Journal of Artificial Intelligence Research 60 (2017) 881-936 Submitted 03/17; published 12/17

On Monte Carlo Tree Search and Reinforcement Learning

Tom Vodopivec TOM.VODOPIVEC@FRI.UNI-LJ.SI
Faculty of Computer and Information Science
University of Ljubljana
Večna pot 113, Ljubljana, Slovenia

Spyridon Samothrakis SSAMOT@ESSEX.AC.UK
Institute of Data Science and Analytics
University of Essex
Wivenhoe Park, Colchester CO4 3SQ, Essex, U.K.

Branko Šter BRANKO.STER@FRI.UNI-LJ.SI
Faculty of Computer and Information Science
University of Ljubljana
Večna pot 113, Ljubljana, Slovenia

Abstract

Fuelled by successes in Computer Go, Monte Carlo tree search (MCTS) has achieved widespread adoption within the games community. Its links to traditional reinforcement learning (RL) methods have been outlined in the past; however, the use of RL techniques within tree search has not been thoroughly studied yet. In this paper we re-examine in depth this close relation between the two fields; our goal is to improve the cross-awareness between the two communities. 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 confirm that planning methods inspired by RL in conjunction with online search demonstrate encouraging results on several classic board games and in arcade video game competitions, where our algorithm recently ranked first. Our study promotes a unified view of learning, planning, and search.

1. Introduction

Monte Carlo tree search (MCTS, Coulom, 2006; Browne et al., 2012) was introduced as a search and planning framework for finding optimal decisions by sampling a given model. One of its first incarnations, upper confidence bounds for trees (UCT, Kocsis & Szepesvári, 2006), initiated almost a revolution in game-playing agents. Provided an agent has access to an internal mental simulator, considerable improvement in performance could be achieved compared to more traditional search methods, especially in the game of Go (Gelly & Wang, 2006; Silver et al., 2016). Following a showcase of strong results, the algorithm achieved widespread adoption across diverse domains (Browne et al., 2012).

Although MCTS was developed as a standalone algorithm, its connection with reinforcement learning (RL) was suggested shortly after (Silver, 2009). This link, however, was not easily apparent and has not been widely adopted in the game artificial intelligence (AI) community. The relationship between the two fields remained somewhat opaque, with most applied researchers being unaware

of the connections between MCTS and RL methods. One of the main factors for this lack of cross-fertilization is the absence of a common language between dedicated game researchers and RL researchers. Whereas in our previous work we marginally explored the benefits of enhancing MCTS with RL concepts (Vodopivec & Šter, 2014), here we take a much wider view and rather address the general issues described above. With two main contributions we try to narrow the gap between the MCTS and RL communities, while emphasizing and building upon the studies of Silver (2009).

As our first contribution, we outline the reasons for the separate evolution of the two communities and comprehensively analyse the connection between the fields of MCTS and RL. Focusing on the game AI community by starting from the MCTS point-of-view, we identify both the similarities and the differences between the two fields – we thoroughly describe the MCTS mechanics with RL theory and analyse which of them are unusual for traditional RL methods and can thus be understood as novel. We also survey the numerous MCTS enhancements that (intentionally or unintentionally) integrate RL mechanics and explain them with RL terminology. Reinforcement-learning experts might regard this first part of our study as an extended background, which might also help them appreciate some of the qualities of MCTS.

As our second contribution, to demonstrate how RL mechanics can be beneficial also in MCTS-like algorithms, we develop the *temporal-difference tree search (TDTS)* framework, which can be understood both as a generalization of MCTS with temporal-difference (TD) learning, as well as an extension of traditional TD learning methods with novel MCTS concepts. We showcase a TDTS algorithm, named *Sarsa-UCT*(λ), where we (1) successfully apply a bootstrapping learning method to the original MCTS framework when using an incrementally-growing tree structure with a tabular, non-approximated representation, and without domain-specific features; (2) efficiently combine it with an upper confidence bounds (UCB) selection policy (Auer, Cesa-Bianchi, & Fischer, 2002); and (3) evaluate and analyse TD backups under such conditions. The new algorithm extends UCT with the mechanism of eligibility traces and increases its performance when the new parameters are tuned to an informed value. Furthermore, it has recently achieved strong results in the *general video game AI (GVG-AI) competition* (Perez et al., 2015), including two first positions in two-player competitions in 2016 and 2017.

2. Background

First we describe Markov decision processes – a framework for modelling decision-making problems that both Monte Carlo tree search and reinforcement learning methods can solve. Then we outline the basics of the two fields in question.

2.1 Markov Decision Processes

Decision problems (or *tasks*) are often modelled using *Markov decision processes (MDPs)*. An MDP is composed of the following:

- *States* $s \in \mathcal{S}$, where s is a state in general and $S_t \in \mathcal{S}$ is the (particular) state at time t.
- *Actions* $a \in \mathcal{A}$, 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* (e.g., endgame 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 moving 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 time step, 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 can be a simple mapping from the current state to actions, or can also be more complex, e.g., a search process. Solving an MDP requires finding a policy that maximizes the *cumulative discounted reward*, which is also known as the *return*

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

The (state) *value function* $v_\pi(s)$ is the expected return when the agent follows policy π from state s, while the state-action value function $q_\pi(s,a)$ is the expected return when the agent performs action a in state s and then follows the policy π. There are clear links between MDPs and bandit problems: the latter can be seen as "prototypical MDPs if formulated as multiple-situation problems" (Sutton & Barto, 2017).

MDP tasks may be *episodic* or *continuing*. Episodic tasks consist of sequences of limited length, referred to as *episodes*. We will denote the final time step of a single episode as T, but knowing that T is specific for each particular episode (i.e., episodes might have different lengths). 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

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T \ , \tag{2}$$

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 paper 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.

2.2 Reinforcement Learning

Reinforcement learning (RL) is an established paradigm for agents learning from experience (Sutton & Barto, 1998, 2017). In an RL problem, an *agent* performs *actions* in an *environment* and receives feedback in form of *rewards* and *observations* of the current *state* of the environment. The goal of the agent is to maximize the expected return. It tries to remember or *learn* which actions in

which states led to higher rewards in order to use this knowledge to take better actions in the future. Although there are several ways of defining the RL problem (e.g., with operant conditioning in psychology), the MDP formalism is more common in AI research.

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 exploring the state space in a trial-and-error fashion; and it must be able to learn from (its own) *experience* (i.e., samples of interaction with the environment). The problem is harder when the rewards are delayed in time. Another major challenge is the *exploration-exploitation dilemma*, where 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. This dilemma is central also to MDPs and bandit problems.

In general, RL problems can be solved by learning a value function and then using it to design a good policy. 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 (2012).

2.3 Temporal-Difference Learning

Temporal-difference (TD) learning (Sutton, 1988) 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. It decreases the variance of evaluations and increases the bias, but usually increases the learning performance (Sutton & Barto, 1998). The term *temporal-differences* stems from the fact that the *previous* estimate of the state value in the next time step affects the *current* estimate of the state value in the current time step.

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

$$V(S_t) \leftarrow V(S_t) + \alpha_n \big[G_t - V(S_t) \big] , \quad \forall t < T , \tag{3}$$

where the *update step-size* $\alpha_n \in [0, 1]$ decreases with the number of visits to the node n, often as $\alpha_n = 1/n$. The first episode starts with the *initial state values* $V_{\text{init}}(s)$ and with $n = 1$. 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:

$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \big[(R_{t+1} + \gamma V_t(S_{t+1})) - V_t(S_t) \big] . \tag{4}$$

Updating estimates based on other estimates is is known as *bootstrapping*. 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:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) . \tag{5}$$

Between the targets G_t and $R_{t+1} + \gamma V_t(S_{t+1})$ it is possible to define intermediate targets, called n-step returns (see Appendix A), and also to combine them in different ways, one of them being the λ-return (which can be expressed in various ways, see for example Fairbank and Alonso (2012)).

When approaching the same problem from a different perspective, we can say that it is not necessary that a single TD error δ_t updates only the last visited state value V_t, as in (4), 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 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, which are updated for all states at each time step. Two popular variants of the update mechanics are the *replacing* and the *accumulating* eligibility traces, where

$$E_t(s) = \begin{cases} 1 & \text{if } s = S_t \text{ (replacing)}, \\ \gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t \text{ (accumulating)}, \\ \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t. \end{cases} \tag{6}$$

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

$$V_{t+1}(s) = V_t(s) + \alpha_t \delta_t E_t(s)\,, \quad \forall s \in \mathcal{S}\,, \tag{7}$$

or offline at the end of each episode as

$$V(s) \leftarrow V(s) + \alpha_n \sum_{t=0}^{T} \delta_t E_t(s)\,, \quad \forall s \in \mathcal{S}\,. \tag{8}$$

Merging TD backups with eligibility traces results in arguably the most popular policy-evaluation method, TD(λ) (Sutton, 1988). When $\lambda = 0$, the algorithm updates only a single previous state, whereas when $\lambda = 1$, it updates all states, and with the same TD error – under offline updating it produces exactly the same backups as Monte Carlo methods (Sutton & Barto, 1998). 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 (Section 3.2). In this way eligibility traces augment MC methods with bootstrapping. Appendix A gives the details of the TD(λ) algorithm.

2.4 Monte Carlo Tree Search and the UCT Algorithm

In general, Monte Carlo tree search (MCTS) algorithms employ strong (heuristic) *search methods* to identify the best available action in a given situation (Browne et al., 2012). They gain knowledge by *simulating* the given problem and thus require at least its *generative model* (also known as a *forward*, *simulation*, or *sample* model), which may be simpler than the complete transition model of a given task. The MCTS planning process incrementally builds an *asymmetric search tree* that is guided in the most promising direction by an exploratory action-selection policy, which is computed iteratively. It improves when increasing the number of iterations. An *MCTS iteration* usually consists of four consecutive phases: (1) *selection* of actions already memorized in the tree (descent from the root to a leaf node), (2) *expansion* of the tree with new nodes, (3) *playout*, i.e., selection of actions until a terminal state is reached, and (4) *backpropagation* of the feedback up the tree. A

tree policy guides the selection phase and a *default policy* guides the playout phase. We refer to a *simulation* as to the sequence of actions performed during the selection and playout phases.

The *MCTS* framework was devised in 2006 when Coulom (2006) combined Monte Carlo search with an *incremental tree structure*. Simultaneously, Kocsis and Szepesvári (2006) combined tree search with the *UCB1 selection policy* for bandits (Auer et al., 2002), designing the first and currently the most popular MCTS method – the *upper confidence bounds for trees (UCT)* algorithm.

In terms of the MCTS framework, the first UCT implementation (Kocsis & Szepesvári, 2006) uses UCB1 selection as tree policy and random selection as default policy; 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) (Kocsis, Szepesvári, & Willemson, 2006); considers the MDP-based discounting of rewards; and distinguishes between terminal and nonterminal rewards by updating the values of visited states each with its appropriate return (i.e., with the sum of rewards following its visit in a simulation). In contrast, MCTS practitioners have widely adopted a variant of UCT that memorizes only one new state per iteration; does not identify transpositions and builds a tree instead of a graph; disregards discounting of rewards; and does not distinguish between terminal and nonterminal rewards – it backs up only the final outcome (i.e., the sum of all rewards in a simulation), updating all visited states with the same value (Browne et al., 2012). We refer to these two variants as the *original* and the *standard* UCT algorithm, respectively. The pseudocode of each algorithm 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 (Browne et al., 2012). 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 (Lorentz, 2008). Finally, applied MCTS algorithms are often further augmented with domain-specific enhancements, with some popular examples being the *move-average sampling technique* (MAST, Finnsson & Björnsson, 2008), the *all-moves-as-first* heuristic (AMAF, Gelly & Wang, 2006), and *rapid action value estimation* (RAVE, Gelly & Silver, 2011).

3. The Connection Between Monte Carlo Tree Search and Reinforcement Learning

First, we discuss about the 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 describe the similarities of MCTS methods with RL applied to planning: we link the basic terminology to show that many of the mechanics are in practice equal. We also investigate 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 (Sutton & Barto, 1998) and have also been re-analysed by several researchers in the recent years (Asmuth & Littman, 2011; Silver, Sutton, & Müller, 2012; Sutton & Barto, 2017). 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 method 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 mechanics 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 games are often episodic, but knowledge (i.e., learning) is usually not transferred between episodes or different 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 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, Perez et al., 2015, for example) and usually it does not disclose the transition probabilities and does not simulate the moves of other agents involved in the task. For example, in multi-player games, the other players are often understood as a part of the environment that cannot be easily predicted, and are rather modelled separately. Summing the above, MCTS methods also 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 (otherwise the Markov property of MDPs does not hold): 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 MDP-solving methods 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 (2015) 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, Gelly & Wang, 2006, 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 (Gelly, Wang, Munos, & Teytaud, 2006). Even though the original UCT algorithm (Kocsis & Szepesvári, 2006) was designed and analysed using MDPs (Section 2.4), the community adopted a variant that ignores discounting of rewards, which is an integral part of MDPs (Section 2.1) 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 following section makes a step towards this goal and illustrates how can RL methods using Monte Carlo sampling be understood as a generalization of MCTS methods.

3.2 The Similarities: Linking the Terminology

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

3.2.1 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. When a model of the task is available, 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 – it comprises the visited states and selected actions in both the selection and the playout phase. Episodes and trajectories have a length of one or more (simulated) *time steps*, with one selected action per time step.

3.2.2 OBSERVING REWARDS

The gathered experience (i.e., the trajectories and the collected rewards) is used to improve the value estimates. MDP-based methods, including RL and the standard UCT variant (Kocsis & Szepesvári, 2006), 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 complies with RL theory, whereas the *standard* UCT variant (Browne et al., 2012) does not (Section 2.4); however, since UCT is often used on tasks with terminal rewards only (e.g., like many classic games), the issue is mitigated.

