Monte Carlo Tree Search Strategies

A dissertation presented

by

Tom Vodopivec

to

The Faculty of Computer and Information Science

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in the subject of

Computer and Information Science

Ljubljana, 2018

Investing in your future
Operation Part Financed by The European Union
European Social Fund

Monte Carlo Tree Search Strategies

A dissertation presented

by

Tom Vodopivec

to

The Faculty of Computer and Information Science

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in the subject of

Computer and Information Science

Ljubljana, 2018

APPROVAL

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

— Tom Vodopivec —

January 2018

The submission has been approved by

dr. Branko Šter

Full Professor of Computer and Information Science

ADVISOR AND EXAMINER

dr. Matej Guid

Assistant Professor of Computer and Information Science

EXAMINER

dr. Mark Winands

Associate Professor of Computer and Information Science

EXTERNAL EXAMINER

Maastricht University, Department of Data Science & Knowledge Engineering

PREVIOUS PUBLICATION

I hereby declare that the research reported herein was previously published/submitted for publication in peer reviewed journals or publicly presented at the following occasions:

[1] Tom Vodopivec and Branko Šter. Enhancing upper confidence bounds for trees with temporal difference values. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, Dortmund, 2014. IEEE. doi: 10.1109/CIG.2014.6932895

[2] Tom Vodopivec. *Monte Carlo tree search: a reinforcement learning method*. Invited lecture, Maastricht University, Department of Knowledge Engineering, Maastricht, The Netherlands, 19/06/2015.

[3] Tom Vodopivec, Spyridon Samothrakis, and Branko Šter. On Monte Carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, 2017.

[4] Raluca D. Gaina, Adrien Couëtoux, Dennis J.N.J. Soemers, Mark H.M. Winands, Tom Vodopivec, Florian Kirchgeßner, Jialin Liu, Simon M. Lucas, and Diego Pérez-Liébana. The 2016 Two-player GVGAI competition. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1-1, 2017. doi: 10.1109/TCIAIG.2017.2771241

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Ljubljana.

Anki in Goranu

Od malih nog sta me spodbujala k razmišljanju in radovednosti. Sooblikovala sta moj značaj in željo po znanju – za vedno bosta zaslužna za uspehe v mojem življenju.

To Anka and Goran

Since I was small you nourished my curiosity. You helped to shape my personality and desire for knowledge – you will always have merit for the successes in my life.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*
Tom Vodopivec
Strategije drevesnega preiskovanja Monte Carlo

POVZETEK

Po preboju pri igri go so metode drevesnega preiskovanja Monte Carlo (ang. Monte Carlo tree search – MCTS) sprožile bliskovit napredek agentov za igranje iger: raziskovalna skupnost je od takrat razvila veliko variant in izboljšav algoritma MCTS ter s tem zagotovila napredek umetne inteligence ne samo pri igrah, ampak tudi v številnih drugih domenah. Čeprav metode MCTS združujejo splošnost naključnega vzorčenja z natančnostjo drevesnega preiskovanja, imajo lahko v praksi težave s počasno konvergenco – to še posebej velja za temeljne algoritme MCTS, ki ne uporabljajo dodatnih izboljšav. Zaradi tega jih raziskovalci in programerji pogosto združujejo z ekspertnim znanjem, hevristikami in ročno izdelanimi strategijami. Kljub izrazitim dosežkom tako izboljšanih metod (primer je AlphaGo, ki je nedavno prekosil najboljšega človeškega igralca igre Go na svetu in s tem premagal ta velik izziv umetne inteligence), takšne domensko-specifične izboljšave zmanjšujejo splošnost številnih aplikativnih algoritmov. Izboljšati temeljne algoritme MCTS, brez izgube njihove splošnosti in prilagodljivosti, je težko in predstavlja enega aktualnih raziskovalnih izzivov. Ta disertacija uvaja nov pristop za nadgradnjo temeljnih metod MCTS in izpopolnjuje temeljno razumevanje tega področja v luči starejšega ter uveljavljenega področja spodbujevalnega učenja (ang. reinforcement learning). Povezava med drevesnim preiskovanjem Monte Carlo, ki ga skupnost uvršča med metode za preiskovanje in planiranje, ter spodbujevalnim učenjem je že bila nakazana v preteklosti, a še ni bila temeljito preučena in tudi še ni pomembno vplivala na širšo skupnost umetne inteligence. S to motivacijo v tem delu poglobljeno analiziramo povezavo med tema dvema področjema, tako da identificiramo in opišemo podobnosti ter razlike med njima. Uvajamo praktičen pristop razširitve metod MCTS s koncepti iz spodbujevalnega učenja: naše novo ogrodje, *drevesno preiskovanje s časovnimi razlikami* (ang. temporal difference tree search – TDTS), pooseblja novo družino algoritmov, ki delujejo po konceptih MCTS, obenem pa za učenje koristijo

časovne razlike (ang. temporal differences) namesto vzorčenja Monte Carlo. To lahko razumemo kot posplošitev metod MCTS z učenjem s časovnimi razlikami in sočasno kot posplošitev klasičnih metod učenja s časovnimi razlikami z drevesnim preiskovanjem in ostalimi koncepti iz metod MCTS (kot so postopna širitev drevesa in uporaba privzete strategije). S pomočjo metod TDTS pokažemo, da uporaba uveljavljenih konceptov iz spodbujevalnega učenja v navezi z drevesnim preiskovanjem odpira možnosti za razvoj širokega spektra novih algoritmov, od katerih so klasične metode MCTS le ena izmed variant. V naših eksperimentih preizkusimo več tovrstnih algoritmov, osredotočimo pa se na razširitev algoritma UCT (ang. upper confidence bounds for trees) z algoritmom Sarsa(λ), ki je eden temeljnih algoritmov spodbujevalnega učenja. Naše meritve potrjujejo, da algoritmi TDTS dosegajo boljše rezultate na enostavnih igrah za enega igralca, klasičnih igrah za dva igralca in arkadnih video igrah: novi algoritmi ohranjajo robustnost in računsko ter pomnilniško zahtevnost, obenem pa konsistentno prekašajo algoritme MCTS. Naše ugotovitve zmanjšujejo razkorak med drevesnim preiskovanjem Monte Carlo in spodbujevalnim učenjem ter pozivajo k močnejšemu nadaljnjemu povezovanju teh dveh področij. Nazadnje, ta disertacija spodbuja k raziskovanju in uveljavljanju bolj enotnega pogleda na dve izmed temeljnih paradigem umetne inteligence – preiskovanje in učenje.

Ključne besede: umetna inteligenca, preiskovanje, planiranje, strojno učenje, drevesno preiskovanje Monte Carlo, spodbujevalno učenje, učenje s časovnimi razlikami, UCB, UCT, Sarsa, preiskovanje s časovnimi razlikami, namizne igre, video igre

ZAHVALA

Maša, največja zahvala gre tebi, saj si skupaj z menoj preživljala ta doktorska leta vse od začetka do konca. Potrpežljivo si prenašala moje delovne večere, mesece v tujini, dolgotrajno objavljanje člankov, pedagoške obremenitve in še vse kar je prišlo zraven. Ljubezen moja, res hvala za vse – skupaj sva uspešno zaključila ta življenjski projekt! ... pa rekla si tudi, da morem napisat, da si mi kuhala, pucala, pospravljala in skrbela za Marlija :)

Brane, velika zahvala gre tudi tebi, saj si mi bil vedno pripravljen nuditi podporo – zjutraj in zvečer, med tednom in med vikendi, si mi pomagal poganjati eksperimente, pisati enačbe in pripravljati odgovore na recenzije vseh vrst in oblik. Še enkrat hvala, da si me uvedel v to izredno zanimivo znanstveno področje in za mentorstvo skozi ta leta.

Vsi bližnji sodelavci in sošolci – Davor, Nejc, Uroš, Jure B., Jure D., Rok, Mattia in Brane (še enkrat) – fantje hvala vam za vse debate, anekdote, štose, odojke, čebovane, fantastiše food, turkish delight, WoTe, itd. Skupaj smo kljub "doktorskim" sitnostim poskrbeli za veselo in zabavno vzdušje. Še bomo kakšno rekli!

Dragi Spyros, najini pogovori v Essexu so obrodili odlično sodelovanje. Hvala za pomoč pri oblikovanju idej in piljenju člankov v zadnjih nekaj letih – resnično si mi olajšal pot do cilja. Želim ti vso srečo še naprej in upam, da ti bom lahko povrnil pomoč.

Hvala Diego, Mark in Simon za družbo na konferencah, gostoljubnost na raziskovalnih obiskih in za lepe spomine, ki sem jih prinesel domov iz tujine.

Hvala vsem članom komisije mojega doktorskega dela za vložen trud in potrpežljivost v zadnjem letu in pol – vaši komentarji so zelo pripomogli k uspešni objavi mojih raziskav in h kakovosti mojega doktorata.

Zahvala Evropski uniji, ki je preko Evropskega socialnega sklada sofinancirala velik del mojega doktorskega študija.

Hvala Marley za naslovno sliko – si zaslužiš veliko priboljškov :)

— Tom Vodopivec, Ljubljana, januar 2018.

University *of Ljubljana*
Faculty *of Computer and Information Science*
Tom Vodopivec
Monte Carlo Tree Search Strategies

ABSTRACT

Since their breakthrough in computer Go, Monte Carlo tree search (MCTS) methods have initiated almost a revolution in game-playing agents: the artificial intelligence (AI) community has since developed an enormous amount of MCTS variants and enhancements that advanced the state of the art not only in games, but also in several other domains. Although MCTS methods merge the generality of random sampling with the precision of tree search, their convergence rate can be relatively low in practice, especially when not aided by additional enhancements. This is why practitioners often combine them with expert or prior knowledge, heuristics, and handcrafted strategies. Despite the outstanding results (like the AlphaGo engine, which defeated the best human Go players, prodigiously overcoming this grand challenge of AI), such task-specific enhancements decrease the generality of many applied MCTS algorithms. Improving the performance of core MCTS methods, while retaining their generality and scalability, has proven difficult and is a current research challenge. This thesis presents a new approach for general improvement of MCTS methods and, at the same time, advances the fundamental theory behind MCTS by taking inspiration from the older and well-established field of reinforcement learning (RL). The links between MCTS, which is regarded as a search and planning framework, and the RL theory have already been outlined in the past; however, they have neither been thoroughly studied yet, nor have the existing studies significantly influenced the larger game AI community. Motivated by this, we re-examine in depth the close relation between the two fields and detail not only the similarities, but identify and emphasize also the differences between them. We present a practical way of extending MCTS methods with RL dynamics: we develop the *temporal difference tree search (TDTS)* framework, a novel class of MCTS-like algorithms that learn via temporal-differences (TD) instead of Monte Carlo sampling. This can be understood both as a generalization of MCTS with TD learning, as well

as an extension of traditional TD learning methods with tree search and novel MCTS concepts. Through TDTS we show that a straightforward adaptation of RL semantics within tree search can lead to a wealth of new algorithms, for which the traditional MCTS is only one of the variants. We experiment with several such algorithms, focusing on an extension of the upper confidence bounds for trees (UCT) algorithm with the Sarsa(λ) algorithm – an on-policy TD learning method. Our evaluations confirm that MCTS-like methods inspired by RL dynamics demonstrate superior results on several classic board games and arcade video games: the new algorithms preserve robustness and computational complexity, while consistently outperforming their basic MCTS counterparts. Our findings encourage cross-fertilization between the game AI and RL communities, hopefully narrowing the gap between them, and promote a unified view of search and learning methods, recommending it as a promising research direction.

Key words: artificial intelligence (AI), search, planning, machine learning, Monte Carlo tree search (MCTS), reinforcement learning, temporal-difference (TD) learning, upper confidence bounds (UCB), upper confidence bounds for trees (UCT), Sarsa, eligibility traces, temporal-difference search, board games, video games

ACKNOWLEDGEMENTS

Maša, the biggest thanks goes to you, because you have been pushing through these doctoral years of mine from the beginning till the end. You have patiently endured all my working evenings, months abroad, tiresome publishing processes, pedagogic workload and everything else that came by. My love, really thank you for everything – together we successfully completed this big life project! ... and, as you instructed me, I must also mention that you've been cleaning, ironing, cooking, and taking care of Marley :)

Branko, also a big thank you, because every time when I needed support, you were more than prepared to provide it. During weekdays and weekends you helped me run experiments, write equations, and craft cover letters for all kinds of reviewers. Thanks again for introducing me to science and thanks for your supervision throughout these years.

My close co-workers and classmates – Davor, Nejc, Uroš, Jure B., Jure D., Rok, Mattia and Branko (again) – thanks for all the gags, odojks, čebovans, fantastiše foods, turkish delights, WoTs, etc. Despite the doctoral annoyances, together we really had a great time.

Dear Spyros Samothrakis, our discussions in Essex started a great cooperation. Thank you for all the help in shaping our ideas and papers in the last few years. You truly helped me reach my goal faster and I still owe you for this. Wish you all the best my friend.

Thanks Diego Perez, Mark Winands, and Simon Lucas for your hospitality, for hanging out at conferences, and for the wonderful memories I brought back home from abroad.

Thanks to the doctoral committee members for the effort you put in reviewing this work and for your patience in the last year and a half. Your feedback significantly helped me in publishing my research and improving the quality of this work.

I am also grateful to the European Union for funding a substantial part of my doctoral study through the European Social Fund.

Thanks Marley for the cover photo – you deserve lots of treats :)

— Tom Vodopivec, Ljubljana, January 2018.

CONTENTS

I

Introduction

In 1997, the supercomputer Deep Blue won a six-game match against the World Chess Champion Garry Kasparov, making history [1]. The next great challenge of artificial intelligence (AI) became the ancient Asian game of Go [2]. All methods that were applied with great success to games until then, were ineffective in Go because of its orders-of-magnitude larger state space and branching factor – the state-of-the-art algorithms have been reaching only a human-beginner level of play. This lasted for nearly a decade, until in 2006 a new search paradigm unexpectedly elevated the playing strength of Computer Go players to a human-master level [3] – the field of *Monte Carlo Tree Search (MCTS)* [4] was born. Fuelled by these successes, MCTS quickly gained scientific attention and initiated almost a revolution in game-playing agents. In the following years, the game AI community devised an enormous amount of MCTS variants, extensions, and enhancements [5], and successfully applied them to very diverse domains, including a wide range of games: MCTS-based algorithms became the new state of the art in Go, Hex, Othello, and numerous other games. It took only another decade, and the grand challenge of Go fell; in March 2016, the AlphaGo engine [6] combined MCTS algorithms with deep neural networks [7] and defeated one of the world's best human Go players, and in May 2017 it defeated the current top Go player in the world.

MCTS provides a strong framework, based on the generality of random sampling and the precision of tree search. Provided an agent has access to an internal mental simulator, a considerable improvement in performance can be achieved compared to more traditional search methods. Despite this, the convergence rate of basic MCTS methods, including the most popular *upper confidence bounds for trees (UCT)* algorithm [8], can be relatively low in practice, especially when not aided by task-specific enhancements. Practitioners often need to resort to expert knowledge in form of handcrafted strategies, evaluation functions, and heuristics (see [3], for example). This decreases the generality of many applied MCTS algorithms. Furthermore, such enhancements are often computationally heavy, whereas their benefit varies greatly from task to task; it is difficult to identify the most suitable ones (and to configure them correctly) for a specific task. Improving the performance of core MCTS methods, while retaining their generality and scalability, is a research challenge [5].

We believe that the older and well-established field of reinforcement learning (RL) [9] can advance the fundamental view of MCTS methods, and in this way uncover new possibilities for generally improving them. Although MCTS has been introduced

as a search and planning framework for finding optimal decisions through sampling a given domain, strong relationships with RL have been suggested shortly after [10]. This link, however, was not immediately apparent and has not been widely adopted in the game artificial intelligence (AI) community. The relationship between the two fields remained somewhat cryptic, with most applied researchers treating MCTS as unrelated to RL methods. This distinction was also aided by the non-existence of a common language between dedicated game researchers and RL researchers and by the different fundamental motivations of both communities. For these reasons, the use of RL techniques within tree search has not been thoroughly studied yet.

Our goal is to advance the field of Monte Carlo tree search by deepening its theoretical insights towards the field of reinforcement learning, and in doing so, we hope to lessen the gap between the two communities and to encourage their cross-fertilization. At the same time, as a proof of concept, we want to exploit these ideas to produce new general and domain-independent MCTS algorithms.

In this thesis we overview the reasons for the two communities evolving separately, examine in depth the close relation between the two fields, and explore the benefits of enhancing MCTS with RL concepts. Focusing on the game AI community by starting from an MCTS point-of-view, we thoroughly describe the similarities between the two classes of methods; we illustrate how can RL methods using Monte Carlo sampling be understood as a generalization of MCTS methods. But, at the same time, we also explicitly emphasize the differences – we identify which MCTS concepts are not customary for traditional RL methods and can thus be understood as novel. We also survey the numerous MCTS algorithms and enhancements that (intentionally or unintentionally) integrate RL dynamics and explain them with RL terminology. Following these insights, we present a practical way of extending MCTS with RL dynamics: we develop new MCTS-like algorithms with the help of the *temporal-difference (TD) learning* [11] paradigm, which combines the advantages of Monte Carlo sampling (as used in MCTS) and dynamic programming [12] and often outperforms both, without sacrificing generality [9]. We develop the *temporal-difference tree search (TDTS)* framework, which can be understood both as a generalization of MCTS with TD learning, as well as an extension of traditional TD learning methods with novel MCTS concepts. It promotes a unified view of the two fields, building upon the line of research led by David Silver [10], who developed the temporal-difference search framework, which is similar to ours. We produce the first successful application of a TD learning method

to the original MCTS framework when using an incrementally-increasing tree structure with a tabular, non-approximated representation, and without domain-specific features; its efficient combination with the upper confidence bounds (UCB) [13] bandit policy; and the evaluation and analysis of TD value-updates (backups) under such conditions.

Through our new framework, we devise several MCTS-RL algorithms, including a generalization of the UCT algorithm with the TD-learning algorithm Sarsa(λ) [14]. We evaluate them on established benchmarks – on several toy games, single-player games, and two-player games. Our algorithms preserve the robustness and computational complexity, while consistently outperforming their MCTS counterparts: they reveal that learning from temporal-differences performs better than learning through Monte Carlo sampling also in MCTS-like settings. Finally, our *Sarsa-UCT(λ)* algorithm has been recently achieving strong results in the General Video Game AI (GVG-AI) competition [15], including two first positions in two-player competitions in 2016 and 2017.

1.1 Scientific contributions

In this thesis we studied several topics, from which we obtained three major and one minor contribution (Figure 1.1):

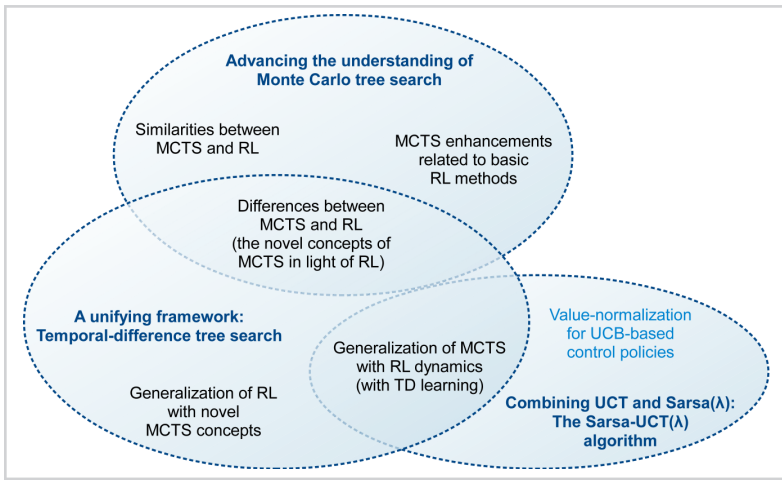

Figure 1.1

The topics studied in this thesis, grouped by the main contributions. We regard *value-normalization for UCB-based control policies* as a minor contribution that is not directly bound to MCTS or RL, but is more general and could be applied also elsewhere.

- *Advancing the theoretical understanding of Monte Carlo tree search in light of the reinforcement learning theory.* We thoroughly examine the relation between Monte Carlo tree search and reinforcement learning. We overview the reasons for the two communities evolving separately and detail not only the similarities between the two classes of methods, but also the differences – we identify the MCTS dynamics that can be understood as novel in light of the RL theory. We overview the existing MCTS enhancements and relate their dynamics to those of RL, and investigate the studies that (knowingly or unknowingly) observed some core RL concepts from an MCTS perspective. Our work popularizes and encourages the adoption of RL concepts and methods within the game AI community, hopefully narrowing the gap between the two fields.

- *A framework that unifies Monte Carlo tree search and reinforcement learning dynamics.* To explore the benefits of unifying MCTS and RL methods, we extend the RL theory to include the novel MCTS-concepts: we introduce into RL the concept of *representation policy* and the concept of non-memorized part of the search space, based on which we identify the need of additional assumptions regarding the non-memorized values. We analyse and evaluate temporal-difference value-updates (backups) under such conditions. Our new framework, denoted as *temporal-difference tree search*, can be understood both as a generalization of MCTS with TD learning and eligibility traces, as well as an extension of traditional TD learning methods with tree search and novel MCTS concepts. It supports a unifying view of both fields and introduces a new class of search and learning algorithms, opening several promising research directions.

- *A generalization of the UCT algorithm with an established temporal-difference learning method.* As a proof of concept of our unifying framework, we devise the first successful application of a TD learning method with eligibility traces – the Sarsa(λ) algorithm – to an MCTS method (the original UCT algorithm) when using an incrementally-increasing tree structure with a tabular representation, without value function approximation, and without domain-specific features. We showcase that the new algorithm, *Sarsa-UCT(λ)*, increases the performance of UCT when the new parameters are tuned to an informed value.

- *A value-normalization technique for efficient use of UCB selection policies within reinforcement learning algorithms.* For correct convergence, UCB (upper confidence bounds) selection policies require the feedback distribution (or the value estimates) to be normalized in a specific range. In the general reinforcement learning setting, it can be difficult to predict the feedback (reward) distribution and therefore it can be difficult to define the normalization bounds. We devised a *space-local value-normalization* technique that enables efficient use of UCB selection policies also in such settings, that is, also in combination with general RL algorithms. It computes the normalization bounds individually for each part of the search space and adapts them online by considering the feedback observed so far. The novel technique can be applied as a non-parametric enhancement to any search or learning algorithm that uses selection policies that require normalization of values.

1.2 Dissertation overview

In the next chapter we provide the background on MCTS, its basics framework, and the UCT algorithm, including a brief description of multi-armed bandits. We proceed with a thorough analysis of the relation between MCTS and RL (Chapter 3). We discuss the motivations behind each field and the reasons for the communities growing apart. We describe Markov decision processes and temporal-difference learning, so that we can link the terminologies of RL and MCTS and observe their similarities and differences. We conclude the chapter with a survey of MCTS enhancements that resemble established RL concepts. In the second part of our work we merge MCTS and RL into a common framework (Chapter 4) by extending the RL theory with MCTS concepts, generalizing UCT with TD learning, and developing an efficient value-normalization technique for UCB-based policies. There we also explain the relation of the newly-introduced parameters with known MCTS enhancements and provide detailed implementational remarks about the new algorithms. Then we overview the studies that also previously addressed the relation between the two fields and studies that produces algorithms by unifying concepts from both fields (Chapter 5). The analysis and experimental evaluation of our algorithms is performed on popular toy games (Chapter 6) and real games (Chapter 7). We discuss the performance of our algorithms and the limitations of our analysis, and propose promising research directions (Chapter 8). Finally, we comment on the impact of our contributions (Chapter 9).

2

Monte Carlo tree search

We introduce the main paradigm around which this thesis is centred – Monte Carlo tree search. Here we describe the basic background of the field, along with its key concepts, terminology, and dynamics.

2.1 Background

The field of *Monte Carlo tree search (MCTS)* was born in 2006 when Coulom [4] devised efficient strategies for combining *Monte Carlo search* with an *incremental tree structure*, coining the term *MCTS*. Simultaneously, Kocsis and Szepesvari [8] combined tree search with the *UCB1* action-selection policy [13], which is an asymptotically-optimal policy for solving the *multi-armed bandit* problem from the field of probability theory. This way, they designed the first and currently the most popular MCTS method – the *upper confidence bounds applied to trees (UCT)* algorithm.

Since then, MCTS methods have proven useful in numerous search and planning domains (e.g., scheduling, feature selection, constraint satisfaction, etc.) [5], but they particularly excel at decision problems that can be formulated as *combinatorial games*. MCTS methods are currently the state-of-the-art choice for a wide range of game-playing algorithms, spanning from classic games like Othello [16], Amazons [17], and Arimaa [18], to real-time games like Ms Pac-man [19], the Physical Travelling Salesman Problem [20], and Starcraft [21]. The General Game Playing competition [22] has also been dominated by MCTS-based algorithms for years [23–25]. Some MCTS-based players are stronger than top-level human players, e.g. in the game of Hex [26].

The most notable success of MCTS has been achieved on the ancient Asian game of Go [2] . There, the state space is orders of magnitude larger than, for example, the state space of chess. This is one of the reasons why methods that work well at the latter (like Alpha-Beta pruning [27], for example) prove much less effective on Go. Before the invention of MCTS, the best Go algorithms were achieving a human-beginner performance; MCTS quickly managed to elevate the playing strength to human-master level [4, 28, 29] – an accomplishment that was thought to be decades away; but even more impressive, in ten years since its inception it helped the *AlphaGo* engine [6] defeat the best human Go player in the world (in May 2017).

2.2 The framework

In general, MCTS algorithms employ strong (heuristic) *search* methods to identify the best available decision in a given situation [5]. They gain knowledge by *simulating* action sequences on the given problem and thus require at least its *generative model* (also known as a *forward*, *simulation*, or *sample* model).

The MCTS's search process memorizes the gathered knowledge by incrementally building a *search tree*. The growth of the tree is asymmetric and is guided in the most promising direction by an exploratory action-selection policy. Nodes in the tree represent actions (or states) of the given task and hold the values of their estimates, upon which future actions are selected. Like reinforcement learning methods [9], MCTS algorithms also deal with delayed feedback and the exploration-exploitation dilemma – the need to balance between exploration of unvisited state space and exploitation of knowledge to maximize the performance in a limited time setting.

The MCTS tree is computed iteratively and usually improves in quality when increasing the number of iterations. An *MCTS iteration* usually consists of four phases (Figure 2.1):

1. *selection* of actions already memorized in the tree (descent from the root to a leaf node);
2. *playout*, i.e., selection of actions until a terminal state is reached;
3. *expansion* of the tree with new nodes;
4. *backpropagation* of the feedback up the tree.

A *tree policy* guides the selection phase; it defines the policy for selecting actions in the memorized part of the state space. When the algorithm reaches a leaf node, it explores further into the non-memorized or unvisited state space with another action-selection policy until a terminal state is reached and an outcome is produced – it uses a *default policy* to guide the playout phase. The outcome is propagated and memorized up the tree as the newly-gained experience (feedback) about the task. We refer to the sequence of actions performed during the selection and playout phases as a *simulation*.

The MCTS framework is most often used for planning: the search is stopped after a number of iterations or when a computational time limit is reached, and then the memorized tree is used to output the "best" action available in the root node. The

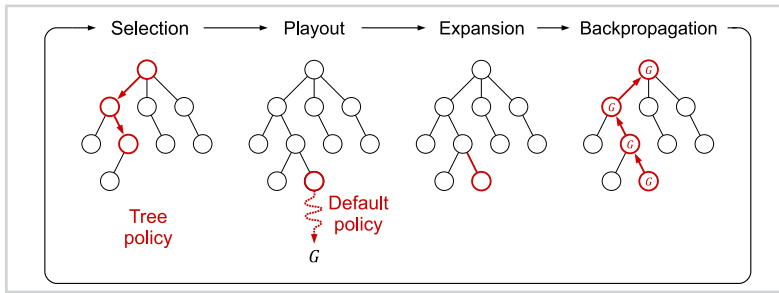

Figure 2.1

The iterative process of Monte Carlo Tree Search methods [30]. The feedback G gathered in each simulation (i.e., in the selection and playout phase) is propagated back the path traversed in the tree to update the memorized knowledge.

MCTS framework does not explicitly define the "best" action; however, the highest-estimated one or the most visited one are usually chosen, with the latter being more robust and, consequently, more popular.

It is also a known approach to delete the tree after each search and build it anew in the next search (in the next batch of MCTS iterations). The decision whether to employ such a technique is problem-specific. Such *forgetting* of already-gained knowledge inherently slows down the convergence rate, because the same knowledge might have to be discovered again; however, it also prevents previous estimates from biasing the future estimates, which is beneficial when past knowledge gets outdated quickly. This happens, for example, when the root state changes significantly between individual searches or when the simulation model is too noisy or only partially correct, hence producing incorrect estimates.

Although basic MCTS algorithms build a search *tree*, many of them perform better when building a directed *graph* based on *transpositions* [31]. To use transpositions, an algorithm must be able to observe and *identify* equal states, which is not always possible or might be computationally expensive.

The key advantage of MCTS algorithms is their generality, as they can be applied to any problem that can be formulated as a sequence of actions. Furthermore, they have a relatively low sensitivity to large state spaces, do not require domain-specific knowledge, but, when knowledge is available, they can efficiently use it. MCTS methods are also *anytime* – they can output an approximate solution after arbitrary execution time; and usually improve their performance when given more computational time.

Due to the large branching factors of the problems where MCTS is applied to, the convergence of basic MCTS algorithms may be slow. This has encouraged the

development of a multitude of extensions, enhancements, and variations of MCTS algorithms, many of which rely on strong heuristics or domain-specific knowledge. A comprehensive survey about MCTS and its extensions can be found in Browne et al.[5]. The MCTS enhancements and extensions that are relevant for our work are described throughout this thesis.

2.3 The UCT algorithm

The *upper confidence bounds for trees (UCT)* [8] is one of the first MCTS algorithms and is still the main representative of the field. It is one of the most studied, generally effective and widely used MCTS algorithms. One of the key characteristics that contributed to its success is that the UCT's tree policy guides the exploration by treating action-selection in each state (tree node) as an instance of a *multi-armed bandit problem*.

Multi-armed bandits

In a *multi-armed bandit problem* [32] the observer (e.g., a learning or planning algorithm) can choose multiple times one of the available actions, where each action returns a reward R based on some probability distribution. The true expected reward that is returned by each action – the *true value* of an action – is unknown to the observer. The goal is to maximize the collected reward in a given number of action-selections (i.e., online). Therefore, the observer should try to estimate the true value by trial and error. Each time, the observer decides either to *exploit* its current knowledge by selecting the best-evaluated action so far or to *explore* other actions to gather more knowledge – to improve the estimates. Balancing between these two tasks is known as the *exploration-exploitation dilemma*. The term *multi-armed bandit* was derived from the analogy to casino slot machines (also known as one-armed bandits).

The UCT tree policy

The UCT algorithm employs a tree policy that selects child nodes with the highest value of

$$Q_{\mathrm{UCT}} = Q_{\mathrm{MC}} + c_{n_p,n}\,, \tag{2.1}$$

where

$$Q_{\mathrm{MC}} = \frac{\sum G_i}{n} \tag{2.2}$$

represents the average feedback, i.e., the sum of rewards R, that was gathered from several MCTS iterations i that passed through a node and

$$c_{n_\mathrm{p},n} = C_p \sqrt{\frac{2 \ln n_\mathrm{p}}{n}} \tag{2.3}$$

is the *exploration bias* defined by the number of visits n to a node, the number of visits to its parent node n_p, and a weighting parameter $C_p \geq 0$. The parameter C_p defines how exploratory the algorithm is; a $C_p = 0$ produces greedy action-selection without exploration. Nodes with no visits ($n = 0$) are given the highest priority. By default, ties are broken randomly. For the algorithm to converge correctly, the term Q_{MC} must have a value in range $[0, 1]$, hence the same applies to individual feedbacks G_i. In such case, a default value of $C_p = 1$ proves to work well in practice.

The tree policy equations of UCT are based on the *UCB1* selection policy [13], which has been proven to optimally solve the exploration-exploitation dilemma in the limit. UCB policies select actions by considering the *upper confidence bound* of their estimates: the exploration bias from Equation (2.3) is a measure of uncertainty (variance) of the current estimated value of an action. The first UCB selection policies were devised by Auer et al. [13], who applied optimal bounds for a limited distribution to multi-armed bandits. They derived from Agrawal [33], who made the bounds easier to compute by defining them as a function of the total reward obtained by a single bandit, and from Lai and Robbins [34], who first analysed the optimal bounds on such distributions.

Two basic UCT variants

With regards to the MCTS framework, we acknowledge two distinct variants of the UCT algorithm: the *original UCT* algorithm [8] (this is the first developed version), and the more popular *standard UCT* algorithm, which is a modification that become widely adopted by MCTS practitioners [5]. Both algorithms use the tree policy described above and random action-selection as the default policy; however, the standard version simplifies some aspects of the original version.

The *original UCT* algorithm memorizes all the visited states in each iteration (when enough memory is available); identifies equal states that can be reached from different

sequences of actions (i.e., *transpositions*), evaluates them by averaging the outcomes, and stores them in a transposition table (i.e., builds a directed graph) [35]; considers the discounting of rewards (see Section 3.2); and distinguishes between terminal and non-terminal rewards by updating the values of visited states each with its appropriate sum of rewards following its visit in a simulation.

The *standard UCT* algorithm [5] memorizes only one new state per iteration and does not identify equal states, so does not use transpositions and builds a tree instead of a graph. It disregards discounting of rewards and does not distinguish between terminal and non-terminal rewards – it backpropagates only the final outcome (i.e., the sum of all rewards in a simulation). This leads to all visited states being updated with the same value (in a single simulation), which makes it easier to satisfy the convergence requirements of the UCB-based tree policy. Practitioners often re-extend this variant with transpositions [18, 25, 36]. Pseudocodes for both variants can be found in the original papers.

Although basic UCT variants use a uniform-random default policy, strong MCTS algorithms are most often equipped with informed playout policies (i.e. policies that incorporate a-priori knowledge), which greatly increase the performance of the algorithm [5]. Also, the playouts are often truncated (i.e., stopped before reaching a terminal state) to better search the space close to the current state, rather than diving deep into parts of the space that are less likely to be ever visited [37]. Finally, applied MCTS algorithms are often further augmented with domain-specific enhancements, with some popular examples being the *move-average sampling technique (MAST)* [23], the *all-moves-as-first (AMAF)* heuristic [38], and *rapid action value estimation (RAVE)* [39].

3

Relation to reinforcement learning

The main scientific contributions presented in this thesis derive from our understanding of the relation between Monte Carlo tree search (MCTS) and reinforcement learning (RL) methods. Here we thoroughly explain and justify this relation, before presenting our algorithms later on. This chapter attempts to increase the awareness of a unifying view of both fields, with the goal of improving their cross-fertilization.

