Masaryk University
Faculty of Informatics

Monte Carlo Tree Search in Verification of Markov Decision Processes

Master's Thesis

Ondřej Slámečka

Brno, Fall 2017

Masaryk University
Faculty of Informatics

Monte Carlo Tree Search in Verification of Markov Decision Processes

Master's Thesis

Ondřej Slámečka

Brno, Fall 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ondřej Slámečka

Advisor: doc. RNDr. Tomáš Brázdil, Ph.D.

Acknowledgement

I would like to thank my advisor Tomáš Brázdil for his guidance and invaluable advice, Pranav Ashok with whom I thoroughly enjoyed working on this research, and Jan Křetínský for his amazing insights.

My deepest gratitude goes to my family for their support during my studies. I would also like to thank all my friends for their patience with me in the last five years.

My research was supported by project MUNI/C/1367/2015.

Abstract

We explore how Monte Carlo tree search type algorithms can be used for verification of Markov decision processes (with complete information) due to their balance between exhaustive and heuristic search. Several new algorithms either based on UCT (the most successful MCTS algorithm) are proposed and experimentally evaluated on standard models. Our results show there are standard models where they outperform previously known methods. Moreover, we show the suggested algorithms perform very well on models where BRTDP underestimates certain paths to goals.

Keywords

Monte Carlo Tree Search, Markov Decision Process, Verification, Reachability, Learning Algorithm

Contents

1 Introduction

Zero-configuration networking protocol is used to automatically set up computer networks without the need for external help. The computers select their IP addresses in the created network randomly with the goal to achieve a working configuration.

Another protocol is IEEE 802.11 Wireless LAN for communication of wireless network devices. Part of the protocol describes what to do when two signals are sent simultaneously and collide. The senders then use random delays to avoid subsequent collisions.

Scientists model these and many similar real-world problems with *Markov Decision Processes* (MDPs). Controllers in these processes take actions of their choice but the results of these actions are not certain – in a zero-conf network a chosen IP address might be available with high probability or taken with small, a signal in a wireless LAN might get delivered or it might collide with another signal.

Various properties of Markov Decision Processes have been well studied in the last seventy years. The usual target is to attain the highest reward from the process, e.g., research as much as possible on Mars [1]. However, in many situations, it is desirable to learn what is the probability the process succeeds (network configures) or fails (signal is not delivered) – to encourage actions increasing the likelihood of success, or avoid them for the negative case.

Classical algorithms for verification (proving properties) of MDPs based on dynamic programming can be used for finding the maximum probability of success or failure (maximum over all possible decisions in each step – strategies). Under some circumstances, these algorithms are very good but recently it has been shown that learning based methods like BRTDP can outperform them on many MDPs [2].

Monte Carlo Tree Search is a heuristic search algorithm which has been successfully used to find strategies with high rewards in Markov decision processes. Recently the algorithm had a big success in the field of computer Go.

In this thesis, we explore how Monte Carlo Tree Search can be used to find strategies maximising given properties. Three algorithms are suggested: one a variation of Upper Confidence Tree (UCT) algorithm (a variant of MCTS), one a fusion of UCT and BRTDP and the last one a BRTDP variant using a part of the UCT algorithm idea. Measurements show that these algorithms have an advantage on several models.

The thesis is structured as follows. After this introduction, the second chapter is devoted to Markov Decision Processes and prior work on their verification. The third chapter describes Monte Carlo Tree Search and its application to maximising rewards in MDPs and games. In the fourth chapter our new algorithms are described. In the fifth chapter, the algorithms are evaluated on models available with the PRISM project as well as models newly created specifically for better comparison. The sixth chapter concludes the results with suggestions for future work.

The results in the fourth and fifth chapters are original results of a collaboration of Pranav Ashok, Tomáš Brázdil, Jan Křetínský and the author of this thesis. The MCTS-BRTDP algorithm was suggested by the author of this thesis and the BMCTS and BRTDP-UCB algorithms by Pranav Ashok. The algorithms were implemented as part of the PRISM model checker mostly when pair programming with Pranav Ashok.

2 Markov Decision Processes

At first, we look at Markov chains, a simpler type of probabilistic processes which do not offer choices to be made. Then we generalize to Markov Decision Processes. After that, we introduce the problem of *verification* (checking if a given MDP satisfies a given property), overview known solutions with a focus on value iteration and Bounded Real-Time Dynamic Programming (which will be used as a part of new algorithms in later chapters).

It is assumed the reader is familiar with basic notions of probability theory, namely those of *probability space* and *probability measure*. Function $f : X \to [0,1]$ is a *probability distribution* over a countable set X if $\sum_{x \in X} f(x) = 1$. $\mathcal{D}(S)$ denotes the set of probability distributions on a set S. Usually the probability space $(\Omega, \mathcal{F}, P)$ is implicitly known and we use $P(E)$ to denote the probability of an event E. We refer the reader to [3] for a proper treatment of probability theory.

2.1 Markov Chains

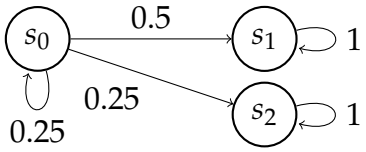
Markov chains are a simpler formalism than Markov decision process and provide a good first intuition about probabilistic models. In a Markov chain, there are no decisions, only probabilities of transition.

Definition 1. Let S be a finite set of states. A sequence of random variables $(X_i : S \to \{0,1\})_{i=0}^{\infty}$ is called a *finite discrete-time time-homogeneous Markov chain*, if the probability of moving to a state is given only by the current state, that is[1] $P(X_{k+1} = s_{k+1} \mid X_k = s_k) = P(X_{k+1} = s_{k+1} \mid X_k = s_k, X_{k-1} = s_{k-1}, \ldots, X_1 = s_1)$ for every $s_k \in S$, if the conditional probabilities are defined, and $P(X_{k+1} = s_{k+1} \mid X_k = s_k) = P(X_k = s_k \mid X_{k-1} = s_{k-1})$.

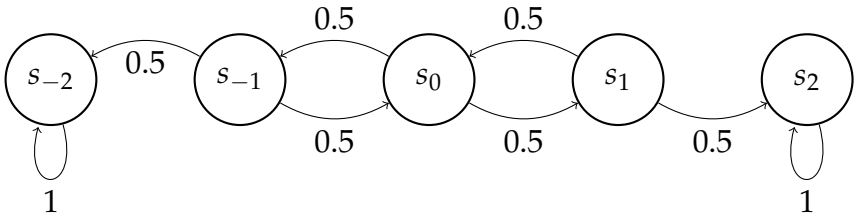
In many applications it is natural to have an initial state s ($X_1 = s$) and a chain can then be drawn as a graph with probabilistic transitions. Note that the weights of the outgoing edges from a node sum to one, as the transition probabilities form a distribution on the set of states (for simplicity we do not draw the transitions with zero probability).

1. The first condition ensures that only the last state of "history" is considered, the second that the length of the history does not matter either.

Example 1. In this example s_0 is the initial state, $P(X_2 = s_1 \mid X_1 = s_0) = 0.5$, $P(X_2 = s_2 \mid X_1 = s_0) = 0.25$, $P(X_2 = s_0 \mid X_1 = s_0) = 0.25$, $P(X_{i+1} = s_1 \mid X_i = s_1) = 1$, $P(X_{i+1} = s_2 \mid X_i = s_2) = 1$, for all $i \in \mathbb{N}$.



Example 2. The Drunkard's Walk is a well-known example of a Markov chain. One can imagine a drunk person starting in the middle of a road (state s_0) and then moving left or right at random. How many times will the drunk visit the middle of the road? How many steps will it take the drunk on average to reach a ditch (s_{-2}, s_2)?



2.2 Markov Decision Processes

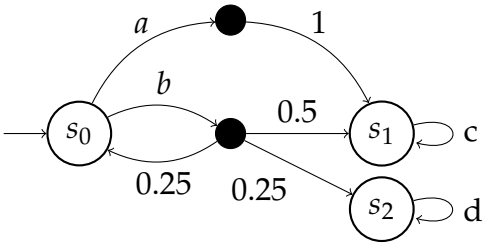
Markov decision processes are similar to Markov chains, except now a controller interacting with the process can in each state pick an action and the next state is chosen according to a distribution on states corresponding to this action.

Definition 2. A *Markov Decision Process* is a tuple (S, s_0, A, E, Δ), where S is a finite set of states, $s_0 \in S$ is the initial state, A is a finite set of actions, $E : S \to \mathcal{P}(A)$ gives the set of enabled actions in a state, and $\Delta : S \times A \to \mathcal{D}(S)$ is a partial transition function which assigns a probability distribution on states to an action and a state.

It is assumed without loss of generality that for all $s \neq s'$ it holds that $E(s) \cap E(s') = \emptyset$. If this did not hold the actions could just be renamed.