3.2.3 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.

3.2.4 GUIDING THE EXPLORATION

In RL, trajectories are guided by a *control policy*, which dictates what actions will be performed, and, consequently, which states will be visited. It encompasses the MCTS's tree policy and default policy. A policy may explicitly maintain the probabilities of selecting actions in states $\pi(a|s)$ (all or a part of them), or may compute them only when needed and discard them afterwards (e.g., compute them only for the current state). The latter usually requires less computational resources, hence it is frequent in practical implementations of RL methods, and is also typical for MCTS algorithms. Reinforcement-learning methods that learn by Monte Carlo sampling classify as *Monte Carlo control* methods, and MCTS methods can be classified as such as well. A popular policy used in classic RL algorithms is the *ε-greedy* policy, where the best-valued action is assigned a high probability ($1 - \varepsilon + \varepsilon/|\mathcal{A}|$) and other actions are assigned low probabilities ($\varepsilon/|\mathcal{A}|$). Similarly to MCTS methods, recently also RL methods started to employ stronger exploration policies (Sutton & Barto, 2017), with some of these based on (contextual) bandits (Chu, Li, Reyzin, & Schapire, 2011; Osband, Blundell, Pritzel, & Van Roy, 2016).

3.2.5 FINDING A GOOD POLICY

Gathering samples and evaluating the value function by following a fixed policy is known as *policy evaluation*. Refining a policy based on previous evaluations (i.e., based on the value function) is known as *policy improvement*. Iterating through these two processes in alternation is known as the paradigm of *generalized policy iteration* (GPI, Sutton & Barto, 1998). 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. Also, the current policy and value function of a 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 – i.e., the *anytime* behaviour. Lastly, increasing the number of GPI iterations usually results in a more optimal solution, the same as in MCTS.

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

3.2.6 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 evaluating *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* (Sutton & Barto, 1998) (or *post-decision states,* Van Roy, Bertsekas, Lee, & Tsitsiklis, 1997). For example, on deterministic tasks $Q(s, a) = V(s')$, 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. MCTS methods can also be implemented either way. Such mechanics were observed by several MCTS researchers: Childs, Brodeur, and Kocsis (2008) distinguish between *UCT1* (evaluating state-actions) and *UCT2* (evaluating afterstates) as two variants of the UCT algorithm when used with transpositions; and Saffidine, Cazenave, and Mehat (2010) 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.

3.2.7 FIRST-VISIT AND EVERY-VISIT UPDATES

When building a tree, each node in the trajectory is visited exactly once per episode; however, when building a directed graph, a single node might be visited multiple times in an episode. In such a setting, RL algorithms might update the value of a node either for its *every visit* or only for its *first visit* in an episode. 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 occurrence of a specific node, and would ignore its other occurrences. Multiple updates per episode may cause a slower convergence in tasks with cycles, hence every-visit algorithms are less robust and often perform worse (Singh & Sutton, 1996; Sutton & Barto, 1998). Nevertheless, when transpositions are not used, then there is no difference between the two approaches. Childs et al. (2008) 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.

3.2.8 ON-POLICY AND OFF-POLICY EXPLORATION

From an RL point of view, an agent might be acting using 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 (Sutton & Barto, 2017). This is useful in, for example, planning tasks, where during the available simulation time it is important to keep exploring, but at the end it is preferred to output the best action according to a greedy policy. 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

2. This is not to be confused with Q-learning (Watkins, 1989) – the use of afterstates is independent of the learning algorithm.

target policy). Coulom (2006) analysed such backup methods along proposing the basic MCTS framework, but he did not derive them from the RL theory. Also, Feldman and Domshlak (2014) 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).

3.2.9 TEMPORAL-DIFFERENCE LEARNING AND MCTS

Reinforcement-learning methods that evaluate and improve policies with bootstrapping TD-learning backups instead of Monte Carlo backups (Section 2) are known as *TD control* methods. Since these are a generalization of MC control methods, they are related also to MCTS methods. MCTS methods that evaluate nodes by averaging the outcomes produce backups that equal those of a TD(1) algorithm (Sutton, 1988), and their four MCTS phases resemble an iteration of a Sarsa(1) algorithm (Rummery & Niranjan, 1994) – an on-policy TD control algorithm. MCTS methods that evaluate nodes differently (e.g., that back up the maximum value of children nodes, as proposed in Coulom, 2006) resemble off-policy TD methods like Q-learning (Watkins, 1989), for example. In the context of MCTS, Childs et al. (2008) analysed bootstrapping mechanics soon after the invention of MCTS, 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 to TD(0).

3.2.10 DESCRIBING MCTS ALGORITHMS WITH RL TERMINOLOGY

Summing the above, the search mechanics of MCTS are comparable to those of *Monte Carlo control* (and to TD control with $\lambda = 1$). The *original* UCT (Kocsis & Szepesvári, 2006) searches identically as an *offline on-policy every-visit MC control* algorithm (Sutton & Barto, 1998) that uses UCB1 as the policy; this similarity stems from the fact that the original UCT algorithm is based on MDPs. On the other hand, the *standard* UCT (Browne et al., 2012) 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 in the next section. 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 Section 4.

3.3 The Differences: The Novelties of Monte Carlo Tree Search

As shown so far, many of the MCTS mechanics can be described with RL theory; despite the different points of view, the underlying concepts are in practice very similar. Silver et al. (2012) 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 *expansion*. We argue that these two concepts are not commonly seen as standard RL procedures and they seem to be the key innovations of MCTS in relation to RL theory.

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 suggest that features can be learned on the fly (Silver et al., 2016). *Tabular* RL algorithms that 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 (Browne et al., 2012). 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. This memorization is then handled in the MCTS *expansion phase*. There has been plenty of research on such memorization techniques, ranging from replacement schemes for transposition tables (Kocsis et al., 2006), which have already been thoroughly studied in the α-β framework (Breuker, Uiterwijk, & van den Herik, 1994), to MCTS expansion-phase enhancements, such as progressive unpruning and progressive widening (Coulom, 2007; Chaslot, Winands, van den Herik, Uiterwijk, & Bouzy, 2008).

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 either 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 (Utgoff, 1989), adaptive state aggregation (Bertsekas & Castanon, 1989), automatic online feature identification (Geramifard et al., 2011), online basis function construction (Keller, Mannor, & Precup, 2006), and incremental representations (Silver, 2009). We will refer to such methods as *adaptive representations*.

Within the RL community, the use of adaptive representations is decades-old (Sutton et al., 1993), but recently it gained even more prominence through the successes of Go (Silver et al., 2016). Also, advances in representation learning and global function approximation (which, for example, resolve catastrophic forgetting, Riedmiller, 2005) led to the popularization of neural networks (Mnih et al., 2015, 2016). 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 (Bellemare et al., 2016; Pathak et al., 2017). Evolutionary methods have also seen a resurgence (Salimans, Ho, Chen, & Sutskever,

2017); these too have a long history within RL contexts (Stanley & Miikkulainen, 2002); they are less sensitive to hyperparamaters, features, and reward shaping. Finally, as an evolution of Dyna-like architectures (Sutton, 1990), Silver et al. (2017) 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 (or state-action) space. There was a number of methods developed (Menache, Mannor, & Shimkin, 2005; Yu & Bertsekas, 2009; Mahmood & Sutton, 2013) 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 (Whiteson et al., 2007; Ratitch & Precup, 2004), 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. In addition, 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 (Li, Littman, & Walsh, 2008; Walsh, Goschin, & Littman, 2010; Szita & Szepesvári, 2011) – 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.

3.4 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 mechanics as traditional RL methods.

3.4.1 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 (Watkins, 1989). Coulom (2006) 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 (2011) 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, Winands, Pepels, and Sturtevant (2014) 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 (Silver et al., 2016) and the studies leading to it (Section 5.3) 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 (2013) provide simi-

lar conclusions also for several MCTS-minimax hybrid approaches that do not require evaluation functions.

3.4.2 REWARD-WEIGHTING

In the context of MCTS, the eligibility traces mechanism translates to weighting the iterations according to their duration, e.g., diminishing the weight of rewards with increasing distance to the 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, several MCTS reward-weighting schemes could classify as simplified or similar approaches: Xie and Liu (2009) partition the sequence of MCTS iterations by time, and weight differently the feedback from each partition; Cowling, Ward, and Powley (2012) exponentially decrease the reward based on the number of plies from the root to the terminal state with a discount parameter in range $[0, 1]$; and Kocsis and Szepesvári (2006) diminish the UCBs exploration rate with increasing tree depth, which is similar to decreasing the weight of memorized rewards close to the root.

3.4.3 INITIAL BIAS AND HEURISTICS

Some MCTS enhancements strongly relate to the RL concepts of initial values (V_{init}) and forgetting (driven by the update step-size α). Gelly and Wang (2006) name the mechanic of assigning initial values in MCTS as the *first-play urgency*. Chaslot et al. (2008) propose the *progressive bias* enhancement, where the impact of the initial bias decreases when the state gets more visits. The urgency or progressive bias are often set according to heuristic or previously-learned knowledge – a widespread approach is to use *history heuristics* (Schaeffer, 1989), i.e., the information from previously-played moves. Kozelek (2009) used it in the MCTS selection phase and achieved a significant improvement in the game of Arimaa. Finnsson and Björnsson (2008) 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 (2007) set the initial values of unvisited nodes in the tree to the same value as their grandfathers and labelled this as the *grandfather heuristic*. Nijssen and Winands (2011) propose the *progressive history* enhancement – a type of progressive bias that uses history score as the heuristic bias. Other popular MCTS enhancements, such as *AMAF* (Gelly & Wang, 2006), *RAVE* (Gelly & Silver, 2011), and *last good reply* (LGR, Drake, 2009; Baier & Drake, 2010), are also related to history heuristics.

3.4.4 FORGETTING

Apart from progressive bias (Chaslot et al., 2008), several other "forgetting" mechanics were applied to MCTS. The most basic and popular one is discarding the tree (partially or completely) after each search (Browne et al., 2012). Baier and Drake (2010) indirectly applied forgetting through the one-move and two-move LGR playout policies (Drake, 2009), which remember and then prioritize moves that previously led to a win in response to an opponents move. Tak, Winands, and Bjornsson (2014) observe the benefits of forgetting the estimates produced by domain-independent playout policies and propose the application of a decay factor to the *n-gram selection technique* (Tak, Winands, & Björnsson, 2012) and to MAST (Finnsson & Björnsson, 2008). Hashimoto, Kishimoto, Yoshizoe, and Ikeda (2012) proposed the *accelerated UCT* algorithm, which assigns higher importance to recently visited actions. It proved beneficial on several two-player games and increased the strength of the Computer Go algorithm Fuego (Enzenberger, Muller, Arneson, & Segal,

2010). Feldman and Domshlak (2014) 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* (Sutton & Barto, 1998), because its parameter α operates in a similar way as the learning rate α used in RL algorithms, for example, as in (3).

4. Integrating Monte Carlo Tree Search and Reinforcement Learning

In the previous sections we covered the strong connection between MCTS and 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. As an example proof of how can these concepts be efficiently combined, here we introduce a framework that unifies the advantages of both fields, and develop an algorithm that generalizes UCT in such a way.

4.1 A Representation Policy for Reinforcement Learning

To develop a framework that could describe the incremental representations used by MCTS algorithms, we first require 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 the notion of a *representation policy* – this 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 (Silver et al., 2012) the representation policy keeps a fixed size and shape of the feature-approximated state space, whereas in the standard UCT algorithm (Browne et al., 2012), 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 3.3). 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.

4.2 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 (3), 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 (7) or (8), 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 (4) 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*.

With 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, Finnsson & Björnsson, 2008), 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 (see Appendix B for a more

complex, but arguably better option). 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 – in Appendix B we give more justification and details on example basic assumptions. 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.

4.3 The Temporal-Difference Tree Search Framework

Based on the extended RL theory, we design the *temporal-difference tree search (TDTS)* framework – a generalization of MCTS that replaces Monte Carlo (MC) backups with bootstrapping backups (Section 2.3) and that introduces into MCTS other established RL concepts (Section 3.2). 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 representation; (3) playout – control in the non-memorized part of the search space; and (4) backpropagation – updating the value estimates. In case of online backups, the backpropagation phase is invoked after each time step during the selection and playout phases (the remaining phases are not affected by this). The following subsections describe the main benefits of TDTS in regards to RL methods, and in regards to MCTS methods.

4.3.1 THE BENEFITS OF MCTS CONCEPTS FOR RL

Traditional RL methods can benefit from several MCTS-inspired mechanics, implemented by the TDTS framework, such as incremental representations, ingenuous generalizations (MCTS enhancements) of the state space that is not memorized in the main representation, and strong heuristic control policies (especially for the playout phase), for example. The main proof of the advantages of such mechanics is a decade of MCTS successes in domains where RL could previously not achieve such performance (Browne et al., 2012), with the most outstanding examples being Go (Gelly & Wang, 2006), general game playing (Finnsson & Bjornsson, 2009), and general video game playing (Perez et al., 2015). Also, the strength of joint MCTS and RL methods was observed in several

studies (Section 5.3), with the most remarkable being the use of a playout policy in the AlphaGo engine (Silver et al., 2016).

Many of these MCTS mechanics and their benefits stem from the use of a representation that does not keep estimates for the whole state space at once and that gets adapted online (Section 3.3). This is controlled by the TDTS representation policy and we regard it as the key novelty of the TDTS framework (and MCTS methods in general) in comparison to traditional RL methods. Below we detail how such "incomplete" representations can be beneficial.

