University of Ljubljana
Faculty of Computer and Information Science

Aleksandra Deleva

TD learning in Monte Carlo tree search

masters thesis

the 2nd cycle masters study programme
computer and information science

Mentor: prof. dr. Branko Šter

Ljubljana, 2015

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aleksandra Deleva

Učenje s časovnimi razlikami pri drevesnem preiskovanju Monte Carlo

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Branko Šter

Ljubljana, 2015

Declaration of Masters Thesis authorship

I, the undersigned Aleksandra Deleva am the author of the Master Thesis entitled:

TD learning in Monte Carlo tree search

With my signature, I declare that:

- the submitted Thesis is my own unaided work under the supervision of prof. dr. Branko Šter
- all electronic forms of the Masters Thesis, title (Slovenian, English), abstract (Slovenian, English) and keywords (Slovenian, English) are identical to the printed form of the Masters Thesis,
- I agree with the publication of the electronic form of the Masters Thesis in the collection "Dela FRI".

In Ljubljana, 24. September 2015 Author's signature:

Acknowledgments

I would like to thank my mentor prof. dr. Branko Šter and his teaching assistant Tom Vodopivec, for their guidance and helpful suggestions throughout the process of writing this thesis. I would also like to thank my parents, my brother, my boyfriend and my close friends, for their unconditional love and support.

Aleksandra Deleva, 2015

"Evolve solutions; when you find a good one, don't stop."

— David Eagleman

Contents

List of Figures

Povzetek

Drevesno preiskovanje Monte Carlo(MCTS) je postalo znano po zaslugi uspehov v igri Go, pri kateri računalnik nikoli prej ni premagal človeškega mojstra. Nastalo je več različic algoritma. Ena izmed najbolj znanih različic je Zgornja meja zaupanja za drevesa oz. UCT (Kocsis in Szepesvari). Mnoge izboljšave osnovnega algoritma MCTS vključujejo uporabo domenskih hevristik, zaradi katerih pa algoritem izgubi na splošnosti.

Cilj tega magistrskega dela je bil raziskati, kako izboljšati algoritem MCTS brez ogrožanja njegove splošnosti. Paradigma spodbujevalnega učenja, ki se imenuje učenje s časovnimi razlikami, omogoča uporabo kombinacije dveh konceptov, dinamičnega programiranja in metod Monte Carlo. Moj cilj je bil vključiti prednosti učenja s časovnimi razlikami v algoritem MCTS. Na ta način se spremeni način posodabljanja vrednosti vozlišč glede na rezultat oz. nagrado.

Iz rezultatov je mogoče sklepati, da je kombinacija algoritma MCTS in učenja s časovnimi razlikami dobra ideja. Na novo razvit algoritem Sarsa-TS(λ) kaže na splošno izboljšanje uspešnosti igranja. Ker pa so igre, na katerih so bili izvedeni poskusi, zelo različne narave, se učinek algoritma na uspešnost posameznih iger lahko precej razlikuje.

Ključne besede

Drevesno preiskovanje Monte Carlo, Monte Carlo, Drevesno preiskovanje, Zgornja meja zaupanja za drevesa, Učenje s časovnimi razlikami, Umetna inteligenca

Abstract

Monte Carlo tree search (MCTS) has become well known with its success in the game of Go. A game in which a computer has never before won in a game against a human master player. There have been multiple variations of the algorithm since. One of the best known versions is the Upper Confidence Bounds for Trees (UCT) by Kocsis and Szepesvári. Many of the enhancements to the basic MCTS algorithm include usage of domain specific heuristics, which make the algorithm less general.

The goal of this thesis is to investigate how to improve the MCTS algorithm without compromising its generality. A Reinforcement Learning (RL) paradigm, called Temporal Difference (TD) learning, is a method that makes use of two concepts, Dynamic Programming (DP) and the Monte Carlo (MC) method. Our goal was to try to incorporate the advantages of the TD learning paradigm into the MCTS algorithm. The main idea was to change how rewards for each node are calculated, and when they are updated.

From the results of the experiments, one can conclude that a combination of the MCTS algorithm and the TD learning paradigm is after all a good idea. The newly developed Sarsa-TS(λ) shows a general improvement on the performance. Since the games we have done our experiments on are all very different, the effect the algorithm has on the performance varies.

Keywords

Monte Carlo tree search, Monte Carlo, Tree search, Upper Confidence Bounds for Trees, Temporal Difference learning, Reinforcement learning, Artificial

Intelligence

Chapter 1

Introduction

Monte Carlo tree search (MCTS) [1] over the years has become one of the well known algorithms used in game playing. This algorithm has shown its strength by playing games with very little or no domain knowledge. One of the first great successes was playing the game of Go. This is a game that has a very high branching factor, and a computer has never beaten a human until MCTS was applied to it.

The algorithm itself is organized in four steps: selection, expansion, simulation and back-up. The first two steps are called a tree policy and their goal is to select a node according to some policy or expand the tree. The simulation part is called a default policy, and is basically an execution of the game till its end in a stochastic manner. After all steps have finished in the predefined order, the back-up goes through all the nodes visited during the tree policy and updates their value. MCTS does updates at the end of the play-out. The main goal of the tree policy is to tackle the exploitation-exploration dilemma. This is dependent on the node selection method used.

Later on many versions and variations of the MCTS algorithm were developed. One of the most known and used versions is called Upper Confidence bounds for Trees (UCT) and was developed by Kocsis and Szepesvári [2] [3]. The node selection strategy used in the UCT works by forming bandit problems out of each node selection. In this manner the exploitation-exploration

dilemma is tackled. To improve the performance of the UCT, a lot of research was done using different kinds of heuristics. Unfortunately, the usage of domain-specific knowledge decreases the generality of the algorithm and in such form it works well only for the game it was improved for.

Temporal Difference (TD) learning [4] has been developed as a combination of two concepts related to Reinforcement learning (RL), Dynamic Programming (DP) and the Monte Carlo (MC) method. Similar to DP it updates its estimates before even having the final outcome, and similar to MC it does not require a model of the domain.

Our research is directed to finding a way to incorporate the advantages from the TD learning paradigm, in order to try to improve the performance of the MCTS algorithm. We started off with a version of the UCT algorithm, and changed its node selection strategy to ϵ-greedy. The ϵ helps increase or decrease the amount of exploration that will be done. Further on we continued by calculating the rewards on each step of the tree policy. Finally by updating the back-up step we incorporated TD into our algorithm. First thing to note is that each node is affected only by its descendants, and second is that the degree at which one node affects another is dependent on the distance between them. Lastly, the nodes are updated all through the play out as opposed to MCTS which has to wait till the end to do the updates.

1.1 Related Work

The idea to combine the advantages of MCTS methods with the TD learning is not a new one. A method called temporal difference search was developed [5]. In this method, TD learning is used in a combination with simulation based search methods, as is MCTS. The TD learning in this method is used to learn the weights of a heuristic function. This method was unsuccessful with MCTS, but on the other hand it performed well when combined with the alpha-beta search. We on the other hand do not lean on the domain specific features.

Another algorithm that was developed is called TDMC(λ) [6]. This method has shown to be better than the plain TD(λ) in the game of Othello. The algorithm uses MC to evaluate the probability to win in non terminal nodes. They haven't used MCTS but they have proposed it as further research possibilities. We on the other hand started with a version of MCTS and incorporated the TD learning. We do this by changing up the back-propagation step to use the TD in order to calculate the node values. These values will be further on used by the selection step. We use eligibility traces from TD that decay the values depending on the distance from the current node.

Some experimenting with weights has been tried and developed in [7] They made use of λ as a trace-decay parameter between 0 and 1, by decreasing the reward depending on the number of layers to the root, and updating all with the same value, which is different from the TD approach we use.

Our research was also influenced by the work done by Vodopivec and Šter [8] [9]. Their research is also using the Temporal Difference paradigm in the MCTS algorithm. They normalize the Upper Confidence Bounds for trees, and later combine it with TD learning.

1.2 Overview of the thesis

The general outline is as follows:

- Chapter 2: In this chapter we give the necessary background knowledge needed, in order to get to know the basic concepts how the algorithm works in its core. We also give an example of the UCT version of the MCTS algorithm, since it is the starting point to our further development.

- Chapter 3: In this chapter we introduce the TD learning paradigm. An overview of the TD(λ) and afterwards of the Sarsa(λ) algorithm are given. These algorithms are the second segment we used to incorporate into our research.

- Chapter 4: A brief introduction of the competitions we were part of during the creation of our algorithm. Our development and testing was done using the framework given by the competition site.
- Chapter 5: Here we introduce the algorithm developed with this thesis.
- Chapter 6: In this chapter we display the results from testing our algorithm. The results from the competitions we were a part of are also reviewed.
- Chapter 7: Lastly we share our opinion on the work done, and possible future enhancements and research.

Chapter 2

Monte Carlo Tree Search

2.1 Monte Carlo methods

Monte Carlo (MC) methods work by sampling randomly a certain domain. After the sampling has ended the samples gathered are evaluated and a result is returned. The number of samples greatly affects the result‘s accuracy. When applied to games, the MC is considering moves possible from the current position. Sampling is done by running simulations, where a simulation is a played out game. To return a result, it evaluates the gathered samples.

Not using heuristics nor requiring more information about the domain makes MC superior and more general compared to other search methods. The MC method also performs much better in games with a high branching factor, since this operation is very expensive for other tree search methods, while MC just increases its amount of sampling. On the other hand, not needing to know game specifics can be seen also as a disadvantage, since it would not see some special or tactical moves that are obvious for other tree search methods that use heuristics.

2.2 Monte Carlo tree search

Monte-Carlo Tree Search (MCTS) is an algorithm that can be applied to any finite length game. It uses the best-first strategy, and it approximates its results by using stochastic simulations. The MCTS algorithm creates a partial tree while executing simulations. With each simulation the tree becomes more complete and the estimates become more accurate. A simulation is a series of pseudo random moves from the player that is controlled by Artificial Intelligence (AI) and his opponent.

When building a tree the basic MCTS algorithm usually has some sort of budget, it may be a limitation in number of iterations, time, memory consumed or similar. When the budget has been met, the algorithm stops and returns the best move so far. The tree is built in a way that each node represents a state in the domain, and each link to the child nodes represents an action that can be taken to get to that node.

The algorithm is divided into four fazes that are to be executed in each iteration: called selection, expansion, simulation and back-propagation. These steps are done in this order until the predefined budget is reached [11].

2.2.1 Selection

In the selection step (Figure 2.1) we descend through the tree by applying a child selection policy. The child selection policy controls the balance between exploration and exploitation.

Exploration is looking into less promising moves in order to search for moves that may have a high value potential. On the other hand the exploitation is selection of moves that are known to be promising and have given good results in previous iterations. The selection step lasts until an expendable node is reached. An expandable node is a non terminal state that has unexpanded child nodes.

