University of Alberta

TEMPORAL ABSTRACTION IN MONTE CARLO TREE SEARCH

by

Mostafa Vafadost

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Abstract

Given nothing but the generative model of the environment, Monte Carlo Tree Search techniques have recently shown spectacular results on domains previously thought to be intractable. In this thesis we try to develop generic techniques for temporal abstraction inside MCTS that would allow the effective construction of medium/long term plans in arbitrary Atari 2600 games. We introduce two methods: 1- CTW-UCT, that uses CTW algorithm to extract information from expert trajectories in order to make use of this information in search. 2- Variable Time Scale (VTS), that finds the desired time scale for taking each action online. We found that, CTW-UCT did not achieve satisfactory results. The VTS algorithm, however, was shown to be a promising algorithm. Although VTS did not achieve the highest individual scores on any game, it performed close to the best on most of the games. In other words, compared to different UCT algorithms that make use of macro actions with repeated actions of pre-specified lengths, VTS achieved comparable results to the best score on most of the games. We conclude therefore that, for domains where we do not know the desired time scale, VTS can be a good option.

Acknowledgements

I would like to express my utmost gratitude to my supervisor, Professor Michael Bowling, for his invaluable guidance, great support and incredible patience. I would also like to thank my other supervisor Dr. Joel Veness, whose constructive feedbacks and suggestions directed me in this research. I cannot imagine this project done without their priceless assistance and excellent knowledge.

I would also like to thank Mr. Mohammad Mahdi Ajallooeian, Mr. Amir Malekzadeh and Mr. Hossein Shahbazi for their valuable help. Last but not least, I thank my family and friends who have always been there for me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Games are an important application area for Artificial Intelligence (AI). This is largely since many important properties of real-world problems also feature in games, with the advantage that, in many cases, they are much simpler, more abstract, and easier to understand. This facilitates research on fundamental issues by relieving the researcher from many engineering and fine-tuning concerns. Thus, they are easier domains for analyzing and evaluating algorithms, which makes them well suited for building and improving AI methods. Moreover, the commercial applicability of such techniques is a good motivation for AI research as well.

Much AI research has been done on games during the past decades: including chess [1][2], checkers [3][4], Go [5], backgammon[6][7].

A common approach is to use a combination of search and heuristic techniques to decide on each action. In many cases, a game can be well modelled by a search tree. Nodes are different states and edges are different choices of actions that can be taken from the corresponding state. Then, the problem of playing a game is reduced to finding a path in the tree in which the agent achieves the highest reward. Looking at all nodes in the search tree to find the optimal node, is a trivial method if the search space can be stored in memory. Nevertheless, the search space of many problems, especially in the real-world, is usually too large to be stored or traversed, hence building the complete search tree is almost always impossible. Therefore, there is a need for pruning the search tree to avoid unnecessary searching of some parts of the tree that are less likely, or even impossible, to contain the optimum. Searching in smaller trees, as a result, makes algorithms more effective. Typically, pruning the search tree requires estimating an evaluation function for any state. These estimation functions have been explored by researchers and are known as heuristic functions in the literature [8][9].

Heuristic functions can result in impressive improvements in different games and problems [10][2]. However, finding good heuristic functions can be time-consuming. Although there are some general heuristic design methods [11][12][13], they usually require considerable domain-specific knowledge to apply. As such finding good heuristics can be difficult in some applications. Using sample-based techniques, such as Monte-Carlo methods, are an attractive alternative to finding good static evaluation functions. In particular, Abramson demonstrated that Monte-Carlo simulations can be utilized in evaluating the value of a state [14]. Later, Brugmann [15], Ginsberg [16] and Maven [17] applied Monte-Carlo methods to various games.

A Monte-Carlo simulation evaluates a game played according to a random (or even guided) trajectory that starts from a node in the game state space and continues until the game finishes. The final accumulated reward that a Monte-Carlo simulation obtains can then be used as a replacement for the heuristic evaluation function. Letting more Monte-Carlo simulations run from a node allows this estimated value move closer to the value that we can actually expect to achieve from that node. Monte-Carlo search algorithms have led to considerable improvements in solving different games such as poker [18], and Go [19].

On domains with a high branching factor and large search space, it historically took a while before Monte Carlo simulation methods provided significant improvement [15][19]. The key breakthrough was the introduction of Monte Carlo Tree Search (MCTS): by Coulom [20], Kocsis et al. [21], Chaslot et al. [22]. MCTS is a best-first search algorithm that uses Monte Carlo simulations to evaluate the nodes of the tree. In MCTS, as the first step, a selection strategy drives the search to one of the leaves. The tree is then expanded from that leaf. When an unseen state is chosen as the new expansion point in the tree, a Monte Carlo simulation is used to find the estimated value of that state. Finally, the value obtained by the Monte Carlo simulation is back propagated to its ancestors. Consequently, the algorithm is able to decide wisely in the next action selections. The best action is chosen by looking at the root level when the search is finished.

Upper Confidence Bounds for Trees (UCT) is a MCTS algorithm proposed by Kocsis et al. [21], in which the selection strategy consists of solving multiple stochastic Multi-Arm Bandit (MAB) problems. A k-arm bandit is a gambling machine that has k arms. When an arm is pulled, a reward is sampled from an underlying distribution associated with that arm. The multi-arm bandit problem, requires accumulating the highest possible cumulative reward given some budget of pulls. Successful algorithms, in the long run need to focus most of their attention on the most promising arm, while still exploring sufficiently often to not be misled by the stochastic nature of the returns. In UCT, at each level of the tree, actions are considered as arms of a bandit problem, with the arm that we choose being selected according to a particular choice of MAB algorithm. The bandit algorithm used by UCT is known as UCB1. UCB1 [23] has some good properties such as having the optimal regret. UCT is also proven to converge to the optimal solution [21]. With UCT continuously having remarkable success in games such as Go [24], many researchers tried to further enhance and improve it by applying ideas such as Rapid Action Value Estimation [25], transposition [26], parameter tuning [27][28], using heuristic or domain specific knowledge [29][30], and also reducing the variance [31].

Despite all these effort, UCT, the same as other MCTS algorithms, still has some issues when the branching factor is big. Some progressive strategies, such as progressive widening [32], progressive biased and progressive unpruning [29] have been proposed to address this problem.

The problem of temporal abstraction has a long history in AI [33][34][35]. As of yet, little investigation of this issue has been done in the context of MCTS algorithms. A few works, such as the one by Naddaf [36] used constant-length macro-actions within different algorithms and tested them on different Atari 2600 games. Powley et al. [37] also used the same idea for improving their algorithm for a routing problem in the 2012 WCCI PTSP competition[38]. Accordingly, we know already that crude workarounds such as repeating actions for pre-specified durations can lead to significant performance improvements. What we lack, however, is something more principled. The goal of this thesis is therefore, to develop a generic technique for temporal abstraction inside MCTS that would allow the effective construction of medium/long term plans in arbitrary Atari 2600 games. An ideal solution would learn abstract/macro-actions online automatically, given nothing but sample trajectories.

Atari 2600 is a desirable domain for developing and testing generic AI algorithms, as it has some important properties. First, the games are simple and abstracted and yet at the same time they look more similar to real world problems compared to other game domains such as game of Go. Besides, the Atari 2600 has a large collection of games with distinct properties and attributes. Moreover, the Arcade Learning Environment (ALE) is an open source research software framework implemented on top of Stella, an Atari 2600 emulator. Although the ALE emulator is simple and relatively fast, two additional simple games were designed and implemented as early experimentation required very fast domains. These two games are very similar to Atari 2600 games, and are used as the preliminary games for the algorithms.

Two approaches are explored in this thesis: First, we have designed an algorithm based on UCT that adapts the length of the macro actions online. The motivation is the fact that, the number of times an optimal policy picks an action consecutively is not a constant number and it can vary in different situations. In the second approach, we try to extract information from expert trajectories to find the most promising macro action in each state for search. This is done by using the Context Tree Weighting (CTW) algorithm as the macro action generator of the algorithm.

This thesis is structured as follows: in chapter 2 the background is surveyed and the related work is discussed. Chapters 3 and 4 present the ideas and the algorithms implemented. In chapter 5, we talk about the experimental setup, evaluation method and the results on both the preliminary games and Atari 2600 games. Finally, chapter 6 contains the conclusion followed by suggestions for future work.

Chapter 2

Background and Related work [1]

2.1 AI in games

Games can be modeled as tree structured graphs. The root of the tree is the initial state of the game. Nodes are different states of the game that are connected to each other by edges. Each edge represents an action that can be taken to move from one state to the next. Usually, in games, only a few game states, named terminal states, can be evaluated completely. These are the leaves of the tree and they represent finished game states. For all other states there may be different possible ways a game can end. Thus, evaluating non-terminal states requires considering different sequences of actions that can transition the current game state into a terminal state. Obviously, the number of players plays an important role here: if there are two or more players involved, since the other players can play with different strategies, all possible scenarios should be considered. In this case, some nodes are added in order to play in the opponent's turn. For games that have non-deterministic elements such as rolling a dice, chance nodes are added to the tree. For deterministic single player games the situation is much more simple, as the only challenge in evaluating non-terminal states is finding the best possible sequence of actions that could be taken to reach a terminal state.

The category of single player games is a special case of games where there is no one else to defeat except for the puzzle maker. The player solves a puzzle or tries to accomplish a task or overcome a challenge. In multiplayer games, however, there are other opponents that can take different actions each aiming to increase their own utility. Compared to multi-player games, single player games are significantly simpler and less complex to analyze and evaluate. This is because of the fact that finding the best strategy, as opposed to multiplayer games, does not require consideration of the opponents' behaviour, and only the reactions

[1]Some parts of this chapter follow the flow of background section of Browne et al. survey on Monte Carlo Tree Search methods [39]

of the environment are needed to be dealt with. If the environment is deterministic and fully observable, a full tree search is a usual search algorithm that comes to mind. The whole tree is generated at the beginning of the search, and then by looking at all terminal nodes, the best one will be picked. The best strategy for the agent to play is the sequence of actions that changes the initial state to this best node.

In developing algorithms for a particular game, heuristic information and game-specific knowledge can improve the performance. However, a secondary goal of AI research on games is the hope that these techniques will scale to real-world problems. Thus, algorithms designed to reach this goal, should not be allowed to use problem specific tricks and problem-specific information and knowledge, unless they can be extracted methodologically, e.g. by providing some basic and easily-accessed information, such as expertly played sequences of actions.

One noteworthy attempt by the community at this goal is the General Game Playing competition. General Game Players (GGPs) are algorithms focused on solving a wider range of games rather than playing only one game at a human expert level. The purpose of GGPs therefore, is to build intelligent algorithms that can deal with real-world problems even when they face new challenges. These algorithms are designed to find good solutions to unseen problems. Thus they have to give up on using game-specific and heuristic knowledge and this will result in a lower level of performance. Some examples of MCTS algorithms as GGPs are CadiaPlayer [40], Ary [41] and Centurio [42].

In this thesis the focus is on playing deterministic single player games.

2.2 AI algorithms to play games

For playing games, many methods have been proposed in the AI literature. For example, in two player adversarial games, Minimax methods aim to maximize the agent's reward by minimizing the opponent's maximum achievable reward. It produces the search tree, where each level alternates between maximizing the agents reward and minimizing the opponent's reward. The $\alpha - \beta$ pruning algorithm was later proposed to prune this Minimax tree. The search can stop before reaching a terminal state, but this requires an estimation function for non-terminal states. Max^n [43] is the adapted version of Minimax to games with more than two players or to non-zero-sum games. Expectimax [44] is the stochastic version of Minimax, for non-deterministic environments. A stochastic environment is an environment where the outcome of taking an action is not deterministic and follows a stochastic distribution on successor states. Miximax[45] is for games with stochastic and imperfect information. *-minimax[45] search is an adaptation of $\alpha - \beta$ pruning algorithm for games that have probabilistic elements and events such as rolling dice.

2.3 Atari 2600

Atari 2600 is a video game console designed in 1977 that has 128 Bytes of RAM and 1.19 Mhz CPU. It has sold over 30 million units since its initial release. More than 500 original games were released over the course of 30 years, most of which were released in late 70s and early 80s. Many iconic games were also implemented for the Atari 2600, such as PacMan. These Atari 2600 games vary a lot: Freeway is an action game, Phoenix and Beamrider are shooters, and Ice hockey and Skiing are sport games. Figure 2.1 shows screenshots of four Atari 2600 games.

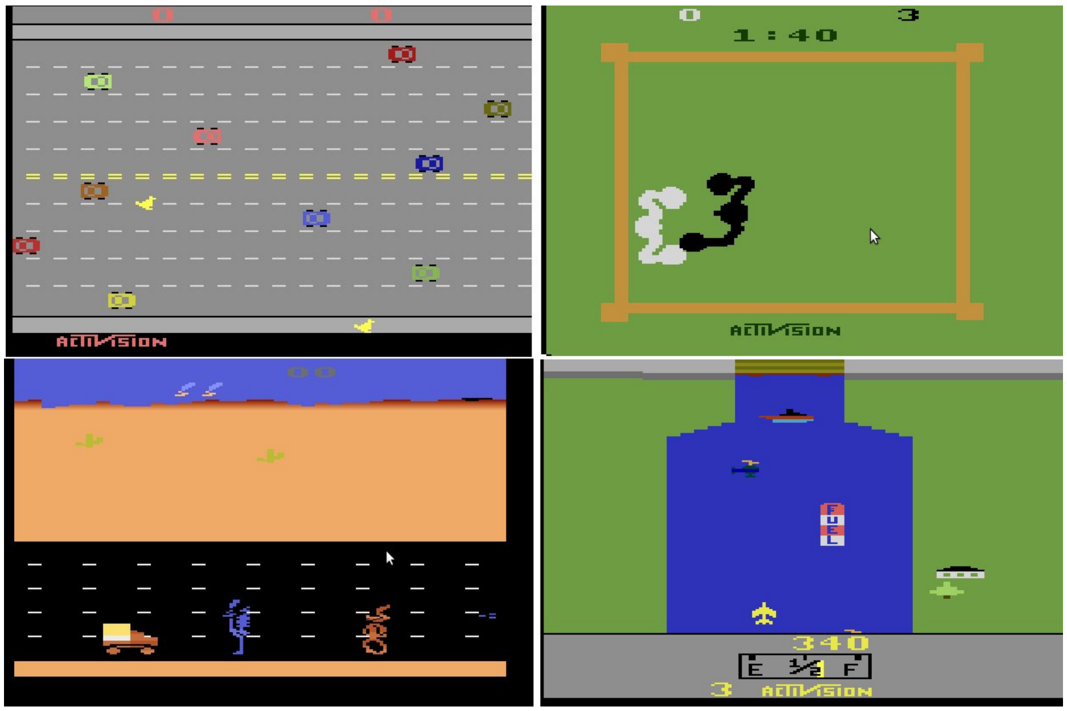

Figure 2.1: Screenshots of 4 Atari 2600 games. Atari 2600 has a varied and broad set of games

Atari 2600 has a number of properties that make it a unique environment for development and improvement of AI algorithms [36], some of which include: a large and diverse game library for both single player and two player games, simple but varying set of features, small action space, open source high speed emulator, and having access to the generative model.