When planning, many of the states visited in the simulations might never get visited for real. For example, in tasks that are naturally centred on the current position of the learning agent (such as games), the simulated states that occur far away from the current state of the agent are less likely to occur for real. Allocating and updating estimates for all the (visited) states might therefore be a waste of memory space and computational resources; it might be better to rather spend the available memory on the states that are more likely to get visited – on those closer to the current state. This applies also to approximate representations: in general, rather that approximating poorly the whole space, it might be better to accurately approximate the relevant part of the space and less accurately (or not at all) the remaining part. This is exactly what MCTS methods do – for the states near the root they store all the information (in a tabular tree representation) with high accuracy, but for the (usually) vast state space visited in the playouts they either omit the estimates or apply low-accuracy generalization methods that estimate the states that are not memorized in the tree (Baier, 2015). MCTS researchers soon confirmed that, given unlimited memory and time, memorizing only one new visited state in each simulated episode decreases the performance only slightly in comparison to memorizing all of them (Coulom, 2006; Gelly et al., 2006); our experiments also re-confirm this (Section 6.1). Popular examples that prove the advantages of low-accuracy generalization methods are MAST (Finnsson & Björnsson, 2008), AMAF (Gelly & Wang, 2006), and RAVE (Gelly & Silver, 2011) – we experiment with these three (Section 6.2), but there are many more (Browne et al., 2012). Altogether, explicit adaptive and incremental representations provide alternative efficient ways for solving tasks with large state spaces (Section 3.3). And furthermore, they do not require domain-specific features to approximate the state space, but they can still be combined with such features to produce approximate representations. Finally, the same concepts and reasoning apply beyond planning tasks also to learning tasks, where the agent gathers real experience.

4.3.2 THE BENEFITS OF RL CONCEPTS FOR MCTS

The benefits of integrating RL mechanics (Section 3.2) with MCTS have been confirmed by several researchers (Section 5), with the AlphaGo engine being the most outstanding proof (Silver et al., 2016). 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.

4.4 The Sarsa-UCT(λ) Algorithm

As an instance of a TDTS method, we combine Sarsa(λ) (Rummery & Niranjan, 1994) and UCT (Kocsis & Szepesvári, 2006) 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 studied MCTS algorithm – it is the backbone of several MCTS enhancements 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.

Algorithm 1 presents an Sarsa-UCT(λ) iteration when employing an incremental tree representation and evaluating actions with afterstates (hence the use of V-values instead of Q-values). The essential difference with the standard UCT algorithm (Section 2.4) is the backpropagation method (lines 23–38); however, the algorithms behave equally when $\lambda = \gamma = 1$. A re-improvement over the standard UCT is observing non-terminal rewards (lines 15–16), just like in the original UCT. This is more principled, since the algorithm does not give credit to an action for the rewards that happened before that action was taken (Sections 2.1 and 3.2). This may 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 still work correctly, albeit the convergence might be slower.

Most importantly, Sarsa-UCT(λ) can be used interchangeably with UCT in any of the existing MCTS enhancements – for example, in the same way as in our previous work (Vodopivec & Šter, 2014), where we combined a similar algorithm (TD-UCT) with AMAF (Gelly & Wang, 2006) and RAVE (Gelly & Silver, 2011). Also, the presented implementation has practically the same computational complexity as UCT – an iteration requires $O(T)$ time, where T is the number of steps in the episode – with only a slight constant increase due to the more sophisticated backups.

The *eligibility-trace* mechanism is efficiently computed with decaying and accumulating the TD errors (line 33) during the backpropagation phase (the derivation is in Appendix A). 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 at their first visits in each episode: such code would extend each $episode$ entry (line 16) with a first-visit flag, and check for it before backing up (line 34). 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 UCBs exploration bias (line 44) must be modified to consider the sum of children visit counters instead of the parent's counter $tree(s).n$.

The algorithm has no need to memorize the probabilities of selecting actions $\pi(a|s)$, because these are implicitly determined by the UCB1 policy (line 46) at each time step: the policy assigns a probability 1 to the action with the highest value $Q_{\mathrm{UCB}}(s, a)$ and probabilities 0 to the remaining actions. When multiple actions are best, they are given equal probability (i.e., random tie-breaking).

Algorithm 1 An iteration of a tabular offline Sarsa-UCT(λ) algorithm that builds a tree and evaluates actions with afterstates.

parameters: C_p, α (optional), $\gamma, \lambda, V_{\text{init}}, V_{\text{playout}}$ ▷ V_{init} and V_{playout} can be constants or functions
global variable: $tree$ ▷ the memorized experience

procedure SARSA-UCT-ITERATION(S_0)
    $episode \leftarrow$ GENERATEEPISODE(S_0)
    EXPANDTREE($episode$)
    BACKUPTDERRORS($episode$)

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)
        $(s, R) \leftarrow$SIMULATETRANSITION(s, a)
        Append (s, R) to $episode$
    return $episode$

procedure EXPANDTREE($episode$)
    $S_{\text{new}} \leftarrow$ first s in $episode$ that is not in $tree$
    Insert S_{new} in $tree$
    $tree(S_{\text{new}}).V \leftarrow V_{\text{init}}(S_{\text{new}})$ ▷ initial state value
    $tree(S_{\text{new}}).n \leftarrow 0$ ▷ counter of updates

procedure BACKUPTDERRORS($episode$)
    $\delta_{\text{sum}} \leftarrow 0$ ▷ cumulative decayed TD error
    $V_{\text{next}} \leftarrow 0$
    for $i =$ LENGTH($episode$) **down to** 1
        $(s, R) \leftarrow episode(i)$
        if s is in $tree$
            $V_{\text{current}} \leftarrow tree(s).V$
        else ▷ assumed playout value
            $V_{\text{current}} \leftarrow V_{\text{playout}}(s)$
        $\delta \leftarrow R + \gamma V_{\text{next}} - V_{\text{current}}$ ▷ single TD error
        $\delta_{\text{sum}} \leftarrow \lambda\gamma\delta_{\text{sum}} + \delta$ ▷ decay and accumulate
        if s is in $tree$ ▷ update value
            $tree(s).n \leftarrow tree(s).n + 1$
            $\alpha \leftarrow \frac{1}{tree(s).n}$ ▷ example of MC-like step-size
            $tree(s).V \leftarrow tree(s).V + \alpha\delta_{sum}$
        $V_{\text{next}} \leftarrow V_{\text{current}}$ ▷ remember the value from before the update

procedure UCB1TREEPOLICY(s)
    for each a_i in s
        $S_i \leftarrow$ afterstate where a_i leads to from s
        if S_i is in $tree$
            $V_{\text{norm}} \leftarrow$ NORMALIZE$\big(tree(S_i).V\big)$ ▷ normalization to $[0, 1]$
            $Q_{\text{UCB}}(s, a_i) \leftarrow V_{\text{norm}} + C_p\sqrt{\frac{2\ln(tree(s).n)}{tree(S_i).n}}$
        else
            $Q_{\text{UCB}}(s, a_i) \leftarrow \infty$
    return $\text{argmax}_a \left(Q_{\text{UCB}}(s, a)\right)$

The UCB1 selection policy requires values normalized in the interval $[0, 1]$. Most MCTS methods solve this by fixing the parameter C_p to the maximum possible return of an episode (Kocsis & Szepesvári, 2006); however, such bound is in practice very wide when tasks have nonterminal rewards or when γ or λ are less than 1. Hence, in Sarsa-UCT(λ) we introduce a normalization technique for V-values that produces a tighter bound (line 42, details not given in pseudocode): each tree node memorizes the all-time maximum and minimum values of its direct children nodes and uses them as normalization bounds. In nodes where these *local* bounds are not yet valid (e.g., in new nodes), the *global* all-time minimum and maximum return are used instead. We label this technique as *space-local value normalization*. It is analogous to dynamically adapting the parameter C_p for each state.

Algorithm 1 presents a specific variant of Sarsa-UCT(λ) 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 44) with $-V_{\mathrm{norm}}$ instead of V_{norm} when simulating the opponent's moves; we use this approach when evaluating the algorithm on classic games (Section 6.2). When extending to multi-player tasks, a general approach 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. Similarly, 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 basic way for applying the algorithm 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 on the GVG-AI games (Section 6.3).

4.5 The Parameters and their Mechanics

Our Sarsa-UCT(λ) algorithm extends the *standard* UCT algorithm with four new parameters: the update step-size α, the eligibility trace decay rate λ, the initial state values $V_{\mathrm{init}}(s)$, and the assumed playout state values $V_{\mathrm{playout}}(s)$. 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 (Kocsis & Szepesvári, 2006), as mentioned in Section 2.4, and the exploration rate C_p is present in most UCT variants. The new parameters generalize UCT with several mechanics (Table 1), many of which had already been analysed by the MCTS community through different MCTS-extensions (Section 3.4), 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.

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 the searches. A popular approach is to decrease α inversely with the number of updates or visits of a state (as presented in Algorithm 1, line 35) – this is the default MC approach, as given by (3). It corresponds to averaging the feedback, giving equal weight to each

Parameters	Control mechanics	Example uses in MCTS
Exploration rate $C_p \geq 0$	The exploration tendency of the UCB (Auer et al., 2002) policy.	In the original UCT (Kocsis & Szepesvári, 2006) and in its variants. Also an alternative value-normalization method for the UCB policy.
Update step-size $\alpha \in [0, 1]$	The forgetting rate: the impact of initial values and the weight of individual feedbacks.	Forgetting techniques (Baier & Drake, 2010; Stankiewicz, 2011; Hashimoto et al., 2012; Tak et al., 2014; Feldman & Domshlak, 2014), progressive bias (Chaslot et al., 2008), progressive history (Nijssen & Winands, 2011; Nijssen, 2013), BRUE(α) (Feldman & Domshlak, 2014)
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 (Gelly & Wang, 2006), history heuristics (Schaeffer, 1989; Kozelek, 2009), grandfather heuristics and use of offline-learned values (Gelly & Silver, 2007), several enhancements with heuristics and expert knowledge in the tree and expansion phases (Browne et al., 2012).
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 (Kocsis & Szepesvári, 2006), omitted from many of its later variants (Browne et al., 2012).
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 (Xie & Liu, 2009; Cowling et al., 2012), indirectly also in the original UCT through adapting the C_p value (Kocsis & Szepesvári, 2006).
Playout values $V_{\text{playout}}(s)$ or $Q_{\text{playout}}(s, a)$	Prior or expert knowledge in the playout: guide the playout policy and impact the accumulation of TD errors in the playout.	Several enhancements with heuristics and expert knowledge in the playout phase (Browne et al., 2012).

Table 1: The parameters of Sarsa-UCT(λ).

update. In such case, when the starting value of α is 1, the *initial values* $V_{\text{init}}(s)$ and $Q_{\text{init}}(s, a)$ have no impact because they get overwritten after the first visit. However, for other starting values of α, the initial values provide a way of biasing the estimates with *prior* or *expert knowledge*. When the initial values are good approximations of the true values, it is sensible to start with an $\alpha < 1$ to not override them – the smaller the starting α value, the more the algorithm relies on initial values. 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 $[0, 1]$.

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 by (6). This is a way of modelling the *reliability* of distant feedback (e.g., long playouts might be less reliable) and might be useful when the generative model is not accurate or the policy is far from optimal (e.g., random). Except for this, the two mechanics differ essentially due the role of γ in (4). 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 sum of rewards, which might be undesired. On the other hand, a $\lambda < 1$ enables *bootstrapping* TD backups, which decrease the variance of the evaluations and increases their bias (Section 4.7). In this way the estimates change more slowly (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 (6), we can implement *every-visit* or *first-visit* update mechanics. Due to the above, in the rest of this paper we will be mostly interested in the effect of the trace decay rate λ, since this is the parameter that actually changes backup mechanics towards TD learning, whereas γ is not central to us, since is has already been known to MCTS methods.

The assumed *playout values* $V_{\text{playout}}(s)$ and $Q_{\text{playout}}(s, a)$ guide the playout policy and are not necessarily constant or equal for every playout state, but might get adapted online – they provide an entry point for expert knowledge into the playout phase and for MCTS generalization enhancements, as detailed in Section 4.2.