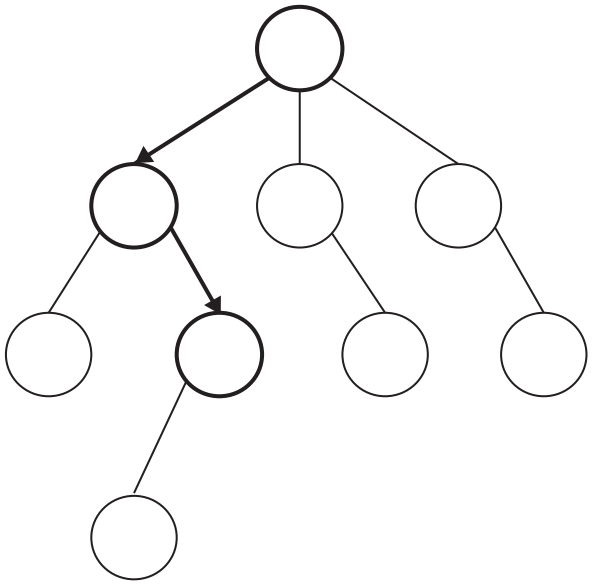

Figure 2.1: Selection step

2.2.2 Expansion

The expansion step (Figure 2.2) starts when the game has reached a state that does not exist in the tree built so far. One or more nodes can be added to the tree. The number of nodes to be added depends on the available actions and the strategy that is used.

2.2.3 Simulation

After nodes have been added to the tree, the simulation step (Figure 2.3) comes in to play to finish the game by selecting actions according to a so called default policy. The default way of doing the simulation step is taking random actions till the end is reached. Another way is using a policy that weights actions, this way it is possible to get better results. A drawback of using weights is time consumption.

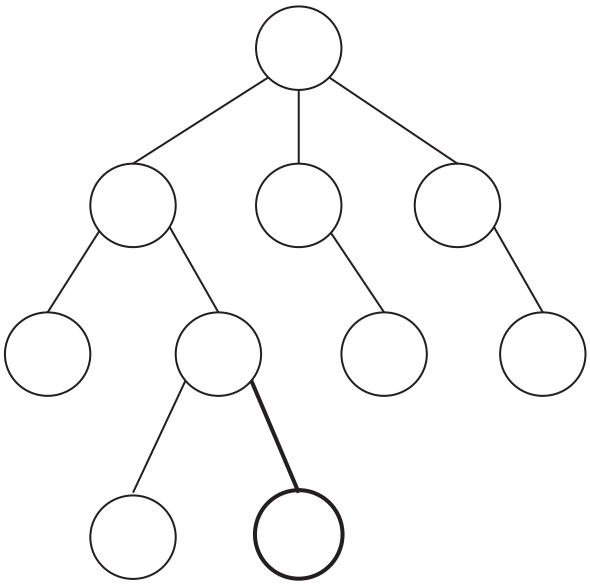

Figure 2.2: Expansion step

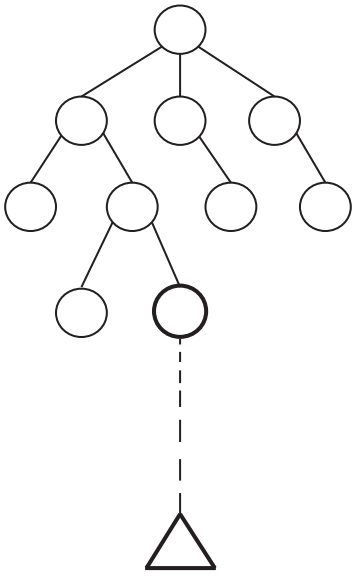

Figure 2.3: Simulation step

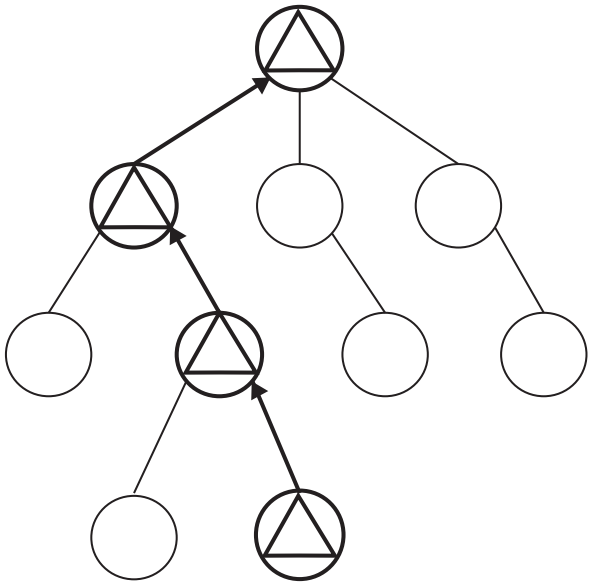

Figure 2.4: Back propagation step

2.2.4 Back-propagation

When the simulation step has finished and the end of the game is reached, the back-propagation (Figure 2.4) steps in. What this step does, is traverse through the steps that were taken to get to this outcome and update their visit count and value according to the outcome.

2.2.5 Move selection

When the computational budget is met, the search is terminated. An action that is a child of the root node is returned as the best move. There exist many ways to choose criteria by which a best child can be picked. Some winning action criteria are mentioned by Schadd [10]:

- Max child: Selection of highest reward root child
- Robust child: Selection of the most visited child
- Max-Robust child: Selection of child with both highest reward and visit

count. If there is no such child, the search continues until such a child node is found

- Secure child: Selection of the child that maximizes a lower confidence bound

2.2.6 Definition of policies

Two separate policies can be defined from the first three steps [1]:

- Tree policy: It represents the first two steps, selection and expansion. That is selecting and adding a leaf node to the tree
- Default policy: It represents the simulation to a terminal node, and producing an estimate.

The last step of back propagation does not use a policy since its only function is to update statistics of the tree.

2.3 Upper Confidence Bounds for trees

When searching the tree, promising nodes need to be also investigated. Without using heuristics, there is no other way but to invest a lot of computational resources in order to investigate them all. Since computational resources are often limited, dedicating only a certain amount of resources into investigating new unknown nodes so that new promising nodes would be found is a better combination.

This kind of problem is very similar to bandit problems. Choosing an action can be made based on rewards collected from past actions. So a balance needs to be made between choosing actions that may pay off in the long run (exploration) or the secure option that has already proven to be good (exploitation). This problem is solved by using a policy that chooses action, such that it would minimize regret and maximize reward.

This problem was an inspiration for Kocsis and Szepesvári to incorporate the UCB as a tree policy in MCTS, which resulted with the Upper confidence Bounds for Trees (UCT) algorithm [2][3]. UCB, being an efficient and simple way to stay within the best regret bounds, turned out to be a good way to tackle the exploration-exploitation dilemma.

Every time a node selection needs to be done, it can be seen as a multi-armed bandit [1]:

$$UCB = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} \tag{2.1}$$

The average reward $\overline{X}_j$ is within $[0, 1]$. C_p is a constant greater than 0, Kocsis and Szepesvári have chosen it to be $C_p = \frac{1}{\sqrt{2}}$. The number of times the parent has been visited is represented with n, and n_j is the number of times the child node j has been visited.

Exploration of less known choices is encouraged by the second part of the equation $2C_p\sqrt{\frac{2\ln n}{n_j}}$, while the choices that are known to be good and have been more visited are exploited by $\overline{X}_j$. As the number of visits of the j child node increases, its contribution decreases. The unexplored nodes have greater probability to be chosen. When the number of visits n_j equals to zero, $UCT = \infty$, which is the highest value.

After the selection of a new node with the UCT policy, the MCTS algorithm continues with the default policy to finish the game and get to an end state. Whether it is a win or loss, this value is back-propagated to all the nodes that were visited on the way, from the selected by the tree policy to the root of the tree. The nodes hold information of the total reward $Q(v)$ they have gotten so far, and the number of times they were visited $N(v)$.

The Algorithm 1 and Algorithm 2 are part I and part II of the pseudo code that shows the MCTS algorithm with the use of UCT [1]. Values that are kept in each node are total reward $Q(v)$, number of visits $N(v)$, incoming action $a(v)$ and state $s(v)$. The result from the algorithm is an action leading to the highest reward child node.

Algorithm 1 The MCTS algorithm (Part I)

```
1: procedure MCTS UCT(s_0)
2:     create root node v_0 with state s_0
3:     while within computational budget do
4:         v ← TreePolicy(v_0)
5:         Δ ← DefaultPolicy(s(v_l))
6:         BackUp(Δ, v_l)
7:     end while
8:     return a(BestChild(v_0, 0))
9: end procedure
10:
11: procedure TreePolicy(v)
12:     while within v is nonterminal do
13:         if v is not fully expanded then
14:             return Expand(v)
15:         else
16:             v ← BestChild(v, C_p)
17:         end if
18:     end while
19:     return v
20: end procedure
```

Algorithm 2 The MCTS algorithm (part II)

procedure EXPAND(v)
    choose $a \in$ untried actions $fromA(s(v))$
    add a new child $v\prime$ to v
    with $s(v\prime) = f(s(v), a)$
    and $a(v\prime) = a$
    return v
end procedure

procedure BESTCHILD(v)
    $s_i = \underset{v\prime \in \text{children of} v}{\text{argmax}} \frac{Q(v\prime)}{N(v\prime)} + c\sqrt{\frac{2\ln N(v)}{N(v\prime)}}$
end procedure

procedure DEFAULTPOLICY(s)
    while s is nonterminal **do**
        choose $a \in A(s)$ uniformly at random
        $s \leftarrow f(s, a)$
    end while
    return reward for state s
end procedure

procedure BACKUP(v, Δ)
    while s is not null **do**
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of v
    end while
    return reward for state s
end procedure

Chapter 3

Temporal Difference Learning

One of the great achievements in reinforcement learning is the Temporal Difference (TD) learning paradigm [4]. It is a combination of Dynamic programming (DP) and Monte Carlo (MC) methods. TD updates estimates even before having a final outcome, similar to DP, and learns without having a model of the domain, similar to MC.

TD methods collect experience following a certain policy, and use it for prediction. Both TD and MC work in a manner that they update the non-terminal states with estimates according to the policy used, with the difference of when they do the update. MC doesn't do any updates until the end of the episode, since it does not know the actual return which will be the target for the value of the state S_t at time t, $V(S_t)$.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{3.1}$$

where α is a constant step parameter, and G_t is the complete return.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T, \tag{3.2}$$

where T is the terminal time step. On the other hand, TD methods only have to wait for one time step to be able to form a target. It uses R_{t+1}, which is the observed reward, and the value of the state $V(S_{t+1})$ to make an update to the state value $V(S_t)$. TD is considered as a bootstrapping

method since it bases updates on existing estimates. TD(0) is the simplest of all TD methods. We shall explain the usage of the parameter later on. The update for the state S_t at time t, $V(S_t)$ for TD(0):

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)], \tag{3.3}$$

where the target for the update is $R_{t+1} + \gamma V(S_{t+1})$, instead of G_t as in MC. The parameter γ is a discount rate.

