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

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Abstract

We explore how Monte Carlo tree search type algorithms can be used for verifictation of Markov decision processes (with complete information) by balancing between exhaustive search and heuristic search (which might spend a long time in local minima). Several new algorithms either based on UCT (most successful MCTS algorithm) or its variations are proposed and experimentally evaluated on standard models. Our results show MCTS type algorithms perform as well as BRTDP on most standard models but are faster on models where BRTDP underestimates certain paths to goals.

Keywords

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

Contents

1 Introduction

A rover called Curiosity landed on Mars in August 2012 and is performing research in the Gale crater since then. It has to make decisions with uncertain consequences to maximize its scientific output but also to maintain operational state.

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

Science models these and many similar real-world problems with *Markov Decision Processes* (MDPs). The participants in these processes take actions of their choice but the results of these actions are not certain – the rover may finish an experiment with high probability but also break with small, the network may be configured with high probability or not with small.

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's the probability the process fails (rover breaks) or succeeds (the network configures) – to avoid taking such actions, prevent adversaries from forcing them or, for the positive case, to encourage them.

Classical algorithms for MDP verification (the act of proving properties) based on dynamic programming can be used for finding the maximum probability of 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 high reward strategies in Markov decision processes. Recently the algorithm had a big success in the field of computer Go.

In this thesis we explore how can Monte Carlo Tree Search be used to find strategies maximizing given properties. The result are three algortihms: one a variation of Upper Confidence Tree algorithm (an MCTS variant), 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 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 maximizing 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 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

We start by introducing Markov chains, a simpler type of probabilistic processes which does not offer choices to be made, then continue to generalize to Markov Decision Processes. We then overview known approaches to *verification* (checking if a given MDP satisfies a given property) with special focus on Bounded Real-Time Dynamic Programming which will be used in later chapters.

It is assumed the reader is familiar with basic notions of probability theory, namely those of *probability space* and *probabilty 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

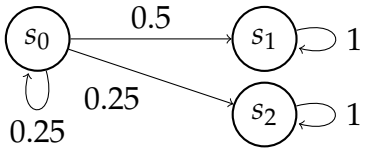
Markov chain is a simpler formalism than Markov decision process and provides a good first intuition about probabilistic models. In a Markov chain there are no decisions, only probabilities of transition.

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-homogenous 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 $P(X_{k+1} = s_{k+1} \mid X_k = s_k) = P(X_k = s_k \mid X_{k-1} = s_{k-1})$.

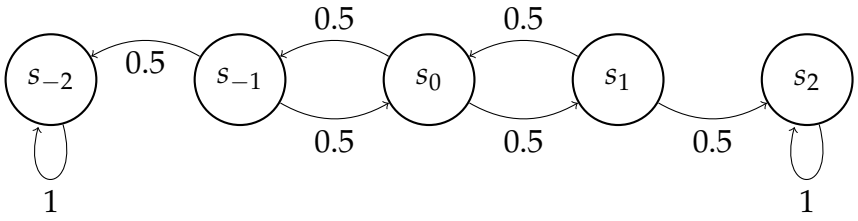
In many applications it is natural to have an initial state s ($X_1 = s$) and a chain can then be drawn as a graph with probabilistic transitions. Note that the weights of the outgoing edges from a node sum to one, as the transition probabilities form a distribution on the set of states (for simplicity 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 randomly left or right. 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

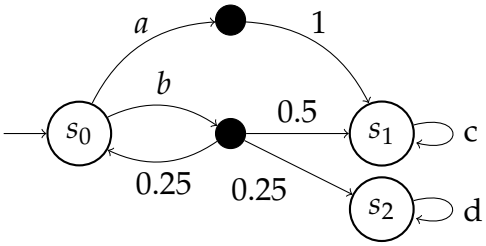
Markov decison processes are similar to Markov chains, except now an controller interacting with the process can in each state pick an action and the next state is chosen according to a distribution on states corresponding to this action.

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 distrubution 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 didn't 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.

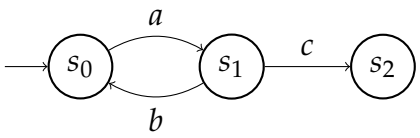


Figure 2.1: An MDP with a non-trivial end component.