First, we discuss different points of view on learning, planning, and search, which are probably among the reasons why the RL community and the search-and-games community evolved separately. Then, we introduce to the reader Markov decision processes – a framework for modelling decision-making problems that both MCTS and RL methods can solve. We proceed with the basics of reinforcement learning, introduce the terminology of RL concepts that make up the MCTS search dynamics, and link RL concepts to the corresponding MCTS iteration phases, while emphasizing the similarities. We investigate also the fundamental differences – the novel concepts that MCTS methods introduce from an RL perspective. Lastly, we survey the MCTS enhancements that implement similar mechanics as RL.

3.1 On learning, planning, and search

Strong links between *learning* and *planning* methods have been observed for decades [9] and have also been re-analysed by several researchers in the recent years [40–42]. Both paradigms solve similar problems and with similar approaches; both estimate the same value functions and both update estimates incrementally. The key difference is the source of experience: whether it comes from real interaction with the environment (in learning) or from simulated interaction (in planning). This implies that any learning methods, including RL methods, can be used for planning, when applied to a simulated environment. Thereafter, given that MCTS is regarded as a search and planning method, its search dynamics are comparable with those of sample-based RL methods, and the overall MCTS framework is comparable with RL applied to planning. However, this connection might not be obvious at first sight – there are many researchers that perceive only a marginal relation between the two fields. Below we suggest several causes for this discrepancy, but we also argue that the fields in fact describe very similar concepts, only from different points of view.

The viewpoints of these two communities are different because the underlying motivations and goals of (heuristic) search and (reinforcement) learning methods are fundamentally different. MCTS researchers are less interested in generic learning solutions

and more keen to attack specific games with specific approaches; the knowledge (i.e., learning) is usually not transferred between different searches and games. On the other hand, the RL community focuses mostly on universally-applicable solutions for a wider class of problems and its philosophical foundations discourage using (too much) expert knowledge in unprincipled ways (although this can prove very effective in practice).

As a consequence of the above, search methods are in general more focused on creating strong selection policies, often by exploiting heuristics (as is the case also with MCTS), whereas RL methods are more focused on devising strong techniques for updating the gained knowledge, giving less attention to selection policies. An example of this is how the two classes of methods handle exploration: MCTS methods usually entrust it to the selection policy itself (which is most often the UCB policy), whereas RL methods often set optimistic initial estimates, which then guide the selection policy towards less-visited parts of the state-space. Despite these differences, both methods still need to perform all the common tasks listed above – both need to efficiently select actions, update the knowledge, and balance the exploration-exploitation dilemma.

Another major reason is the common assumption in RL that we do not have a (good) model of the environment. Reinforcement learning understands the sample-based planning scenario as a special case – as the application of learning methods to a simulated environment, for which we do require a model. Nevertheless, it is not necessary to have a perfect or complete model, but an approximate or generative one might be enough (as in MCTS). Alternatively, the model need not be given, but the methods can learn it online (i.e., keep track of the transition probabilities). Recent RL-inspired studies confirm that MCTS-like algorithms can also be applied in such a way to non-simulated experience (Section 5.3). Furthermore, even in traditional MCTS, where a generative model is available, the model is not necessarily perfect or complete; it might be highly stochastic or simulate only part of the environment (as in the GVG-AI competition [15], for example), it usually does not disclose the transition probabilities, and it does not simulate the moves of other agents involved in the task. For example, in multi-player games, the other players are usually understood as a part of the environment that cannot be easily predicted, and are usually modelled separately. Considering the above, also MCTS methods need to deal with model-related issues, like RL methods.

Search methods often build a tree structure based on decisions, due to most games being of such nature, and regard state-observability as a bonus that allows the use of

transposition tables. On the other hand, in RL state-observability is often taken for granted: RL research usually focuses on more generic, fully connected graphs of states and actions, where transpositions are encoded by default in the representation. In practice, graphs are rarely explicitly represented, simpler RL methods rather use tabular representations (equal to transposition tables), and most RL methods use function approximation techniques. Despite these differences in the underlying representations, both search and RL methods strive to generalize the gained knowledge – transfer it to relevant parts of the state-space. When linking MCTS with RL, Baier [43] uses the notion of "neighbourhoods" for parts of the state-space that share knowledge (that are represented by the same estimates): in MCTS the neighbourhoods are usually defined by specific enhancements that group states according to expert knowledge or heuristic criteria (such as AMAF [38], for example), whereas in RL they are defined by the features used for function approximation (which are, in practice, also often selected according to expert knowledge, but can often be very natural, such as direct board images, for example).

Lastly, the games and search community embraced MCTS as a search technique also due to its quick successes in the game of Go [3]. Even though the original UCT algorithm [8] was designed and analysed using Markov decision processes (MDPs) (Section 3.2), the community adopted a variant that ignores discounting of rewards, which is an integral part of MDPs and RL methods.

All the above might have led to the lack of common language between the MCTS and RL fields. As a consequence, much research has been done on similar concepts, albeit from different perspectives – improving this cross-awareness would benefit both communities. The rest of this chapter makes a step towards this goal and illustrates how can RL methods be understood as a generalization of MCTS methods.

3.2 Markov decision processes

One way of describing *decision problems* (also referred to as *tasks*) that both RL and MCTS methods can solve is modelling them as *Markov decision processes (MDPs)* [44]. These form the basic substrate of decision-making algorithms; all the work in this thesis is based upon such representations.

An MDP (example in Figure 3.1) is composed of

- *States* $\mathcal{S}$, where s is a state in general and $S_t \in \mathcal{S}$ is the (particular) state at

time t.

- *Actions* $\mathcal{A}$, where a is an action in general and $A_t \in \mathcal{A}(S_t)$ is the action at time t, chosen among the available actions in state S_t. If a state has no actions then it is *terminal* (also known as a *stopping* or *absorbing* state), for example, ending positions in games.
- *Transition probabilities* $p(s'|s,a)$: the probability of moving to state s' when taking action a in state s: $\Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$.
- *Rewards* $r(s,a,s')$: the expected reward after taking action a in state s and going to state s', where $R_{t+1} = r(S_t, A_t, S_{t+1})$.
- The *reward discount rate* $\gamma \in [0,1]$, which decreases the importance of later-received rewards.

At each discrete *time step* t, an action is selected according to the *policy* $\pi(a|s)$, which defines the probabilities of selecting actions in states – it defines how will an agent behave in a given situation. It may be a simple mapping from the current state to actions, or may also be more complex, e.g., a search process. An MDP is *deterministic*

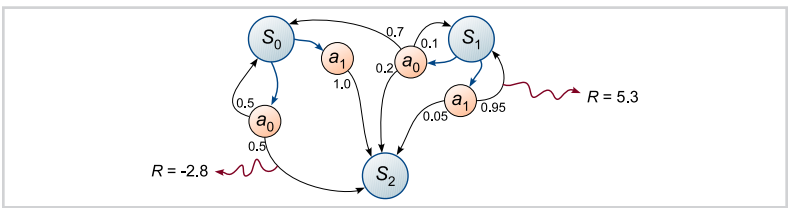

Figure 3.1

An example Markov decision process with three states. The state S_2 is terminal since it has no actions.

when there is no randomness in the transitions – when every action leads to states with probability either 1 or 0; otherwise it is *stochastic*; and the same applies to policies.

Solving an MDP requires finding a policy that maximizes the *cumulative discounted reward* or the *return*

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots . \tag{3.1}$$

In other words, an optimal policy maximizes the return from any state. To devise a good policy, an agent can move through the MDP state space (explore it in a trial-and-error fashion) and remember – *learn* – which actions in which states led to high rewards

in order to use this knowledge to select better actions in the future. Such algorithms usually try to learn (estimate) the *values* of states $V(s)$ and state-actions $Q(s,a)$. The value of a state $V_\pi(s)$ is the expected accumulated reward in the future (i.e., the expected return) when the agent continues from that state following a policy π; it differs from the immediate reward. The value of a state-action $Q_\pi(s,a)$ is the expected return when the agent performs action a in state s and then follows the policy π. The *value function* assigns such values to all states or state-actions. An optimal value function assigns such expected returns as if the agent were following an optimal policy.

Simultaneously to learning the value function, an agent may as well keep track of transition probabilities $p(s'|s,a)$, that is, learn a *transition model* of an MDP. A model can be used for *planning*, that is, for learning from simulated interactions with the environment. Sometimes, a model might be already available – it might be given to the algorithm before learning begins. We distinguish between *complete models*, which hold all the information about an MDP (including individual transition probabilities and rewards), and *forward models*, which can be used as a black box for outputting the next state and reward without explicitly knowing the underlying structure of the MDP. The latter are simpler to construct and are customary used by MCTS methods, as already mentioned in Section 2.2.

In an interactive setting, the agent has to repeatedly decide whether to choose actions which seem the best, based on its gathered experience (i.e. the exploitation of knowledge), or to explore unknown space in search of a better solution. This is the same *exploration-exploitation dilemma* that is present in multi-armed bandit (MAB) problems, as we described in Section 2.3. In fact, MDPs can be understood as a generalization of bandit problems: a MAB problem is formally equivalent to a single-state MDP, because the final outcome (sum of rewards) is known immediately after selecting an action[42]. Solving problems where the outcome is not known immediately (where the feedback is delayed) is more difficult; these classify as *reinforcement learning* problems [9], which we describe in the next Section.

MDP tasks may be *episodic* or *continuing*. Episodic tasks consist of sequences of limited length (not fixed), referred to as *episodes*. We will denote the final time step of a single episode as T, but knowing that its value is specific for the particular episode. On the other hand, continuing tasks have no episodes (or can be understood to have a single episode with $T = \infty$). The return from the current time step t until the final

time step T in an episodic situation is

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T \,, \qquad (3.2)$$

where $T - t$ is the remaining duration of the episode.

Discounting ($\gamma < 1$) makes early-collected rewards more important, which in practice results in preferring the shortest path to the solution. Apart from this, discounting is necessary to ensure convergence in MDPs of infinite size, in MDPs with cycles, and when function approximation is used. However, throughout this thesis we will emphasize the case of γ being set to 1. This is due to the fact that many of the games we experiment with have tree-like structures (without cycles) and are customarily not treated with $\gamma < 1$; in most games it is irrelevant how many moves are required to achieve a win.

To summarize, formally, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$ with a solution policy π, which can be devised via learning or planning with the help of state values $V(s)$ and state-action values $Q(s, a)$ and a (learnt or given) model.

3.3 Reinforcement learning

Reinforcement learning is a well-understood and established paradigm for agents learning from experience [9]. It is often considered a class of problems (spanning over control theory, game theory, optimal control, etc.), but may as well be considered a class of methods that can solve RL problems. It is regarded as one of the most general known learning paradigms, because it can efficiently learn from a simple feedback in the form of a scalar signal.

We derive most of our fundamental understanding of RL from the comprehensive work of Sutton and Barto [9]. A second edition of their book is in progress, where they address also MCTS methods [42].

The basic model

In a RL problem (Figure 3.2), an *agent* performs *actions* in an *environment*, gathers *experience* (i.e., sample interactions with the environment) by observing the *state* of the environment, and receives feedback in form of *rewards*. The agent uses rewards as basis for adapting its *behaviour* (i.e., identifying which actions to take in a certain state) to

maximize its cumulative reward in the future. Rewards may be immediate, but may as well be delayed in time, which makes the problem more difficult. It is particularly suitable to model such an environment with Markov decision processes (MDPs). Since multi-armed bandit problems are special cases of MDPs, algorithms for solving them also classify as RL methods.

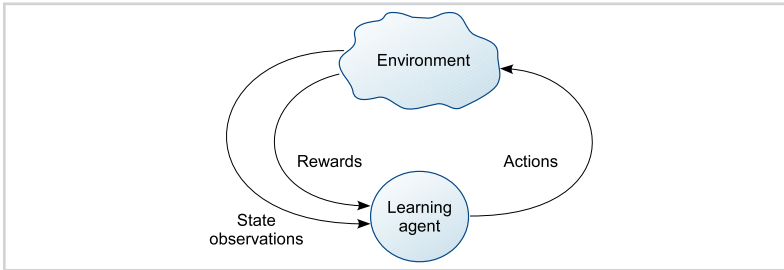

Figure 3.2

The basic reinforcement learning model [45]. State observations may carry full or partial information about the current state of the environment.

An environment might be *fully observable* or *partially observable*. In the former, *observations* contain all the information that is required to uniquely distinguish all the states (and state-actions) in the environment. In the latter, observations carry limited or noisy information about the current state, which makes the problem harder, because the agent has to imply its location in the state space – at best, it can narrow down to a subset of possible states. This is usually modelled with *partially observable MDPs* [46] (which are related to imperfect-information games); however, these are not in the focus of this thesis.

Since the agent is not told which actions to take, the RL problem is harder than the *supervised learning* problem. The agent is required to discover the best behaviour by analysing the environment's model or by *learning* from experience. The latter can be already available (given in advance) or might be gathered through the *exploration* of the environment (by trying out different actions). Same as in MAB and MDP problems, the exploration-exploitation dilemma is a central challenge also for RL problems, because the agent has to decide whether to take actions that yielded high reward in the past or to explore less-known actions, which might turn out to be even better.

In general, RL problems can be solved by learning a value function and then using it to design a good policy (Section 3.2). Methods that operate in such a way are known as *action-value* methods, with the three most common being dynamic programming, Monte Carlo methods, and temporal-difference methods. The latter two classify

as *sample-based* RL methods, which, unlike dynamic programming, do not require a model of the environment, but can learn from sample interactions. In this study we focus on these; however, RL problems can also be solved differently, without explicitly learning the value function – for example, with policy gradient methods and evolutionary algorithms. A survey of many state-of-the-art RL algorithms is given by Wiering and Otterlo [47].

Dynamic programming

The *dynamic programming (DP)* paradigm is the theoretical foundation of numerous RL methods. It provides the means to iteratively compute the optimal value function using the *Bellman equation* [12]. The most distinguished DP methods are *policy iteration* and *value iteration*. The main drawback of DP methods is that they require a complete and accurate model of the environment (i.e., of the MDP) [42]. Due to this, they do not scale well to large state spaces.

Monte Carlo methods

Unlike dynamic programming methods, large state spaces are exactly where *Monte Carlo (MC)* methods prove useful. Since MC methods learn from experience, they do not require knowledge of the environment – do not require a model; therefore, they are not affected by the main drawback of DP methods. When no model is available, MC methods gather experience only from *online* sampling, but when a model is available, they can use it to gather experience also from *simulated* sampling, like MCTS methods do. Moreover, it is enough if the model outputs only sample transitions (e.g., like a generative model, as described in Section 2.2), which is still less demanding than the complete transition model needed by DP methods. In general, MC methods learn a value function by averaging the experienced rewards. This is also their main drawback: depending on the task, they might produce only a near-optimal solution in a finite amount of time. However, when given enough computational time, they can find an arbitrarily accurate solution. MC methods had been treated separately from other RL methods for decades, until 1998 [9], but their first recorded use in such a setting dates back to 1968 [48].

Temporal-difference methods

The *temporal-difference (TD) learning* paradigm unifies the advantages of both DP and MC methods, and is regarded as the biggest achievement of decades of research in the reinforcement learning field. TD methods can learn from sampled experience, like MC methods, but offer better convergence guarantees, like DP methods. Two of the most popular RL algorithms employ TD learning: TD(λ) [11] and Q-learning [49]. We describe these methods further below in this chapter and put them to use in Chapter 4, where we show how to generalize and improve the learning aspects of MCTS with such well-understood RL dynamics.

Value-function approximation

RL methods are often inherently associated with techniques for *state-space approximation*. While it is true that many basic RL algorithms have difficulties handling large state spaces and thus benefit from such approximation (which is also known as *value-function* approximation), this is not always necessary. Algorithms that employ simpler representations, such as *tabular* (i.e., a table with a single entry for each state), also work well on certain tasks. We focus on those, for simplicity and for ease of comparison with MCTS methods, which are usually implemented exactly with such discrete representations. Despite this, the ideas we present in this thesis also extend to more complex or approximate representations.

3.4 Linking the terminology

We illustrate the similarities between MCTS and sample-based RL methods by linking the terminologies of both fields; we position the dynamics that make up the MCTS iteration phases within the RL theory and acknowledge the MCTS researchers who observed some traditional RL dynamics from an MCTS perspective.

Sampling-based planning

From an RL point of view, *Monte Carlo learning methods* sample through the state space of a given task to gather experience about it in an *episodic* fashion. Increasing the number of samples produces more experience to learn from, therefore, it usually improves the performance of the algorithm. When a model of the task is available, the experience can be simulated – in such case the simulated *episodes* are analogous

to MCTS *simulations*. A *trajectory* is the path that a learning algorithm takes in an episode; in MCTS this is the path from the root node to a terminal position in an iteration – it comprises the visited states and selected actions in both the selection and playout phases. Episodes and trajectories have a length of one or more (simulated) *time steps*, with one selected action per time step. Each occurrence of state s or state-action pair (s, a) in a trajectory is called a *visit* to s or (s, a), respectively.

Observing rewards

The gathered experience (i.e., the trajectories and the collected rewards) can be used to improve the value estimates – to update the value function. MDP-based methods, including RL and the standard UCT variant[8], assign credit to a state (or state-action) by considering the rewards that were collected exclusively *after* that state was visited. Otherwise, the evaluation might get biased by assigning credit to states and actions that do not deserve it. Therefore, RL algorithms remember the time of occurrence of each reward in the trajectory – they distinguish between terminal and non-terminal rewards – and update the value of each visited state with its appropriate return (i.e., sum of rewards). Considering this, the *original* UCT variant [8] complies with RL theory, whereas the *standard* UCT variant [5] ignores the time of occurrence of rewards and updates all the visited nodes with the same final sum of rewards. This may lead to sub-optimal (or erroneous) behaviour of the standard UCT variant on tasks that exhibit non-terminal rewards; however, such implementation is often used exactly on tasks with terminal rewards only (e.g., on classic games that exhibit an outcome only in a terminal position), which mitigates this drawback.

Memorizing feedback

A *backup* is the process of updating memorized knowledge (i.e., updating the value function) with newly-gathered feedback; we refer to it as a synonym for the MCTS backpropagation[1] phase. An *online* algorithm computes backups after each time step, whereas an *offline* algorithm computes backups in batch at the end of each episode. This is analogous to performing the MCTS backpropagation phase multiple times in a single MCTS iteration (after each action), or performing it once, at the end of the playout phase, as assumed in the classic MCTS framework. Thus, the classic MCTS framework proposes offline algorithms. Basic MC methods in RL are also offline.

[1] *Backpropagation* is also a training method for artificial neural networks.

Guiding the exploration – control

In RL, the concept of *control* assumes that trajectories are guided by a *control policy*, which dictates which actions will be performed, and, consequently, which states will be visited. It encloses both the MCTS tree policy and the MCTS default policy. RL methods that follow a control policy are generally known as *RL control* methods, and those that learn by Monte Carlo sampling classify as *Monte Carlo control* methods.

A policy may explicitly maintain the probabilities of selecting actions in states $\pi(a|s)$ (all or a part of them), or might compute them only when needed, and discard them afterwards (e.g., compute them only for the current state, online). The latter requires less memory resources and usually only marginally increases the computational time, hence it is frequent in practical implementations of RL methods, and is also typical for MCTS algorithms.

To guarantee convergence, a RL sampling algorithm must visit all states (or state-actions) an infinite number of times in the limit, therefore, its control policy must always keep choosing each action in each state with a probability greater than zero – it must keep exploring forever. Two examples of such policies are the UCB1 policy (described in Section 2.3) and the *ε-greedy* policy [9]. The latter is popular among basic RL methods due to its simplicity; it defines action-selection probabilities as

$$\pi(a|S_t) = \begin{cases} 1-\varepsilon & \text{for the best-evaluated action in state } S_t \text{ at time } t, \\ \varepsilon/|\mathscr{A}(S_t)| & \text{otherwise.} \end{cases} \tag{3.3}$$

The parameter $\varepsilon \in [0, 1]$ balances the trade-off between exploration and exploitation by defining the probability of not choosing the best action, but rather a sub-optimal one. For example, an $\varepsilon = 1$ results in a random policy, whereas a $\varepsilon = 0$ results in a greedy policy. Therefore, it defines the *exploration rate* of the policy, similarly as the parameter C_p in the UCT algorithm.

A sample-based RL algorithm is guaranteed to converge to an *optimal* solution if, beside using an exploratory control policy, its exploration rate decays to zero in the limit [50]. Hence, decreasing ε with time often yields a higher performance, and becomes even more similar to UCB1, which gets less exploratory with increasing the number of visits. The decay is usually linear or exponential, depending on whether the total duration is known in advance or if the task is finite or infinite.

Similarly to MCTS methods, recently also RL methods started to employ stronger

exploration policies [42], with some of these based on (contextual) bandits [51, 52].

Finding a good policy

Gathering samples and evaluating the value function by following a fixed control policy is known as *policy evaluation*. Refining the control policy based on previous evaluations (i.e., based on the value function) is known as *policy improvement*. Iterating through these two processes in alternation (Figure 3.3) makes up the paradigm of *generalized policy iteration (GPI)* [9]. A vast range of learning and search methods can be essentially described as GPI, including MCTS methods: in the selection phase the tree policy takes decisions based on previous estimates (i.e., policy improvement) and in the backpropagation phase current feedback is used to update the estimates (i.e., policy evaluation), repeating these steps in each iteration.

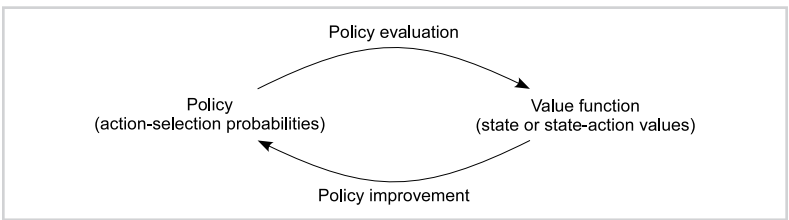

Figure 3.3

Generalized policy iteration [9]: in each iteration, the policy is evaluated to produce a value function and then, upon this value function, the policy is greedily improved.

When policy improvement is greedy with respect to the value function produced by policy evaluation, GPI converges towards the optimal policy and optimal value function. It is not necessary that the improvement is completely greedy – it is enough if the improved policy is only moved *towards* a greedy policy. The process can be run for an arbitrary number of iterations, producing an arbitrarily close-to-optimal policy and value function. Rephrasing from another perspective, the current policy and value function of GPI (or RL) algorithm can usually be retrieved at any moment, which is analogous to MCTS algorithms' ability to return an output in-between each iteration, that is, the *anytime* behaviour of MCTS methods. Lastly, increasing the number of GPI iterations usually results in a more optimal solution, the same as in MCTS.

Assigning value to actions

When improving a policy, the actions get compared to identify the better ones. Thus, they need a value to base the comparison on. This is achieved either by directly evaluat-

ing *state-actions* (Q-values) or by evaluating states (V-values) and assigning each action the value of the state it leads to. The latter is known as evaluating actions with *afterstates* [9] (or *post-decision states* [53]). For example, on deterministic tasks, this equals to $Q(s,a) \leftarrow V(s')$, where s' is the afterstate of action a from state s.

Evaluating with afterstates[2] improves the learning rate on tasks where multiple move sequences lead to the same position – it is useful on games with transpositions. However, when an algorithm does not identify transpositions, there is no difference between the two approaches. When the task is stochastic (i.e., the same action leads to different states) or allows simultaneous actions (e.g., in a multi-agent setting), evaluating actions with afterstates is more complex than evaluating directly state-actions – it is not enough to simply assign the value of an arbitrary (or last-observed) afterstate to the preceding action, but some average over all possible afterstates of that action is used instead by considering the transition probabilities.

MCTS methods can also be implemented either way. Such mechanics have been observed by several MCTS researchers: Childs et al. [31] distinguish between *UCT1* (evaluating state-actions) and *UCT2* (evaluating afterstates) as two variants of the UCT algorithm when used with transpositions; and Saffidine et al. [36] analyse the same concept when they compare storing feedback in graph edges to storing it in graph nodes, but do not mention the link to RL.

First-visit and every-visit updates

MCTS methods by default store the value function as a tree structure, whereas RL methods most often store it as a directed graph, i.e., they use transpositions by default. When storing it as a tree, each node in the trajectory is visited exactly once per episode; however, when storing it as a graph, a single node might be visited several times per episode. In such a setting, there are two common approaches for updating the value of a node in an episode: either multiple times per episode (once for each visits of that node), or only once per episode (for the first visit of that node). Algorithms that use the first approach are known as *every-visit*, while those that use the latter are known as *first-visit*. MCTS algorithms usually do not observe when was a state first visited – in the backpropagation phase they update their estimates for every occurrence up to the tree root. A first-visit MCTS algorithm would update only the closest-to-the-root

[2] This is not to be confused with Q-learning [49] – the use of afterstates is independent of the learning algorithm.

occurrence of a specific node, and would ignore its other occurrences. Updating multiple times may cause a significantly slower convergence due to possible cycles in a given task, hence every-visit algorithms are less robust and often perform worse than first-visit algorithms [9, 54]. Nevertheless, when transpositions are not used, then there is no difference between the two approaches.

Childs et al. [31] also recognized these issues when adapting UCT to use transpositions. They noted the relation to the RL theory and suggested to use its solutions – specifically, its first-visit mechanics – to tackle such problems in MCTS.

On-policy and off-policy exploration – evaluating one policy while following another

From an RL point of view, an agent might be involved with one or two control policies. In *on-policy* control the agent is evaluating and simultaneously improving the exact policy that it follows. Conversely, in *off-policy* control, the agent is following one policy, but may be evaluating another – it is following a *behaviour* policy while evaluating a *target* policy [42]. Only the behaviour policy must be exploratory, whereas the target policy can be deterministic or greedy. This is useful in, for example, planning tasks, where during the available simulation time it is important to keep exploring (to ascertain convergence towards the optimal solution), but at the end it is preferred to output the best action according to a greedy policy (to maximally exploit the gathered knowledge).

These mechanics directly relate to the MCTS backpropagation phase. On-policy MCTS algorithms backup the exact feedback values, keeping an average for each node (e.g., the basic UCT algorithm), whereas off-policy MCTS algorithms backup some other value instead, for example, the value of the best child, valuing a node by maximizing across children values (which is analogous to a greedy target policy). Coulom [4] analysed such backup methods along proposing the basic MCTS framework, but he did not derive from the RL theory. Also, Feldman and Domshlak [55] analysed the behaviour of MC search methods and found beneficial to treat separately the goal of evaluating previous actions and the goal of identifying new optimal actions, which they named as the principle of the *separation of concerns*. Following this principle, they are probably the first that explicitly differentiate between on-policy and off-policy behaviour in an MCTS setting. Their *MCTS2e* scheme could be understood as a framework for off-policy MCTS algorithms (with the addition of online model-learning).

Describing MCTS algorithms with RL terminology

Summarizing this section, the search dynamics of MCTS are comparable to those of *Monte Carlo control*. For example, the *original* UCT algorithm [8] searches identically as an *offline on-policy every-visit MC control* algorithm [9] that uses UCB1 as control policy; this similarity stems from the fact that the original UCT algorithm is based on MDPs. On the other hand, the *standard* UCT algorithm [5] also equals to the same RL algorithm, but when the latter is modified to perform naïve offline updates (i.e., to backup only the final sum of rewards in an episode, ignoring the information of when were individual rewards received), to not identify states (i.e., not use transpositions), and to employ an *incremental representation* of the state space, which we discuss later. Similar reasoning would apply to other MCTS algorithms as well. Although the two basic UCT variants employ a separate random policy in the playout phase, using only the UCB1 policy in both the tree and playout phases produces the same behaviour – candidate actions in the playout have no visits so far, so UCB1 chooses them at random. However, when the playout policy is not random, the (default) MCTS and RL learning procedures differ, as we further discuss in Chapter 4.

3.5 Temporal-difference learning

Evaluating policies with Monte Carlo sampling is only one way of solving the reinforcement learning problem. An alternative is to use *temporal-difference (TD) learning*, which often performs better [9].

Temporal-difference learning [11] is probably the best known RL method and can be understood as a combination of Monte Carlo (MC) methods and dynamic programming (DP). Like MC methods, it gathers experience from sampling the search space without requiring any model of the environment. Like DP, it updates state value estimates based on previous estimates, instead of waiting until receiving the final feedback. Updating estimates based on other estimates is known as *bootstrapping*, and it decreases the variance of evaluations and increases the bias, but usually improves the learning performance [9]. The term *temporal differences* derives from the fact that the *previous* value of the state in the next time step affects the *current* value of the state in the current time step.

Comparing Monte Carlo and temporal-difference updates

Monte Carlo methods usually compute the value of each visited state S_t as the average of returns G gathered from n episodes that have visited state S_t so far. When rewritten in incremental form, the state value *estimates* $V(s)$ get updated at the end of each episode (when $t = T$) by

$$V(S_t) \leftarrow V(S_t) + \alpha_n \Big[G_t - V(S_t) \Big], \quad \forall t < T\,, \tag{3.4}$$

where the *update step-size* $\alpha_n \in [0, 1]$ decreases with the number of visits to the node, normally as $\alpha_n = 1/n$. The first episode has $n = 1$, and $V_0(s)$ are also regarded as the *initial state values* $V_{\text{init}}(s)$. Monte Carlo methods normally have to wait until time T before updating the value estimates (since G is not available sooner) – they perform *offline* updates.

On the other hand, TD methods may perform *online* updates at each time step t by considering the predicted state value $R_{t+1} + \gamma V_t(S_{t+1})$ as the target instead of G_t:

$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \Big[(R_{t+1} + \gamma V_t(S_{t+1})) - V_t(S_t) \Big]. \tag{3.5}$$

The difference between the predicted state value $R_{t+1} + \gamma V_t(S_{t+1})$ and the current state value $V_t(S_t)$ is called the *TD error* δ_t:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)\,. \tag{3.6}$$

Eligibility traces

It is not necessary that a single TD error δ_t updates only the last visited state value V_t, as in Equation (3.5), but it can be used to update the values of several (or all) previously visited states. Such an algorithm *traces* which states were previously visited and gives them credit, that is, *eligibility*, regarding how far back (how long ago) they were visited. This is known as performing *n-step* backups instead of *one-step* backups; and the parameter defining the weight of the backed-up TD error is known as the *eligibility trace decay rate* $\lambda \in [0, 1]$. *Eligibility traces* E are memory variables that are updated for all states at each time step. Two popular update mechanics are the *accumulating eligibility traces*, where

$$E_t(s) = \begin{cases} \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t, \\ \gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t, \end{cases} \tag{3.7}$$

and *replacing eligibility traces*, where

$$E_t(s) = \begin{cases} \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t, \\ 1 & \text{if } s = S_t. \end{cases} \tag{3.8}$$

By using eligibility traces, state values can be updated online after each time step according to

$$V_{t+1}(s) = V_t(s) + \alpha_t \delta_t E_t(s), \quad \forall s \in \mathscr{S}, \tag{3.9}$$

or offline at the end of each episode as

$$V(s) \leftarrow V(s) + \alpha_n \sum_{t=0}^{T} \delta_t E_t(s), \quad \forall s \in \mathscr{S}. \tag{3.10}$$

Merging TD backups with eligibility traces results in arguably the most popular policy-evaluation method, TD(λ) [11]. When $\lambda = 0$, the algorithm updates only a single previous state, the same as given by Equation (3.5). Conversely, when $\lambda = 1$, it updates all states, and with the same TD error – it produces exactly the same backups as Monte Carlo methods by Equation (3.4) [9]. Thus, the TD(λ) method may be viewed as a generalization of MC methods, where accumulating traces are related to every-visit MC and replacing traces to first-visit MC. In this way, eligibility traces augment MC methods with bootstrapping. Figure 3.4 illustrates the differences (and similarities) between Monte Carlo and temporal-difference backups.

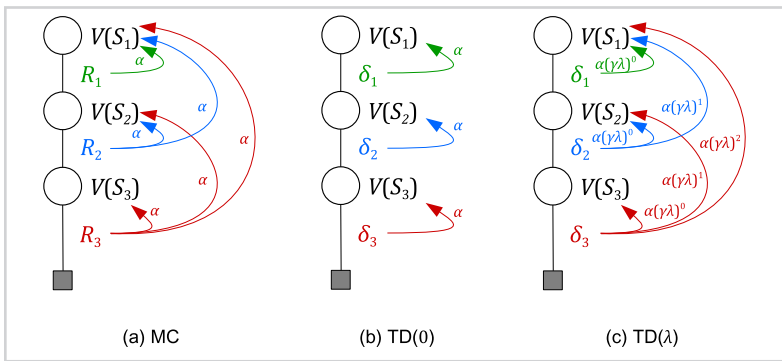

Figure 3.4

Comparison of Monte Carlo and temporal-difference backups: (a) MC updates every state-value estimate $V(s)$ towards the rewards R collected after visiting that state (with a step-size α), (b) TD(0) updates an estimate only with the following TD-error δ, and (c) TD(λ) updates every estimate with all following TD-errors, decayed with γ and λ (eligibility). The grey squares represent terminal states.

Temporal-difference control and Monte Carlo tree search

Reinforcement-learning methods that evaluate and improve policies with bootstrapping TD-learning backups instead of Monte Carlo backups are known as *TD control* methods. These are a generalization of Monte Carlo control methods (due to $\lambda = 1$ resulting in equal backups, as described earlier). A popular on-policy[3] TD control algorithm that uses eligibility traces is the *Sarsa(λ)* algorithm [14], which we later employ to extend the UCT algorithm (Section 4.3). The most popular off-policy TD control method is *Q-learning* [49], and its use with eligibility traces – $Q(\lambda)$.

In Section 3.4 we observed that many of the MCTS search dynamics can be described by MC control. Given that TD control generalizes MC control, it is also strongly related to MCTS. In fact, MCTS methods that evaluate nodes by averaging the outcomes produce backups that equal those of a TD(1) algorithm, and their four MCTS phases resemble an iteration of a Sarsa(1) algorithm. Conversely, MCTS methods evaluate nodes differently (e.g., that back up the maximum value of children nodes [4]) resemble off-policy TD methods like Q(1), for example.

In the context of MCTS, Childs et al. [31] partially analysed bootstrapping dynamics soon after the invention of this field, although without mentioning their relation to temporal differences and RL theory. They presented several methods of backpropagation when adapting the UCT algorithm to use transpositions: their *UCT3* algorithm updates the estimates according to previous estimates – it bootstraps similarly as TD(0).

3.6 The novelties of Monte Carlo tree search

Although the strong similarity of MCTS and RL methods has been observed by several RL researchers (Section 5.1), to our knowledge, the novelties of MCTS methods in light of RL have rarely been emphasized. As shown so far, many of the MCTS dynamics can be described with RL theory; despite the different points of view, the underlying concepts are in practice very similar. Silver [41] explicitly noted that "once all states have been visited and added into the search tree, Monte-Carlo tree search is equivalent to Monte-Carlo control using table lookup." However, what happens *before* all states have been visited? How do RL and MCTS methods relate until then? These questions lead us to the key difference between these two classes of methods, to the aspects of MCTS that we have not linked with RL yet – the MCTS concepts of *playout* and

[3] Not to confuse *on-policy* control with *online* backups (Section 3.4).

expansion. We argue that these two concepts are not not commonly seen as standard RL procedures and they seem to be (alongside the constant replanning) the key innovations of MCTS in relation to RL theory.