As the original goal of AI research has been to develop a truly intelligent agent that can solve problems in general, an ideal comparison and evaluation method needs to test on a varied and broad set of problems. However, the set of games and test environments which are being used by different researchers is often not broad enough that one can claim such a general statement. And even if they cover a wide range of games with different characteristics, designing a set of experiments on all of them requires adaptation of the algorithms to all interfaces, whilst the Arcade Learning Environment (ALE) [46], is a framework on top of a Atari 2600 emulator that satisfies most of these requirements for being an excellent AI platform. ALE supports methods for saving a state and loading it later. This feature allows us to use it as a generative model with which we can construct a search tree.

Thus, ALE is an AI platform for evaluating and comparing general AI algorithms and as there are hundreds of games available the evaluation can be thorough. The phenomenon called over-fitting can apply in general evaluation and comparison of algorithms in AI as well as machine learning. In machine learning, when over-fitting appears, the obtained result shows higher performance than the actual performance in practice. In evaluation of generic algorithms in AI also, the training and testing games have to be disjoint to avoid achieving optimistically high performances.

2.4 Policy

A policy on a search tree of a deterministic domain, $\pi(action, state)$, is a probability function of action and state. $\pi(a, s)$ is a transition function that returns the probability of taking action a from node s. This probability determines how probable taking that action is. Simply put, policies are strategies. An optimal policy is one that maximizes the expected reward.

At each time step t, the agent takes actions according to $\pi(action, state)$ and then the environment responds with a reward signalling the beneficence of the action taken.

2.5 Monte Carlo Planing

For games with a small state space, the optimal policy can be found offline by using algorithms such as exhaustive Expectimax [47] or Q-learning [48] [31]. However, games and problems that we encounter in practice usually have either too large state spaces and/or high branching factors. In such situations, the aforementioned approaches for solving small problems are infeasible. Sample-based algorithms, on the other hand, are among the few feasible techniques that can be utilized in these cases. Kearns et al. [49] proposed a sample based look-ahead search. They demonstrated that a randomly sampled look-ahead tree can find a near optimal solution for an MDP in time independent of the size of the state space (yet still exponential in the search horizon). Abramson [14] showed sampling can be useful in estimating the value of an action.

One approach to searching in large state space problems is online planning. This involves repeatedly using local search to implicitly focus more on the more relevant parts of the tree. The goal is to find a good approximation of the best policy. A search algorithm starts searching from the current state, and after some criteria are satisfied, an estimated best action will be decided on, and the algorithm moves to the next time step. Because of finding an approximation instead of the best policy, typical search algorithms prune many parts of the search tree and this results in an effective reduction in the running time.

In online planning, a common way of constructing the search tree is to use a depth-limited iterative deepening [47]. This approach, however, works poorly in domains with large branching factors. The reason is the iterative deepening algorithm does not search deep in the state space for such domains. The outcome, therefore, would be a unsatisfactory result that could be far from the optimal solution.

A K-step rollout policy is an example of an earlier Monte-Carlo planning algorithm [50]. A K-step rollout policy is defined recursively using a k-1 step rollout policy and a default policy. A 1-step rollout policy is a simple example of such algorithms. It only needs a default policy and is as follows. At each time step, all actions are considered as different choices of the first action to take. Then, for each action, the default policy plays the game to the end a fixed number of times. The average outcome is the reward of taking that action. The action that earns the largest reward is the chosen action. The same procedure happens in a k-step rollout policy, but instead of using the default policy a k-1 step rollout policy finds the best action. In a 2-step policy, at each time step, all actions are different choices that should be considered. When any of them is being processed, first, that action is taken and then a 1-step rollout policy finds the corresponding reward. Compared to a k-1 step rollout policy, a k step rollout policy finds a more accurate estimated reward for each action. The major downside with this kind of approach is that its running time is exponential in k.

2.6 Monte Carlo Tree Search

MCTS is a best first search Monte Carlo planning algorithm. It uses two different policies to build a search tree: a tree policy and a rollout policy. MCTS algorithms repeatedly run a procedure called a simulation until a constraint is satisfied. The constraint that stops the search can be the number of simulations, or a limit on the memory or time. After running each simulation, the statistics kept in the tree get more accurate and informative, which allows the next simulations to be directed to more relevant parts of the search space.

In each simulation, the algorithm starts from the root initiated with the current state, and then the tree policy guides it to a leaf. In the second step, for expanding the tree, one (or more) of the children of the selected leaf will be chosen. Afterwards, the rollout policy plays the game until it reaches a terminal state. The outcome is a reward which is used to update the estimated value for that leaf [2]. Finally, the obtained reward is back-propagated to all of the ancestors of the leaf, which again update their return estimates. This step is called back-propagation.

Finally, when we have either run out of thinking time or search enough, the most promising action can be selected. This is the action that connects the root to one of its children. Different strategies can be used in this part of the algorithm [29]: 1- the child that has been seen the most, 2- the one that has the highest estimated expected value, 3- the one with the most estimated expected value and the most number of visits if there exists any, otherwise more simulations are needed until one is found or a certain criterion such as a limit on the number of simulations is satisfied. 4-the child with the highest lower confidence bound.

2.6.1 Details of MCTS

A detailed description of the four steps of MCTS are as follows:

Selection

The tree policy in MCTS algorithms is a policy that deals with the exploration-exploitation dilemma. The selection directs the search from the root to a node which is either a terminal state or an expandable node. An expandable node is a node that has not been completely expanded yet. In another words, Some of its children are not visited in the search tree.

After some simulations, the statistics kept in the tree will be more informative. Still, they can be noisy: nodes with currently high values could have been lucky so far and nodes with lower values unlucky. If the tree policy always picks the currently promising nodes, the optimal one can be missed. In contrast, if the tree policy explores all the time, many number of simulations are wasted in searching irrelevant parts of the tree, while this search

[2]Sometime in the literature, the term simulation is used in place of the term rollout.

effort could guide the search to the optimal solution in the relevant parts of the tree. This is an example of the exploration-exploitation dilemma. The tree policy must balance between exploration and exploitation.

Expansion

After selection has been done, and the tree has been traversed from the root to an expandable node, One (or more) of this node's children should be added to the tree. One simple method is to randomly pick one of the children from the set of unvisited children. Strategies such as picking the first unseen child, would bias the tree toward the first action. If there is access to more than only the search information, the expansion strategy can grow the tree more informatively. For example, in some domains heuristic information can be used to help choose which child to expand [32][29].

Simulation

After the expansion step, the game is played until it reaches a terminal state. The simulation step, also called a playout or a Monte-Carlo rollout, follows using a policy called the rollout policy. The rollout policy plays the game until a terminal state is visited. The outcome of a simulation is the cumulative reward it receives along the simulation trajectory. An example of a trivial rollout policy is to play randomly. The rollout policy can also use domain specific knowledge provided by an external source such as human played trajectory or it can be learnt online automatically [51].

Back propagation

When the rollout is finished, the outcome should be reported to all its ancestors. They can use this information to make better decisions in the future. Updated statistics stored in nodes help future tree policies to select the relevant parts of the tree more accurately, and thus the exploration is focused on a smaller part of the search tree. Nodes use the new information to update their statistics. For instance if the value kept in each node is the average reward of all simulations that go through this node, after receiving this new information, all ancestors must revise their estimated expected value.

2.6.2 Discount factor

Without paying attention to the temporal aspect of rewards, even by using a smart rollout policy, the difference between different choices of actions may not be apparent. This is often caused by having all rollouts end in a near optimal terminal state. More specifically, in many games and environments it makes sense to achieve a certain reward earlier if possible. Geometrically discounted rewards help the algorithm perform better in these situations

If the discount factor is $0 \leq \lambda \leq 1$ then the value of a transition that receives reward R at time t increases $R \times \lambda^t$ instead of R. This simply, puts emphasis on the time step a reward is achieved.

2.7 Upper Confidence Bounds for Trees (UCT)

MCTS methods can be implemented in many different ways. A crucial step that varies between many MCTS methods is how the selection is done. One such method is UCT.

UCT is a popular instance of MCTS that also has many good properties. As opposed to many MCTS methods, it has a theoretical guarantee of convergence to the optimal solution. Previous attempts on MCTS did not have proof of convergence, even though they were popularly used because of their good performance on many problems. Lack of theoretical proof, however, means the repetition of MCTS steps can be useless and does not necessarily improve the performance.

The tree policy of MCTS has access to the statistics kept in each node and has to decide which path to traverse. It either chooses a currently promising path or a path that does not look good but potentially can turn out to be the best. In UCT, the exploration-exploitation dilemma is solved by sequentially applying UCB1 for the tree policy.

Multi Arm Bandit is a gambling machine with many arms e.g. a k-arm bandit problem has k arms. The reward of pulling each arm follows a specific stochastic distribution. The Bandit problem aims to accumlate the highest rewards given a limited number of tryouts, which in the long run is to find the most promising arm. In UCT, the node selection problem is reduced to a MAB problem. Each arm represents a possible action, and the reward received because of pulling an arm is the outcome of the simulation.

At each node, all children are ranked according to their UCB1 value, which is a type of upper confidence bound. This adheres to the principle of Optimism in Face of Uncertainty. This principle argues that we can have a good balance between exploration and exploration if we are optimistic about things which we are uncertain of. The number of times each child is visited, is a good estimate for how certain we are about the value of node. Thus the less a node is visited, the higher its confidence bound would be. Simply put, UCT picks the child that satisfies the formula 2.1:

$$k = argmax_{i \in children\ of\ p} v(i) + C \times \sqrt{\frac{\ln(visited(p))}{visited(i)}} \tag{2.1}$$

where p is the parent node, v(x) is the expected value of node x, and C is a constant value that controls the exploration/exploitation trade-off. This has to be tuned in practice, with the optimal value being highly dependent on the problem. Although in each simulation only some MAB problems are solved, nodes in the higher levels of the tree are more likely to be seen by the tree policy and thus more arm pulls are performed for those MAB problems. This brings more information to these problems, and future arm selections will be done more selectively. This is also true for all nodes in the related parts of the search tree and their MAB problem.

Most variations of UCT first require that the children be visited more than some fixed amount of times, before then reverting to Equation 2.1 (a threshold) and then use this formula as the selection strategy. In tuning the constant value, C, for big C values the algorithm tends to explore more, and if this value is very big relative to the expected values, the exploration will resemble that of full tree search. On the other hand, when C is a small value, the algorithm tends to go through currently promising branches in the search tree, but may miss the optimal actions due to a lack of exploration.

There are many ways to improve the performance of UCT. Some suggested using modified formulas in the node selection step [29][52]. Chaslot et al. [28] tried to automatically tune the parameters e.g. the C constant factor. Many researchers also tried to enhance and improve UCT by applying ideas such as Rapid Action Value Estimation [25], transposition tables [26], parameter tuning [27], using heuristic or domain specific knowledge [29][30], and also reducing the variance of the rollout value estimates [31].

2.8 Progressive strategies in UCT

UCT has difficulty in domains with high branching factors, thus, some techniques have been proposed to address this. In these situations, the limited number of simulations distributed on all actions may not be able to distinguish different actions to find the optimal one. In UCT, when nodes are only visited a few times, even a random strategy might work better than a poor tree strategy. Chaslot et al. [29] introduced two approaches to address this problem: progressive biasing, and progressive unpruning. Progressive bias, adds another heuristic function to the UCT formula. This heuristic function uses problem specific knowledge to guide the search to those parts of the tree that are more likely to have higher expected values. The new search algorithm, therefore, is directed toward relevant parts of the tree that are suggested by the heuristic function. Progressive unpruning, on the other hand, first reduces the branching factor to a small number, and later, gradually adds more branches as the search continues to grow the tree. Progressive unpruning uses a heuristic function to control the rate at which new children are added to a search node. Both these algorithms have shown significant improvements, and can be used individually, or in combination.

Finally, Coulom [32], proposed progressive widening, which is a very similar algorithm to progressive unpruning but was invented independently [32][29]. Progressive widening sorts all possible branches, using Elo rating [53] and builds a stochastic distribution over all possible branches. In the Elo ranking an occurrence of a pattern is being considered as a victory for that pattern over the other patterns. This replaces the heuristic function used in progressive unpruning.

2.9 Context Tree Weighting

Context tree weighting (CTW) is a binary sequence predictor proposed by Willems et al. [54]. CTW has both a theoretical proof of efficiency and is known to perform well in practice on many different types of data [55].

CTW uses a Bayesian mixture for prediction. It computes a mixture of all suffix trees with depths less than a pre-specified depth. It uses a weighted context tree, which allows it to update its model linearly with depth. A context tree is a perfect binary tree of depth D. In each node, the number of ones and zeros are stored and are updated as more bits are seen. Then, a weighted probability for each node is calculated. For calculating the prediction, CTW looks at the path corresponding to the sequence. Figure 2.2 shows a sample context tree for source sequence 0101110 with past ...01. *a* counts the number of 0s and *b* counts the number of 1s for each past sequence. The probability of the next bit of the input sequence is computed based on the probabilities stored at each node and propagated to the root node. *a*, *b* and the probability of each node get updated along a single path in tree every time a bit of the input sequence comes in.

