A Monte Carlo AIXI Approximation

Joel Veness JOEL.VENESS@NICTA.COM.AU

University of New South Wales and National ICT Australia

Kee Siong Ng KEESIONG.NG@NICTA.COM.AU

National ICT Australia and The Australian National University

Marcus Hutter MARCUS.HUTTER@ANU.EDU.AU

The Australian National University and National ICT Australia

David Silver SILVER@CS.UALBERTA.CA

University of Alberta

4 September 2009

Abstract

This paper describes a computationally feasible approximation to the AIXI agent, a universal reinforcement learning agent for arbitrary environments. AIXI is scaled down in two key ways: First, the class of environment models is restricted to all prediction suffix trees of a fixed maximum depth. This allows a Bayesian mixture of environment models to be computed in time proportional to the logarithm of the size of the model class. Secondly, the finite-horizon expectimax search is approximated by an asymptotically convergent Monte Carlo Tree Search technique. This scaled down AIXI agent is empirically shown to be effective on a wide class of toy problem domains, ranging from simple fully observable games to small POMDPs. We explore the limits of this approximate agent and propose a general heuristic framework for scaling this technique to much larger problems.

Contents

Keywords

Reinforcement Learning (RL); Context Tree Weighting (CTW); Monte Carlo Tree Search (MCTS); Upper Confidence bounds applied to Trees (UCT); Partially Observable Markov Decision Process (POMDP); Prediction Suffix Trees (PST).

1 Introduction

A main difficulty of doing research in artificial general intelligence has always been in defining exactly what *artificial general intelligence* means. There are many possible definitions [LH07], but the AIXI formulation [Hut05] seems to capture in concrete quantitative terms many of the qualitative attributes usually associated with intelligence.

The general reinforcement learning problem. Consider an agent that exists within some (unknown to the agent) environment. The agent interacts with the environment in cycles. At each cycle, the agent executes an action and receives in turn an observation and a reward. There is no explicit notion of state, neither with respect to the environment nor internally to the agent. The *general reinforcement learning problem* is to construct an agent that, over time, collects as much reward as possible in this setting.

The AIXI agent. The AIXI agent is a mathematical solution to the general reinforcement learning problem. The AIXI setup mirrors that of the general reinforcement problem, however the environment is assumed to be an unknown but computable function; i.e. the observations and rewards received by the agent given its actions can be computed by a Turing machine. Furthermore, the AIXI agent is assumed to exist for a finite, but arbitrarily large amount of time. The AIXI agent results from a synthesis of two ideas:

1. the use of a finite-horizon expectimax operation from sequential decision theory for action selection; and
2. an extension of Solomonoff's universal induction scheme [Sol64] for future prediction in the agent context.

More formally, let $U(q, a_1a_2 \dots a_n)$ denote the output of a universal Turing machine U supplied with program q and input $a_1a_2 \dots a_n$, $m \in \mathbb{N}$ a finite lookahead horizon, and $\ell(q)$ the length in bits of program q. The action picked by AIXI at time t, having executed actions $a_1a_2 \dots a_{t-1}$ and received the sequence of observation-reward pairs $o_1r_1o_2r_2 \dots o_{t-1}r_{t-1}$ from the environment, is given by:

$$a_t^* = \arg\max_{a_t} \sum_{o_tr_t} \dots \max_{a_{t+m}} \sum_{o_{t+m}r_{t+m}} [r_t + \cdots + r_{t+m}] \sum_{q:U(q,a_1 \dots a_{t+m}) = o_1r_1 \dots o_{t+m}r_{t+m}} 2^{-\ell(q)}. \qquad (1)$$

Intuitively, the agent considers the sum of the total reward over all possible futures (up to m steps ahead), weighs each of them by the complexity of programs (consistent with the agent's past) that can generate that future, and then picks the action that maximises expected future rewards. Equation (1) embodies in one line the major ideas of Bayes, Ockham, Epicurus, Turing, von Neumann, Bellman, Kolmogorov, and Solomonoff. The AIXI agent is rigorously shown in [Hut05] to be optimal in different senses of the word. (Technically, AIXI is Pareto optimal and 'self-optimising' in different classes of environment.) In particular, the AIXI agent will rapidly learn an accurate model of the environment and proceed to act optimally to achieve its goal.

The AIXI formulation also takes into account stochastic environments because Equation (1) can be shown to be formally equivalent to the following expression:

$$a_t^* = \arg\max_{a_t} \sum_{o_t r_t} \ldots \max_{a_{t+m}} \sum_{o_{t+m} r_{t+m}} [r_t + \cdots + r_{t+m}] \sum_{\rho \in \mathcal{M}} 2^{-K(\rho)} \rho(o_1 r_1 \ldots o_{t+m} r_{t+m} \,|\, a_1 \ldots a_{t+m}), \tag{2}$$

where $\rho(o_1 r_1 \ldots o_{t+m} r_{t+m} \,|\, a_1 \ldots a_{t+m})$ is the probability of $o_1 r_1 \ldots o_{t+m} r_{t+m}$ given actions $a_1 \ldots a_{t+m}$. Class $\mathcal{M}$ consists of all enumerable chronological semimeasures [Hut05], which includes all computable ρ, and $K(\rho)$ denotes the Kolmogorov complexity of ρ [LV08].

An accessible overview of the AIXI agent can be found in [Leg08]. A complete description of the agent is given in [Hut05].

AIXI as a principle. The AIXI formulation is best understood as a rigorous *definition* of optimal decision making in general *unknown* environments, and not as an algorithmic solution to the general AI problem. (AIXI after all, is only asymptotically computable.) As such, its role in general AI research should be viewed as, for example, the same way the minimax and empirical risk minimisation principles are viewed in decision theory and statistical machine learning research. These principles define what is optimal behaviour if computational complexity is not an issue, and can provide important theoretical guidance in the design of practical algorithms. It is in this light that we see AIXI. This paper is an attempt to scale AIXI down to produce a practical agent that can perform well in a wide range of different, unknown and potentially noisy environments.

Approximating AIXI. As can be seen in Equation (1), there are two parts to AIXI. The first is the expectimax search into the future which we will call *planning*. The second is the use of a Bayesian mixture over Turing machines to predict future observations and rewards based on past experience; we will call that *learning*. Both parts need to be approximated for computational tractability. There are many different approaches one can try. In this paper, we opted to use a generalised version of the UCT algorithm [KS06] for planning and a generalised version of the Context Tree Weighting algorithm [WST95] for learning. This harmonious combination of ideas, together with the attendant theoretical and experimental results, form the main contribution of this paper.

Paper organisation. The paper is organised as follows. Section 2 describes the basic agent setting and discusses some design issues. Section 3 then presents a Monte Carlo Tree Search procedure that we will use to approximate the expectimax operation in AIXI. This is followed by a description of the context tree weighting algorithm and how it can be generalised for use in the agent setting in Section 4. We put the two ideas together in Section 5 to form our agent algorithm. Theoretical and experimental results are then presented in Sections 6 and 7. We end with a discussion of related work and other topics in Section 8.