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 $\text{last}(\rho)$.

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.

Completness 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

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.

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 modelled as Markov decision processes and the properties are concerned about 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 maximizing 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 strategy maximizing the reachability probability. Puterman [5] proves this for reward maximization and we can use his proof with a simple reduction [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 only into a new sink state (with reward 0).

In this section we describe value iteration and strategy iteration, which are standard algorithms 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 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.

2.3.1 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). Limit on the number of iterations may be introduced if we want to check time bounded properties.

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 good 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.

4. Known for introducing the term dynamic programming.

Algorithm 1: Value Iteration

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

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.

2.3.2 Strategy Iteration

TODO Strategy Iteration (also Policy Iteration). Describe origin How it works Pros and cons

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 $\mathcal{M} = (S, A, E, \Delta)$, its initial state s_0, 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

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

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.

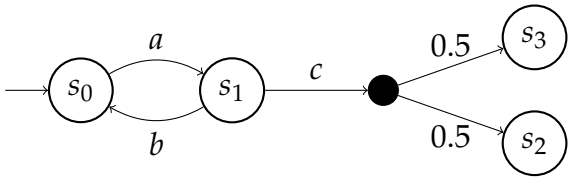
We start with an important assumption, that that $\mathcal{M}$ does not contain any end component besides two trivial end components, one containing only the target state 1 with $F = \{1\}$, the other only the state 0 with $V(0) = 0$.

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 maximizing 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.

BRTDP for MDPs with End Components

Unfortunately 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 $\epsilon < 0.5$.



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"

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 explore 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.

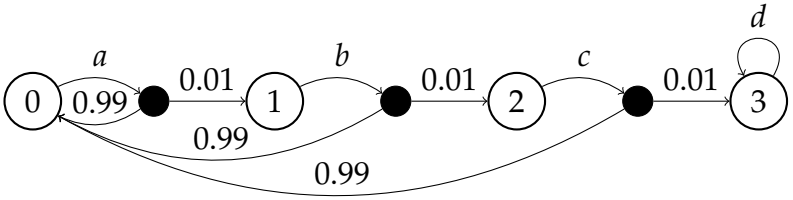
If an end component is found it is collapsed, i.e. the states are merged into a single state and functions E, Δ are naturally transformed to work the same way in the modified MDP.

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.

Variants of BRTDP

When in state s and 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 be 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. For example the HIGH-PROB TODO



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

3 Monte Carlo Tree Search

Monte Carlo methods[1] are using random sampling to estimate the correct solution to a problem. The first serious 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] independetly explored the idea of maintaing 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 resulting rewards of 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 proceeds at random and upon terminating it adds a new leaf to the tree and updates its ancestors with the result. This is summarized in Algorithm 3.

Algorithm 3: General Monte Carlo Tree Search method

function MCTS(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)
    $\Delta \leftarrow$ Rollout(v_l)
    Backup(v_l, Δ)
  return Action from v_0 to the best node (by some given metric).

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.

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

The tree is traversed greedily, in every step a node with maximal UCT value is chosen. Note that 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. See [17] for insights into choice of C in some specific situations.

Algorithm 4 is implementation of the general 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 state, 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).

TODO: More explanation. e.g. why does it return what it returns, what does q/n mean...

TODO: Example execution.

As UCT is the most common type of MCTS algorithm, the terms UCT and MCTS are often used interchangeably in literature.

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. Lastly we briefly compare the application of MCTS to Go and Chess. Observing where it performs good and where it does not provides insight the algorithm and it is a useful starting point for understanding the results of evaluation in chapter 5.