Example 3. The MDP $(\{s_0, s_1, s_2\}, s_0, \{a, b\}, E, \Delta)$, where $E(s_0) = \{a, b\}$, $E(s_1)\{c\}, E(s_2) = \{d\}$, and $\Delta(s_0, a) = \{(s_1, 1)\}$, $\Delta(s_0, b) = \{(s_1, 0.5), (s_2, 0.25), (s_0, 0.25)\}$, $\Delta(s_1, c) = \{(s_1, 1)\}, \Delta(s_2, d) = \{(s_2, 1)\}$ is depicted below. The edges labeled with letters denote the available actions and lead to smaller black dots, which mark the point of random choice of the successor state. With actions c, d the black dots are omitted and the transition probability 1 is understood implicitly.

The controller making decisions should choose action a if they want to get to s_1, or (possibly repeatedly) choose b if they want to get to s_2 (the achievement of this goal is not guaranteed).



A standard use of Markov decision processes has been in areas where it is useful to operate with rewards.

Definition 3. If (S, s_0, A, E, Δ) is a Markov decision process, and $R : S \times A \times S \to \mathbb{R}$ is a function, then $(S, s_0, A, E, \Delta, R)$ is a *Markov decision process with rewards*.

Definition 4 (Path)**.** An *infinite path* is a sequence $\omega = s_0 a_0 s_1 a_1 \ldots$ such that $a_i \in E(s_i)$ for all $i \in \mathbb{N}$. The set of all infinite paths of is denoted $IPaths$.

A *finite path* is a prefix of an infinite path such that it ends with a state. The last state for a finite path ρ is denoted last(ρ). The set of all finite paths is denoted $FPaths$.

When following a path one wants to avoid getting stuck in an infinitely repeated cycle of states and actions. The parts of MDP where this happens are called end components. An example end component is shown in Figure 2.1.

Definition 5 (End component)**.** Let $\mathcal{M} = (S, s_0, A, E, \Delta)$ be an MDP, let $S' \subseteq S$ and $A' \subseteq \bigcup_{s' \in S'} E(s')$. The pair (S', A') is an *end component*, if for every $s \in S', s' \in S, a \in A'$ it holds that

$$\Delta(s, a)(s') > 0 \implies s' \in S'$$

and there is a path between every two $s, s' \in S'$ using only actions A'.

An end component is *maximal* if it is maximal with respect to the point-wise ordering of subsets.

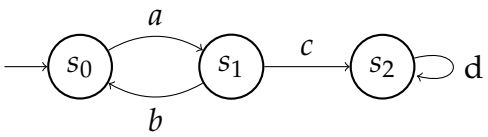


Figure 2.1: An MDP with a non-trivial end component.

The following definition introduces what is commonly called strategy, policy, scheduler, controller or adversary. It is the decision maker in the MDP which looks at the path traversed so far and assigns each action a in the last (current) state the probability of a being chosen.

Definition 6 (Strategy)**.** Let $\mathcal{M} = (S, A, E, \Delta)$ be a Markov decision process and $\rho \in FPaths$. A *strategy* is a function $\sigma : FPath \to Dist(A)$ such that[2] $\sigma(\rho)(a) > 0 \implies a \in E(\text{last}(\rho))$.

A strategy σ is *memoryless* if $\sigma(\rho)$ depends only on $\text{last}(\rho)$. A strategy σ is *deterministic* if $\sigma(\rho)(a) = 1$ for some $a \in E(\text{last}(\rho))$.

What happens once the strategy is fixed? The MDP becomes a Markov chain. This can be seen in the examples, e.g. if the strategy is to always choose b, the MDP from Example 3 becomes the MC from Example 1. Importantly when a strategy is fixed there is a probability measure[3] denoted $P^{\sigma}_{\mathcal{M},s}$ over the set of $IPaths$ with an initial state s, which assigns to a set of paths the probability they will be traversed.

2. The condition ensures that only actions which can be chosen have non-zero probability assigned by the strategy.
3. While this may be intuitive it is not a trivial statement, see Theorem 2.4 in [4] for a formal treatment.

Completeness of information about an MDP. One can distinguish between various levels of knowledge about an MDP, usually, depending on the source of the model. Complete information corresponds to knowing every part of the definition of an MDP, as opposed to limited information where the MDP can only be used as a black box from which can information be extracted by sampling. This thesis is concerned only with MDPs with complete information.

2.3 Verification

Formal verification is the act of proving that a given system satisfies a given property. *Model checking* is an approach to formal verification which uses a model of the system to verify the property. In our case, the systems are modeled as Markov decision processes and the properties are about the maximum probability of reaching a state.

To simplify notation, in this section we often refer to a fixed MDP $\mathcal{M} = (S, s_0, A, E, \Delta)$, and a set of target states F.

Reachability probability. Let $\Diamond F$ be the set of all infinite paths that reach a state in F. In this thesis we are concerned with maximising the reachability probability $P^{\sigma}_{\mathcal{M},s_0}(\Diamond F)$ over the set of all strategies σ. We define the *value function* $V(s) = \sup_{\sigma} P^{\sigma}_{\mathcal{M},s}(\Diamond F)$ for every state $s \in S$.

Importantly there is always a memoryless, deterministic strategy maximising the reachability probability. This was proven by Puterman [5] for reward maximization and the proof can be used with a simple reduction for the case of verification [6]. For any MDP create an MDP with rewards, such that the reward function is zero everywhere, except when entering a target state it gives reward 1 and then transitions into a new sink state (with reward 0).

In this section, we describe value iteration, which is a standard algorithm for computing the maximum probability of reaching a state in F. In the next section, a more involved algorithm called BRTDP is described.

One approach not explained here is so-called *strategy iteration* which starts with an arbitrary strategy and gradually improves it as long as a change to the strategy is beneficial. Strategy iteration has a clear stopping criterion as opposed to value iteration, however, in practice, it does not perform better [7].

Another approach we do not explain is formulating the problem as a system linear equations and solving it. The solution is precise and has guaranteed correctness, but the computation quickly becomes expensive as the model grows in size. See [7] for a description of this method.

Value Iteration

Value iteration (in its variation for computing expected maximum rewards) is a dynamic programming algorithm which was first described by Richard Bellman[4] [8] in 1957. We present its variation for computing the maximum reachability probability.

Before showing the algorithm, we first note that it will need to process all states. However, the values in some states are quite easy to compute and a preprocessing will allow for reduction of the state space. These easy states are in set $Z \subseteq S$ of *zero states* or set $F' \subseteq S$ of *extended target states*. States $z \in Z$ are such that $P^{\sigma}_{\mathcal{M},z}(\Diamond F) = 0$ holds, and states $f \in F'$ are such that $P^{\sigma}_{\mathcal{M},f}(\Diamond F) = 1$ holds, both for any strategy σ. Their computation is rather straightforward [7] (Section 4.1). Note that $F \subseteq F'$.

The main idea of value iteration is materialized in the following recurrence relation for newly introduced variables $x^n_s, s \in S, n \in \mathbb{N}$.

$$x^n_s = \begin{cases} 1 & \text{if } s \in F' \\ 0 & \text{if } s \in Z \vee (s \notin F' \wedge n = 0) \\ \max\limits_{a \in E(s)} \sum\limits_{s' \in S} \Delta(s,a)(s') \cdot x^{n-1}_{s'} & \text{otherwise} \end{cases}$$

By computing x^n for $n = 1, 2, \ldots$ we gain increasingly precise estimate of the actual maximum reachability probability, formally $\lim_{n\to\infty} x^n_s = P_{\mathcal{M},s}(\Diamond F)$. We refer to a proof of this statement in [5] with the same reduction as in the case of existence of memoryless optimal strategy.

This recurrence relation is now turned into a dynamic programming algorithm as shown in Algorithm 1. Instead of iterating n times, the algorithm proceeds with its computations while the convergence is not slow (the threshold is given by some ϵ).

4. Known for introducing the term dynamic programming.

Algorithm 1: Value Iteration

1: $\forall s \in S,\ s \leftarrow 1$ if $x \in F$ else 0
2: **do**
3: **for each** $s \in S \setminus (F \cup Z)$ **do**
4: $x_s := \max_{a \in E(s)} \sum_{s' \in S} \Delta(s,a)(s') \cdot x_{s'}$
5: **while** the change of x_s for any s is greater than ϵ

Value iteration is an easy method for computing the reachability probability of a given MDP. However, we only have a proof of convergence and not a useful stopping criterion. Furthermore, it is doing extra work on models where only a small part needs to be explored to find a good strategy as it has to compute its results for all the states.

The solution to the first issue is the *interval iteration* algorithm [9], an algorithm in parts similar to value iteration but which maintains lower and upper bounds of the sought probability. The algorithm has a well-defined stopping criterion and a bound on the running time. We will explore solutions to the second issue later with heuristic methods.

2.4 Bounded Real-Time Dynamic Programming

When only a portion of an MDP needs to be searched to find the right strategy there is an opportunity to employ algorithms which avoid searching the whole state space. Bounded Real-Time Dynamic Programming (BRTDP) is such an algorithm. Originally developed for the objective of finding the best-profit strategy [10] the algorithm was adapted to the problem of verification [2].