Algorithm 3 TD(0)

$V(s) = 0$
for each episode **do**
    Initialize S
    for every step of episode **do**
        $A \leftarrow A\prime$ from S using policy
        Take action A,
        observe reward R and next state $S\prime$
        $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S\prime) - V(S)]$
        $S \leftarrow S\prime$
        until S is terminal
    end for
end for

TD uses so-called sample backups. Full backups need all the possible successors, while sample backups only need a sample. Sample backups compute the backed-up value by looking ahead at the successor and using its value along with the current reward.

The need of MC methods to wait for the entire episode is something that needs to be considered, since some episodes can last for a really long time or are really slow; there are applications that do not even have episodes at all. This feature makes MC not be the best choice. On the other hand TD being implemented in a fully incremental and on-line fashion, requires one step waiting time. This feature makes TD very compatible with these types

of properties.

3.1 Sarsa

When looking at the prediction problem, just like MC, TD also needs to consider the exploitation exploration dilemma. This results in two approaches: on-policy and off-policy. So here we will explain the on-policy approach called Sarsa.

Algorithm 4 Sarsa

$Q(s, a)$ initialize arbitrarily
for each episode **do**
    Initialize S
    Choose A from s using policy
    for every step of episode **do**
        Take action A,
        observe reward R and next state $S\prime$
        Choose
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S\prime, A\prime) - Q(S, A)]$
        $S \leftarrow S\prime$
        until S is terminal
    end for
end for

For this approach, considering state-value pairs will change to considering state-action value pairs instead. So before, in order to learn the value of a state we were looking at states transitioning to other states. Now we would like to learn the state action-value, and we shall learn this by looking at the transition from one state-action into another state-action. Since this transitions whether is state-value or state-action value are similar, the TD(0) convergence still applies. In every transition from a non-terminal node, an

update is done:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (3.4)$$

$Q(S_{t+1}, A_{t+1})$ is defined as zero when it is terminal.

Sarsa just like TD is a step by step method that learns very quickly and can in time adjust its policies depending on what has been learned during an episode. MC in this case is inferior to Sarsa. If MC finds itself in a situation where the policy is not really working, it will not know till the very end of the episode, and can get stuck in a state.

3.2 n-step TD method

MC methods back up all the states with the observed rewards from the episode at the very end of it. TD takes the next reward and the value of the state from the next step, after what a backing up is done on multiple rewards. The number of rewards depends on the strategy used, it can be a one-step, two-step or n-step backup. These kind of methods that go up to n steps are called n-step TD methods. TD(0) is a one-step method. Since TD target back-up is the discounted value estimate of the following state and the first reward:

$$R_{t+1} + \gamma V(S_{t+1}), \quad (3.5)$$

which is a one-step approximate return. So generally the n-step return can be defined as:

$$G_t^{t+n}(V(S_{t+n})) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + .. + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}). \quad (3.6)$$

n-step backups are considered more of an increment than an update. In the estimated value $V(S_t)$, the increment $\Delta_t(S_t)$ is produced by the n-step

backup:

$$\Delta_t(S_t) = \alpha[G_t^{t+n}(V_t(S_{t+n})) - V_t(S_t)]. \tag{3.7}$$

When updates are made during an episode, right after an increment, we call this on-line updating:

$$V_{t+1}(s) = V_t(s) + \Delta_t(s), \qquad \forall s \in S, \tag{3.8}$$

while off-line updating would not do any changes to the estimates during the episode. It sums up all the gathered estimates at the end of an episode. The MC methods and one step TD are considered the extremes for the off-line and on-line methods.

3.3 TD(λ)

The λ in TD(λ) is indicating the use of a so called eligibility trace. The eligibility traces can be viewed in two manners. The forward view, otherwise called theoretical, and the backward view which can also be referred to as the mechanistic view. The forward view can not be implemented easily since in each step it uses information that will become known later on. In this section, we shall do an overview of the forward view first.

Toward any linear combination of n-step returns a backup can be done, as long as the weights sum up to 1. A complex backup is a sort of average of simpler component backups. So a way to understand the TD(λ) can be as a way to average the n-step backups. All n-step backups normalized by $1-\lambda$ and proportionally weighted by λ^{n-1}, where λ is a value between 0 and 1, are contained in the average. Normalization is done so the weights would sum to 1.

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{t+n}(V_t(S_{t+n})), \tag{3.9}$$

is a backup called the λ-return. With every step the weights become smaller by λ. When there exists a terminal state, the sum is finite:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_t^{t+n}(V_t(S_{t+n})) + \lambda^{T-t-1} G_t. \tag{3.10}$$

The target of the λ-return algorithm is a λ-return backup. The value of the state increment on each step

$$\Delta_t(S_t) = \alpha[G_t^\lambda - V_t(S_t)] \tag{3.11}$$

The complete return with the end at time T is:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-1} R_T \tag{3.12}$$

Individual n-step backups are

$$G_t^{t+1} = R_{t+1} + \gamma V_t(S_{t+1}) \tag{3.13}$$
$$G_t^{t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2}) \tag{3.14}$$
$$\dots \tag{3.15}$$
$$G_t^{t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^n V_t(S_{t+n}) \tag{3.16}$$

By following this we can derive how a state is updated.

$$G_t^\lambda = G_t^{t+1} + \sum_{n=2}^{T-t-1} \lambda^{n-1}(G_t^{t+n} - G_t^{t+n-1}) - \lambda^{T-t-1} G_t^{T-1} + \lambda^{T-t-1} G_t.$$

Here we introduce the so called TD-errors:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \tag{3.17}$$
$$\delta_{t+1} = R_{t+2} + \gamma V_t(S_{t+2}) - V_t(S_{t+1}) \tag{3.18}$$
$$\dots \tag{3.19}$$
$$\delta_{t+m} = R_{t+m+1} + \gamma V_t(S_{t+m+1}) - V_t(S_{t+m}) \tag{3.20}$$

It can be shown that

$$G_t^{t+n} - G_t^{t+n-1} = \gamma^{n-1}\delta_{t+n-1} \tag{3.21}$$

$$G_t - G_t^{T-t-1} = \gamma^{T-t-1}\delta_{T-1} \tag{3.22}$$

We also assume that $V_t(S_T) = 0$. The update rule may be written using TD-errors:

$$\Delta_t(S_t) = \alpha[G_t^{\lambda} - V_t(S_t)]$$

$$\alpha[G_t^{t+1} \sum_{n=2}^{T-t-1} (\lambda\gamma)^{n-1} + \delta_{t+n-1} + \gamma^{T-t-1}\delta_{T-1} - V_t(S_t)]$$

The first TD-error is

$$G_t^{t+1} - V_t(S_t) = \delta_t \tag{3.23}$$

The update is then:

$$\Delta_t(S_t) = \alpha[\sum_{n=1}^{T-t}(\lambda\gamma)^{n-1}\delta_{t+n-1}]$$

$$\alpha[(\lambda\gamma)^0\delta_t + (\lambda\gamma)^1\delta_{t+1} + \cdots + (\lambda\gamma)^{T-t-1}\delta_{T-1}]$$

The λ-return for $\lambda = 0$ is the previously mentioned TD(0). For $\lambda = 1$ and $\gamma = 1$, on the other hand, it is same as the previously mentioned MC. This property can be seen better if we separate the formula.

Now that we have seen the how the forward view works, we shall do an overview of the backward view of TD(λ). We denote eligibility trace for non-visited states $E_t(s) \in R^+$ as

$$E_t(s) = \lambda\gamma E_{t-1}(s), \qquad \forall s \in S, s \neq S_t \tag{3.24}$$

where s is the state, λ is called the trace-decay parameter and γ is the discount rate. The trace for visited states, S_t, is also decayed by $\lambda\gamma$, but it is also incremented by 1:

$$E_t(S_t) = \lambda\gamma E_{t-1}(S_t) + 1 \tag{3.25}$$

Each time a state is visited, the trace accumulates, but then it decays as less visits come along. In a way, $\lambda\gamma$ shows how long ago was a state visited.

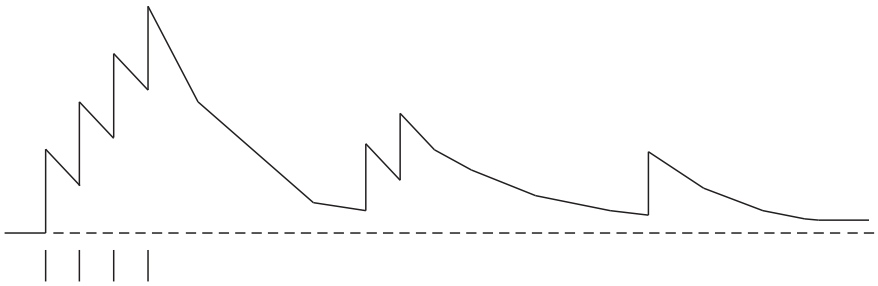

Figure 3.1: Accumulation of eligibility traces

The TD error for state value prediction is

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \tag{3.26}$$

This error provokes recent states to be updated

$$\Delta V_t(s_t) = \alpha\delta_t E_t(s), \qquad \forall s \in S \tag{3.27}$$

The mechanistic view is looking backwards. So we are assigning the TD error to the previous states.

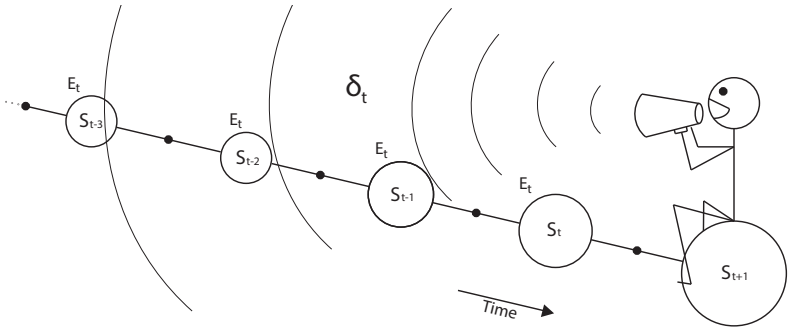

Figure 3.2: The backward view

Algorithm 5 TD(λ)

$V(s) \leftarrow 0$
for each episode **do**
  Initialize $E(s) = 0, \forall s \in S$
  Initialize S
  for every step of episode **do**
    $A \leftarrow$action given by policy π for S
    Take action A,
    observe R reward and $S\prime$ state
    $\delta \leftarrow R + \gamma V(S\prime) - V(S)$
    $E(S) \leftarrow E(S) + 1$
    for all $s \in S$ **do**
      $V(s) \leftarrow V(s) + \alpha\delta E(s)$
      $E(s) \leftarrow \lambda\gamma E(s)$
    end for
    $S \leftarrow S\prime$
    until S is terminal
  end for
end for

As we can see from Figure 3.2, the states are not updated equally. De-

pending on how far they are from the current state, the further they are the smaller the impact from the current state is on them. This effect is controlled by the λ parameter that is less than 1 but not zero. In case when λ and γ are 1, the impact of the state is equal for any distance of a state. In case when only λ is one, only γ has influence over the decay.