Although CTW is a binary sequence predictor, it can be easily modified to predict sequences of integers as well. Thus, it can learn a sequence of integers to build a probability model. Thus we can use this to learn a stochastic policy. Given a threshold on probability, CTW can find all sequences of actions that are more likely than the threshold to appear as the next sequence. CTW has numerous applications in many different areas [56] [51].

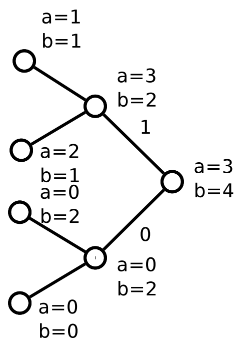

Figure 2.2: Sample context tree for source sequence 0101110 with past ...01. *a* counts the number of 0s and *b* counts the number of 1s for each past sequence.

2.10 Temporal Abstraction in MCTS

Although temporal abstraction techniques are studied in the history of AI [33] [57] [58], they have not been adapted to MCTS algorithms. Only a few works on MCTS have used temporal abstractions, and this was done mostly in a simplistic manner. Naddaf [36] used constant-length macro actions on different algorithms to reduce the size of the state space and the result was a significantly improvement. Powley et al. [37] used the same technique on their MCTS algorithm in a route finding problem. They ended up first in the 2012 WCCI PTSP competition[38]. A more principled approach, however, is still lacking.

Chapter 3

Variable Time Scale

In this chapter we introduce the Variable Time Scale algorithm (VTS). VTS is a modification to MCTS algorithms, which tries to automatically find the best time scale for planning as part of the search process. VTS is able to search much deeper in large state space domains and thus can outperform vanilla MCTS algorithms. It benefits from using macro actions to explore a more compact state space, therefore it can explore deeper into the future compared to vanilla MCTS.

However there is a catch: although many problems may benefit from compacting the state space, this does not necessarily improve performance. VTS acts smartly regarding this issue: to ensure beneficence of compacting the state-space, VTS modifies the length of the macro actions used during the search process.

The motivation for this idea comes from the fact that in many games, different temporal intervals are required to abstract the situation effectively. Sometimes, quick responses are needed, while in other situations the best policy is to execute the same action repeatedly for a number of times. For example, in an imaginary game, the agent needs to get away from a bullet's path and then it has to jump and shoot a bullet at an enemy. Subsequently, the agent needs to go to the right for 10 seconds and then stop for 5 seconds. The former part of this scenario needs quick responses. While, the latter part requires a lengthier perspective in time. In other words, the number of times that the optimal policy picks an action consecutively is not a constant value and it can vary in different situations of the same environment.

Consider when the action set is $\{A, B\}$ and the optimal policy is the following sequence of actions: AAABBBBBBBAAAAAA. Ideally, if we could find the proper repetition time for each action then the optimal solution would be found with a depth three search tree. A macro action of size three in the top level of the tree: AAA; then, a macro action of size seven: BBBBBBB; finally, a macro action of size six: AAAAAA would be the macro action ending in the optimal terminal state, which is in depth three of the tree 3.1.

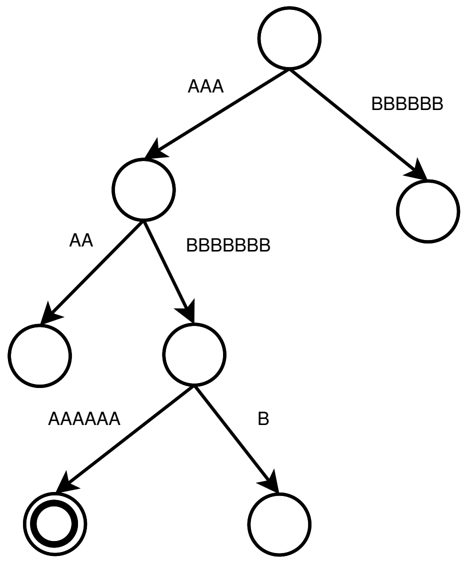

Figure 3.1: If we could find the best repetition time of each action in each state, the optimal policy would be found in a node much closer to the root

But how do we achieve this? VTS extends MCTS by adding a new type of node to the search tree, allowing the algorithm to make decisions on how to abstract the problem. Determining the best length of the macro action — that should be used in the following sub-tree, is what is done at this new node. Simply put, the new node tries to find the proper number of times that each action should be repeated before switching to a new one.

3.1 How does VTS work?

Before starting to explain the detailed workings of VTS, it is necessary to specify what we mean by macro actions in VTS. Macro actions generated in VTS are only a small set of simple macro actions: repetitions of single actions.

Assuming the action space is size two, $\{A, B\}$, The set of macro actions that are a repetition of a single action and their length is less than 7, is a set of 14 elements: $A, B, AA, BB, ..., A^7, B^7$. One method to utilize these macro actions in UCT is that, they can be considered as different branches of each node. However, this drasticly increases the branching factor and would not be handled well by the tree policy. The new branching factor is: number of actions $\times$ highest length of macro actions considered.

A two layer approach, however, can fix this problem (almost). One layer decides on the proper number of times that an action should be chosen consecutively, while the other layer chooses the action. For example, if the first layer chooses that the proper length is 2, then the choices of macro actions in the second layer are AA and BB. To apply this idea on UCT, regular nodes of the search tree are replaced by this two layer structure.

Nodes in the first layer are called compactor nodes, and nodes in the second layer are called decision nodes. Through each path from the root to any of the leaves, nodes alternatively change from one type to another. The children of a compactor node are all decision nodes and each decision node is also connected to only compactor nodes. In Compactor nodes, different integer numbers are different choices for macro action lengths, and each is connected to a decision node which uses that length in taking the action. Thus, each branch in a decision node is labeled by one of the actions, repeated K times, where K is the number from the parent compactor node. The policies of choosing a branch in both compactor nodes and decision nodes are the same as any other node, e.g. in UCT the UCB1 formula is used. The structure of the tree is shown in 3.2.

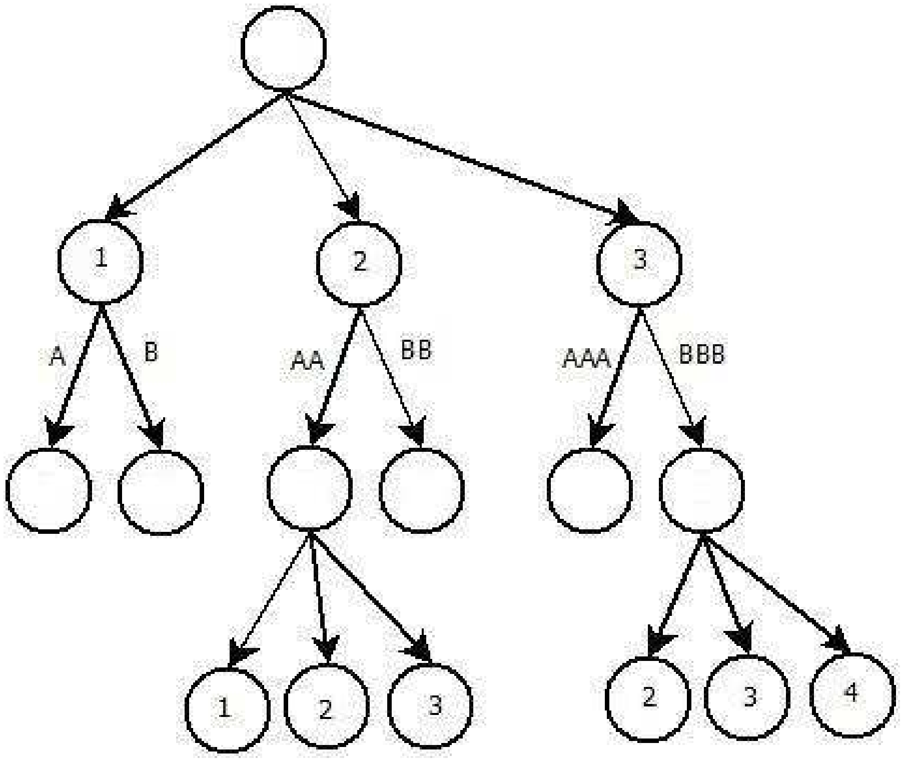

Figure 3.2: This is a very simple example of how a two layer tree structure could be. Using integer values that are powers of two makes this structure more efficient. Decision nodes find the action, while compactor nodes find the proper repetition of each action. Through the path from root to the leaves, the node types are alternately changing from one type to another. Each decision node, decides between different macro actions with the specified length. While a compactor node tries to find the desired macro action length, given a small set of different lengths.

For designing compactors, there is a trade-off between the branching factor and the maximum length of macro actions: if we consider all macro actions of length less than or equal to L, for small values of L, some macro actions that are needed in the optimal policy are not considered in branches of a node. These macro actions that have length higher than L have to be broken into several parts in order to be used in the tree search. On the other hand, when L is big, the branching factor is also big. A high branching factor reduces the efficiency, because to get the same performance, it requires more simulations.

Our approach to compensate for this problem is that we put a limit on the branching factor of the compactors, but the set of lengths they can consider vary across the search tree. Therefore, each compactor node uses a small set of different lengths as its branches, adjusted specially for that node, according to the path from the root. For example, if at one level of the tree the choices ar e 2, 4 and 8, and the algorithm chooses length 8 over the other two, then the lower compactor node may consider lengths 4, 8 and 16. And this will let the algorithm learn to use longer/shorter macro actions if needed. Using integer values that are different powers of two, makes the adaption strategy to be able to quickly reach to the proper lengths during the search. Also, any particular power of 2, 2^L, can be reached in $O(L)$. This means, if the best time scale for an action is T, the outcome of repeating that action T times, can be produced in the search tree in $O(log(T))$ steps. To keep the branching factor low, we always consider 3 consecutive powers of two as the adaptive set.

The same idea can be used in the root level of the tree as well. That is by considering the final action that the algorithm returns in one time step, the initial set of lengths of the root for the next time step may change.

This setup still may require many simulations to find the proper length, even though it will get some satisfactory results. In order to use the benefits of compactor nodes and avoid wasting the search effort in finding the proper length redundantly, compactor nodes are only added to the tree with probability P. Reducing the number of compactors, makes the search effort focus more on the action selection, while with a more desired P the compactor nodes can set the length to the desirable value when they get the chance. P is parameter that must be tuned for different problems. Big values of P, let the search change the macro actions length more frequently, while a smaller value results in searching with an almost constant length macro actions for a long period of time. Thus, the P should be set according to the problem: when the desired length does not frequently change over time, a smaller value of P is more suitable, otherwise a larger value is more appropriate.

Summing up, the new structure of the search tree is as follows. The root is a compactor node. Each compactor node is connected to some decision nodes corresponding to different lengths that should be considered. The children of decision nodes are either decision nodes or compactor nodes. Each child of a decision node is a compactor node with probability P.

3.2 Algorithm

In the search tree, we use a new type of node called multiplier, which is in fact a compactor node that is the mechanism of adapting to the suitable length. Each tree node is associated with an integer number that is the length of the macro actions in that node. The edges from a multiplier node are values $\frac{1}{2}$, 1 and 2. This value is multiplied by the value associated with the parent node becomes the value associated with the child. The children of a decision node are all actions, each repeated K times where K is the number associated with that decision node. Figure 3.3.

We specialize our approach to UCT, In each simulation, the algorithm starts at the root of the tree and traverses down the tree, until it reaches a leaf. Each node is either a Multiplier node or a Decision node. In both node types, the UCB1 formula is used to make decisions. The root itself is always a Multiplier node. During the expansion part of the algorithm, if the node is a multiplier node then its children are all decision nodes each associated with a macro action length, while when the node is a decision node, each child is a multiplier node with probability P and a decision node with probability 1-P. Pseudocode for the algorithm is shown in Algorithm 1.

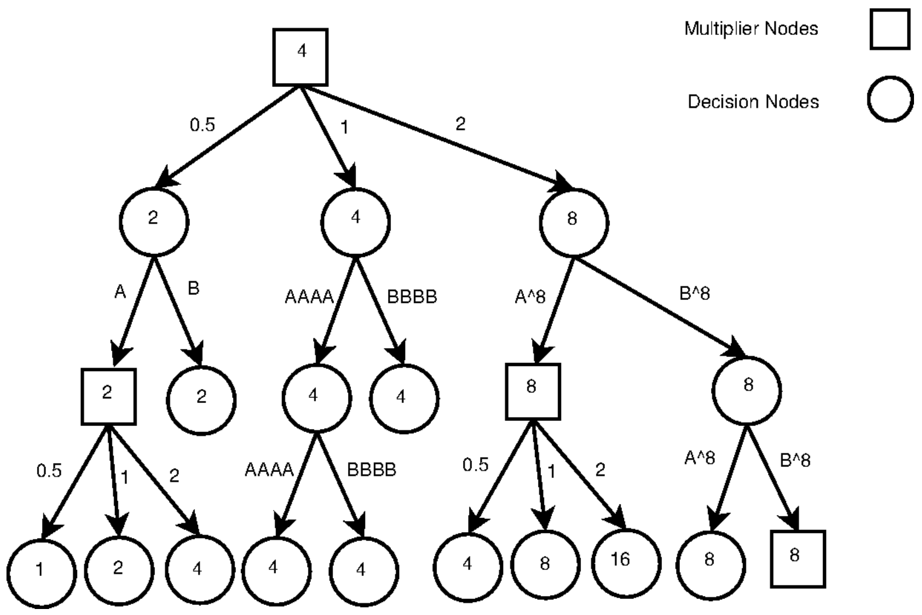

Figure 3.3: VTS tree structure: In VTS, multiplier nodes are only inserted in the tree with probability P. They multiply the length of the macro actions by $\frac{1}{2}$, 1, or 2. A number is associated with each node which is the multiplication of the values on the edges of the path from the root to that node. The length of the macro actions that we consider in a decision node is the value associated with that node.