For precise understanding of the model of games we present the following definition. The perfect-information in games corresponds to full observability in MDP. The reader can notice other similarities with MDPs as well, for example games have states, actions and enabled actions, as well as rewards. 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,

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 $\underset{v' \text{ a child of } v}{\operatorname{argmax}} \frac{v'.q}{v'.visits} + c\sqrt{\frac{2 \ln v.visits}{v'.visits}}$

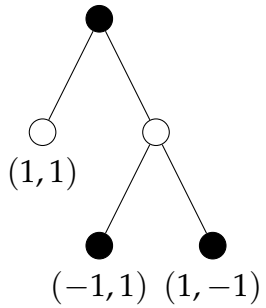
function Rollout(s)
  while s is not terminal **do**
    choose $a \in E(s)$ uniformly at random
    $s \leftarrow \Delta(s, a)$

function Backup(s)
  while v is not null **do**
    $v.visits \leftarrow v.visits + 1$
    $v.q \leftarrow v.q + r$
    $v \leftarrow$ parent of v

- $s_0 \in S$ 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 the function which determines who plays in each state, and finally
- $u = (u_1, \ldots, u_n)$ is the tuple of utility (payoff, reward) functions where each $u_i, i \in N$ is a function $u_i : F \to \mathbb{R}$.

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.



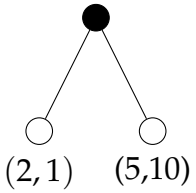
2. Technically there are no $s \in S, a \in E(s)$ such that $\Delta(s, a) = s_0$, and for any two $s, s' \in S$ there are no actions $a \in E(s), a' \in E(s')$ such that $\Delta(s, a) = \Delta(s', a')$.

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[4,5].

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.
5. The *minimax theorem* was proven by John von Neumann.

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.

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.

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

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 5: Negamax

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

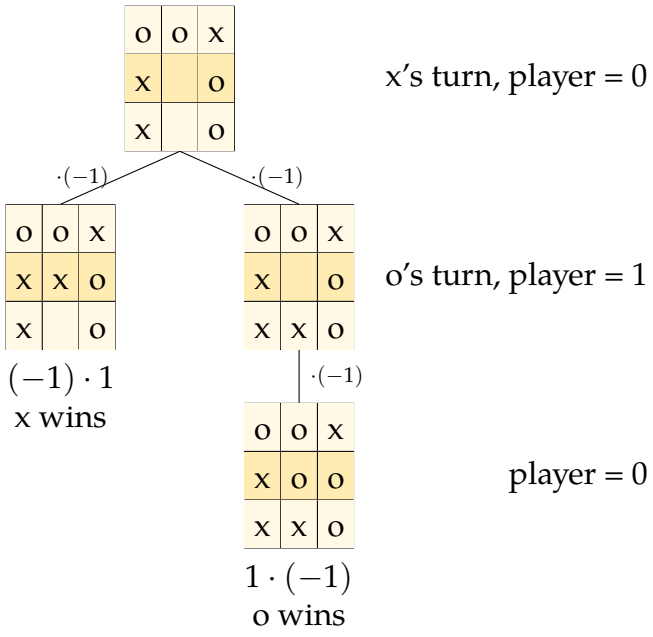


Figure 3.1: Negamax algorithm applied to tic-tac-toe.

Algorithm 6: Negamax MCTS Backup

function BACKUP(v, Δ)
  while v is not *null* **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta$
    $\Delta \leftarrow -\Delta$
    $v \leftarrow$ 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 proceeds to add the immediate turns into the tree. They are explored with random rollouts and the algorithm proceeds to the parts of tree which either seem to lead victory or which are unsufficiently 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 MCTS converges to minimax [14], so eventually, after a lot of MCTS 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 UCT for a long time.

While UCT achieved success in many games [16], for example in chess it does not perform well. 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 [19].

3.3.3 Computer Go

Go is an example of a game where MCTS has proven to be a good choice, most recently with the AlphaGo program [15]. We shortly

introduce Go and compare it with chess where MCTS did not achieve success.

The basic rules of Go are simple. The standard board is a 9×9 (for beginners) or 19×19 grid. Two players, black and white, alternate in their moves, each placing a single stone of their color on an intersection of lines. By surrounding the stones of the opponent a player captures all the surrounded stones. The game ends when both players agree to end and the winner is the player with the greater sum of controlled territory and captured stones. The rules are well explained in detail at http://playgo.to/iwtg/en/.