As in the previous section, we refer to a fixed MDP $\mathcal{M} = (S, s_0, A, E, \Delta)$, and a set of target states F. Further we fix ϵ, an argument of the algorithm which sets the required precision (maximum allowed distance of the result of the algorithm from the correct value).

Recall we have a unary value function defined and for use in the algorithm let us define the binary *value function* $V : S \times A \to [0,1]$ for all $s \in S$ and $a \in E(s)$ as follows

$$V(s,a) := \sum_{s' \in S} \Delta(s,a)(s')V(s')$$

This serves to represent the value in s after taking action a. BRTDP is learning V by monotonically tightening its lower and upper bounds $L, U : S \times A \to [0,1]$ during simulated runs of $\mathcal{M}$ from the given initial state. When $\max_{a \in E(s_0)} U(s_0,a) - \max_{a \in E(s_0)} L(s_0,a) < \epsilon$ the algorithm terminates.

We start with an important assumption, that that $\mathcal{M}$ does not contain any end component besides two trivial end components, one containing only the target state 1 with $F = \{1\}$, the other only the state 0 with $V(0) = 0$.

With this assumption BRTDP can be implemented as Algorithm 2, alternating between simulation and update phases until it has sufficiently good knowledge about $V(s_0)$. In the simulation phase the, algorithm samples a finite path from the initial state to one of states $\{1,0\}$, each time using an action maximising the upper bound. In the update phase, the algorithm traverses the path backwards and performs the Bellman update (known from value iteration), using the best-known bounds, i.e. $U(s) = \max_{a \in E(s)} U(s,a)$ and $L(s) = \max_{a \in E(s)} L(s,a)$. With the assumption of no non-trivial BRTDP converges almost surely to the correct value [2], that is the actual probability almost always lies between $L(s_0), U(s_0)$, and the bounds converge such that they are at most ϵ (for a given ϵ) far from each other.

Algorithm 2: BRTDP for MDPs without end components

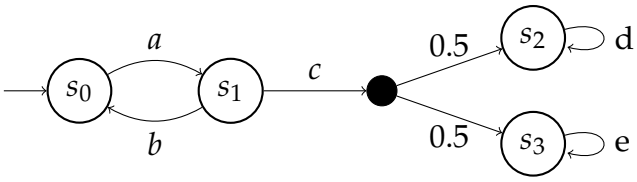
$U(s,a) \leftarrow 1, L(s,a) \leftarrow 1 \; \forall s \in S, a \in E(s)$
$U(0,a) \leftarrow 0, L(1,a) \leftarrow 1 \; \forall a \in A$
$\omega \leftarrow s_0$
while $U(s_0) - L(s_0) < \epsilon$ **do**
    # Simulation Phase
    while $last(\omega) \notin \{0,1\}$ **do**
        $a \leftarrow$ sample uniformly from $\operatorname{argmax}_{a \in E(last(\omega))} U(last(\omega), a)$
        $s \leftarrow^{a}$ sample according to $\Delta(last(\omega), a)$
        $\omega \leftarrow \omega \; a \; s$
    # Update Phase
    while ω is not empty **do**
        $pop(\omega)$
        $a \leftarrow pop(\omega)$
        $s \leftarrow last(\omega)$
        $U(s,a) := \sum_{s' \in S} \Delta(s,a)(s') U(s')$
        $L(s,a) := \sum_{s' \in S} \Delta(s,a)(s') L(s')$
return $(U(s_0), L(s_0))$

a. The implementation of this line may vary, see Subsection "Variants of BRTDP"

BRTDP for MDPs with End Components

Algorithm 2 is not guaranteed to converge when the MDP contains non-trivial end components.

Example 4. Below is an MDP with an end component $(\{s_0, s_1\}, \{a, c\})$, and let $F = \{s_2\}$. When BRTDP updates state s_0 or s_1 there is always a state (the other one in the EC) which has upper bound 1. This way the upper bound remains 1 after every iteration, even though the lower bound is correctly 0.5. The algorithm does not converge for $\epsilon < 0.5$.



Fortunately, there is an on-the-fly method[5] for resolving the problem in BRTDP [2], which we present to an extent important for our future use of BRTDP.

During the simulation phase, the algorithm periodically (every k_i steps) creates an auxiliary MDP based on the states visited so far and their neighbours. The algorithm then identifies maximal ECs in this auxiliary MDP.

If an end component is found it is collapsed, i.e. the states are merged into a single state and functions E, Δ are naturally transformed to work the same way in the modified MDP.

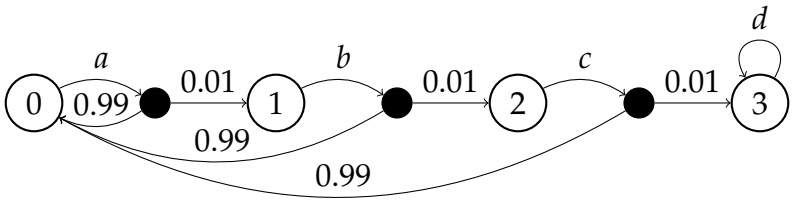
Finally, the merged state has its U, L updated for every action in the end component. If there was a target state in the end component then the merged state is marked as a target state. If there was no target state and there is no outgoing action from the merged state then it is marked as a zero state.

5. The knowledge we have about the MDP during computation suffices – knowing the whole MDP is not necessary.

Variants of BRTDP

When BRTDP is in state s and it has chosen action a, the choice of the next state does not necessarily have to be done by sampling the transition distribution $\Delta(s, a)$ (we call this variant HIGH-PROB) but can be instead chosen with probability $\Delta(s, a)(s') \cdot (U(s') - L(s'))$, we call this variant MAX-DIFF. One can think of other variants, for example round-robin choice.

Example 5. We conclude with an example MDP, which is hard for most variants of BRTDP when the target is the final state. For example the HIGH-PROB variant has probability 0.99 of returning to the initial state from every state until reaching the final state.



3 Monte Carlo Tree Search

Monte Carlo methods[1] are using random sampling to estimate a correct solution to a problem. The first practical use of Monte Carlo methods was by Stanislaw Ulam and John von Neumann during their work on the Manhattan project, but the technique has since spread into many areas of science due to its general applicability.

One of the celebrated Monte Carlo methods is the simulated annealing algorithm (so called due to its origin in statistical physics) which is an improved version of Metropolis algorithm (invented by a Manhattan project scientist Nicholas C. Metropolis and others).

This method found its way into game theory in 1993 when it was applied to the board game Go [11]. The approach was later further improved [12], but the real breakthrough came in 2006 when Coulom [13] and Kocsis, Szepesvári [14] independently explored the idea of maintaining a tree which would guide the search for strategies – thus discovering Monte Carlo Tree Search. This was eventually used in the AlphaGo program [15], the first computer program to beat professional human players.

Monte Carlo Tree Search (MCTS) is, in short, a heuristic search algorithm for finding good strategies in complex decision processes by combining standard approaches of artificial intelligence and computational statistics: tree search and sampling.

In this chapter, the general MCTS scheme is defined, and a concrete instance called UCT is shown together with its application to maximizing rewards in MDPs and games. The chapter is based mostly on a thorough MCTS survey paper [16].

Since the research into MCTS focuses mainly on using MCTS for reward maximization, we demonstrate the algorithms on the reward maximization problem too in this chapter unlike the rest of the thesis.

1. Not to be confused with Monte Carlo algorithms which are precisely defined as the algorithms solving the decision problems in classes BPP and RP.

3.1 General MCTS Scheme

MCTS iteratively builds a tree which approximates possible rewards attainable by strategies in the decision process. In each iteration, the tree guides the search to balance between exploitation of known good strategies and exploration of new strategies. When the search leaves the tree, it adds one layer of new nodes, proceeds at random from one of the new nodes, and upon terminating it updates the ancestors of the node with the result. The algorithm can be splt into four stages:

1. *Selection* – TreePolicy traverses the tree until a leaf is reached. Often a greedy choice of node maximising a heuristic value.
2. *Expansion* – if the leaf's corresponding state is not terminal, then nodes for its succcessors are added to the tree.
3. *Simulation* – traverse the MDP from state of one of the new nodes in a random manner until a terminal state is reached.
4. *Backpropagation* – update the path from the selected node to the root with the result of the simulation.

Algorithm 3: General Monte Carlo Tree Search method

```
1: function MCTS(s_0)
2:     Let v_0 be the root of the MCTS tree, with v_0.state = s_0.
3:     while within computational budget do
4:         v_l ← TreePolicy(v_0)
5:         r ← Rollout(v_l)
6:         Backup(v_l, r)
7:     return Action from v_0 to the best node (by some given metric).
```

Node v_l is a leaf of the tree selected by the tree policy. The leaf may be expanded – the corresponding state's successors in the MDP are then added as leafs. A simulation (rollout) then starts from a new leaf. r is the reward gained by this rollout.