Algorithm 1 Simulate(s,T,h,depth,prd)   //s: State, T: Tree, h: History

```
if $\gamma^{depth} < \epsilon$ then
    return 0
end if
if current_state $\in$ leaves of T then
    $c = Expand(current\_state)$
    $Open(T, c)$
    return Rollout(c)
end if
choices = $\phi$
if current_state $\in$ DecisionNodes then
    choices = $\{a^{prd} | \forall a \in Actions\}$
    $m \leftarrow argmax_{b \in choices} V(current\_state + b) + c\sqrt{\frac{\ln(N(current\_state))}{N(current\_state+b)}}$
    $s' = \mathcal{G}$(s,m)     // the new state, s', is generated by taking action m from state s
    $df = \gamma^{prd}$
    $h' = h + m$
    $depth' = depth + prd$
    $prd' = prd$
else if current_state $\in$ MultiplierNodes then
    choices = $prd \times \{\frac{1}{2}, 1, 2\}$
    $prd' \leftarrow argmax_{b \in choices} V(current\_state + b) + c\sqrt{\frac{\ln(N(current\_state))}{N(current\_state+b)}}$
    $r = 0$
    $s' = s$
    $h' = h + prd'$
    $df = 1$
    $depth' = depth$
end if
$R \leftarrow r + df \times Simulate(s', T, h', depth', prd')$
$N(current\_state)$= $N(curernt\_state) + 1$
$V(current\_state) = V(current\_state)+ \frac{R - V(current\_state)}{N(current\_state)}$
return R
```

Algorithm 2 Open(T,c)

```
if c $\in$ DecisionNodes then
    edges = $\{a^{prd} | \forall a \in Actions\}$
    for each e in edges do
        if $random < P$ then
            chlid = $MultiplierNode$
        else
            child = $DecisionNode.$
        end if
        $add(T, c + child)$    // for each action, a new node is added to the list of children of
        c
    end for
else if h $\in$ MultiplierNodes then
    edges = $prd \times \{\frac{1}{2}, 1, 2\}$
    for each e in edges do
        child = $DecisionNode.$
        $add(T, c + child)$
    end for
end if
```

Chapter 4

UCT with CTW macro actions algorithms

In this chapter we introduce CTW-UCT algorithm that automatically generates and uses the generated macro actions in search. Macro actions used in CTW-UCT are sequences of actions varying in length and content by following a probability model that changes over the search process and is built by the Context Tree Weighting (CTW) algorithm using samples of expert play.

As mentioned before, algorithms can benefit from using macro actions to build smaller and more abstract search spaces that can lead to significant improvement, if done intelligently. On the other hand, if the problem is not abstracted well, the search may be guided to irrelevant parts of the state space, and thus the optimal and near optimal solutions may be missed, or require even more simulations to find it.

By looking at sequences of actions that an agent has taken to play a game, we can see some sequences and patterns that occur frequently. Hence, if this kind of information could be extracted from the current search results or from offline (e.g. human played) expert trajectories, it would be logical to allow the search algorithms to consider these repeated and frequently used patterns as their different options in the search.

As an example of pieces of information hidden in well-played trajectories for an imaginary game consider the following scenarios: when the agent frequently needs to jump and shoot the opponent, or when it is taking the stairs to go from one floor to other floors, or when the agent is running away from an enemy that is trying to chase it. If we could extract these pieces of information and make use of them, it would result in a smarter search algorithm. For example, we could always consider shooting after jumping, as the action that should be taken. Also, when the agent is taking the stairs to go to another floor, it should repeatedly keep taking the same action for a number of times to achieve this sub-goal. In the latter case, also, assume the agent is going to the right direction and the enemy is trying to chase it while it is a few steps behind. In this situation, going to the left direction is probably not a smart decision that we could also find out by having access to a successful game trajectories: Most likely, going to the left was rarely seen after a number of times that the agent has gone in the right direction.

Although considering the most likely action (or the sequence of actions) using the current search data is smart, it may not always be the best choice. Even if it is, finding the proper time to take the new action (or actions) might be not an easy task. For example, in the first scenario mentioned above, when the agent jumps and shoots, the timing of shooting the bullet may be very important and a wrong timing may result in poor performance.

To act smartly regarding this issue, we use a probability model trained with well played trajectories. This model distributes probabilities over all possible choices of actions (or macro-actions), allowing the algorithm to benefit from using this information. Ideally, it would split the search space into small meaningful trajectories (macro actions), while other non-meaningful choices are also considered in the search.

In CTW-UCT, the CTW algorithm builds this probability model. CTW-UCT is a modified version of MCTS. In this thesis, we specialize our approach to UCT. Thus, in CTW-UCT macro actions are generated by CTW and are considered as different branches in the UCT nodes, allowing the algorithm to abstract the problem space. CTW is fed with the information gained during the search. In addition, the expert human played trajectories can also be used in CTW to build a more informative model.

4.1 How does CTW-UCT work?

Before explaining the CTW-UCT algorithm, it is necessary to talk about some issues. First, a short explanation on how do we make use of CTW is provided. Second, a strategy for dealing with large branching factors is discussed, as all of these algorithms are dealing with this problem, as they consider different choices of macro-actions in their nodes. Finally, we examine some simpler algorithms designed using the same idea, which will motivate the CTW-UCT algorithm. These algorithms all use a progressive widening technique to reduce the branching factor and CTW to build a probability model. This probability model is the preference function on different macro actions, determining the order to add macro-actions as the search progresses.

4.1.1 How is CTW used?

CTW uses both online and offline information to build its model: the current search tree and expert (e.g. human played) trajectories. The offline expert trajectories are fed to CTW before starting the search, while the search information is given to CTW during the search process. The online information that CTW uses, however, is only the sequence of actions that takes the game state from the initial state to the current state that CTW is building the model for. This is due to the fact that, only sequences that are believed to be well-played, should be given to CTW. The tree policy in UCT, also, picks the action that is believed to be the best choice at each time. Thus, for each node of the tree, we only let CTW learn the current sequence of actions that changes the initial state (the root of the tree) to the current node. (Figure 4.1)

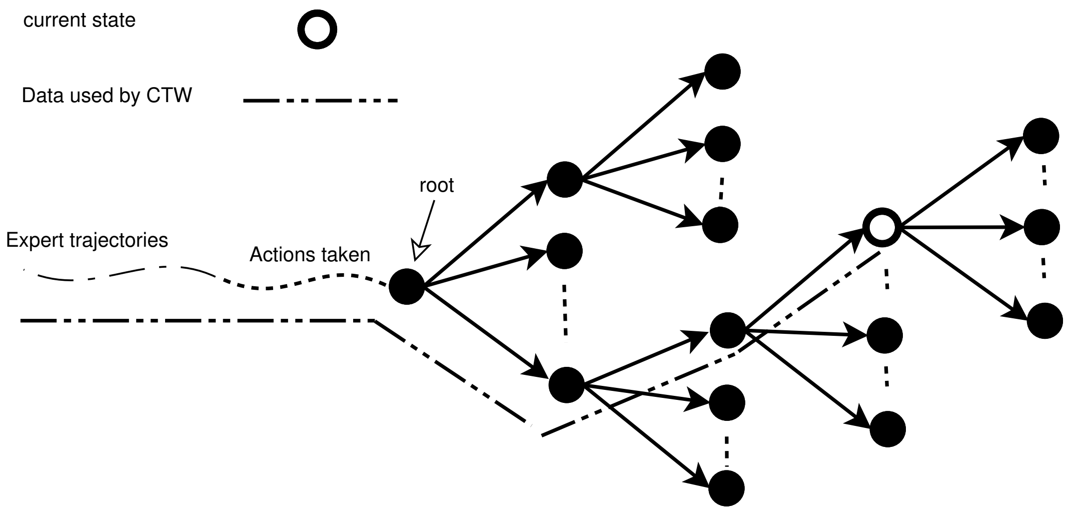

Figure 4.1: CTW using offline and online information. The offline expert trajectories are fed to CTW before starting the search, while the search information is given to CTW during the search process. The online information that CTW uses, however, is only the sequence of actions that takes the game state from the initial state to the current state that CTW is building the model for.

4.1.2 How is the large branching factor dealt with?

As mentioned in the background (Section 2.8), there are some strategies for dealing with problems that have large branching factors. In the algorithms proposed in this chapter, we need such mechanisms as we will end up with many different macro actions in each node. For example if only all macro actions of length 5 are considered, assuming the size of the action space is 18 (as it is 18 for atari games), the branching factor is $18^5 \simeq 2000000$.

Our solution to the large branching factor is to use the progressive widening strategy. Each time a node is created, the search opens the first few pre-specified (F) children, according to the expansion policy (which is a probability model made by CTW in our algorithms). Next, the pruning is done and, according to the progressive widening formula, new nodes are added: $T_{n+1} = T_n + A \times B^n, t_0 = 0$. T_n is the number of simulations required for adding the (n+F)th move. A and B are parameters that should be tuned according to the problem and they tune the pace of adding new nodes to the tree.

4.1.3 Algorithm 1: CTW-UCT1

CTW-UCT1 is a UCT algorithm that uses constant length macro actions of length L. Macro-actions used in CTW-UCT1 are all repetitions of single actions. In fact, this algorithm is the same as constant length UCT, but it uses a progressive widening. The Figure 4.2 demonstrates the search tree in more details.

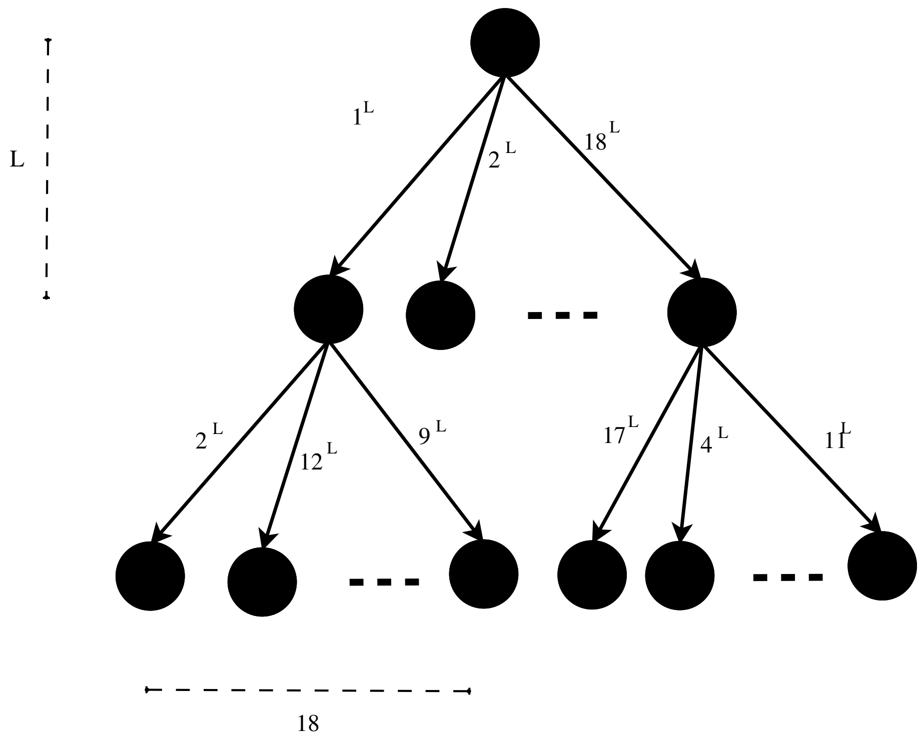

Figure 4.2: CTW-UCT1 algorithm search tree. CTW-UCT1 uses constant length macro actions of length L. Macro-actions are all repetitions of single actions. Therefore the branching factor is 18.

4.1.4 Algorithm 2: CTW-UCT2

CTW-UCT2 is also a UCT algorithm that uses constant length macro actions of length L. However, compared to CTW-UCT1, the macro-actions used here are more generic and can be any sequence of actions of length L. Therefore, if the size of the action space is S, then number of branches in each node is S^L. For example, for the problem in hand, Atari 2600, this is 18^L. This huge branching factor is dealt with using progressive widening strategy as well. Each node starts with an empty set of children and over time, macro-actions suggested by CTW are added. At time t_n when a new node is going to be added to the tree, all macro-actions of length L, that are not seen already, have the chance to be selected according to the CTW probability model. Figure 4.3 demonstrates the search tree in more details.

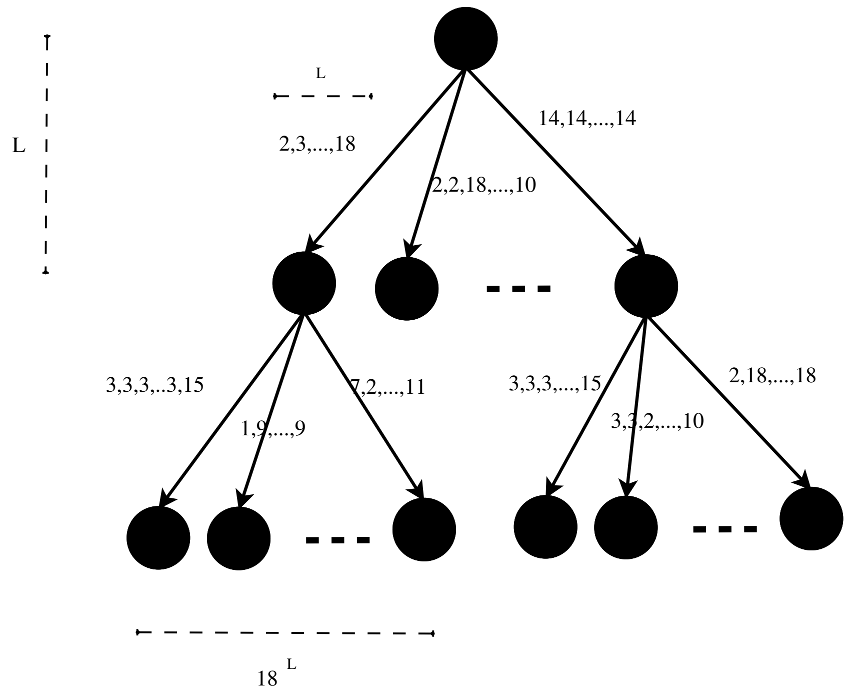

Figure 4.3: CTW-UCT2 algorithm search tree. CTW-UCT2 uses macro-actions of length L. Each macro-action can be any sequence of actions of length L. They are all added by CTW to the search tree. As all possible macro-actions are considered, the branching Factor is 18^L and thus progressive widening technique is used.