The case when λ and γ are 1 is also called TD(1), and can be considered a MC implementation if also the α parameter is same as in the MC algorithm. In case of TD(1) implementation that is on-line, the method learns about changes in policy immediately. This feature makes this implementation better than the previously explained MC.

3.3.1 Sarsa(λ)

The combination of eligibility traces with the previously mentioned algorithm Sarsa is called Sarsa(λ). Similarly so Sarsa, here the state value pairs from TD(λ) are substituted with state action pairs. So the trace denoted with $E(s, a)$ would be:

$$E_t(s, a) = \lambda\gamma E_{t-1}(s, a) + I_{sS_t} I_{aA_t}, \qquad \forall s \in S, a \in A \tag{3.28}$$

The I_{sS_t} and I_{aA_t} are 1 if $s = S_t$ or $a = A_t$ accordingly. They are called indicators of identity. Just like before, the $V_t(s)$ state value is substituted with the $Q_t(s, a)$

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t) \tag{3.29}$$

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta E_t(s, a), \qquad \forall s, a \tag{3.30}$$

We can refer to the Sarsa algorithm explained previously as the one step Sarsa. These two algorithms use a policy with which they make approximations of $q(s, a)$, and update it along the way.

Algorithm 6 Sarsa(λ)

Initialize $Q(s,a) = 0, \forall s \in S, \forall a \in A$
for each episode **do**
    Initialize $E(s,a) = 0, \forall s \in S, \forall a \in A$
    Initialize S, A
    for every step of episode **do**
        Take action A,
        observe reward R and next state $S\prime$
        Choose $A\prime$ from $S\prime$ using policy
        $\delta \leftarrow R + \gamma V(S\prime, A\prime) - V(S, A)$
        $E(S,A) \leftarrow E(S,A) + 1$
        for all $s \in S, a \in A$ **do**
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
            $E(s,a) \leftarrow \lambda\gamma E(s,a)$
        end for
        $S \leftarrow S\prime, A \leftarrow A\prime$
        until S is terminal
    end for
end for

Chapter 4

The General Video Game Playing Competition

The General Video Game Playing Competition [12] is a popular competition for AI. Its focus is on having the agents play multiple games that are now known to it. The types of games played are 2D single player arcade games.

Figure 4.1: The GVG-AI competition site [12]

Such competitions give reliable benchmarks on how good the competitors agent is.

4.1 The Framework

4.1.1 Video Game Description Language (VGDL)

The description language is built from objects and their interaction in a 2D rectangular space. Objects are able to move according to specified behavior or according to physics. Through collision objects can also interact.

A game has a description of the level, that is a description of start location of objects in the 2D space, and a description of the game, which describes the interactions that are possible and the dynamic of the game. A game is terminated after a predefined number of steps.

The description of the game is consisted of [13]:

- Mapping of the levels: Translation of level description into objects for the initial state of the game
- Sprite set: Classes of objects used in a tree structure. Properties of these objects are the description how they are affected by physics, how they are visualized, and resources they have. The last property is dynamic since it describes the expendable resources like mana, ammunition etc.
- Interaction set: Describes what can happen when two objects collide.
- Termination set: Describes the ways a game can end.

4.1.2 The General Video Game AI (GVG-AI) Framework

The framework that is used in this competition is Java VGDL. It loads games that are described with the VGDL, and gives an interface with which

new controllers can be developed. These controllers can control the players actions.

The VGDL gives a model through which the controller can learn about the state of the game, but not about the game definition. This property is not provided, since its the job of the agent to get to know the game and find a way to win it.

The implementation is created in a way that the controllers get 40 milliseconds to return an action, if the time passes and a move is not returned, a NIL value is returned. Methods implemented receive a so called State observation object that carries information like [13]:

- Time step, score and state
- The actions the agent can play at a time step
- Observations list
- Observation grid
- History of agent events

4.2 The Competition

4.2.1 Game sets

The competition has 30 single player games each with 5 levels. The games are grouped in sets, 10 games each.

- Training set: The games n this set are known to the competitor
- Validation set: Is a set of the games from the previous competitions Test set. The games are not known to the competitor. When a competitor uploads his controller, he can try it out on the Validation and Training sets.

- Test set: This set is the set of games used for the final results in the competition. The controllers can not try their games on this set before the end of the competition.

The scoring is dependent on the game itself. There are generally three types of scoring:

- Binary: Where a result not equal to zero is only given when the game is won
- Incremental: Where score increments or decreases depending on events.
- Discontinuous: Is using the Incremental system, but there are events that bring increase in score that is much bigger than other more common events.

Game termination conditions are also defined in the framework. A more general one is Counters, which is when sprites are created or destroyed. Some games end when reaching a certain point in the level. The last termination condition is dependent on number of steps, which is set to 2000 for all games.

4.2.2 Evaluation

The competition ends with each controller executing the sets mentioned previously. So 500 games in total for the test set. Different statistics are collected for each 50 plays of each game and after summing and analyzing a score table declaring the winner is conducted

4.3 Overview of some of the games in the framework

Here we will give a brief overview of some of the games that are in the GVG-AI framework. As we mentioned earlier, the games are 2D single player arcade games.

4.3.1 Chase

The game Chase is a racing arcade game. The main character played by the player is a police officer, chasing criminals not to flee the country. The game play is a car chase. On the road there are other cars and obstacles that the player must avoid in order catch the criminal. The criminal must be reached a few times before he will actually stop. The game has a time limit in which the criminal should be arrested. There is a bar showing how slowe the player is to getting the criminal. Each time the player gets to the criminal the bar fills up a bit. The game score is rising throughout the game, but also score points are lost when obstacles are not avoided well.

4.3.2 The legend of Zelda

The Legend of Zelda is consisted of puzzles, exploration and battle. To win the game not much exploration is needed, though it helpful since the player is rewarded with items. These items can are further on helpful to use through the game. The player has a life meter which can be refilled by collecting certain items. The game is consisted of an overworld, an area where the character can communicate with other characters and gain items. Another area are the dungeons which are a labyrinth and usually have one greater item. To get to the item the character has to fight enemies or solve puzzles. The character has a map and a compass that help him find his way. The game does not have scoring throughout the game. The character can only collect items to help him get through the areas easier.

Chapter 5

Sarsa-TS(λ)

We decided to use the GVG-AI framework as a starting point. It is a framework that provides basic implementations of several algorithms, between which there is also an implementation of the MCTS algorithm. This framework by itself offers many games built in, so the algorithms can be tested, locally or by uploading it on the competition website. The amount of games that an algorithm can be tested on, helps to see how good the algorithm works on a more general level, not only specifically to one or few games.

We started off by downloading and setting up the GVG-AI framework. We settled for the implementation called OLMCTS as a start, which is an implementation of the previously described MCTS algorithm using UCT. For the selection step of the MCTS algorithm, we decided on the ϵ-greedy approach. The greedy selection method exploits the actions that are known to pay off the most with the current knowledge. The ϵ-greedy, on the other hand, is greedy most of the time. The ϵ parameter is the probability with which algorithm will explore other less known options.

The next step was incorporating the λ-return algorithm into MCTS. The MCTS algorithm for each node keeps a total reward, which is updated in the backup step. When the algorithm gets to a leaf node or the computational budget has been met, it updates all the nodes that were a part of the tree policy with the collected reward throughout the simulation. So the first step

we took towards λ-return, was calculating the reward collected on each step of the tree policy and saving it in the node. Next in the back-up step as in MCTS we update only the nodes that were part of the tree policy. This time the update on the total reward of a node is done with the rewards collected from the current node and all its descendants. Here we introduce λ as a trace-decay parameter. As in the previously explained λ-return algorithm, it is a value between $[0, 1]$. Each reward that is added to total reward of the node fades by λ depending on how far it is from the current node.

The last step was getting to incorporate Sarsa(λ) into the algorithm. We introduce γ as a discount factor. As in the Sarsa(λ) we compute the TD error for each node:

$$\delta_t = R_t + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t) \tag{5.1}$$

We compute a backup so that we compute the TD error of a node, and update all its ancestors with the TD error

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta E_t(s, a), \qquad \forall s, a, \tag{5.2}$$

where we take α to be $1/n_j$,where n_j is the number of visits of the node. And the eligibility trace $E_t(s, a)$ is

$$E_t(s, a) = \lambda\gamma E_{t-1}(s, a), \qquad \forall s \in S, s \neq S_t, \forall a \in A. \tag{5.3}$$

The pseudo code of the algorithm is presented below.

Algorithm 7 The Sarsa-TS(λ) algorithm (Part I)

```
1: procedure SarsaTSearch(s_0)
2:     create root node v_0 with state s_0
3:     while within computational budget do
4:         v ← TreePolicy(v_0)
5:         DefaultPolicy(s(v_l))
6:         BackUp(v_l)
7:     end while
8:     return a(BestChild(v_0, 0))
9: end procedure
10:
11: procedure TreePolicy(v)
12:     while within v is nonterminal do
13:         if v is not fully expanded then
14:             return Expand(v)
15:         else
16:             v ← ε − greedy(v)
17:         end if
18:     end while
19:     return v
20: end procedure
21: procedure Expand(v)
22:     choose a ∈ untried actions fromA(s(v))
23:     add a new child v' to v
24:     with s(v') = f(s(v), a)
25:     and a(v') = a
26:     calculate nodes step reward R(v)
27:     return v
28: end procedure
29:
```

Algorithm 8 The Sarsa-TS(λ) (part II)

procedure ϵ-GREEDY(v)
  if random number $< \epsilon$ **then**
    $s_i = random_{v\prime \in \text{children of v}} Q(v\prime)$
    return $Expand(v)$
  else
    $s_i = \underset{v\prime \in \text{children of} v}{\text{argmax}} \; Q(v\prime)$
  end if
  calculate nodes step reward $R(v)$
end procedure

procedure DEFAULTPOLICY(s)
  while s is nonterminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  end while
  calculate nodes step reward $R(v)$
end procedure

procedure BACKUP(v)
  initialize λ and γ to a value between $[0, 1]$
  $\delta = R_t + \gamma Q(v\prime) - Q(v)$
  while depth of $v >= v_0$ depth **do**
    $N(v) \leftarrow N(v) + 1$
    $\alpha \leftarrow 1/N(v)$
    $E(v) \leftarrow (\lambda\gamma)^{degree}$
    $Q(v) \leftarrow Q(v) + \alpha\delta E(v)$
    $v \leftarrow$ parent of v
    degree $\leftarrow$ degree+1
  end while
end procedure

procedure BESTCHILD(v)
  $s_i = \underset{v\prime \in \text{children of} v}{\text{argmax}} \; Q(v\prime)$
end procedure

Chapter 6

Experiments

To test the performance of the newly developed algorithm Sarsa-TS(λ), which is using ϵ-greedy as a node selection method, we compared it with two variants from the MCTS algorithm explained in Chapter 2, using UCT or ϵ-greedy as a selection method, and Sarsa-TS(λ) using UCT with no additional normalizations. Each graph displays values in percentage on y-axis and λ or γ spanning between 0 and 1 accordingly. The results when λ is changing, we take γ to be 1, and vice versa. For the MCTS algorithms, the same value is shown across as a straight line over the x axis, since they do not contain λ and γ as parameters.