4.6 An Example of the Algorithm Operation

We present a simple example to illustrate the algorithm in operation. Assume that the root node corresponds to state S_t, which is followed by two memorized states, S_{t+1} and S_{t+2} (Figure 1). The

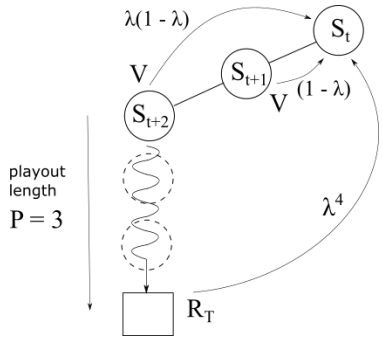


Figure 1: A simple example showing the algorithm operation.

latter is followed by a random simulated playout until the end of the episode, where the reward R_T is delivered ($T = t + 5$). The playout length is $P = 3$. We assume playout values are equal. The root state S_t is then updated (after the end of the episode) as

$$\begin{aligned} V(S_t) \leftarrow V(S_t) \quad + \quad \alpha_n \big[& \quad \big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big) \\ & + \ \gamma\lambda \ \ \big(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1})\big) \\ & + (\gamma\lambda)^2 \big(R_{t+3} + \gamma V_{\text{playout}} - V(S_{t+2})\big) \\ & + (\gamma\lambda)^3 \left(R_{t+4} + \gamma V_{\text{playout}} - V_{\text{playout}}\right) \\ & + (\gamma\lambda)^4 \left(R_T - V_{\text{playout}}\right) \big] \ . \end{aligned} \tag{9}$$

When $\gamma = 1$ and the only non-zero reward is R_T, we get

$$\begin{aligned} V(S_t) \;\leftarrow\; & V(S_t) + \alpha_n \big[\quad \big(V(S_{t+1}) - V(S_t)\big) \\ & \qquad + \; \lambda \big(V(S_{t+2}) - V(S_{t+1})\big) \\ & \qquad + \; \lambda^2 \big(V_{\text{playout}} - V(S_{t+2})\big) \\ & \qquad + \; \lambda^3 \big(V_{\text{playout}} - V_{\text{playout}}\big) \\ & \qquad + \; \lambda^4 \big(R_T - V_{\text{playout}}\big) \big] \end{aligned} \tag{10}$$

$$\begin{aligned} = \;\; & V(S_t) + \frac{1}{n} \big[\lambda^4 R_T + (\lambda^2 - \lambda^4) V_{\text{playout}} + \lambda(1-\lambda) V(S_{t+2}) \\ & \qquad + (1-\lambda) V(S_{t+1}) - V(S_t) \big] \; . \end{aligned} \tag{11}$$

The term $\lambda^4 R_T$ equals to $\lambda^{T-t-1} R_T$, where $T - t$ is the number of nodes from the root to the final state. Therefore, when $\lambda < 1$, the play length $T - t$ impacts the state-value updates. On the other hand, when $\lambda = 1$, the play length has no impact (and neither have the playout values V_{playout}), since the algorithm produces ordinary Monte Carlo backups by

$$V(S_t) \leftarrow V(S_t) + \frac{1}{n} \big[R_T - V(S_t) \big] \; . \tag{12}$$

4.7 Why is $\lambda < 1$ Beneficial?

Bootstrapping backups ($\lambda < 1$) reduce the variance of the estimates, but increase their bias. In large tasks, when performing planning (as in MCTS), many states get visited only a few times, causing the variance to have a much larger impact than the bias. Therefore, a $\lambda < 1$ can be used to decrease the variance and possibly improve the overall performance of the algorithm (Silver et al., 2012).

Here we explore another reason why $\lambda < 1$ can work better than $\lambda = 1$ in an MCTS-like setting. The same as in the previous example, we assume $\gamma = 1$ and that R_t is the only non-zero reward. Initially, all children of the root node are played out once before any of them gets further explored. Assume that the possible outcomes are win ($R_T = 1$) and lose ($R_T = 0$) – each may happen with some probability (see Figure 2). After all the children have been created and played out, one of

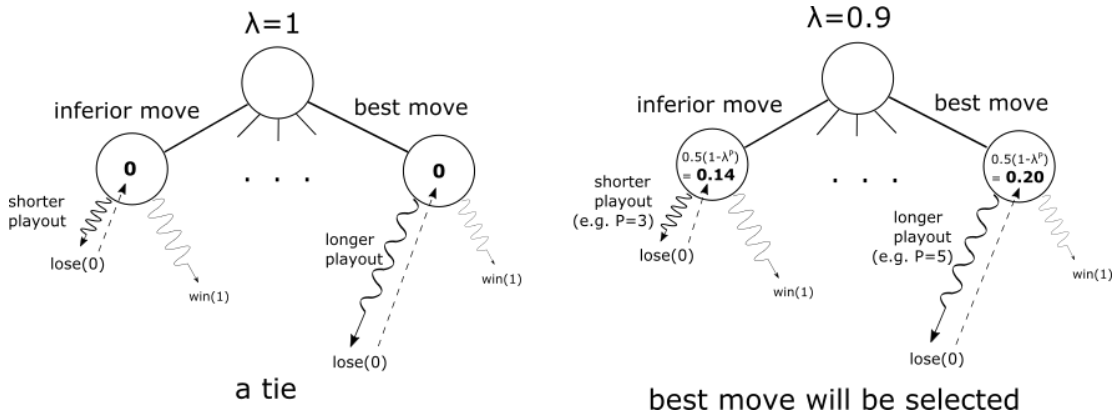


Figure 2: An example where the playout length impacts the state values ($\lambda = 0.9$).

them is to be selected for further exploration.

Imagine that two moves compete: the best one and an inferior one. Assume that both have led to a defeat (after random playout). In case no further nodes are added and no further playouts performed, the move leading to the higher-valued node will be selected. In case of the ordinary MCTS ($\lambda = 1$), after both playouts the value of 0 will be propagated to both nodes. Therefore, there will be a tie, which is broken randomly – both moves will have equal chance of getting selected. On the other hand, when $\lambda < 1$, the values of these states are not necessarily zero. All children estimates get updated as

$$V \leftarrow V + \alpha \left[(V_{\text{playout}} - V) + \lambda (V_{\text{playout}} - V_{\text{playout}}) + ... + \lambda^P (R_T - V_{\text{playout}}) \right] . \tag{13}$$

By assuming that V are initially $V_{\text{init}} = 0.5$, $\alpha = 1/n$, and $R_T = 0$, the update is computed as

$$V \leftarrow 0.5 + \lambda^P (R_T - 0.5) = 0.5 + \lambda^P (0 - 0.5) = 0.5(1 - \lambda^P) . \tag{14}$$

Therefore, when $\lambda = 1$, both nodes would have a value of 0, whereas when $\lambda < 1$, *a higher P (longer playout) would lead to a higher V value*. In this way, better moves get higher values, even when the outcome is essentially the same. One may argue that there is no guarantee that better moves for one player lead to longer playouts in cases where the second player wins. However, this seems to be usually the case in games.

An alternative scenario is to assume that the two playouts both led to a win. In such case a $\lambda = 1$ would again result in a tie, due to both estimates being 1. On the other hand, when $\lambda < 1$ the estimates would be

$$V \leftarrow 0.5 + \lambda^P (R_T - 0.5) = 0.5 + \lambda^P (1 - 0.5) = 0.5(1 + \lambda^P) . \tag{15}$$

In this case, smaller P values cause higher V values – better moves have higher values since they lead to shorter playouts when the first player is winning.

This example concentrates on the children of the root node; however, the same principles apply when further nodes are added to the tree. The example also applies both to single-player and multi-player scenarios. It is important to notice that the reward here is not discounted ($\gamma < 1$). For example, the win is always worth 1, regardless of the number of moves.

In the study above and in our experiments (Section 6) we try to compare the impact of a $\lambda < 1$ in comparison to $\lambda = 1$ in MCTS-like algorithms. There have been several attempts to find appropriate values of this parameter in RL. Sutton and Singh (1994) suggested that at each time step the eligibility trace λ should be chosen as equal to the transition probability of the preceding transition. While deriving rigorous upper bounds on the error of TD algorithms, Kearns and Singh (2000) provided schedules for adaptive updating of λ. Downey and Sanner (2010) applied a Bayesian approach to setting λ. Konidaris, Niekum, and Thomas (2011) devised the TD$_\gamma$ algorithm, which removes the parameter λ. White and White (2016) confirm that adapting λ is much harder than adapting the learning rate α, and proposed a linear-complexity λ-adaptation algorithm. Although the focus of our paper is not on optimizing λ, most of these approaches could easily be applied concurrently with our algorithm.

5. Survey of Research Inspired by both Fields

We survey 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 (Silver et al., 2012). Then we survey the studies that combined techniques from both fields (without explicitly commenting 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 (2007) 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, Silver et al. (2012) 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*. Baier (2015) presents a concise "learning view on MCTS" in the introductory part of his thesis. 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* (Gelly & Wang, 2006) and *RAVE* (Gelly & Silver, 2011), 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 (Rimmel & Teytaud, 2010; Browne et al., 2012) 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 (1998). A second edition is in progress, where they also discuss MCTS methods (Sutton & Barto, 2017) in a similar way as in the TD search study (Silver et al., 2012).

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, the TD search study (Silver et al., 2012) is heavily focused on its playing strength in Go and in finding good function-approximation features for achieving this, and Baier (2015) 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. 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 mechanics (both similar and different) with concepts and terms from both communities, that surveys MCTS enhancements that implement similar mechanics 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, on-line 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 mechanics 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 (Silver et al., 2012) 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 were 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 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 (Silver et al., 2012). 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. 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.

Hester and Stone (2013) in their RL framework for robots *TEXPLORE* employ an algorithm similar to the one we proposed. 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 (Hester & Stone, 2010) and use UCT only to guide the state-space exploration. They present an extension of the original UCT with λ-returns, 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).

Keller and Helmert (2013) 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 mechanics, specifically, the use of dynamic-programming (DP) backups instead of Monte Carlo backups. Their *MaxUCT* algorithm modifies the backpropagation to perform *asynchronous value iteration* (Sutton & Barto, 1998) with online model-learning. Due to the backup of maximum values instead of means, which was first explored in an MCTS setting by Coulom (2006) and later by Ramanujan and Selman (2011), such algorithms are related to Q-learning (Watkins, 1989). Due to model-learning, they are also related to *adaptive real-time dynamic programming* (Barto, Bradtke, & Singh, 1995) and the *Dyna* framework (Sutton, 1990). Feldman and Domshlak (2014b) continue the work above and analyse the use of DP updates in MCTS more in depth (Feldman & Domshlak, 2014a).

Simultaneously to our research, Khandelwal, Liebman, Niekum, and Stone (2016) 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* (Sutton & Barto, 1998) 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. Lastly, Sarsa-UCT(λ) normalizes the value-estimates before passing them to the UCB selection policy.

The researchers noted so far mainly focused on backpropagation methods, but, as described, some also extended the basic UCT algorithm with online model-learning – a well-understood mechanic in RL (Sutton & Barto, 1998). Veness, Ng, Hutter, Uther, and Silver (2011b) 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, naming it ρ*UCT*.

In addition to Veness et al. (2011b) and Silver et al. (2012), there are more researchers that acknowledge a strong relation between RL and MCTS. Asmuth and Littman (2011) derive from RL and MDPs when producing the *BFS3* algorithm by extending the *forward search sparse sampling* technique (Walsh et al., 2010), which is an MCTS-like sample-based planner inspired by *sparse sampling* (Kearns, Mansour, & Ng, 2002). They restate the RL view of learning and planning and reconfirm the strong connection between the two from a Bayesian perspective. Silver and Veness (2010) extend MCTS to partially observable MDPs, producing the *partially observable Monte-Carlo planning* algorithm; Guez, Silver, and Dayan (2013) 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 (2010) develop the *contextual MCTS* algorithm by enhancing the UCT playout phase with *tile coding* (a known function approximation method in RL, Sutton & Barto, 1998). Wang and Sebag (2013) develop the *multi-objective MCTS* algorithm and regard it as "the first extension of MCTS to multi-objective reinforcement learning (Gábor, Kalmár, & Szepesvári, 1998)".

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 (2007) were the first to combine the benefits of both fields: they used offline TD-learned values of shape features from the *RLGO* player (Silver, Sutton, & Müller, 2007) as initial estimates for the MCTS-based player MoGo (Gelly et al., 2006). Soon afterwards, Silver, Sutton, and Müller (2008) 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 Dyna (Sutton, 1990). Daswani, Sunehag, and Hutter (2014) suggest using UCT as an oracle to gather training samples that are then used for RL feature-learning, which can be seen as an augmentation of Dyna-2 for feature-learning. Finnsson and Björnsson (2010) employ *gradient-descent TD* (Sutton, 1988) 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 (Genesereth, Love, & Pell, 2005). Ilhan and Etaner-Uyar (2017) also learn a linear function approximator online, but through the *true online Sarsa(λ)* algorithm (Van Seijen, Mahmood, Pilarski, Machado, & Sutton, 2016) 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, Rohlfshagen, and Lucas (2011) employ a similar approach to Finnsson and Björnsson (2010), but they learn the approximator already offline and evaluate the performance on the game of Othello; they observe that guiding the default policy is more beneficial than guiding the tree policy. Osaki, Shibahara, Tajima, and Kotani (2008) developed the *TDMC(λ)* algorithm, which enhances TD learning by using winning probabilities as substitutes for rewards in nonterminal positions. They gather these probabilities with plain Monte Carlo sampling; however, as future research they propose to use the UCT algorithm.

The *AlphaGo* (Silver et al., 2016) engine is certainly the most successful combination of RL and MCTS estimates in modern history. In March 2016, it overcame the *grand challenge of Go* (Gelly et al., 2012) 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 (LeCun, Bengio, & Hinton, 2015), 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.

Although not directly influenced by MCTS, Veness, Silver, Uther, and Blair (2009) 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. Experimental Evaluation of Temporal-Difference Tree Search Algorithms

Our main experimental goal is to discover whether swapping Monte Carlo (MC) backups with temporal-difference (TD) backups increases the planning performance in MCTS-like algorithms. Therefore, we evaluate 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.

First 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. Then we proceed with the evaluation of Sarsa-UCT(λ) (Algorithm 1) on classic two-player games, where it learns from self-play, and on real-time arcade video games, where little computation time is available per move.

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 Toy Games

We test a tabular first-visit Sarsa(λ) algorithm (Rummery & Niranjan, 1994) 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 policies are either random, ε-greedy, or UCB1 with and without the value-normalization as described in Section 4.4. The playout policy is random. These configurations produce algorithms that span from the on-policy MC control (Sutton & Barto, 1998) to our Sarsa-UCT(λ) algorithm, including both the *original* (Kocsis & Szepesvári, 2006) and *standard* (Browne et al., 2012) UCT variants.

