (2014), pp. 1–129.

[123] Miguel Nicolau et al. “Evolutionary behavior tree approaches for navigating platform games”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 9.3 (2016), pp. 227–238.

[124] Tomasz P Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. “Semantic backpropagation for designing search operators in genetic programming”. In: *IEEE Transactions on Evolutionary Computation* 19.3 (2014), pp. 326–340.

[125] Judea Pearl. “The solution for the branching factor of the alpha-beta pruning algorithm and its optimality”. In: *Communications of the ACM* 25.8 (1982), pp. 559–564.

[126] Tom Pepels, Mark HM Winands, and Marc Lanctot. “Real-time monte carlo tree search in ms pac-man”. In: *IEEE Transactions on Computational Intelligence and AI in games* 6.3 (2014), pp. 245–257.

[127] Diego Perez, Spyridon Samothrakis, and Simon Lucas. “Knowledge-based fast evolutionary MCTS for general video game playing”. In: *2014 IEEE Conference on Computational Intelligence and Games*. IEEE. 2014, pp. 1–8.

[128] Diego Perez et al. “Evolving behaviour trees for the mario ai competition using grammatical evolution”. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2011, pp. 123–132.

[129] Diego Perez et al. “Rolling horizon evolution versus tree search for navigation in single-player real-time games”. In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 2013, pp. 351–358.

[130] Diego Perez-Liebana et al. "General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms". In: *IEEE Transactions on Games* 11.3 (2019), pp. 195–214.

[131] Diego Perez-Liebana et al. "General video game ai: Competition, challenges and opportunities". In: *Thirtieth AAAI Conference on Artificial Intelligence.* 2016.

[132] Diego Perez Liebana et al. "Open loop search for general video game playing". In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation.* 2015, pp. 337–344.

[133] James Pettit and David Helmbold. "Evolutionary learning of policies for MCTS simulations". In: *Proceedings of the international conference on the foundations of digital games.* 2012, pp. 212–219.

[134] Eric Piette et al. "Ludii–The Ludemic General Game System". In: *arXiv preprint arXiv:1905.05013* (2019).

[135] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.

[136] Xin Qiu and Risto Miikkulainen. "Enhancing Evolutionary Conversion Rate Optimization via Multi-Armed Bandit Algorithms". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 33. 2019, pp. 9581–9588.

[137] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. "On adversarial search spaces and sampling-based planning". In: *Twentieth International Conference on Automated Planning and Scheduling.* 2010.

[138] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. "On the behavior of UCT in synthetic search spaces". In: *Proc. 21st Int. Conf. Automat. Plan. Sched., Freiburg, Germany.* 2011.

[139] Ingo Rechenberg. "Evolution Strategy: Nature's Way of Optimization". In: *Optimization: Methods and Applications, Possibilities and Limitations.* Ed. by H. W. Bergmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 106–126. ISBN: 978-3-642-83814-9.

[140] Norman Richards, David E Moriarty, and Risto Miikkulainen. "Evolving neural networks to play Go". In: *Applied intelligence* 8 (1998), pp. 85–96.

[141] Franz Rothlauf and Franz Rothlauf. *Representations for genetic and evolutionary algorithms.* Springer, 2006.

[142] Conor Ryan, John James Collins, and Michael O Neill. “Grammatical evolution: Evolving programs for an arbitrary language”. In: *Genetic Programming: First European Workshop, EuroGP’98 Paris, France, April 14–15, 1998 Proceedings 1*. Springer. 1998, pp. 83–96.

[143] Ethan Dreyfuss Sam Schreiber Michael Genesereth et al. *GGP Base*. GitHub Repository. 2010. URL: `https://github.com/ggp-org/ggp-base`.

[144] Maarten PD Schadd et al. “Single-player Monte-Carlo tree search for SameGame”. In: *Knowledge-Based Systems* 34 (2012), pp. 3–11.

[145] Jonathan Schaeffer and Robert Lake. “Solving the game of checkers”. In: *Games of no chance* 29 (1996), pp. 119–133.

[146] Tom Schaul. “A video game description language for model-based or interactive learning”. In: *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE. 2013, pp. 1–8.

[147] Marc Schoenauer, Fabien Teytaud, and Olivier Teytaud. “A rigorous runtime analysis for quasi-random restarts and decreasing stepsize”. In: *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer. 2011, pp. 37–48.

[148] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[149] Steven L Scott. “A modern Bayesian look at the multi-armed bandit”. In: *Applied Stochastic Models in Business and Industry* 26.6 (2010), pp. 639–658.