A Go player can observe that unlike in Chess there is seldom a quick way to lose, as a loss of a small part of territory may be reversed, however losing an important piece is hardly reversible. This gives us an important example of games where MCTS performs well (in Go it is worth exploring a plethora of moves) and where it does not (e.g. in Chess, due to the trap states described above). It seems plausible this generalizes well to territory versus piece based games.

We end our detour to games with Arimaa, a game specifically designed in 2002 to be easy for humans but hard for computers, for example by allowing more moves per turn. Eventually Arimaa players lost to a program in 2015 [20]. The program uses techniques similar to chess programs like alpha-beta pruning and on top of that employs heuristics inspired by the best human players. See [21] for MCTS related insights into Arimaa.

4 MCTS in MDP Verification

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

These algorithms can be altered by changing the tree node selection heuristic. We suggest and evaluate two formulas, UCB and VCB. Here U_i is the known upper bound (of the sought probability) in the state corresponding to the tree node i, n_i is the number of visits of the node and n is the number of iterations (visits of the root node).

$$UCB_i = U_i + C\sqrt{\frac{2\ln n}{n_i}}$$

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. TODO: However experiments show its performance is similar to UCB.

$$VCB_i = \frac{v_i}{n_i} + C\sqrt{\frac{2\ln n}{n_i}}$$

Furthermore we experimented with a formula which would utilize our confidence in the learned upper bound but experiments show its performance is very bad (TODO: check this is the case on more models).

$$CCB_i = U_i(1 - (U_i - L_i)) + C\sqrt{\frac{2\ln n}{n_i}}$$

4.1 Bounded MCTS

BMCTS is an algorithm which implements the general MCTS scheme in the most straightforard way while maintaining guarantees about the

result. The tree node selection is done using the heuristics described above, in each step of each rollout the action is chosen uniformly at random.

The main distinction is in the update phase. To maintain the lower and upper bounds, backpropagation using the Bellman update is performed from the final state of the rollout to the state corresponding to the selected tree node, and then on the states corresponding to the nodes on the taken tree path. Compare this with the usual 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 seem more relevant for maximizing the sought probability.

Algorithm 7: BMCTS

function BMCTS(s_0)
  Let v_0 be the root of the MCTS tree, with $v_0.state = s_0$.
  while $U(s_0) - L(s_0) > \epsilon$ **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    $\Delta \leftarrow$ Rollout(v_l)
    Backup(v_l, Δ)
  return Action from v_0 to the best node (by some given metric).
function Rollout(s)
function TreePolicy(s)
  repeat
  until $s \neq s_0$
function Backup(s_0, s)
  repeat
    $U(s,a) := \sum_{s' \in S} \Delta(s,a)(s')U(s')$
    $L(s,a) := \sum_{s' \in S} \Delta(s,a)(s')L(s')$
    $s \leftarrow parent(s)$
  until $s \neq s_0$

TODO: Prove AC of this with UCB. This should be easy as eventually this will sample all the paths in the MDP. The problem might be that there is a small chance we will cycle in an end component.

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 as compared to BMCTS but still utilizing exploration which makes it possible to overcome parts of the MDP which are hard for BRTDP.

TODO: How do we deal with end components and why we don't do it. What if we grow a tree into an end component? What happens if a state already is in a tree?

The functions TreePolicy and Backup are implemented in the same way as in Algorithm 7.

TODO: Solve the problem with growing the tree into a MEC by removing the subtree containing nodes of the MEC when BRTDP collapses it.

Algorithm 8: MCTS-BRTDP

function MCTS-BRTDP(s_0)
  Let v_0 be the root of the MCTS tree, with $v_0.state = s_0$.
  while $U(s_0) - L(s_0) > \epsilon$ **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    $\Delta \leftarrow$ an iteration of BRTDP(v_l)
    Backup(v_l, Δ)
  return Action from v_0 to the best node (by some given metric).

TODO: Prove AC with UCB. This should somehow explain what happens if a state already is in a tree?

4.3 UCB in BRTDP

We further present one algorithm which is not based on MCTS but incorporates the UCB exploration term into BRTDP. TODO: pseudocode? probably not, it is just BRTDP where the action is chosen according to UCB, is it AC? can we prove it?

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 [22] (*probabilistic model checker*) 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 [23]. 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 the definitive guide to the language is available online on the PRISM homepage [24].