4.1.5 Algorithm 3: CTW-UCT3

CTW-UCT3 is an algorithm based on UCT too. Compared to CTW-UCT1 and CTW-UCT2, it uses macro actions of different lengths. Macro actions used in this algorithm can also be any sequences of actions and are suggested by CTW. At time t_n when a new node is added to the tree, CTW suggests a new macro action that is likely to be a good option. This macro action is then added to the tree and search continues. As the number of bits CTW uses to represent any sequence is inversely proportional to the likelihood of that sequence being the next sequence (num of bits $\sim$ -$lg($ probabilty of the sequence being the next sequence$)$ + Constant Value), a threshold on the number of bits that any macro action can be represented with, is a threshold on its likelihood. In other words, the higher probability of a sequence happening (compared to all sequences of the same size), the shorter is the number of bits that CTW needs to represent that sequence, thus CTW-UCT3 only adds macro-actions that can be represented by a small pre-specified number of bits. Simply put, it only adds macro actions with probability higher than a pre-specified threshold.

Figure 4.4 demonstrates the search tree in more details.

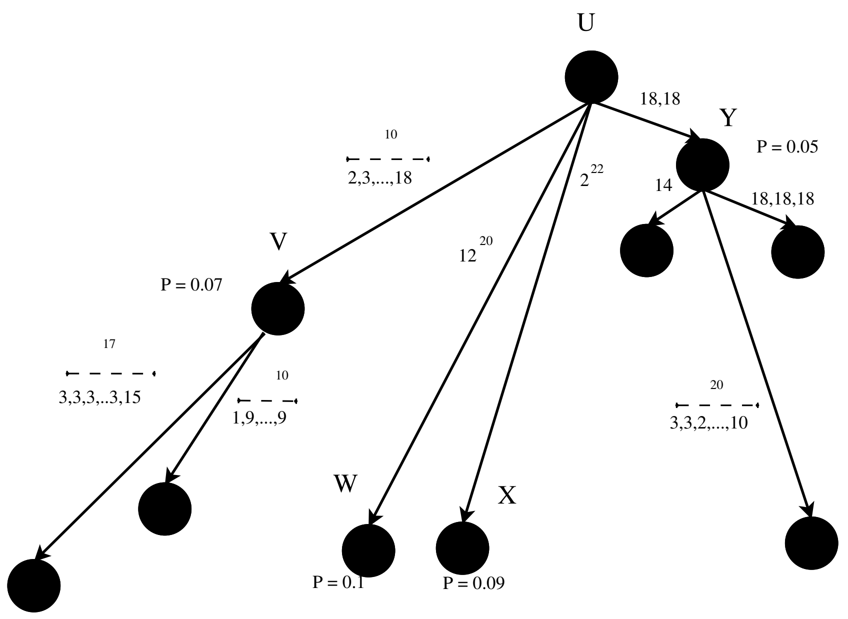

Figure 4.4: CTW-UCT3 algorithm search tree. CTW-UCT3 uses macro-actions of different lengths. Each macro action can be any sequence of actions that is more likely than a threshold to be the next sequence of actions. They are all added by CTW.

4.1.6 Algorithm 4: CTW-UCT

CTW-UCT modifies CTW-UCT3 to compensate for an important issue regarding the exploration in search. Since CTW always generates macro actions to be considered in search, in many situations the first few actions for all these macro actions are the same or very similar to each other. This phenomenon practically results in failing to explore many other actions without a large number of simulations. In order to explore more in CTW-UCT, therefore, the policy of adding macro-actions in each node is replaced by another policy with ϵ probability. This policy in CTW-UCT is a random policy (each time an action is selected randomly). This solves the problem of lacking exploration in the CTW-UCT3 to some extents. Figure 4.5 and Algorithm 4.1.6 describe the algorithm in more details.

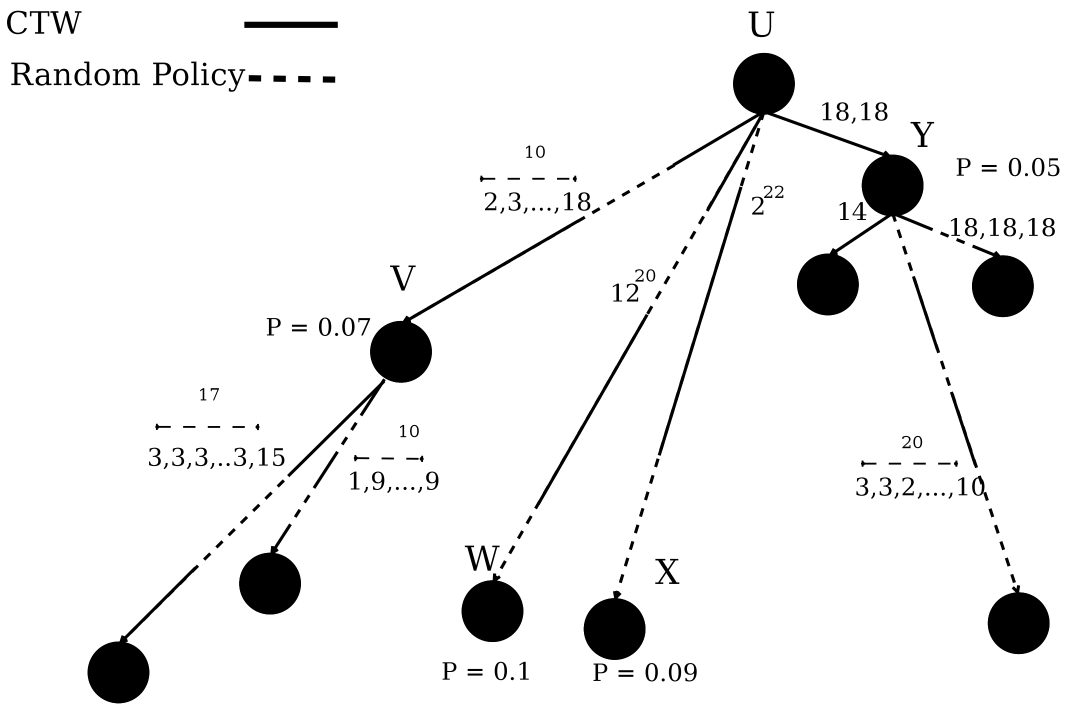

Figure 4.5: CTW-UCT algorithm search tree. CTW-UCT modifies CTW-UCT3 to add some exploration. In order to do so, the policy of adding macro-actions in each node is replaced by a random policy with ϵ probability. Thus, each edge is a macro-action that is a mixture of CTW suggested actions and sometimes random actions.

Algorithm 3 CTWUCTSimulate(s,T,h,depth)   //s: State, T: Tree, h: History

```
if $\gamma^{depth} < \epsilon$ then
    return 0
end if
if current_state $\in$ leaves of T then
    $Open(T, current\_state)$
    return Rollout(current_state)
end if
if number of simulations = $t_{add}$ then
    // a new macro-action is being added to the list of children of the current state
    $t_{add} = t_{add} + (A + B^n)$
    1:
    Macroaction m = []
    while bits needed by ctw < threshold do
        if $random < \epsilon$ then
            m = m + a random action
        else
            //CTW.next() is a function that returns an action according to the probability
            // model built by CTW
            m = m + ctw.next()
        end if
        //CTW should update its model after an action is added to the macro action
        ctw.update()
    end while
    if $m \in current\_state.children$ then
        goto 1     //to find a new macro-action
    end if
    current_state.children.add(m)
    // When the macro action is generated. CTW should revert its model to the current
    // state's model
    ctw.remove(m)
end if
choices = current_state.children
$m \leftarrow argmax_{b \in choices} V(b) + c\sqrt{\frac{\ln(N(current\_state))}{N(b)}}$
$s' = \mathcal{G}$(s,m)     // the new state, s', is generated by taking macro-action m from state s
$df = \gamma^{m.length}$
$h' = h + m$
$depth' = depth + m.length$
$ctw.update(m)$     CTW updates its model along the path
$R \leftarrow r + df \times CTWUCTSimulate(s', T, h', depth')$
ctw.remove(m)
$N(current\_state)$= $N(current\_state) + 1$
$V(current\_state) = V(current\_state) + \frac{R - V(current\_state)}{N(current\_state)}$
return R
```

Chapter 5

Evaluation Method and Experimental Results

In this chapter we talk about our evaluation method and compare the results of our different algorithms on both a set of preliminary games and test set of games. We evaluate the algorithms proposed in the two previous sections, VTS and CTW-UCT. We compare these methods to UCT using constant macro actions of varying lengths. In this section, we talk about the experimental setup, evaluation method and results interpretation.

As mentioned before, to avoid over-fitting, the preliminary and testing games need to be disjoint. The nature of this research also requires very fast domains. Thus as for the preliminary set, two very simple games are implemented with very fast simulation time, allowing the algorithms to achieve early results and thus, to be evaluated and tested in a much shorter time. The test games are Atari 2600 games, which make use of the Arcade Learning Environment framework [46].

The proposed algorithms are compared to different UCT algorithms, with each one making use of macro actions of different pre-specified lengths. This is due to the fact that as we allow both VTS and CTW-UCT to generate macro actions of different lengths, it would not be entirely fair to compare VTS to vanilla UCT. Each of these UCT algorithms uses a fixed power of two as its pre-specified macro action length. And since neither VTS, nor CTW-UCT is allowed to use macro actions of length higher than 64 in the experiments, the pre-specified length for these algorithms is also less than or equal to 64. Thus different UCT algorithms that we compare to our algorithms are: CA1, CA2, CA4, CA8, CA16, CA32 and CA64 (CA stands for ”Constant Action”).

5.1 Preliminary Set

5.1.1 Game 1: The Shooter

The shooter is a very simple game. All objects in this game are circles. There are two different types of objects: the agent and the enemies. All enemies come from the left side of the screen and they proceed to the right side with different speeds. When the agent shoots a bullet at an enemy, it decreases the speed of the hit enemy. If an enemy touches the agent, the agent dies. Therefore to play well, the agent has to run away from enemies and shoot them to reduce their speed in order to facilitate running away. The agent is periodically rewarded for staying alive every 30 steps. Also, each time the agent shoots an enemy for the first time it receives some reward in the range of [1,6] based on their distance. Figure 5.1 shows some screenshots of the game.

5.1.2 Game 2: The Cookie Lover!

The cookie lover is another simple game. All objects in this game are also shaped as circles of different radii. Three object types are used in the game: the agent, healthy cookies and poisoned cookies. The goal is to collect as many healthy cookies as possible and to avoid poisoned cookies. The agent walks around the screen trying to accomplish this goal. New cookies are added to the screen and old ones disappear after a while. As time goes on, the new healthy cookies added get smaller and the new poisoned cookies get bigger than previous ones. As for collecting a healthy cookie, the agent is rewarded by 4 points. Also the agent is periodically rewarded for staying alive every 100 steps. Figure 5.2 shows some screenshots of the game.

5.2 Experimental Setup

We tuned our algorithms based on the two preliminary games mentioned in Section 5.1. We then tested these methods on 10 different Atari games: seaquest, riverraid, ms-pacman, road-runner, boxing, enduro, ice-hockey, phoenix, wizard-of-wor and beam-rider.

The discount factor and the parameters of progressive widening are the same for all runs. Since Atari games have a wide range of possible scores, we use two different exploration factors for UCT: one for games in which the reward of taking an action is less than or equal to 1, and one for games that have higher values. The epsilon is 0.1 for both CTW-UCT and VTS algorithms. The length of the horizon for all rollouts is 16.5 seconds (1000 frames) into the future. Each episode runs until the agent dies or the time limit is exceeded. For most of the games each episode's time limit is 5 minutes and 33 seconds (20,000 frames). Some of the games need more resources and running 20,000 frames takes more than 120 hours. For these games each episode's time limit is set to 2 minutes (7200 frames). Actions are selected every 5 frames. The number of simulations in each search step used to construct the search tree is 4096 for the preliminary games and 256 for test games. The test games are randomly chosen from different categories of Atari games. Each algorithm is run for more than 15 episodes per game.

5.3 Evaluation Method

Evaluation and comparison of generic algorithms on a large set of varied domains (such as Atari 2600) is not an easy task. This is because different games can have quite different score ranges. Thus, if algorithm A beats algorithm B on one of the games and the opposite happens on another game, it is hard to compare these results at first glance. Normalizing all scores to an interval of [0,1] is one way to solve this problem. Suppose we have different algorithms and we want to compute the normalized scores on one of the games. The normalized values can be found by scaling these values into [0,1] interval shown in Formula 5.1.

$$N(v|v_1, v_2, ..., v_k) = \frac{v - min_{1\leq i\leq k}v(i)}{max_{1\leq i\leq k}v(i) - min_{1\leq i\leq k}v(i)} \tag{5.1}$$

This method is the inter-algorithm normalization introduced in Bellemare et al. [46]. Here the best algorithm will achieve the score of 1 and the worst will achieve a score of 0. All other algorithms achieve a score between 0 and 1.

Once the normalized scores are computed we can compare them by using different methods discussed in bellemare et al. [46]: 1- the average score, 2- the median score, and 3- the score distribution.

Both the average score and the median score are methods for aggregating the normalized scores for comparing results of different algorithms over a set of games. The median score compared to the average score, is less sensitive to outliers, however, it does not always show a thorough comparison. For example the following two sets of rewards have the same median, while they are not equally good: $\{0, 0, 1, 1, 1\}$ and $\{1, 1, 1, 1, 1\}$. As we will see in the next section, our results on the game Enduro suffer this problem.

The score distribution is a generalization of the median score and is helpful in analyzing the results. For each value on the x-axis, which is representing a normalized score, the score distribution shows the fraction of games on the y-axis that the algorithm has done equal or better than that score. It can be interpreted as a kind of reverse empirical CDF. Thus, the area under each plot is the average score. If one plot is constantly above another plot, it is performing better. To make it more clear a few examples are shown in Figure 5.3: an ideal algorithm, the worst possible algorithm, an always mediocre algorithm, and an ordinary algorithm.