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 3). The *Random walk* game gives a reward of $+1$ when reaching the rightmost state and 0 otherwise, whereas the *Shortest walk* game gives a reward of 0 when reaching the rightmost state and -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 (Sutton & Barto, 1998) and MCTS communities (Cazenave, 2009; Saffidine et al., 2010). On Random walk we merely test the quality of policy evaluation using a random 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.

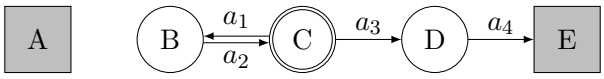


Figure 3: A five-state *left-right* toy game with an example episode that starts from C and terminates in E after four actions.

We observe the quality of the value function, the one-step planning performance, and the overall planning performance. The first is measured as the root mean square error (RMSE) of the learned values from the optimal values. The second is measured as the probability of selecting the optimal action in the starting state by a fully greedy policy – this is analogous to the *failure rate* (Kocsis & Szepesvári, 2006) and *choice-error probability* (Feldman & Domshlak, 2012) metrics. The third 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 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.

The main control parameters are the eligibility trace decay rate $\lambda \in [0, 1]$ and the available computational time per search, which is expressed as the number of either simulated time steps or simulated episodes (MCTS iterations) during planning. 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 is held constant at 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 (chosen according to the length of the optimal solution of the Shortest walk game); the assumed playout values Q_{playout} equal Q_{init} or 0; the ε-greedy policy exploration rate ε is constant at 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 UCB1 exploration rate C_p is fixed on 1. The UCB exploration bias is computed according to Algorithm 1 (line 44). For each configuration we ran at least 10000 repeats and averaged the results, so that the confidence bounds are insignificantly small.

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 (Figure 4 examples a small subset of our results, for full results and additional findings see Appendix C). This has been long-known for the default RL setting (Sutton & Barto, 1998) (i.e., when building a directed graph and memorizing all nodes) and has recently been observed in specific MCTS configurations (Silver et al., 2008; Hester & Stone, 2013; Khandelwal et al., 2016); however, it has not been observed yet in the default MCTS setting when using an incremental tree structure, i.e., when adding a limited number of nodes per episode and without using transpositions. Furthermore, TD backups produce a larger gain over MC backups exactly when not using transpositions; when using transpositions, they produce only a marginal improvement.

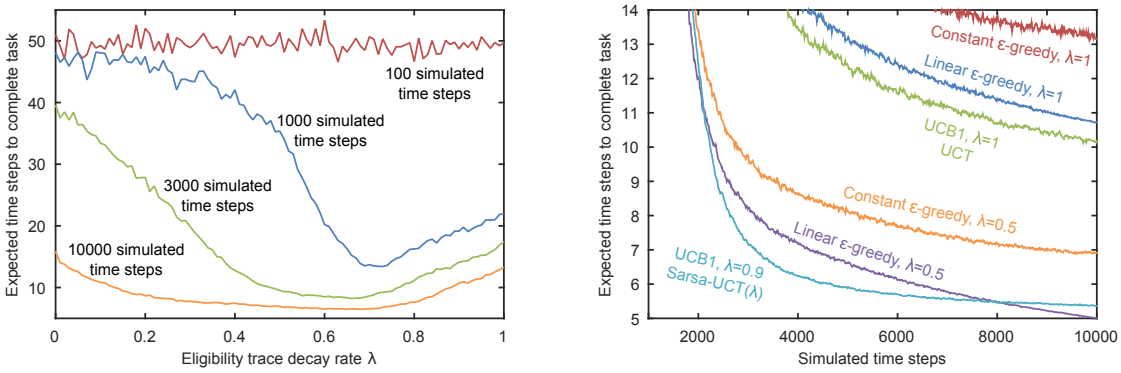


Figure 4: The overall planning performance of TDTS algorithms when building a tree (without using transpositions) and adding one node per episode on a Shortest walk game of length 11, where an optimal policy completes the task in 5 time steps. (a) Sensitivity to the eligibility trace decay rate λ when using an ε-greedy policy. (b) The benefit of TD backups under different policies. UCB1 uses space-local value normalization (Section 4.4). The given values of λ are those that perform best with their respective policies.

The optimal λ is usually larger than 0 and smaller than 1, but $\lambda = 0$ often performs worse than $\lambda = 1$ (Figure 4a). There seems to be 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 terms of performance). In general, when increasing the available computational time, the threshold and the optimal λ decrease towards some value below 1. Decreasing the size of the game (playing a game with less states) has a similar effect.

A $\lambda < 1$ also performs better regardless of the employed policy (Figure 4b); this confirms that (in this domain) the Sarsa-UCT(λ) algorithm is superior to the UCT algorithm. 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. We also note that our experiments do not identify which policy performs best – values of C_p and ε that we have not tested might further improve the two policies and even change the balance between them.

An inversely decreasing update step-size α performs better than a constant one, as expected due to the convergence requirements of such algorithms and because the task is stationary and small (Sutton & Barto, 1998). Initial values V_{init} improve the performance when set to an informed value (e.g., to 0.5 in the Random walk game), but may also seriously detriment it when set badly (e.g., to 10 in the Shortest walk game). Setting playout values V_{playout} to equal initial values performs better than setting them to 0, except when the initial values are bad, as just noted. In such case the second option is safer (less biased). 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. Nonetheless,

because both initial and playout values have no impact when $\lambda = 1$ and $\gamma = 1$, MC backups are safer regarding the bias.

6.2 Classic Two-Player Games

We investigate the planning performance of TD backups when learning from simulated self-play on adversarial multi-agent tasks. Hence, we measure the playing strength of the Sarsa-UCT(λ) algorithm on the games of Tic-tac-toe, Connect four, Gomoku, and Hex. We also compare its performance with popular MCTS enhancements.

The playing strength 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. Sarsa-UCT(λ) computes TD errors from one-ply successor states, same as the *TD-Gammon* algorithm (Tesauro, 1994), and in contrast to the *TD search* algorithm (Silver et al., 2012), which computed them from two-ply successors in the game of Go (i.e., ignoring the values of opponent's simulated moves). To mimic an adversarial setting, we implement the minimax behaviour in Sarsa-UCT(λ): we extend Algorithm 1 (line 44) to compute the Q-values with $-V_{\text{norm}}$ instead of V_{norm} (i.e., to select the lowest-evaluated actions) when simulating the opponents' moves.

The 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. 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 36). 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 games feedback a reward of 1 for a win, 0.5 for a draw, and 0 for a loss. These results are averaged from at least 2000 and up to 20000 repeats; the resulting 95%-confidence bounds are insignificantly small for our observations.

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. 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; 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. 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 results (Figure 5) 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 otherwise. 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.

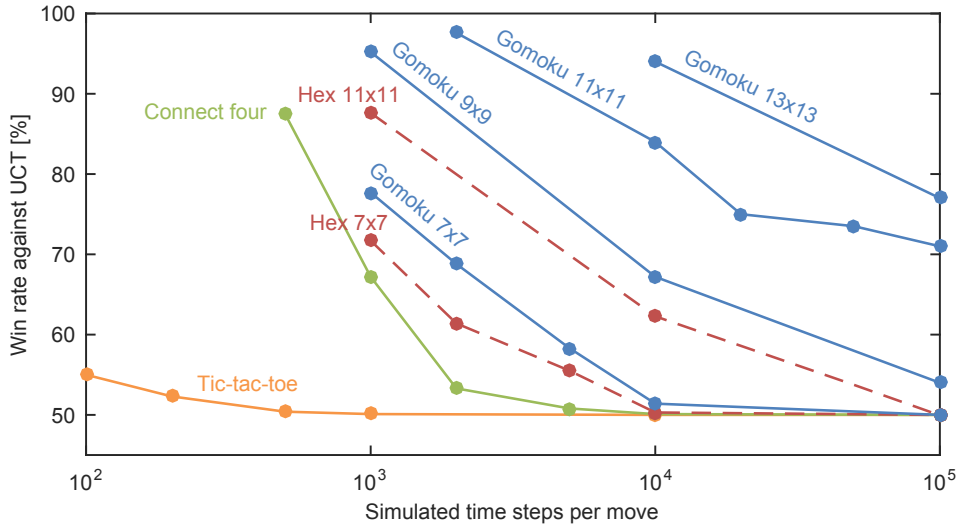


Figure 5: Performance of Sarsa-UCT(λ) on classic two-player games.

Since these two-player games produce non-zero rewards only in terminal states, UCB1 performs well also without value normalization (the results in Figure 5 show such configuration). Still, even on these games, space-local value normalization further increases the performance when $\lambda < 1$ and decreases the sensitivity to λ. Also concurrently tuning 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. Therefore, it also affects the optimal value of C_p. See Appendix D for full results on two-player games.

We also assess the gain in performance of TD backups in regards to the popular MCTS enhancements MAST (Finnsson & Björnsson, 2008), AMAF (Gelly & Wang, 2006), and RAVE (Gelly & Silver, 2011), which are beneficial on Hex, Go, and general game playing (Genesereth et al., 2005). Similar to our previous observations, extending UCT with TD backups is most competitive when the available time per move is low (Table 2); otherwise, the individual enhancements usually outperform Sarsa-UCT(λ). However, TD backups seem more robust as they are beneficial regardless of the game and they cannot be detrimental (as explained earlier). Also, TD backups can be efficiently combined with these (and other) MCTS enhancements: here we illustrate this by combining Sarsa-UCT(λ) with the MAST enhancement (last column in Table 2), whereas in our previous study (Vodopivec & Šter, 2014) we experimented with several variants of UCT algorithms with TD backups in combination with AMAF and RAVE enhancements. In both cases we observe a major boost in performance of such joint algorithms on the games Connect four, Hex, and Gomoku. We also observed that UCT algorithms enhanced with TD backups are equally robust to parameter values, available computation time, and game size, as is the non-enhanced UCT algorithm, and are less sensitive compared to a UCT algorithm that uses AMAF or RAVE. Apart from our experiments, Silver (2009) observed that TD(0) backups improve the performance of the UCT algorithm also when combined with the Fuego playout policy (Enzenberger et al., 2010) on Go. We provide a thorough discussion on TD backups and MCTS enhancements in Section 7.

Game	Simulated time steps per move	Win rate against UCT [%] UCT + MAST	UCT + AMAF	UCT + RAVE	Sarsa-UCT(λ)	Sarsa-UCT(λ) + MAST
Connect four	500	67.5	63.7	68.0	87.5	99.3
	2000	60.2	59.0	60.0	53.3	83.7
	10000	49.5	23.5	50.8	50.1	52.8
Hex 11x11	1000	93.3	55.7	57.5	87.7	98.8
	10000	68.5	50.0	56.3	62.0	87.0
	100000	59.8	50.0	55.3	50.0	59.3
Gomoku 9x9	1000	91.4	71.3	75.6	95.2	99.8
	10000	80.0	77.2	74.6	67.2	96.5
	100000	54.8	61.5	67.5	54.0	65.8

Table 2: Performance of Sarsa-UCT(λ) compared to established MCTS enhancements.

We note the implementational choices of the tested MCTS enhancements. MAST assists the control policy with tie-breaking in the tree phase and performs softmax-random selection in the playout phase (same as in the original implementation in Finnsson & Björnsson, 2008). Its estimates are re-set after each move, which equals to *move decay* (Tak et al., 2014) with a factor 0, i.e., 100% forgetting. When used in combination with the Sarsa-UCT(λ) algorithm, its estimates get updated with the same TD errors as used for updating the ordinary state-value estimates in the tree. Two variants of AMAF were evaluated: one that updates the same estimates as the UCT algorithm, in this way affecting the UCB-exploratory bias, and one that keeps separate estimates and does not affect the exploratory bias. The results are given for the better variant. RAVE computes the value of each action as a weighted sum of RAVE and MC (or TD) estimates. The weighting factor is computed for each action a in each state s by $\max(1 - \frac{\text{visits}(s,a)}{V_{\text{RAVE}}}, 0)$, where V_{RAVE} is an arbitrary threshold (the same as in Helmbold & Parker-Wood, 2009). The parameters were optimized in the same way as described earlier for Sarsa-UCT(λ).

6.3 The General Video Game AI Competition

The *general video game AI (GVG-AI) competition* (Perez et al., 2015) challenges AI players on single-player and two-player arcade games that derive from early computer video games (e.g., Pac-man, Sokoban, The legend of Zelda, etc.). The games run in real-time and impose a 40-milliseconds limit per move. During this time, the players may determine the best moves by simulating episodes with the help of a generative 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 2.1), 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 GVG-AI 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 (actions).

At each competition event, the players 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, an algorithms performance gets averaged from 50 executions on each game (totalling in 500 executions). The algorithms get ranked separately for each game according to the three metrics, and then a Formula-1 score system is applied for each game: the algorithms get awarded points in respect to their rank. Finally, the points are summed across all games and the winner is the algorithm with the 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).

We implement our single-player Sarsa-UCT(λ) controller by extending the implementation of the *standard UCT* variant that is provided in the GVG-AI framework (it is labelled as the *sample-OLMCTS* controller). Since we are mainly interested in the benefit of TD backups, we configure 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 44). 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. It 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. Reward-modelling is also the same as in the framework-given UCT implementation: the algorithm observes rewards only at the end of simulations and computes them by $victoryStatus \times 10^7 + score$, where $victoryStatus$ is either -1, 0, or 1. Before applying the UCB equation, the algorithms normalize the value estimates to $[0, 1]$ based on the minimum and maximum return observed so far.

We experimentally measure the win rate and score of our player on the GVG-AI single-player training sets 1, 2, and 3 (each set consists of 10 games). With each value of λ we compute at least 50 repeats for each of the 5 levels of each of the 30 games. The results (Table 3) 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. A $\lambda = 0.6$ performed best in our local experiments, and a $\lambda = 0.8$ seemed to perform similarly well on the first test set on the GVG-AI server, so we submitted a single-player Sarsa-UCT(0.8) to the CIG'15 and CEEC'15 single-player GVG-AI competitions, and a Sarsa-UCT(0.6) to subsequent competitions.