2 The Agent Setting and Some Design Issues

Notation. A string $x_1x_2\dots x_n$ of length n is denoted by $x_{1:n}$. The prefix $x_{1:j}$ of $x_{1:n}$, $j \le n$, is denoted by $x_{\le j}$ or $x_{<j+1}$. The notation generalises for blocks of symbols: e.g. $ax_{1:n}$ denotes $a_1x_1a_2x_2\dots a_nx_n$ and $ax_{<j}$ denotes $a_1x_1a_2x_2\dots a_{j-1}x_{j-1}$. The empty string is denoted by ϵ. The concatenation of two strings s and r is denoted by sr.

Agent setting. The (finite) action, observation, and reward spaces are denoted by $\mathcal{A}, \mathcal{O}$, and $\mathcal{R}$ respectively. Also, $\mathcal{X}$ denotes the joint perception space $\mathcal{O} \times \mathcal{R}$.

Definition 1. *A history is a string $h \in (\mathcal{A} \times \mathcal{X})^n$, for some $n \ge 0$. A partial history is the prefix of some history.*

The set of all history strings of maximum length n will be denoted by $(\mathcal{A} \times \mathcal{X})^{\le n}$.

The following definition states that the agent's model of the environment takes the form of a probability distribution over possible observation-reward sequences conditioned on actions taken by the agent.

Definition 2. *An environment model ρ is a sequence of functions $\{\rho_0, \rho_1, \dots\}$, $\rho_n \colon \mathcal{A}^n \to$ Density $(\mathcal{X}^n)$, that satisfies:*

1. $\forall a_{1:n} \forall x_{<n} : \rho_n(x_{<n} \,|\, a_{<n}) = \sum_{x_n \in \mathcal{X}} \rho_n(x_{1:n} \,|\, a_{1:n})$
2. $\forall a_{<n} \forall x_{<n} : \rho_n(x_{<n} \,|\, a_{<n}) > 0.$

The first condition (called the chronological condition in [Hut05]) captures the natural constraint that action a_n has no effect on observations made before it. The second condition enforces the requirement that the probability of every possible observation-reward sequence is non-zero. This ensures that conditional probabilities are always defined. It is not a serious restriction in practice, as probabilities can get arbitrarily small. For convenience, we drop the index t in ρ_t from here onwards.

Given an environment model ρ, we have the following identities:

$$\rho(x_n \,|\, ax_{<n}a_n) = \frac{\rho(x_{1:n} \,|\, a_{1:n})}{\rho(x_{<n} \,|\, a_{<n})} \tag{3}$$

$$\rho(x_{1:n} \,|\, a_{1:n}) = \rho(x_1 \,|\, a_1)\rho(x_2 \,|\, a_1x_1a_2)\cdots\rho(x_n \,|\, ax_{<n}a_n) \tag{4}$$

Reward, policy and value functions. We represent the notion of *reward* as a numeric value that represents the magnitude of instantaneous pleasure experienced by the agent at any given time step. Our agent is a hedonist; its goal is to accumulate as much reward as it can during its lifetime. More precisely, in our setting the agent is only interested in maximising its future reward up to a fixed, finite, but arbitrarily large horizon $m \in \mathbb{N}$.

In order to act rationally, our agent seeks a *policy* that will allow it to maximise its future reward. Formally, a policy is a function that maps a history to an action. If we define $R_k(aor_{\le t}) := r_k$ for $1 \le k \le t$, then we have the following definition for the expected future value of an agent acting under a particular policy:

Definition 3. *Given history $ax_{<t}$, the m-horizon expected future reward of an agent acting under policy $\pi \colon (\mathcal{A} \times \mathcal{X})^{\leq t+m} \to \mathcal{A}$ with respect to an environment model ρ is:*

$$v_\rho^m(\pi, ax_{<t}) := \mathbb{E}_{x_{t:t+m} \sim \rho} \left[\sum_{i=t}^{t+m} R_i(ax_{\leq t+m}) \right], \tag{5}$$

where for $t \leq k \leq t+m$, $a_k := \pi(ax_{<k})$. The quantity $v_\rho^m(\pi, ax_{<t}a_t)$ is defined similarly, except that a_t is now no longer defined by π.

The optimal policy π^* is the policy that maximises Equation (5). The maximal achievable expected future reward of an agent with history $h \in (\mathcal{A} \times \mathcal{X})^{t-1}$ in environment ρ looking m steps ahead is $V_\rho^m(h) := v_\rho^m(\pi^*, h)$. It is easy to see that

$$V_\rho^m(h) = \max_{a_t} \sum_{x_t} \rho(x_t \,|\, ha_t) \cdots \max_{a_{t+m}} \sum_{x_{t+m}} \rho(x_{t+m} \,|\, hax_{t:t+m-1}a_{t+m}) \left[\sum_{i=t}^{t+m} r_i \right]. \tag{6}$$

All of our subsequent efforts can be viewed as attempting to define an algorithm that determines a policy as close to the optimal policy as possible given reasonable resource constraints. Our agent is *model based*: we learn a model of the environment and use it to estimate the future value of our various actions at each time step. These estimates allow the agent to make an approximate best action given limited computational resources.

We now discuss some high-level design issues before presenting our algorithm in the next section.

Perceptual aliasing. A major problem in general reinforcement learning is *perceptual aliasing* [Chr92], which refers to the situation where the instantaneous perceptual information (a single observation in our setting) does not provide enough information for the agent to act optimally. This problem is closely related to the question of what constitutes a state, an issue we discuss next.

State vs history based agents. A *Markov* state [SB98] provides a sufficient statistic for all future observations, and therefore provides sufficient information to represent optimal behaviour. No perceptual aliasing can occur with a Markov state. In Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs) all underlying *environmental states* are Markov.

A *compact* state representation is often assumed to generalise well and therefore enable efficient learning and planning. A common approach in reinforcement learning (RL) [SB98] is to approximate the environmental state by using a small number of handcrafted features. However, this approach requires both that the environmental state is known, and that sufficient domain knowledge is available to select the features.

In the general RL problem, neither the states nor the domain properties are known in advance. One approach to general RL is to find a compact representation of state that is approximately Markov [McC96, Sha07, SJR04, ST04], or a compact representation of state that maximises some performance criterion [Hut09b, Hut09a]. In practice, a Markov representation is rarely achieved in complex domains, and these methods must introduce some approximation, and therefore some level of perceptual aliasing.

In contrast, we focus on learning and planning methods that use the agent's history as its representation of state. A history representation can be generally applied without any domain knowledge. Importantly, a history representation requires no approximation and introduces no aliasing: each history is a perfect Markov state (or k-Markov for length k histories). In return for these advantages, we give up on compactness. The number of states in a history representation is exponential in the horizon length (or k for length k histories), and many of these histories may be equivalent. Nevertheless, a history representation can sometimes be more compact than the environmental state, as it ignores extraneous factors that do not affect the agent's direct observations.

Predictive environment models. In order to form non-trivial plans that span multiple time steps, our agent needs to be able to predict the effects of its interaction with the environment. If a model of the environment is known, search-based methods offer one way of generating such plans. However, a general RL agent does not start with a model of the environment; it must learn one over time. Our agent builds an approximate model of the true environment from the experience it gathers when interacting with the real world, and uses it for online planning.