After the algorithm is interrupted and asked for a result or it runs out of time, there are various ways how to choose an action in the end, for example, the most visited child might be chosen, or the child with the highest reward.

3.2 Upper Confidence Bound for Trees

The most common implementation of the general scheme is *Upper Confidence Bound for Trees* (UCT) which utilizes formula 3.1 to select nodes in the tree traversal. In this formula $\overline{X}_i$ represents the expected outcome from node i, n is the number of visits to all nodes, n_i is the number of visits to node i. C is an arbitrary constant.

$$UCT_i = \overline{X}_i + 2C\sqrt{\frac{2\ln n}{n_i}} \tag{3.1}$$

The tree is traversed greedily, in every step a node with maximal UCT value is chosen. By convention, unvisited nodes have their UCT value equal to ∞. Importantly, when a node j is visited, its sibling's i UCT value is increased as n increased but n_i did not – this contributes to balance between exploitation of known good strategies and exploration of new. Constant C instructs the algorithm how much weight to give to exploration, we often call it the *tree heuristic constant*. See [17] for insights into choice of C in some specific situations.

Algorithm 4 is an implementation of the general scheme of Algorithm 3. As a minor simplification, the algorithm expects an MDP with rewards (S, s_0, A, E, R) and a marked set of terminal states, such that a reward is paid out only when a terminal state is reached.

Every tree node v has information about its corresponding state ($v.state$), the action which lead to it ($v.action$), the number of times it has been visited so far ($v.visits$), and the reward which has been collected after traversing through it ($v.q$, the Q refers to Q-learning). For a node v value $v.q/v.visits$ thus represents an estimate of the actually attainable reward.

Thus the whole tree is the algorithm's estimate of the actual reward attainable from various parts of the MDP. In each tree node it greedily chooses to confirm it can get a good reward or goes to explore a little known part of the MDP – the choice depends on the tree heuristic constant C. A random rollout is performed, and the algorithm then adds a new node to the tree which allows for a more precise estimate in the chosen part of the tree.

Algorithm 4: Upper Confidence Bound for Trees

function UCT(s_0)
    Let v_0 be the root of the MCTS tree, with $v_0.state = s_0$.
    while within computational budget **do**
        $v_l \leftarrow$ TreePolicy(v_0)
        $r \leftarrow$ Rollout(v_l)
        Backup(v_l, r)
    return BestChild($v_0, 0$).*action*
function TreePolicy(v)
    while v is not terminal **do**
        if v is not fully expanded **then**
            Expand(v)
        else
            $v \leftarrow$ BestChild(v, C)
function Expand(v)
    choose an untried action a from $A(v.state)$
    add a new child v' to v, $v'.state = \Delta(v.state, a), v'.action = a$
function BestChild(v, C)
    return $\underset{v' \text{ a child of } v}{\operatorname{argmax}} \frac{v'.q}{v'.visits} + C\sqrt{\frac{2 \ln v.visits}{v'.visits}}$
function Rollout(s)
    while s is not terminal **do**
        choose $a \in E(s)$ uniformly at random
        $s \leftarrow \Delta(s, a)$
    return reward for reaching s
function Backup(s, r)
    while v is not null **do**
        $v.visits \leftarrow v.visits + 1$
        $v.q \leftarrow v.q + r$
        $v \leftarrow$ parent of v

3.3 Solving Games

In this section a brief introduction to game theory is given, starting with definitions of games, strategies, and solutions to games. We proceed by presenting the standard *minimax* algorithm, then showing how MCTS can be used to solve games and how it compares with minimax. Observing how MCTS is used on another domain might provide useful insight.

We present the following definition for a precise understanding of games. A reader can notice similarities with MDPs, for example, games have states, actions and enabled actions, as well as rewards, moreover, perfect information corresponds to complete information. On the other hand, the transition function is deterministic in our definition.

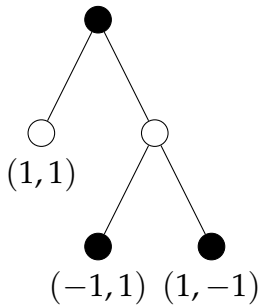
Definition 7. A *perfect-information extensive-form game* is a tuple $G = (N, S, s_0, F, A, E, \Delta, \rho, u)$, where

- $N \subseteq \mathbb{N}$ is a set of n players, for $n \in \mathbb{N}$,
- S is a set of states,
- $s_0 \in S$ is the initial state,
- $F \subseteq S$ is a set of terminal states,
- A is the set of actions,
- $E : S \to \mathcal{P}^+(A)$ is a function which tells the player which non-empty set of actions can be played in a given state,
- $\Delta : (S \setminus F) \times A \to A$ is the transition function, defined such that game states cannot repeat[2],
- $\rho : S \to N$ is a function assigning who is on turn in a given state,
- $u = (u_1, \ldots, u_n)$ is the tuple of utility (payoff, reward) functions where each $u_i, i \in N$ is a function $u_i : F \to \mathbb{R}$.

2. Technically there are no $s \in S, a \in E(s)$ such that $\Delta(s, a) = s_0$, and for any two $s, s' \in S$ there are no actions $a \in E(s), a' \in E(s')$ such that $\Delta(s, a) = \Delta(s', a')$.

Game starts in state s_0 and players take turns until a terminal state (in F) is reached. In a turn a player in state s chooses action a from $E(s)$ and the action leads to state $\Delta(s, a)$. If the state $\Delta(s, a)$ is terminal, then every player i receives the reward $u_i(\Delta(s, a))$ and the game ends.

Example 6. A common choice of the utility function is $+1, 0, -1$ for victory, draw and loss, respectively. Below is a depiction of such a game in the usual way – using a tree. There are two players, the nodes represent game states, leafs are the final states. The root of the tree is the initial state. Actions are given implicitly by choice of the next node. Functions E and Δ are given by the drawn edges. ρ is given by the color of the nodes and the payoff function is represented by a tuple in each leaf.

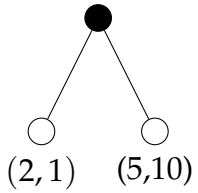


Definition 8. A *strategy* of player i in a game G is a function $\pi_i : S \to \mathcal{D}(A)$, such that for all $s \in S$ and $a \in A$ it holds that if $\pi_i(s)(a) > 0$ then $a \in E(s)$.

A *strategy profile* is an N-tuple with a strategy for each player.

We assume the players are rational and thus pick strategies which optimize for their goals. A goal may be to maximize the player's reward, another goal may be to get a higher score than the opponents.

Example 7. In the game shown below let Alice be player 1 and Bob player 2. There is a single turn made by Alice in the black node. If her goal is to beat Bob she will pick the left node. If her goal is to maximize her profit she will choose the right one.



In the following text we restrict ourselves to two player games in which players alternate in taking actions.

3.3.1 Zero-sum games and minimax

In zero-sum games one player wins at the cost of the other losing. There are plenty of examples of such games, e.g., Chess, Go, which makes them an important subject of study.

Definition 9. Let $G = (N, S, s_0, F, A, E, \Delta, \rho, u)$ be a game. G is said to be *zero-sum* if the rewards always sum to zero, that is $\sum_{i \in N} u_i(s) = 0$ for all $s \in F$.

An important type of strategy is *minimax*. A player playing this strategy is minimizing their potential maximum loss[3]. If both players play a minimax strategy, the strategy profile is a Nash equilibrium[4].

The *minimax algorithm* computes the value of each leaf in a game tree assuming the opponent tries to harm the player as much as possible. That is if player 1 is computing which turn to play, she uses the fact that she can maximize her profit in her future turns but assumes player 2 will try to minimize it. After all options are computed in this alternating manner, player 1 plays the action going into the subtree with the highest value node.

---

3. Some might prefer calling it *maximin* for maximization of the minimum reward, which is equivalent to the first definition in zero-sum games.
4. A strategic profile is a Nash equilibrium, if no player can get a higher reward by switching a strategy.

Now we use the zero-sum property and our assumption of alternating turns to simplify the algorithm for zero-sum games in two steps.

First, only a single payoff function can be used, let it be u_1. Now higher values mean player 1 is winning and lower values that player 2 is winning. Thus player 1 maximizes in her turns, expects player 2 to minimize in his turns. Player 2 minimizes in his turns, expects player 1 to maximize in her turns.

Second, instead of alternating between maximization and minimization, an algorithm can always maximize if it alters the sign of the value in each step.

These simplifications result in Algorithm 5, the *negamax algorithm*. Player 1 gets the value by evaluating $\textsc{Negamax}(\text{state}, 0)$, player 2 gets the value by evaluating $-\textsc{Negamax}(\text{state}, 1)$.

An example execution of the algorithm can be seen on a game of tic-tac-toe in Figure 3.1. X is the first player, O is the second player. At the beginning X can place her mark in the middle of the board and win, or place it at the bottom and lose. The left branch has value 1, the right branch has value -1.