5.4 Results

In this section, we first show the results of all algorithms on the two preliminary games. Then, as for the test games, the results of all algorithms except for all CTW-UCT algorithms are shown and discussed. As the CTW-UCT algorithms did not achieve comparable results to other algorithms in the preliminary set, we did not run them on the test games.

5.4.1 preliminary Games

Results on the preliminary set are shown in Tables 5.1 and 5.2. The result of the best CA algorithm and VTS, both the average and the median, were only slightly different on the two preliminary games. As the results were not really distinguishable, we ran the best CA algorithm and VTS for many more trials in order to find a confidence interval in which we could be certain whether VTS is able to achieve higher rewards or at least as good as the best CA algorithm. In the Shooter game, VTS stays behind by a small margin, and in the Cookie Lover game, they are almost identical, however VTS leads by a very small difference. Note that, among all CA algorithms, CA32 is the best choice based on the preliminary games.

Multiplier nodes in VTS are added with probability p. This is because we believe most of the time spent in search should be on finding the action rather than the desired time scale. As an experiment, running VTS with probablity 1 on the Shooter game resulted in score of 347.5 instead of 388.5.

As for the algorithms that use CTW, none achieved a score comparable to CA algorithms. However, as an experiment we collected the number of times that the action chosen by the CA1 algorithm could be predicted by CTW. CTW was given the set of actions and was asked to predict the most likely one. In 74 percent of the times the prediction was the same as the action taken by UCT. While CTW-UCT was not able to exploit this predictive information, it suggests there is further opportunity here. We believe the main problem may be lack of exploration. We designed a very simple test to see if adding more exploration helps us achieving higher scores. We compared CTW-UCT3 and a version of CTW-UCT

Table 5.1: Results on the preliminary game 1, The Shooter. Since the results were not really distinguishable, we ran the best CA algorithm, CA32, and VTS for many more trials in order to find a confidence interval.

	The Shooter	
	Average	Median
CA1	356.02	359
CA2	370.939	367
CA4	371.77	368
CA8	364.682	366
CA16	365.493	355
CA32*	392.074	358
CA64	362.149	332
VTS*	388.504	357
CTW-UCT1	69.111	88
CTW-UCT2	130.864	150
CTW-UCT3	46.42	48
CTW-UCT	51.6	58
random agent	19.28	16

with 0.5 probability of adding CTW macro actions on two Atari games with a very short time span (Freeway and Seaquest). It resulted in increasing the score from 1.59 to 1.8 on Freeway and 115.2 to 250.4 on Seaquest.

5.4.2 Test Games

The results of all algorithms on the 10 Atari games in the test set are shown in Table 5.3.

As mentioned earlier, one way of comparing different algorithms over a set of games is to use normalized scores. Table 5.4 and Figure 5.4 show the average of the inter algorithm scores and also the median. VTS outperforms the vanilla UCT, CA1, but it achieves the average and median score lower than CA2 and CA4.

However, there is an important point: Although VTS did not achieve higher scores than the best CA algorithms on the test games, it performed well on most of the games. Besides, the desired time scale on the preliminary games is different from the desired time scale on the test games. While the preliminary games are supposed to help us tune our algorithm, the best choice based on them (CA32) is outperformed by VTS on the test games. Therefore VTS can be useful when we do not know the desired time scale of the problem. The score distribution plots of VTS and all CA algorithms are shown in Figure 5.5. These curves help us thoroughly analyze the results.

Comparing two different curves, we can talk about different scenarios. If one curve always stays above the other curve, this means its corresponding algorithm has achieved

Table 5.2: Results on the preliminary game 2, The Cookie Lover. Since the results were not really distinguishable, we ran the best CA algorithm, CA32, and VTS for many more trials in order to find a confidence interval.

	The Cookie Lover	
	Average	Median
CA1	114.004	105
CA2	137.002	146
CA4	172.911	182
CA8	179.129	183
CA16	179.244	187
CA32*	179.564	183
CA64	178.831	187
VTS*	183.147	191
CTW-UCT1	46.96	58
CTW-UCT2	61.76	58
CTW-UCT3	55.36	58
CTW-UCT	53.2	58
random agent	28.08	26

Table 5.3: results on the test games

		CA1	CA2	CA4	CA8	CA16	CA32	CA64	VTS
seaquest	average	2727.4	7797.95	14293.1	18572.4	16153.3	16608	15591.6	17345.6
	median	1880	6880	15420	19540	17400	17560	16510	18470
riverraid	average	4638.6	6066.74	5296.05	7978.95	7737.67	6455.12	4197.21	7252.56
	median	4610	5830	5280	8120	8810	6510	4150	6960
ms-pacman	average	12850.7	14969.8	15092	15380.5	15060.2	10396.2	11050.2	13592.4
	median	12751	15161	15841	16941	14741	10621	10711	14571
road-runner	average	3835.82	3972.55	8129.64	4075.09	3329.82	4145.27	4279.64	5616.73
	median	3710	3960	8280	4070	3240	4170	4480	5180
boxing	average	100	100	100	100	100	100	100	100
	median	100	100	100	100	100	100	100	100
enduro	average	187.286	200	200	200	200	194.067	198.733	200
	median	200	200	200	200	200	200	200	200
ice-hockey	average	12	16	16.7143	19.3571	20.9286	20	14.6429	18.2857
	median	12	16	17	20	21	20	15	19
phoenix	average	9929.43	8930.57	6601.14	6090	5868.86	5728.86	3769.06	6427.71
	median	10540	11180	5420	5120	5170	5280	3660	5290
wizard-of-wor	average	6870.87	8464.35	9945	7431.82	6176.96	5210.59	3583.33	8063.81
	median	7090	8470	10330	8170	6630	5650	4310	9470
beam-rider	average	13737.9	14895	13812	14293.3	13574.8	12592	2755.1	13166.9
	median	14940	15360	15300	14910	14020	12576	2160	14000

Table 5.4: The Average and Median scores on the test games

	Average	Median
CA1	0.348449	0.359233
CA2	0.657665	0.580041
CA4	0.762347	0.702524
CA8	0.767934	0.713345
CA16	0.706563	0.626923
CA32	0.49517	0.461855
CA64	0.259703	0.177754
VTS	0.727302	0.662623

higher rewards than the other one. Another case is when one curve, algorithm A, is lower than the other curve, algorithm B, in the left side of the plot and then crosses the other curve somewhere in the middle and stays above until they meet at (1,0). This means algorithm B has achieved both higher and lower scores compared to algorithm A. For comparing such algorithms we can look at the area under the curves as it is the average score. Also, for the X values where the confidence intervals of these curves do not intersect this difference is statistically significant.

5.4.3 Statistical Significance

In this section, we discuss whether the results are statistically significant. In order to do this, we compare the results obtained by VTS to CA1 as it is the vanilla UCT, CA32 as it is the best choice based on the preliminary games, and also we compare VTS to all CA algorithms that have higher or comparable average and median scores: CA4, CA8 and CA16.

This comparison is done by looking at the score distribution curves. We use bootstrapping for generating these curves. By using bootstrapping, we plot many score distribution curves for each algorithm, then for each value on the x axis, we throw away 5 percent from the top and 5 percent from the bottom. Therefore, we find a 90 % confidence interval around the score distribution curve of each algorithm. Subsequently, for all intervals that the confidence intervals do not intersect the difference is statistically significant at a 90 % confidence.

Figure 5.6 shows the two plots of CA1 and VTS and their confidence intervals. As it can be seen, except for the very beginning of the curves the confidence intervals do not share any common area. This is due to the fact that VTS has had better performance than CA1 on most of the games.

Figure 5.10 shows the two plots of CA16 and VTS. The confidence intervals of the two curves most of the time do not intersect. This means CA16 has significantly achieved high scores on more games than VTS. And VTS was more successful in not achieving low scores than CA16. Figure 5.8 and Figure 5.9 also show almost the same situation. However, the difference in the confidence intervals are not as distinct as for CA16.

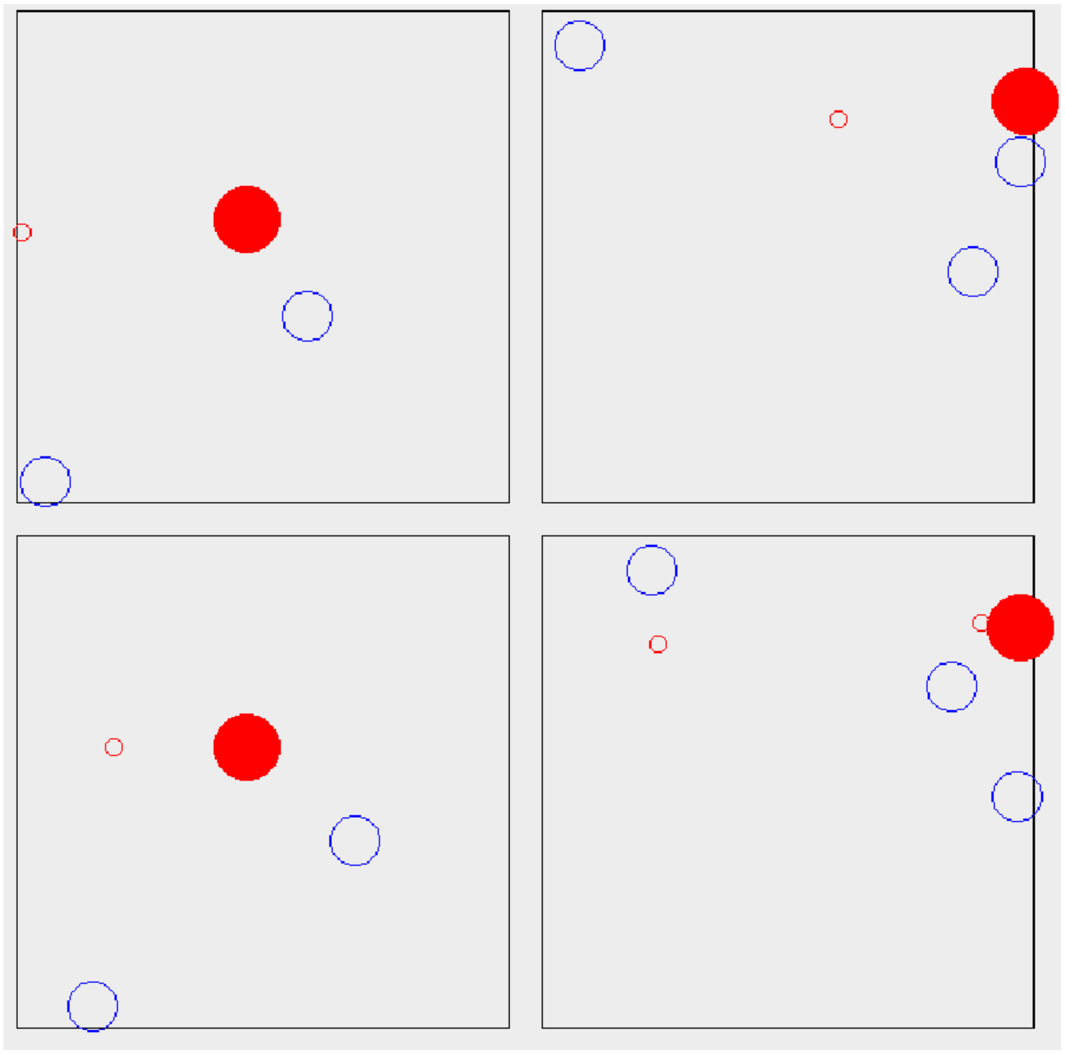

Figure 5.1: Four screenshots of different situations in the Shooter game. All objects in this game are circles. There are two different types of objects: the agent and the enemies. All enemies come from the left side of the screen and they proceed to the right side with different speeds. When the agent shoots a bullet at an enemy, it decreases the speed of the hit enemy. If an enemy touches the agent, the agent dies. The enemies are blue in the screenshots.

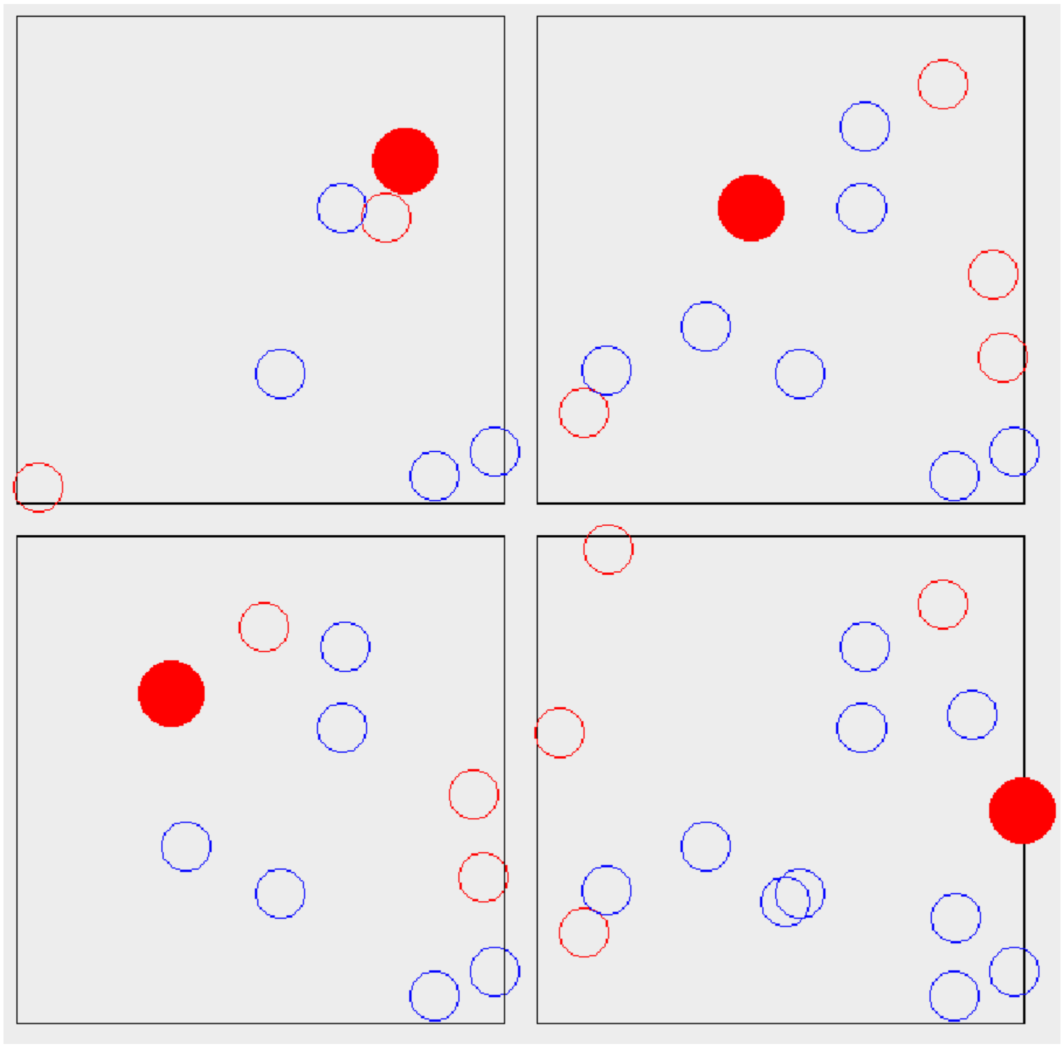

Figure 5.2: Four screenshots of different situations in the Cookie Lover game. All objects in this game are shaped as circles of different radii. Three object types are used in the game: the agent, healthy cookies and poisoned cookies. The goal is to collect as many healthy cookies as possible and to avoid poisoned cookies. The agent walks around the screen trying to accomplish this goal. New cookies are added to the screen and old ones disappear after a while. Healthy cookies are red and poised cookies are blue in the screenshots.