Because of limited processing power at hand, we were able to compute tests over 20 games with 5 levels, and 100 repetitions each. This way, we got 1000 samples per game.The games the algorithms were tested on are part of the GVG-AI framework explained in Chapter 4. The games are arcade, 2D, single player games.

As computational budget games were given the average amount of iterations they need, for each game it is different. This property was acquired with additional tests run previous to the final tests that are displayed further on. Roll-out depth of the search tree is fixed to 10. The number of moves from each node is predetermined from the framework, different depending on the game. The ϵ in ϵ-greedy is taken to be 5%. The C_p for the algorithms us-

ing UCT is taken to be 0.2. The variance is calculated with 95% confidence. In the following sections we shall preview the tests made on all games, and two games will be reviewed separately in a more detailed manner. Lastly, we shall review the results of the on-line GVG-AI competition.

6.1 Experiments on a variety of games

In order to show how the algorithm works on a more general level, we chose 20 of the games offered by the GVG-AI framework to test on. We did 100 repetitions on 5 levels for each game. The complete list of games used for the experiment:

- Aliens
- Boulderdash
- Butterflies
- Chase
- Frogs
- Missile command
- Portals
- Sokoban
- Survive zombies
- Zelda
- Camel race
- Dig Dug
- Firestorms
- Infection
- Firecaster
- overload
- Pacman
- Sea quest
- Whack a mole
- Eggomania

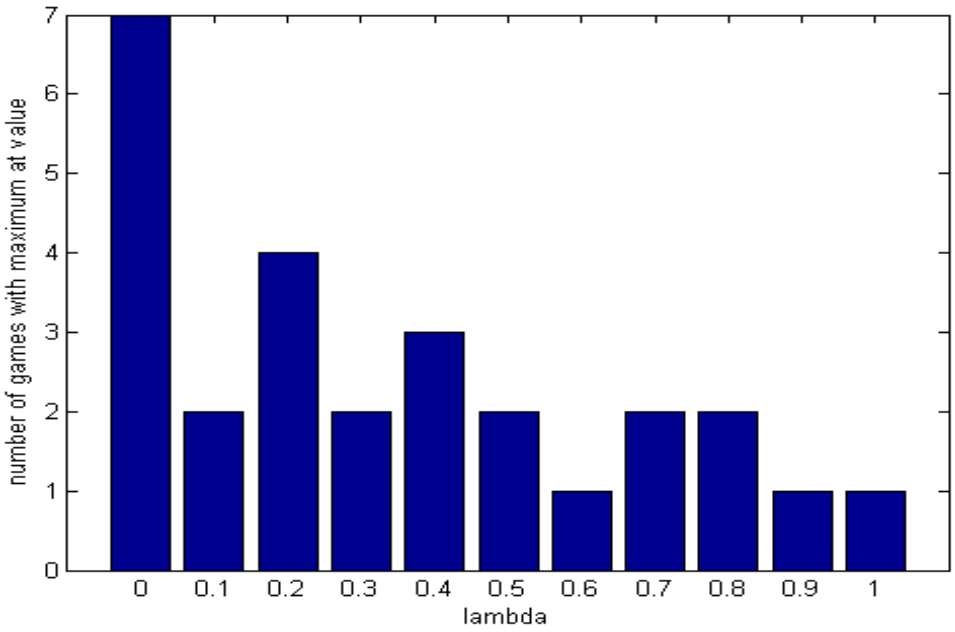

Figure 6.1: Number of games with maximum win percentage for values of λ between 0 and 1

To examine the results, we compare them on three different parameters:

- win percentage
- score for a game that was won
- number of steps taken in the game that was won

First of all, we shall take a look at Figure 6.1 and Figure 6.2. As all games are tested on different values for λ and γ, they reach a maximum win percentage when a certain value is used of λ or γ accordingly. The figures display how many games reached their maximum win percentage for each value of λ or γ. We can observe that many of the games reach it at 0. This is very dependent on the nature of the game. The way the scoring is done in the game, is it given all throughout the game or only at the end of the game, can have a

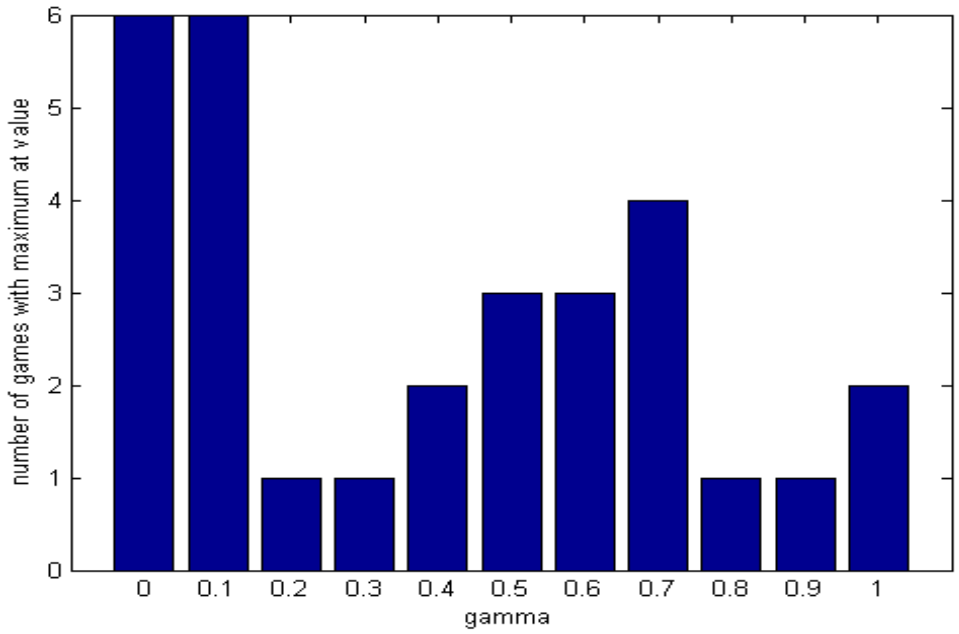

Figure 6.2: Number of games with maximum win percentage for each value of γ between 0 and 1

great effect on how much it benefits from looking deeper in the tree or just one step back.

When comparing the two histograms, we can observe that no matter if we are changing the λ or γ parameter most games have a maximum win at 0 at either one of the parameters. Further on, in Figure 6.1 we notice a more linear decrease, whereas in Figure 6.2 other than 0 most of the games have their maximum somewhere in the middle between 0 and 1.

Next we shall take a look at Figure 6.3 and Figure 6.4. These graphs represent the win percentage average for all games used, the first one shows results for λ between 0 and 1 and γ is set to 1. The latter has this reversed, λ is the one set to 1.

From the graphs we can see that we benefit just by switching from UCT to ϵ-greedy as a node selection method. Further on we can see that the

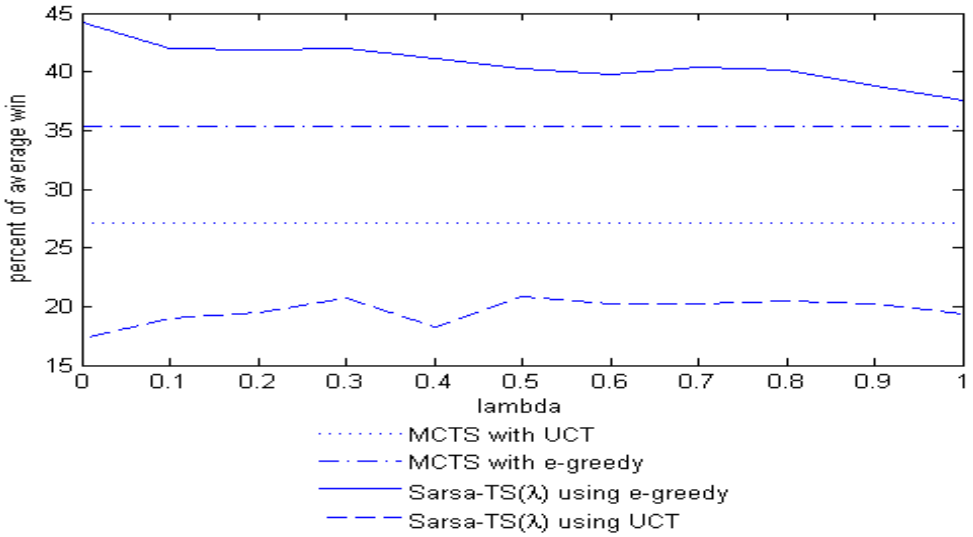

Figure 6.3: Percentage of games that were won for different values of λ between 0 and 1, for all 20 games

Sarsa-TS(λ) using UCT did not do very well. This is due to the algorithm not being normalized. Lastly we look at the curve of the Sarsa-TS(λ) using the ϵ-greedy method.

The first thing we can observe is that for $\lambda = 1$ the algorithm does not give equal results to the MCTS algorithm. While conceptually the algorithms are presumably the same, what gives Sarsa-TS(λ) an advantage is the way rewards are calculated, and updated.

As we mentioned earlier in Chapter 5, we update the nodes values regularly, not only at the end. The usage of eligibility trace λ and discount parameter γ decrease the influence of a child node on the current one, depending on the distance between them. When looking at Figure 6.3, for values of λ between 0 and 1, we can notice that from 0 it decreases linearly towards 1. While Figure 6.4 has its maximum between both the extremes. Notable is also the influence of the λ parameter, which resulted in a slightly higher maximum win percentage average than the γ parameter.