Approximation via online planning. If the problem is small, model-based RL methods such as Value Iteration for MDPs can easily derive an optimal policy. However this is not appropriate for the larger problems more typical of the real world. Local search is one way to address this problem. Instead of solving the problem in its entirety, an approximate solution is computed before each decision is made. This approach has met with much success on difficult decision problems within the game playing research community and on large-sized POMDPs [RPPCD08].

Scalability. The general RL problem is extremely difficult. On any real world problem, an agent is necessarily restricted to making approximately correct decisions. One of the distinguishing features of sophisticated heuristic decision making frameworks, such as those used in computer chess or computer go, is the ability of these frameworks to provide acceptable performance on hardware ranging from mobile phones through to supercomputers. To take advantage of the fast-paced advances in computer technology, we claim that *a good autonomous agent framework should naturally and automatically scales with increasing computational resources*. Both the learning and planning components of our approximate AIXI agent have been designed with scalability in mind.

Anytime decision making. One of the key resources in real world decision making is time. As we are interested in a practical general agent framework, it is imperative that our agent be able to make good approximate decisions *on demand*. Different application domains have different real-world time constraints. We seek an agent framework that can make good, approximate decisions given anything from 10 milliseconds to 10 days thinking time per action.

3 Monte Carlo Tree Search with Model Updates

In this section we describe Predictive UCT, a Monte Carlo Tree Search (MCTS) technique for stochastic, partially observable domains that uses an incrementally updated environment model ρ to predict and evaluate the possible outcomes of future action sequences.

The Predictive UCT algorithm is a straightforward generalisation of the UCT algorithm [KS06], a Monte Carlo planning algorithm that has proven effective in solving large state space discounted, or finite horizon MDPs. The generalisation requires two parts:

- The use of an environment model that is conditioned on the agent's history, rather than a Markov state.
- The updating of the environment model during search. This is essential for the algorithm to utilise the extra information an agent will have at a hypothetical, *particular* future time point.

The generalisation involves a change in perspective which has significant practical ramifications in the context of general RL agents. Our extensions to UCT allow Predictive UCT, in combination with a sufficiently powerful predictive environment model ρ, to implicitly take into account the value of information in search and be applicable to partially observable domains.

Overview. Predictive UCT is a best-first Monte Carlo Tree Search technique that iteratively constructs a search tree in memory. The tree is composed of two interleaved types of nodes: decision nodes and chance nodes. These correspond to the alternating max and $\sum$ operations in expectimax. Each node in the tree corresponds to a (partial) history h. If h ends with an action, it is a chance node; if h ends with an observation, it is a decision node. Each node contains a statistical estimate of the future reward.

Initially, the tree starts with a single decision node containing $|\mathcal{A}|$ children. Much like in existing MCTS methods [CWU$^+$08], there are four conceptual phases to a single iteration of Predictive UCT. The first is the *selection* phase, where the search tree is traversed from the root node to an existing leaf chance node n. The second is the *expansion* phase, where a new decision node is added as a child to n. The third is the *simulation* phase, where a playout policy in conjunction with the environment model ρ is used to sample a possible future path from n until a fixed distance from the root is reached. Finally, the *backpropagation* phase updates the value estimates for each node on the reverse trajectory leading back to the root. Whilst time remains, these four conceptual operations are repeated. Once the time limit is reached, an approximate best action can be selected by looking at the value estimates of the children of the root node.

During the selection phase, action selection at decision nodes is done using a policy that balances exploration and exploitation. This policy has two main effects:

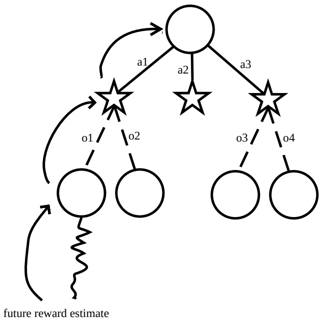

Figure 1: A Predictive UCT search tree

- to move the estimates of the future reward towards the maximum attainable future reward if the agent acted optimally.
- to cause asymmetric growth of the search tree towards areas that have high predicted reward, implicitly pruning large parts of the search space.

The future reward at leaf nodes is estimated by choosing actions according to a heuristic policy until a total of m actions have been made by the agent, where m is the search horizon. This heuristic estimate helps the agent to focus its exploration on useful parts of the search tree, and in practice allows for a much larger horizon than a brute-force expectimax search.

Predictive UCT builds a sparse search tree in the sense that observations are only added to chance nodes once they have been generated along some sample path. A full expectimax search tree would not be sparse; each possible stochastic outcome will be represented by a distinct node in the search tree. For expectimax, the branching factor at chance nodes is thus $|O|$, which means that searching to even moderate sized m is intractable.

Figure 1 shows an example Predictive UCT tree. Chance nodes are denoted with stars. Decision nodes are denoted by circles. The dashed lines from a star node indicate that not all of the children have been expanded. The squiggly line at the base of the leftmost leaf denotes the execution of a playout policy. The arrows proceeding up from this node indicate the flow of information back up the tree; this is defined in more detail in Section 3.

Action selection at decision nodes. A decision node will always contain $|\mathcal{A}|$ distinct children, all of whom are chance nodes. Associated with each decision node representing a particular history h will be a value function estimate, $\hat{V}(h)$. During the selection phase, a child will need to be picked for further exploration. Action selection in MCTS poses a classic exploration/exploitation dilemma. On one hand we need to allocate enough visits to all children to ensure that we have accurate estimates for them, but on the other hand we need to allocate enough visits to the maximal action to ensure convergence of the node to the value of the maximal child node.

Like UCT, Predictive UCT recursively uses the UCB policy [Aue02] from the n-armed bandit setting at each decision node to determine which action needs further exploration. Although the uniform logarithmic regret bound no longer carries across from the bandit setting, the UCB policy has been shown to work well in practice in complex domains such as Computer Go [GW06] and General Game Playing [FB08]. This policy has the advantage of ensuring that at each decision node, every action eventually gets explored an infinite number of times, with the best action being selected exponentially more often than actions of lesser utility.

Definition 4. *The visit count $T(h)$ of a decision node h is the number of times h has been sampled by the Predictive UCT algorithm. The visit count of the chance node found by taking action a at h is defined similarly, and is denoted by $T(ha)$.*

Definition 5. *Suppose m is the search horizon and each single time-step reward is bounded in the interval $[\alpha, \beta]$. Given a node representing a history h in the search tree, the action picked by the UCB action selection policy is:*

$$a_{UCB}(h) := \arg\max_{a \in \mathcal{A}} \begin{cases} \frac{1}{m(\beta-\alpha)} \hat{V}(ha) + C \sqrt{\frac{\log(T(h))}{T(ha)}} & \text{if } T(ha) > 0; \\ \infty & \text{otherwise}, \end{cases} \tag{7}$$

where $C \in \mathbb{R}$ is a positive parameter that controls the ratio of exploration to exploitation. If there are multiple maximal actions, one is chosen uniformly at random.