[150] Claude E Shannon. “XXII. Programming a computer for playing chess”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[151] Brian Sheppard. “World-championship-caliber Scrabble”. In: *Artificial Intelligence* 134.1-2 (2002), pp. 241–275.

[152] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[153] David Silver et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

[154] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[155] David Silver et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359.

[156] Chiara F Sironi and Mark HM Winands. “Analysis of self-adaptive monte carlo tree search in general video game playing”. In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2018, pp. 1–4.

[157] Chiara F Sironi and Mark HM Winands. “On-line parameter tuning for Monte-Carlo tree search in general game playing”. In: *Computer Games: 6th Workshop, CGW 2017, Held in Conjunction with the 26th International Conference on Artificial Intelligence, IJCAI 2017, Melbourne, VIC, Australia, August, 20, 2017, Revised Selected Papers 6*. Springer. 2018, pp. 75–95.

[158] Chiara F Sironi et al. “Self-adaptive mcts for general video game playing”. In: *International Conference on the Applications of Evolutionary Computation*. Springer. 2018, pp. 358–375.

[159] Aleksandrs Slivkins et al. “Introduction to multi-armed bandits”. In: *Foundations and Trends® in Machine Learning* 12.1-2 (2019), pp. 1–286.

[160] Kenneth O Stanley et al. “Real-time evolution of neural networks in the NERO video game”. In: *AAAI*. Vol. 3. 2006, p. 13.

[161] Felipe Petroski Such et al. “Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning”. In: *arXiv preprint arXiv:1712.06567* (2017).

[162] Maciej Świechowski et al. “Monte Carlo tree search: A review of recent modifications and applications”. In: *Artificial Intelligence Review* (2022), pp. 1–66.

[163] William R Thompson. “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples”. In: *Biometrika* 25.3/4 (1933), pp. 285–294.

[164] Yuandong Tian et al. “Elf: An extensive, lightweight and flexible research platform for real-time strategy games”. In: *Advances in Neural Information Processing Systems* 30 (2017).

[165] Julian Togelius, Simon M Lucas, and Renzo De Nardi. “Computational intelligence in racing games”. In: *Advanced Intelligent Paradigms in Computer Games* (2007), pp. 39–69.

[166] Julian Togelius et al. “Super mario evolution”. In: *2009 ieee symposium on computational intelligence and games*. IEEE. 2009, pp. 156–161.

[167] Marco Tomassini et al. “A study of fitness distance correlation as a difficulty measure in genetic programming”. In: *Evolutionary computation* 13.2 (2005), pp. 213–239.

[168] John Tromp and Gunnar Farnebäck. “Combinatorics of go”. In: *Computers and Games: 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers 5*. Springer. 2007, pp. 84–99.

[169] Andrew Trusty, Santiago Santiago Ontañón, and Ashwin Ram. “Stochastic plan optimization in real-time strategy games”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 4. 1. 2008, pp. 126–131.

[170] Nguyen Quang Uy et al. “Semantically-based crossover in genetic programming: application to real-valued symbolic regression”. In: *Genetic Programming and Evolvable Machines* 12 (2011), pp. 91–119.

[171] John Von Neumann and Oskar Morgenstern. “Theory of games and economic behavior”. In: *Theory of games and economic behavior*. Princeton university press, 2007.

[172] Che Wang et al. “Portfolio online evolution in StarCraft”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 12. 1. 2016, pp. 114–120.

[173] Linnan Wang et al. *AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search*. 2019. arXiv: `1903.11059 [cs.CV]`.

[174] Mark HM Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. “Monte-Carlo tree search solver”. In: *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*. Springer. 2008, pp. 25–36.

[175] Sewall Wright et al. “The roles of mutation, inbreeding, crossbreeding, and selection in evolution”. In: (1932).