Algorithm 5: Negamax

1: **Input:** Two-player zero-sum game[a] $(N, S, s_0, F, A, E, \Delta, \rho, u)$, where players alternate in taking actions.
2: **Output:** Maximum payoff for player 0, its negation for player 1.
3: **function** NEGAMAX(*state*, *player*)
4: **if** $state \in F$, i.e. a leaf in the game tree **then**
5: **return** [b] $[1, -1][player] \cdot u_1(state)$ otherwise
6: **return** $\max\{-\text{NEGAMAX}(child, 1 - player) \mid child \text{ of } state^{c}\}$

a. The *player* numbers are shifted down by one for easy manipulation.
b. The $[1, -1][player]$ evaluates to 1 for player 0 and to -1 for player 1.
c. Precisely for all enabled actions $a \in E(state)$ every $child \in \Delta(state, a)$.

For larger games a variation called *minimax (negamax) search* is used, which performs the computation only to a limited depth, and then uses a heuristic to evaluate the last node it processes (unless it is a leaf).

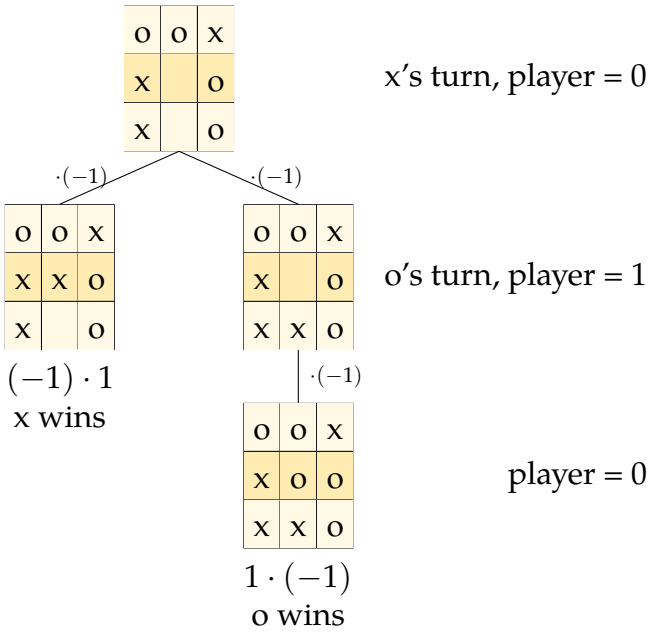


Figure 3.1: Negamax algorithm applied to tic-tac-toe.

3.3.2 Solving with MCTS

From the negamax algorithm in the previous section there is an easy step to solving games with MCTS. Algorithm 6 is an implementation of the Backup function in UCT for finding a good action a player should take in a zero-sum game.

Algorithm 6: Negamax MCTS Backup

```
1: function BACKUP(v, r)
2:     while v is not null do
3:         N(v) ← N(v) + 1
4:         Q(v) ← Q(v) + r
5:         r ← −r
6:         v ← parent of v
```

In comparison with minimax search MCTS may ignore unimportant branches of the game tree and instead explore in greater detail the more promising branches.

For example, when solving a tic-tac-toe game, the algorithm starts with the current board state in the root of the tree, and then adds the nodes representing the next possible states as its children. They are explored with random rollouts, and the algorithm proceeds to the parts of the tree which either seem to lead victory or which are insufficiently explored. The balance between exploitation and exploration is dependent on the choice of the constant in the tree heuristic. See [18] for an elaborate explanation with illustrations.

An important theoretical result is that UCT converges to minimax [14], so eventually, after a lot of UCT iterations, there is a negligible difference between the results of the two methods and they pick the same next move. However in applications we want to know when to use minimax (Algorithm 5) and when to use UCT (Algorithm 4 with Algorithm 6) without iterating for a long time.

A lot of researches have studied this question. While UCT achieved success in many games [16], for example in chess it does not perform well[5]. The reason being that chess games contain a lot of trap states (states from which the opponent has a guaranteed victory) which are reachable just in a few turns. UCT then spends a lot of time exploring these unimportant parts of the search tree [20].

5. Days prior to submitting this thesis a preprint article [19] appeared, describing a successful use of MCTS to play chess. The approach, however, employs model-specific neural networks to guide the search and so it seems that the observations made here (in tree traversal MCTS has only a very limited information about the possible outcomes of each choice) still translate to the domain of MDP verification (as the models are usually not used again, it is not effective to train a network for them). On the other hand, this surely deserves a deeper investigation.

4 MCTS in MDP Verification

In this chapter three new algorithms for verification of Markov decision processes are presented. One called follows closely the general MCTS scheme but the updates maintain lower and upper bounds. The second, is a fusion of MCTS and BRTDP, where each rollout is a single iteration of BRTDP. The third one uses the UCB formula to select the next action in BRTDP.

Before proceeding, recall from section 2.4 that $L, U : S \times A \to [0, 1]$, where S, A are the sets of states and actions of an MDP, are the lower and upper bounds of the value function V. The section also defined $U(s) = \max_{a \in E(s)} U(s, a)$ and $L(s) = \max_{a \in E(s)} L(s, a)$.

The algorithms we suggest can be altered by changing the tree node selection heuristic. We suggest and evaluate three formulas: UCB, VCB, CCB. Here n_i is the number of visits of the node, n is the number of iterations (visits of the root node), s_i is the number of visits of the state corresponding to node i, and s_r is the number of visits to the initial state. s_i/s_r thus represents a guess[1] of the probability of reaching the state corresponding to node i.

$$UCB(i) = \frac{s_i}{s_r} \cdot U(i) + C\sqrt{\frac{2 \ln n}{n_i}}$$

In the VCB (Victory Confidence Bound), v_i is the number of times a target state has been hit when rolling out from state i or its child. However experiments show its performance is similar to UCB in practice or worse on some examples.

$$VCB(i) = \frac{v_i}{n_i} + C\sqrt{\frac{2 \ln n}{n_i}}$$

Furthermore we experimented with a formula which would utilize our confidence in the learned upper bound but found out that on our models its performance is very bad.

$$CCB(i) = U(i)(1 - (U(i) - L(i))) + C\sqrt{\frac{2 \ln n}{n_i}}$$

1. A biased guess as we update s_j in tree traversal too.

4.1 Bounded MCTS

BMCTS, Algorithm 7, is an algorithm which implements the general MCTS scheme in the most straightforward way while guaranteeing approximate correctness of the result.

Function TreePolicy is implemented as in Algorithm 4 and uses one of the formulas described at the beginning of this chapter. It starts at the root, and then repeatedly chooses a child with the highest value given by a formula until it reaches a leaf v_l. If the corresponding state $v_l.state$ has successors, they are added in nodes as children of the node and one of them is returned. If not, then v_l is returned.

In each step of a rollout, an action is chosen uniformly at random, and a state is then sampled according to the transition distribution.

The primary distinction is in the backpropagation phase. To maintain the lower and upper bounds, backpropagation using the Bellman update is performed first in the MDP (Backup), and then in the tree (TreeBackup).

Backup: Perform Bellman updates from the terminal state reached in the rollout to the state corresponding to the selected tree node. This is the same procedure as in BRTDP.

TreeBackup: Perform Bellman updates from the states corresponding to the nodes on the taken tree path. Here the algorithm updates the bounds for every action as the tree does not differentiate between actions.

Compare this backpropagation with a typical MCTS implementation where this is not necessary and updates are performed only on the tree path.

This algorithm is an improvement upon random sampling as it guides the random rollouts towards the parts of the decision process which look more relevant to maximizing the sought probability. However, the random rollouts prove to be ineffective in practice, which leads us to our main algorithm.

Algorithm 7: BMCTS

```
1:  function BMCTS(s_0)
2:      Let v_0 be the root of the MCTS tree, with v_0.state = s_0.
3:      while U(s_0) − L(s_0) > ϵ do
4:          v_l ← TreePolicy(v_0)
5:          ω ← Rollout(v_l.state)
6:          Backup(ω)
7:          TreeBackup(v_l)
8:      return (L(s_0), U(s_0))
9:  function Rollout(s)
10:     ω ← s
11:     while last(ω) is not terminal do
12:         a ← sample uniformly from E(s)
13:         s ← sample according to Δ(last(ω), a)
14:         ω ← ω a s
15:     return ω
16: function Backup(ω)
17:     while ω is not empty do
18:         pop(ω)
19:         a ← pop(ω)
20:         s ← pop(ω)
21:         U(s, a) := Σ_{s'∈S} Δ(s, a)(s')U(s')
22:         L(s, a) := Σ_{s'∈S} Δ(s, a)(s')L(s')
23: function TreeBackup(v)
24:     v ← v.parent            ▷ The leaf's update was done in Backup
25:     while v ≠ v_0 do
26:         s ← v.state
27:         for a ∈ E(s) do
28:             U(s, a) := Σ_{s'∈S} Δ(s, a)(s')U(s')
29:             L(s, a) := Σ_{s'∈S} Δ(s, a)(s')L(s')
30:         v ← v.parent
```

4.2 MCTS-BRTDP

MCTS-BRTDP is a variation of BMCTS in which each rollout is a single iteration of BRTDP as described in chapter 2. This allows for significantly faster updates of the bounds as compared to BMCTS but still utilizing exploration which makes it possible to overcome parts of the MDP which are hard for BRTDP.