Note that we need a linear scaling of $\hat{V}(ha)$ in Definition 5 because the UCB policy is only applicable for rewards confined to the $[0, 1]$ interval.

Chance nodes. Chance nodes follow immediately after an action is selected from a decision node. Each chance node ha following a decision node h contains an estimate of the future utility denoted by $\hat{V}(ha)$. Also associated with the chance node ha is a density $\rho(\cdot \mid ha)$ over observation-reward pairs.

After an action a is performed at node h, $\rho(\cdot \mid ha)$ is sampled once to generate the next observation-reward pair or. If o has not been seen before, the node hao is added as a child of ha. We will use the notation $\mathcal{O}_{ha}$ to denote the subset of $\mathcal{O}$ representing the children of partial history ha created so far.

Estimating future reward at leaf nodes. If a leaf decision node is encountered at depth $k < m$ in the tree, a means of estimating the future reward for the remaining $m - k$ time steps is required. The agent applies its heuristic playout function Π to estimate the sum of future rewards $\sum_{i=k}^{m} r_i$. A particularly simple, pessimistic baseline playout function is Π_{random}, which chooses an action uniformly at random at each time step.

A more sophisticated playout function that uses action probabilities estimated from previously taken real-world actions could potentially provide a better estimate. The quality of the actions suggested by such a predictor can be expected to improve over time, since it is trying to predict actions that are chosen by the agent after a Predictive UCT search. This powerful and intuitive method of constructing a generic heuristic will be explored further in a subsequent section.

Asymptotically, the heuristic playout policy makes no contribution to the value function estimates of Predictive UCT. When the remaining depth is zero, the playout policy always returns zero reward. As the number of simulations tends to infinity, the structure of the Predictive UCT search tree is equivalent to the exact depth m expectimax tree with high probability. This implies that the asymptotic value function estimates of Predictive UCT are invariant to the choice of playout function. However, when search time is limited, the choice of playout policy will be a major determining factor of the overall performance of the agent.

Reward backup. After the selection phase is completed, a path of nodes $n_1 n_2 \dots n_k$, $k \leq m$, will have been traversed from the root of the search tree n_1 to some leaf n_k. For each $1 \leq j \leq k$, the statistics maintained for (partial) history h_{n_j} associated with node n_j will be updated as follows:

$$\hat{V}(h_{n_j}) \leftarrow \frac{T(h_{n_j})}{T(h_{n_j}) + 1} \hat{V}(h_{n_j}) + \frac{1}{T(h_{n_j}) + 1} \sum_{i=j}^{m} r_i \tag{8}$$

$$T(h_{n_j}) \leftarrow T(h_{n_j}) + 1 \tag{9}$$

Note that the same backup equations are applied to both decision and chance nodes.

Incremental model updating. Recall from Definition 2 that an environment model ρ is a sequence of functions $\{\rho_0, \rho_1, \rho_2, \dots\}$, where $\rho_t : \mathcal{A}^t \rightarrow \mathit{Density}\ (\mathcal{X}^t)$. When invoking the Sample routine to decide on an action, many hypothetical future experiences will be generated, with ρ_t being used to simulate the environment at time t. For the algorithm to work well in practice, we need to be able to perform the following two operations in time sublinear with respect to the length of the agent's entire experience string.

- Update - given $\rho_t(x_{1:t} \,|\, a_{1:t})$, a_{t+1}, and x_{t+1}, produce $\rho_{t+1}(x_{1:t+1} \,|\, a_{1:t+1})$
- Revert - given $\rho_{t+1}(x_{1:t+1} \,|\, a_{1:t+1})$, recover $\rho_t(x_{1:t} \,|\, a_{1:t})$

The revert operation is needed to restore the environment model to ρ_t after each simulation to time $t + m$ is performed. In Section 4, we will show how these requirements can be met efficiently by a certain kind of Bayesian mixture over a rich model class.

Pseudocode. We now give the pseudocode of the entire Predictive UCT algorithm.

Algorithm 1 is responsible for determining an approximate best action. Given the current history h, it first constructs a search tree containing estimates $\hat{V}_\rho^m(ha)$ for each $a \in \mathcal{A}$, and then selects a maximising action. An important property of Algorithm 1 is that it is *anytime*; an approximate best action is always available, whose quality improves with extra computation time.

Algorithm 1 Predictive UCT(ρ, h, m)

```
Require: An environment model $\rho$
Require: A history $h$
Require: A search horizon $m \in \mathbb{N}$
Initialise($\Psi$)
repeat
    Sample($\Psi, h, m$)
    $\rho \leftarrow$ Revert($\rho, m$)
until out of time
return BestAction($\Psi, h$)
```

For simplicity of exposition, Initialise can be understood to simply clear the entire search tree Ψ. In practice, it is possible to carry across information from one time step to another. If Ψ_t is the search tree obtained at the end of time t, and aor is the agent's actual action and experience at time t, then we can keep the subtree rooted at node $\Psi_t(hao)$ in Ψ_t and make that the search tree Ψ_{t+1} for use at the beginning of the next time step. The remainder of the nodes in Ψ_t can then be deleted.

As a Monte Carlo Tree Search routine, Algorithm 1 is embarrassingly parallel. The main idea is to concurrently invoke the Sample routine whilst providing appropriate locking mechanisms for the nodes in the search tree. An efficient parallel implementation is beyond the scope of the paper, but it is worth noting that ideas [CWH08] applicable to high performance Monte Carlo Go programs are easily transferred to our setting.

Algorithm 2 implements a single run through some trajectory in the search tree. It uses the SelectAction routine to choose moves at interior nodes, and invokes the playout policy at unexplored leaf nodes. After a complete path of length m is completed, the recursion takes care that every visited node along the path to the leaf is updated as per Section 3.

The action chosen by SelectAction is specified by the UCB policy described in Definition 5. If the selected child has not been explored before, then a new node is added to the search tree. The constant C is a parameter that is used to control the shape of the search tree; lower values of C create deep, selective search trees, whilst higher values lead to shorter, bushier trees.

4 Extensions of Context Tree Weighting

Context Tree Weighting (CTW) [WST95, WST97] is a theoretically well-motivated online binary sequence prediction algorithm that works well in practice [BEYY04]. It is an online Bayesian model averaging algorithm that computes a mixture of all prediction suffix trees [RST96] of a given bounded depth, with higher prior weight given to simpler models. We examine in this section several extensions of CTW needed for its use in the context of agents. Along the way, we will describe the CTW algorithm in detail.