The PRISM language describes *modules* (interacting actors), 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 actor 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 + ...
```

An example module is described below and 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);
```

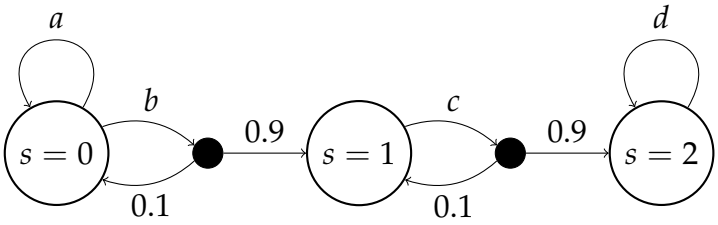


Figure 5.1: Module M

```
    [d] s=2 -> (s'=2);
endmodule
```

A property we might be interested in is the maximum probability of eventually reaching state `s=2`. TODO: Describe the logic. TODO: Warn that only support Pmax and we don't support timed properties (e.g. $F <= x$ is not supported).

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` directory. Java is a required prerequisite. Once built the PRISM binary is available at `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. The `-ex` flag will be described later. Running the command below confirms our expectations: the maximum probability of eventually reaching state `s=2` is 1.

```
./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 UCB exploration constant can be chosen with the `-ucb1constant v`. If not provided the value is set to $1/\sqrt{2}$.

The method can be chosen out of the following: `MCTS_BRTDP`, `MCTS` (TODO: Rename to BMCTS), `BRTDP`, `BRTDP_UCB`, or empty for value iteration. The next state heuristics are `HIGH_PROB`, `MAX_DIFF`. The variation of BRTDP using UCB to select the next action is chosen by adding switch `-next_action 2`.

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 offers three symbolic computation engines based on binary decision diagrams, namely MTBDD (multi-terminal BDD, `-mtbdd` switch), sparse TODO.

All the heuristic methods use a data structure implemented in `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/src/heuristics`.

To represent the MCTS tree we use classes `MCTree.java` and `MCNode.java`. The tree class has an important method `unfold` which asks the model generator to add the next states of a given state to the explicit model and adds them to the tree.

The next state heuristics are implemented in a straightforward way inside directory `nextstate`. The UCB heuristic is implemented in directory `treeheuristic`.

MCTS-BRTDP is implemented in `search/MctsBrtdp.java`. The entry point is method `computeProb`, subsequently `monteCarloTreeSearch` is invoked until the stopping condition is reached (see method `isDone`). Method `monteCarloTreeSearch` first selects and expands a tree state, then uses `exploreAndUpdate` implemented in BRTDP (`search/HeuristicBrtdp.java`) as the default policy and propagates the values using updates to the root.

TODO: Describe treeSelectAndExpand?

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 the BRTDP-way by upper bound.

5.3 Behaviour on Small Models

Describe how do MCTS based methods solve few small models like
* bin tree.
* the BRTDP adversary.
* On the thing below, might be good to compare with BRTDP.