The functions TreePolicy and TreeBackup are implemented in the same way as in Algorithm 7. Recall that the tree is traversed greedily by choosing uniformly one of the nodes with maximal heuristic value. Tree backpropagation updates bounds on the states corresponding to the nodes on the traversed path. Note that we avoid using Backup as that is done by BRTDP.

Moreover, the implementation of BRTDP used here maintains a map from states to lists of nodes in the tree. Every time a MEC is collapsed in BRTDP, we remove the subtrees in the MCTS tree where the root node corresponds to a state of the collapsed MEC.

This ensures that the algorithm maintains a consistent representation of the MDP throughout the computation. Since MCTS does not restrict BRTDP access to any part of the MDP it follows that the algorithm almost surely converges to the correct value.

Algorithm 8: MCTS-BRTDP

```
1: function MCTS-BRTDP(s_0)
2:     Let v_0 be the root of the MCTS tree, with v_0.state = s_0.
3:     while U(s_0) − L(s_0) > ε do
4:         v_l ← TreePolicy(v_0)
5:         an iteration of BRTDP(v_l)
6:         TreeBackup(v_l)
7:     return (L(s_0), U(s_0))
```

4.3 UCB in BRTDP

We further present one algorithm which is not based on MCTS but incorporates the UCB exploration formula into BRTDP. Instead of choosing an action with the greatest upper bound, it chooses an action with the greatest UCB value.

Line 7 of Algorithm 2 is thus replaced with the line below, where s is the current state of the simulation ($last(\omega)$ in BRTDP). The range of the argmax are all states i which can be reached from s through an allowed action a with non-zero probability. Here n_i is the number of visits of state i, and n denotes the number BRTDP iterations run so far.

$$a \leftarrow \text{sample uni. from} \operatorname*{argmax}_{a \in E(s), \forall i \text{ s.t. } \Delta(s,a)(i) > 0} U(i) + C\sqrt{\frac{2 \ln n}{n_i}}$$

This algorithm eventually explores as much as the previously mentioned variants of BRTDP if necessary.

5 Evaluation

This chapter explains the basics of working with PRISM model checker, then describes how are the algorithms implemented as a part of PRISM, how they behave on small models, and how do they compare to other methods on standard models.

5.1 PRISM, Probabilistic Model Checker

PRISM [21] is a program/framework for formal modelling and analysis of probabilistic systems. We show how to use PRISM to describe MDPs, their properties, and how to check them.

Describing MDPs with the PRISM language

PRISM has a language for description of Markov decision processes based on the formalism of Alur and Henzinger [22]. A brief example is given below for the readers who wish to try our algorithms on small models which they can describe and solve by hand. The definitive guide to the language is available online on the PRISM homepage [23].

The PRISM language describes *modules* (interacting controllers), their states (using variables), and transitions between the states. An example single PRISM language line is below. The line translates as: if condition `guard` is satisfied the controller can choose action `act` and with probability `prob_1` update `update_1` will happen, with probability `prob_2` update `update_2` will happen, and so on.

```
[act] guard -> prob_1 : update_1 + prob_2 : update_2 + ...
```

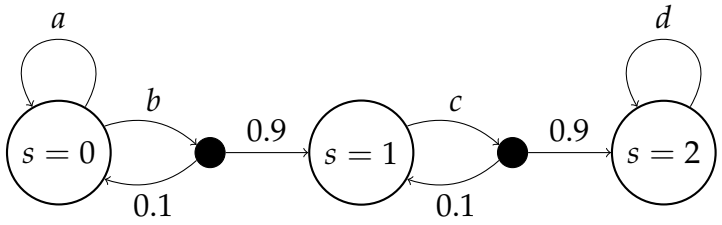


Figure 5.1: Module M

An example module is described below. It is equivalent to the MDP depicted in Figure 5.1.

```
mdp // Tell PRISM this file describes an MDP
module M
    s : [0 .. 3] init 0;
    [a] s=0 -> (s'=0);
    [b] s=0 -> 0.9:(s'=1) + 0.1:(s'=0);
    [c] s=1 -> 0.9:(s'=2) + 0.1:(s'=1);
    [d] s=2 -> (s'=2);
endmodule
```

A property we might be interested in is the maximum probability of eventually reaching state `s=2`, this could be described to PRISM with the string `Pmax=? [F s=2]`, where F represents the $\Diamond$ symbol used previously. A user can use standard logical connectives for describing the target states. At the moment our implementation does not support timed properties (i.e., `F <= x` is not supported), however, we expect it is not too hard to implement such functionality.

Running PRISM

We have a model description and a property but before we analyze it PRISM has to be installed. There is a modified version of PRISM distributed with this thesis. It can be built by issuing the `make` command inside the `prism/prism` directory. Java is a required prerequisite. Once built the PRISM binary is available at `prism/prism/bin/prism`.

Now use PRISM to analyze the model – its name is passed as the first argument and `-pf` specifies the property we want to check. Flag `-ex` will be described later. Running the command below confirms our expectations: the maximum probability of eventually reaching state `s=2` is 1. If not specified otherwise, the algorithm used is value iteration.

```
./prism modelM.nm -pf 'Pmax=? [F s=2]' -ex
```

The following tells PRISM to use MCTS-BRTDP where the next state in BRTDP is chosen to be the one with highest upper bound.

```
./prism modelM.nm -pf 'Pmax=? [F s=2]' -heuristic_verbose \
 -heuristic MCTS_BRTDP -next_state HIGH_PROB
```

The heuristic method can be chosen out of the following: `MCTS_BRTDP`, `BRTDP`, `BRTDP_UCB`. BMCTS is chosen by using method `MCTS_BRTDP` together with `-next_action 5`. The next state heuristics we use are `HIGH_PROB`, `MAX_DIFF`. The variation of BRTDP using UCB to select the next action is chosen by adding `-next_action 2`. The tree heuristic constant can be chosen with `-ucb1constant`.

Data structures

The `-ex` switch used in the value iteration example above tells PRISM to use the explicit computation engine. The explicit computation engine explores the MDP and stores it in a sparse matrix before the value iteration algorithm is run. PRISM also offers three symbolic computation engines based on binary decision diagrams.

All the heuristic methods use a data structure implemented in `prism/prism/src/heuristic/CachedModelGenerator.java`. With this *model generator* data structure PRISM will not build the whole MDP from its description unless asked to. Asking the model generator to reveal parts of the model results in the construction of an *explicit model* which is cached in the memory. Such construction is computationally expensive and often unnecessary which is when the heuristic methods perform so well.

5.2 Implementation

We describe how the pseudocode described in chapter 3 maps to the implementation in PRISM. The implementation can be found in `prism/prism/src/heuristics` attached to the thesis.

To represent the MCTS tree we use classes `MCTree` and `MCNode`. The tree class has method `unfold` which adds the next states of a given state to the explicit model, and then puts them to the tree in nodes.

The next state heuristics are implemented in a straightforward way inside directory `nextstate`. The tree heuristics described earlier are implemented in directory `treeheuristic`.

MCTS-BRTDP is implemented in `search/MctsBrtdp.java`. The entry point is `computeProb`, subsequently `monteCarloTreeSearch` is invoked until the stopping condition is reached (see method `isDone`).

Method `monteCarloTreeSearch` first selects and expands a tree state (using `treeSelectAndExpand`), then uses `exploreAndUpdate` implemented in BRTDP (`search/HeuristicBrtdp.java`) as the rollout and propagates the values using updates to the root.

As we have observed problems only once during thousands of runs on our models, we have not implemented the removal of subtrees induced by nodes corresponding to a state which is contained in a collapsed MEC. The increase in complexity of such implementation should not be significant.

BMCTS is implemented by modifying MCTS-BRTDP. The modification is turned on when the flag `next_action 5` is added to a command using MCTS-BRTDP. This change makes the BRTDP implementation chose next action uniformly at random instead of MAX-DIFF or HIGH-PROB.

5.3 Behaviour on Small Models

For small models, we used visualization to observe how MCTS-BRTDP solves them. To render the progress of MCTS-BRTDP into a series of pictures, the last lines of `MctsBrtdp.monteCarloTreeSearch` have to be uncommented. Due to the randomized nature of the algorithms, a researcher should observe more runs before making conclusions.

Three simple models are presented with a description of the methods' approach to solving them. The first is a model resembling a binary tree, second is the BRTDP adversary, the third model offers a choice between a simple path and a complex cloud. They can be found in directory `small_models` attached to the thesis.