Algorithm 2 SAMPLE(ρ, Ψ, h, m)

Require: An environment model ρ
Require: A search tree Ψ
Require: A (partial) history h
Require: A remaining search horizon $m \in \mathbb{N}$

```
if $m = 0$ then
    return 0
else if $\Psi(h)$ is a chance node then
    Generate $(o, r)$ from $\rho(or \mid h)$
    Create node $\Psi(hor)$ if $T(hor) = 0$
    reward ← $r$ + SAMPLE($\rho, \Psi, hor, m - 1$)
else if $T(h) = 0$ then
    reward ← PLAYOUT($\rho, h, m$)
else
    $a$ ← SELECTACTION($\Psi, h$)
    reward ← SAMPLE($\rho, \Psi, ha, m$)
end if
$\hat{V}(h) \leftarrow \frac{1}{T(h)+1}[reward + T(h)\hat{V}(h)]$
$T(h) \leftarrow T(h) + 1$
return reward
```

Algorithm 3 SELECTACTION(Ψ, h)

Require: A search tree Ψ
Require: A history h
Require: An exploration/exploitation constant C

```
$\mathcal{U} = \{a \in \mathcal{A} \colon T(ha) = 0\}$
if $\mathcal{U} \neq \{\}$ then
    Pick $a \in \mathcal{U}$ uniformly at random
    Create node $\Psi(ha)$
    return a
else
    return $\arg\max_{a \in \mathcal{A}} \left\{ \frac{1}{m(\beta-\alpha)} \hat{V}(ha) + C \sqrt{\frac{\log(T(h))}{T(ha)}} \right\}$
end if
```

Action-conditional CTW. We first look at how CTW can be generalised for use as environment models (Definition 2), which are functions of the form $\rho_n : \mathcal{A}^n \to Density\ (\mathcal{X}^n)$. This means we need an extension of CTW that, incrementally, takes as input a sequence of actions and produces as output successive conditional probabilities over observations and rewards. The high-level view of the algorithm is as follows: we process observations and rewards one bit at a time using standard CTW, but bits representing actions are simply appended to the input sequence without updating the context tree. The algorithm is now

Algorithm 4 PLAYOUT(ρ, h, m)

```
Require: An environment model $\rho$
Require: A history $h$
Require: A remaining search horizon $m \in \mathbb{N}$
Require: A playout function $\Pi$
reward $\leftarrow 0$
for $i = 1$ to $m$ do
    Generate $a$ from $\Pi(h)$
    Generate $(o, r)$ from $\rho(or \,|\, ha)$
    reward $\leftarrow$ reward $+ r$
    $h \leftarrow haor$
end for
return reward
```

described in detail. If we drop the action sequence throughout the following description, the algorithm reduces to the standard CTW algorithm.

Krichevsky-Trofimov estimator. We start with a brief review of the KT estimator [KT81] for Bernoulli distributions. Given a binary string $y_{1:t}$ with a zeroes and b ones, the KT estimate of the probability of the next symbol is as follows:

$$\Pr_{kt}(Y_{t+1} = 1 \,|\, y_{1:t}) := \frac{b + 1/2}{a + b + 1} \tag{10}$$

$$\Pr_{kt}(Y_{t+1} = 0 \,|\, y_{1:t}) := 1 - \Pr_{kt}(Y_{t+1} = 1 \,|\, y_{1:t}). \tag{11}$$

The KT estimator is obtained via a Bayesian analysis by putting a $(\frac{1}{2}, \frac{1}{2})$-Beta prior on the parameter of the Bernoulli distribution. From (10)-(11), we obtain the following expression for the block probability of a string:

$$\Pr_{kt}(y_{1:t}) = \Pr_{kt}(y_1 \,|\, \epsilon)\Pr_{kt}(y_2 \,|\, y_1) \cdots \Pr_{kt}(y_t \,|\, y_{1:t-1}).$$

Given a binary string s, one can establish that $\Pr_{kt}(s)$ depends only on the number of zeroes a_s and ones b_s in s. If we let $0^a 1^b$ denote a string with a zeroes and b ones then:

$$\Pr_{kt}(s) = \Pr_{kt}(0^{a_s} 1^{b_s}) = \frac{1/2(1 + 1/2) \cdots (a_s - 1/2) 1/2(1 + 1/2) \cdots (b_s - 1/2)}{(a_s + b_s)!}. \tag{12}$$

We write $\Pr_{kt}(a, b)$ to denote $\Pr_{kt}(0^a 1^b)$ in the following. The quantity $\Pr_{kt}(a, b)$ can be updated incrementally as follows:

$$\Pr_{kt}(a + 1, b) = \frac{a + 1/2}{a + b + 1} \Pr_{kt}(a, b) \tag{13}$$

$$\Pr_{kt}(a, b + 1) = \frac{b + 1/2}{a + b + 1} \Pr_{kt}(a, b), \tag{14}$$

with the base case being $\Pr_{kt}(0, 0) = 1$.

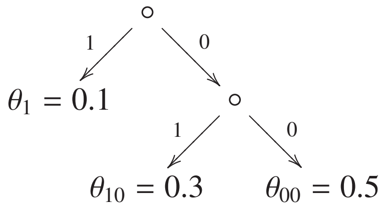

Figure 2: An example prediction suffix tree

Prediction Suffix Trees. We next describe prediction suffix trees, which are a form of variable-order Markov models.

Definition 6. *A prediction suffix tree (PST) is a pair* (M, Θ) *satisfying the following:*

1. *M is a binary tree where the left and right edges are labelled 1 and 0 respectively; and*
2. *associated with each leaf node l in M is a probability distribution over* $\{0, 1\}$ *parameterised by* $\theta_l \in \Theta$ *(the probability of 1).*

We call M the model of the PST and Θ *the parameter of the PST, in accordance with the terminology of [WST95], .*

A prediction suffix tree (M, Θ) maps each binary string $y_{1:n}$, where $n \geq$ the depth of M, to a probability distribution over $\{0, 1\}$ in the natural way: we traverse the model M by moving left or right at depth d depending on whether the bit y_{n-d} is one or zero until we reach a leaf node l in M, at which time we return θ_l. For example, the PST shown in Figure 2 maps the string 110 to $\theta_{10} = 0.3$. At the root node (depth 0), we move right because $y_3 = 0$. We then move left because $y_{3-1} = 1$. We say θ_{10} is the distribution associated with the string 110. Sometimes we need to refer to the leaf node holding the distribution associated with a string h; we denote that by $M(h)$, where M is the model of the PST used to process the string.

To use a prediction suffix tree of depth d for binary sequence prediction, we start with the distribution $\theta_l := \Pr_{kt}(1 \mid \epsilon) = 1/2$ at each leaf node l of the tree. The first d bits $y_{1:d}$ of the input sequence are set aside for use as an initial context and the variable h denoting the bit sequence seen so far is set to $y_{1:d}$. We then repeat the following steps as long as needed:

1. predict the next bit using the distribution θ_h associated with h;
2. observe the next bit y, update θ_h using Formula (10) by incrementing either a or b according to the value of y, and then set $h := hy$.

Action-conditional PST. The above describes how a PST is used for binary sequence prediction. In the agent setting, we reduce the problem of predicting history sequences with general non-binary alphabets to that of predicting the bit representations of those sequences. Further, we only ever condition on actions and this is achieved by appending bit representations of actions to the input sequence without a corresponding update of the KT estimators. These ideas are now formalised.