Finding goal states or solutions in large state spaces may lead to intractable problems, either in computational or memory needs, when approached with naive or brute-force algorithms. Complete exploration of such spaces is often impossible, but, on the other hand, there is danger of local optimality if the algorithm is not exploratory enough. There are two main memorization approaches when solving tasks where it is infeasible to directly memorize the whole state space due to its size: (1) describing the whole state space by approximation, or (2) keeping in memory only a part of the state space at once.

The first is the leading approach in RL algorithms, which usually employ *value function approximation* techniques. These may also use domain-specific features, which might considerably increase performance if the feature set is carefully chosen. Good quality features, however, might not be readily available, as has been the case with the game of Go for many years – though recent developments seem to imply that features can be learned on the fly [6]. *Tabular* RL algorithms, which do not use approximation, but instead memorize each value with a single table entry, are often inefficient on large tasks.

The second approach is typical for search methods, especially for MCTS methods [5]. These are usually tabular, but memorize only the part of the state space that is considered most relevant. This relevancy is sometimes defined with heuristic criteria, but most often it is directly entrusted to the selection policy – an MCTS method usually expects that its tree policy will guide the exploration towards the most relevant part of the state space, which should as such be worth memorizing. The memorization is handled in the MCTS *expansion phase*. There has been plenty of research on such memorization techniques, ranging from replacement schemes for transposition tables [35], which have already been thoroughly studied in the α-β framework [56], to MCTS expansion enhancements, such as progressive unpruning [57] and progressive widening [58]. The latter two are also related to the first play urgency concept [38].

The approaches used by MCTS methods may be generally described as *mechanics for changing the state-space representation online*, that is, changing the architecture or size of the memorized model simultaneously to learning or planning from it. In this sense, the basic MCTS framework foresees the use of a direct-lookup table (representing ei-

ther a tree or a graph) that expands in each iteration – that gets *incremented online*. Such concepts have been receiving high attention over the last decades (also before the invention of MCTS) in the field of machine learning in general. They are strongly related to methods like incremental decision trees [59], adaptive state aggregation [60], automatic online feature identification [61], online basis function construction [62], incremental representations [10], etc. We will refer to such methods as *adaptive representations*.

Within the RL community, the use of adaptive representations is decades-old [63], but recently it gained even more prominence through the successes of Go [6]. Also, advances in representation learning and global function approximation (which, for example, resolve catastrophic forgetting [64]) led to the popularization of neural networks [65, 66]. Along the same lines, we have seen the introduction of unified neural-network architectures that imbue agents with even better exploration capabilities inspired by the notions of artificial curiosity and novelty-seeking behaviour [67, 68]. Evolutionary methods have also seen a resurgence [69]; these too have a long history within RL contexts [70]; they are less sensitive to hyperparamaters, initial features, and reward shaping. Finally, as an evolution of Dyna-like architectures [71], Silver et al. [72] explore how learning and planning can be combined into a single end-to-end system. Neural network are global function approximators, where a single update can potentially change the value/policy function for the entire state space. There has been a number of methods developed [73–75] where the aim is to learn a set of basis functions that capture a higher-level representation of the state space, which might potentially allow for more "local" higher level features. We have also seen more explicit cases of local function approximations [76, 77], where an update changes the value/policy function only for a (local) part of the space.

Despite the research, we are not aware of RL methods that employ explicit (tabular) adaptive or incremental representations as in MCTS methods. Also, the latter in general memorize and update only a subset of the visited states – they explicitly distinguish between memorized and non-memorized parts of an episode. Although the RL theory distinguishes between well-explored and unexplored states – through the KWIK learning formalism [78–80] – it is (usually) not familiar with the notion of non-memorized (i.e., non-represented) parts of space. Therefore, it does not recognize the *playout phase* and neither the use of a separate policy for it, such as the MCTS *default policy*. Lastly, although classic RL theory can describe a random MCTS default policy (if we assume

the control policy to select random actions when in absence of estimates), it can hardly describe more complex MCTS default policies, which are customary for strong MCTS algorithms.

3.7 Survey of MCTS enhancements that relate to RL

We conclude our linking of MCTS and RL with an overview of the numerous MCTS enhancements that are not directly inspired by RL theory, but still exhibit similar dynamics as traditional RL methods. These enhancements are mostly related to complex backups, weighting and discounting techniques, biasing the estimates with prior, initial, or heuristic knowledge, and forgetting techniques.

Max-backups

As noted previously, MCTS-like algorithms that backup maximum values instead of means are related to off-policy TD learning algorithms; they most often resemble the Q-learning algorithm [49]. Coulom [4] was the first to experiment with this in an MCTS setting and observed that backing up the means is better when there are fewer visits per state, whereas using some type of max-backups is better when the number of visit is high.

Ramanujan and Selman [81] develop the $UCTMAX_H$ algorithm – an adversarial variant of UCT that uses a heuristic board evaluation function and computes minimaxing backups. They show that such backups provide a boost in performance when a reasonable domain-specific heuristic is known.

Lanctot et al. [82] also use minimaxing for backing up heuristic evaluations; however, they propose not to replace playouts with such evaluations, but rather to store the feedback from both as two separate sources of information for guiding the selection policy. This is similar to how AlphaGo [6] and the studies leading to it (Chapter 5) combine prior knowledge with online feedback. The authors show this might lead to stronger play performance on domains that require both short-term tactics and long-term strategy – the former gets tackled by the minimax backups, and the latter by the ordinary MCTS playout backups. Baier and Winands [83] provide similar conclusions also for several MCTS-minimax hybrid approaches that do not require evaluation functions.

Reward-weighting

The discounting mechanics of eligibility traces in RL decrease a reward's weight proportionally with the distance from the state it was received. In the context of MCTS, this translates to weighting the iterations according to their duration, e.g., diminishing the weight of rewards with increasing distance to game end. To our knowledge, there has been no proper implementation of eligibility traces in the complete MCTS framework (including the non-memorized playout phase) yet; however, the MCTS reward-weighting schemes noted here could classify as simplified or similar approaches.

Cowling et al. [84] exponentially decrease the final reward based on the number of plies from the root to the terminal state with a discount parameter in range $[0, 1]$. This results in all nodes in the tree being updated with the same value, regardless of each nodes' distance to the terminal state – this differs from eligibility-trace mechanics, which keep decaying rewards also within the tree, in this way updating each visited node with a different value. Their experiments show benefits of such decaying.

Kocsis and Szepesvari [8] diminish the exploration rate in the UCT algorithm when descending the tree (i.e., with increasing tree depth). Such decaying of the UCT's parameter C_p achieves a similar effect as diminishing the weight of node-estimates closer to the root, and is in turn similar to diminishing the weight of those memorized rewards that are far from the terminal position. Such weighting is more limited than the eligibility trace mechanism, because it disregards that the rewards stored in a single node may have been gathered from episodes of different duration – nodes at the same tree level may have different distances to the terminal position. Therefore, the rewards are not differentiated according to the game duration, but rather according to the distance from the starting position.

Xie and Liu [85] develop a similar mechanism by modifying the MCTS backpropagation step – they weight MCTS iterations according to their recency. They assume that early iterations contain less reliable value estimates, hence should be given a lower weight than later iterations. They partition the sequence of iterations and weight differently each partition. Their results show that later partitions deserve a higher weight, since these generally contain shorter iterations, which are more accurate.

Initial bias, heuristics, and forgetting

In the MCTS selection phase, nodes (i.e., states) in the tree are compared by their value and selected accordingly. Adhering to the basic framework, values for unvisited states are not defined yet. This may cause the selection mechanism to be unpredictable. Hence, it is sensible to assign a value to nodes prior to their first visit – assign them an initial bias – and later overrule this bias with the estimates produced by the tree policy, e.g., overrule the bias with UCB1 values. This is analogous to the use of initial values V_{init} and the update step-size parameter α in RL methods (Section 3.5); the initial values correspond to the values that are assigned to nodes before their first visit, and the update step-size defines the rate at which the initial values get overruled by the new estimates – it controls the rate of "forgetting" past knowledge.

In the context of MCTS, Gelly and Wang [38] define the idea of assigning initial values as the *first-play urgency*. They fully replace the initial bias immediately after the first visit of a node, and show that this improves the performance of the UCT algorithm. Chaslot et al.[57] propose to calculate the bias with a heuristic evaluation function and decrease its impact when the state gets more visits. They label this as the *progressive bias* enhancement for MCTS methods; this is analogous to a continuous transition (instead of a discrete one) from the first-play urgency to the value learned by the tree policy.

Several MCTS enhancements set the initial or progressive bias according to heuristic or previously-learned knowledge. A widespread approach is to use *history heuristics* [86], i.e., the information from previously-played moves, to improve the tree policy and the playout policy. Kozelek [18] distinguished the use of such information either on a *tree-tree level* or a *tree-playout level* (depending in which MCTS phases the information is collected, and then used). He employed history heuristics in the selection phase to significantly improve the performance of MCTS methods on the game Arimaa. Finnsson and Bjornsson [23] used it in form of the *move-average sampling technique (MAST)* for seeding node values also in the playout phase in *CadiaPlayer*. Gelly and Silver [28] improved MCTS in a similar way by setting the initial values of unvisited nodes in the tree to the same value as their grandfathers and labelled this as the *grandfather heuristic*; however, they observe the improvement is smaller than using values learned offline by reinforcement learning methods. Nijssen and Winands [87] propose the *progressive history* enhancement – a type of progressive bias that uses history

score as the heuristic bias; a node's history score is expressed as the average win rate of all the episodes that visited that node. Their enhancement proves helpful in multi-player games. Other popular enhancements, such as *all moves as first (AMAF)* [38], *rapid action value estimation (RAVE)* [39], and *last good reply (LGR)* [88], are also related to history heuristics.

Beside the progressive bias [57] enhancement (described above), several other "forgetting" dynamics have been applied to MCTS. The most basic and popular one is discarding the tree (partially or completely) after each search [5], as already described in Section 2.2. Hashimoto et al. [89] proposed a backup operator that assigns higher importance to recently visited actions (and lower importance to other actions). Their *accelerated UCT* algorithm selects actions according to *accelerated winning rates*. The latter are computed through the notion of a forgetting "velocity", which defines the decaying rate individually for each value estimate based on the recency of its visit; its update scheme is similar to those used for RL eligibility traces (Section 3.5). They study the benefits of such an approach on several two-player games and observe it improves the strength of the Computer Go algorithm Fuego [90].

Baier and Drake [91] applied forgetting not directly to the estimates, but indirectly through the one-move and two-move LGR playout policies [88], which remember and then prioritize moves that previously led to a win in response to an opponents move (or combination of moves). They extend LGR policies to forget moves that (after being memorized) lead to a loss; this proved beneficial on Go.

Tak et al. [92] also observe the benefits of forgetting the estimates produced by domain-independent playout policies: they propose the application of a decay factor to the *n-gram selection technique (NST)* [93] and to MAST [23] (both used by the General Game Playing champion program CadiaPlayer [24]). They analyse three decaying schemes: *move decay* (previously studied in combination with NST by Stankiewicz [94]), *batch decay*, and *simulation decay*. Their experiments show that decaying significantly improves the performance of such playout policies on several types of games, except on single-player games. Their extension of the NST policy with a decay factor generalizes the forgetting LGR policies studied by Baier and Drake [91]; the latter behave as using a maximal decay factor, completely forgetting the previous estimate.

Feldman and Domshlak [55] developed a forgetting variant of their *best recommendation with uniform exploration (BRUE)* algorithm: the resulting *BRUE*(α) algorithm could be linked to *constant-α Monte Carlo methods* [9], because its parameter α oper-

ates in a similar way as the learning rate α used in RL algorithms, for example, as in Equation (3.4).

4

Merging Monte Carlo tree search and reinforcement learning

In the previous chapters we covered the strong connection between Monte Carlo tree search (MCTS) and reinforcement learning (RL) methods. We explained how RL theory offers a rich description of the MCTS backpropagation phase, whereas MCTS introduces into RL the distinction between a memorized and non-memorized part of the state space. Following these insights, we present a line of reasoning that extends both MCTS and RL into a single framework that unifies the advantages of both fields. As a proof of concept we showcase an algorithm that generalizes UCT with TD learning in such a way and link its new parameters with existing MCTS enhancements. We also introduce an efficient value-normalization technique for the requirements of UCB selection policies when used with general RL methods. Lastly, to encourage a faster adoption of our methods, we provide extensive implementation guidelines.

4.1 Extending the reinforcement learning theory

To develop a framework that could describe the incremental representations used by MCTS algorithms, we first require an extension of RL theory with novel MCTS concepts – we need the RL theory to acknowledge a *non-represented* (i.e., non-memorized) part of the state space – this is the part that is not described (estimated) by the representation model at a given moment. Based on this we introduce into RL the notions of a representation policy and a playout value function.

Representation policy

A *representation policy* defines how is the representation model of the state space (or value-function) adapted online; it defines the boundary between the memorized and non-memorized parts of the state space and how it changes through time. The policy can dictate either a fixed or adaptive representation. For example, in TD search applied to Go [41] the representation policy keeps a fixed size and shape of the feature-approximated state space, whereas in the standard UCT algorithm [5], it increments the lookup table (tree or graph) by one entry in each MCTS iteration and discards it after each batch of iterations.

Sample-based (RL and MCTS) search algorithms could be understood as a combination of a learning algorithm, a control policy, and a representation policy. These define how the estimates get updated, how actions get selected, and how is the underlying representation model adapted, respectively. Incremental representations, as used in MCTS, are only one type of adaptive representations, as described previously

(Section 3.6). In this sense, the representation policy can also be understood as a generalization of the MCTS expansion phase (which defines when and how to expand the tree in MCTS algorithms). Also, it introduces into RL the notion of a *playout*, which is the part of a trajectory (in an episode) where states and actions have no memorized estimates and hence cannot be updated.

The notions of a representation policy and a non-represented part of the state space extend beyond tabular representations also to approximate representations (e.g., value-function approximation). In fact, using an adaptive representation is not exclusive with function approximation, but it is complementary – approximate models can also be made adaptive. This offers another dimension of variation: for example, on one end, an incremental approximate representation can weakly describe the whole state space and get more features added with time, in this way improving the overall approximation accuracy; whereas, on the other end, it can initially approximate only a part of the state space (with high accuracy) and then use the newly-added features to extend the representation to previously non-represented states (instead of improving the accuracy). The first example is more common to RL, whereas the second is more common to MCTS, but an algorithm can be configured to anything in-between. Following all the above, when function approximation is used, the representation policy not only defines the boundaries between the memorized and non-memorized state space, but it also impacts how will the approximation accuracy change over time.

Assumptions about playout estimates

The notion of a non-represented part of the state space is common to MCTS algorithms and can also be easily handled by RL algorithms that perform Monte Carlo backups: in each episode the algorithm updates the represented (memorized) estimates with the return G, as in Equation (3.4), and skips the update on the non-represented estimates in the playout. On the other hand, more general (online and offline) RL algorithms that perform TD(λ) backups – which update the estimates with TD errors δ, as by Equations (3.9) or (3.10), and not directly with the return G – might have difficulties in computing the TD errors in the non-represented part of the state space, because the value estimates $V_t(S_{t+1})$ and $V_t(S_t)$ required by Equation (3.5) are not available. A solution is to avoid computing TD backups in the playout phase by treating the last-encountered represented state (i.e., the tree-leaf node) as a terminal state and then observe and backup only the discounted sum of rewards collected afterwards. This

should be easy to implement, although we consider it is less principled, as it formally requires to change the TD-backup rules according to the current position in the state space.

As an arguably more principled alternative solution, we suggest to make assumptions on the missing values of playout states (and actions) in a similar way as assumptions are made on the initial values $V_{\text{init}}(s)$ and $Q_{\text{init}}(s,a)$. We introduce the notion of a *playout value function*, which replaces the missing value estimates in the non-memorized part of the state space with *playout values* $V_{\text{playout}}(s)$ and $Q_{\text{playout}}(s,a)$. The employed (or assumed) playout value function is arbitrary, but it impacts the control policy, because the latter makes decisions also based on the values in the playout. For example, assuming equal playout values for all states would result in a policy to select random actions in the playout (as in the default MCTS setting). We refer to the part of an RL control policy that bases its decisions on playout values (rather than on the ordinary values) as the *playout policy*.

By the default MCTS setting, the playout value function has no memory to store estimates: in such case it can be regarded as a rule-based assumption on the values of the states (and actions) encountered in the playout – in this way it can serve as an entry point for evaluation functions, heuristics, and expert knowledge about the given task. On the other hand, the playout value function can also be configured to use memory; in such case it could be regarded as an additional, secondary representation of the state space. In this way it can recreate playout-related MCTS enhancements that generalize the gathered experience through low-accuracy approximation of the state space, such as storing the values of all actions regardless of where they occur in the state space (i.e., the MAST enhancement [23]), for example.

Domain-specific knowledge or heuristics inserted through playout values might produce more informed backups and speed up the convergence – this is probably the best option in practice. However, when such knowledge is not available, more basic assumptions must be made. When the reward discount rate $\gamma = 1$, a natural assumption that performs well (by experimental observation, Section 6) is to set playout values to equal the initial values: $V_{\text{playout}}(s) = V_{\text{init}}(s)$ for all states s. In such case the effect of both values is similar – they emphasize exploration when set optimistically (Section 4.5) and vice versa. Otherwise, when $\gamma < 1$, to avoid an undesired accumulation of TD errors in the playout, the easiest solution is to set them to equal 0; although this produces a TD error towards 0 at each transition from the memorized part of the space

(the MCTS tree phase) into the non-memorized part (the playout). A more complex, but arguably better option is to set the first playout value to match $V(S_{\text{leaf}})/\gamma$, where S_{leaf} is the last-encountered memorized state (the tree-leaf node), and then divide each next playout value with γ – in such case the TD errors would be produced only by rewards R_t, indifferently of the configuration of the algorithm. Regardless of the chosen playout value function, an RL algorithm would converge as long as the primary representation (i.e., the tree in MCTS) keeps expanding towards the terminal states of a given task, thus reducing the length of the playout phase towards zero in the limit.

From a theoretical perspective, we regard playout values as a by-product of acknowledging a non-memorized part of the state space and using adaptive representation policies. A formal analysis of how such values generally impact RL algorithms is not in our focus, here we only suggest them as a viable alternative to ignoring the issue of missing estimates in the playout. To summarize, although in theory the assumed playout values fill the place of missing value estimates for TD-like backups, in practice they can be regarded more as a placeholder (entry point) for expert knowledge about the given problem domain.

4.2 *The temporal-difference tree search framework*

Based on the extended RL theory from the previous section, we introduce the *temporal-difference tree search (TDTS)* framework: a generalization of MCTS that replaces Monte Carlo (MC) backups with temporal-difference (TD) backups and eligibility traces (Figure 4.1). From an RL point of view, TDTS describes sample-based TD control algorithms that use eligibility traces (including MC control algorithms when $\lambda = 1$) and recognize the novel MCTS-inspired notions that we introduced previously – mainly the non-memorized part of the state space and a representation policy for adaptive (incremental) models. Although the TDTS framework is inspired by planning tasks, where the experience is simulated, it can also be applied to learning tasks (in case a model is not available).

The framework encompasses both online and offline backups, first-visit and every-visit updating, and on-policy and off-policy control. It can be used on top of any kind of representation policy: constant or adaptive, exact or approximate (tree, graph, tabular, function approximation, etc.). With the help of a representation policy and playout value function, it can fully recreate the four MCTS iteration phases: (1) selection – control in the memorized part of the search space; (2) expansion – changing the

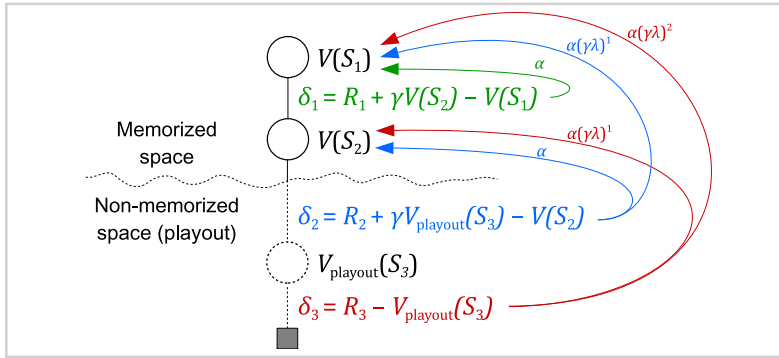

Figure 4.1

Temporal-difference tree search (TDTS) backups. Unlike traditional RL methods, the TDTS framework distinguishes parts of the state space where estimates are not memorized (and not updated). Unlike traditional MCTS methods, TDTS employs TD-learning update mechanics (Section 3.5) instead of Monte Carlo backups.

representation; (3) playout – control in the non-memorized part of the search space; and (4) backpropagation – updating the value estimates.

In general, RL can empower MCTS with strong and well-understood backup (i.e., learning) techniques. Considering this, since the TDTS framework replaces MC backups in MCTS with bootstrapping backups, our further analysis and evaluation primarily focuses on the benefits of this replacement – on the comparison of such backups in the default MCTS setting. Therefore, from here on we assume that the analysed TDTS algorithms are by default configured to match MCTS – to perform offline backups and use a direct-lookup table (representing a tree or a graph) with the representation policy expanding it in each iteration. This allows an easier comparison with traditional MCTS algorithms (such as UCT), and for an easier extension from MCTS to TDTS (in Section 5.2 we further justify why we deem this important). Despite the configuration, our analysis and experiments would apply similarly also to more complex representations (for example, value-function approximation) and to online updates, as previously noted.

4.3 The Sarsa-UCT algorithm

As an instance of a TDTS method, we combine Sarsa(λ) and UCT into the *Sarsa-UCT(λ)* algorithm. This is an on-policy generalization of UCT with TD backups and eligibility traces, as well as a generalization of Sarsa(λ) with an incremental representation policy and playout values. We choose UCT, because it is the most popular and well-studied MCTS algorithm – it forms the backbone of several MCTS enhance-

ments and game-playing engines; improving its base performance might improve the performance of several algorithms that depend on it. On the other side we choose Sarsa(λ), because it is the canonical on-policy TD (and RL) algorithm, and because it can fully reproduce the behaviour of the original UCT algorithm (as explained in Section 3.5). Furthermore, it has less requirements for convergence compared to off-policy RL methods (for example, compared to Q-learning).

Algorithm 1 presents the Sarsa-UCT(λ) iteration when it builds a tree representation and evaluates actions with afterstates (hence the use of V-values instead of Q-values). The essential difference with the *standard UCT* algorithm (Section 2.3) is the backpropagation method (lines 29–49); however, the algorithms behave equally when $\lambda = \gamma = 1$, and thus can be used interchangeably in any of the existing UCT-enhancements. A re-improvement over the standard UCT is distinguishing the intermediate rewards (lines 17–18), just like in the original UCT. This might be omitted when the cost of observing rewards (e.g., computing the score of a game) after each time step is too high or when in the given task there are only terminal rewards; the algorithm would work correctly also if only the terminal outcomes were observed, just with a slower convergence rate.

The *eligibility-trace* mechanism is efficiently computed with decaying and accumulating the TD errors during the backpropagation phase (line 40). Such implementation results in *every-visit* behaviour (when the algorithm is enhanced with transposition tables) and in *offline* backups at the end of each episode. A *first-visit* variant updates the state values only for their first visits in each episode (which is often beneficial, as explained in Section 3.4): such code would extend each *episode* entry (line 18) with a first-visit flag, and check for it before backing up (line 41). Furthermore, performing *online* backups (i.e., after each time step) when using transpositions might increase the convergence rate in exchange for a higher computational cost; it is a reasonable trade-off when states get revisited often in the course of an episode or when the cost of simulating the task is orders of magnitude higher than performing backups. When used with transpositions, the UCB's exploration bias (line 54) must be modified to consider the sum of children visit counters $\sum n(S_i)$ instead of the parent's counter $n(s)$; this is necessary for the UCB1 policy to work correctly [13]. Additional details about implementing such algorithms with transpositions and offline updates are given in Section 4.6.

The algorithm has no need to memorize the probabilities of selecting actions $\pi(a|s)$,

Algorithm 1

An iteration of a tabular offline Sarsa-UCT(λ) algorithm that builds a tree and evaluates actions with afterstates. Produces offline backups and every-visit behaviour. The parameters V_{init} and V_{playout} can be constants or functions mapping states to values.

parameters: C_p, V_{init}, V_{playout}, γ, λ, α (optional)
global tables / data structures: $tree, V, n$ ▷ the memorized experience
procedure SarsaUCTIteration(S_0)
    $episode \leftarrow$ GenerateEpisode(S_0)
    ExpandTree($episode$)
    BackupTDErrors($episode$)
end procedure
procedure GenerateEpisode(S_0)
    $episode \leftarrow$ empty list
    $s \leftarrow S_0$ ▷ initial state
    while s is not terminal
        if s is in *tree* ▷ selection phase
            $a \leftarrow$ UCB1TreePolicy(s)
        else ▷ playout phase
            $a \leftarrow$ RandomPlayoutPolicy(s)
        end if
        $(s, R) \leftarrow$SimulateTransition(s, a)
        Append (s, R) to *episode*
    end while
    return episode
end procedure
procedure ExpandTree($episode$)
    $S_{\text{new}} \leftarrow$ first s in *episode* that is not in *tree*
    Insert S_{new} in *tree*
    $V(S_{\text{new}}) \leftarrow V_{\text{init}}(S_{\text{new}})$ ▷ initialize value estimate
    $n(S_{\text{new}}) \leftarrow 0$ ▷ initialize visit counter
    InitNormalizationBounds(S_{new})
end procedure

```
29: procedure BackupTDErrors(episode)
30:     δ_sum ← 0                                   ▷ cumulative decayed TD error
31:     V_next ← 0
32:     for i = Length(episode) down to 1
33:         (s, R) ← episode(i)
34:         if s is in tree
35:             V_current ← V(s)
36:         else                                    ▷ assumed playout value
37:             V_current ← V_playout(s)
38:         end if
39:         δ ← R + γV_next − V_current             ▷ single TD error
40:         δ_sum ← λγδ_sum + δ                     ▷ decay and accumulate
41:         if s is in tree                         ▷ update value
42:             n(s) ← n(s) + 1
43:             α ← 1/n(s)                          ▷ MC-like step-size
44:             V(s) ← V(s) + αδ_sum
45:         end if
46:         V_next ← V_current
47:         UpdateNormalizationBounds(δ_sum, s)
48:     end for
49: end procedure
50: procedure UCB1TreePolicy(s)
51:     for each s --a_i--> S_i                     ▷ i.e., for each afterstate S_i
52:         if S_i is in tree
53:             V_norm ← GetNormalizedEstimate(S_i)
54:             Q_UCB(s, a_i) ← V_norm + C_p sqrt(2 ln(n(s)) / n(S_i))
55:         else
56:             Q_UCB(s, a_i) ← ∞
57:         end if
58:     end for
59:     return argmax_a (Q_UCB(s, a))
60: end procedure
```

because these are implicitly determined by the UCB1 policy (line 59) at each time step: the policy assigns a probability 1 to the action with the highest value $Q_{\text{UCB}}(s, a)$ and probabilities 0 to the remaining actions. When multiple actions are best, they are given equal probability (i.e., random tie-breaking). Our algorithm employs a novel value-normalization technique (lines 27, 47, and 53) that enables the use of the UCB1 policy with TD learning. We detail it in the next section.

The computational cost of Sarsa-UCT(λ) is nearly equal to that of UCT, larger only by a small constant factor that is insignificant in practice. The increase is due to the accumulation of TD errors during backups and due to value-normalization before applying the UCB1 policy. The number of backups per episode is still $O(M)$, where M is the number of memorized nodes in the episode (i.e., the length of the tree traversal in the MCTS selection phase). An optimization that is not presented in our pseudocode is to omit memorizing individual states and rewards in the playout (lines 17–18) and instead compute only the playout-part of the cumulative TD error δ_{sum}, which can later be used as the initial value in the backup procedure (line 30).

The variant of Sarsa-UCT(λ) that is given in Algorithm 1 is designed for single-player tasks with full observability (with perfect information). It works both on deterministic and stochastic tasks. Adapting the algorithm to other types of tasks is as easy (or as difficult) as adapting the basic UCT algorithm – the same methods apply. For example, a straightforward implementation of minimax behaviour for two-player sequential zero-sum adversary games is to compute Q-values (line 54) with $-V_{\text{norm}}$ instead of V_{norm} when simulating the opponent's moves; we use this approach when evaluating the algorithm on classic games (Section 7.1). A general approach for multi-player tasks is to learn a value function for each player – to memorize a tuple of value estimates for each state and update these with tuples of rewards. Extending to games with simultaneous moves can be done by replacing the action a with a tuple of actions (containing the actions of all players). Lastly, a simple way for applying it to partially observable tasks (with imperfect information) is to build the tree from selected actions a instead of observed states s, this way ignoring the hidden information in the observations (and also the stochasticity of the task); we used this approach when implementing a Sarsa-UCT(λ) player for the General Video Game AI competition (Section 7.2).

4.4 Space-local normalization of value estimates

To guarantee convergence, UCB selection policies (including UCB1) require that the value estimates ($V(s)$ or $Q(s,a)$) are in range $[0,1]$, as noted in Section 2.3. This constraint can be satisfied by normalizing the estimates or by setting the weighting parameter C_p to an upper bound with regards to the minimum and maximum possible value of the estimates – the two approaches are mathematically equal, according to Equations (2.1) and (2.3).

Common techniques

When performing MC backups ($\lambda = 1$) without reward-discounting ($\gamma = 1$), as in many MCTS methods, on finite tasks with rewards only in terminal states this can be easily solved by setting the rewards in the desired range – e.g., many classic games can return a reward of 1 for a win, 0.5 for a draw, and 0 for a loss. When the rewards cannot be changed, then the minimum and maximum possible final rewards can be used as normalization bounds (or for setting the parameter C_p accordingly).

When the time of occurrence of non-terminal rewards is ignored, like in the *standard UCT* algorithm [5], the same backup value is used to update the estimates of all visited states in an episode. In such case, it is easier to guarantee convergence, because all the estimates can be normalized with the same bounds – the parameter C_p can be equal across all states and actions (the simplicity of this approach might be one reason why the standard UCT variant became more popular than the original one).

But, when the time of occurrence of rewards is considered, as is customary for RL methods, then the optimal bounds may differ across the state-space when the given task includes non-terminal rewards. In such case, on tasks with non-terminal rewards, but still in range $[0,1]$, the rewards can be divided with the maximum possible duration $T_{\max}$ of an episode to ascertain that all possible returns are in range $[0,1]$. This equals to multiplying C_p with $T_{\max}$ and using this "worst-case" normalization bound across all states. This was proposed alongside the *original UCT* algorithm [8]; however, it is rarely used in practice, because it often produces too wide bounds on large parts of the search space; the high value of C_p makes the learning algorithm too exploratory for most applied domains – it slows down the convergence rate.

A novel technique

An alternative to the approach above is adapting C_p to individual parts of the search space (i.e., to each state or state-action) online, according to the reward distribution in the given task. Such an approach is efficient even in the general RL setting, where the true distribution of rewards may be unknown (and not necessarily in range $[0, 1]$), and where both λ and γ may be less than 1, due to the use of TD backups and eligibility traces, as in our TDTS methods. To our knowledge, there has been no significant research on such techniques yet.

Following this line of thought, we devised a value-normalization method that computes the normalization bounds individually for each part of the search space and adapts them online by considering the rewards observed so far. We refer to this method as the *space-local value normalization* (Algorithm 2).

For each state (tree node or table entry), beside storing the value estimate (V or Q) and the number of visits n, the method stores also a *local* lower and upper bound of the value estimate of that state: $b_{\text{lower}}(s)$ and $b_{\text{upper}}(s)$, respectively. These bounds are used to normalize the value estimate in the range $[0, 1]$ before passing it to the UCB selection policy (Algorithm 1, lines 53–54). The method stores also a *global* lower and upper bound over all the estimates: B_{lower} and B_{upper}, respectively. The global bounds are used instead of the local bounds when the latter are not yet valid – for example, in states with zero visits. In the case when also the global bounds are invalid, normalization cannot be performed.

The global bounds are updated at each backup to memorize the minimum and maximum cumulative discounted TD-error δ_{sum} (Algorithm 1, lines 39–40) that was encountered so far – they remember the all-time minimum and maximum target value towards which any estimate was ever updated (the all-time minimum and maximum discounted return G could also be used instead). These bounds can be initialized according to prior knowledge (or heuristics) of the given task, e.g., to 0 and 1 for classic games with win/lose/draw outcomes.

The local bounds for each state value estimate are updated at each backup (Algorithm 1, line 47) to remember the all-time minimum and maximum value estimates of the children nodes (i.e., the one-step successor states). This is meaningful, because the value estimates already include the information about rewards in specific parts of the search space (due to RL backup rules). Furthermore, they also indirectly supply

Algorithm 2

The space-local value normalization used in the Sarsa-UCT algorithm.

global variables: $B_{\text{upper}}, B_{\text{lower}}$ ▷ set arbitrary initial values
global tables: $b_{\text{upper}}, b_{\text{lower}}$
procedure InitNormalizationBounds(s) ▷ initialize local bounds
    $b_{\text{upper}}(s) \leftarrow -\infty$
    $b_{\text{lower}}(s) \leftarrow +\infty$
end procedure
procedure GetNormalizedEstimate(s)
    if $b_{\text{lower}}(s) < b_{\text{upper}}(s)$ ▷ use local bounds
        return $\big(V(s) - b_{\text{lower}}(s)\big) / \big(b_{\text{upper}}(s) - b_{\text{lower}}(s)\big)$
    else if $B_{\text{lower}} < B_{\text{upper}}$ ▷ use global bounds
        return $\big(V(s) - B_{\text{lower}}\big) / \big(B_{\text{upper}} - B_{\text{lower}}\big)$
    else ▷ both bounds invalid
        return $V(s)$
    end if
end procedure
procedure UpdateNormalizationBounds(δ_{sum}, s)
    if $G > B_{\text{upper}}$ ▷ global bounds
        $B_{\text{upper}} \leftarrow \delta_{\text{sum}}$ ▷ G can also be used instead of δ_{sum}
    end if
    if $G < B_{\text{lower}}$
        $B_{\text{lower}} \leftarrow \delta_{\text{sum}}$
    end if
    if $V(s) > b_{\text{upper}}(tree(s).parent)$ ▷ parent's local bounds
        $b_{\text{upper}}(tree(s).parent) \leftarrow V(s)$
    end if
    if $V(s) < b_{\text{lower}}(tree(s).parent)$
        $b_{\text{lower}}(tree(s).parent) \leftarrow V(s)$
    end if
end procedure

information about the learning parameters α, γ, and λ into the local bounds; this is a step up over the global bounds. By default, the local bounds get initialized to $\pm\infty$, but can also be initialized to more informed values, for example to V_{init} (or Q_{init}). In such case, when the initial value estimates V_{init} are not reliable, it might be better to overwrite the local bounds after the first visit of a state.