```
init -> cloud of tens of states -> target
     \> easy path               -> target
```

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 provide a brief and incomplete description of several models just to offer basic familiarity and refer the reader to the thorough descriptions on PRISM's website. http://www.prismmodelchecker.org/casestudies/

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.

The description of all the models in the PRISM language can be found inside `tests/reachability/models` in the source codes attached to this thesis.

Zero-configuration networking

Zeroconf[1] model corresponds to a set of computers establishing a network without a given leader. Such leader could be a human setting static IP addresses or a DHCP server which would need to be configured up front – in both cases there is extra work required.

The zero-configuration networking protocol describes how should the computers proceed. Upon connecting to the network a computer picks an IP address at random and broadcasts its choice via an ARP packet called *probe* repeated K times ($K = 4$ by the standard) with two second delay. If another computer responds to one these ARP packets the original sender will then pick another IP address and repeat the process. If it does not receive a response it then broadcasts twice an ARP packet asserting this computer's use of the chosen IP address.

IEEE 1394 FireWire

FireWire is an interface standard for serial bus

Wireless LAN

Randomised Consensus Shared Coin Protocol

mer

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.

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

1. http://www.prismmodelchecker.org/casestudies/zeroconf.php

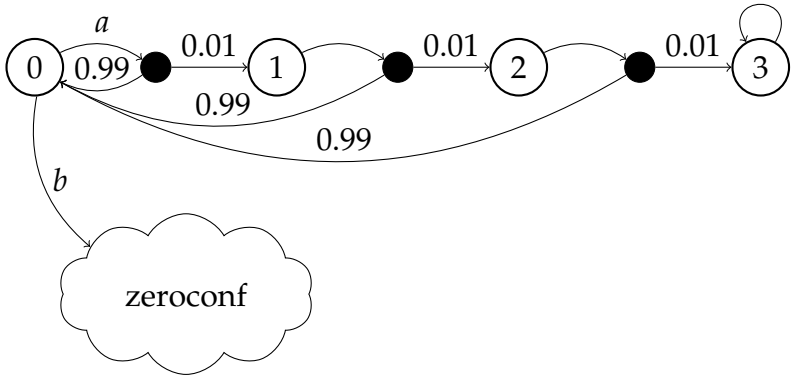


Figure 5.2: Combining zeroconf model with the BRTDP adversary in a branching manner.

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 results of our measurements. In the presented tables rows correspond to models and their configuration, columns to methods and chosen exploration constant. Each table cell contains comma separated running time in seconds and number of visited states of the MDP[2,3]. Each experiment was repeated ten times and the results were averaged[4].

TODO: describe the machine and JVM configuration.

TODO: set up an example runbenchmarks script so that anyone can verify our results right away. TODO: Our full results are stored in measurements attached with the thesis

2. The number of states is, unlike running time, agnostic to implementation details.
3. Value iteration has to construct the whole model in memory so the number of states it visits is the total number of states.
4. There was no significant variance in the measured times so averaging provides representative results.

In Table 5.1 we can see TODO: Table 1, comparison of VI, BRTDP, BRTDP-UCB, MCTS-BRTDP on the PRISM suite (zeroconf, firewire-impl, wlan, coin, leader, mer). Note that MCTS-BRTDP is almost as fast as BRTDP.

TODO: Somewhere where BMCTS is good?

TODO: Table 2, comparison of VI, BRTDP, MCTS-BRTDP on branch-zeroconf, about 6 lines, 3 where BRTDP < MCTS-BRTDP < VI, and 3 where it is the other way. Note that overall MCTS-BRTDP does not do the worst or the best anywhere, making it a universal choice.

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 MCTS-BRTDP is the only method which can actually solve the problem. Moreover it takes only few thousands of states to be explored and a few seconds of running time.

Table 5.1: Comparison on standard PRISM models

	VI	BRTDP, MAX-DIFF	BRTDP-UCB	MCTS-BRTDP
zeroconf $N = 1, K = 10$	204,			
zeroconf $N = x, K = z$	204,			
zeroconf $N = y, K = zz$	204,			
wlan6	204,			
coin...	204,			
leader	204,			
mer	204,			

Table 5.2: caption

branch_zeroconf	BRTDP, MAX-DIFF	VI
$N = 1, K = 10$	46,	204,

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 very strong heuristic which itself often balances well between exploration and exploitation in common MDP models which was our goal when designing the MCTS based algorithm. Our MCTS-BRTDP algorithm performs only slightly worse on the PRISM benchmark suite while it performs significantly better on models hard for BRTDP. Even though value iteration does well on such hard-for-BRTDP models it loses on the PRISM suite, making MCTS-BRTDP a good universal choice for any MDP.

There is still 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. This could be used to suggest new formulas for tree node selection or other variations, however due to complexity of the models this might be a very hard 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.

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. 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.

20. LEWIS, Andy. Game Over, Arimaa. *ICGA Journal*. 2015, vol. 38, no. 1, pp. 55–62. Available from DOI: `10.3233/icg-2015-38108`.

21. JAKL, Tomáš. *Arimaa challenge - comparission study of MCTS versus alpha-beta methods*. 2011. Bachelor Thesis. Charles University, Prague.

22. 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.

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

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