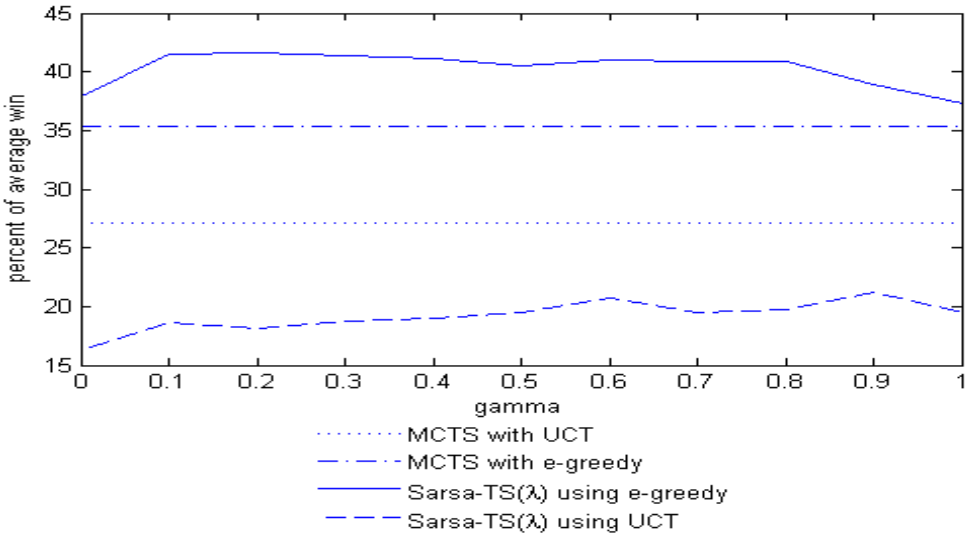

Figure 6.4: Percentage of games that were won for values different values for γ between 0 and 1, for all 20 games

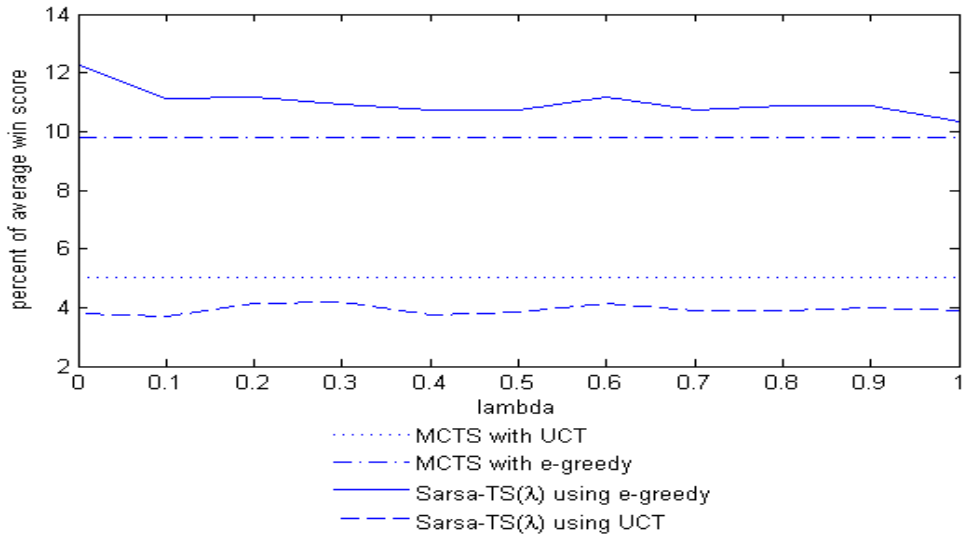

Figure 6.5: Percentage of score in games that were won for different values of λ between 0 and 1, for all 20 games

The second parameter we analyzed is the percentage of score in games that were won. The graphs displayed in Figure 6.5 and Figure 6.6 show the results. Similar conclusions come out of these two graphs. The score is in these games closely linked to the information if the game has been won. Then again, this is a property that is very dependent on the type of game, and its way of scoring.

The last parameter we analyzed is the steps taken when a game is won. The graphs displayed in Figure 6.7 and Figure 6.8 show the results. We can see from both the graphs than a clear correlation between between the previous results, and these can not be made. The steps taken in our case do not indicate on a better score or a win.

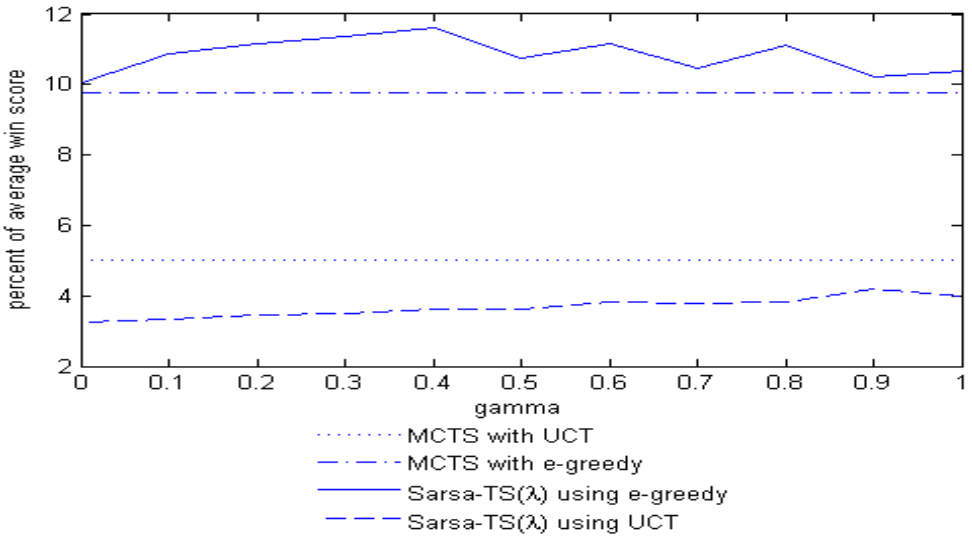

Figure 6.6: Percentage of score in games that were won for different values of γ between 0 and 1, for all 20 games

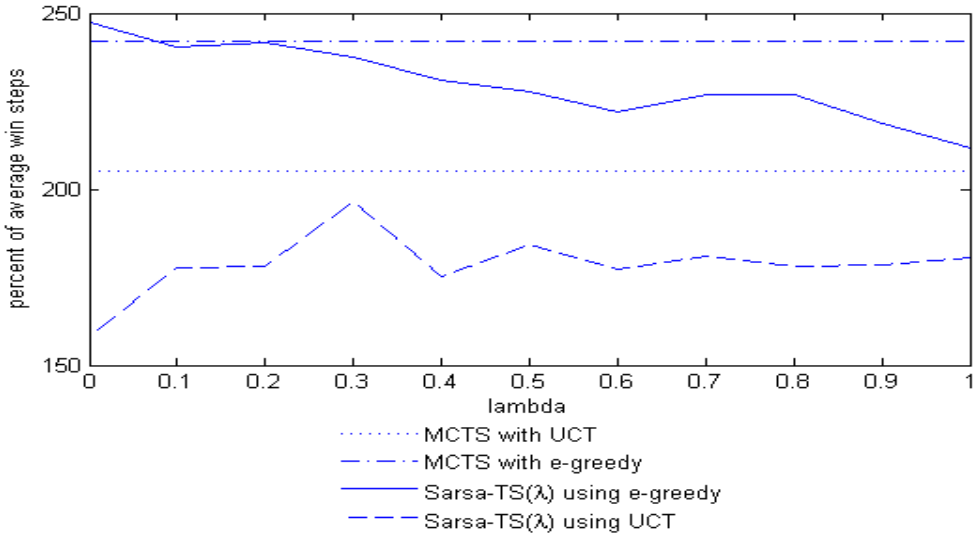

Figure 6.7: Percentage of steps taken in games that were won for different values of λ between 0 and 1, for all 20 games

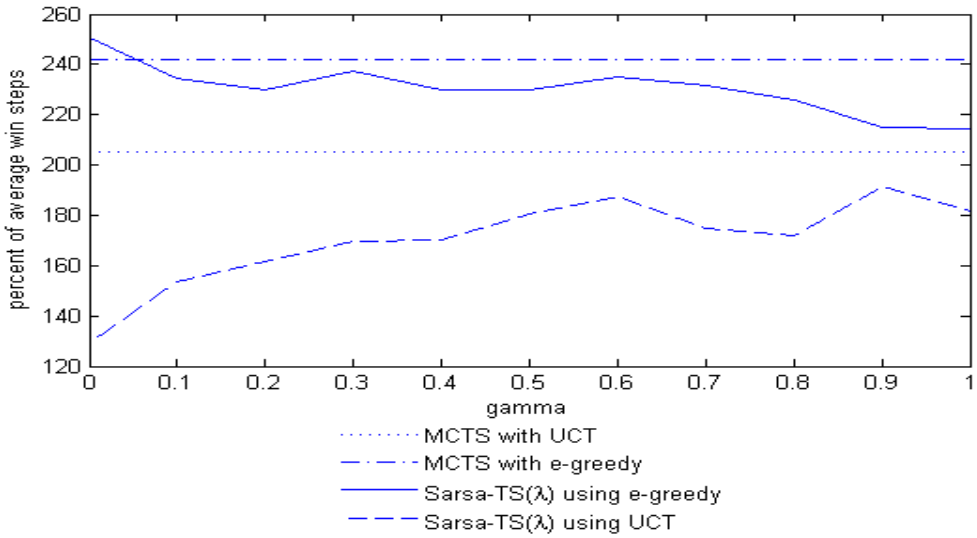

Figure 6.8: Percentage of steps taken in games that were won for γ between 0 and 1, for all 20 games

6.2 Case study: Chase

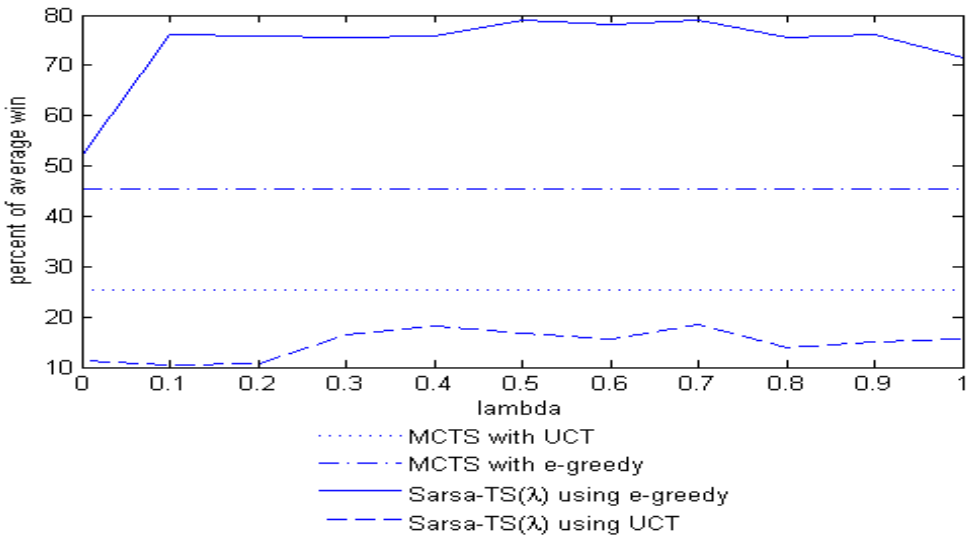

Figure 6.9: Percentage of games that were won for values different values for λ between 0 and 1, for the game Chase

The game of Chase is a single player arcade game. It is from beginning to end a car chase, with the goal of the main character, a police officer, to catch the bad guy, a criminal trying to flee. The games score constantly changes. Throughout the game it rises, but the player also can get penalties each time he makes a mistake in avoiding obstacles. Negative scoring does give information when the player has done something wrong, so in this case by looking back at steps taken, we can identify these moments and score the appropriate nodes with a smaller value. The experiments were done over 100 repetitions on 5 levels of the game.