For convenience, we will assume without loss of generality that $|\mathcal{A}| = 2^{l_\mathcal{A}}$ and $|\mathcal{X}| = 2^{l_\mathcal{X}}$ for some $l_\mathcal{A}, l_\mathcal{X} > 0$. Given $a \in \mathcal{A}$, we denote by $[\![a]\!] = a[1, l_\mathcal{A}] = a[1]a[2] \dots a[l_\mathcal{A}] \in \{0,1\}^{l_\mathcal{A}}$ the bit representation of a. Observation and reward symbols are treated similarly. Further, the bit representation of a symbol sequence $x_{1:t}$ is denoted by $[\![x_{1:t}]\!] = [\![x_1]\!][\![x_2]\!] \dots [\![x_t]\!]$. The ith bit in $[\![x_{1:t}]\!]$ is denoted by $[\![x_{1:t}]\!][i]$ and the first l bits of $[\![x_{1:t}]\!]$ is denoted by $[\![x_{1:t}]\!][1,l]$.

To do action-conditional prediction using a PST, we again start with $\theta_l := \Pr_{kt}(1 \,|\, \epsilon) = 1/2$ at each leaf node l of the tree. We also set aside a sufficiently long initial portion of the binary history sequence corresponding to the first few cycles to initialise the variable h as usual. The following steps are then repeated as long as needed:

1. set $h := h[\![a]\!]$, where a is the current selected action;
2. for $i := 1$ to $l_\mathcal{X}$ do
   (a) predict the next bit using the distribution θ_h associated with h;
   (b) observe the next bit $x[i]$, update θ_h using Formula (10) according to the value of $x[i]$, and then set $h := hx[i]$.

Now, let M be the model of a prediction suffix tree, $L(M)$ the leaf nodes of M, $a_{1:t} \in \mathcal{A}^t$ an action sequence, and $x_{1:t} \in \mathcal{X}^t$ an observation-reward sequence. We have the following expression for the probability of $x_{1:t}$ given M and $a_{1:t}$:

$$\begin{aligned}\Pr(x_{1:t} \,|\, M, a_{1:t}) &= \prod_{i=1}^{t} \prod_{j=1}^{l_\mathcal{X}} \Pr(x_i[j] \,|\, M, [\![ax_{<i}a_i]\!]x_i[1, j-1]) \\ &= \prod_{n \in L(M)} \Pr_{kt}([\![x_{1:t}]\!]_{|n}), \end{aligned} \tag{15}$$

where $[\![x_{1:t}]\!]_{|n}$ is the (non-contiguous) subsequence of $[\![x_{1:t}]\!]$ that ended up in leaf node n in M. More precisely,

$$[\![x_{1:t}]\!]_{|n} := [\![x_{1:t}]\!][l_1][\![x_{1:t}]\!][l_2] \cdots [\![x_{1:t}]\!][l_n],$$

where $1 \leq l_1 < l_2 < \cdots < l_n \leq t$ and, for each i, $i \in \{l_1, \dots l_n\}$ iff $M([\![x_{1:t}]\!][1, i-1]) = n$.

The above deals with action-conditional prediction using a single PST. We now show how we can perform action-conditional prediction using a Bayesian mixture of PSTs in an efficient way. First, we need a prior distribution on models of PSTs.

A prior on models of PSTs. Our prior, containing an Ockham-like bias favouring simple models, is derived from a natural prefix coding of the tree structure of a PST. The coding scheme works as follows: given a model of a PST of maximum depth D, a preorder traversal of the tree is performed. Each time an internal node is encountered, we write down 1. Each time a leaf node is encountered, we write a 0 if the depth of the leaf node is less than D; otherwise we write nothing. For example, if $D = 3$, the code for the model shown in Figure 2 is 10100; if $D = 2$, the code for the same model is 101. The cost $\Gamma_D(M)$ of a model M is the length of its code, which is given by the number of nodes in M minus the number of leaf nodes in M of depth D. One can show that

$$\sum_{M \in C_D} 2^{-\Gamma_D(M)} = 1,$$

where C_D is the set of all models of prediction suffix trees with depth at most D; i.e. the prefix code is complete. We remark that the above is another way of describing the coding scheme in [WST95]. We use $2^{-\Gamma_D(\cdot)}$, which penalises large trees, to determine the prior weight of each PST model.

Context trees. The following is a key ingredient of the (action-conditional) CTW algorithm.

Definition 7. *A context tree of depth D is a perfect binary tree of depth D where the left and right edges are labelled 1 and 0 respectively and attached to each node (both internal and leaf) is a probability on* $\{0, 1\}^*$.

The node probabilities in a context tree are estimated from data using KT estimators as follows. We update a context tree with the history sequence similarly to the way we use a PST, except that

1. the probabilities at each node in the path from the root to a leaf traversed by an observed bit is updated; and
2. we maintain block probabilities using Equations (12)-(14) instead of conditional probabilities (Equation (10)) like in a PST. (This is done for computational reasons to ease the calculation of the posterior probabilities of models in the algorithm.)

The process can be best understood with an example. Figure 3 (left) shows a context tree of depth two. For expositional reasons, we show binary sequences at the nodes; the node probabilities are computed from these. Initially, the binary sequence at each node is empty. Suppose 1001 is the history sequence. Setting aside the first two bits 10 as an initial context, the tree in the middle of Figure 3 shows what we have after processing the third bit 0. The tree on the right is the tree we have after processing the fourth bit 1. In practice, we of course only have to store the counts of zeros and ones instead of complete subsequences at each node because, as we saw earlier in (12), $\Pr_{kt}(s) = \Pr_{kt}(a_s, b_s)$. Since the node probabilities are completely determined by the input sequence, we shall henceforth speak unambiguously about *the* context tree after seeing a sequence.

The context tree of depth D after seeing a sequence h has the following important properties:

1. the model of every PST of depth at most D can be obtained from the context tree by pruning off appropriate subtrees and treating them as leaf nodes;

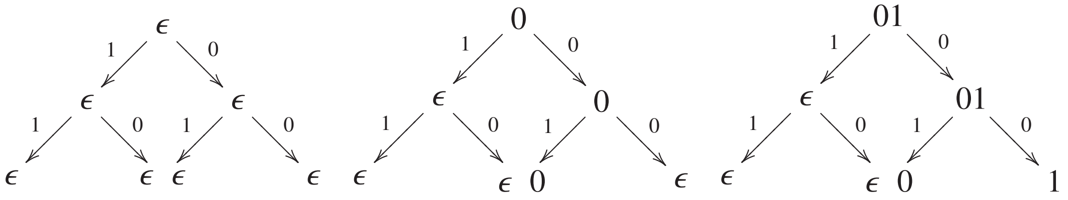

Figure 3: A depth-2 context tree (left); trees after processing two bits (middle and right)

2. the block probability of h as computed by each PST of depth at most D can be obtained from the node probabilities of the context tree via Equation (15).

These properties, together with an application of the distributive law, form the basis of the highly efficient (action-conditional) CTW algorithm. We now formalise these insights.