Our new method is helpful when it is difficult or impossible to define in advance the minimum and maximum values of all the estimates. Due to this, it enables the efficient use of UCB selection policies within RL methods in general, therefore, also with algorithms that employ TD backups and eligibility traces or reward-discounting. It can serve as a non-parametric enhancement to any search or learning algorithm that uses selection policies that require normalization of values. Its evaluation is presented in Section 6.2 (Figures 6.9–6.11) and in Section 7.1 (Figure 7.7). We experimented also with some variants of it – with local bounds memorizing the minimum and maximum value estimate of all successor states (e.g., of the whole tree-branch) from each state, and with memorizing the minimum and maximum discounted return (backup target value) of each state. Both proved to perform worse than the one-step-successor bounds described above, but still performed better than using only global normalization.

Our normalization approach can also be understood as a method for online adaptation (or tuning) of the UCB's exploration rate C_p, where it changes both in time and space – each state (node in the tree) gets its own value of C_p at each time step. Little research has been made about this in the context of MCTS: Kozelek [18] experimented with some schemes for changing C_p in time (but still using the same value across the whole state space), but did not get promising results; and Chaslot et al.[95] configured neural networks to output the values of parameters similar to C_p based on the current state (online), but give few details about their approach and experiment only on their MoGo player for Go. The research above was focused on tuning C_p to directly optimize the algorithms performance, whereas our primary goal is to adapt C_p for sake of the normalization requirements of the UCB policy (indirectly affecting performance).

Silver [10] also suggests adapting the exploration rate C_p online to efficiently integrate the UCT's tree policy (i.e., the UCB1 selection policy) into his TD search framework, but he gives no details about these experiments except that nothing performed as robustly and effectively as a simple ε-greedy control policy with a constant ε. In contrast, our experiments (see Chapter 6) show that when using our normalization technique, an ε-greedy policy with a decreasing ε performs better than with a

constant ε, and the UCB1 control policy preforms even better. This might be due to our approach adapting C_p also space-locally, and not only through time.

4.5 The parameters and their mechanics

Our TDTS framework and the Sarsa-UCT(λ) algorithm extend the *standard* UCT algorithm with four new parameters: the update step-size α, the eligibility trace decay rate λ, the initial state values $V_{\text{init}}(s)$, and the assumed playout state values $V_{\text{playout}}(s)$. The latter two can be implemented either as constant parameters or as functions that assign values to states online. The reward discount rate γ is already present in the *original* UCT algorithm [8], as mentioned in Section 2.3, and the exploration rate C_p is present in most UCT variants. The new parameters generalize UCT with several mechanics (Table 4.1), many of which had already been analysed by the MCTS community through different MCTS-extensions (Section 3.7), but also with some new ones in the light of the RL theory that we described in the previous sections. A similar approach can be employed to extend every MCTS algorithm.

Update step-size and initial values

The *update step-size* α controls the weight given to individual updates. For correct convergence, it should be decreasing towards zero in the limit. When α is less-than-inversely decreasing, e.g., linearly, more recent updates get higher weight. This is sensible when the policy is improving with time, implying that recent feedback is better. This way, α can also be understood as a *forgetting rate* of past experience – either on a time-step basis, or in-between individual searches, e.g., retaining or forgetting the tree in-between searches.

A common technique is to decrease α inversely with the number of updates or visits of a state (as presented in Algorithm 1, line 43) [9]. This corresponds to averaging the feedback – giving equal weight to each update – and is also the default approach of MCTS methods [5]. In such case, when the starting value of α is 1, *initial values* $V_{\text{init}}(s)$ and $Q_{\text{init}}(s,a)$ have no impact because they get overwritten after the first visit – this is the default MC approach, as given by Equation (3.4). On the other hand, when the starting value of α is less than 1, initial values do not get completely overwritten: the smaller the starting α value, the bigger impact they have. This is sensible when initial values are good approximations of the true values. In this way, initial values provide the means to introduce *prior knowledge*, *expert knowledge*, and *heuristics*. All the above

Table 4.1

The parameters of Sarsa-UCT(λ).

Parameter	· Control mechanics · Related uses in MCTS
Exploration rate $C_p \geq 0$	· The exploration tendency of the UCB [13] control policy. · In the original UCT [8] and in its variants. Also an alternative value-normalization method for the UCB1 policy.
Update step-size $\alpha \in [0, 1]$	· The forgetting rate: the impact of initial values and the weight of individual feedbacks. · Forgetting techniques [55, 89, 91, 92, 94], progressive bias [57], progressive history [87, 96], BRUE(α) [55].
Initial values $V_{\text{init}}(s)$ or $Q_{\text{init}}(s, a)$	· Prior or expert knowledge in form of initial-estimate bias (when the initial update step-size α is less than 1). Emphasize exploration when chosen optimistically. · First-play urgency [38], history heuristics [18, 86], grandfather heuristics and use of offline-learned values [28], several enhancements with heuristics and expert knowledge in the tree and expansion phases [5].
Reward discount rate $\gamma \in [0, 1]$	· The discounting of long-term rewards: prioritizing short-term rewards when $\gamma < 1$ (changes the task). The reliability of distant feedback. · In the original UCT [8], omitted from many of its later variants [5].
Eligibility trace decay rate $\lambda \in [0, 1]$	· Bootstrapping backups when $\lambda < 1$. First-visit or every-visit behaviour. The reliability of distant feedback. Trade-off between variance and bias. · Reward-weighting schemes [84, 85], indirectly also in the original UCT through adapting the C_p value [8].
Playout values $V_{\text{playout}}(s)$ or $Q_{\text{playout}}(s, a)$	· Prior or expert knowledge in the playout: guide the playout control policy and impact the accumulation of TD errors in the playout. · Several enhancements with heuristics and expert knowledge in the playout phase [5].

allows the implementation of numerous enhancements (including "forgetting") that have been studied also in MCTS (Section 3.7).

Initial values may also be chosen optimistically (set to high values compared to the reward distribution) for the algorithm to favour exploration, for example, using $V_{\text{init}}(s) = 1$ in games with only terminal rewards that are in range $[0, 1]$ (as is the case in many classic games). In general, initial values can strongly affect the convergence rate of the algorithm, hence it is reasonable to set them according to the expected distribution of rewards in a given task (i.e., in range $[0, 1]$ for the example given above).

Reward discount rate and eligibility trace decay rate

The *reward discount rate* γ and the *eligibility trace decay rate* λ both diminish the impact of a reward proportionally to the distance from where it was received, as given in Equations (3.7)–(3.10). This is a way of modelling the *reliability* of distant feedback (e.g., long playouts might be less reliable) and is useful when the generative model is not accurate or the control policy is far from optimal (e.g., random).

Except for the above, the two mechanics differ essentially due the role of γ in Equation (3.5). A $\gamma < 1$ gives higher importance to closer rewards, which is a way of favouring the *shortest-path* to the solution, even when it is sub-optimal – this changes the goal of the given task. It causes the estimates to converge to a different value than the final reward, which might be undesired.

On the other hand, a $\lambda < 1$ enables *bootstrapping* TD backups, which often outperform pure MC backups (i.e., $\lambda = 1$) [9]. In this way the estimates change more slowly (same as when $\gamma < 1$), but they still converge towards the same, non-decayed value. Furthermore, through adopting different update schemes for λ, as given by Equations (3.7) and (3.8), we can implement *every-visit* or *first-visit* update dynamics, respectively.

Playout values

The assumed *playout values* $V_{\text{playout}(s)}$ and $Q_{\text{playout}(s,a)}$ guide the playout control policy (unless it ignores them, as is the case with a fully random policy, for example). They are not necessarily constant or equal for every playout state, but might also get computed (adapted) online – this is a way of introducing heuristic or expert knowledge in the playout phase, for example, with the help of *evaluation functions* (which are popular among applied MCTS algorithms [5]).

When $\gamma < 1$ or $\lambda < 1$ playout values may strongly affect the convergence speed, because they influence the accumulation of TD errors in the playout, as discussed in Section 4.1.

4.6 Implementation remarks

Most MCTS algorithms could be enhanced to perform TD backups with eligibility traces instead of MC backups, in a similar way as we enhanced UCT into Sarsa-UCT(λ). A great advantage of this approach is that it cannot detriment the performance of an algorithm (when implemented correctly) – in the worst-case, the trace decay rate λ can be left on 1, resulting in ordinary MC backups. Furthermore, significant improvements might be achieved (see results in Chapters 6 and 7) when $\lambda < 1$; although, when extending complex algorithms in such a way, the parameters (existing and new) might have to be re-tuned.

We hereby encourage the research community to further explore the potential of our TDTS framework and of TD backups in general by applying them to the huge variety of MCTS algorithms, variants, and extensions on different domains. We also encourage the use of our space-local value normalization technique on algorithms that employ UCB selection policies. To ease this process, here we outline some general guidelines about the necessary implementation choices when extending MCTS methods in this way.

4.6.1 Online updates

TD methods can update value estimates online, this is, backup the observed reward after each time step, unlike MC methods, which backup exclusively at episode end (i.e., offline). This can significantly improve the convergence rate on tasks that exhibit transpositions and non-terminal rewards (when these are both correctly observed by the algorithm). Otherwise, when either condition is false, online updating brings no benefit over offline updating, but only consumes more computational power, hence it is not advised.

Online updating produces $O(T^2)$ backup steps in a single episode instead of $O(T)$ steps, where T is the duration of the episode (i.e., the number of time steps). This might decrease the overall performance of the algorithm if it is run in a limited-time setting (e.g., when it is used for planning, like MCTS methods), because it might gather less

experience (compute less MCTS iterations). Therefore, when deciding whether to perform online updates, one should primarily consider the expected duration of episodes in the given task and the computational cost of simulating the task – one should determine whether the cost of a single backup is negligible compared to simulating a single step of the task. Finally, other components of the algorithm that affect its computational complexity and the quality of the estimates should also be considered, such as hand-coded or heuristic strategies, evaluation functions, etc.

4.6.2 Off-policy control

As described in Section 3.4, backing up some other value instead of the reward produced by the control policy changes the dynamics of RL algorithms from on-policy to off-policy control. These are two fundamentally different classes of RL methods, hence, doing so has strong implications for the convergence conditions of the learning process. Off-policy methods are more difficult to implement, are slower to converge, may be computationally more expensive, and may exhibit worse online performance; but on the other hand, they are also more general, less prone to getting stuck in local optima, and might produce more optimal (target) policies – they might be more advisable for planning (as in the case of MCTS methods), where the online performance is not crucial.

Our Sarsa-UCT(λ) algorithm is by definition an on-policy method, so it is not suitable for backing up differently than defined in Algorithm 1 (lines 39–40). It should backup only rewards produced by the control policy and not, for example, the maximum value of direct successor states (i.e., children nodes), as is sometimes used in MCTS algorithms (i.e., the "Max" value backup [4]).

Implementation of off-policy TDTS methods is a complex topic, especially if also transpositions are used, and is as such out of the scope of this thesis.

4.6.3 Terminal and non-terminal rewards

The presented Sarsa-UCT(λ) algorithm observes the rewards at each time step, like the *original UCT* algorithm, but it could also be simplified to consider only the final sum of rewards, ignoring the time of occurrence of individual rewards and assuming that all rewards were received in the final state, like the *standard UCT* algorithm. When ignoring the time of occurrence of rewards the quality of the value estimate of the starting state (i.e., the very first root node) is not affected; however, the value function

for all the other states will be biased if the given task exhibits non-terminal rewards. The further away a state is from the starting state, the more biased its estimate might be, because it might get unjustly credited (or blamed) for rewards that occurred *before* visiting that state (the same applies to actions). Therefore, the value function might be wrong, causing slow convergence or even divergence of the learning process.

Many MCTS algorithms discard the tree after each search, that is, reset the estimates after each batch of episodes; this alleviates the problem above, because in this way the visited states have biased estimates less frequently (this depends on the branching factor and the amount of transpositions in a task). However, periodically discarding the estimates has deeper implications – it might both improve or detriment the overall performance, depending on the task itself and on the quality of the generative model (depending on whether past experience is helpful and relevant, or whether it is biased and outdated).

Despite the possibly-incorrect value function, ignoring the time of occurrence of rewards can also be beneficial. Besides simplifying the normalization of value estimates for use with UCB-based control policies, as we described in Section 4.4, it might also lower the overall computational complexity of the algorithm. The basic RL theory does not consider that observing a reward might incur a computational cost due to how rewards are modelled in a given task – the basic MDP model does not include such dynamics. For example, in some tasks rewards are expressed as changes in score at each time step, but observing the score might require significant computational resources – this is the case with numerous games in the General Game Playing [22] competition. Hence, computing the score (observing the reward) at each time step instead of computing it only in terminal positions might decrease the number of simulations the algorithm can produce in a limited amount of time, which might decrease its overall performance.

Summing up, when the computational cost of observing the rewards is negligible, it is advisable to consider their time of occurrence and use an appropriate value-normalization technique if the algorithm employs a UCB selection policy (e.g., the *space-local value normalization* presented in Section 4.4). Otherwise, when the computational cost of observing the rewards is significant, a task-specific analysis might be required to assess what performs best. However, if one already established that discarding the estimates after each batch of episodes is beneficial, then considering only the final sum of rewards might be acceptable (but only when the number of episodes in

each batch is reasonably low and when the biased states do not get revisited too often during a batch).

4.6.4 Transpositions

Identifying and exploiting transpositions in general brings great benefits to learning algorithms; RL methods do this by default, whereas MCTS methods treat this as an enhancement. Still, using transpositions is not always advisable – understanding their drawbacks and implications is critical. Furthermore, a correct implementation might not be trivial, because it depends heavily on the choice and configuration of the underlying learning algorithm.

The cost of observing states

Similar as with observing rewards, observing states (or some of their features) might also incur computational costs. Computing a unique identifier from each observed state (e.g., passing the features through a hashing function), which is usually required to access or update the value estimates, might further increase the cost, especially when the state description is complex, such as in the GVG-AI competition games [15], for example.

In deterministic tasks, the same sequence of actions always leads to the same position in the state space – each sequence uniquely defines a state. As a result, there is no need to directly observe states, because these sequences already serve as identifiers; this is the default implementation of basic MCTS methods that build a tree without using transpositions.

In non-deterministic tasks, on the other hand, states *must* be identified by observing them if an algorithm is to learn the true value function. But, when it is enough to learn only an approximate of the true value function, the algorithm can still identify states based on sequences of actions – this is the same as if the algorithm would assume the task is deterministic, ignoring the stochasticity. The produced value function is an approximate, because multiple successor states that derive from the same action get evaluated as one. This works sufficiently well when there is little diversity in the reward distribution among these "grouped" states and when the task is not too stochastic. It is also a viable approach when states can be only partially observed, or cannot be observed at all (this transcends into the theory of partially observable MDPs [46], which is out of the scope of this thesis).

The benefits of identifying transpositions

A *transposition* is a state that can be reached from different sequences of actions. An algorithm can identify them by comparing the state features or the unique identifiers (e.g., hash values) of the states visited so far. When states are already identified by observing them (due to the reasons described above), transpositions can be exploited without additional computational cost.

The main advantage of exploiting transpositions is that the value estimates of such states get updated more frequently. This speeds up the convergence towards their true value; it can considerably improve the performance of the algorithm, depending on the number of transpositions that are present in a task (the more, the better). The performance can be increased even further by computing updates online instead of offline (this cannot be done without using transpositions), as described earlier.

On the other hand, if states are to be observed only to identify transpositions (and need not to be observed otherwise), then the improvement in learning rate must compensate the additional computational cost. Therefore, using transpositions is advisable on highly non-deterministic tasks, where the states must be observed either way, or when the cost of observation is low compared to the cost of simulating a time step of the task.

Using transpositions in general RL methods

When using transpositions in combination with TD learning and eligibility traces one must decide whether to implement every-visit or first-visit behaviour (i.e., accumulating or replacing traces, respectively). The latter is preferable due to better convergence guarantees, as noted in Section 3.4, but requires a slightly more complex implementation: the algorithm needs to remember the time step of the first visit of each state (or state-action) in the last episode, so that if a state gets visited multiple times in an episode, all but the first visit will be ignored during backup (an estimate will be updated only once per episode). Therefore, to obtain first-visit behaviour, besides the ordinary counter of visits n, each state (tree node) requires an additional counter n_e to remember the number of updates, i.e., to count the number of episodes in which a state was visited. The new counter n_e replaces n when computing the update step-size α (Algorithm 1, line 43). The ordinary visit counter n still affects the exploratory bias of the UCB control policy (line 54). When some other control policy is used, e.g.,

ε-greedy, n might not be required.

When using transpositions with on-policy control (e.g., as in our Sarsa-UCT(λ) algorithm), implementing the above is enough; however, when using them with off-policy control (e.g., with Q-learning), a correct and efficient implementation is more difficult, as already mentioned earlier.

A further improvement

Lastly, when using transpositions, the backup process can be devised to update the estimates on all possible paths from the terminal state towards the root, and not only on the one trajectory that was traversed by the algorithm. This might remarkably improve the convergence rate by increasing the frequency of updating individual estimates, but might also heavily increase the computational cost of backups. Furthermore, care must be taken not to break the convergence prerequisites of RL control methods; the estimates on the "alternative" paths must be updated differently: instead of updating them towards the last return (as the estimates on the visited trajectory are), they must be updated by considering only the change of value of their successor (children) states, and their visits counters should not be incremented. The algorithm must keep counters of how many times it visited a direct successor state from each state (i.e., node). These counters are necessary for recomputing a node's value when any of its successors change in value.

In the context of MCTS, these convergence issues have been observed by Childs et al. [31] when they analysed methods of backpropagation using UCT with transpositions – their *UCT3* algorithm backpropagates the same as we suggest above, and, as expected, the authors report it might be beneficial when simulating the task is computationally more expensive than updating the estimates this way. Saffidine et al. [36], also studied the idea of updating all parents when using transpositions – they name this as the *update-all* backpropagation. They likewise recognize the problem with convergence, and demonstrate it on a counterexample.

4.6.5 Summary

Most of the implementational choices we detailed in this section depend heavily on the computational complexity of the task that a planning algorithm is solving. On one side, when the cost of simulating a time step of the task is high, sophisticated backup and memorization methods can improve the performance by extracting as much as

possible information out of each "precious" piece of experience. On the other side, when the cost of observing rewards, observing states, or uniquely identifying states is high, this might unnecessarily slow down the algorithm, causing it to gather less experience in a limited time. Furthermore, the (relative) amount of non-determinism, of transpositions, and of non-terminal rewards in the task, influence the gain of such computationally-heavier methods (and define whether they are sensible to implement at all), as described throughout this section.

Summarizing, prior to implementing such an algorithm, it boils down to assessing what is the optimal ratio of distributing the computational budget for the given task: whether to gather more learning samples (perform more simulations), or to make better use of the learning samples. The same applies whether to use evaluation functions (which is a well-known dilemma, hence we do not detail it). In the end, it is highly probable that domain-specific experimentation will be needed; however, we hope that the guidelines presented here might help the reader to develop a better intuition about implementing TDTS, MCTS, and RL algorithms in general.

5

Survey of research inspired by both fields

This chapter surveys the existing literature that was inspired by both Monte Carlo tree search (MCTS) and reinforcement learning (RL) methods. First we describe the studies that touch on the relation between MCTS with RL methods; there we deem as the most important the line of research leading to the temporal-difference search algorithm [41]. Then we survey the studies that combined techniques from both fields without explicitly commenting on the relation between them.

5.1 Studies that describe the relation between MCTS and RL

We provide an overview of the studies that previously described the connection between MCTS and RL algorithms and emphasize how our study upgrades and consolidates these descriptions.

Gelly and Silver [28] outlined the links between RL and MCTS soon after the invention of the latter: they describe the UCT algorithm as "a value-based reinforcement-learning algorithm that focusses exclusively on the start state and the tree of subsequent states". Continuing this line of research [97, 98], Silver et al.[10, 41] advanced the understanding of RL and MCTS from two separate classes of algorithms into a common one by devising a generalization of MCTS known as the *temporal-difference (TD) search* (Section 5.2).

Baier [43] presents a concise "learning view on MCTS" in the introductory part of his thesis. He covers some of the topics we presented in the previous chapters: he links bandit problems with RL (due to their role in the UCT algorithm); links planning and search with learning; and examines the concepts of policy evaluation and policy improvement in the UCT algorithm. He is the only researcher we know of that links with RL also some MCTS enhancements: he compares the widespread use of function-approximation techniques in RL with the popular enhancements *AMAF* [38] and *RAVE* [39], observing that both MCTS and RL make use of generalization techniques that transfer knowledge to "similar" states. Several other studies (Section 5.3) were inspired by both fields, and although most of them acknowledge their relation, they provide either only brief descriptions [5, 99] or none at all.

The RL view on planning and heuristic search, and its strong connection to learning, is most comprehensively described in the canonical RL textbook by Sutton and Barto [9]. A second edition is in progress, where they also overview MCTS methods [42] in a similar way as in the TD search study [41].

The studies emphasized above are the only ones that describe the relation between

MCTS and RL more in detail. Nevertheless, none of them is primarily focused on the relation itself. For example, Silver et al.[41] state that MCTS methods (including UCT) equal TD search under "specific circumstances", and that therefore they classify as RL methods; however, these circumstances were not explored, as their analysis was heavily focused on improving playing strength in Go and finding good function-approximation features for achieving this. Also, Baier [43] is primarily focused on comparing and evaluating MCTS enhancements for one-player and two-player domains. Due to this, to our opinion the connection between the two fields has not been thoroughly analysed and described yet. Not all the aspects of MCTS (as being more novel) have been considered yet, and the gap between the two communities has also not been given attention yet.

This is where our work comes in. Unlike previous studies, we explicitly highlight the issue of the RL and MCTS communities growing apart and primarily focus on improving the cross-awareness between them. We try to achieve this by providing a comprehensive description of the relation between the two classes of methods they represent, upgrading the descriptions from the previous studies and focusing on exactly those "specific circumstances" in which TD search and MCTS are equal. We argue that our work is the first that discusses the reasons behind this gap, that discusses the different perspectives, that fully relates the terminology by explaining the dynamics (both similar and different) with concepts and terms from both communities, that surveys MCTS enhancements that implement similar dynamics as basic RL algorithms, and that analytically explores how can bootstrapping be beneficial in basic MCTS-like settings (i.e., when using an incremental table representation) and evaluates it on complex real-time (arcade video) games.

Previous studies have already linked some basic concepts in MCTS and RL: they explained the connection of MCTS to sample-based planning in RL (i.e., gathering experience, requiring a simulative model, performing backups, using a control policy), the connection to MDPs, the role of the exploration-exploitation dilemma, and the concepts of policy evaluation and policy improvement. On top of this similarities, we add the distinction between terminal and non-terminal rewards, online and offline backups, every-visit and first-visit updating, on-policy and off-policy control, and evaluating actions with afterstates or state-actions. Also, we acknowledge the MCTS researchers that re-observed these dynamics from an MCTS perspective. And furthermore, we differentiate the original MDP-derived variant of the UCT algorithm from

the standard variant adopted by the community. Finally, besides the similarities listed above, we also discuss the differences that basic RL theory cannot directly explain (the novelties of MCTS) – a non-memorized part of the search space due to the use of incremental (adaptive) representations.

5.2 Temporal-difference search

The TD search algorithm [41] is the first proper online integration of TD learning into the MCTS framework: it generalizes MCTS by replacing Monte Carlo backups with bootstrapping TD backups. Since our TDTS framework also extends MCTS with TD backups, it is related to TD search. The two equal when TDTS is configured to use on-policy control with an ε-greedy policy, value function approximation (opposed to a tabular representation), not to use incremental representations, but to pre-allocate the representation for the whole state space (eliminating the playout policy and playout values), and to update all the visited states in each iteration. Despite this similarity, the two frameworks have been developed independently and with somewhat different goals: TD search adopts TD backups in MCTS to improve tree search in combination with Go-specific heuristics, whereas we adopt them to prove the connection and benefits of RL concepts in tree search algorithms in general. This led to two conceptually different frameworks – TDTS introduces the notions of a non-represented part of the state space, a representation policy, playout value function, and playout policy (the latter is known to MCTS, but not to RL). Therefore, it implements by default the four MCTS phases, unlike TD search, which by default has no playout and expansion phases, and no playout policy. Lastly, TD search seems centred on on-policy control methods, whereas we introduce TDTS with more generality – it is intended also for off-policy control or any other kind of backups. In general, TDTS can be understood as an upgrade of TD search with the MCTS-inspired notions listed above, dealing with an incremental state representation in a more principled way.

Unfortunately, despite the promising results, the TD search framework has not become very popular among the game AI community and MCTS practitioners. We suggest this might be because the analysed TD search algorithm was configured specifically for Go, and based on RL-heavy background: much emphasis was put in function approximation methods – in learning the weights of a heuristic evaluation function based on features specific to Go, which makes their analysis specific to features useful in this domain and makes the algorithm more difficult to implement in other domains.

By considering these observations, we introduce the TDTS framework more from a games-and-search perspective. The example TDTS algorithms we experiment with are focused on the default MCTS setting (with a UCB1 policy, tree representation, expansion and playout phases) and they do not include domain-specific features, thus retaining more generality. In this way, they are easier to implement as an extension of traditional MCTS algorithms, and they preserve the computational speed of the original UCT algorithm, unlike TD search, which was observed as being significantly slower [41]. Also our analysis of bootstrapping backups is focused on such a setting, so that we more clearly convey their benefits. Lastly, we do not focus the empirical evaluation on a single game, but on several games of different types, and achieve improvement also in combination with the UCB1 policy. With this we hope the ideas will get better recognition in both communities.

5.3 Research influenced by both MCTS and RL

Several researchers extended MCTS with RL mechanics or vice versa. The researchers vary in how strongly they perceive the relation between the two fields: some understand them as more (or completely) separate, some acknowledge a (strong) connection, and some take it for granted without mentioning it.

Keller and Helmert [100] propose the *trial-based heuristic tree-search* framework, which is a generalization of heuristic search methods, including MCTS. In their framework they analyse a subset of RL dynamics, specifically, the use of dynamic-programming (DP) backups instead of Monte Carlo backups. Their *MaxUCT* algorithm modifies the backpropagation to perform *asynchronous value iteration* [9], with online model-learning. Due to the backup of maximum values instead of means, which was first explored in an MCTS setting by Coulom [4] and later by Ramanujan and Selman [81], such algorithms are related to Q-learning [49]. Due to model-learning, they are also related to *adaptive real-time dynamic programming* [101] and the *Dyna* framework [71]. Feldman and Domshlak [102] continue the work above and analyse the use of DP updates in MCTS more in depth [103].

Hester and Stone[104] in their RL framework for robots *TEXPLORE* employ an algorithm similar to the one we propose in this thesis (Section 4.3). However, they are not focused on the improvement of MCTS methods, nor their relation to RL, but rather focus on their novel model-learning method [105] and use UCT mainly to guide the state-space exploration. They present an extension of the original UCT

with *λ-returns* [9], which they labelled as *UCT(λ)*. Their method does not employ a playout phase and does not expand the memory structure (tree) incrementally, and most importantly, does not take the mean of the returns, but rather the maximum, which differentiates it from standard on-policy MCTS algorithms. Our new algorithm is similar to theirs, but it preserves all the characteristics of MCTS, while memorizing the mean of returns, thus employing the Sarsa(λ) algorithm instead of Q(λ).

Simultaneously to our research, Khandelwal et al.[106] devised four UCT variants that employ temporal-difference updates in the backpropagation phase. They test the algorithms on benchmarks from the International Planning Competition (IPC) and on grid-world problems and observe that such backup methods are more beneficial than tuning the UCT action-selection policy. Their findings are aligned with ours and provide further motivation for applying bootstrapping backups to the MCTS setting. One of their algorithms, MCTS(λ), is a combination of on-policy *λ-returns* [9] and UCT, which updates the estimates similarly to the example configuration of the Sarsa-UCT(λ) algorithm that we used for our experiments. Otherwise, Sarsa-UCT(λ) is more general as it preserves all the RL-based parameters, whereas MCTS(λ) omits the update step-size and reward discount rate. Also, MCTS(λ) backups do not consider the playout phase; the algorithm adds all the visited states to the search tree. This is unlike to Sarsa-UCT(λ), which distinguishes between the memorized and non-memorized values. Also, in MCTS(λ), the offline TD-backups at the end of the episode are slightly different, because they compute TD errors from the already-updated successor estimate, instead of considering the previous, not-yet-updated value. Lastly, Sarsa-UCT(λ) normalizes the value-estimates before passing them to the UCB selection policy.

The researchers noted above mainly focused on backpropagation methods, but, as described, some also extended the basic UCT algorithm with *online model-learning*. Veness et al. [107] were one of the first to explore this in an MCTS context. They derived from RL theory to implement a generalization of UCT that uses Bayesian model-learning, labelling it *ρUCT*.

Besides Veness et al. [107] and Silver et al.[41], there are more researchers that acknowledge a strong relation between RL and MCTS. Asmuth and Littman [40] derive from RL and MDPs when extending the *forward search sparse sampling (FSSS)* technique [79] – a sample-based planner inspired by *sparse sampling*[108] – to produce the *BFS3* algorithm, which they deem as "an application of Bayesian techniques for rein-

forcement learning". They restate the RL view of learning and planning and reconfirm the strong connection between the two from a Bayesian perspective. Silver and Veness [109] extend MCTS to partially observable MDPs, producing the *partially observable Monte-Carlo planning* algorithm; Guez et al. [110] further upgrade this algorithm with principles from model-based RL into the *Bayes-adaptive Monte-Carlo planner* and show it outperforms similar RL algorithms on several benchmarks. Rimmel and Teytaud [99] explicitly acknowledge that MCTS is analogous to RL, and, inspired by this, develop the *contextual MCTS* algorithm by enhancing the UCT playout phase with *tile coding* (a known function approximation method in RL [9]). Wang and Sebag [111] develop the *multi-objective MCTS (MOMCTS)* algorithm and regard it as "the first extension of MCTS to multi-objective reinforcement learning [112]".

Some studies treat RL and MCTS more like two standalone groups of algorithms, but use the value-estimations of both to develop stronger algorithms. Gelly and Silver [28] were the first to combine the benefits of RL and MCTS. On the game of Go they used offline TD-learned values of shape features from the *RLGO* player [97] as initial estimates for the MCTS-based player *MoGo* [3]. Soon afterwards, Silver et al. [98] extended this "one-time" interaction between RL and MCTS to an "interleaving" interaction by defining a two-memory architecture, noted as *Dyna-2* – an extension of Sutton's *Dyna* [71]. Dyna-2 employs a short-term memory that is updated by MCTS during simulation (i.e., during planning), and a long-term memory that is updated with RL methods based on real interaction with the environment. A combined value from both memories is used for action selection. Daswani et al. [113] suggest using UCT as an oracle to gather training samples for RL feature-learning, which is similar to using Dyna-2 for feature-learning [98]. Finnsson and Bjornsson [114] employ *gradient-descent TD* [11] for learning a linear function approximator online; they use it to guide the MCTS tree policy and default policy in *CadiaPlayer*, a twice-champion program in the General Game Playing competition [22]. Ilhan and Etaner-Uyar [115] also learn a linear function approximator online, but through the *true online Sarsa(λ)* algorithm [116] and they use it only in the playout for informing an ε-greedy default policy; they improve the performance of vanilla UCT on a set of GVG-AI games. Robles et al. [117] employ a similar approach, but they learn the approximator already offline and evaluate their approach on the game of Othello; they observe that guiding the default policy is more beneficial than guiding the tree policy. Osaki et al. [118] developed the *TDMC(λ)* algorithm, which enhances TD learning by using winning

probabilities as substitutes for rewards in non-terminal positions. They gather these probabilities with plain MC sampling; however, as future research they propose to use the UCT algorithm. This is similar to one of our algorithms, only that we derive from the opposite direction – we take UCT and integrate TD learning in its framework.

The *AlphaGo* engine[6] is certainly the most successful combination of RL and MCTS estimates in modern history. In March 2016, it overcame the *grand challenge of Go* [2] by defeating one of the world's best human Go players. In the beginning of 2017, it competed in unofficial internet fast-paced matches against several Go world champions, achieving 60 wins and no losses. Finally, in May 2017, it officially defeated the current top Go player in the world in a full-length three-game match, winning all three games. AlphaGo, however, is much more than a combination of MCTS and RL; it employs a multitude of AI techniques, including supervised learning, reinforcement learning, tree search, and, most importantly, deep neural networks [7], which are key to its success. Its playout policy and value estimators are represented with neural networks, both pre-trained offline from a database of matches and from self-play, and then further refined during online play. An UCT-like selection algorithm, labelled *asynchronous policy and value MCTS algorithm*, is used for online search. It observes both the estimates from the pre-trained network and from Monte Carlo evaluations (playouts) – it evaluates positions (and actions) as a weighted sum of the two estimates.

Lastly, although not directly influenced by MCTS, Veness et al.[119] show the benefits of combining bootstrapping backups with minimax search. Their algorithm achieved master-level play in Chess, specifically due to the search component – it is a successful example of a study on the intersection of the same two communities that we are also addressing.

6

Analysis on toy benchmarks

We inquire whether swapping Monte Carlo (MC) backups with temporal-difference (TD) backups increases the planning performance in MCTS-like algorithms. Hence, our primary goal is testing whether an eligibility trace decay rate $\lambda < 1$ performs better than $\lambda = 1$ when *temporal-difference tree search (TDTS)* algorithms are configured similarly to MCTS. We also observe the optimal value of λ under different settings and the performance-sensitivity of the new parameters of TDTS methods.

In this chapter we experiment with several TDTS algorithms on single-player toy games, which allow us to evaluate a large number of configurations due to the low computation time. In the next chapter we proceed with the evaluation on more complex (real) games.

As previously noted, discounting ($\gamma < 1$) is not new to MCTS methods, hence, assessing its impact experimentally is not in the focus of this study. Furthermore, using a $\gamma < 1$ causes an algorithm to search for the shortest path to the solution, which is not in line with classic games that have only terminal outcomes – there, every solution (victory) is equally optimal, regardless of its distance. Lastly, we have no need for discounting, because there are no cycles in our games and function approximation is not used. Due to all the above, we employ a discount rate $\gamma = 1$ throughout all of our experiments.

6.1 Experimental settings

We detail the benchmarked algorithms and their configurations, benchmark tasks, measured metrics, and experimental control variables.

Algorithms

We test a tabular first-visit Sarsa(λ) algorithm [14] with and without using transpositions, and with and without memorizing all nodes per episode – that is, we test an on-policy TD(λ) control algorithm with replacing traces when building a directed graph and when building a tree, when using a fixed representation model and when using an incremental one. The control policies are either random, ε-greedy, or UCB1 with value-normalization as described in Section 4.4. The playout control policy is random. These configurations produce algorithms that span from on-policy MC control [9] to our Sarsa-UCT(λ) algorithm, including both the *original* [8] and *standard* [5] UCT variants.

Benchmark tasks

The benchmark tasks are two single-player *left-right* toy games that base on a discrete one-dimensional state space with the starting position in the middle and terminal states at both ends (Figure 6.1). The game *Random walk* gives a reward of +1 when reaching the rightmost state and rewards of 0 otherwise, whereas the game *Shortest walk* gives a reward of 0 when reaching the rightmost state and a rewards of −1 (a penalty) for every other move. The latter is more difficult and could be interpreted as a shortest-path problem. Variants of such games are popular among both the RL [9] and MCTS communities [36, 120]. A single play (an episode) could in theory be infinitely long due to the player never reaching one of the goal states; however, for practical reasons in our experiments we use a limit of 10000 time steps (moves) per episode.