If we take a look at Figure 6.9 or Figure 6.10, both show that Sarsa-TS(λ) gives better results. A thing to note is that the best values are somewhere between both the extremes 0 and 1, in both cases. So looking back only one step is too little, but also looking at the whole way back is also not a

good idea. One of the reasons for this is that looking too far back is time consuming. As we observed earlier, when λ is a value smaller than one and γ is one, the win percentage average is slightly larger.

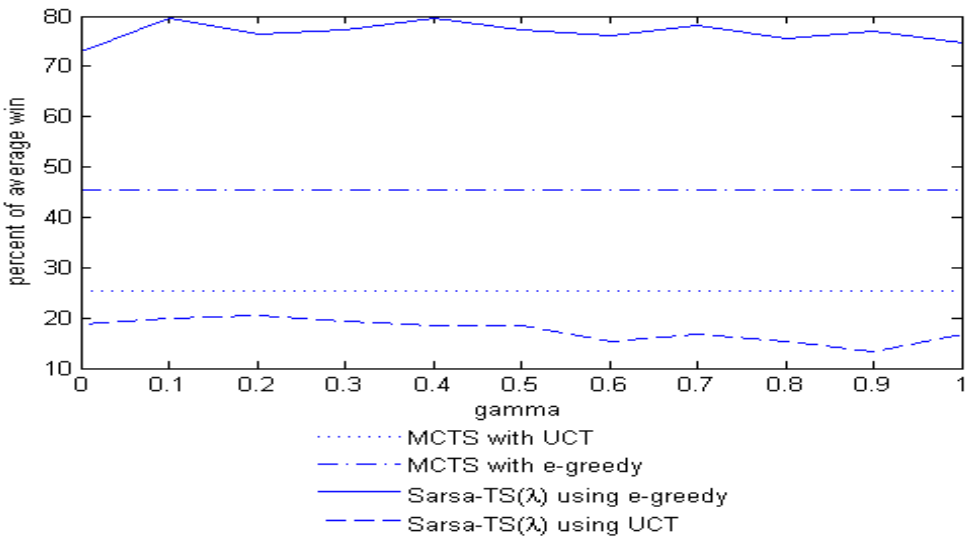

Figure 6.10: Percentage of games that were won for different values of γ between 0 and 1, for the game of Chase

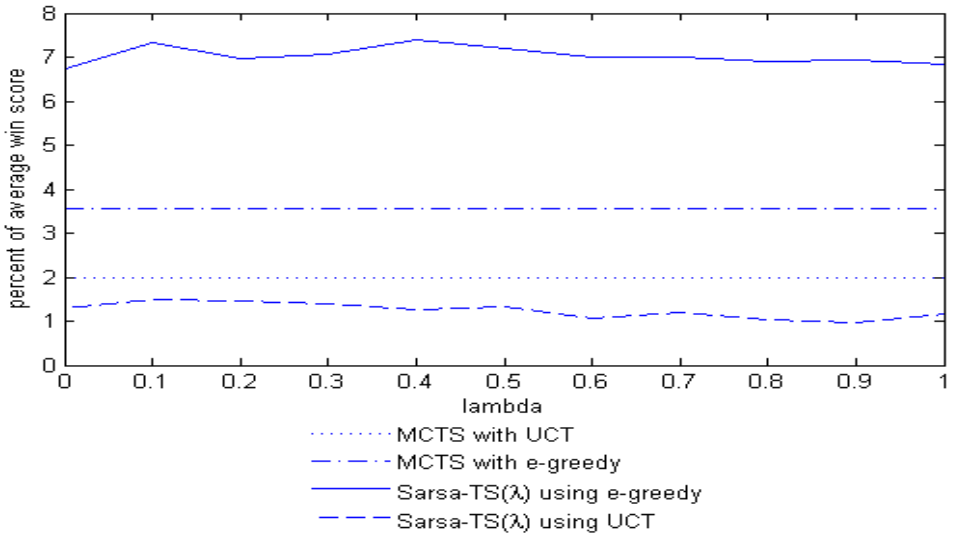

Figure 6.11: Percentage of score in games that were won for different values of λ between 0 and 1, for the game of Chase

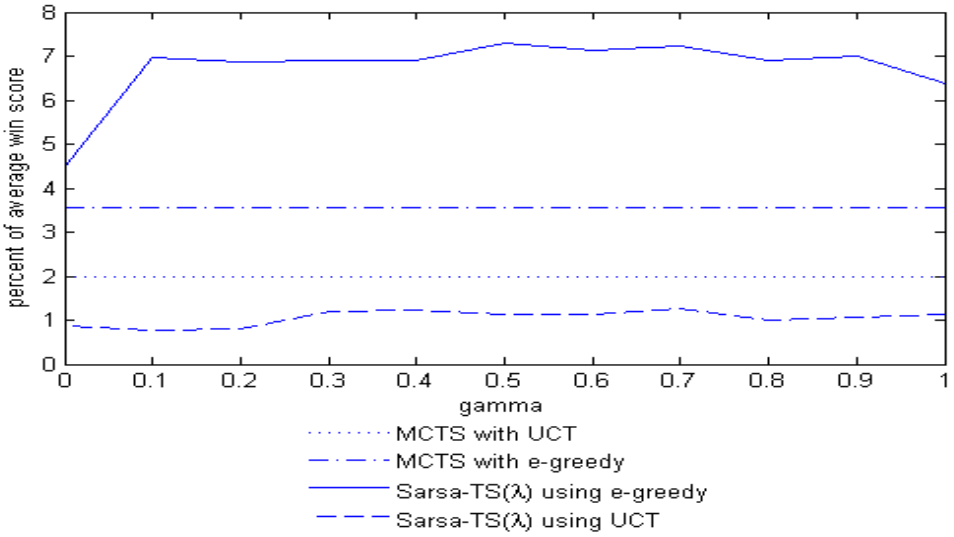

Figure 6.12: Percentage of score in games that were won for different values of γ between 0 and 1, for the game of Chase

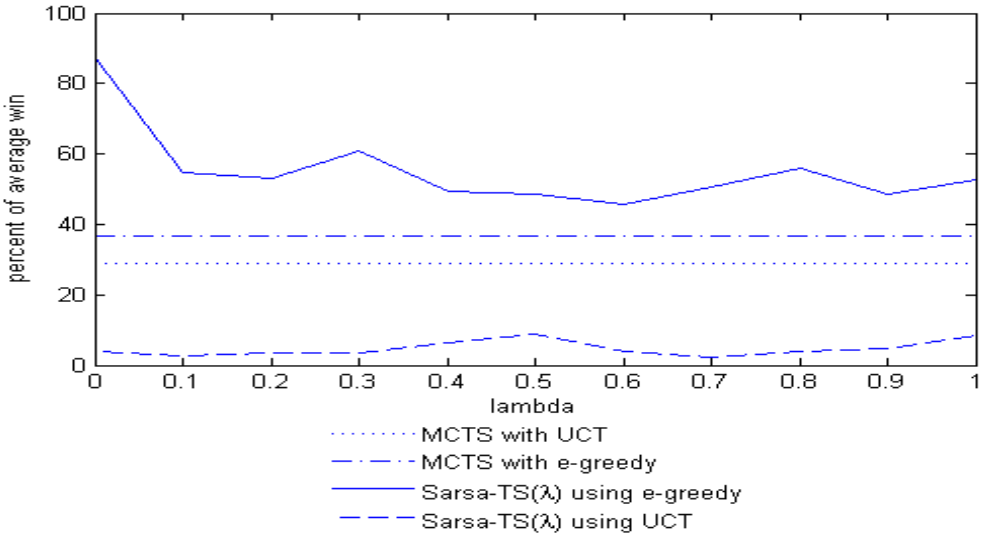

Figure 6.13: Percentage of games that were won for different values of λ between 0 and 1, for the game Zelda

6.3 Case study: The legend of Zelda

The game Legend of Zelda is a single player game filled with puzzles and action. The player collects items throughout the game, that help him get through it easier. There isn't any scoring other than at the end of the game when the player either wins or loses. Because the scores in the game are trivially small, there isn't much use of looking too far back at the scores, since they do not carry a lot of information.The experiments were done over 100 repetitions on 5 levels of the game.

If we take a look at Figure 6.13 or Figure 6.14, we can immediately notice that the best results are gotten when looking only one step back. In this game specifically looking further back can do more damage than good. But looking at the previous step does show itself as a good strategy.