Weighted probabilities. We first need to define the weighted probabilities at each node of the context tree. Suppose $a_{1:t}$ is the action sequence and $x_{1:t}$ is the observation-reward sequence. Let $[\![x_{1:t}]\!]_{|n}$ be the (non-contiguous) subsequence of $[\![x_{1:t}]\!]$ that ended up in node n of the context tree. The weighted probability P^n_w of each node n in the context tree is defined inductively as follows:

$$P^n_w([\![x_{1:t}]\!]_{|n} \mid [\![a_{1:t}]\!])$$
$$:= \begin{cases} \Pr_{kt}([\![x_{1:t}]\!]_{|n}) & \text{if } n \text{ is a leaf node} \\ \frac{1}{2}\Pr_{kt}([\![x_{1:t}]\!]_{|n}) + \frac{1}{2}P^{n_l}_w([\![x_{1:t}]\!]_{|n_l} \mid [\![a_{1:t}]\!])P^{n_r}_w([\![x_{1:t}]\!]_{|n_r} \mid [\![a_{1:t}]\!]) & \text{otherwise,} \end{cases}$$

where n_l and n_r are the left and right children of n respectively. Note that the set of sequences $\{\, [\![x_{1:t}]\!]_{|n}$: n is a node in the context tree $\}$ has a dependence on the action sequence $[\![a_{1:t}]\!]$.

If n is a node at depth d in a tree, we denote by $p(n) \in \{0,1\}^d$ the path description to node n in the tree.

Lemma 1 ([WST95])**.** *Let D be the depth of the context tree. For each node n in the context tree at depth d, we have for all $a_{1:t} \in \mathcal{A}^t$, for all $x_{1:t} \in \mathcal{X}^t$,*

$$P^n_w([\![x_{1:t}]\!]_{|n} \mid [\![a_{1:t}]\!]) = \sum_{M \in C_{D-d}} 2^{-\Gamma_{D-d}(M)} \prod_{l \in L(M)} \Pr_{kt}([\![x_{1:t}]\!]_{|p(n)p(l)}), \tag{16}$$

where $[\![x_{1:t}]\!]_{|p(n)p(l)}$ is the (non-contiguous) subsequence of $[\![x_{1:t}]\!]$ that ended up in the node with path description $p(n)p(l)$ in the context tree.

Proof. The proof proceeds by induction on d. The statement is clearly true for the leaf nodes at depth D. Assume now the statement is true for all nodes at depth $d+1$, where $0 \le d < D$. Consider a node n at depth d. Letting $\overline{d} = D - d$, we have

$$P^n_w([\![x_{1:t}]\!]_{|n} \mid [\![a_{1:t}]\!])$$
$$= \frac{1}{2}\Pr_{kt}([\![x_{1:t}]\!]_{|n}) + \frac{1}{2}P^{n_l}_w([\![x_{1:t}]\!]_{|n_l} \mid [\![a_{1:t}]\!])P^{n_r}_w([\![x_{1:t}]\!]_{|n_r} \mid [\![a_{1:t}]\!])$$

$$
\begin{aligned}
&= \frac{1}{2}\mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|n}) + \frac{1}{2}\left[\sum_{M \in C_{\overline{d+1}}} 2^{-\Gamma_{\overline{d+1}}(M)} \prod_{l \in L(M)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(n_l)p(l)})\right] \\
&\qquad\qquad \left[\sum_{M \in C_{\overline{d+1}}} 2^{-\Gamma_{\overline{d+1}}(M)} \prod_{l \in L(M)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(n_r)p(l)})\right] \\
&= \frac{1}{2}\mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|n}) \ + \sum_{M_1 \in C_{\overline{d+1}}} \sum_{M_2 \in C_{\overline{d+1}}} 2^{-(\Gamma_{\overline{d+1}}(M_1)+\Gamma_{\overline{d+1}}(M_2)+1)}. \\
&\qquad\qquad \left[\prod_{l \in L(M_1)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(n_l)p(l)})\right]\left[\prod_{l \in L(M_2)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(n_r)p(l)})\right] \\
&= \frac{1}{2}\mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|n}) + \sum_{\widehat{M_1M_2} \in C_{\overline{d}}} 2^{-\Gamma_{\overline{d}}(\widehat{M_1M_2})} \prod_{l \in L(\widehat{M_1M_2})} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(n)p(l)}) \\
&= \sum_{M \in C_{D-d}} 2^{-\Gamma_{D-d}(M)} \prod_{l \in L(M)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(n)p(l)}),
\end{aligned}
$$

where $\widehat{M_1M_2}$ denotes the tree in $C_{\overline{d}}$ whose left and right subtrees are M_1 and M_2 respectively. □

CTW as an optimal Bayesian mixture predictor. A corollary of Lemma 1 is that at the root node λ of the context tree we have

$$
\begin{aligned}
P_w^\lambda([\![x_{1:t}]\!] \mid [\![a_{1:t}]\!]) &= \sum_{M \in C_D} 2^{-\Gamma_D(M)} \prod_{l \in L(M)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|p(l)}) && (17) \\
&= \sum_{M \in C_D} 2^{-\Gamma_D(M)} \prod_{l \in L(M)} \mathrm{Pr}_{kt}([\![x_{1:t}]\!]_{|l}) && (18) \\
&= \sum_{M \in C_D} 2^{-\Gamma_D(M)} \mathrm{Pr}(x_{1:t} \mid M, a_{1:t}), && (19)
\end{aligned}
$$

where the last step follows from Equation (15). Note carefully that $[\![x_{1:t}]\!]_{|p(l)}$ in line (17) denotes the subsequence of $[\![x_{1:t}]\!]$ that ended in the node pointed to by $p(l)$ in the context tree but $[\![x_{1:t}]\!]_{|l}$ in line (18) denotes the subsequence of $[\![x_{1:t}]\!]$ that ended in the leaf node l in M if M is used as the only model to process $[\![x_{1:t}]\!]$. Equation (19) shows that the quantity computed by the (action-conditional) CTW algorithm is exactly a Bayesian mixture of (action-conditional) PSTs.

The weighted probability P_w^λ is a block probability. To recover the conditional probability of x_t given $ax_{<t}a_t$, we simply evaluate

$$
P_w^\lambda([\![x_t]\!] \mid [\![ax_{<t}a_t]\!]) = \frac{P_w^\lambda([\![x_{1:t}]\!] \mid [\![a_{1:t}]\!])}{P_w^\lambda([\![x_{<t}]\!] \mid [\![a_{<t}]\!])},
$$

which follows directly from Equation (3). To sample from this conditional probability, we simply sample the individual bits of x_t one by one. For brevity, we will sometimes use the following notation for P_w^λ:

$$\Upsilon(x_{1:t} \,|\, a_{1:t}) := P_w^\lambda([\![x_{1:t}]\!] \,|\, [\![a_{1:t}]\!])$$
$$\Upsilon(x_t \,|\, ax_{<t}a_t) := P_w^\lambda([\![x_t]\!] \,|\, [\![ax_{<t}a_t]\!]).$$

In summary, to do action-conditional prediction using a context tree, we set aside a sufficiently long initial portion of the binary history sequence corresponding to the first few cycles to initialise the variable h and then repeat the following steps as long as needed:

1. set $h := h[\![a]\!]$, where a is the current selected action;
2. for $i := 1$ to $l_{\mathcal{X}}$ do
   (a) predict the next bit using the weighted probability P_w^λ;
   (b) observe the next bit $x[i]$, update the context tree using h and $x[i]$, calculate the new weighted probability P_w^λ, and then set $h := hx[i]$.