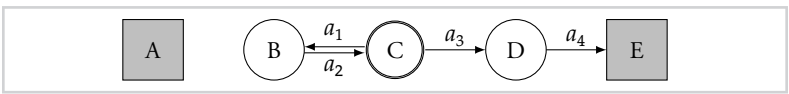

Figure 6.1

A five-state *left-right* toy game with an example episode that starts from C and terminates in E after four actions.

On Random walk we test the quality of *policy evaluation* with a random control policy, whereas on Shortest walk we test the quality of *policy iteration* with ε-greedy and UCB1 policies. We experiment on odd game-sizes from 5 to 21.

Metrics

We observe the quality of the value function, the one-step planning performance, and the overall planning performance.

The *quality of the value function* is measured as the root mean square error (RMSE) of the learned values from the optimal values: when transpositions are used we measure the RMSE across all states, but when a tree is build we measure the RMSE only for the root-node and its children (because, in a tree, several nodes might relate to the same state, but those closer to the root have most impact when performing MCTS-like planning).

The *one-step planning performance* is measured as the probability of selecting the optimal action in the starting state by a fully greedy policy. It is analogous to the *failure rate* [8] and *choice-error probability* [121] metrics.

The *overall planning performance* is measured as the expected number of time steps required to complete the task – to reach the rightmost state from the starting state – when following the same control policy as is used for learning; such policy is usually stochastic, hence the *expected* number of time steps is computed from multiple repeats given the current value estimates.

Control variables

The main control parameters are the eligibility trace decay rate $\lambda \in [0, 1]$ and the available computational time per search. The latter is expressed as the number of simulated episodes per search or as the number of simulated time steps (equivalent to the number of simulated actions) per search.

Other configuration details of the algorithms: the update step-size α is either inversely decreasing with the number of episodes that visited a state or it is held constant on 0.01, 0.05, 0.10, or 0.20; the initialization values V_{init} are set constant to -5, 0, 0.5, 1, 5, 10, or 20; the assumed playout values V_{playout} equal V_{init} or 0; the ε-greedy control policy exploration rate ε is constant on 0.1 or linearly decreases from 0.1 towards 0; in each episode (iteration), the algorithm memorizes either all newly-visited nodes or 1, 5, or 10 newly-visited nodes, in order of visit. The reward discount rate γ and UCB1 exploration rate C_p are fixed on 1. The UCB exploration bias is computed according to Algorithm 1 (line 54).

The above produces millions of different configurations. For each configuration we ran at least 10000 repeats and averaged the results. This was enough to achieve insignificantly small confidence bounds on most experiments.

6.2 Results and findings

The full extent of produced results exceeds the needs of this work, so here we present only the most illustrative examples, and focus rather on a detailed summary of our findings. Except where stated differently, the figures in this section portray the performance on the Shortest walk toy game of size 11 of a TDTS algorithm that employs Sarsa(λ) with an ε-greedy control policy (with $\varepsilon = 0.1$) and random playout control policy, that builds a tree, does not use transpositions, adds one new node per episode, sets initial values to 0, and assumes playout values equal to initial values. The confidence bounds are insignificantly small due to a high number of repeats.

The most relevant metrics

The two planning-performance metrics provide similar conclusions; although, the *overall planning performance* is arguably the most informative (Figures 6.3 and 6.2), because it considers the ranking of all memorized actions and not only of those in the root state. This is even more relevant when the memory structure is preserved between searches – when the MCTS tree is not discarded after each search – because actions deeper in the tree might eventually become root-actions later on.

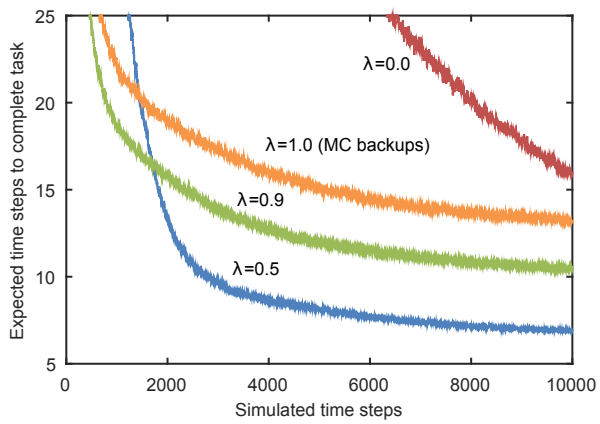

Figure 6.2

Overall planning performance. The algorithm performs best when $\lambda > 0$ and $\lambda < 1$. An optimal policy wins this game in five time steps.

When not using transpositions, both planning metrics show an improvement of $\lambda < 1$ over $\lambda = 1$, which is in contrast with the metric assessing the quality of the value function (the RMSE of root-state children) – the latter shows a drastic increase in RMSE when $\lambda < 1$ (Figure 6.4). This is expected, because a $\lambda < 1$ causes the value estimates to change more slowly; more updates are required to reach the target backup value. However, devising an optimal policy does not require an optimal value-function [9] – it suffices to optimally rank the available actions to select the best one, regardless of how much their estimates differ from the true values. This has been also observed in MCTS, where different algorithms might have approximately equal error in value estimates, but very different regrets [122]. As a consequence, reducing the variance in value estimates at the cost of higher bias can be beneficial, because the selection algorithm can then rank actions more accurately [123]. Considering all the above,

the two planning metrics we chose are more reliable; whereas expressing the quality of the value function as RMSE is less suitable for such benchmarks. This is why we used the overall planning performance (the expected number of time steps to complete the task) as the main metric for our observations on the toy-game experiments.

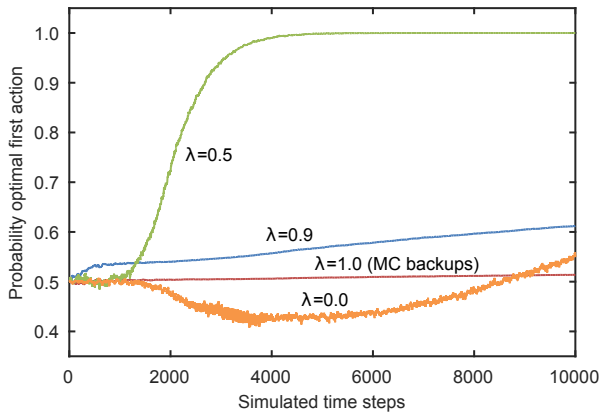

Figure 6.3

One-step planning performance. When using a $\lambda < 1$, the algorithm might learn an optimal policy significantly faster than using lambda $\lambda = 1$.

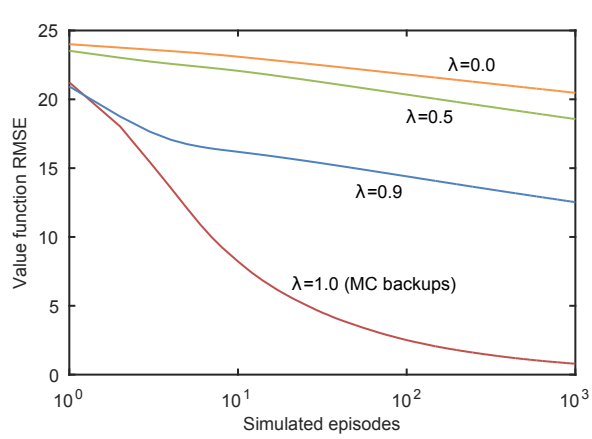

Figure 6.4

Quality of the value function. Algorithms that use a $\lambda < 1$ converge considerably slower towards the optimal value function. However, this metric might be misleading, because a $\lambda < 1$ still improves on the policy (Figures 6.3 and 6.2). Since a tree is built, the root mean squared error (RMSE) is observed only for the root state and its children.

The benefits of TD backups and eligibility traces

The experiments confirm that $\lambda < 1$ performs better than $\lambda = 1$ under the majority of configurations, implying that TD backups are beneficial over MC backups (visible from nearly every figure in this section). This has been long-known for the default RL setting [9], that is, when building a directed graph and memorizing all nodes, and has also been recently observed in specific MCTS configurations [98, 104, 106]. However, it has not been observed yet in the default MCTS setting – when using an incremental tree structure, when adding a limited number of nodes per episode, and when not using transpositions. Furthermore, we observe that TD backups are much more beneficial (produce a larger gain) exactly when not using transpositions (Figure 6.5); when using transpositions they produce only a marginal improvement in our toy games.

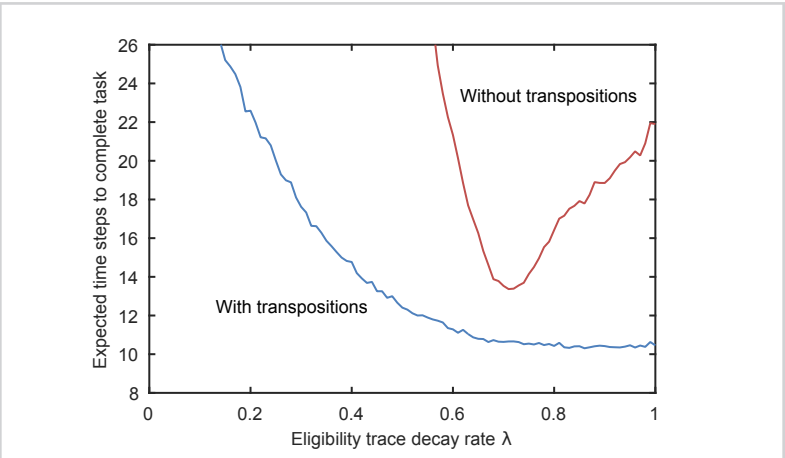

Figure 6.5

TD backups ($\lambda < 1$) are more beneficial when not using transpositions. The results were computed after the algorithm simulated 1000 time steps (the setting is the same as described in the beginning of this section).

The optimal value of the eligibility trace decay rate λ is usually larger than 0 and smaller than 1, but $\lambda = 0$ often performs worse than $\lambda = 1$ (Figure 6.6). There is a λ-threshold above which the performance is at least equal to using MC backups (i.e., using $\lambda = 1$) – a safe range for setting its value. In general, when increasing the available computational time, the threshold and the optimal λ decrease towards some value below 1 (Figure 6.7). Decreasing the size of the game (playing a game with less states) has a similar effect.

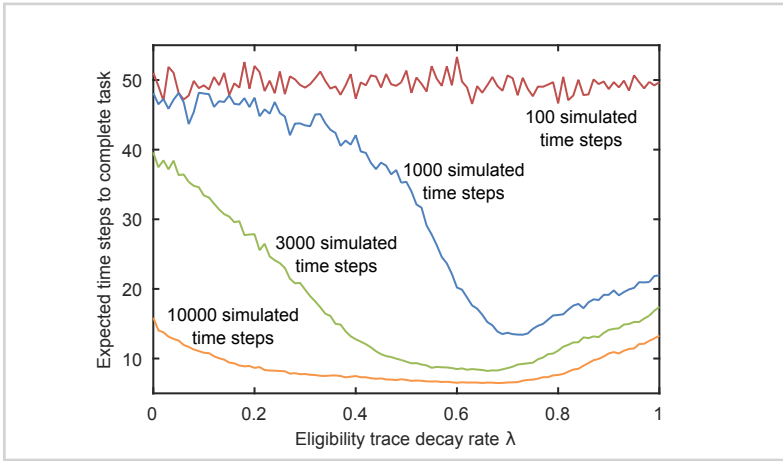

Figure 6.6

Sensitivity to the eligibility trace decay rate λ. With increasing the number of simulated time steps, the optimal λ limits towards some value below 1.

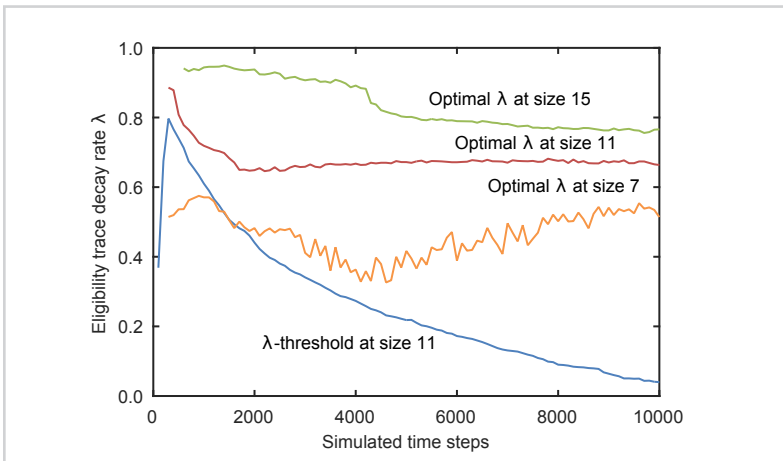

Figure 6.7

Increasing the available computation time or decreasing the game size in general shifts the optimal λ and the λ-threshold towards a lower value. The threshold defines below which λ value the algorithm performs worse than using ordinary MC backups (a safe range for setting the parameter). When given little computation time, the algorithms perform poorly regardless of λ; therefore the results are very noisy up to a few hundred simulated time steps per move.

Different control policies

A $\lambda < 1$ also performs better regardless of the employed control policy (Figure 6.8); this confirms that (in this domain) the Sarsa-UCT(λ) algorithm is superior to the UCT algorithm when λ is set to an informed value However, we stress that when the UCB1

policy is used on tasks with unknown or non-terminal rewards (such as the Shortest walk game), space-local value normalization is necessary for TD backups ($\lambda < 1$) to improve on MC backups (as we explain below).

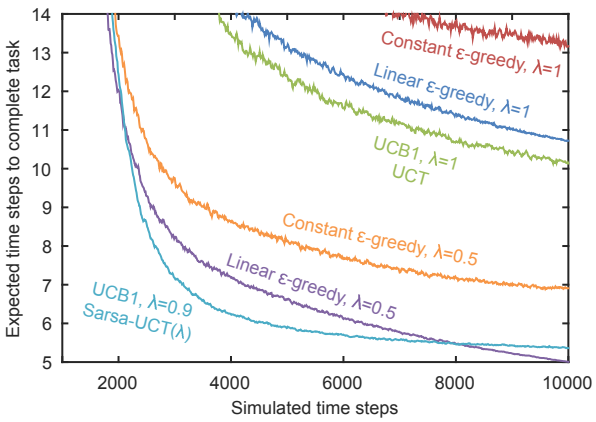

Figure 6.8

The performance of different control policies. All policies perform best when $\lambda < 1$; this confirms that (in this domain) the Sarsa-UCT(λ) algorithm is superior to the UCT algorithm. The UCB1 policy (with space-local value normalization) performs best, but an ε-greedy policy with a linearly-decreasing ε also performs comparably well. The given values of λ are those that perform best with their respective policies. An optimal policy wins this game in five time steps.

The UCB1 policy with the proposed space-local value normalization technique considerably outperforms the ε-greedy policy with the constant exploration rate, and slightly outperforms the ε-greedy policy with the linearly-decreasing exploration rate.

The ε-greedy policy with a decreasing exploration rate performs surprisingly well: it produces an optimal policy in the end (as expected, since ε becomes 0) and reaches a near-optimal policy only slightly slower than the UCB1 policy. It requires the number of simulated time steps (or episodes) to be known in advance, but despite this, it is still simpler than UCB1 in the sense that it is unaffected by the reward distribution – it does not require value-normalization.

The default value of the UCB exploration rate $C_p = 1$ performs well on most settings. We did not experiment with a decreasing exploration rate C_p. Also, other values of C_p and ε that we have not tested might further improve the UCB and ε-greedy policies and even change the balance between them; due to this, we stress that our experiments do *not* identify which policy performs best in general.

The UCB1 control policy and value normalization

The Shortest walk toy game is challenging for the UCB1 policy due to its (in theory) unbounded cumulative reward: the algorithm gets a −1 for each move, and a single episode can last up to 10000 moves (we do not allow longer episodes). Due to this, also when the algorithm performs MC backups ($\lambda = 1$), the UCB1 policy performs considerably better with the help of our space-local value normalization instead of an ordinary global normalization (Figure 6.9). Even more important, when using UCB1 with global normalization we could not improve on MC backups in any of our experiments (Figure 6.10); we discover that when the UCB1 policy is used with TD backups, regardless of the setting, the space-local value normalization is necessary for a $\lambda < 1$ to prove beneficial. In such case, the gain is remarkable when transpositions are not used, and still considerable when they are used (Figure 6.11). Lastly, the space-local normalization technique seems robust to the choice of initial values, playout values, game size, and number of simulated time steps – it provides an improvement regardless of these settings.

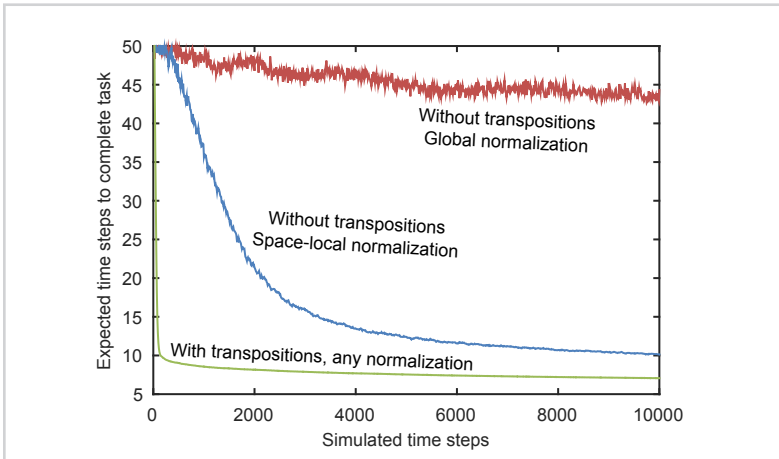

Figure 6.9

Efficiency of space-local normalization on UCB1 with MC backups ($\lambda = 1$). When not using transpositions the gain is remarkable, whereas when using them it is minimal, although still statistically significant (not visible in figure).

We implemented the global normalization both with constant bounds (on Shortest walk we set the upper bound to the optimal score and the lower bound to −10000) as well as with the algorithm adaptively changing the bounds according to the minimum and maximum score encountered so far; however, there was no significant difference in

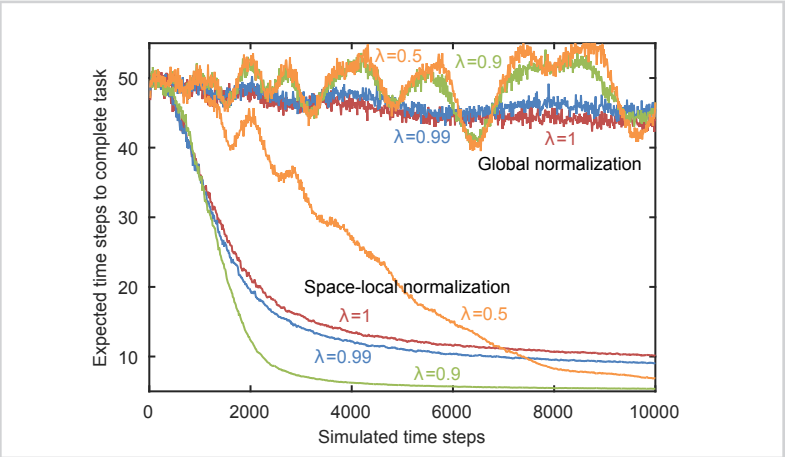

Figure 6.10

Global normalization and space-local normalization of value estimates when using the UCB1 control policy, without using transpositions. Space-local value normalization is required for TD backups ($\lambda < 1$) to improve on MC backups ($\lambda = 1$).

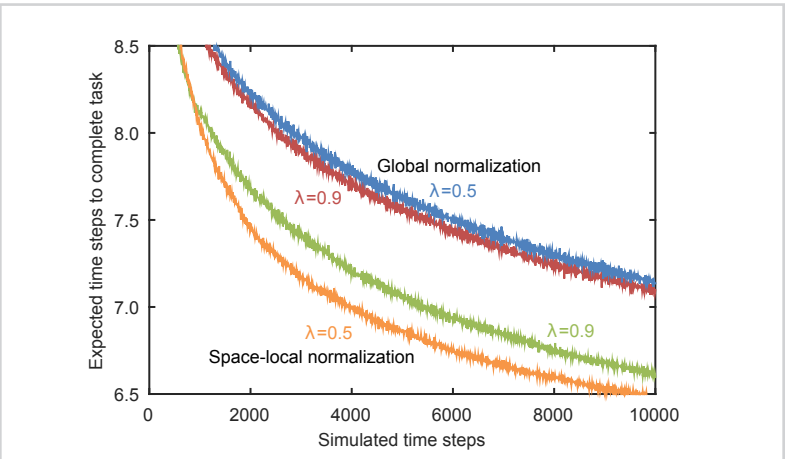

Figure 6.11

Global normalization and space-local normalization of value estimates when using the UCB1 control policy and transpositions.

the performance of the two approaches. We have not experimented with hand-tuned global bounds.

Impact of initial and playout values

When $\lambda = 1$, initial values V_{init} have no impact (as explained in Section 4.5), but when $\lambda < 1$, they might improve the performance if set to an informed value (e.g., 0.5 in the Random walk game), but may also seriously detriment it when set badly (e.g., 5 in the Shortest walk game when using $\lambda = 0.1$, Figure 6.12). Setting playout values V_{playout} to equal initial values performs better than setting them to 0, except when the initial values are bad. In this sense, setting them to 0 is slightly safer. Despite the above, on most tasks it is enough to set the initial values approximately in the rewards range. This can be achieved in many tasks, e.g., in classic two-player games the outcome is usually 1 or 0, suggesting initial values of 0.5 (neutral) or 1.0 (optimistic). Nonetheless, because both initial and playout values have no impact when $\lambda = 1$ and $\gamma = 1$, MC backups are safer regarding the bias. Although we did not experiment with $\gamma < 1$, we believe the impact of initial values would be similar as described above due to the discounting mechanics of γ (discussed in Section 4.5); but, on the other hand, the impact of playout values might be heavier due to the resulting TD errors in the playout phase (discussed in Section 4.1).

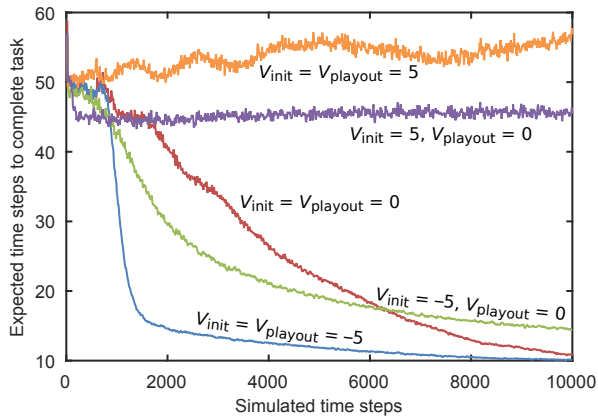

Figure 6.12

The impact of initial and playout values when performing TD backups; results when $\lambda = 0.1$. Tuning initial values might further increase the performance, but also detriment it when set badly. The same holds for playout values, although their impact is smaller: setting them to 0 is slightly safer, but produces less gain.

Impact of the update step-size, number of added nodes, and transpositions

In our first batch of experiments we observed that an inversely decreasing update step-size α performed better than a constant α in all of the settings, so we have not experimented with a constant α in combination with the UCB1 policy. The benefits of this scheme for updating α were expected due to the convergence requirements of such algorithms and because the task is stationary and small [9].

We also reconfirm that adding only one node per episode instead of all nodes does not critically inhibit convergence [3, 4]: the impact is minimal when using transpositions and moderate when not using them (Figure 6.13). In contrast, we observe the performance might drop considerably when omitting the use of transpositions, i.e., when changing the representation from a graph to a tree; the amount of deterioration is strongly affected by the value of λ (Figure 6.5).

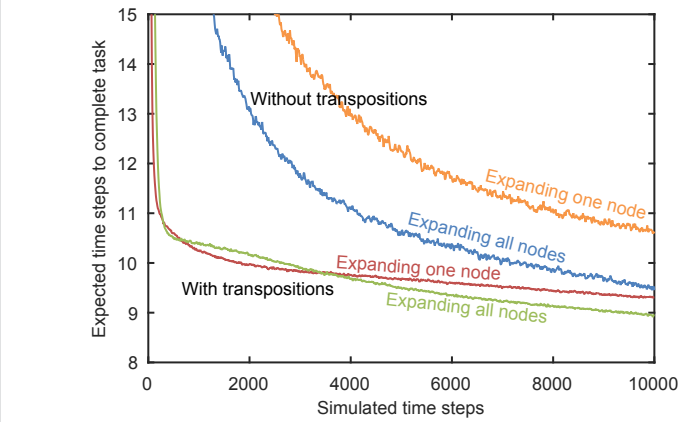

Figure 6.13

The impact of transpositions and number of added nodes per iteration; results when $\lambda = 0.9$. The use of transpositions has a bigger impact on the performance compared to memorizing all the visited states in an iteration.

6.3 An analytic example

Our results show an overall faster convergence of TD(λ) updates over MC updates, with the difference being largest in the early episodes of the algorithms. We analyse why should TD(λ) updates work better in such case – in the early iterations of an MCTS algorithm.

When using TD(λ), the values of the states visited during the playout phase may be assumed to have initial values V_{init}, such as 0.5. Let us assume that $\gamma = 1$ and that the task has only a terminal non-zero reward (e.g., the *Random walk* game). This results in TD errors in the playout being zero; the only non-zero TD error is the final one, owing to the final reward. The value of the root is updated as

$$V_n(S_0) = V_{n-1}(S_0) + \alpha \left[\delta_0 + \lambda\delta_1 + \lambda^2\delta_2 + \ldots + \lambda^{T-P-1}\delta_{T-P-1} + \lambda^{T-1}\delta_{T-1}\right],$$

where P is the length of the playout and T is the total length of an episode. For example, in the early MCTS iterations, the only nodes in the tree are the root node and its children. In such case

$$\begin{aligned} V_n(S_0) &= V_{n-1}(S_0) + \alpha \left[\delta_0 + \lambda^{T-1}\delta_{T-1}\right] \\ &= V_{n-1}(S_0) + \alpha \left[0 + \lambda^P(R_T - V_{\text{init}})\right]. \end{aligned}$$

Therefore, when $\lambda < 1$, a smaller P results in a larger update. In this way, those actions that lead to shorter simulations contribute more to the update, causing child nodes to differ in value regarding how fast the final state was reached. On the other hand, when $\lambda = 1$ there is no difference in state values, since λ^P is always 1.

By using a simple *Random Walk* example, we show the difference between these two cases: $\lambda < 1$ and $\lambda = 1$. Suppose the game has five states: A, B, C, D and E (Figure 6.1). There are two possible actions, leading to the left or right neighbour state. C is the initial state ($S_0 = \text{C}$), whereas A and E are terminal states. The only reward $R = 1$ is delivered when going from D to E, otherwise $R = 0$. An example episode would consist of taking actions

$$\begin{aligned} a_0 &= \text{left}, \\ a_1 &= \text{right}, \\ a_2 &= \text{right, and} \\ a_3 &= \text{right}; \end{aligned}$$

the final state is E at time $t = 4$. At the end of the episode, a TD(λ) algorithm with *replacing traces* produces the eligibility traces

$$\begin{aligned} e_4(\text{B}) &= \lambda^2, \\ e_4(\text{C}) &= \lambda^1, \text{ and} \\ e_4(\text{D}) &= \lambda^0 = 1. \end{aligned}$$

Although

$$V_1(\mathrm{B}) = V_0(\mathrm{B}) + \alpha[\delta_t + \lambda\delta_{t+1} + \lambda^2\delta_{t+2}],$$

in the beginning all the TD errors except δ_{t+2} are 0, and $\alpha_1 = 1/1 = 1$. Therefore,

$$\begin{aligned} V_1(\mathrm{B}) &= V_0(\mathrm{B}) + \alpha\lambda^2\delta_{t+2} \\ &= V_{\text{init}} + \alpha\lambda^2(R_T - V_{\text{init}}) \\ &= 0.5 + \lambda^2(1 - 0.5) \\ &= 0.5(1 + \lambda^2). \end{aligned}$$

Similarly,

$$\begin{aligned} V_1(\mathrm{C}) &= 0.5(1 + \lambda), \text{ and} \\ V_1(\mathrm{D}) &= 0.5(1 + \lambda^0) = 1. \end{aligned}$$

We now compare what happens when $\lambda < 1$ and when $\lambda = 1$. In the first case, say $\lambda = 0.7$,

$$\begin{aligned} V_1(\mathrm{B}) &= 0.5(1 + 0.49) = 0.745, \\ V_1(\mathrm{C}) &= 0.5(1 + 0.7) = 0.85, \text{ and} \\ V_1(\mathrm{D}) &= 1. \end{aligned}$$

When $\lambda = 1$,

$$V_1(\mathrm{B}) = V_1(\mathrm{C}) = V_1(\mathrm{D}) = 1.$$

Observe that at $\lambda < 1$ the values of the states B, C and D monotonically increase, whereas at $\lambda = 1$ they are all 1. In the former case, the best greedy action at C would be going right, whereas in the latter both left and right would be equally good. Further note that at $\lambda = 1$, another episode (or more consequential episodes) finishing in E would still leave all state values at 1.

Performance on real games

7

In the previous chapter we assessed the benefits of temporal-difference tree search (TDTS) configurations on single-player toy games. Here, we continue with the evaluation of Sarsa-UCT(λ) (Algorithm 1) on classic two-player games, where it learns from self-play, and on real-time single-player video games, where little computational time is available per move.

7.1 Classic two-player adversary games

We investigate the planning performance of TD backups in MCTS-like algorithms when learning from simulated self-play on adversary multi-agent tasks. For this, we measure the playing strength of the Sarsa-UCT(λ) algorithm on the classic games of Tic-tac-toe, Connect four, Gomoku, and Hex. These games have been frequently used for benchmarking MCTS algorithms[5].

The games

We briefly describe the classic two-player games we used as benchmark problems in our experiments. In all of them players take turns in placing "pieces" on an initially empty "board", one piece at a time. Players might either win, draw, or lose – the outcome is defined only in terminal positions. Therefore, we implement the games to feedback a reward of 1 for a win, 0.5 for a draw, and 0 for a loss in terminal positions; and a reward of 0 in non-terminal positions.

Tic-tac-toe (also known as *Noughts and Crosses*) is played on a 3×3 board with a winning condition of three pieces in a straight line (either horizontally, vertically, or diagonally). Although the first player has an advantage, the game always end in a draw if both players play optimally. Because of its small search space, Tic-tac-toe can be regarded as a toy game. Here, we evaluated the algorithms at lower computational times per move (compared to other games), otherwise all matches would end in draws.

Gomoku (also known as *Five in a row* or *Gobang*) is an extension of Tic-tac-toe: the board is a square of arbitrary size (the side is usually of even length in range $[9, 19]$) and the first player that places five own pieces in a straight line wins. It can be understood as a very simplified variant of the game of Go. The basic version of Gomoku was solved up to boards of size 15×15. Additional rules have been proposed to lower the advantage of the first player; however, our implementation does not use them.

In *Connect four* (also known as *Four in a row*) the players alternate in dropping pieces from the top of a fixed-size board that is seven column wide and six rows high.

They may choose any column that is not filled up. The winning player is the first who connects at least four own pieces in a straight line. The game has been solved with the outcome that the first player can force a win with perfect play if starting in the middle column.

In *Hex* the players alternate in placing pieces on empty places on an arbitrary-sized rhombus board with a hexagonal grid. The winner is the first player to form a connected path of own pieces between two opposing borders of his own colour. The game cannot end in a draw. It was solved for board sizes up to 8×8. The first player has an advantage, so a "pie rule" is generally used after the first move; however, we do not implement it.

Configuration

The playing strength of the evaluated algorithm is expressed as the average win rate against a *standard UCT* player; draws count towards 50%. Both players have an equal amount of available time per move and equal configuration: both learn from simulated self-play, do not use transpositions, add one new node per episode, preserve the tree between moves, and output the action with the highest value after each search.

The Sarsa-UCT(λ) algorithm is configured to compute TD errors from one-ply successor states, same as, e.g., the *TD-Gammon* algorithm [124], and in contrast to the *TD search* algorithm [41], which computed them from two-ply successors on the game of Go (i.e., in this way ignoring the values of opponent's simulated moves).

To mimic an adversary setting, we implement the minimax behaviour in Sarsa-UCT(λ): we extend Algorithm 1 to compute Q-values (line 54) with $-V_{\text{norm}}$ instead of V_{norm} (i.e., to select the lowest-evaluated actions) when simulating the opponent's moves. This equals to modelling the real opponent player as if it is using the same learning (or search) algorithm.

The experimental control parameters are the eligibility trace decay rate λ, the exploration rate C_p, and the available time per move, which is expressed as the number of simulated time steps (simulated moves) per real move. The fixed parameters are $\gamma = 1$, $V_{\text{init}} = 0.5$, and $V_{\text{playout}} = V_{\text{init}}$. The update step-size α decreases inversely with the number of node visits (as given in Algorithm 1, line 43). We experiment with the number of simulated time steps per move from 10^1 to 10^5 and with board sizes of Gomoku and Hex from 5×5 to 13×13 (Tic-tac-toe and Connect four have fixed board sizes). The results presented in this section are averaged from at least 2000

and up to 20000 repeats; the 95%-confidence bounds are insignificantly small, except where stated otherwise.

Parameter optimization

For each experimental setting we first found the exploration rate C_p where two *standard UCT* players performed most equally one against the other, i.e., where both had the most equal win and draw rates for each starting position (Figure 7.1). Then, we fixed one UCT player on this C_p value, and re-optimized the C_p value of the other player to check if there exists a counter-setting that performs better (Figure 7.2); however, we found none – the optimal value of C_p did not significantly change in any setting. Finally, we swapped one UCT player with the Sarsa-UCT(λ) player, set it the same C_p value, and searched for its highest win rate with regard to λ (Figure 7.3). The parameter C_p was optimized with a resolution of 0.05 in range $[0, 1.5]$, whereas λ was tested at values 0, 0.1, 0.2, 0.4, 0.6, 0.8, 0.9, 0.95, 0.99, 0.999, 0.9999, and 1. The optimizations were performed either manually or with a linear reward-penalty learning automata [125].

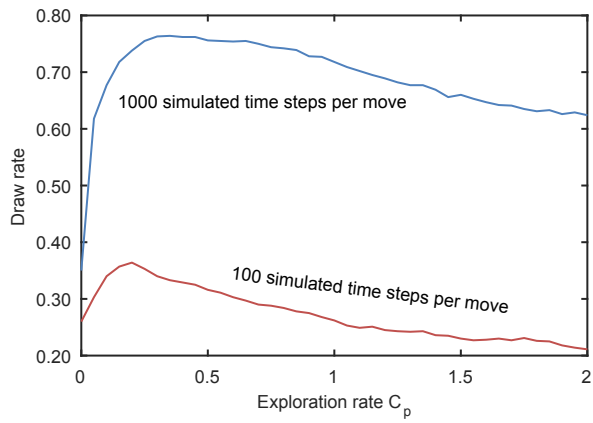

Figure 7.1

First step of parameter optimization: search of the exploration rate C_p where two standard UCT players perform most equally. Example results for Tic-tac-toe at 100 and 1000 simulated time steps per move.