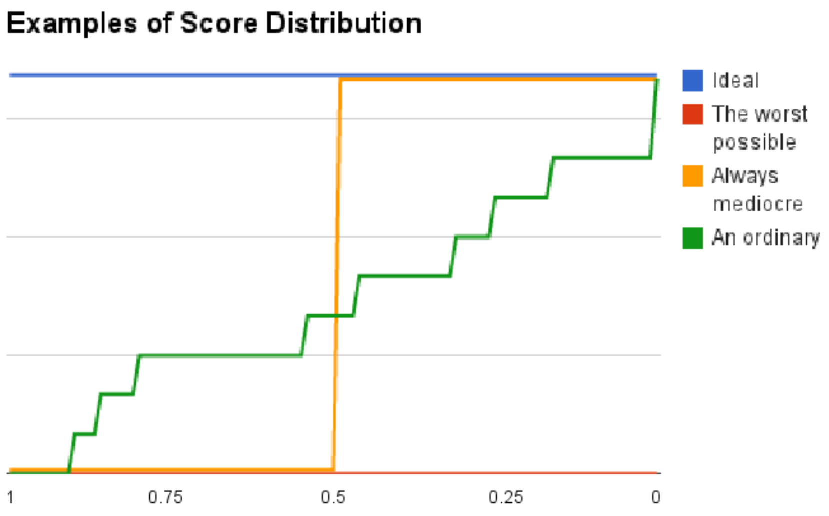

Figure 5.3: Score distribution plot samples: an Ideal algorithm, always mediocre algorithm, the worst possible algorithm, and an ordinary algorithm

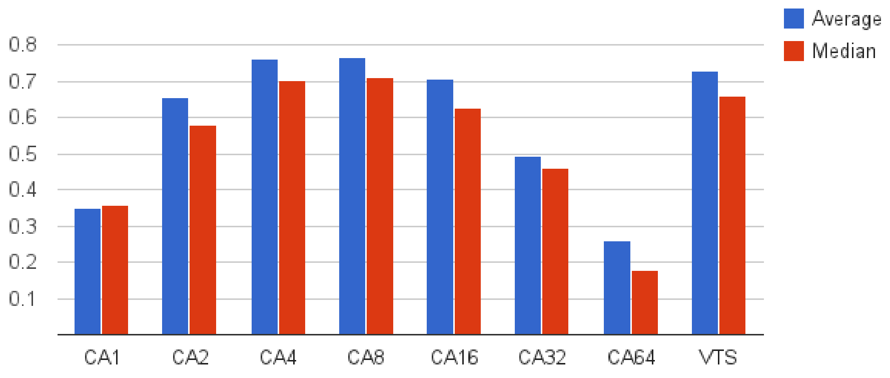

Figure 5.4: The average and the median scores on test games

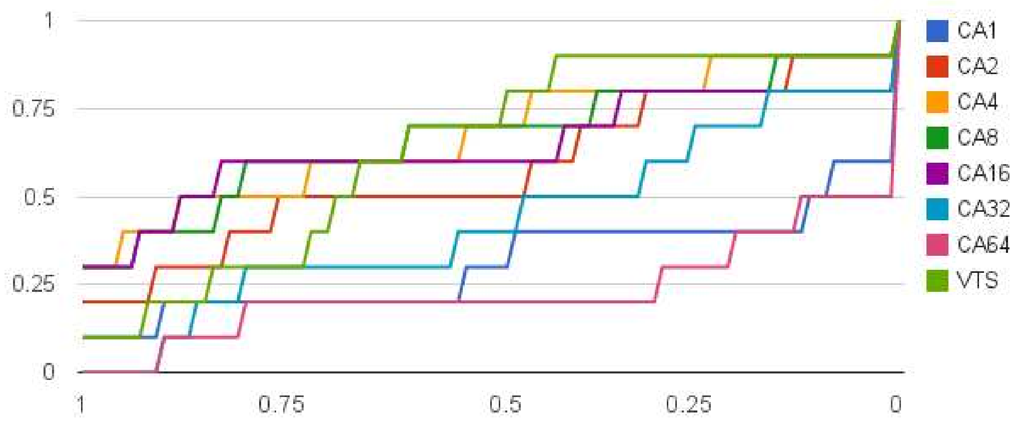

Figure 5.5: The score distribution curves are reverse empirical CDF, for each normalized score x , it shows on how many games the algorithm has achieved x or higher.

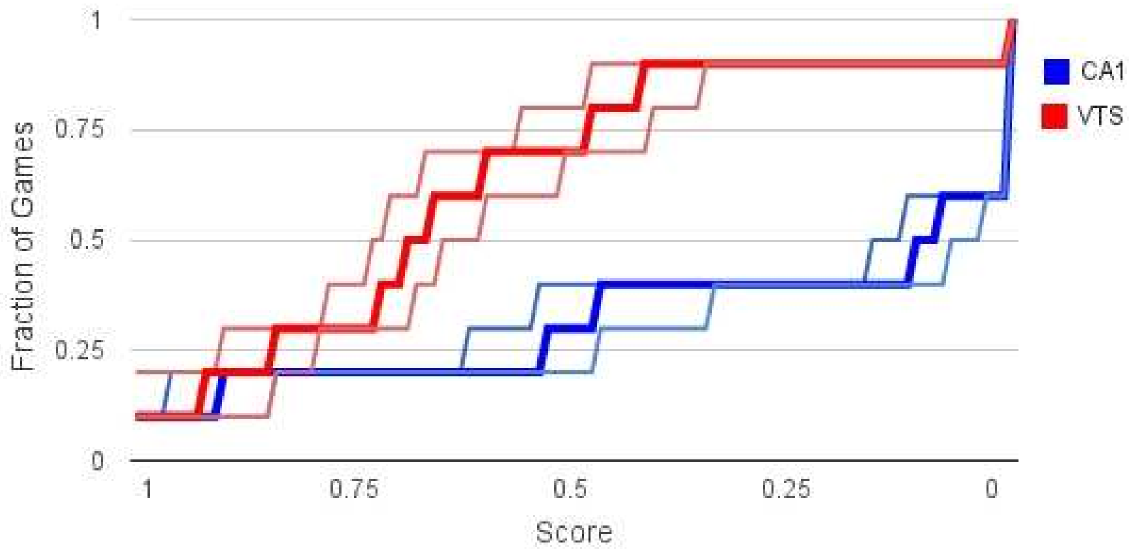

Figure 5.6: The score distribution curves of CA1 and VTS and their confidence intervals. The VTS stays above the CA1 algorithm.

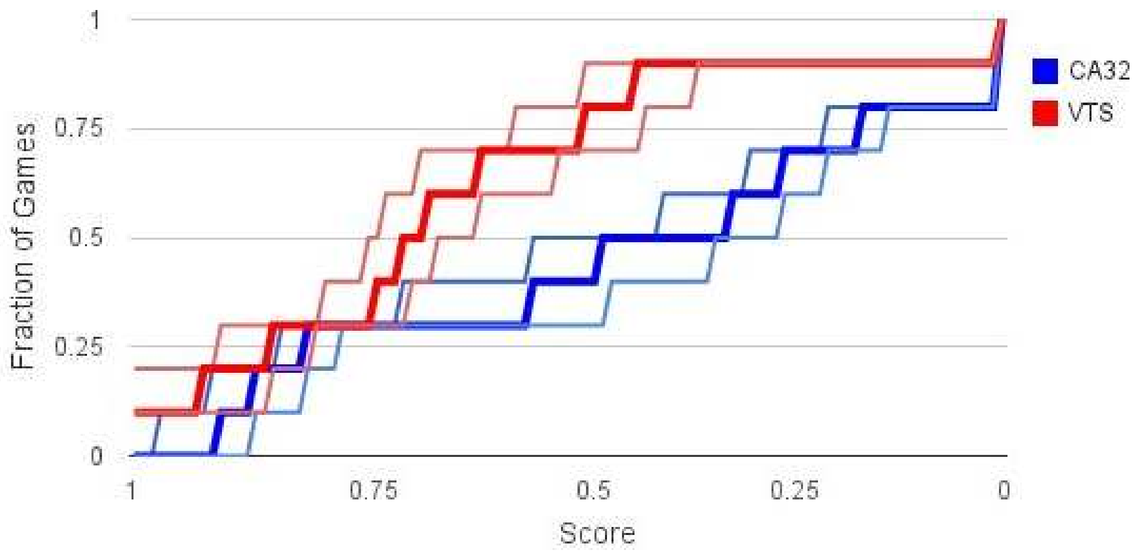

Figure 5.7: The score distribution curves of CA32 and VTS and their confidence intervals. The VTS stays above the CA32 algorithm.

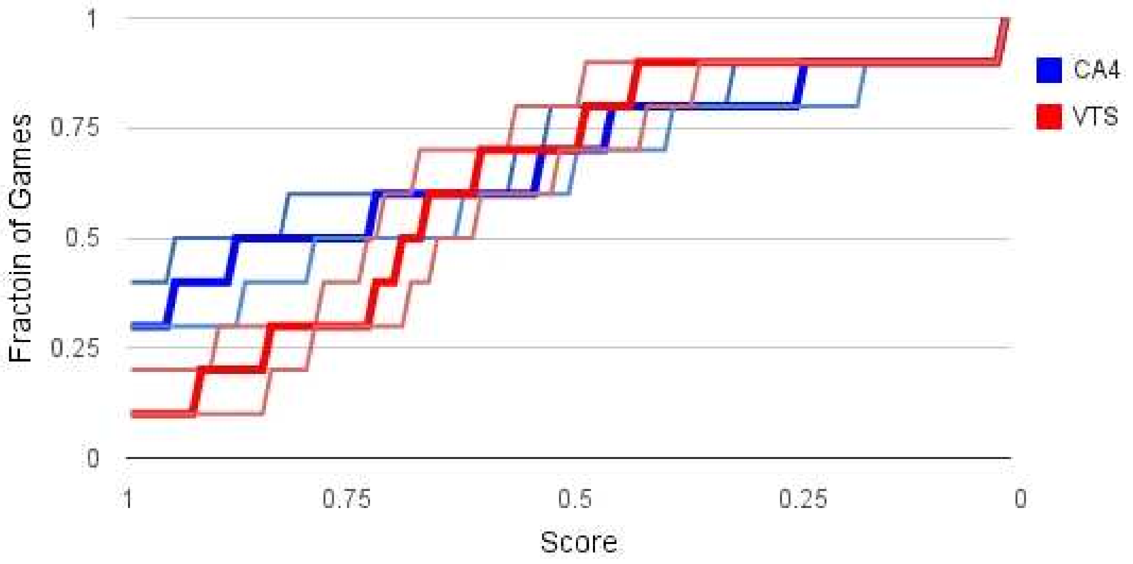

Figure 5.8: The score distribution curves of CA4 and VTS and their confidence intervals.

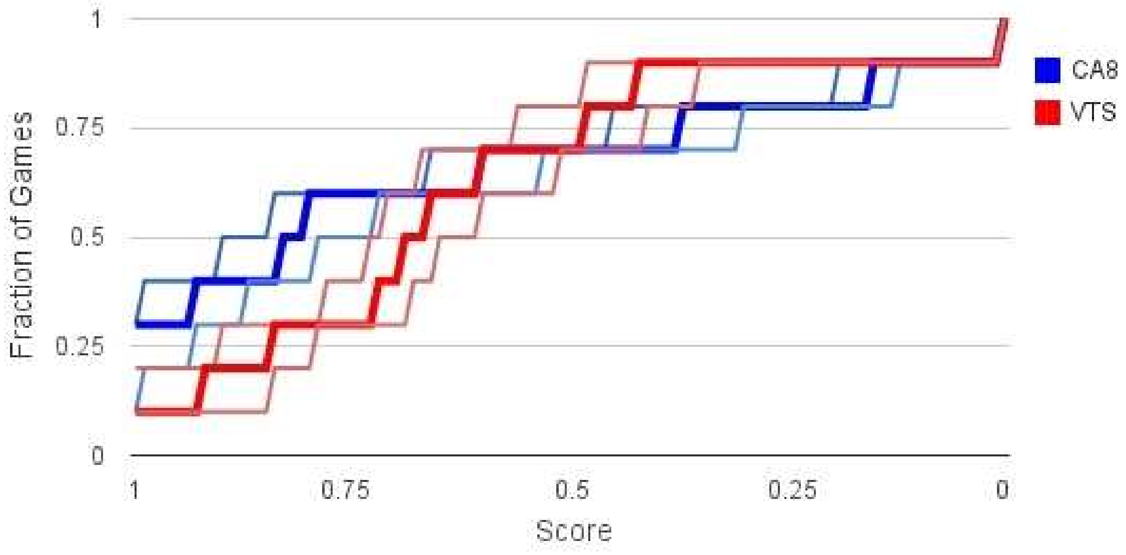

Figure 5.9: The score distribution curves of CA8 and VTS and their confidence intervals.