Acronyms

AI	Artificial Intelligence
ALE	Arcade Learning Environment
AMAF	All-Moves-As-First
AOS	Adaptive Operator Selection
B-RMHC	Biased-mutation RMHC
Bandit-based RMHC	Bandit-based Random-Mutation Hill-Climber
BB	Bridge Burning
BGP	Bandit-based Genetic Programming
CAC	Context Aware Crossover
CCG	Collectible Card Game
CERL	Collaborative Evolutionary Reinforcement Learning
cGA	Compact Genetic Algorithm
CGP	Cartesian Genetic Programming
CLT	Central Limit Theorem
CMAB	Combinatorial Multi-Armed Bandit
COEP	Continuous Online Evolutionary Planning
CoG	Conference on Games
CRO	Conversion Rate Optimisation
D-UCB	Discounted Upper Confidence Bounds
DeepGA	Deep Genetic Algorithm
DMAB	Dynamic Multi-Armed Bandit
DNN	Deep Neural Network
DOTA	Defense of the Ancients
EA	Evolutionary Algorithm
EA-MCTS	Evolutionary Algorithm Monte Carlo Tree Search
EAaltActions	RHEA then MCTS for alternative actions
EAroll	RHEA with rollouts
EAroll-att	EAroll plus NPC attitude check
EAroll-occ	EAroll plus occlusion detection
EAroll-seqPlan	EAroll plus sequence planning
ED	Edit Distance
EDI	Explicitly Defined Intron
ELF	Extensive, Lightweight and Flexible
EMCTS	Evolutionary Monte Carlo Tree Search
EP	Evolutionary Programming
ES	Evolution Strategy

Fast-Evo MCTS	Fast Evolutionary Monte Carlo Tree Search
FDC	Fitness Distance Correlation
FH-EMCTS	Flexible Horizon Evolutionary Monte Carlo Tree Search
FLA	Fitness Landscape Analysis
FOP	Function Optimisation Problem
FRGS	Fast Random Genetic Search
FRRMAB	Fitness-Rank-Rate-based Multi-Armed Bandit
GA	Genetic Algorithm
GAI	General Artificial Intelligence
GGP-Base	General Game Playing Base
GP	Genetic Programming
GVGAI	General Video Game AI
GVGP	General Video Game Playing
HOO	Hierarchical Optimistic Optimisation
IDDFS	Iterative Deepening Depth-First Search
KB Fast-Evo MCTS	Knowledge Based Fast Evolutionary Monte Carlo Tree Search
KB MCTS	Knowledge Based Monte Carlo Tree Search
LOCM	Legends Of Code and Magic
MAB	Multi-Armed Bandit
MCTS	Monte Carlo Tree Search
MCTSnets	Monte Carlo Tree Search Networks
MCTSPO	MCTS for Policy Optimization
MDP	Markov Decision Process
ML	Machine Learning
MMO	Multi-Modal Optimisation
MOEA/D	Multi-Objective Evolutionary Algorithm based on Decomposition
MOGP	Multi-Objective Genetic Programming
MSE	Mean Squared Error
NCD	Normalised Compression Distance
NN	Neural Network
NPC	Non-Player Character
NTBEA	N-Tuple Bandit Evolutionary Algorithm

OEP	Online Evolutionary Planning
QRDS	Quasi-random Restart with Decreasing Step-size
RHEA	Rolling Horizon Evolutionary Algorithm
RL	Reinforcement Learning
RMHC	Random-Mutation Hill-Climber
rtNEAT	Real-time NeuroEvolution of Augmenting Topologies
RTS	Real-Time Strategy
SA-MCTS$_{\mathrm{EA}}$	SA-MCTS with a simple Evolutionary Algorithm
SA-MCTS$_{\mathrm{NEA}}$	SA-MCTS with N-Tuple Bandit Evolutionary Algorithm
SA-MCTS$_{\mathrm{NMC}}$	SA-MCTS with Naïve Monte Carlo
SA-MCTS	Self-Adaptive Monte Carlo Tree Search
SAC	Semantics Aware Crossover
SCC	Semantic Similarity-based Crossover
SCD	Semantic-based Crowding Distance
SDO	Semantic-based distance as an additional criteriOn
SIEA-MCTS	Semantically-Inspired Evolutionary Algorithm Monte Carlo Tree Search
SMAC	Sequential Model-based Algorithm Configuration
SPO	Stochastic Plan Optimisation
SS	Sampling Semantics
SSC	Semantic Similarity-based Crossover
SSD	Sampling Semantic Distance
SSi	Semantic Similarity
STPS-RHEA	Statistical Tree-based Population Seeding RHEA
SW-UCB	Sliding Window Upper Confidence Bounds
SWcGA	Sliding Window compact Genetic Algorithm
TRPO	Trust Region Policy Optimization
TS	Thompson Sampling
UCB$_{++}$	UCB1 with Tree and Agent Variables
UCB$_{+}$	UCB1 with Tree Variables
UCB$_{\#}$	UCB1 with Tree, Agent and Game Variables

UCB1	Upper Confidence Bounds
UCT	Upper Confidence Bounds for Trees
URDS	UCB Random-restarts with Decreasing Step-size
VGC	Video Game Championship
VGDL	Video Game Descriptive Language