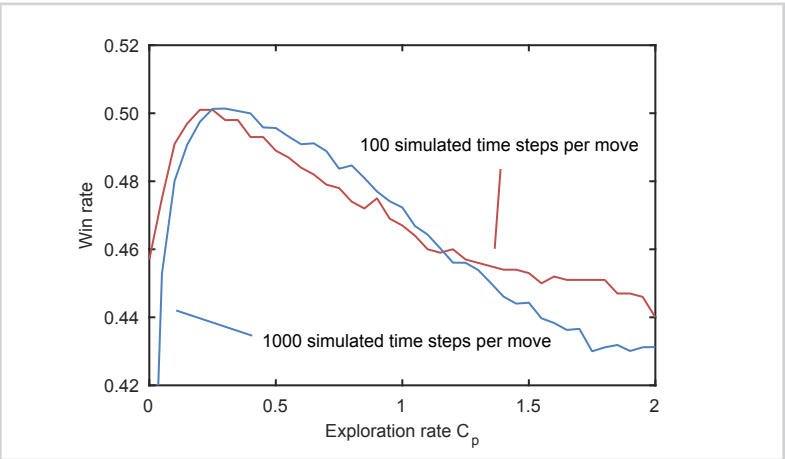

Figure 7.2

Second step of parameter optimization: search of a exploration rate C_p that yields highest performance against an opponent that uses the C_p value found in the first optimization step. Example results for Tic-tac-toe at 100 and 1000 simulated time steps per move.

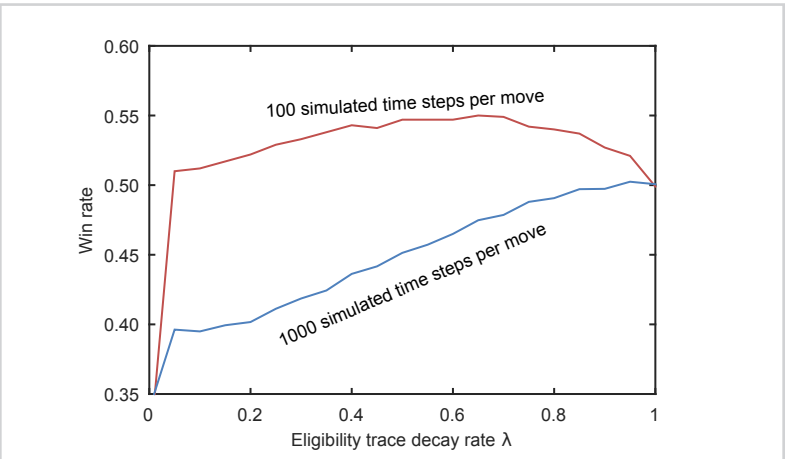

Figure 7.3

Last step of parameter optimization: search of the eligibility trace decay rate λ for the Sarsa-UCT(λ) algorithm that yields the highest gain in performance over the standard UCT algorithm. Example results for Tic-tac-toe at 100 and 1000 simulated time steps per move.

The benefits of TD backups

Our measurements show that tuning the eligibility trace decay rate λ increases the performance of the algorithm; however, the optimal value of λ strongly depends on the amount of computation time available per move (Figures 7.4 and 7.5): by increasing

the number of simulated time steps the optimal value shifts towards 1. Due to the this, increasing the number of simulated time steps per move also strongly impacts the gain in performance of Sarsa-UCT(λ) over the UCT algorithm. Figure 7.5 confirms our theoretical expectation (from Chapters 3 and 4) that the UCT algorithm behaves identically to Sarsa-UCT(1) – their performances are equal, despite our distinct im-

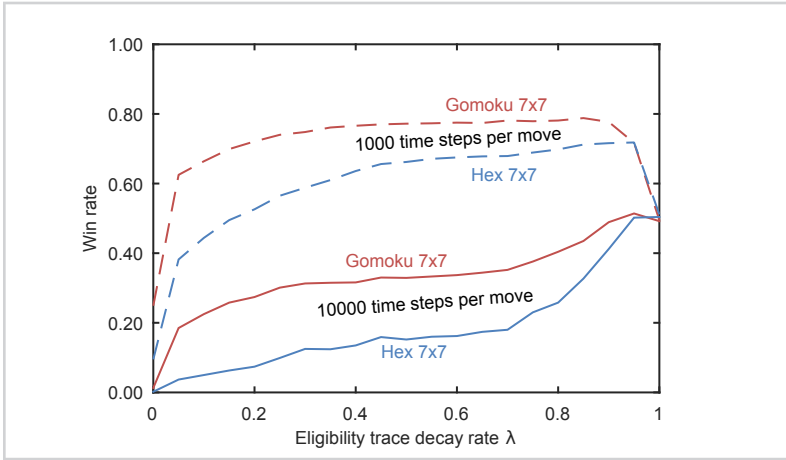

Figure 7.4

The sensitivity of Sarsa-UCT(λ), expressed as win rate against the UCT algorithm, to the parameter λ and the available computation time on Gomoku and Hex. Increasing the number of simulated time steps per move causes the optimal value of λ to shift towards 1.

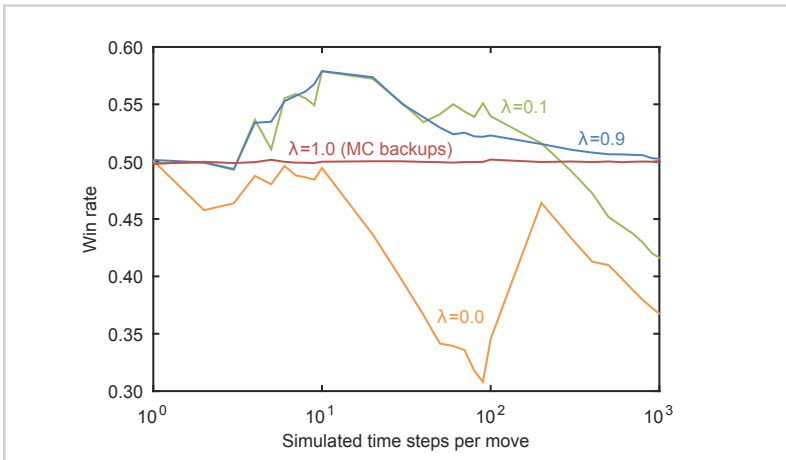

Figure 7.5

The sensitivity to λ and the available computation time on Tic-tac-toe. The results show that UCT behaves identically to Sarsa-UCT(1), as expected (Chapters 3 and 4) – their performances are equal, despite our distinct implementations of the two algorithms.

plementations of the two algorithms.

Sarsa-UCT vs. UCT

Our main results show that Sarsa-UCT(λ) outperforms UCT on all tested games when the amount of available time per move is low, and that it performs at least as well as UCT when otherwise (Figure 7.6). This confirms that TD backups converge faster also when learning from self-play. The gain diminishes when increasing the time because both algorithms play closer to optimal and more matches end in draws, but primarily because the optimal value of λ goes towards 1, which effectively results in both algorithms behaving equally – for this reason Sarsa-UCT(λ) cannot perform worse than UCT even when further increasing the computation time. On the other hand, the effect of enlarging the state space is inverse (e.g., a larger board size in Gomoku and Hex) – it increases the gain of Sarsa-UCT(λ) and lowers the optimal λ value. Considering the above, the drop in gain of the Sarsa-UCT(λ) algorithm when increasing the time is directly related to the optimal value of λ. In turn, what is the optimal value of λ in general (what affects it) has been one of the central research challenges in the field of reinforcement learning for decades [9] (and its solution is not in the scope of this thesis). See Appendix A for full results on two-player games.

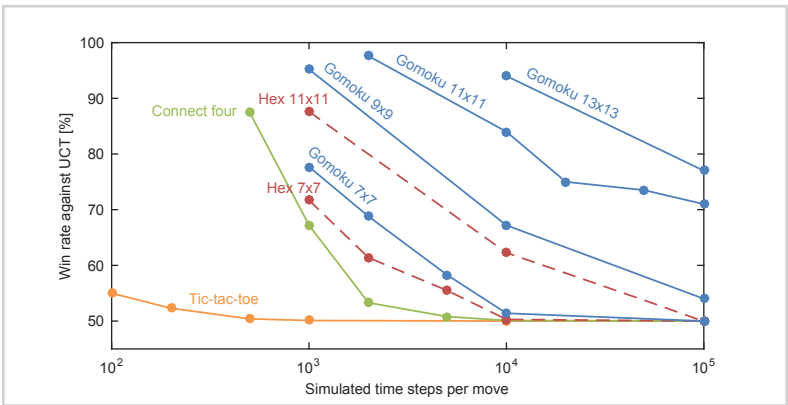

Figure 7.6

Performance of Sarsa-UCT(λ) on classic two-player games. Value-normalization is not used; otherwise, the advantage of Sarsa-UCT over UCT would be even higher (Figure 7.7).

Since these two-player games produce non-zero rewards only in terminal states, UCB1 performs well also without value normalization (the results in Figure 7.6 show such configuration). Still, even on these games, space-local value normalization fur-

ther increases the performance when $\lambda < 1$ and generally decreases the sensitivity to λ (Figure 7.7).

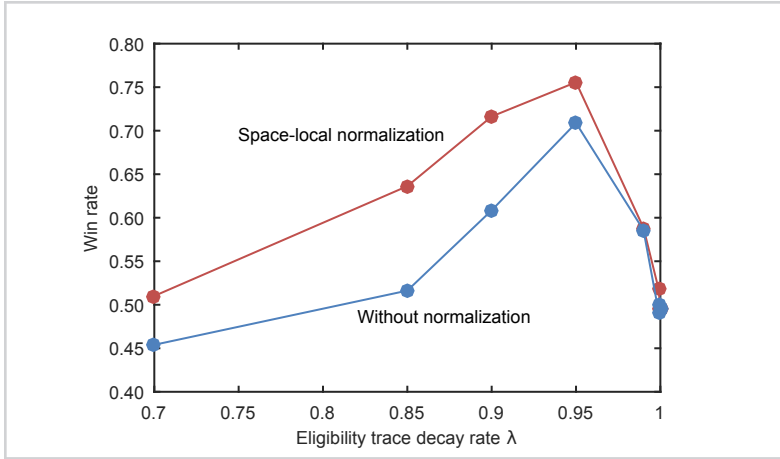

Figure 7.7

The benefit of space-local value normalization on Sarsa-UCT(λ), expressed as win rate against UCT. Example results on Gomoku 11×11 at 100000 simulated time steps per move and $C_p = 0.05$. Each point was computed from at least 4000 repeats.

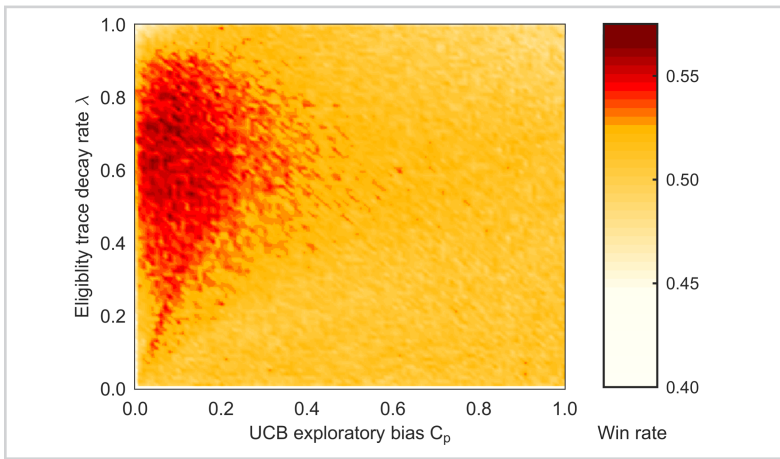

Figure 7.8

Sensitivity of Sarsa-UCT(λ) to λ and C_p expressed as win rate against the standard UCT player on Tic-tac-toe. Best performance is at approximately $\lambda = 0.65$ and $C_p = 0.1$; note the poor performance when λ is near 0. Example results at 100 simulated time steps per move. (Figure granulation is due to a lower number of experimental repeats per point – 1000.)

Lastly, in preliminary experiments we observe that concurrently tuning both λ and C_p might further increase the performance. This is because the value of λ affects the

magnitude of TD updates, and consequently also the bounds for value normalization (required by the UCB1 policy). Therefore, it also affects the optimal value of C_p: for example, in Tic-tac-toe with 100 simulated time steps per move, at $\lambda = 1$, the optimal value of C_p is approximately 0.3, whereas at $\lambda = 0.15$, its optimal value is approximately 0.05 (Figure 7.8). In contrast, the value of Cp does not seem to affect the optimal value of λ; however, we tested this only in the Tic-tac-toe setting described above – there the optimal $\lambda = 0.65$ regardless of the C_p value.

7.2 Real-time video games

We assess the performance of Sarsa-UCT(λ) also on tasks where there is little computational time available per decision. Real-time video games are examples of such tasks. The *General Video Game AI (GVG-AI)* [15] framework is especially suitable for this, as it includes a wide selection of such games and, furthermore, provides an excellent platform for comparative evaluation through the *GVG-AI competition*. The results from the competitions and the source code of the algorithms presented in this section are available on the GVG-AI web site (www.gvgai.net).

The General Video Game AI framework

The GVG-AI framework [15] is a code package in Java for implementing and evaluating AI players (i.e., learning and planning algorithms) for two-dimensional arcade games that take inspiration from early computer games (e.g., Pac-man, Sokoban, The legend of Zelda, etc.). It contains several example algorithms, including two variants of the UCT algorithm, and offers for experimentation a large set of different games – 120 single-player and 60 two-player games to date – with 5 levels for each game.

The games run in real time and enforce a 40-milliseconds time limit per move. During this time, the AI players may determine the best moves by simulating episodes with the help of a forward model of the game. At each moment, the game returns the same observations as would be seen by a human player: the time step, score (a real number), victory status (win, loss, ongoing), a list of available actions, a list of visible objects and their details, and a history of interaction-events. In terms of MDPs (Section 3.2), states can be uniquely identified from the list of current objects and the history of events, whereas rewards can be modelled as changes in the victory status and score. The games have up to six actions and up to hundreds of observation objects per state, and are usually limited to a duration of 2000 time steps (sequential actions).

The General Video Game AI competition

The GVG-AI competition [15, 126] ranks AI algorithms based on their performance on the games from the GVG-AI framework. At each competition event, the algorithms – also refereed to as *controllers* – face a set of 10 new and previously-unseen games and get evaluated according to three goals (metrics): the primary goal of each algorithm is to maximize its win rate (i.e., changing the victory status to "win"), the secondary goal is to maximize its score, and the tertiary goal is to minimize the number of required time steps.

On single-player competitions, the algorithms get executed 500 times in total: 10 times on each of the 5 levels of each of the 10 games. The average of each of the three metrics gets computed for each game from its 50 executions. Then the algorithms get ranked separately for each game, with the secondary metric (score) being a tie-breaker in case the first metric (win rate) equals among more contestants; the tertiary metric (number of time steps) is another tie-breaker in case both the primary and secondary metric equal. Then a Formula-1 score system is applied for each game: the algorithms get awarded points in respect to their rank – from the first to the tenth position the points go as following: 25, 18, 15, 12, 10, 8, 6, 4, 2, and 1. The rest get 0 points. Finally, the points are summed across all games and the algorithms get ranked according to them – the winner is the algorithm with most points.

On two-player competitions, the algorithms are executed in pairs in a round-robin fashion, resulting in 6000 executions per algorithm; the ranking is similar, but uses a Glicko rating (an improvement of the Elo rating).

Two Sarsa-UCT controllers

We develop two Sarsa-UCT(λ) controllers for GVG-AI games: one for single-player competitions (labelled *ToV01*) and one for two-player competitions (labelled *ToV02*). Both are implemented as an extension of the *standard UCT* algorithm provided in the GVG-AI framework (the latter is labelled as the *sampleOLMCTS* controller).

Since we are mainly interested in the benefit of TD backups, we configure the single-player Sarsa-UCT(λ) to fully match UCT ($\gamma = 1$, and $V_{\text{init}} = V_{\text{playout}} = 0$), except for the use of space-local value normalization and the eligibility trace decay rate λ, which is the only control parameter. The exploration rate $C_p = 1$ in both algorithms and is used according to Algorithm 1 (line 54). The algorithm observes neither the

list of objects nor the history, so it does not identify states – it builds a tree based on actions, without using transpositions. It regards action-sequences as (approximate) identifiers of the current state, thus ignoring the possible stochasticity of the underlying task (assuming a deterministic task). Therefore, in stochastic GVG-AI games, multiple successor states that derive from the same sequence of actions get evaluated as one (as mentioned in Section 4.6). The algorithm truncates simulations after 10 plies from the current game state; in this way, it simulates a higher number of shallow episodes rather than few deep ones, which is analogous to a more breadth-first-search behaviour. The tree is discarded after each move and built anew in the next search. After each search, it outputs the action with the highest number of visits. We leave the reward-modelling the same as in the framework-given UCT implementation: the algorithm observes rewards only at the end of simulations and computes them by

$$R = 10^7 \cdot \mathit{victoryStatus} + \mathit{score}\,, \tag{7.1}$$

where *victoryStatus* is either 1 (win), −1 (loss), or 0 (non-terminal state). Before applying the UCB equation, the algorithms normalize the value estimates to $[0, 1]$ based on the minimum and maximum return observed so far.

In contrast to our single-player controller, we configure the two-player Sarsa-UCT(λ) controller to exploit more of its potential. The algorithm observes not only the final reward, but also intermediate rewards (the same as the original UCT algorithm) by computing the difference in score after every game tick. It expands the tree with all the visited nodes in an iteration and retains it between searches. It gradually forgets old knowledge via a more sophisticated update scheme for α and it searches for the shortest path to the solution by discounting rewards with a $\gamma < 1$. The opponent is modelled as a completely random player and is assumed as part of the environment – its value-estimates are not memorized. The scoring considers the win status of the opponent with a weight of 33%, compared to the weight of the own win state. Lastly, the algorithm is augmented with two specifically-designed enhancements related to the MCTS playout phase:

- *Weighted-random playouts*. The algorithm performs a weighted random selection of actions in the playout. The weight of each action is set uniformly randomly at the beginning of each playout. This exploits the two-dimensionality of GVG-AI games, causing the avatar to move further away from its current position – to be more explorative and revisit the same states less often.

- *Dynamic playout length.* The playouts of GVG-AI algorithms are often truncated after a number of actions to produce a higher number of shallow iterations instead of few deep ones. The ToVo2 controller starts each search with short playouts and then it prolongs them as the number of iterations increases. This emphasizes the search in the immediate vicinity of the avatar, and, when there is enough computation time, to search also for more distant goals that might be otherwise out of reach.

Apart from these enhancements, the algorithm uses no prior knowledge and no offline training. During development, we manually tuned the learning parameters with experimentation. We settled on the following parameter values: exploration rate $C_p = \sqrt{2}$, reward discount rate $\gamma = 0.99$, eligibility trace decay rate $\lambda = 0.6$, forgetting 50% of knowledge after each search, and dynamic playout length starting at 5 and increasing by 5 every 5 iterations up to a maximum length of 50.

Local experiments on single-player games

Before participating in the competitions, we tested whether our single-player Sarsa-UCT(λ) controller performs better than the UCT controller from the GVG-AI framework. We use the space-local value normalization also in the latter, but otherwise we leave it equal – this is done to fairly assess the impact of TD-backups by ruling out the impact of value-normalization.

We experimentally measure the win rate and score of the UCT and Sarsa-UCT(λ) algorithms on the GVG-AI single-player training sets 1, 2, and 3 (each set consists of 10 games). The only control parameter is the eligibility trace decay rate λ – we test the values 1, 0.95, 0.9, 0.8, 0.6, 0.3, 0.1, and 0.0. We perform at least 50 repeats for each of the 5 levels of each of the 30 games. The experiments run in real-time on an Intel Core i7-3630 2.40 GHz CPU with 16 GB of 1600 MHz DDR3 RAM; the algorithms produce roughly from 200 to 2000 simulated moves (calls to the StateObservation.advance() procedure in the GVG-AI framework) per real move (i.e., 40 milliseconds), depending on the game.

The results (Table 7.1) reveal that Sarsa-UCT(λ) improves on UCT when using a $\lambda < 1$; the gain in score is higher than the gain in win rate. On this set of 30 games, a λ of 0.6 performs best both in terms of win rate and score, as well as in terms of robustness – it elevates the win rate in 53% of the games, while lowering it in 37% of them, and elevates the score in 70% of the games, while lowering it in 23% of them.

The relative increase in win rate is 6% and in score is 8% in the median across games (not presented in the table).

Table 7.1

Performance of Sarsa-UCT(λ) on the first 30 games of the GVG-AI framework. A setting of $\lambda = 1$ corresponds to the UCT algorithm.

		Eligibility trace decay rate λ							
		1	0.9	0.8	0.6	0.3	0.1	0	Best**
Win rate	Average [%]	32.3	33.0	33.4	33.8	33.0	32.4	8.1	36.1
	Count better*	*n/a*	11	13	16	13	8	0	19
	Count worse*	*n/a*	12	11	11	14	17	26	0
Score	Average	41.8	44.5	46.8	50.9	47.9	41.2	2.2	53.8
	Count better*	*n/a*	14	12	21	16	10	0	23
	Count worse*	*n/a*	13	14	7	12	17	26	0

*Number of games (out of 30) where Sarsa-UCT(λ) performs better/worse than UCT.

**Assuming the most suitable value of λ is used for each game.

Although each value presented above is an average from at least 7500 sample results, we cannot give confidence bounds due to the large variance in win rates and scores across different games. We took this into consideration when performing the statistical analysis and, although not presented here, observing the distributions of results more in detail leads to similar conclusions.

Competition results

Although $\lambda = 0.6$ performed best in local single-player experiments, $\lambda = 0.8$ seemed to perform similarly well on the first test set on the GVG-AI server, so at first we decided to settle for a slightly safer setting (a value closer to 1 is considered safer, as explained in Section 6.2); therefore, we submitted a Sarsa-UCT(0.8) controller to the CIG'15 and CEEC'15 single-player GVG-AI competitions, and Sarsa-UCT(0.6) to subsequent competitions.

So far, the majority of the competitions resulted in favour of our controllers. The

Table 7.2

Results from GVG-AI single-player competitions.

		Single-player competitions				
		CIG'15	CEEC'15	GECCO'16	CIG'16	GECCO'17
Final rank	Sarsa-UCT	11/52	16/55	9/30	26/29	9/22
(a lower value is better)	Standard UCT	29/52	23/55	14/30	20/29	16/22
Ranking points	Sarsa-UCT	35	24	50	8	56
	Standard UCT	8	18	28	22	23
Average win rate [%]	Sarsa-UCT	31.4	11.8	27.6	4.8	26.8
	Standard UCT	29.4	13.4	20.4	2.0	23.2
Average score	Sarsa-UCT	14.6	-37.8	5.6	-34.2	994.9
	Standard UCT	10.4	-33.9	3.9	-22.9	944.4
Count games (out of 10)	Win and score	+6 -4	+4/-6	+8/-2	+4/-5	+4/-4
where Sarsa-UCT is	Win	+4/-2	+0/-3	+6/-1	+1/-2	+4/-3
better/worse than UCT	Score	+8/-2	+5/-4	+8/-2	+4/-5	+2/-6

single-player variant outperformed the framework-given UCT controller in 4 out of 5 competitions (Table 7.2), and, most importantly, our two-player version outperformed all other controllers and ranked first in 2 out of 3 competitions (Table 7.3), and has also been achieving top positions on all two-player training sets on the GVG-AI server so far. It ranked overall third in the 2016 championship, despite the poor performance on the second competition – the latter was due to a too aggressive configuration of the two domain-specific enhancements (that we have not sufficiently tested prior to submission). Also the single-player variant ranked moderately high considering it employs neither expert or prior knowledge nor domain-specific heuristics or features. A similar observation goes with the two-player variant, despite its two domain-inspired enhancements, which are still very general, suggesting there is plenty of room for improvement. Since the single-player variant differs from UCT only in the use of TD-backups, it allows for a direct comparison. On the other hand, the two-player variant is more complex, so it is difficult to assess how much each of its mechanics contributes to its success – on local tests each of them added some performance. The detailed results for the 2015 competitions are given in Appendix B (for the subsequent competitions we direct the reader at the GVG-AI web page).

Table 7.3

Results from GVG-AI two-player competitions.

		Two-player competitions		
		WCCI'16	CIG'16	CEEC'17
Final rank	Sarsa-UCT	1/14	8/13	1/18
(a lower value is better)	Standard UCT	6/14	6/13	10/18
Ranking points	Sarsa-UCT	171	75	161
	Standard UCT	94	85	55
Average win rate [%]	Sarsa-UCT	57.1	40.4	50.5
	Standard UCT	42.3	39.5	39.2
Average score	Sarsa-UCT	1092.5	-0.9	118.9
	Standard UCT	334.6	4.9	25.6
Count games (out of 10)	Win and score	+8/-2	+5/-5	+9/-1
where Sarsa-UCT is	Win	+7/-1	+5/-4	+9/-0
better/worse than UCT	Score	+7/-3	+3/-7	+9/-0

At most single-player competitions our controller achieved a higher win rate than UCT, except at CEEC'15, but even there it ranked higher, because it excelled in games that were difficult for other algorithms, and in this way it collected more ranking points due to the scoring system. On the other hand, despite the higher average win rate, at CIG'16 it ranked worse than UCT; however, there the performance of both was so poor that both ranked among the last – this set of games was particularly difficult for MCTS algorithms without expert enhancements. Considering the overall performance on the GVG-AI games, apart from the TD backups providing a boost when λ is set to an informed value, our controllers have similar strengths and weaknesses as classic MCTS algorithms. Their performance is high in arcade-like games with plenty of rewards (which provide guidance for the algorithms) and poor in puzzle-like games (which require accurate long-term planning); in games where the original UCT performs poorly (i.e., a win rate of 0%), so usually does also Sarsa-UCT. This can also be

observed by comparing the numerous GVG-AI controllers.

With a deeper analysis we discover that the optimal value of λ depends heavily on the played game – it spans from 1 to just above 0, but its overall optimal value is somewhere in between, which is in line with our previous observations on other types of games. Therefore, in some games a $\lambda < 1$ is beneficial, while in others it is detrimental. From a rough assessment, it seems more beneficial for the games from the training set 3, which are defined as "puzzle" games (do not contain NPC's); however, the sample is small and additional validation would be necessary. Nevertheless, if we were able to determine and use the best λ for each individual game, we could achieve a strong improvement over the UCT algorithm (last column in Table 7.1) – effectively overpowering standard UCT completely. Identifying a correct value of λ is crucial, but extremely problem specific – so far we have not yet reliably identified which GVG-AI game features have most impact. The GVG-AI framework aims specifically at providing a repertoire of games that capture as a wide range of environments as possible, and our results confirm that this is true for different values of λ.

Apart from the two minor two-player enhancements, we have not experimented with combining our GVG-AI controller with established MCTS enhancements. Soemers [127], the author of the *MaastCTS2* GVG-AI controller (single-player champion and two-player runner-up in 2016), gained no benefit when integrating our algorithm into his controller. However, his controller has already been heavily enhanced – with progressive history [87], n-gram selection techniques [93], tree reuse, custom-designed evaluation functions, etc. – which combined might have already covered the benefits of TD-backups. Nevertheless, our two-player Sarsa-UCT(λ) outperforms MaastCTS2 at at 3 out 4 game sets (including competitions and test sets), which suggests our algorithm is competitive also against such heavily-enhanced MCTS algorithms.

Discussion and future work

8

We summarize our results and findings, discuss the limitations of our analysis (propose short-term future work), and describe the research directions that we deem most promising (propose long-term future work).

8.1 Findings

The focus of our experiments and analysis was to assess the benefits of swapping Monte Carlo (MC) backups with temporal-difference (TD) learning backups and eligibility traces in MCTSlike algorithms. To do this, we evaluated the performance of several *temporal-difference tree search (TDTS)* algorithms (Section 4.2) – including the original UCT algorithm, the standard UCT algorithm, and the Sarsa-UCT(λ) algorithm (Section 4.3) – under numerous configurations.

We tested the algorithms' value-function quality, policy quality, planning quality, and playing strength in the following scenarios: policy evaluation, policy iteration, learning, planning, single-player tasks, two-player adversary tasks, learning from self-play, toy games, classic games, and real-time arcade video games (where the low amount of time per move forces quick decision-making). We also competed with two Sarsa-UCT(λ) algorithms at the General Video Game AI competition [15].

Performance

Computing TD backups proved better in most of the scenarios listed above and under most configurations of the algorithms. The increase in performance was most notable on tasks that impose a low number of simulations per decision. It seems that preferring actions that lead to winning a game in a smaller number of moves increases the probability of winning (at least in average, in the games we analysed), and bootstrapping backups ($\lambda < 1$) cause shorter playouts to have more impact in updating the value estimates than longer playouts (Section 6.3), hence their advantage. To the contrary, when using MC backups ($\lambda = 1$), the playout lengths have no impact on the state-value updates; due to this, different states may thus have equal values (especially in the early stages of a game) and ties have to be broken randomly, which leads to selecting sub-optimal moves more often.

Algorithms that use eligibility traces are inherently sensitive to the value of λ. What is the optimal value of λ in general is a well-known research question in the field of RL [9] and fully answering it is not in the scope of this study. Nevertheless, we observe that using a λ close to 1 performs well in practice, but it also produces the

least improvement over $\lambda = 1$. The optimal value of λ differs from task to task: it depends heavily on the size of the search-space, and, most critically, on the number of computed simulations per move (i.e., the number of MCTS iterations per search). In our toy-game experiments, the gain of bootstrapping backups ($\lambda < 1$) increases both when the task diminishes in size and when the algorithms are allowed more simulation time. However, our two-player-game experiments contradict with the above: in this case the gain decreases when more time is available per move. We are not yet certain what causes this discrepancy between the two classes of games. Additional research of how incremental representations and playout concepts impact the convergence rate of general RL methods might clarify this question.

When both the task and the number of simulations per move are fixed, it is not difficult to identify the optimal λ (e.g., with experimentation); however, when the number of simulations is not fixed or cannot be predicted, as in many real applications where the time limit is not constant, then a constant value of $\lambda < 1$ might perform significantly worse than a $\lambda = 1$. Due to this, in practice, such algorithms may have difficulties on tasks where the number of simulations per move is variable, unless the value of λ is left on 1 (the safe option) or is somehow adapted online (which we do not yet fully understand how to achieve). We deem this the main disadvantage of TDTS methods.

Initial values $V_{\text{init}}(s)$ and $Q_{\text{init}}(s,a)$ have no impact when the starting value of the update step-size α is 1 and when $\lambda = 1$; however, the lower the values of these two parameters, the bigger the impact of initial values. In such case, initial values might significantly impact the convergence rate – either improve it when set to an informed value (e.g., through expert knowledge and heuristics) or detriment it when set far from optimal. Regardless, their optimal value usually depends only on the given task, and not so much on the number of simulations per move. Therefore, getting them (at least approximately) right is usually easier than finding an optimal λ; and the same is true also for the rest of the parameters.

Best configuration on toy games

We tested which playout value assumptions, control policies, and value-normalization techniques perform best. The playout value assumption $V_{\text{playout}} = V_{\text{init}}$ proved better than $V_{\text{playout}} = 0$ and better than some other assumptions that we experimented with. When using the UCB1 policy with common value-normalization methods (that is,

a constant value of the exploration rate C_p or normalization through a global upper and lower bound of the returns), the improvement of TD backups is much smaller, and the UCB1 policy often performs worse than the ε-greedy policy. On the other hand, when normalizing the value estimates also locally (for example, with our space-local value normalization technique), the improvement is considerably large, and the UCB1 policy performs equal or better than ε-greedy; however, we stress that we did not optimize the parameters of the two policies, so we cannot claim which one performs better in general. Lastly, we experimented with different variants of space-local normalization, but the one we presented in Section 4.4 performed best.

Computational complexity

Swapping MC backups with TD backups and eligibility traces usually increases the computational cost of an algorithm only by a constant factor that is usually negligible. This is in contrast with Silver et al.[41], who report that their TD search algorithm applied to Go performs significantly slower than vanilla UCT; however, this is reasonable, because they used complex heuristic and handcrafted policies and value-function approximation with features that must be computed from the game state, all of which is computationally expensive. In essence, TDTS methods – and RL methods in general – are *not* necessarily slower than MCTS methods (nor TD backups slower than MC backups), but they have numerous implementational choices (some of which could also be understood as enhancements) that improve the convergence rate by better exploiting the gathered feedback at the expense of an increased computational cost – ample details about this were given in Section 4.6.

8.2 Limitations of our analysis

Several mechanics that we have not assessed might seriously impact the performance of TDTS algorithms, and might even additionally increase it.

RL methods discount rewards (i.e., use a $\gamma < 1$) on continuous tasks to guarantee convergence, nonetheless, doing so on episodic tasks also often improves their performance [9]. We carried out only few rough experiments on toy games and GVG-AI games, where we observed that the effects of γ might be similar to those of the eligibility trace decay rate λ, but a more robust evaluation is necessary.

On toy games we observed that a constant update step-size α performs worse than a MC-like decreasing α (Section 3.5), as is usually the case with small and station-

ary tasks. On more complex and large tasks, however, schemes for adapting α online often perform better – we have only touched this topic (in our two-player GVG-AI controller) and have not yet systematically studied it. For example, linear or exponential schemes that give higher weight to recent rewards while still decreasing α towards 0 in the limit might be sensible, since the control policy improves with time and therefore recent rewards might be more relevant than old ones. Such "forgetting" schemes from the field of MCTS (e.g., [89, 92]) and RL (e.g., [128]) might further improve the performance of TDTS algorithms.

When learning from self-play on two-player games, we configured the TDTS algorithms to compute backups from one-ply successor states. Silver et al. [41] report significantly better performance when using a two-ply TD update instead, therefore, it might be worth verifying how this affects TDTS methods in general.

Lastly, our methods should be evaluated on more complex games and of different types (e.g., multi-player, simultaneous, with imperfect information, etc.), with a higher number of iterations per move, and with variable time per move, to better asses the benefits of bootstrapping updates (i.e., $\lambda < 1$) also in such scenarios. Here we suggest to evaluate the performance of the algorithms not only through self-play (as we did in our two-player games), but also through "go-deep" experiments [129]; the main focus of such experiments is to measure how quickly with increasing the available computational time the algorithms achieve optimal (or near-optimal) action-selection in specific (carefully-picked) situations – this might provide insightful comparison on the learning rates of the new algorithms and on their sensitivity to parameter values.

8.3 *Promising directions*

Beside resolving the limitations of our experiments (as presented above), and studying the impact of several implementational options (as given in Section 4.6), there is also plenty of unexplored potential deriving from the central ideas presented in this thesis. We roughly split the promising long-term research goals in three directions: encouraging cross-fertilization between MCTS and RL, understanding better the impact of representation policies on reinforcement learning methods in general, and identifying the best control policy when using explicit incremental representations in RL methods.