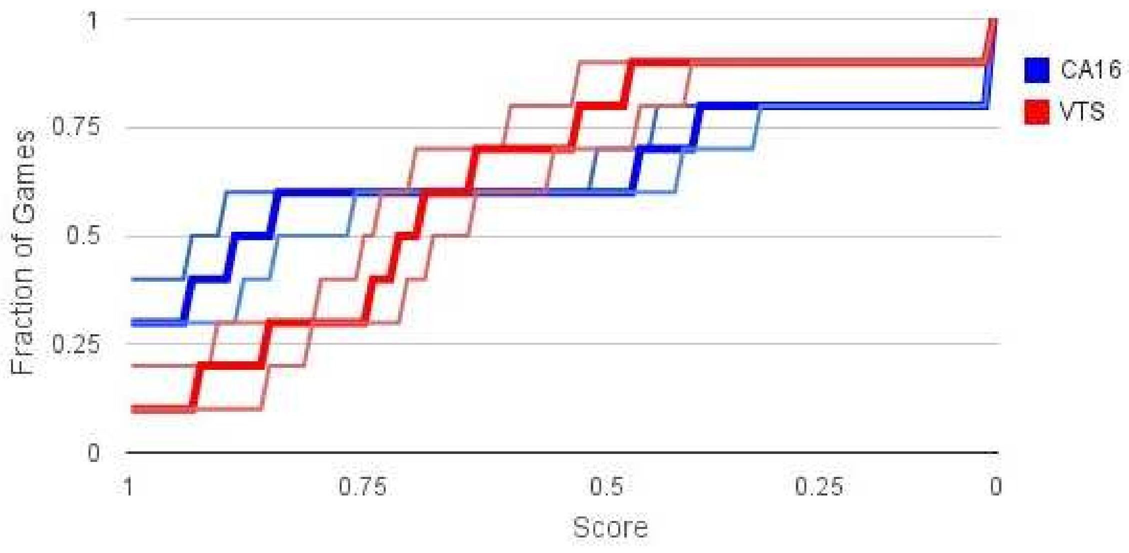

Figure 5.10: The score distribution curves of CA16 and VTS and their confidence intervals.

Chapter 6

Conclusions and Future work

In this thesis we talked about adapting MCTS to better handle temporal abstraction. The motivation is the fact that smart abstraction of the search space can lead to performance improvement. We presented two methods for generating and using macro actions in MCTS algorithms. In the first idea, we used the CTW algorithm to find frequent patterns used in playing a game, and in the second idea we tried to find the best time scale for each action in the search process by modifying the nodes of the search tree. We implemented two simple games as the preliminary games to tune our algorithms on. We then ran the algorithms on 10 Atari games as the test set. We compared these algorithms with different UCT algorithms, each one making use of macro actions of fixed and pre-specified lengths.

In the first idea, we tried to modify MCTS to make use of CTW for learning frequently used patterns that an expert agent takes in different situations of the game. The probability model built by CTW is trained with well played trajectories in order to be able to predict the next most likely actions online. Yet, using this information does not necessarily improve the performance unless we find a mechanism to explore the state space as well. We tried different ideas regarding this issue: 1-using CTW information as a heuristic function to search smarter, 2-using fixed length macro-actions generated by CTW in search, 3- using more likely macro actions according to the model built by CTW in search, 4-randomly choosing between either a random policy or the model built by CTW to generate macro actions for search. None of the above ideas succeeded in an entirely satisfactory way. However, as the model generated by CTW does indeed have useful information, the main problem may be lack of exploration. Although we tried to overcome this problem in the CTW-UCT algorithm, it did not seem to solve the issue. Incorporating the predictive information of CTW over expert trajectories is still an open problem.

As for the second idea, we introduced an algorithm named VTS. VTS tries to find the best time scale for each action and then benefits from using macro-actions of that length in the search process. The VTS algorithm is a modification to MCTS. It uses a new type of node in the search tree which allows different choices of lengths for macro actions in the node's sub-tree. Although, the ideal algorithm that finds the desired time scale during the search should achieve results higher than any CA algorithm, VTS did not achieve as much. On the preliminary games, VTS showed good performance. It ended up first on one of the games with a very small difference to the second place, and on the other game, it was the 2nd best algorithm, following the best CA algorithm with a small difference. However, on the 10 different Atari test games, VTS did not achieve the same level of performance as on the preliminary games.

Although the VTS algorithm did not end up better than the best CA algorithms, it performed above the average on most of the games. As the VTS algorithm was among the top methods on most of the test games, we feel it to be a promising algorithm. Therefore, unless we know the desired time scale in advance for each game, VTS seems a good option.

As a future work to improve the performance of VTS, perhaps the way in which multiplier nodes are added can be improved. Currently, when a multiplier node can get inserted in the tree, it gets added with a fixed probability which may need to be tuned according to the problem.

Bibliography

[1] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk, "Deep thought," *Computers, chess and cognition*, vol. 11, pp. 55–78, 1990.

[2] M. Campbell, A. Hoane, and F. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[3] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook the world man-machine checkers champion," *AI Magazine*, vol. 17, no. 1, p. 21, 1996.

[4] A. Samuel, "Some studies in machine learning using the game of checkers. ," *IBM Journal of Research and Development*, vol. 11, no. 6, pp. 601–617, 1967.

[5] B. Bouzy and T. Cazenave, "Computer Go: an AI oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.

[6] H. Berliner, "Backgammon computer program beats world champion," *Artificial Intelligence*, vol. 14, no. 2, pp. 205–220, 1980.

[7] G. Tesauro, "TD-gammon, a self-teaching backgammon program, achieves master-level play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, 1994.

[8] J. Pearl, "Heuristics: intelligent search strategies for computer problem solving," 1984.

[9] J. Pearl, "Heuristic search theory: Survey of recent results.," in *IJCAI*, vol. 1, pp. 554–562, 1981.

[10] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[11] J. C. Culberson and J. Schaeffer, "Searching with pattern databases," in *Advances in Artifical Intelligence*, pp. 402–416, Springer, 1996.

[12] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald, "Hierarchical a*: Searching abstraction hierarchies efficiently," in *AAAI/IAAI, Vol. 1*, pp. 530–535, 1996.

[13] M. Ernandes and M. Gori, "Likely-admissible and sub-symbolic heuristics,"

[14] B. Abramson, "Expected-outcome: A general model of static evaluation," *Pattern Analysis and Machine Intelligence, IEEE Transactions*, vol. 12, no. 2, pp. 182–193, 1990.

[15] B. Brügmann, "Monte Carlo Go," *White paper*, 1993.

[16] M. Ginsberg, "Gib: Steps toward an expert-level bridge-playing program," in *international joint conference on artificial Intelligence*, vol. 16, pp. 584–593, 1999.

[17] B. Sheppard, "World-championship-caliber scrabble," *Artificial Intelligence*, vol. 134, no. 1, pp. 241–275, 2002.

[18] D. Billings, J. Schaeffer, and D. Szafron, “Using probabilistic knowledge and simulation to play poker,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 697–703, 1999.

[19] B. Bouzy and B. Helmstetter, “Monte-carlo Go developments,” *Advances in computer games*, vol. 10, pp. 159–174, 2003.

[20] R. Coulom, “Efficient selectivity and backup operators in Monte-Carlo Tree Search,” in *Computers and games*, pp. 72–83, Springer, 2007.

[21] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo planning,” *Machine Learning: ECML 2006*, pp. 282–293, 2006.

[22] G. Chaslot, S. De Jong, J.-T. Saito, and J. Uiterwijk, “Monte-carlo tree search in production management problems,” in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pp. 91–98, 2006.

[23] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[24] S. Gelly and D. Silver, “Achieving master level play in 9 x 9 computer Go,” in *Proceedings of AAAI*, pp. 1537–1540, 2008.

[25] S. Gelly and D. Silver, “Combining online and offline knowledge in UCT,” in *Proceedings of the 24th international conference on Machine learning*, pp. 273–280, ACM, 2007.

[26] B. Childs, J. Brodeur, and L. Kocsis, “Transpositions and move groups in Monte carlo tree search,” in *Computational Intelligence and Games, 2008. CIG’08. IEEE Symposium On*, pp. 389–395, IEEE, 2008.

[27] T. Kozelek, “Methods of MCTS and the game arimaa,” *Master’s thesis, Charles University in Prague*, 2009.

[28] G. Chaslot, M. Winands, I. Szita, and H. van den Herik, “Cross-entropy for Monte-carlo tree search,” 2008.

[29] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy, “Progressive strategies for Monte-carlo tree search,” *New Mathematics and Natural Computation*, vol. 4, no. 3, p. 343, 2008.

[30] G. Chaslot, C. Fiter, J. Hoock, A. Rimmel, and O. Teytaud, “Adding expert knowledge and exploration in Monte-Carlo tree search,” *Advances in Computer Games*, pp. 1–13, 2010.

[31] J. Veness, M. Lanctot, and M. Bowling, “Variance reduction in Monte-Carlo tree search,” *Advances in Neural Information Processing Systems*, vol. 24, pp. 1836–1844, 2011.

[32] R. Coulom, “Computing ELO ratings of move patterns in the game of Go,” in *Computer games workshop*, 2007.

[33] R. Sutton, D. Precup, and S. Singh, “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning,” *Artificial Intelligence*, vol. 112, no. 1, pp. 181–211, 1999.

[34] R. I. Brafman and M. Tennenholtz, “Modeling agents as qualitative decision makers,” *Artificial Intelligence*, vol. 94, no. 1, pp. 217–268, 1997.

[35] G. F. DeJong, “Learning to plan in continuous domains,” *Artificial Intelligence*, vol. 65, no. 1, pp. 71–141, 1994.

[36] Y. Naddaf, *Game-independent AI agents for playing atari 2600 console games*. PhD thesis, University of Alberta, 2010.

[37] E. Powley, D. Whitehouse, and P. Cowling, "Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference*, pp. 234–241, IEEE, 2012.

[38] D. Perez, P. Rohlfshagen, and S. Lucas, "The physical travelling salesman problem: WCCI 2012 competition," in *Evolutionary Computation (CEC), 2012 IEEE Congress*, pp. 1–8, IEEE, 2012.

[39] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo Tree Search methods," *Computational Intelligence and AI in Games, IEEE Transactions*, vol. 4, no. 1, pp. 1–43, 2012.

[40] H. Finnsson, *Cadia-player: A general game playing agent*. PhD thesis, Masters thesis, Reykjavik University, 200 7. http://www. ru. is/faculty/yngvi/hif-MSc-thesis. pdf, 2007.

[41] J. Mehat and T. Cazenave, "Monte-Carlo Tree Search for general game playing," *Univ. Paris*, vol. 8, 2008.

[42] M. Möller, M. Schneider, M. Wegner, and T. Schaub, "Centurio, a general game player: Parallel, java-and asp-based," *KI-Künstliche Intelligenz*, vol. 25, no. 1, pp. 17–24, 2011.

[43] C. Luckhart and K. B. Irani, "An algorithmic solution of n-person games.," in *AAAI*, vol. 86, pp. 158–162, 1986.

[44] B. W. Ballard, "The *-minimax search procedure for trees containing chance nodes," *Artificial Intelligence*, vol. 21, no. 3, pp. 327–350, 1983.

[45] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron, "Game-tree search with adaptation in stochastic imperfect-information games," in *Computers and Games*, pp. 21–34, Springer, 2006.

[46] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *arXiv preprint arXiv:1207.4708*, 2012.

[47] S. Russell, P. Norvig, E. Davis, S. Russell, and S. Russell, *Artificial Intelligence: a modern approach*. Prentice hall Upper Saddle River, NJ, 2010.

[48] C. Szepesvári, "Reinforcement learning algorithms for mdps," *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[49] M. Kearns, Y. Mansour, and A. Ng, "A sparse sampling algorithm for near-optimal planning in large Markov Decision Processes," in *International Joint Conference on Artificial Intelligence*, vol. 16, pp. 1324–1331, LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

[50] D. Bertsekas and D. Castanon, "Rollout algorithms for stochastic scheduling problems," *Journal of Heuristics*, vol. 5, no. 1, pp. 89–108, 1999.

[51] J. Veness, K. Ng, M. Hutter, W. Uther, and D. Silver, "A Monte-carlo AIXI approximation," *Journal of Artificial Intelligence Research*, vol. 40, no. 1, pp. 95–142, 2011.

[52] Y. Wang and S. Gelly, "Modifications of UCT and sequence-like simulations for Monte-Carlo Go.," *CIG*, vol. 7, pp. 175–182, 2007.

[53] A. Elo, *The rating of chessplayers, past and present*, vol. 3. Batsford London, 1978.

[54] F. Willems, Y. Shtarkov, and T. Tjalkens, “The context-tree weighting method: Basic properties,” *Information Theory, IEEE Transactions*, vol. 41, no. 3, pp. 653–664, 1995.

[55] R. Begleiter, R. El-Yaniv, and G. Yona, “On prediction using variable order Markov models,” *J. Artif. Intell. Res. (JAIR)*, vol. 22, pp. 385–421, 2004.

[56] K. Sadakane, T. Okazaki, and H. Imai, “Implementing the context tree weighting method for text compression,” in *Data Compression Conference, 2000. Proceedings. DCC 2000*, pp. 123–132, IEEE, 2000.

[57] G. Theocharous and L. Kaelbling, “Approximate planning in POMDPs with macro-actions,” *Advances in Neural Processing Information Systems*, vol. 17, 2003.

[58] A. Coles and A. Smith, “Marvin: A heuristic search planner with online macro-action learning,” *Journal of Artificial Intelligence Research*, vol. 28, no. 1, pp. 119–156, 2007.