In addition to the single-player controller (labelled on GVG-AI as *ToVo1*), we developed also a two-player variant (labelled *ToVo2*) and submitted it to the two-player GVG-AI competitions. In contrast to our single-player controller, here we configured Sarsa-UCT(λ) to use more of its potential: we set it to discount rewards through $\gamma < 1$, memorize all visited states, observe nonterminal rewards, forget via a more sophisticated update scheme for α, and use two minor enhancements: an adaptive playout-truncation threshold and non-uniform-random action-selection in the playouts.

The detailed results and the source code of both algorithms are available on the GVG-AI web site (`www.gvgai.net`). Here we provide a summary of past competitions; so far, the majority of them resulted in favour of our controller. The single-player variant outperformed the framework-given UCT controller in 4 out of 5 competitions (Table 4), and, most importantly, our two-player version outperformed all other controllers and ranked first in 2 out of 3 competitions (Table 5), and has also been achieving top positions on all two-player training sets on the GVG-AI server so far. It

		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 standard UCT ($\lambda = 1$).

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

Table 3: Performance of a Sarsa-UCT(λ) player on the first 30 games of the GVG-AI framework. A setting of $\lambda = 1$ corresponds to the UCT algorithm.

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.

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 entries.

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

		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

Table 4: Results from GVG-AI single-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

Table 5: Results from GVG-AI two-player competitions.

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 3) – 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 (2016), 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, the latter has already been heavily enhanced – with progressive history (Nijssen & Winands, 2011), n-gram selection techniques (Tak et al., 2012), 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 3 out 4 game sets, which suggests our algorithm is competitive also against such heavily-enhanced MCTS algorithms.

7. Discussion and Future Work

Our experiments confirm that swapping Monte Carlo (MC) backups with temporal-difference (TD) backups in MCTS-like algorithms is beneficial, especially 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 (Sections 4.6 and 4.7), 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. Note that although a lower discount factor ($\gamma < 1$) would have a similar effect, its use would be hard to justify since the length of the play does not directly affect the final result (e.g., in most games victory counts the same, no matter how many moves were played).

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 (Sutton & Barto, 1998) 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 $\lambda = 1$. Its optimal value differs from task to task: it depends heavily on the search-space size of the given task, 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 increases on tasks with a larger size and 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 then a constant value of $\lambda < 1$ may 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 or is somehow adapted online (which we have not

experimented with yet). We deem this the main disadvantage of TDTS methods. Getting the rest of the parameters (at least approximately) right is usually easier than finding an optimal λ.

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, 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. As noted previously, the relationship between MCTS and RL is a two-way street: a number of MCTS enhancements are in practice heuristics that appeal to certain characteristics of certain games, but there is no reason why they (at least on principle) could not be transferred back to RL.

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 $\lambda < 1$).

There is also plenty of unexplored potential for the presented TDTS framework and Sarsa-UCT(λ) algorithm: for example, tuning the reward discount rate γ, using transpositions on complex games, and measuring the benefits of online updates. Also, we have not experimented with constant or linear schemes for the update step-size α other than on toy games, where the state space is small. Since the policy usually improves with time, recent rewards might be more relevant than old ones; therefore, forgetting schemes that give higher weight to these rewards might be beneficial.

Sarsa-UCT(λ) is by definition an on-policy method and is not intended for performing backups differently than presented in Algorithm 1, e.g., for backing up the maximum value instead of the last feedback. Hence, combining off-policy TD learning methods, such as Q-learning, with MCTS-like incremental representations might also be interesting. Off-policy algorithms might diverge (Sutton & Barto, 1998); still, in the MCTS literature there were some successful applications of such types of backups (Coulom, 2006).

Finally, in our experimental 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. It is possible to transfer across the specialized control policies and ingenious playout-related generalization approaches from the numerous MCTS enhancements, but, above all, we highlight the exploration of explicit incremental representations for RL algorithms. This is a complex topic, but also intriguing, and deserves more study. Another venue that is worth pursuing is learning a forward model with a limited horizon during an RL learning phase; during acting the RL agent can actively use the forward model by doing local searches, hence possibly further improving the policy. One could take this idea to its limit and learn forward models in *feature* space; this has recently been introduced (Pathak et al., 2017; Silver et al., 2017) in the context of

end-to-end learning, but we postulate the models can be decoupled and one can potentially learn models of various timescales and horizons. Finally, one could combine TDTS-like methods within a framework similar to expert iteration (Anthony et al., 2017; Silver et al., 2017), using TDTS as a policy improvement operator/expert.

8. Conclusion

This study portrayed Monte Carlo tree search (MCTS) from a reinforcement learning (RL) perspective. We outlined that many MCTS methods evaluate states in the same way as the TD(1) learning algorithm and behave similarly to the Sarsa(1) algorithm (two canonical RL algorithms). 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*, and then integrated temporal-difference (TD) learning into MCTS and devised the *temporal-difference tree search (TDTS)* framework. The latter classifies as a specific configuration of Silver's TD search (Silver, 2009), but also extends it with the novel concepts from MCTS. As an instance of a TDTS method, we generalized the UCT algorithm into *Sarsa-UCT(λ)*. In addition to retaining the generality and the same computational cost, the new algorithm performs better on several types of toy games and classic games and achieves high rankings in competitive video games.

This study presented a practical view on extending MCTS with RL mechanics, 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 with an incremental or adaptive representation. If we managed to intrigue the MCTS reader to explore the RL view on planning and search (Sutton & Barto, 2017) or the RL reader to experiment with any of the numerous MCTS enhancements (Browne et al., 2012), specialized control policies, or incremental representations, then we fulfilled our goal.

We argue the fields of reinforcement learning and heuristic search, including Monte Carlo tree 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. 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.

Acknowledgements

The authors wish to thank the anonymous reviewers and associate editor for their help in improving this manuscript – their advice helped clarify and enforce the key message of this study. We would like to thank also Michael Fairbank and Mark Nelson for providing very useful last-minute feedback.

Appendix A: On TD(λ) and the Derivation of Sarsa-UCT(λ)

Between the targets G_t and $R_{t+1} + \gamma V_t(S_{t+1})$ it is possible to define intermediate targets, called the n-step returns:

$$\begin{aligned} G_t^{(1)} &= R_{t+1} + \gamma V_t(S_{t+1}) && (16)\\ G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2}) && (17)\\ &\dots \\ G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V_t(S_{t+n})\,. && (18) \end{aligned}$$

These returns can be averaged together in different ways; the following linear combination is known as the λ-return:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_t^{(n)} + \lambda^{T-t-1} G_t\,, \tag{19}$$

which can also be written as

$$G_t^\lambda = G_t^{(1)} + \sum_{n=2}^{T-t-1} \lambda^{n-1}\gamma^{n-1}\delta_{t+n-1} + \lambda^{T-t-1}(G_t - G_t^{(T-t-1)}) \tag{20}$$

by using the *temporal-difference errors (TD errors)* defined as

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) && (21)\\ \delta_{t+1} &= R_{t+2} + \gamma V_t(S_{t+2}) - V_t(S_{t+1}) && (22)\\ &\dots \\ \delta_{t+m} &= R_{t+m+1} + \gamma V_t(S_{t+m+1}) - V_t(S_{t+m})\,. && (23) \end{aligned}$$

The last term in (20) can be written as

$$G_t - G_t^{(T-t-1)} = \gamma^{T-t-1}(R_T - V_t(S_{T-1})) \tag{24}$$

and the last TD error is

$$\delta_{T-1} = R_T + \gamma \cdot 0 - V_t(S_{T-1})\,. \tag{25}$$

We assume that the terminal state $V_t(s_T) = 0$.

The λ-return algorithm updates state values as

$$\begin{aligned} V_{t+1}(S_t) &= V_t(S_t) + \alpha_t \left[G_t^\lambda - V_t(S_t)\right] \\ &= V_t(S_t) + \alpha_t \left[G_t^{(1)} + \sum_{n=2}^{T-t-1} (\lambda\gamma)^{n-1}\delta_{t+n-1} + (\lambda\gamma)^{T-t-1}\delta_{T-1} - V_t(S_t)\right] \\ &= V_t(S_t) + \alpha_t \left[\delta_t + \sum_{n=2}^{T-t-1} (\lambda\gamma)^{n-1}\delta_{t+n-1} + (\lambda\gamma)^{T-t-1}\delta_{T-1}\right] \\ &= V_t(S_t) + \alpha_t \left[\sum_{n=1}^{T-t} (\lambda\gamma)^{n-1}\delta_{t+n-1}\right] \\ &= V_t(S_t) + \alpha_t \left[(\gamma\lambda)^0\delta_t + (\gamma\lambda)^1\delta_{t+1} + \dots + (\gamma\lambda)^{T-t-2}\delta_{T-2} + (\gamma\lambda)^{T-t-1}\delta_{T-1}\right]\,. \end{aligned}$$

The values of the other states get updated in a similar manner:

$$\begin{aligned} V_{t+1}(S_{t+1}) &= V_t(S_{t+1}) + \alpha_t \left[(\gamma\lambda)^0 \delta_{t+1} + (\gamma\lambda)^1 \delta_{t+2} + ... + (\gamma\lambda)^{T-t-2} \delta_{T-1} \right] , \\ V_{t+1}(S_{t+2}) &= V_t(S_{t+2}) + \alpha_t \left[(\gamma\lambda)^0 \delta_{t+2} + (\gamma\lambda)^1 \delta_{t+3} + ... + (\gamma\lambda)^{T-t-3} \delta_{T-1} \right] , \end{aligned}$$

etc. From the implementational point of view, it is better to consider the so-called backward view of the algorithm, which uses eligibility traces.

The Sarsa-UCT(λ) algorithm is similar to TD(λ). But since the state values of the playout states are not known, they are assumed to have a value of V_{playout}:

$$\begin{aligned} V(S_t) &\leftarrow V(S_t) + \alpha \left[\sum_{n=0}^{T-t-1} (\gamma\lambda)^n \delta_{t+n} \right] \qquad (26) \\ &= V(S_t) + \alpha \Bigg[\sum_{n=0}^{T-t-P-1} (\gamma\lambda)^n \Big(R_{t+n+1} + \gamma V_t(S_{t+n+1}) - V_t(S_{t+n}) \Big) \\ &\qquad + (\gamma\lambda)^{T-t-P} \Big(R_{T-P+1} + \gamma V_{\text{playout}} - V_t(S_{T-P}) \Big) \\ &\quad + \sum_{n=T-t-P+1}^{T-t-2} (\gamma\lambda)^n \Big(R_{t+n+1} + \gamma V_{\text{playout}} - V_{\text{playout}} \Big) \\ &\qquad + (\gamma\lambda)^{T-t-1} \Big(R_T - V_{\text{playout}} \Big) \Bigg] , \end{aligned}$$

where P is the playout length.

Appendix B: Basic Assumptions on Playout Estimates

As already noted in Section 4.2, we regard the need of a playout value function and of playout assumptions a formal by-product of not having the whole state space described by one (primary) representation, but rather to recognize that a part of the search space is not represented at all, or that it is represented by another (secondary) representation – the playout value function.

When implementing a general RL algorithm, one will have to decide how to compute the TD errors in the playout – even ignoring the issue altogether is in fact an assumption made on the playout values (and playout value function). Therefore, we substantiate why we deem this formality meaningful with three example basic assumptions below.

A natural assumption would be that non-memorized estimates hold the assumed initial value, since they have not been updated yet: $V_{\text{playout}} = V_{\text{init}}$. This might perform well when the discount rate $\gamma = 1$, but a potentially undesired effect arises when $\gamma < 1$: at each playout step a negative TD error would be produced due to $(\gamma V_{\text{playout}} - V_{\text{playout}} < 0)$, hence the cumulative backup error would keep increasing with the duration of the playout.

Alternatively, assuming that all the non-memorized estimates have a value of 0 causes no TD errors during the playout regardless of the value of the discount rate γ, but still does not equal to ignoring the issue altogether, because it 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).

Finally, an assumption that equals to ignoring the issue altogether must prevent $\gamma < 1$ from producing bootstrapping errors when entering the playout and during the playout. To achieve

this, playout estimates must have unequal values as to satisfy all the terms $\gamma V_{\text{playout}}(S_{t+1}) = V_{\text{playout}}(S_t)$ that occur in the playout. This can be achieved when the first playout value is set to match $V(S_{\text{leaf}})/\gamma$, where S_{leaf} is the last-encountered memorized state (the tree-leaf node), and each next playout value is further divided with γ. Only in such case the TD errors would be produced only by rewards R_t, regardless of the configuration of the algorithm.

Since ignoring this issue of missing playout estimates in theory equals to the third assumption described above, although it is more difficult to implement directly, we believe that acknowledging this is more principled and in line with the RL theory (and with TD-backup equations). The alternative would be to implement the algorithm so that in the playouts it computes TD errors only from rewards, but we deem this less principled, as it would require two different TD(λ) update rules, one for the represented part of the search space and one for the non-represented part of the search space.