Cross-fertilization between MCTS and RL

The changes introduced to the basic MCTS algorithm by Sarsa-UCT(λ) are generic and can be combined (to the best of our knowledge) with any known enhancement. This is because the introduced update method is principled: the new parameters closely follow the reinforcement learning methodology – they can formally describe concepts such as forgetting, first-visit updating, discounting, initial bias, and other (Section 4.5). As such, Sarsa-UCT(λ) is not better or worse than any other enhancement, but it is meant to complement other approaches, especially ones that embed some knowledge into the tree search, which can be done in Sarsa-UCT(λ) as easily as in UCT or any other basic MCTS algorithm. Therefore, it can likely benefit from enhancements that influence the tree, expansion, and playout phases, from generalization techniques (such as transpositions or MAST [23], as shown earlier, for example), and from position-evaluation functions (which are well-known also to the RL community) and other integrations of domain-specific heuristics or expert knowledge. It could benefit even from other MCTS backpropagation enhancements, although in such case it might be reasonable to keep separate estimates for each backup method, to retain the convergence guarantees of the RL-based backups.

Since the TDTS algorithms are interchangeable with the original MCTS algorithms, extending the existing MCTS algorithms (and MCTS-based game-playing engines) should not be difficult, and, most importantly, the extension is "safe", because the resulting algorithms would perform at least equally well (when $\lambda = 1$) or possibly better (when $\lambda < 1$). For example, in our experiments the algorithms always employed a random playout control policy, and without any handcrafted, heuristic, expert, or prior knowledge; however, it would be meaningful to evaluate them in combination with all these enhancements – this could be done by integrating them into state-of-the-art game-playing algorithms that use MCTS, such as *AlphaGo* [6], *MoHex* [26], or *CadiaPlayer* [24], for example.

Also, we presented an extension of the UCT algorithm with the *Sarsa(λ)* algorithm; however it would be interesting to extend it with other established RL algorithms instead. Sarsa-UCT(λ) is an on-policy method and might diverge when performing backups differently than we devised it to, e.g., if it was implemented to back up the value of the best children instead of the last feedback. Hence, combining off-policy TD learning methods, such as Q-learning [49], with MCTS-like incremental repre-

sentations might also be interesting. There have already been successful applications of similar types of backups in MCTS [4].

Finally, in our experiments and analysis we focused on the benefits of RL concepts for MCTS; however, there is also potential to explore in the other direction – to investigate further the benefits of MCTS concepts for RL. We suggest to take example of strong heuristic (and specialized) control policies and ingenious playout-related generalization approaches from the numerous MCTS enhancements. For example, the use of explicit incremental representations and playout control policies in RL methods is uncommon in general. There is little known about combining off-policy TD learning methods with MCTS-like representations and tree search, but given the successes of off-policy RL methods [130] and recent off-policy MCTS methods [55, 100], it certainly deserves attention. Furthermore, it would also be interesting to integrate such algorithms into established frameworks such as *Dyna* [71] or *Dyna-2* [98], which already provide efficient mechanics for introducing initial bias in the estimates and for transferring knowledge between searches through the concepts of a long-term (permanent) and short-term (transient) memory.

Representation policies and their impact on reinforcement learning methods

The concept of a *representation policy* (Section 4.1) introduces novel MCTS dynamics into RL methods, namely, a policy for adapting the state representation online, and the concept of a playout. The convergence of MC learning methods under such conditions has already been proved [35], but what about other RL methods? What is the effect of not using transpositions – how much does this relate to partially observable MDPs [46], and what insights can be taken from that field? How well do these concepts integrate into more complex, approximate, or non-incremental representations? What is the impact of different playout-value assumptions – what kind of (more sophisticated) assumptions might be required to guarantee convergence?

The questions above open a whole new research dimension for both MCTS and RL methods. Theoretical insight in how these concepts impact the convergence rate and requirements of RL methods in general might help to better understand what defines the optimal value of the learning parameters, such as the eligibility trace decay rate λ, to which our algorithms are especially sensitive. Exploring schemes for adapting λ online, for example, based on the number of simulated time steps per move, might alleviate the main drawback of TDTS algorithms. A first step in this direction might be

describing other, more complex, MCTS methods from a RL point of view (describing them through the TDTS framework) in the same way as we did with basic MCTS methods (i.e., the UCT algorithm) in this thesis. In would be interesting to analyse the existing MCTS-related off-policy dynamic programming methods (e.g., *MaxUCT* [100] and *BRUE* [55]) from this perspective; which could also be easily enhanced with TD backups and eligibility traces.

Additionally, it would be interesting to examine MCTS or RL algorithms with representations that are not directly related to the two fields, such as feature identification [61] or state aggregation [60] methods. The existing convergence proofs of these specific methods might help to analyse the dynamics of such learning algorithms. Also, when using transpositions, it might be worth exploring other replacement schemes [56] for memorizing estimates, or devising new ones; for example on some tasks in might be sensible to memorize a limited number of states not only close to the starting state, but also close to a terminal state, or memorize only such states that display some heuristic qualities.

Best control policy when using incremental representations

What is the best control policy for the general reinforcement learning problem is an open question by itself [42]; and what policy is best when employing explicit incremental (adaptive) representations might have never been even considered before. Although very simple, the ε-greedy selection policy is widely used among traditional RL methods as it often performs very well. Also Silver [10] reported that no policy, including UCB1, performed better than an ε-greedy policy in his experiments with the TD search algorithm. On the other hand, UCB-based policies, despite regarded as difficult to apply to the general reinforcement learning problem [42], have recently started being adopted by RL practitioners [51, 52]. Also in our experiments, UCB1 proved to perform equally well or better than ε-greedy policies when it is combined with a rather simple normalization technique (Section 4.4), even when the algorithm employs TD learning and eligibility traces. This suggest that such UCB-oriented normalization approaches are promising. Analysing them on more complex algorithms (e.g., on such that tackle non-stationary tasks or that use function approximation) might further popularize UCB policies also outside the MCTS field and might help clarify when it is meaningful to use them. To better deal with non-stationary tasks, our normalization technique could be extended with "forgetting" dynamics to gradually adapt the nor-

malization bounds instead of only memorizing the all-time minimum and maximum value estimates.

Lastly, we also encourage studying other selection policies in such settings. For example, policies based on Thompson sampling [131] that bootstrap [132] are emerging alongside contextual-bandit problems as an alternative to UCB and ε-greedy selection policies for (deep) exploration in complex representations (such as neural networks) [133].

Conclusion

9

The goal of this thesis was to fundamentally improve the Monte Carlo tree search paradigm through general, domain-independent enhancements. We achieved this by resorting to the established field of reinforcement learning.

Summary

In this work we thoroughly examined the relation between Monte Carlo tree search (MCTS) methods and reinforcement learning (RL) methods. We described MCTS in terms of RL concepts and re-exposed the similarities between the two fields, but we also explicitly identified and emphasized the differences between the two fields – the novelties that MCTS methods introduce into RL theory. We also identified a large number of existing MCTS algorithms and enhancements that resemble, are related to, or re-observe traditional RL dynamics. We outlined that many MCTS methods evaluate states in the same way as the TD(1) learning algorithm and in general behave similarly to Monte Carlo control and the Sarsa(1) algorithm. We observed that the RL theory is able to better describe and generalize the learning aspects of MCTS (i.e., its backpropagation phase), but, in turn, the MCTS field introduces playout-related mechanics that are unusual for classic RL theory.

With this insight, we first introduced for RL the concepts of *representation policies* and *playout value functions*, which allow to fully describe all the phases of an MCTS iteration, while still abiding to the RL theory. To promote such a unified view of both fields, as a proof of concept we integrated temporal-difference (TD) learning and eligibility traces into MCTS and devised the *temporal-difference tree search (TDTS)* framework. The latter classifies as a specific configuration of Silver's TD search method [10], but also extends it with the novel concepts from MCTS. The parameters and implementational options of TDTS can reproduce a vast range of MCTS enhancements and RL algorithms, illustrating its power. To showcase a TDTS algorithm, we merged Sarsa(λ) and UCT into the *Sarsa-UCT(λ)* algorithm. Simultaneously, we also devised an effective *space-local value normalization* technique for the convergence needs of UCB-based control policies in combination with general RL methods; without it, the UCB1 policy performs poorly when combined with TD updates and eligibility traces. From an MCTS perspective, our new algorithms retain the robustness and computational cost of the UCT algorithm, while improving on its performance; our experiments confirmed this on several types of tasks: on single-player toy games, two-player classic games when learning from self-play, and real-time arcade video games

with low computational time per move.

Importance and impact

This thesis confirms the benefits of extending MCTS with RL dynamics and presents a practical way for achieving this, improving the theoretical understanding of MCTS methods and hopefully narrowing the gap between the TD search and MCTS frameworks. It offers a wealth of unexplored potential, since any MCTS algorithm can be generalized by any RL method in a similar way as we have done; or the other way around – any RL method can be extended into an MCTS-like algorithm by combining it with an incremental or adaptive representation. Moreover, a great advantage of merging algorithms in this way is that the generalized algorithm cannot have a lower performance – in the worst-case the new parameters can be set to mimic the behaviour of the original algorithm. If we managed to intrigue the MCTS reader to explore the RL view on planning and search [42] or the RL reader to experiment with any of the numerous MCTS enhancements [5], including ingenious generalization techniques, specialized control policies, and incremental representations, then we fulfilled our goal.

Finally, we succeed in generally improving Monte Carlo tree search, but we go even beyond: we support that the fields of reinforcement learning and heuristic search address a similar class of problems, only from different points of view – they overlap to a large extent. Hence, rather than perceiving the RL interpretation of MCTS as an alternative, we suggest to perceive it as complementary. Exactly this line of thought allowed us to pinpoint the similarities and differences of both, and to combine their advantages, of which our findings are proof. We regard this a step towards a unified view of learning, planning, and search.

Detailed results from two-player games

A

Table A.1

Sarsa-UCT(λ) on two-player games: full results, including the best values of C_p and λ found in the optimization process. The algorithm did not employ value normalization. The matches were played from both starting positions. Draws count towards a 50% win rate.

Game	Simulated time steps per move	Win rate [%] at best λ	Best λ	Own C_p	Opponent C_p	Num. of matches
Tic-tac-toe	10	58.4	0.10–0.99	0.1	0.2	10000
	50.0	55.4	0.6	0.1	0.2	2000
	100	55	0.65	0.2	0.2	40000
	200	52.3	0.8	0.3	0.2	2000
	500	50.4	0.999	0.25	0.25	240000
	1000	50.1	0.999–1.0	0.3	0.3	60000
Connect four	100	79.7	0.10–0.99	0.25	0.05	2000
	500	87.5	0.10–0.99	0.25	0.25	2000
	1000	67.2	0.65	0.15	0.05	40000
	2000	53.3	0.999	0.25	0.05	2000
	5000	50.8	0.999–1.0	0.15	0.15	12000
	10000	50.1	0.999–1.0	0.2	0.2	2000
Hex 7×7	1000	71.7	0.95	0.05	0.25	20000
	2000	61.4	0.95	0.2	0.2	2000
	5000	55.5	0.8–0.95	0.2	0.2	6000
	10000	50.3	0.999–1.0	0.2	0.2	2000
Hex 11×11	1000	87.7	0.8–0.9999	0.15	0.15	12000
	10000	62.0	0.95–0.9999	0.1	0.1	400
	100000	50.0	0.999–1.0	0.25	0.25	200
Gomoku 7×7	1000	77.7	0.7	0.1	0.1	4000
	2000	68.8	0.9	0.1	0.1	2000
	5000	58.3	0.95	0.15	0.15	2000
	10000	51.4	0.99–1.0	0.15	0.2	2000
Gomoku 9×9	1000	95.2	0.4–0.9999	0.25	0.25	2000
	10000	67.2	0.9–0.95	0.1	0.1	200
	100000	54.0	0.999–0.9999	0.2	0.2	400
Gomoku 11×11	1000	96.0	0.1–0.9999	0.15	0.15	2000
	2000	97.6	0.1–0.9999	0.15	0.15	2000
	10000	84.0	0.4–0.9	0.15	0.15	1000
	20000	75.0	0.6–0.9	0.05	0.05	800
	50000	73.5	0.95	0.05	0.05	200
	100000	71.0	0.95	0.05	0.05	4000
	1000000	53.2	0.9999–1.0	0.2	0.2	200
Gomoku 13×13	10000	94.0	0.9–0.95	0.2	0.2	400
	100000	77.0	0.85	0.01	0.01	400

Detailed results from the GVG-AI 2015 competitions

B

Table B.1

Detailed results from the GVG-AI 2015 comptetitions.

		Win rate [%]		Score		Rank points	
		Sarsa-UCT(λ)	UCT	Sarsa-UCT(λ)	UCT	Sarsa-UCT(λ)	UCT
CIG competition (λ = 0.8)	Game 51	2	4	62.7	49.3		1
	Game 52	100	96	52.6	40.1	25	6
	Game 53	2	8	2.8	1.4		
	Game 54	0	0	0.7	0.8		
	Game 55	0	0	0.4	0.5		
	Game 56	86	76	0.9	0.8	10	1
	Game 57	100	100	10.0	1.7		
	Game 58	0	0	5.4	4.5		
	Game 59	12	10	0.1	0.1		
	Game 60	12	0	10.1	5.0		
CEEC competition (λ = 0.8)	Game 71	4	6	0.0	0.0		
	Game 72	8	14	1.9	1.7		
	Game 73	100	100	6.6	8.9		
	Game 74	0	0	−136.0	−57.6		
	Game 75	0	0	−12.9	−15.5		
	Game 76	0	0	−219.9	−299.9		
	Game 77	0	0	0.2	−0.3	18	
	Game 78	6	14	−1.9	20.1	6	18
	Game 79	0	0	0.3	0.1		
	Game 80	0	0	−16.1	3.0		
	Average	21.6	21.4	−11.6	−11.8		
	Count better*	4		13			
	Count worse*	5		6			

*Number of games (out of 20) where Sarsa-UCT(0.8) performs better/worse than UCT.

Razširjeni povzetek

C

Leta 1997 je superračunalnik Deep Blue v šahovskem dvoboju premagal svetovnega prvaka Kasparova [1]. Kmalu zatem je naslednji velik izziv umetne inteligence pri igranju iger začela postajati starodavna azijska igra *go*. Vse metode, ki so bile v preteklosti z izjemnimi uspehi uporabljene pri drugih igrah (na primer, pri šahu, dami in backgammonu), so bile pri igri go neučinkovite – vse do leta 2006 so najboljši algoritmi igrali na ravni človeških začetnikov [2]. Takrat je preboj naredila nova preiskovalna paradigma, danes znana kot *drevesno preiskovanje Monte Carlo* (ang. Monte Carlo Tree Search – MCTS) [4], ki je dvignila sposobnosti računalniških igralcev go na raven mojstrov [3]. Novo področje je hitro pridobilo znanstveno pozornost in bilo uspešno aplicirano na širokem spektru iger ter tudi drugje: danes poznamo več kot 100 različnih algoritmov MCTS in uporabljajo jih praktično vsi najboljši algoritmi za igranje iger go, hex, othello in drugih iger[5]. Preteklo je le še eno desetletje in padel je tudi naslednji velik izziv umetne inteligence – algoritem *AlphaGo* [6] je s pomočjo globokih nevronskih mrež [7] in drevesnega preiskovanja Monte Carlo maja 2017 premagal z izidom tri proti nič najboljšega igralca go-ja na svetu.

Metode MCTS združujejo splošnost naključnega vzorčenja in natančnost drevesnega preiskovanja. Ob predpostavki, da ima preiskovalni algoritem na razpolago simulacijski model danega problema, te metode dosegajo bistveno boljše rezultate kot tradicionalne metode preiskovanja. Kljub temu pa imajo lahko v praksi težave s počasno konvergenco, kar še posebej drži za temeljne algoritme MCTS, ki se ne poslužujejo dodatnih izboljšav: primer je *algoritem UCT* [8], ki v praksi velja za kanonični algoritem MCTS. Zaradi tega jih raziskovalci in programerji pogosto nadgrajujejo z bolj kompleksnimi, domensko-naravnanimi pristopi – z ekspertnim znanjem, hevristikami in ročno-izdelanimi politikami. Posledično, takšne specifične izboljšave zmanjšujejo splošnost številnih aplikativnih algoritmov MCTS in poleg tega so lahko računsko zelo zahtevne, njihov doprinos pa zelo variira od problema do problema. Izboljšava temeljnih algoritmov MCTS, brez izgube njihove splošnosti in prilagodljivosti, se je izkazala za težavno, zato je to eden od aktualnih raziskovalnih izzivov tega področja.

Naša disertacija razkriva, da lahko k reševanju tega izziva pripomore starejše in uveljavljeno področje spodbujevalnega učenja (ang. reinforcement learning) [9]. Naš cilj je izboljšati temeljno razumevanje drevesnega preiskovanja Monte Carlo s pomočjo konceptov iz spodbujevalnega učenja in s tem odkriti nove možnosti za domensko-neodvisne izboljšave temeljnih algoritmov MCTS. Sočasno tudi želimo zbližati raziskovalne skupnosti teh dveh področij, tako da pokažemo prednosti bolj enotnega po-

gleda na obe področji in tako da ponudimo praktičen način implementacije takšnih algoritmov.

C.1 Prispevki k znanosti

V tej doktorski disertaciji so podani naslednji izvirni prispevki k znanosti;

- *Napredek temeljnega razumevanja drevesnega preiskovanja Monte Carlo v luči teorije spodbujevalnega učenja.* Identifikacija in analiza podobnosti ter razlik med tema dvema področjema. Pregled obstoječih razširitev metod MCTS, ki poustvarjajo temeljne mehanizme spodbujevalnega učenja. Pregled raziskav na področju MCTS, ki vede ali nevede ponovno odkrivajo koncepte spodbujevalnega učenja, le da iz druge perspektive.

- *Združitev konceptov drevesnega preiskovanja Monte Carlo in spodbujevalnega učenja v enotno ogrodje.* Razširitev temeljnih konceptov spodbujevalnega učenja z novimi koncepti iz področja MCTS: uvedba *predstavitvene politike* (ang. representation policy) ter razločevanje dela preiskovalnega prostora, ki ga algoritem ne hrani v pomnilniku (ki ga trenutna predstavitev ne opisuje). Uvedba dodatnih predpostavk glede stanj in akcij, ki niso hranjene v pomnilniku – uvedba *vrednostne funkcije v odigravanju* (ang. playout value function). Uvedba metode *drevesnega preiskovanja s časovnimi razlikami* (ang. temporal difference tree search), ki združuje karakteristike obeh področij.

- *Posplošitev algoritma UCT z uveljavljeno metodo učenja s časovnimi razlikami.* V dokaz zmožnosti nove metode drevesnega preiskovanja s časovnimi razlikami smo razvili algoritem *Sarsa-UCT(λ)*, ki združuje drevesno preiskovanje po principu MCTS in algoritem Sarsa(λ) iz spodbujevalnega učenja. Potencial novega algoritma smo eksperimentalno potrdili na treh tipih iger: enostavne igre za enega igralca, klasične igre za dva igralca in arkadne video igre.

- *Metoda za normiranje vrednosti, ki omogoča uporabo politik UCB za izbiro akcij znotraj algoritmov spodbujevalnega učenja.* Algoritem UCT za izbiro akcij uporablja politiko za mnogoroke bandite UCB, ki zahteva normirane vrednosti akcij. Uvajamo novo metodo normiranja z imenom *prostorsko-lokalno normiranje vrednosti* (ang. space-local value normalization), ki omogoča uporabo politik UCB v kombinaciji s poljubnimi algoritmi spodbujevalnega učenja.

C.2 Drevesno preiskovanje Monte Carlo

Metode MCTS simulirajo naključna zaporedja akcij z namenom pridobivanja znanja za gradnjo preiskovalnega drevesa. Drevo razraščajo asimetrično v najbolj obetavni raziskovalni smeri. Njihove glavne prednosti so, da lahko izstavijo rezultat ob vsakem trenutku izvajanja, da imajo relativno nizko občutljivost na velike prostore stanj in da ne potrebujejo specifičnega znanja problema, če pa je le-ta na razpolago, ga lahko učinkovito izkoristijo. Uporabne so pri vseh problemih, ki jih lahko prevedemo na zaporedje odločitev, na primer, kombinatorična optimizacija, planiranje, vodenje sistemov v realnem času, ipd.

Pri metodah MCTS je značilen iterativni postopek vzorčenja, kjer vsaka iteracija v splošnem sestoji iz štirih glavnih faz v naslednjem vrstnem redu:

1. *sestop* po drevesu oz. po predstavitvi prostora stanj, ki je hranjena v pomnilniku, dokler algoritem ne doseže ali končnega stanja problema ali pa se znajde v delu prostora, ki ga ne hrani v pomnilniku;
2. *odigravanje* do končnega stanja, kjer se pridobi končno povratno informacijo;
3. *širitev* drevesa z novimi vozlišči, ki predstavljajo akcije ali stanja (širitev predstavitve v pomnilniku); in
4. *vzvratni prenos* prejete povratne informacije nazaj po drevesu do začetnega stanja (do korena drevesa).

Prvi fazo usmerja *drevesna politika* (ang. tree policy) izbire akcij, drugo fazo pa *privzeta politika* (ang. default policy) izbire akcij. Čeprav osnovni algoritmi MCTS gradijo preiskovalno *drevo*, jih veliko dosega boljše rezultate, če gradijo usmerjen *graf* na podlagi *transpozicij* [31].

Trenutno najbolj razširjen in splošno-učinkovit algoritem na področju MCTS se imenuje UCT (ang. upper confidence bounds applied to trees), ki sta ga leta 2006 razvila Kocsis in Szepesvari [8]. Drevesno preiskovanje sta združila z asimptotično-optimalno politiko *UCB1* (ang. upper confidence bounds) [13] za mnogoroke bandite (ang. multi-armed bandits) iz teorije iger. Coulom [4] je sočasno razvil učinkovite metode gradnje drevesa in prenosa informacije po njem. Gelly in Silver [28] sta med prvimi uspešno integrirala ekspertno znanje v njun algoritem MCTS in naredila preboj pri igri go. Browne in sod. [5] so pripravili najbolj temeljit pregled področja do danes.

Algoritem UCT uporablja drevesno politiko, ki v vsakem vozlišču drevesa izbere naslednjo akcijo z največjo oceno po enačbi:

$$Q_{\mathrm{UCT}} = Q_{\mathrm{MC}} + c_{n_{\mathrm{p}},n} \,, \tag{C.1}$$

kjer

$$Q_{\mathrm{MC}} = \frac{\sum G_i}{n} \tag{C.2}$$

predstavlja povprečje vrednosti povratnih informacij G, ki jih je algoritem nabral, ko je obiskal določeno vozlišče (stanje ali akcijo) tekom več iteracij i in kjer

$$c_{n_{\mathrm{p}},n} = C_p \sqrt{\frac{2 \ln n_{\mathrm{p}}}{n}} \tag{C.3}$$

predstavlja *raziskovalni potencial* posamezne akcije; potencial je definiran s številom obiskov naslednjega vozlišča n, s številom obiskov trenutnega vozlišča n_p ter z utežjo $C_p \geq 0$. Slednja uravnava raziskovalnost algoritma: na primer, $C_p = 0$ naredi algoritem požrešen, večanje vrednosti C_p pa dela algoritem bolj naključen. Zgornje enačbe temeljijo na politiki izbire akcij UCB1 [13], ki v limiti dokazano optimalno rešuje problem raziskovanja prostora in izkoriščanja znanja (ang. the exploration-exploitation dilemma). Za pravilno konvergenco potrebuje vrednosti Q_{MC} normirane v območju $[0, 1]$.

C.3 Spodbujevalno učenje

Spodbujevalno učenje opisuje učeče agente, ki s poskušanjem ugotavljajo, kako se "dobro" obnašati v danem okolju. Uveljavljeno je kot ena najbolj splošnih paradigem učenja, saj se tovrstni algoritmi lahko učinkovito učijo iz enostavne povratne informacije v obliki skalarnega signala. Naše razumevanje tega področja izhaja iz življenjskega dela Suttona in Barta [9, 42] ter iz pregleda najsodobnejših algoritmov spodbujevalnega učenja, ki sta ga pripravila Wiering in Otterlo [47].

V osnovnem modelu spodbujevalnega učenja nastopa *agent*, ki izvaja *akcije* v danem *okolju*. Ob tem agent opazuje trenutno *stanje* okolja in prejema povratno informacijo v obliki *nagrade* (ang. reward). Agent uporablja nagrade (ki so lahko tudi negativne – kazni) za spreminjanje svojega *obnašanja* tako, da skuša maksimirati prejeto nagrado – agent ugotavlja, katere akcije v katerih stanjih ga lahko privedejo do najvišje skupne

nagrade v prihodnosti. Nagrada, do katere pripelje določena akcija, je lahko časovno zakasnjena. Zato je problem spodbujevalnega učenja težji od problema *nadzorovanega učenja*, kjer učeči agent za vsako izvedeno akcijo dobi neposredno povratno informacijo v obliki optimalne akcije. Pri spodbujevalnem učenju mora agent sam identificirati najboljše akcije preko učenja z raziskovanjem ali preko analize modela okolja, če je ta na razpolago. Agent se uči *politike izbire akcij* (ang. action-selection policy), ki določa verjetnosti izbire akcij za vsa stanja preiskovalnega prostora. Ob tem si pomaga z gradnjo *vrednostne funkcije*, ki ovrednoti vsako stanje glede na pričakovano količino prejete nagrade v prihodnosti, če bi se agent znašel v tem stanju. Če se agent nauči dobre (ali optimalne) vrednostne funkcije, lahko iz nje običajno na trivialen način izlušči tudi optimalno politiko.

Poznamo tri glavne skupine metod, ki lahko rešijo problem spodbujevalnega učenja: *dinamično programiranje* [12], *metode vzorčenja po Monte Carlu* [48] in metode *učenja s časovnimi razlikami* (ang. temporal-difference (TD) learning) [11]. Metode vzorčenja po Monte Carlu in metode učenja s časovnimi razlikami se lahko učijo iz *izkušenj* – iz vzorčnih interakcij z okoljem – kar pomeni, da ne potrebujejo predhodnega poznavanja okolja, temveč lahko vzorce naberejo sprotno med učenjem, ali pa jih simulirajo, če imajo na voljo vsaj simulacijski model okolja. Metode učenja s časovnimi razlikami razumemo kot posplošitev metod Monte Carlo, saj združujejo prednosti dinamičnega programiranja in vzorčenja po Monte Carlu ter tako v praksi pogosto dosegajo hitrejšo konvergenco k optimalni rešitvi [9]. Med najbolj znane algoritme učenja s časovnimi razlikami uvrščamo TD(λ) [11], Q-učenje [49] in Sarsa(λ) [14].

C.4 Pregled literature

Drevesno preiskovanje Monte Carlo je bilo vpeljano kot metoda za planiranje s kombinacijo mnogorokih banditov [13] in drevesnega preiskovanja [8]. Kmalu po njegovem izumu je začela postajati razvidna vez s spodbujevalnim učenjem [10]. Kljub temu pa odnos med obema področjema še ni bil temeljito analiziran in posledično tudi še ni bistveno vplival na raziskovalno skupnost. Nekateri raziskovalci so začeli ponovno odkrivati koncepte spodbujevalnega učenja, namesto da bi bili o njih ozaveščeni in si z njimi pomagali pri razvoju bolj zmogljivih algoritmov.

Naša glavna referenca so raziskave, ki jih je vodil David Silver (ki je pozneje postal vodilni avtor algoritma AlphaGo [6]). Skupaj s sodelavci [28] je prvi obelodanil povezavo med preiskovalnimi mehanizmi metod MCTS in mehanizmi spodbujevalnega

učenja [98]. Ugotovitve je uporabil za razvoj metode *preiskovanja s časovnimi razlikami* (ang. temporal-difference search) [41]. Metode, ki jih razvijamo mi, se uvrščajo pod metode preiskovanja s časovnimi razlikami, toda sočasno jih posplošujejo z novimi koncepti, vezanimi na MCTS. Zaradi tega naše metode ohranjajo več lastnosti (splošnost in učinkovitost) osnovnih metod MCTS in jih je lažje implementirati v obliki razširitev obstoječih metod MCTS.

Številni raziskovalci so bodisi namerno ali nenamerno dodajali mehanizme spodbujevalnega učenja metodam MCTS. Feldman in Domshlak [55, 102, 103] sta v njuni metodi *MCTS2e* ponovno odkrila številne temeljne koncepte spodbujevalnega učenja [9, 71]. Keller in Helmert [100] predlagata enotno ogrodje za hevristično drevesno preiskovanje na podlagi vzorčenja – njun algoritem *MaxUCT* je podoben *asinhroni iteraciji vrednosti* [9] s sprotnim učenjem modela, kar se klasificira kot *adaptivno realno-časovno dinamično programiranje* [101] (podskupina ogrodja Dyna [71]); takšni algoritmi so tesno povezani s Q-učenjem [49]. Hester in Stone [104, 105] sta razširila algoritem UCT s koncepti spodbujevalnega učenja in nov algoritem poimenovala *UCT(λ)*. Slednjega lahko opišemo kot *naivni* $Q(\lambda)$ [9], ki uporablja politiko UCB1 [13]. Drugi raziskovalci, ki so zavestno izhajali iz teorije spodbujevalnega učenja pri razvoju novih algoritmov MCTS, so Veness in sod. [107] (algoritem ρUCT), Asmuth in Littman [40] (algoritem *BFS3*), Wang in Sebag [111] (algoritem *MOMCTS*), Osaki in sod. [118] (algoritem *TDMC(λ)*) ter Baier [43], ki sicer ni razvijal kombiniranih metod, toda v svojem uvodnem delu doktorske disertacije izpostavi prisotnost spodbujevalnega učenja v drevesnem preiskovanju Monte Carlo.

Navajamo obstoječe variante in izboljšave metod MCTS, pri katerih smo ugotovili močno povezavo s temeljnimi koncepti spodbujevalnega učenja: spreminjanje teže nagrad oz. povratne informacije [8, 84, 85], prirejanje začetne vrednosti vozliščem (stanjem ali akcijam) na podlagi predhodnega znanja ali hevristik [18, 23, 28, 38, 39, 57, 87, 88] ter pozabljanje znanja [55, 57, 89, 91, 92, 94].

C.5 Podobnosti in razlike med področjema

S poglobljeno teoretično analizo smo identificirali podobnosti in razlike med področjema. Ugotavljamo, da po eni strani lahko metode MCTS razumemo kot podmnožico metod spodbujevalnega učenja, ki se učijo s pomočjo vzorčenja po Monte Carlu, po drugi strani pa vnašajo na področje spodbujevalnega učenja nove koncepte, ki jih klasično spodbujevalno učenje ne zna opisati.

S pomočjo konceptov spodbujevalnega učenja lahko opišemo številne mehanizme drevesnega preiskovanja Monte Carlo. Tako lahko izpostavimo podobnosti med obema družinama metod:

- soočanje z *zakasnjeno* povratno informacijo;
- iskanje ravnovesja med raziskovanjem prostora ter izkoriščanjem znanja (ang. the *exploration-exploitation dilemma*);
- iterativno ali *epizodično* nabiranje *izkušenj* in vzorcev;
- pomnjenje *obiskanih* stanj ter izbranih akcij, bodisi v obliki preiskovalnega drevesa (privzeto za temeljne metode MCTS) ali grafa na osnovi transpozicij (privzeto za temeljne metode spodbujevalnega učenja);
- vrednotenje obiskanih *stanj* in *akcij* preko mehanizmov vzvratnega prenosa povratne informacije – *učenje* na podlagi *nagrad* iz *okolja*;
- raziskovanje prostora stanj s pomočjo *politik izbire akcij*, ki se tekom učenja (preiskovanja) izboljšujejo na podlagi nabranih izkušenj in usmerjajo raziskovanje v najbolj obetavno smer. To je temeljni mehanizem spodbujevalnega učenja znan kot *posplošena iteracija politike* (ang. generalized policy iteration), s katerim lahko opišemo veliko število algoritmov umetne inteligence. Posplošena iteracija politike dokazano konvergira k optimalni rešitvi, če ima algoritem na razpolago več časa za učenje (ali preiskovanje);
- možnost izstavitve približne rešitve po poljubnem času izvajanja - zaustavitev algoritmov po poljubnem pretečenem času (ravno zaradi tega, ker obe skupini metod delujeta po mehanizmu posplošene iteracije politike); in
- zmožnost učenja iz simulacij – *planiranja* – če je na voljo *simulacijski model* problema (okolja).

Tako lahko originalen algoritem UCT [8] opišemo kot metodo spodbujevalnega učenja, ki uporablja vzorčenje po Monte Carlu in politiko izbire akcij UCB1, in ki ima na razpolago model okolja, s pomočjo katerega lahko simulira interakcijo z okoljem v okviru razpoložljivega računskega časa.

Zgornje ugotovitve kažejo, da lahko spodbujevalno učenje zelo dobro opiše fazi *sestopa* in *vzvratnega prenosa* metod MCTS. Po drugi strani pa ima osnovna teorija spodbujevalnega učenja težave pri opisovanju faz *razširitve* in *odigravanja*. Razlog je v načinu, kako se ena in druga skupina metod spopada s težavo *omejene količine pomnilnika pri reševanju velikih problemov*. Metode spodbujevalnega učenja običajno to rešujejo s *funkcijsko aproksimacijo* prostora stanj – predstavitev v pomnilniku je ves čas učenja fiksna in opisuje celoten prostor stanj, toda z določeno stopnjo nenatančnosti glede na izbrane *značilke*. Po drugi strani pa preiskovalne metode, vključno z MCTS, hranijo v pomnilniku v določenem trenutku le del prostora stanj – tisti del, ki je *najbolj relevanten* glede na izbran kriterij ali *hevristiko*. Na tak način preiskovalne metode spreminjajo predstavitev med postopkom učenja (preiskovanja). Takšno *sprotno spreminjanje* predstavitve (modela prostora stanj, vrednostne funkcije ali politike izbire akcij) je nenavadno za spodbujevalno učenje in ga z osnovnimi koncepti ne more opisati. Nadalje sledi, da spodbujevalno učenje ne pozna *privzete politike* in ne razločuje med delom prostora stanj, ki se nahaja v pomnilniški predstavitvi (ekvivalent zgrajenemu drevesu MCTS), in delom, ki se ne nahaja v pomnilniku (ekvivalent fazi *odigravanja*), za katerega ne beležimo spodbude in ne posodabljamo vrednostne funkcije. Zato osnovno spodbujevalno učenje ne more opisati standardne različice algoritma UCT [5], ki, za razliko od originalne različice [8], ne uporablja transpozicij, gradi drevo namesto usmerjenega grafa in v vsaki iteraciji (epizodi) doda v graf samo eno novo obiskano stanje, namesto vseh obiskanih stanj.