Binary Tree Model

The "binary tree" model features two decisions (*left* and *right*) in each state and each decision has two successors, the *left* one occurs with probability 0.2, the *right* one with 0.8. A path through the model goes through 4 states before it reaches a "leaf" state. Every leaf state of the subtree induced by the left decision in root leads to state 85, every leaf state of the other subtree leads to state 86. We ask what is the maximum probability of reaching state 85.

By running the program repeatedly it can be observed that MCTS-BRTDP almost evenly explores the branches in a balanced way while utilizing BRTDP rollouts to search for promising paths. BRTDP on its own selects a branch it knows the least about until it learns the upper bound for the right part of the tree is 0 and therefore it has to focus on the left part. The VCB for MCTS-BRTDP focuses on the left part quickly due to successful results, however it still explores the right part a bit to look for possibly missed target states.

BRTDP Adversary

In Example 5 MCTS-BRTDP has a clear advantage as it traverses to a leaf of the tree and then adds a node to it with each iteration. Soon the tree reaches the target state and it remains to perform updates equivalent to value iteration.

Cloud And Path Model

The model has an initial state, "left part", and "right part". Left part is a simple path to a target state. Right part is comprised of states with randomly selected choices and transitions but without a target state. From the initial state there is an action to go left with probability 0.8 and right with probability 0.2, and another action with the same effect but reversed probabilities.

MAX-DIFF BRTDP quickly learns that the upper bound in right part is zero, abandons this part of the MDP to focus on the left part and learn its upper and lower bound 0.8.

MCTS-BRTDP is forced to explore the right part too but learns the correct value soon. On the other hand MCTS-BRTDP with the VCB formula finds the target in the left part and then focuses too much on this part even though what it needs is to learn that the right part has upper bound 0 – this variant is thus taking very long to find the correct probability unless configured with a high tree heuristic constant.

Observing the VCB variant of MCTS-BRTDP on this and the binary tree models suggests that it is quick to find good lower bounds. However, then it commits to the good paths of the MDP so much that it has trouble computing a better upper bound in the parts which do not lead to a target way (as quickly as the exploited parts).

5.4 PRISM Benchmark Suite and Other Models

Experimental evaluation was done mainly on standard models distributed with PRISM. Most of the MDPs described by these models are concerned with network configuration where randomness plays important role in achieving a common goal. We refer the reader to thorough descriptions of all the models on PRISM's website.

The description of the standard models in the PRISM language can be found inside `prism/tests/reachability/models` in the source codes attached to this thesis. The properties checked are defined in `prism/tests/reachability/scripts/benchmark.py` for each model.

We also created new MDPs by combining the PRISM models with the MDP which is hard for BRTDP (Example 5). We describe them in the last subsection.

Combining PRISM models with BRTDP adversary

We combined the MDPs in two ways. The first is by *branching* and the resulting shape is shown in Figure 5.2.

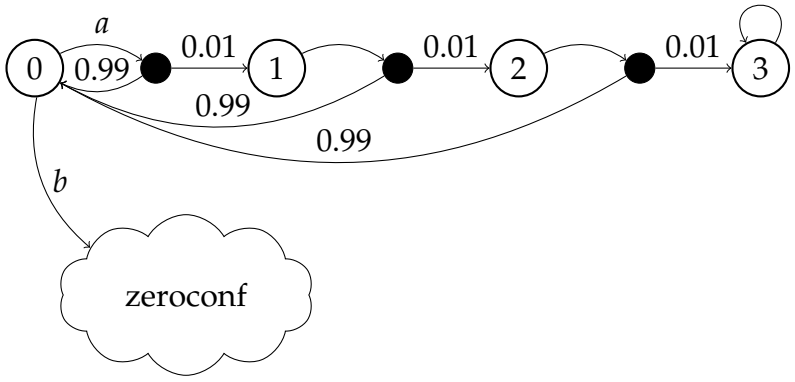


Figure 5.2: Combining zeroconf model with the BRTDP adversary in a branching manner.

The second way to combine MDPs is parallel composition, which is done by using modules in the PRISM language. Each state of the composed MDP is a member of the product set of the sets of states of each of the modules. The choice of the next transition is non-deterministic, i.e. a strategy does not decide only which transition in a module to use but also which module to use (the strategy for the MDP can be viewed as a member of the product set of the set of strategies for each of the modules).

5.5 Experimental Comparison

On the following pages we present the most important results of our measurements. In the presented tables rows correspond to models and their configuration, columns to methods. Each table cell contains comma separated running time in seconds and number of visited states of the MDP[1,2]. Hyphen denotes timeout or insufficient memory. *k* is used as a shorthand for thousands.

Each experiment shown in the tables below was repeated five times and the results were averaged. There was no significant variance (beyond 10 %) in the measured times, so the number of repetitions is sufficient and averaging provides representative results. We set ϵ to 10^{-6}, except for the "branch" models where we used 10^{-2}.

We used a machine equipped with 8 core `Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz` processor, OpenJDK Runtime Environment 1.8 and JVM configured to use 4 GB of memory and 512 MB stack size.

Our benchmark script allows easy, declarative description of each measurement. Its details are described in the appendix. See `prism/prism/tests/reachability/scripts/run_benchmark.py` for an example usage.

In the tables the numbers next to the model name are for zeroconf N, K; for firewire_dl delay, deadline; for consensus (coin) protocol N,K. We use M-B as a shortcut for MCTS-BRTDP-MAXDIFF. We used 1.4 for the tree heuristic constant unless otherwise noted.

1. The number of states is, unlike running time, agnostic to implementation details.
2. Value iteration has to construct the whole model in memory but we show the number of states after VI preprocessing.

In Table 5.1 we can see that MCTS-BRTDP-MAXDIFF keeps up with BRTDP-MAXDIFF on the PRISM benchmark suite. VI and BRTDP-UCB time out on several models. MCTS-BRTDP-VCB-MAXDIFF (not included in the table) performs similarly to MCTS-BRTDP-MAXDIFF, but better on coin and leader models, where it outperforms BRTDP.

Table 5.2 demonstrates that MCTS based methods have a significant advantage on the "branch" models. We can see that BRTDP finds the way to target in zeroconf and has to use that to compute the bounds, while MCTS methods explore the other branch and use it to update faster. However, they pay the cost of exploring a lot more states. MCTS-BRTDP-VCB-MAXDIFF performs worse than its UCB variant as opposed to the previous table.

Measurements show that the only method which works on at least some of the composition-zeroconf models is value iteration (specifically with $K = 40$ the model has 141 thousand states and VI solves it, regardless of choice of N). Other methods always run out of time. But on composition-wlan6 we observed that only MCTS based methods can actually solve the problem. MCTS-BRTDP-VCB-MAXDIFF needs just 2 seconds and over twelve hundred states to explore.

The impact of the tree heuristic constant is usually straightforward. Setting it too low might get the algorithm stuck in bad choices for long, but once a small value enables some exploration a significant increase only makes the algorithm explore more than is necessary. For example, for the branch-zeroconf model, the algorithm finishes faster with constant set to 1.4 rather than 10.

In Table 5.3 we can see that with a careful choice of the UCB constant MCTS-BRTDP-MD can, in fact, outperform BRTDP-MAXDIFF even though the first choice of the UCB constant in Table 5.1 suggested otherwise.

In summary, MCTS based methods work well on several models (when the exploration constant is set properly), but there are models where VI and BRTDP are better. However, MCTS-BRTDP-MAXDIFF seems to work without timeouts (albeit possibly slowly) on many models, making it a kind of universal choice.

Table 5.1: Comparison on standard PRISM models. * UCB constant 8 used to avoid timeout.

	VI	BRTDP, MD	M-B	BMCTS	BRTDP-UCB
z-conf 15, 10	205, 184k	0.1, 762	0.114, 801	1, 1045	12, 399
z-conf 20, 10	213, 184k	0.103, 791	0.111, 823	1.35, 1070	10.7, 417
z-conf 20, 20	-	0.167, 1181	0.151, 1226	0.71, 1459	13.2, 659
z-conf 100, 40	16.2, 354k	0.323, 2225	0.265, 2255	2.02, 2501	-
firewire_dl 360,5000	-	0.02, 44	0.02, 38	0.01, 25	1.1, 25k
wlan 6	-	0.07, 412	0.07, 715	0.07, 572	0.1, 1762
coin 4,3	0.6, 33k	34, 9k	33, 10k*	48, 10k*	36, 9k
leader	6, 0	15, 140k	37, 137k	20, 160k*	17, 170k

Table 5.2: Comparison on a "branch" model.

branch-zeroconf	VI	BRTDP, MD	M-B	BMCTS
$N = 20, K = 20$	205, 1847k	23, 141	11.2, 3034	11.5, 2161
$N = 30, K = 10$	206, 1847k	20, 81	12, 4017	12, 2910
$N = 140, K = 40$	10, 354k	31.6, 261	12.7, 4017	11.9, 2910

Table 5.3: Influence of the exploration constant.