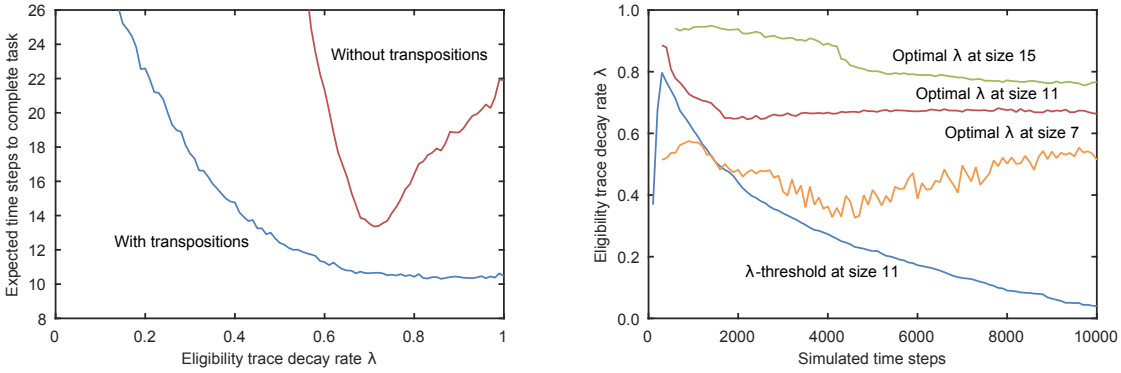
Appendix C: Detailed Results from Toy-Game Experiments

Here we list more results that back up the observations from our toy-game experiments (from Section 6.1): the effect of transpositions (Figure 6a), sensitivity to λ when increasing the computational time and the search space (Figure 6b), the benefits of space-local value normalization for the UCB1 policy (Figures 7a and 7b), and the impact of initial values (Figure 8a).

We also provide additional observations. The experiments reconfirm that adding only one node per episode instead of all nodes does not critically inhibit convergence (Coulom, 2006; Gelly et al., 2006): the impact is minimal when using transpositions and moderate when not using them (Figure 8b). 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 6a).

The two planning-performance metrics provide similar conclusions; although, the *overall planning performance* is arguably the most informative (e.g., by comparing Figures 9a and 9b), because it considers the ranking of all memorized actions and not only of those in the root state. 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 10). 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 (Sutton & Barto, 1998) – 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 (Lanctot, Saffidine, Veness, Archibald, & Winands, 2013). 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 (Veness, Lanctot, & Bowling, 2011a). 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.

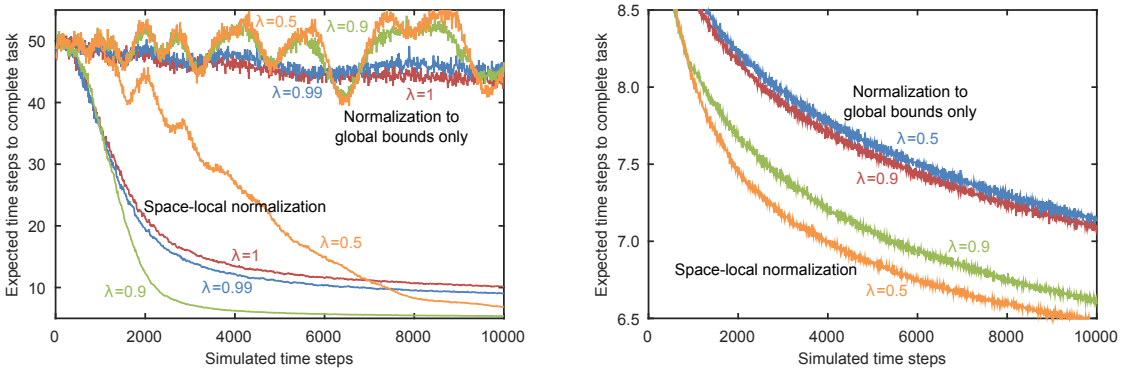
Except where stated differently, the figures in this appendix portray a TDTS algorithm that employs Sarsa(λ) with an ε-greedy policy (with $\varepsilon = 0.1$) and random playout policy, builds a tree, does not use transpositions, adds one new node per episode, sets initial values to 0, and assumes



(a) (b)

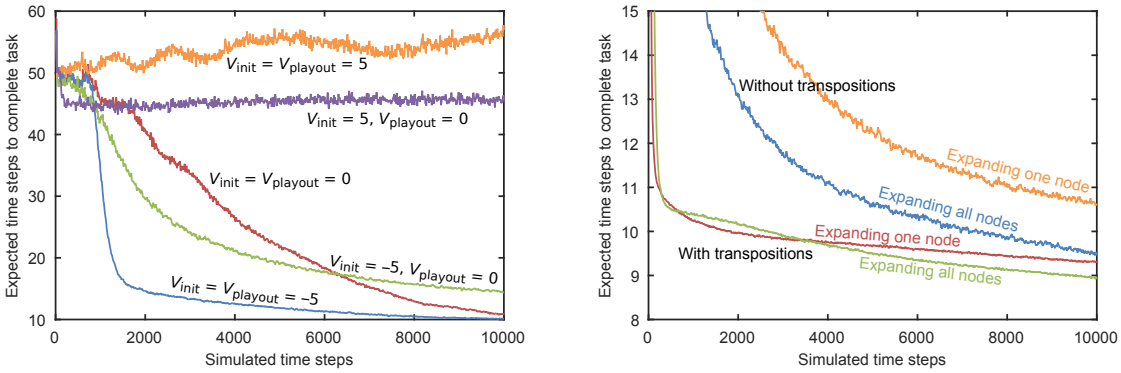
Figure 6: (a) Performing TD backups ($\lambda < 1$) over MC backups ($\lambda = 1$) is more beneficial when not using transpositions. In such case, the optimal λ is usually between 0 and 1. The results were computed after the algorithm simulated 1000 time steps (the setting is the same as described in Section 6.1). (b) Generally, when not using transpositions, increasing the computation time or decreasing the game size 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.

playout values equal to initial values; on the Shortest walk toy game of size 11. The confidence bounds are insignificantly small due to a high number of repeats.



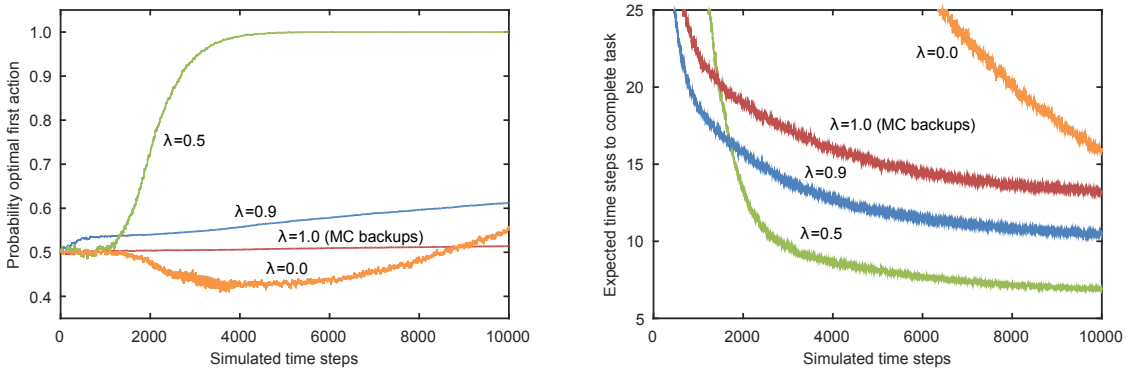
(a) (b)

Figure 7: The performance of the UCB1 policy in combination with our space-local value normalization and in combination with a simpler normalization of values between the maximum and minimum return (sum of rewards within an episode) encountered so far. (a) When not using transpositions, space-local value normalization is necessary for TD backups ($\lambda < 1$) to improve on MC backups ($\lambda = 1$). (b) When using transpositions, it is not necessary, but it still significantly improves the performance.



(a) (b)

Figure 8: (a) 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. (b) 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.



(a) (b)

Figure 9: (a) One-step planning performance and (b) overall planning performance.

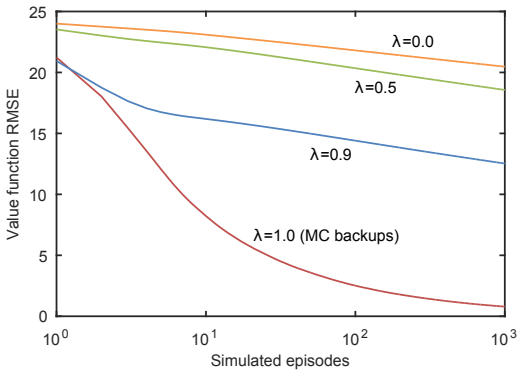
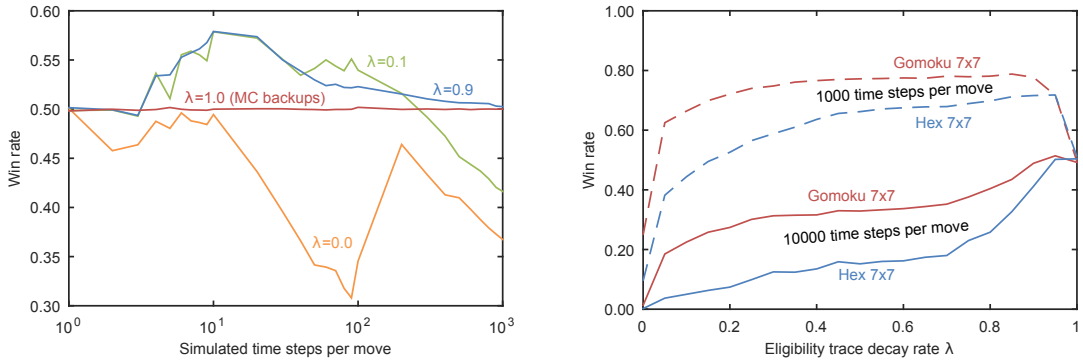


Figure 10: Quality of the value function. Since a tree is built, the root mean squared error (RMSE) is observed only for the root state and its children. Algorithms that use a $\lambda < 1$ converge considerably slower towards the optimal value function. However, this metric might be misleading, because a $\lambda < 1$ still improves on the policy (Figures 9a and 9b).

Appendix D: Detailed Results from Two-Player Games

Here we provide example results about the sensitivity of Sarsa-UCT(λ) to the parameter λ games when increasing the available computational time (number of simulated time steps per move) on Tic-tac-toe (Figure 11a), and Gomoku and Hex (Figure 11b). Furthermore, we illustrate the benefits of our value-normalization technique (Figure 12). Lastly, we give full results of our two-player experiments (Table 6). The configuration of the algorithms and experiments is given in Section 6.2.



(a) (b)

Figure 11: The sensitivity of Sarsa-UCT(λ), expressed as win rate against the UCT algorithm, to the parameter λ and the available computation time on (a) Tic-tac-toe and (b) Gomoku and Hex. On all two-player games, increasing the number of simulated time steps per move causes the optimal value of λ to shift towards 1. The results also show that UCT behaves identically to Sarsa-UCT(1), as expected (Sections 3.2 and 4.4) – their performances are equal, despite our distinct implementations of the two algorithms.

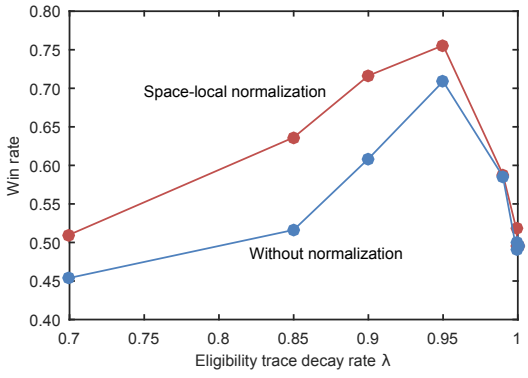


Figure 12: The benefit of space-local value normalization on Sarsa-UCT(λ), expressed as win rate against UCT. Example results on Gomoku 11×11 at 100000 simulated time steps per move and $C_P = 0.05$. Each point was computed from at least 4000 repeats.

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

Table 6: 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.

References

Anthony, T., Tian, Z., & Barber, D. (2017). Thinking fast and slow with deep learning and tree search. *arXiv preprint arXiv:1705.08439*.

Asmuth, J., & Littman, M. (2011). Learning is planning: near Bayes-optimal reinforcement learning via Monte-Carlo tree search. In *Proceedings of The 27th Conference on Uncertainty in Artificial Intelligence (UAI-11)*.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, *47*(2-3), 235–256.

Baier, H., & Drake, P. (2010). The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go.. *2*(4), 303–309.

Baier, H. (2015). *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*. Ph.d. thesis, Maastricht University.

Baier, H., & Winands, M. H. M. (2013). Monte-Carlo Tree Search and minimax hybrids. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, No. c, pp. 1–8. IEEE.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 0–66.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., & Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pp. 1471–1479.

Bertsekas, D., & Castanon, D. (1989). Adaptive aggregation methods for infinite horizon dynamic programming. *Automatic Control, IEEE Transactions on*, *34*(6), 589–598.

Breuker, D. M., Uiterwijk, J. W. H. M., & van den Herik, H. J. (1994). Replacement schemes for transposition tables. *ICCA Journal*, *17*(4), 183–193.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43.

Cazenave, T. (2009). Nested Monte-Carlo Search. In *International Joint Conference on Artificial Intelligence*, pp. 456–461.

Chaslot, G. M. J.-B., Winands, M. H. M., van den Herik, H. J., Uiterwijk, J. W. H. M., & Bouzy, B. (2008). Progressive Strategies For Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, *4*(03), 343–357.

Childs, B. E., Brodeur, J. H., & Kocsis, L. (2008). Transpositions and move groups in Monte Carlo tree search. *2008 IEEE Symposium On Computational Intelligence and Games*, 389–395.

Chu, W., Li, L., Reyzin, L., & Schapire, R. E. (2011). Contextual bandits with linear payoff functions. In *International Conference on Artificial Intelligence and Statistics*, pp. 208–214.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, CG'06, pp. 72–83, Berlin, Heidelberg. Springer-Verlag.