C.6 Drevesno preiskovanje s časovnimi razlikami

S pomočjo dosedanjih ugotovitev predstavljamo novo ogrodje za učenje in preiskovanje: *drevesno preiskovanje s časovnimi razlikami* (ang. temporal difference tree search – TDTS). Slednje smatramo kot posplošitev metod MCTS, saj ohranja vse njihove mehanizme, toda namesto vzorčenja po Monte Carlu uporablja za učenje algoritem TD(λ) [11]. Algoritem TD(λ) pri vrednosti $\lambda = 0$ privede do učenja izključno s časovnimi razlikami, medtem ko $\lambda = 1$ privede do enakega učenja kot metode Monte Carlo – vse vmesne vrednosti pa kombinirajo oba pristopa. Tako so metode MCTS ena izmed robnih variant našega ogrodja TDTS.

Iz drugega zornega kota, TDTS posplošuje algoritme spodbujevalnega učenja z drevesnim preiskovanjem in sprotnim spreminjanjem predstavitve prostora stanj. Preko TDTS vpeljujemo v teorijo spodbujevalnega učenja dva nova koncepta:

- *Predstavitveno politiko*, ki določa, kako se model prostora stanj spreminja. Na primer, pri standardnem algoritmu UCT določa, da se preslikovalna tabela, ki ponazarja drevo ali usmerjen graf, sprotno povečuje za en element na epizodo. Enako predstavitveno politiko uporablja naše ogrodje TDTS. Predstavitveno politiko lahko razumemo kot posplošitev faze *razširitve* metod MCTS.

- *Vrednostno funkcijo v odigravanju*, ki vpeljuje razločevanje med delom preiskovalnega prostora, ki ga algoritem hrani v pomnilniku, in delom, ki ga ne. Funkcija določa vrednosti stanj (in akcij), ki jih predstavitev ne hrani v pomnilniku. Te vrednosti so potrebne, ker metode spodbujevalnega učenja za pravilno delovanje v splošnem potrebujejo vrednosti vseh stanj, te pa v fazi odigravanja niso na voljo. Zato moramo predpostaviti njihovo vrednost na podoben način kot predpostavimo *začetne vrednosti* pri klasičnih metodah spodbujevalnega učenja. Učenje s časovnimi razlikami je primer metode, ki za pravilno posodabljanje vrednostne funkcije potrebuje takšne predpostavke.

Po eni strani lahko TDTS klasificiramo kot metodo *preiskovanja s časovnimi razlikami* [41], toda po drugi strani, TDTS nadgrajuje to isto metodo z novimi koncepti, ki smo jih opisali zgoraj.

C.7 Algoritem Sarsa-UCT

Za potrditev naše ideje smo razvili konkreten primer metode TDTS: *algoritem Sarsa-UCT(λ)* predstavlja prvo učinkovito aplikacijo učenja s časovnimi razlikami na enega izmed temeljnih algoritmov MCTS ob ohranitvi vseh štirih faz MCTS. Izbrali smo algoritma Sarsa(λ) ter UCT, ker sta dva izmed najbolj raziskanih in široko uporabljenih predstavnikov svojih področij. Ključna razlika med Sarsa-UCT(λ) in standardnim algoritmom UCT je v načinu posodabljanja vrednostne funkcije. Sarsa-UCT(λ) ravno tako izbira vozlišča z najvišjo vrednostjo po Enačbi (C.1), vendar namesto vrednosti Q_{MC} uporablja vrednosti $Q_{\text{TD}(\lambda)}$, ki jih računa z uporabo algoritma TD(λ) [11].

Naša metoda ohranja enako računsko in pomnilniško zahtevnost kot originalen algoritem UCT, obenem pa preko štirih novih parametrov (ki izhajajo iz algoritma TD(λ)) omogoča implementacijo širokega spektra obstoječih algoritmov MCTS in številne nove algoritme. Preko teh parametrov lahko algoritmom TDTS določamo hitrost in način pozabljanja znanja, težo predhodnega znanja glede na sprotno pridobljeno znanje, hevristike v fazah sestopa in odigravanja, zanesljivost simulacijskega

modela, zanesljivost dolgoročne povratne informacije, prednost kratkoročnih nagrad, predhodno ali ekspertno znanje, in, najpomembneje, kako močno se bo algoritem posluževal učenja s časovnimi razlikami – parameter λ.

Ker algoritem Sarsa-UCT(λ) tako kot algoritem UCT uporablja drevesno politiko UCB1, za pravilno delovanje potrebuje vrednosti normirane v razponu $[0, 1]$. Če je Sarsa-UCT(λ) nastavljen tako, da popolnoma posnema delovanje algoritma UCT (ko $\lambda = 1$), potem je postopek normiranja enostavnejši – dovolj je nastavitev parametra C_p na ustrezno vrednost za trenutni problem, kot je to praksa pri algoritmu UCT [8]. V nasprotnem primeru (ko $\lambda < 1$) pa je normiranje bolj težavno; zato je politika UCB1 redkeje uporabljena pri algoritmih spodbujevalnega učenja; še pri metodi preiskovanja s časovnimi razlikami [6] so avtorji poročali o nezadovoljivih rezultatih ob njeni uporabi. Za namen normiranja vrednosti pri algoritmu Sarsa-UCT(λ) smo razvili novo tehniko, ki sprotno spreminja zgornjo in spodnjo mejo normiranja za vsako stanje posebej, zato smo jo poimenovali *prostorsko-lokalno normiranje vrednosti* (ang. space-local value normalization). Metoda je neparametrična in se avtomatsko prilagaja nastavljenim parametrom učenja algoritma – apliciramo jo lahko na poljubno metodo spodbujevalnega učenja, tudi izven domen TDTS ali MCTS.

C.8 Eksperimentalna analiza in ugotovitve

Osrednji cilj naše eksperimentalne analize je bil ugotoviti, ali zamenjava učenja z vzorčenjem po Monte Carlu z učenjem s časovnimi razlikami v metodah MCTS izboljša zmogljivost teh metod. V ta namen smo implementirali in primerjali več TDTS algoritmov: od originalnega algoritma UCT, preko standardnega algoritma UCT, do algoritma Sarsa-UCT(λ). Ugotavljali smo, kako vrednost parametra λ, ki uravnava do kolikšne mere se algoritem poslužuje učenja s časovnimi razlikami, vpliva na obnašanje algoritma pri različnih konfiguracijah učnih parametrov in pri različni količini časa na razpolago za učenje. Za testne domene smo izbrali dve enostavni teoretični igri za enega igralca (ki se pogosto pojavljata tudi v drugih raziskavah [9, 36, 120]), štiri klasične igre za dva igralca (križci in krožci, štiri v vrsto, gomoku, in hex) ter nabor arkadnih video iger iz mednarodnega tekmovanja inteligentnih agentov za igranje iger *General Video Game AI competition (GVG-AI)* [15]. Merili smo kakovost vrednostne funkcije, kakovost politike izbire akcij, kakovost planiranja in moč igranja. Slednjo smo izrazili kot delež zmag proti nasprotniku (pri igrah z dvema igralcema) ali kot količino nabrane nagrade (pri igrah z enim igralcem).

Z našimi algoritmi smo se tudi udeležili številnih tekmovanj GVG-AI. V tekmovanjih za enega igralca smo preizkušali osnovni algoritem Sarsa-UCT(λ), ki čim bolj posnema standardni UCT algoritem, razen po uporabi učenja po časovnih razlikah ($\lambda < 1$) – naš algoritem se je uvrstil bolje kot standardni UCT v štirih od petih tekmovanj. V tekmovanjih za dva igralca smo preizkušali različico algoritma Sarsa-UCT(λ), ki ima parametre nastavljene tako, da čim bolj izkorišča potencial TDTS ogrodja – s to različico smo dvakrat (od treh tekmovanj) dosegli prvo mesto, kar izpostavljamo kot najboljši dosežek naših algoritmov.

Ugotavljamo, da novi algoritmi TDTS ohranjajo robustnost in enako računsko ter pomnilniško zahtevnost, obenem pa dosegajo boljše rezultate od tradicionalnih algoritmov MCTS v večini testnih domen in v večini uporabljenih konfiguracij. V splošnem je doprinos večji, ko je na razpolago manj računskega časa; z večanjem računskega časa pa postajajo novi algoritmi bolj enakovredni algoritmom MCTS – to je bolj opazno na igrah z dvema igralcema in manj na igrah z enim igralcem. Doprinos je tudi večji, ko algoritmi ne uporabljajo transpozicij in gradijo drevo (to je privzeta nastavitev metod MCTS).

Uporaba učenja s časovnimi razlikami nekoliko poveča občutljivost algoritmov na začetne vrednosti, predvsem pa naredi algoritme zelo občutljive na vrednost parametra λ: optimalna vrednost tega parametra je odvisna od konfiguracije in danega problema, pri čemer lahko prenizka vrednost povzroči slabše delovanje od čistega učenja z vzorčenjem po Monte Carlu (ko $\lambda = 1$) – to smatramo kot glavno slabost algoritmov TDTS. Ob upoštevanju tega je konservativna nastavitev parametra λ bliže vrednosti 1 bolj varna, vendar tudi ustvari manj doprinosa v zmogljivosti algoritmov. Kako teoretično določiti optimalno vrednost parametra λ, je že dve desetletji eden od glavnih odprtih izzivov spodbujevalnega učenja in je izven zmožnosti te disertacije, da ga razreši. Kljub temu pa opažamo, da lahko z eksperimentiranjem najdemo zadovoljivo vrednost λ, pri čemer je to lažje, če se razpoložljiv čas za učenje ne spreminja. V nasprotnem primeru bi potrebovali dodatne mehanizme za sprotno spreminjanje vrednosti λ, kar pa sodi v napredne tehnike spodbujevalnega učenja [42].

Kljub temu, da smo preizkusili veliko število konfiguracij parametrov algoritmov TDTS, smo v naših eksperimentih merili le različne načine učenja algoritmov, ostale mehanizme (ki smo jih omenili v prejšnjem poglavju) pa smo izpustili. Izmed teh mehanizmov bi v prihodnje bili zanimivi za raziskovanje še predvsem pozabljanje znanja in pripisovanje večje teže bližnjim nagradam. Oba mehanizma imata velik potencial,

da še dodatno izboljšata zmogljivost algoritmov TDTS, saj sta se že v preteklosti izkazala kot dobra tako v domeni spodbujevalnega učenja, kot v domeni MCTS. Poleg tega bi bilo smiselno preveriti algoritme TDTS tudi na drugih tipih iger, na primer, na večigralskih igrah, s sočasnim izvajanjem potez, z nepopolno informacijo, z zašumljenim generacijskim modelom, itd.

Naše ugotovitve nakazujejo, da je še veliko neraziskanega potenciala v metodah in idejah, ki smo jih predstavili v tej disertaciji. V grobem lahko te obetavne raziskovalne smeri razdelimo na dve skupini: (1) izboljšanje sodelovanja in prehoda idej med področjema preiskovanja in spodbujevalnega učenja ter (2) temeljita teoretična analiza, kako predstavitvene politike v splošnem vplivajo na metode spodbujevalnega učenja.

C.9 Zaključek

Glavni cilj te disertacije je bilo izboljšanje temeljnih algoritmov drevesnega preiskovanja Monte Carlo (MCTS) z uporabo domensko-neodvisnih pristopov, ki bi ohranili splošnost novih algoritmov. To smo uspeli doseči z integracijo dobro raziskanih in uveljavljenih metod spodbujevalnega učenja v ogrodje MCTS. S tem smo potrdili, da kombiniranje metod iz teh dveh področij odpira možnosti za razvoj številnih novih algoritmov, ki po zmogljivostih prekašajo klasične algoritme MCTS. S posnemanjem pristopa, ki smo ga uporabili pri naših raziskavah, bi lahko na podoben način nadgradili poljubne algoritme MCTS z močnimi metodami učenja – ali obratno, poljubne metode spodbujevalnega učenja bi lahko nadgradili z drevesnim preiskovanjem in adaptivnimi predstavitvami prostora stanj. S tem zmanjšujemo razkorak med raziskovalnimi skupnostmi teh dveh področij in jih spodbujamo k tesnejšemu sodelovanju, ki bi lahko prineslo razvoj potencialno nove družine algoritmov umetne inteligence.

Čeprav smo izpolnili naš glavni cilj, smo pri naših raziskavah šli še dlje: pokazali smo, da se metode spodbujevalnega učenja in metode preiskovanja v veliki meri prekrivajo, saj se ukvarjajo z zelo podobnimi problemi, le da do njih pristopajo iz drugačnih zornih kotov. Zato predlagamo, da se uvrščanje preiskovalnih metod v področje spodbujevalnega učenja ne razume kot nova ali nasprotujoča razlaga že uveljavljenim razlagam raziskovalne skupnosti, ampak bolj kot komplementarna razlaga, iz katere lahko črpamo veliko znanja in idej za v prihodnje. Dosežki, ki smo jih predstavili v tej disertaciji, izhajajo iz takšnega razmišljanja in ga še naprej spodbujajo. To smatramo kot korak v smeri enotnega pogleda na učenje, planiranje in preiskovanje.

BIBLIOGRAPHY

[1] Murray Campbell, A. Joseph Jr. Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, jan 2002. ISSN 00043702. doi: 10.1016/S0004-3702(01)00129-1. URL http://www.sciencedirect.com/science/article/pii/S0004370201001291.

[2] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012. ISSN 0001-0782. doi: 10.1145/2093548.2093574. URL http://doi.acm.org/10.1145/2093548.2093574.

[3] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006. URL http://hal.inria.fr/inria-00117266.

[4] Rémi Coulom. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*, pages 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-75538-8. doi: 10.1007/978-3-540-75538-8_7. URL http://dx.doi.org/10.1007/978-3-540-75538-8_7.

[5] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, mar 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6145622.

[6] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961. URL http://dx.doi.org/10.1038/nature16961.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015. ISSN 0028-0836. doi: 10.1038/nature14539. URL http://www.nature.com/doifinder/10.1038/nature14539.

[8] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the Seventeenth European Conference on Machine Learning*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293, Berlin/Heidelberg, Germany, 2006. Springer. ISBN 3-540-45375-X. URL http://www.sztaki.hu/~szcsaba/papers/ecml06.pdf.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998. ISBN 0262193981.

[10] David Silver. *Reinforcement Learning and Simulation-Based Search in Computer Go*. Ph.d. thesis, University of Alberta, Edmonton, Alta., Canada, 2009. URL http://dl.acm.org/citation.cfm?id=1834781.

[11] Richard S. Sutton. Learning to predict by the methods of temporal differences. In *Machine Learning*, pages 9–44. Kluwer Academic Publishers, 1988.

[12] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[13] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002. ISSN 0885-6125. doi: 10.1023/A:1013689704352. URL http://dx.doi.org/10.1023/A:1013689704352.

[14] Gavin A. Rummery and Mahesan Niranjan. On-Line {Q}-Learning Using Connectionist Systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, England, 1994. URL http://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/rummery_tr166.pdf.

[15] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon Lucas, Adrien Couetoux, Jerry Lee, Chong-u Lim, and Tommy Thompson. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 2015. ISSN 1943-068X. doi: 10.1109/TCIAIG.2015.2402393. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7038214.

[16] Joseph A. M. Nijssen. *Playing Othello Using Monte Carlo*. B.sc. thesis, Maastricht, 2007.

[17] Julien Kloetzer. Monte-Carlo Opening Books for Amazons. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6515 LNCS, pages 124–135. 2011. ISBN 3642179274. doi: 10.1007/978-3-642-17928-0_12. URL http://link.springer.com/10.1007/978-3-642-17928-0_12.

[18] Tomáš Kozelek. *Methods of MCTS and the game Arimaa*. Master's thesis, Charles University in Prague, 2009. URL http://kozelek.cz/akimot/mt.pdf.

[19] Tom Pepels, Mark H. M. Winands, and Marc Lanctot. Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, sep 2014. ISSN 1943-068X. doi: 10.1109/TCIAIG.2013.2291577. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6731713.

[20] Diego Perez, Edward Powley, Daniel Whitehouse, Spyridon Samothrakis, Simon Lucas, and Peter I. Cowling. The 2013 Multi-objective Physical Travelling Salesman Problem Competition. *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2314–2321, jul 2014. doi: 10.1109/CEC.2014.6900243. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6900243.

[21] Dennis Soemers. *Tactical Planning Using MCTS in the Game of StarCraft*. B.sc. thesis, Maastricht, 2014.

[22] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005. ISSN 0738-4602. doi: 10.1609/aimag.v26i2.1813. URL https://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1813.

[23] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, AAAI'08, pages 259–264. AAAI Press, 2008. ISBN 978-1-57735-368-3. URL http://dl.acm.org/citation.cfm?id=1619995.1620038.

[24] Hilmar Finnsson and Yngvi Björnsson. CADIA PLAYER : A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[25] Jean Mehat and Tristan Cazenave. Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, dec 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2088123. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5604665.

[26] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, dec 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2067212. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5551182.

[27] Arthur L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. II: Recent Progress. *IBM J. Res. Dev.*, 11(6):601–617, nov 1967. ISSN 0018-8646. doi: 10.1147/rd.116.0601. URL http://dx.doi.org/10.1147/rd.116.0601.

[28] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 273–280, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-793-3. doi: 10.1145/1273496.1273531. URL http://doi.acm.org/10.1145/1273496.1273531.

[29] Guillaume M. J-B. Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, and Olivier Teytaud. *Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-12993-3. doi: 10.1007/978-3-642-12993-3_1. URL http://dx.doi.org/10.1007/978-3-642-12993-3_1.

[30] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010. ISSN 1943-068X. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5523941.

[31] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis. Transpositions and move groups in Monte Carlo tree search. *2008 IEEE Symposium On Computational Intelligence and Games*, pages 389–395, dec 2008. doi: 10.1109/CIG.2008.5035667. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5035667.

[32] Donald A. Berry and Bert Fristedt. *Bandit problems: Sequential Allocation of Experiments*. Springer Netherlands, 1985. ISBN 978-94-015-3713-1. doi: 10.1007/978-94-015-3711-7. URL http://link.springer.com/10.1007/978-94-015-3711-7.

[33] Rajeev Agrawal. Sample mean based index policies with O (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995. URL http://www.jstor.org/stable/10.2307/1427934.

[34] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 22:4–22, 1985.

[35] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. Technical Report 1, University of Tartu, Estonia, 2006. URL http://www.ualberta.ca/~szepesva/papers/cg06-ext.pdf.

[36] Abdallah Saffidine, Tristan Cazenave, and Jean Mehat. UCD: Upper Confidence Bound for Rooted Directed Acyclic Graphs. In *International Conference on Technologies and Applications of Artificial Intelligence*, pages 467–473. Ieee, nov 2010. ISBN 978-1-4244-8668-7. doi: 10.1109/TAAI.2010.79. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5695494.

[37] Richard J. Lorentz. Amazons Discover Monte-Carlo. In H.Jaap Herik, Xinhe Xu, Zongmin Ma, and MarkH.M. Winands, editors, *Proceedings of the 6th international conference on Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 13–24. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87607-6.

[38] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for Monte-Carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006. URL https://hal.inria.fr/hal-00115330/document.

[39] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175 (11):1856–1875, jul 2011. ISSN 0004-3702. doi: 10.1016/j.artint.2011.03.007. URL http://dx.doi.org/10.1016/j.artint.2011.03.007.

[40] John Asmuth and Michael L. Littman. Learning is planning: near Bayes-optimal reinforcement learning via Monte-Carlo tree search. *The Computing Research Repository (CoRR)*, abs/1202.3, feb 2012. URL http://arxiv.org/abs/1202.3699.

[41] David Silver, Richard S. Sutton, and Martin Müller. Temporal-difference search in computer Go. *Machine Learning*, 87(2):183–219, feb 2012. ISSN 0885-6125. doi: 10.1007/s10994-012-5280-0. URL http://dx.doi.org/10.1007/s10994-012-5280-0.

[42] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Unpublished work in progress, second edition, 2017.

[43] Hendrik Baier. *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*. Ph.d. thesis, Maastricht University, 2015.

[44] Richard Bellman. A Markovian decision process, 1957. ISSN 01650114.

[45] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[46] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998. ISSN 00043702. doi: 10.1016/S0004-3702(98)00023-X.

[47] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27644-6. doi: 10.1007/978-3-642-27645-3. URL http://link.springer.com/10.1007/978-3-642-27645-3.

[48] D. Michie and R. A. Chambers. Boxes: An Experiment in Adaptive Control. *Machine Intelligence 2*, pages 137–152, 1968.

[49] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. Ph.d. thesis, Cambridge University, England, 1989. URL http://www.cs.rhul.ac.uk/~chrisw/thesis.html.

[50] John N. Tsitsiklis. On the convergence of optimistic policy iteration. *JMLR*, 3:59–72, 2002. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944922.

[51] Wei Chu, Lihong Li, Lev Reyzin, and Robert E Schapire. Contextual bandits with linear payoff functions. In *International Conference on Artificial Intelligence and Statistics*, pages 208–214, 2011.

[52] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems*, pages 4026–4034, 2016.

[53] B. Van Roy, D. P. Bertsekas, Y. Lee, and J. N. Tsitsiklis. A Neuro-Dynamic Programming Approach to Retailer Inventory Management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, volume 4, pages 4052–4057. IEEE, 1997. ISBN 0-7803-4187-2. doi: 10.1109/CDC.1997.652501. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=652501.

[54] Satinder P. Singh and Richard S. Sutton. Reinforcement Learning with Replacing Elibibility Traces. *Machine Learning*, 22:123–158, 1996.

[55] Zohar Feldman and Carmel Domshlak. Simple Regret Optimization in Online Planning for Markov Decision Processes. *Journal of Artificial Intelligence Research*, pages 165–205, 2014.

[56] Dennis M. Breuker, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Replacement schemes for transposition tables. *ICCA Journal*, 17(4):183–193, 1994. ISSN 0920-234X.

[57] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive Strategies For Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.3015.

[58] Rémi Coulom. Computing "Elo Ratings" of Move Patterns in the Game of Go. *Journal of The International Computer Games Association*, 30(4):198–208, 2007.

[59] Paul E. Utgoff. Incremental Induction of Decision Trees. *Machine Learning*, 4(2):161–186, 1989. ISSN 08856125. doi: 10.1023/A:1022699900025. URL http://link.springer.com/10.1023/A:1022699900025.

[60] Dimitri. P. Bertsekas and David.A. Castañón. Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34(6):589–598, jun 1989. ISSN 00189286. doi: 10.1109/9.24227. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=24227.

[61] Alborz Geramifard, Joshua Redding, Jonathan P. How, Finale Doshi, and Nicholas Roy. Online Discovery of Feature Dependencies. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 881–888, 2011.

[62] Philipp W. Keller, Shie Mannor, and Doina Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. *Proceedings of the 23rd international conference on Machine learning - ICML '06*, pages 449–456, 2006. doi: 10.1145/1143844.1143901. URL http://dl.acm.org/citation.cfm?id=1143844.1143901.

[63] Richard S Sutton, Steven D Whitehead, et al. Online learning with random representations. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 314–321, 1993.

[64] Martin Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. In *ECML*, volume 3720, pages 317–328. Springer, 2005.

[65] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

[66] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[67] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[68] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven Exploration by Self-supervised Prediction. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/pathak17a.html.

[69] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[70] Kenneth O Stanley and Risto Miikkulainen. Efficient evolution of neural network topologies. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1757–1762. IEEE, 2002.

[71] Richard S. Sutton. Integrated Architectures for Learning , Planning , and Reacting Based on Approximating Dynamic Programming. In *ICML*, pages 216–224, 1990.

[72] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David P Reichert, Neil Rabinowitz, Andre André Barreto, and Thomas Degris. The Predictron: End-To-End Learning and Planning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning (ICML) 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3191–3199. PMLR, 2017. URL http://proceedings.mlr.press/v70/silver17a.html.

[73] Ishai Menache, Shie Mannor, and Nahum Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238, 2005.

[74] Huizhen Yu and Dimitri P Bertsekas. Basis function adaptation methods for cost approximation in mdp. In *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL'09. IEEE Symposium on*, pages 74–81. IEEE, 2009.

[75] Ashique Rupam Mahmood and Richard S Sutton. Representation search through generate and test. In *AAAI Workshop: Learning Rich Representations from Low-Level Sensors*, 2013.

[76] Shimon Whiteson, Matthew E Taylor, Peter Stone, et al. *Adaptive tile coding for value function approximation*. Computer Science Department, University of Texas at Austin, 2007.

[77] Bohdana Ratitch and Doina Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *European Conference on Machine Learning*, pages 347–358. Springer, 2004.

[78] Lihong Li, Michael Littman, and Thomas J Walsh. Knows What It Knows: A Framework For Self-Aware Learning. *Proceedings of the 25th International Conference on Machine Learning*, pages 568–575, 2008.

[79] Thomas J. Walsh, Sergiu Goschin, and Michael L. Littman. Integrating Sample-based Planning and Model-based Reinforcement Learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[80] István Szita and Csaba Szepesvári. Agnostic KWIK learning and efficient approximate reinforcement learning. *Journal of Machine Learning Research*, 19: 739–772, 2011. ISSN 15337928.

[81] Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. *Proc. 21st Int. Conf. Automat. Plan. Sched., …*, pages 202–209, 2011.

[82] Marc Lanctot, Mark H M Winands, Tom Pepels, and Nathan R. Sturtevant. Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. *IEEE Conference on Computational Intelligence and Games, CIG*, jun 2014. ISSN 23254289. doi: 10.1109/CIG.2014.6932903. URL http://arxiv.org/abs/1406.0486.

[83] Hendrik Baier and Mark H. M. Winands. Monte-Carlo Tree Search and minimax hybrids. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, number c, pages 1–8. IEEE, aug 2013. ISBN 978-1-4673-5311-3. doi: 10.1109/CIG.2013.6633630. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6942254http://ieeexplore.ieee.org/document/6633630/.

[84] Peter I. Cowling, Colin D. Ward, and Edward J. Powley. Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):241–257, dec 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2204883. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6218176.

[85] Fan Xie and Zhiqing Liu. Backpropagation Modification in Monte-Carlo Game Tree Search. In *Proceedings of the Third International Symposium on Intelligent Information Technology Application - Volume 02*, IITA '09, pages 125–128, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3859-4. doi: 10.1109/IITA.2009.331. URL http://dx.doi.org/10.1109/IITA.2009.331.

[86] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989. ISSN 01628828. doi: 10.1109/34.42858. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=42858.

[87] Joseph A. M. Nijssen and Mark H. M. Winands. *Enhancements for Multi-Player Monte-Carlo Tree Search*, pages 238–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-17928-0. doi: 10.1007/978-3-642-17928-0_22. URL http://link.springer.com/10.1007/978-3-642-17928-0_22.

[88] Peter Drake. The Last-Good-Reply Policy for Monte-Carlo Go. *International Computer Games Association Journal*, 32(4):221–227, 2009.

[89] Junichi Hashimoto, Akihiro Kishimoto, Kazuki Yoshizoe, and Kokolo Ikeda. *Accelerated UCT and Its Application to Two-Player Games*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

ISBN 978-3-642-31866-5. doi: 10.1007/978-3-642-31866-5_1. URL http://link.springer.com/10.1007/978-3-642-31866-5_1.

[90] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego – An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, dec 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2083662. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5599855.

[91] Hendrik Baier and Peter D. Drake. The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303–309, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5672398.

[92] Mandy J. W. Tak, Mark H. M. Winands, and Yngvi Bjornsson. Decaying Simulation Strategies. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):395–406, dec 2014. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2310782. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6763042.

[93] Mandy J. W. Tak, Mark H. M. Winands, and Yngvi Björnsson. N-grams and the last-good-reply policy applied in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012. ISSN 1943068X. doi: 10.1109/TCIAIG.2012.2200252.

[94] Jan A. Stankiewicz. *Knowledge-Based Monte-Carlo Tree Search in Havannah*. Master's thesis, Maastricht University, The Netherlands, 2011.

[95] Guillaume M. J-B. Chaslot, Jean-Baptiste Hoock, Fabien Teytaud, and Olivier Teytaud. On the huge benefit of quasi-random mutations for multimodal optimization with application to grid-based tuning of neurocontrollers. *ESANN*, (April):22–24, 2009. URL http://hal.inria.fr/inria-00380125/.

[96] Joseph A. M. Nijssen. *Monte-Carlo Tree Search for Multi-Player Games*. 2013. ISBN 9789461694522.

[97] David Silver, Richard Sutton, and Martin Müller. Reinforcement Learning of Local Shape in the Game of Go. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 1053–1058, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1625275.1625446.

[98] David Silver, Richard S. Sutton, and Martin Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 968–975, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390278. URL http://dl.acm.org/citation.cfm?id=1390278.

[99] Arpad Rimmel and Fabien Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6024 LNCS, pages 201–210. 2010. ISBN 3642122388.

[100] Thomas Keller and Malte Helmert. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Twenty-Third International Conference on Automated Planning and Scheduling*, pages 135–143, 2013.

[101] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, pages 0–66, 1995. ISSN 0004-3702. URL http://www.sciencedirect.com/science/article/pii/0004370294000110.

[102] Zohar Feldman and Carmel Domshlak. On MABs and Separation of Concerns in Monte-Carlo Planning for MDPs. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 120–127, 2014.

[103] Zohar Feldman and Carmel Domshlak. Monte-Carlo Tree Search : To MC or to DP ? In *ECAI14*, 2014.

[104] Todd Hester and Peter Stone. TEXPLORE: Real-time sample-efficient reinforcement learning for robots. *Machine Learning*, 90(3):385–429, 2013. ISSN 08856125. doi: 10.1007/s10994-012-5322-7.

[105] Todd Hester and Peter Stone. Real Time Targeted Exploration in Large Domains. In *The Ninth International Conference on Development and Learning (ICDL)*, 2010.

[106] Piyush Khandelwal, Elad Liebman, Scott Niekum, and Peter Stone. On the Analysis of Complex Backup Strategies in Monte Carlo Tree Search. *International Conference on Machine Learning (ICML)*, 48, 2016.

[107] Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A Monte-Carlo AIXI approximation. *Journal of Artificial Intelligence Research*, 40(1):95–142, 2011. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=2016945.2016949.

[108] Michael Kearns, Y Mansour, and Ay Ng. A Sparse Sampling Algorithm for Near-Optimal. *Machine learning*, pages 193–208, 2002.

[109] David Silver and Joel Veness. Monte-Carlo Planning in Large POMDPs. *Advances in neural information processing systems (NIPS)*, pages 1–9, 2010. URL http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps/.

[110] Arthur Guez, David Silver, and Peter Dayan. Scalable and efficient bayes-adaptive reinforcement learning based on Monte-Carlo tree search. *Journal of Artificial Intelligence Research*, 48:841–883, 2013. ISSN 10769757. doi: 10.1613/jair.4117.

[111] Weijia Wang and Michèle Sebag. Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search. *Machine Learning*, 92(2-3):403–429, sep 2013. ISSN 0885-6125. doi: 10.1007/s10994-013-5369-0. URL http://link.springer.com/10.1007/s10994-013-5369-0.

[112] Zoltán Gábor, Zsolt Kalmár, and Csaba Szepesvári. Multi-criteria Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, volume 98, pages 197–205, 1998. ISBN 1-55860-556-8.

[113] Mayank Daswani, Peter Sunehag, and Marcus Hutter. Feature Reinforcement Learning : State of the Art. In *AAAI-14 Workshop*, pages 2–5, 2014.

[114] Hilmar Finnsson and Yngvi Björnsson. Learning Simulation Control in General Game-Playing Agents. *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, (September):954–959, 2010. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/download/1892/2124.

[115] Ercüment Ilhan and A. Şima Etaner-Uyar. Monte Carlo Tree Search with Temporal-Difference Learning for General Video Game Playing. In *IEEE Conference on Computational Intelligence and Games*, New York, USA, 2017.

[116] Harm Van Seijen, A. Rupam Mahmood, Patrick M Pilarski, Marlos C. Machado, and Richard S Sutton. True Online Temporal-Difference Learning. *Journal of Machine Learning Research*, 17 (September), dec 2016. ISSN 15337928. URL http://arxiv.org/abs/1512.04087.

[117] David Robles, Philipp Rohlfshagen, and Simon M. Lucas. Learning non-random moves for playing Othello: Improving Monte Carlo Tree Search. *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, (September 2016):305–312, 2011. doi: 10.1109/CIG.2011.6032021.

[118] Yasuhiro Osaki, Kazutomo Shibahara, Yasuhiro Tajima, and Yoshiyuki Kotani. An Othello evaluation function based on Temporal Difference Learning using probability of winning. *2008 IEEE Symposium On Computational Intelligence and Games*, pages 205–211, dec 2008. doi: 10.1109/CIG.2008.5035641. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5035641.

[119] Joel Veness, David Silver, William Uther, and Alan Blair. Bootstrapping from game tree search. *Proceedings of Advances in Neural Information Processing Systems*, pages 1–9, 2009.

[120] Tristan Cazenave. Nested Monte-Carlo Search. In *International Joint Conference on Artificial Intelligence*, pages 456–461, 2009.

[121] Zohar Feldman and Carmel Domshlak. Online Planning in MDPs Rationality and Optimization. 2012.

[122] Marc Lanctot, Abdallah Saffidine, Joel Veness, Christopher Archibald, and Mark H. M. Winands. Monte Carlo *-Minimax Search. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), Beijing, China, August 3-9, 2013*, pages 580–586, 2013. URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6862.

[123] Joel Veness, Marc Lanctot, and Michael Bowling. Variance reduction in monte-carlo tree search. *Proceedings of Advances in Neural Information Processing Systems*, pages 1–9, 2011.

[124] Gerald Tesauro. TD-Gammon, a Self-teaching Backgammon Program, Achieves Master-level Play. *Neural Comput.*, 6(2):215–219, 1994. ISSN 0899-7667. doi: 10.1162/neco.1994.6.2.215. URL http://dx.doi.org/10.1162/neco.1994.6.2.215.

[125] Kumpati S. Narendra and Mandayam A. L. Thathachar. *Learning automata - an introduction*. Prentice Hall, 1989. ISBN 978-0-13-527011-0.

[126] Diego Perez-liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General Video Game AI: Competition, Challenges and Opportunities. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[127] Dennis J. N. J. Soemers. *Enhancements for Real-Time Monte-Carlo Tree Search in General Video Game Playing*. Master's thesis, Maastricht University, The Netherlands, 2016.

[128] Tom Vodopivec and Branko Šter. Forgetting Early Estimates in Monte Carlo Control Methods. *Journal of Electrical Engineering and Computer Science*, 82(3): 85–92, 2015.

[129] Matej Guid and Ivan Bratko. Factors affecting diminishing returns for searching deeper. *ICGA Journal*, 30: 75–84, 06 2007.

[130] Matthieu Geist and Bruno Scherrer. Off-policy learning with eligibility traces: a survey. *The Journal of Machine Learning Research*, 15:289–333, 2014. URL http://dl.acm.org/citation.cfm?id=2627445.

[131] William R. Thompson. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25(3/4):285, dec 1933. ISSN 00063444. doi: 10.2307/2332286. URL http://www.jstor.org/stable/2332286?origin=crossref.

[132] Dean Eckles and Maurits Kaptein. Thompson sampling with the online bootstrap. *ArXiv e-prints*, page 13, 2014. URL http://arxiv.org/abs/1410.4009.

[133] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN. *ArXiv e-prints*, feb 2016. URL http://arxiv.org/abs/1602.04621.

INDEX