MCTS-BRTDP-MD, C:	1	2	4	6	8	10	12
leader	33, 137k	27, 150k	12, 138k	13, 137k	14, 136k	15, 140k	14, 139k

6 Conclusion

We introduced Markov decision processes, the problem of their verification, and known approaches to its solution. Monte Carlo tree search was described in its most common variant UCT (Upper Confidence bound applied to Trees), together with its applications to maximizing rewards in MDPs and solving games. We suggested various new algorithms by combining the known approaches to MDP verification with the techniques of MCTS. These algorithms were implemented and evaluated on standard and new models.

We have observed the MAX-DIFF variant of BRTDP is a powerful heuristic which itself often balances well between exploration and exploitation in common MDP models. Our MCTS based algorithms perform comparably on the PRISM benchmark suite, depending on the exact model and configuration. However, our methods perform significantly better on many models which are hard for BRTDP. Still, there are models where value iteration is the best choice.

There remains a lot of work to be done in order to properly understand how MCTS based methods may be applied in MDP verification. A quantitative study of the algorithms' executions would help understand which parts of an MDP are explored even though they are not important and which important parts are explored too late. Such observations could be used to suggest new formulas for tree node selection or other variations, however, due to the complexity of the models, this might be a demanding task.

Another interesting area of research might be into new algorithms where the MCTS approach has better chances to improve the search, for example, one might try running MCTS and BRTDP in stages, each time for a limited number of iterations until the bounds are sufficiently close. Interleaving of MCTS-BRTDP-VCB (to get a precise lower bound quickly) with BRTDP (to eliminate overly optimistic upper bounds) might also work for some MDPs.

There are also rather easy practical tasks like adding support for time-bounded properties or extracting the strategy from the solution.

Bibliography

1. BRESINA, John; DEARDEN, Richard; MEULEAU, Nicolas; RAMAKRISHNAN, Sailesh; SMITH, David; WASHINGTON, Rich. Planning Under Continuous Time and Resource Uncertainty: A Challenge for AI. In: *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*. Alberta, Canada: Morgan Kaufmann Publishers Inc., 2002, pp. 77–84. UAI'02. ISBN 1-55860-897-4.
2. BRÁZDIL, Tomáš; CHATTERJEE, Krishnendu; CHMELIK, Martin; FOREJT, Vojtech; KŘETÍNSKÝ, Jan; KWIATKOWSKA, Marta Z.; PARKER, David; UJMA, Mateusz. Verification of Markov Decision Processes using Learning Algorithms. *CoRR*. 2014, vol. abs/1402.2967. Available also from: `http://arxiv.org/abs/1402.2967`.
3. ROSENTHAL, J.S. *A First Look at Rigorous Probability Theory*. World Scientific, 2000. ISBN 9789810243227. Available also from: `https://books.google.de/books?id=Fjr0P25SUbYC`.
4. KEMENY, John G.; SNELL, J. Laurie; KNAPP, Anthony W. *Denumerable Markov Chains*. Springer New York, 1976. Available from DOI: `10.1007/978-1-4684-9455-6`.
5. PUTERMAN, Martin L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1994. ISBN 0471619779.
6. COURCOUBETIS, Costas; YANNAKAKIS, Mihalis. The Complexity of Probabilistic Verification. *J. ACM*. 1995, vol. 42, no. 4, pp. 857–907. ISSN 0004-5411. Available from DOI: `10.1145/210332.210339`.
7. FOREJT, V.; KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. Automated Verification Techniques for Probabilistic Systems. In: BERNARDO, M.; ISSARNY, V. (eds.). *Formal Methods for Eternal Networked Software Systems (SFM'11)*. Springer, 2011, vol. 6659, pp. 53–113. LNCS.
8. BELLMAN, Richard. A Markovian Decision Process. 1957, vol. 6, pp. 15.

9. HADDAD, Serge; MONMEGE, Benjamin. Interval iteration algorithm for MDPs and IMDPs. *Theoretical Computer Science*. 2017. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/j.tcs.2016.12.003.

10. MCMAHAN, H. Brendan; LIKHACHEV, Maxim; GORDON, Geoffrey J. Bounded Real-time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees. In: *Proceedings of the 22Nd International Conference on Machine Learning*. Bonn, Germany: ACM, 2005, pp. 569–576. ICML '05. ISBN 1-59593-180-5. Available from DOI: 10.1145/1102351.1102423.

11. BRÜGMANN, Bernd. *Monte Carlo Go*. München, Germany, 1993.

12. BOUZY B., Helmstetter B. Monte-Carlo Go Developments. In: *Van Den Herik H.J., Iida H., Heinz E.A. (eds) Advances in Computer Games. IFIP — The International Federation for Information Processing, vol. 135*. Springer, Boston, MA, 2004.

13. COULOM, Rémi. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: *Proceedings of the 5th International Conference on Computers and Games*. Turin, Italy: Springer-Verlag, 2007, pp. 72–83. CG'06. ISBN 978-3-540-75537-1. Available also from: http://dl.acm.org/citation.cfm?id=1777826.1777833.

14. KOCSIS, Levente; SZEPESVÁRI, Csaba. Bandit Based Monte-carlo Planning. In: *Proceedings of the 17th European Conference on Machine Learning*. Berlin, Germany: Springer-Verlag, 2006, pp. 282–293. ECML'06. ISBN 978-3-540-45375-8. Available from DOI: 10.1007/11871842_29.

15. SILVER, D. et al. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016, vol. 529, pp. 484–489. Available from DOI: 10.1038/nature16961.

16. BROWNE, Cameron et al. A Survey of Monte Carlo Tree Search Methods. 2012, vol. 4:1, pp. 1–43.

17. KOCSIS, Levente; SZEPESVÁRI, Csaba; WILLEMSON, Jan. *Improved Monte-Carlo Search*. 2006. Technical report. University of Tartu, Estonia.

18. WHEELER, Tim. *AlphaGo Zero - How and Why it Works*. 2017. Available also from: `http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/`.

19. SILVER, David et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. Available from eprint: `arXiv:1712.01815`.

20. RAMANUJAN, Raghuram; SABHARWAL, Ashish; SELMAN, Bart. On Adversarial Search Spaces and Sampling-based Planning. In: *Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling*. Toronto, Ontario, Canada: AAAI Press, 2010, pp. 242–245. ICAPS'10.

21. KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: GOPALAKRISHNAN, G.; QADEER, S. (eds.). *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer, 2011, vol. 6806, pp. 585–591. LNCS.

22. ALUR, R.; HENZINGER, T. Reactive Modules. *Formal Methods in System Design*. 1999, vol. 15, no. 1, pp. 7–48.

23. *PRISM Manual | The PRISM Language / Introduction* [online] [visited on 2017-10-17]. Available from: `http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction`.

A Benchmarking

To allow the reader to benchmark our implementation we are providing a brief description of how to use, modify, and interpret the results of our benchmark script.

Our core benchmark script with definitions is to be found in `prism/tests/reachability/scripts/benchmark.py`, in the same directory is `run_benchmark.py` which is used to describe the actual benchmark to be run.

For every model there is a class like `ZeroconfParameters`, whose objects hold the configuration necessary to run a benchmark of the model. To generate an unconfigured instance of the class use a function starting with `empty_`, e.g., `empty_ZeroconfParameters()`.

Now the parameters object needs to be configured. If the model has any parameters, these can be set with function `for_constants(` *parameter name, list of constants, parameter object generator*`)`. The result is again a generator[1] of parameters holding objects, each with one of the values set as the parameter. The script implements shortcuts for several constants, e.g., `forK` which is useful for zeroconf.

Next a method for solving the model has to be chosen. This is done using `for_method(`*list of methods, parameter object generator*`)`. For MCTS based methods a constant for the tree formula has to be set with `for_ucb_constants`.

Now the generator is passed to function `run(`*# of repetitions, parameter object generator*`)` which prints method name, UCB constant (might be `None`), and model parameters, and then invokes *evaluator functions*.

```
run(5,
  for_ucb_constants([0.5, 1, 5],
    for_method(['BRTDP', 'MCTS_BRTDP_MAXDIFF', 'VI'],
      forN([10],
        forK([10],
          empty_ZeroconfParameters())))))
```

1. This allows for convenient chaining of configuration functions. See Python documentation for an explanation of generators.

Function `vi_evaluator` repeatedly invokes PRISM, computes statistics of the results and prints the following:

1. average time to construct the model,
2. average time to check the property,
3. the result of verification,
4. total number of states,
5. number of states VI had to process,
6. number of timeouts,
7. number of errors.

Function `heuristic_evaluator` repeatedly invokes PRISM, computes statistics of the results and prints the following:

1. average time to check the property,
2. average time minus the minimum time to check the property,
3. the maximum time minus average time,
4. number of heuristic trials,
5. average number of steps,
6. minimum lower bound,
7. maximum upper bound,
8. average number of states explored,
9. the number of timeouts,
10. the number of errors.

For convenience the script offers function `yield_generators_list` which takes a list of generator objects and returns a single generator with the same functionality as the passed objects together. As we did not need to change the allowed time to run the script we have hard-coded value called `timeout` in `benchmark.py`.