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

Cowling, P. I., Ward, C. D., & Powley, E. J. (2012). 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.

Daswani, M., Sunehag, P., & Hutter, M. (2014). Feature Reinforcement Learning : State of the Art. In *AAAI-14 Workshop*, pp. 2–5.

Downey, C., & Sanner, S. (2010). Temporal difference bayesian model averaging: A bayesian perspective on adapting lambda. *International Conference on Machine Learning (ICML)*.

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

Enzenberger, M., Muller, M., Arneson, B., & Segal, R. (2010). 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.

Fairbank, M., & Alonso, E. (2012). Value-gradient learning. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pp. 1–8. IEEE.

Feldman, Z., & Domshlak, C. (2014). Simple Regret Optimization in Online Planning for Markov Decision Processes. *Journal of Artificial Intelligence Research*, 165–205.

Feldman, Z., & Domshlak, C. (2012). Online Planning in MDPs Rationality and Optimization..

Feldman, Z., & Domshlak, C. (2014a). Monte-Carlo Tree Search : To MC or to DP ?. In *ECAI14*.

Feldman, Z., & Domshlak, C. (2014b). On MABs and Separation of Concerns in Monte-Carlo Planning for MDPs. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, pp. 120–127.

Finnsson, H., & Björnsson, Y. (2008). Simulation-based approach to general game playing. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, AAAI'08, pp. 259–264. AAAI Press.

Finnsson, H., & Bjornsson, Y. (2009). CADIA PLAYER : A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*(1), 4–15.

Finnsson, H., & Björnsson, Y. (2010). Learning Simulation Control in General Game-Playing Agents. *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pp. 954–959.

Gábor, Z., Kalmár, Z., & Szepesvári, C. (1998). Multi-criteria Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, Vol. 98, pp. 197–205.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pp. 273–280, New York, NY, USA. ACM.

Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, *175*(11), 1856–1875.

Gelly, S., & Wang, Y. (2006). Exploration exploitation in go: UCT for Monte-Carlo go. *Proceedings of Advances in Neural Information Processing Systems*.

Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., & Teytaud, O. (2012). The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, *55*(3), 106–113.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Research report RR-6062, INRIA.

Genesereth, M., Love, N., & Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI Magazine*, *26*(2), 62–72.

Geramifard, A., Redding, J., How, J. P., Doshi, F., & Roy, N. (2011). Online Discovery of Feature Dependencies. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 881–888.

Guez, A., Silver, D., & Dayan, P. (2013). Scalable and efficient bayes-adaptive reinforcement learning based on Monte-Carlo tree search. *Journal of Artificial Intelligence Research*, *48*, 841–883.

Hashimoto, J., Kishimoto, A., Yoshizoe, K., & Ikeda, K. (2012). Accelerated UCT and its application to two-player games. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7168 LNCS*, 1–12.

Helmbold, D., & Parker-Wood, A. (2009). All-Moves-As-First Heuristics in Monte-Carlo Go.. *IC-AI*, 605–610.

Hester, T., & Stone, P. (2010). Real Time Targeted Exploration in Large Domains. In *The Ninth International Conference on Development and Learning (ICDL)*.

Hester, T., & Stone, P. (2013). TEXPLORE: Real-time sample-efficient reinforcement learning for robots. *Machine Learning*, *90*(3), 385–429.

Ilhan, E., & Etaner-Uyar, A. . (2017). Monte Carlo Tree Search with Temporal-Difference Learning for General Video Game Playing. In *IEEE Conference on Computational Intelligence and Games*, New York, USA.

Kearns, M. J., & Singh, S. P. (2000). Bias-variance error bounds for temporal difference updates. *COLT*.

Kearns, M., Mansour, Y., & Ng, A. (2002). A Sparse Sampling Algorithm for Near-Optimal. *Machine learning*, 193–208.

Keller, P. W., Mannor, S., & Precup, D. (2006). Automatic basis function construction for approximate dynamic programming and reinforcement learning. *Proceedings of the 23rd international conference on Machine learning - ICML '06*, 449–456.

Keller, T., & Helmert, M. (2013). Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Twenty-Third International Conference on Automated Planning and Scheduling*, pp. 135–143.

Khandelwal, P., Liebman, E., Niekum, S., & Stone, P. (2016). On the Analysis of Complex Backup Strategies in Monte Carlo Tree Search. *International Conference on Machine Learning (ICML)*, *48*.

Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., & Spiliopoulou, M. (Eds.), *Proceedings of the Seventeenth European Conference on Machine Learning*, Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293, Berlin/Heidelberg, Germany. Springer.

Kocsis, L., Szepesvári, C., & Willemson, J. (2006). Improved monte-carlo search. Tech. rep. 1, University of Tartu, Estonia.

Konidaris, G., Niekum, S., & Thomas, P. (2011). Tdgamma: Re-evaluating complex backups in temporal difference learning. *NIPS*.

Kozelek, T. (2009). *Methods of MCTS and the game Arimaa*. Master's thesis, Charles University in Prague.

Lanctot, M., Saffidine, A., Veness, J., Archibald, C., & Winands, M. H. M. (2013). Monte Carlo *-Minimax Search. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), Beijing, China, August 3-9, 2013*, pp. 580–586.

Lanctot, M., Winands, M. H. M., Pepels, T., & Sturtevant, N. R. (2014). Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. *IEEE Conference on Computatonal Intelligence and Games, CIG*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Li, L., Littman, M., & Walsh, T. J. (2008). Knows What It Knows: A Framework For Self-Aware Learning. *Proceedings of the 25th International Conference on Machine Learning*, 568–575.

Lorentz, R. J. (2008). Amazons Discover Monte-Carlo. In Herik, H., Xu, X., Ma, Z., & Winands, M. (Eds.), *Proceedings of the 6th international conference on Computers and Games*, Vol. 5131 of *Lecture Notes in Computer Science*, pp. 13–24. Springer Berlin Heidelberg.

Mahmood, A. R., & Sutton, R. S. (2013). Representation search through generate and test.. In *AAAI Workshop: Learning Rich Representations from Low-Level Sensors*.

Menache, I., Mannor, S., & Shimkin, N. (2005). Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, *134*(1), 215–238.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Nijssen, J. A. M. (2013). *Monte-Carlo Tree Search for Multi-Player Games*.

Nijssen, J. A. M., & Winands, M. H. M. (2011). *Enhancements for Multi-Player Monte-Carlo Tree Search*, pp. 238–249. Springer Berlin Heidelberg, Berlin, Heidelberg.

Osaki, Y., Shibahara, K., Tajima, Y., & Kotani, Y. (2008). An Othello evaluation function based on Temporal Difference Learning using probability of winning. *2008 IEEE Symposium On Computational Intelligence and Games*, 205–211.

Osband, I., Blundell, C., Pritzel, A., & Van Roy, B. (2016). Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems*, pp. 4026–4034.

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven Exploration by Self-supervised Prediction. In Precup, D., & Teh, Y. W. (Eds.), *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70 of *Proceedings of Machine Learning Research*, pp. 2778–2787, International Convention Centre, Sydney, Australia. PMLR.

Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couetoux, A., Lee, J., Lim, C.-u., & Thompson, T. (2015). The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*.

Ramanujan, R., & Selman, B. (2011). Trade-offs in sampling-based adversarial planning. *Proc. 21st Int. Conf. Automat. Plan. Sched., ...*, 202–209.

Ratitch, B., & Precup, D. (2004). Sparse distributed memories for on-line value-based reinforcement learning. In *European Conference on Machine Learning*, pp. 347–358. Springer.

Riedmiller, M. (2005). Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. In *ECML*, Vol. 3720, pp. 317–328. Springer.

Rimmel, A., & Teytaud, F. (2010). 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)*, Vol. 6024 LNCS, pp. 201–210.

Robles, D., Rohlfshagen, P., & Lucas, S. M. (2011). Learning non-random moves for playing Othello: Improving Monte Carlo Tree Search. *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, pp. 305–312.

Rummery, G., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems*. No. September.

Saffidine, A., Cazenave, T., & Mehat, J. (2010). UCD: Upper Confidence Bound for Rooted Directed Acyclic Graphs. In *International Conference on Technologies and Applications of Artificial Intelligence*, pp. 467–473. Ieee.

Salimans, T., Ho, J., Chen, X., & Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice..

Silver, D. (2009). *Reinforcement learning and simulation-based search in computer Go*. Ph.D. thesis, University of Alberta, Edmonton, Alta., Canada.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359.

Silver, D., Sutton, R. S., & Müller, M. (2007). Reinforcement learning of local shape in the game of Go. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pp. 1053–1058, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Silver, D., Sutton, R. S., & Müller, M. (2008). Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pp. 968–975, New York, NY, USA. ACM.

Silver, D., Sutton, R. S., & Müller, M. (2012). Temporal-difference search in computer Go. *Machine Learning*, *87*(2), 183–219.

Silver, D., van Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., Dulac-Arnold, G., Reichert, D. P., Rabinowitz, N., Barreto, A. A., & Degris, T. (2017). The Predictron: End-To-End

Learning and Planning. In Precup, D., & Teh, Y. W. (Eds.), *Proceedings of the 34th International Conference on Machine Learning (ICML) 2017, Sydney, NSW, Australia, 6-11 August 2017*, Vol. 70 of *Proceedings of Machine Learning Research*, pp. 3191–3199. PMLR.

Silver, D., & Veness, J. (2010). Monte-Carlo Planning in Large POMDPs. *Advances in neural information processing systems (NIPS)*, 1–9.

Singh, S. P., & Sutton, R. S. (1996). Reinforcement Learning with Replacing Elibibility Traces. *Machine Learning*, *22*, 123–158.

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

Stankiewicz, J. A. (2011). *Knowledge-Based Monte-Carlo Tree Search in Havannah*. Master's thesis, Maastricht University, The Netherlands.

Stanley, K. O., & Miikkulainen, R. (2002). Efficient evolution of neural network topologies. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, Vol. 2, pp. 1757–1762. IEEE.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. In *Machine Learning*, pp. 9–44. Kluwer Academic Publishers.

Sutton, R. S. (1990). Integrated Architectures for Learning , Planning , and Reacting Based on Approximating Dynamic Programming. In *ICML*, pp. 216–224.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press.

Sutton, R. S., & Barto, A. G. (2017). *Reinforcement Learning: An Introduction* (second edition). Unpublished work in progress.

Sutton, R. S., & Singh, S. P. (1994). On step-size and bias in temporal-difference learning..

Sutton, R. S., Whitehead, S. D., et al. (1993). Online learning with random representations. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 314–321.

Szita, I., & Szepesvári, C. (2011). Agnostic KWIK learning and efficient approximate reinforcement learning. *Journal of Machine Learning Research*, *19*, 739–772.

Tak, M. J. W., Winands, M. H. M., & Björnsson, Y. (2012). 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.

Tak, M., Winands, M., & Bjornsson, Y. (2014). Decaying Simulation Strategies.. *6*(4), 395–406.

Tesauro, G. (1994). TD-Gammon, a Self-teaching Backgammon Program, Achieves Master-level Play. *Neural Comput.*, *6*(2), 215–219.

Utgoff, P. E. (1989). Incremental Induction of Decision Trees. *Machine Learning*, *4*(2), 161–186.

Van Roy, B., Bertsekas, D. P., Lee, Y., & Tsitsiklis, J. N. (1997). A Neuro-Dynamic Programming Approach to Retailer Inventory Management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, Vol. 4, pp. 4052–4057. IEEE.

Van Seijen, H., Mahmood, A. R., Pilarski, P. M., Machado, M. C., & Sutton, R. S. (2016). True Online Temporal-Difference Learning. *Journal of Machine Learning Research*, *17*(September).

Veness, J., Lanctot, M., & Bowling, M. (2011a). Variance reduction in monte-carlo tree search. *Proceedings of Advances in Neural Information Processing Systems*, 1–9.

Veness, J., Ng, K. S., Hutter, M., Uther, W., & Silver, D. (2011b). A Monte-Carlo AIXI approximation. *Journal of Artificial Intelligence Research*, *40*(1), 95–142.

Veness, J., Silver, D., Uther, W., & Blair, A. (2009). Bootstrapping from game tree search. *Proceedings of Advances in Neural Information Processing Systems*, 1–9.

Vodopivec, T., & Šter, B. (2014). Enhancing Upper Confidence Bounds for Trees with Temporal Difference Values. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 286–293, Dortmund, Germany.

Walsh, T. J., Goschin, S., & Littman, M. L. (2010). Integrating Sample-based Planning and Model-based Reinforcement Learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Wang, W., & Sebag, M. (2013). Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search. *Machine Learning*, *92*(2-3), 403–429.

Watkins, C. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, Cambridge University, England.

White, M., & White, A. (2016). A greedy approach to adapting the trace parameter for temporal difference learning. *AAMAS*.

Whiteson, S., Taylor, M. E., Stone, P., et al. (2007). *Adaptive tile coding for value function approximation*. Computer Science Department, University of Texas at Austin.

Wiering, M., & Otterlo, M. V. (2012). *Reinforcement Learning: State-of-the-Art*, Vol. 12 of *Adaptation, Learning, and Optimization*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Xie, F., & Liu, Z. (2009). Backpropagation Modification in Monte-Carlo Game Tree Search. In *Proceedings of the Third International Symposium on Intelligent Information Technology Application - Volume 02*, IITA '09, pp. 125–128, Washington, DC, USA. IEEE Computer Society.

Yu, H., & Bertsekas, D. P. (2009). Basis function adaptation methods for cost approximation in mdp. In *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL'09. IEEE Symposium on*, pp. 74–81. IEEE.