As we have already noticed before, using λ that is smaller than one and γ as one, we were able to get slightly better results. Also the win score

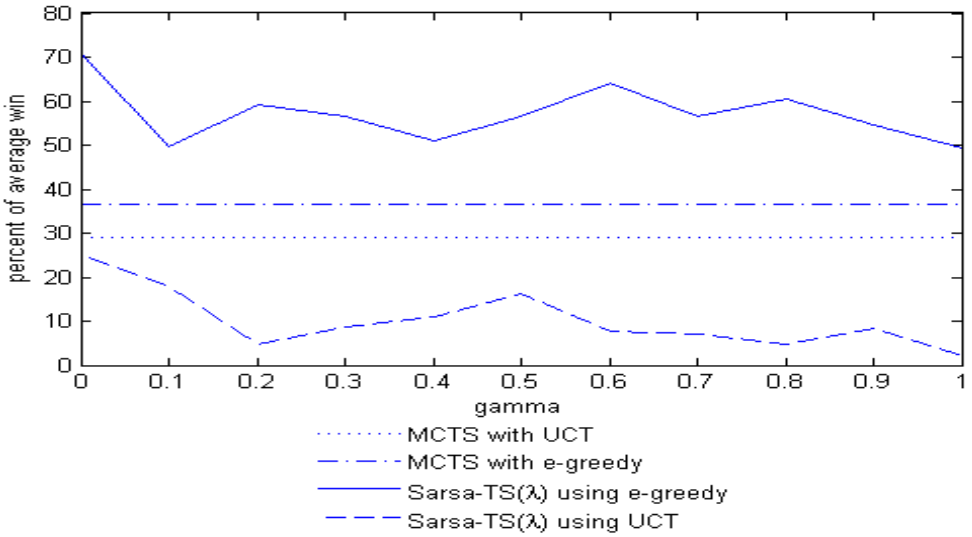

Figure 6.14: Percentage of games that were won for different values of γ between 0 and 1, for the game Zelda

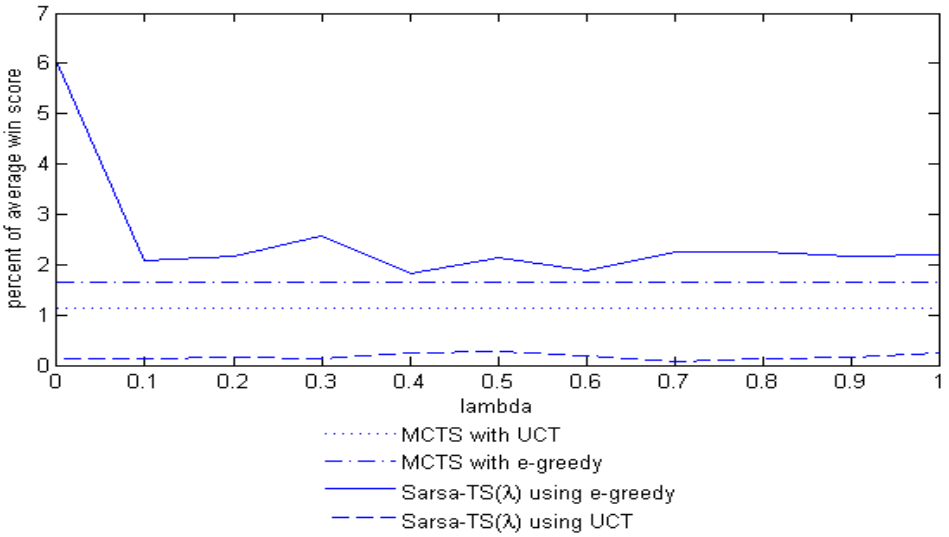

Figure 6.15: Percentage of score in games that were won for different values of λ between 0 and 1, for the game Zelda

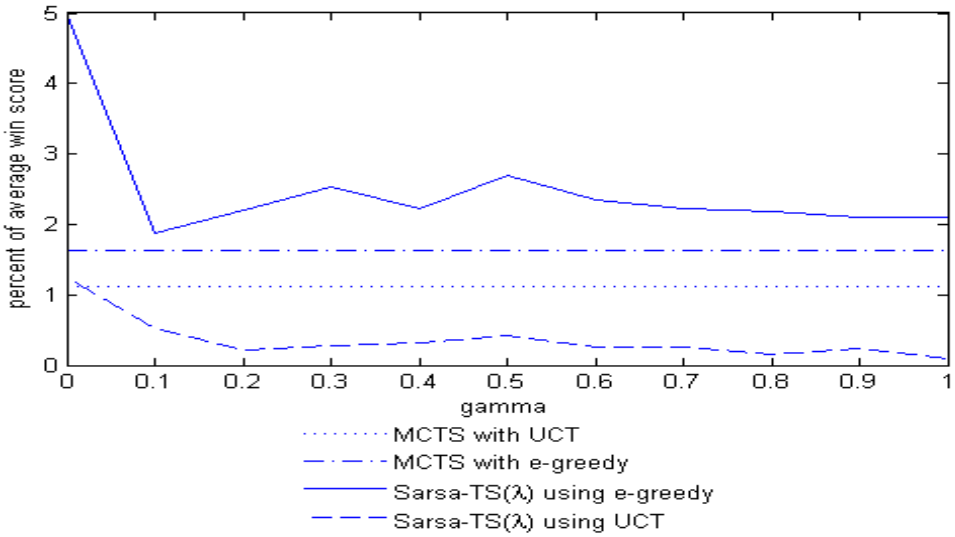

Figure 6.16: Percentage of score in games that were won for different values of γ between 0 and 1, for the game Zelda

parameter does give a very similar output as the win percentage average. They are correlated to each other, as the win percentage average grows so does the win score. According to this, they are both valid choices for showing the performance of the algorithm.

6.4 Results from participation in the GVG-AI competition

The competition is organized in three legs, and at the end of the three legs results are summed up and the overall winner will be announced. We were a part of two different legs of the GVG-AI competition, the Genetic and Evolutionary Computation Conference (GECCO) [14], the IEEE Conference on Computational Intelligence and Games (CIG) [15] and the Computer Science and Electronic Engineering Conference (CEEC) [16].

For each competition, there are two sets of 10 games, which are at the time being unknown to the participants by name and content. There is a validation set and a test set, which give the final rankings. The test set from GECCO is the validation set for CIG, and a training set for the CEEC.

6.4.1 GECCO

For this competition we chose to submit the newly developed Sarsa-TS(λ) with λ set to 0.4 and γ set to 1. In the validation set we were ranked at the 15th place, scoring 27 points and winning 15 out of 50 games.

The complete list of games for the validation set is Roguelike, Surround, Catapults, Plants, Plaque attack, Jaws, Labyrinth, Boulder chase, Escape, Lemmings If we take a closer look at the results by game. Thee games where our algorithm worked well are Plaque attack, Jaws and Lemmings.

For the test set we were ranked at the 51 place with 100 games won out of 500. The games used in the test set are unknown at the time being of writing this thesis, so unfortunately we can not analyze the games.

Rank	Username	Country	Description	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total
15	delevaa [3870]	Slovenia	Description	0	0	0	0	10	2	0	0	0	15	**27**

Figure 6.17: Validation rankings for GECCO, scoring on games separately

6.4.2 CIG

For this competition we submitted the same code of our algorithm Sarsa-TS(λ) with λ set to 0.4 and γ set to 1. Here we were ranked at the 17th place winning 15 out of 50 games, scoring 25 points. Details of the games separately are unknown at the time of writing, so unfortunately we can not analyze the games in more detail.

For the test set we were ranked at the 52 place winning 119 out of 500 games.

Rank	Username	Country	Description	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total
17	delevaa [4385]	Slovenia	Description	0	15	0	0	0	8	0	0	0	0	23

Figure 6.18: Validation rankings for CIG, scoring on games separately

6.4.3 CEEC

For this competition we chose to set λ to 0 and γ set to 1, using the Sarsa-TS(λ) algorithm. At the time of writing the competition is still in progress and the current rating is 11th place, winning 13 out of 50 games and scoring 71 points. The games are also unknown.

So comparing the results from the CEEC competition and the CIG and GECCO, we got better scoring when using λ set to 0 instead of 0.4. This could be very dependent on the games used and how the scoring works.

Rank	Username	Country	Description	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total
11	delevaa	Slovenia	Description	1	1	6	25	6	6	2	8	4	12	71

Figure 6.19: Validation rankings for CEEC, scoring on games separately

Chapter 7

Conclusions and future research

The Monte Carlo tree search algorithm over the time has become one of the preferred choices for solving problems in many domains, not just games. The goal of our research was to try to enhance one of the versions of MCTS, more precisely the UCT algorithm. We started by changing the node selection method with the ϵ-greedy method. Later on we started analyzing the TD learning paradigm, and ended up incorporating the Sarsa(λ) algorithm into the UCT. This resulted in our Sarsa-TS(λ) algorithm. We incorporated the use of eligibility traces, λ as a trace-decay parameter, and γ as a discount factor.

The experiments we did show a general improvement in the results, when compared to the algorithm with which we started our research. Tests were done so that either the eligibility trace or the discount parameter was set to one and the other one is ranging between 0 and 1. In both cases, the best results over all were never when both parameters are set to 1. Each game responds differently to our algorithm. In some cases the improvement is drastic, in other it is only a slight improvement or none at all. The majority of games we tested on generally show an improvement, but each game has a different value as its maximum. One of the things that influence this is the game itself. We weight the nodes with the scoring, and every game has a different scoring system. Some are scored throughout, some only at the end.

For further research there still remain open questions. The reasons why the best general value is at 0, what other factors influence this other than the types of games used for testing. If it is the games, can we determine the best values to use according to the way the scoring is done in them.

Our algorithm Sarsa-TS(λ) weights the nodes so that its descendants, depending on the distance, have different amount of influence. The values of the nodes are updated at each step taken, as opposed to updating at the end. Its performance encourages further research to combining the MCTS with the TD learning paradigm. Exploring these concepts may end up being a whole new class of enhancements for MCTS.

Bibliography

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P.I. Cowling, P. Rohlfshagen, S.Tavener, D. Perez, S. Samothrakisand S. Colton, "A Survey of Monte Carlo Tree Search Methods", IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, 2012, pp. 1-43.

[2] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in Euro. Conf. Mach. Learn. Berlin, Germany: Springer, 2006, pp. 282–293.

[3] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved Monte- Carlo Search," Univ. Tartu, Estonia, Tech. Rep. 1, 2006.

[4] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. The MIT Press, 1998.

[5] D. Silver, R. S. Sutton, and M. M̈uller,"Temporal-difference search in computer Go," Machine Learning, vol. 87, no. 2, pp. 183–219, Feb. 2012.

[6] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani, "An Othello evaluation function based on Temporal Difference Learning using probability of winning," 2008 IEEE Symposium On Computational Intelligence and Games, pp. 205–211, Dec. 2008

[7] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 4, pp. 241–257, Dec. 2012.

[8] Vodopivec, T.; Ster, B., "Enhancing upper confidence bounds for trees with temporal difference values," in Computational Intelligence and Games (CIG), 2014 IEEE Conference on , vol., no., pp.1-8, 26-29 Aug. 2014

[9] Vodopivec, T.; Ster, B., "Relation between Monte Carlo tree search and reinforcement learning", technical report, Faculty of Computer and Information Science, Slovenia, 2015

[10] F. C. Schadd, "Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis," M.S. thesis, Maastricht Univ., Netherlands, 2009.

[11] G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI," in Proc. Artif. Intell. Interact. Digital Entert. Conf., Stanford Univ., California, 2008, pp. 216–217. Board Game Thurn and Taxis," M.S. thesis, Maastricht Univ., Netherlands, 2009

[12] General Video Game AI Competition –2015, http://www.gvgai.net/

[13] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C. Lim, T. Thompson,"The 2014 General Game Playing Competition" in IEEE Transactions on Computational Intelligence and AI in Games(2015)

[14] Genetic and Evolutionary Computation Conference, http://www.sigevo.org/gecco-2015/

[15] 2015 IEEE Conference on Computational Intelligence and Games, http://cig2015.nctu.edu.tw/

[16] Computer Science and Electronic Engineering Conference, http://ceec.uk/