Note that in practice, the context tree need only be constructed incrementally as needed. The depth of the context tree can thus take on non-trivial values. This memory requirement of maintaining a context tree is discussed further in Section 7.

Reversing an update. As explained in Section 3, the REVERT operation is performed many times during search and it needs to be efficient. Saving and restoring a copy of the context tree is unsatisfactory. Luckily, the block probability estimated by CTW using a context depth of D at time t can be recovered from the block probability estimated at time $t+m$ in $O(mD)$ operations in a rather straightforward way. Alternatively, a copy on write implementation can be used to modify the context tree during the simulation phase.

Predicate CTW. As foreshadowed in [Bun92, HS97], the CTW algorithm can be generalised to work with rich logical tree models [BD98, KW01, Llo03, Ng05, LN07] in place of prediction suffix trees. A full description of this extension, especially the part on predicate definition/enumeration and search, is beyond the scope of the paper and will be reported elsewhere. Here we outline the main ideas and point out how the extension can be used to incorporate useful background knowledge into our agent.

Definition 8. *Let $\mathcal{P} = \{p_0, p_1, \ldots, p_m\}$ be a set of predicates (boolean functions) on histories $h \in (\mathcal{A} \times \mathcal{X})^n, n \geq 0$. A $\mathcal{P}$-model is a binary tree where each internal node is labelled with a predicate in $\mathcal{P}$ and the left and right outgoing edges at the node are labelled True and False respectively. A $\mathcal{P}$-tree is a pair $(M_{\mathcal{P}}, \Theta)$ where $M_{\mathcal{P}}$ is a $\mathcal{P}$-model and associated with each leaf node l in $M_{\mathcal{P}}$ is a probability distribution over $\{0,1\}$ parameterised by $\theta_l \in \Theta$.*

A $\mathcal{P}$-tree $(M_{\mathcal{P}}, \Theta)$ represents a function g from histories to probability distributions on $\{0,1\}$ in the usual way. For each history h, $g(h) = \theta_{l_h}$, where l_h is the leaf node reached by pushing h down the model $M_{\mathcal{P}}$ according to whether it satisfies the predicates at the internal nodes and $\theta_{l_h} \in \Theta$ is the distribution at l_h.

The use of general predicates on histories in $\mathcal{P}$-trees is a powerful way of extending the notion of a "context" in applications. To begin with, it is easy to see that, with a suitable choice of predicate class $\mathcal{P}$, both prediction suffix trees (Definition 6) and looping suffix trees [HJ06] can be represented as $\mathcal{P}$-trees. Much more background contextual information can be provided in this way to the agent to aid learning and action selection.

The following is a generalisation of Definition 7.

Definition 9. *Let $\mathcal{P} = \{p_0, p_1, \ldots, p_m\}$ be a set of predicates on histories. A $\mathcal{P}$-context tree is a perfect binary tree of depth $m + 1$ where*

1. *each internal node at depth i is labelled by $p_i \in \mathcal{P}$ and the left and right outgoing edges at the node are labelled True and False respectively; and*
2. *attached to each node (both internal and leaf) is a probability on $\{0, 1\}^*$.*

We remark here that the (action-conditional) CTW algorithm can be generalised to work with $\mathcal{P}$-context trees in a natural way, and that a result analogous to Lemma 1 but with respect to a much richer class of models can be established. A proof of a similar result is in [HS97]. Section 7 describes some experiments showing how predicate CTW can help in more difficult problems.

5 Putting it All Together

We now describe how the entire agent is constructed. At a high level, the combination is simple. The agent uses the action-conditional (predicate) CTW predictor presented in Section 4 as a model Υ of the (unknown) environment. At each time step, the agent first invokes the Predictive UCT routine to estimate the value of each action. The agent then picks an action according to some standard exploration/exploitation strategy, such as ϵ-Greedy or Softmax [SB98]. It then receives an observation-reward pair from the environment which is then used to update Υ. Communication between the agent and the environment is done via binary codings of action, observation, and reward symbols. Figure 4 gives an overview of the agent/environment interaction loop.

It is worth noting that, in principle, the AIXI agent does not need to explore according to any heuristic policy. This is since the value of information, in terms of expected future reward, is implicitly captured in the expectimax operation defined in Equations (1) and (2). Theoretically, ignoring all computational concerns, it is sufficient just to choose a large horizon and pick the action with the highest expected value at each timestep.

Unfortunately, this result does not carry over to our approximate AIXI agent. In practice, the true environment will not be contained in our restricted model class, nor will we perform enough Predictive UCT simulations to converge to the optimal expectimax action, nor will the search horizon be as large as the agent's maximal lifespan. Thus, the exploration/exploitation dilemma is a non-trivial problem for our agent. We found that the standard heuristic solutions to this problem, such as ϵ-Greedy and Softmax exploration, were sufficient for obtaining good empirical results. We will revisit this issue in Section 7.

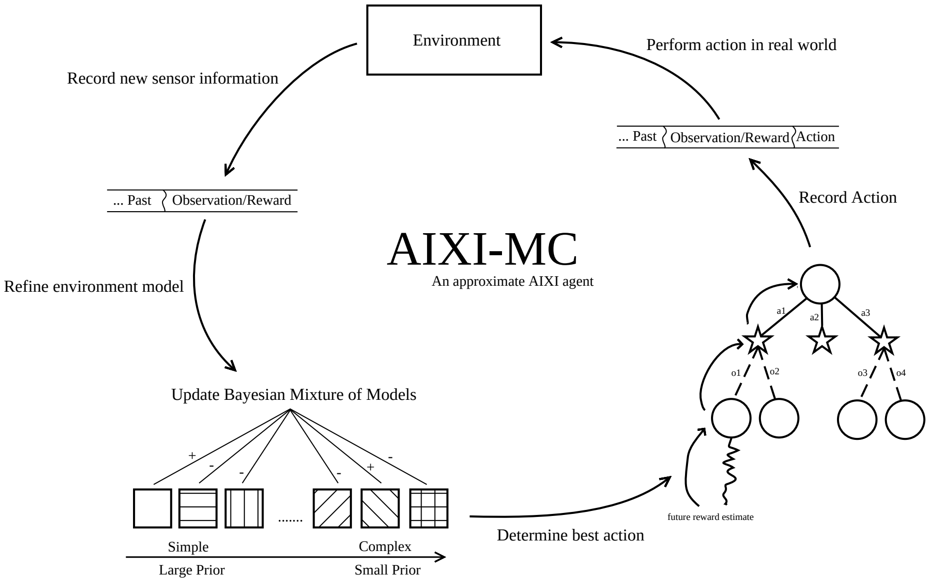

Figure 4: The AIXI-MC agent loop

6 Theoretical Results

Some theoretical properties of our algorithm are now explored.

Model class approximation. We first study the relationship between Υ and the universal predictor in AIXI. Using Υ in place of ρ in Equation (6), the optimal action for an agent at time t, having experienced $ax_{1:t-1}$, is given by

$$\begin{aligned}
a_t^* &= \arg\max_{a_t} \sum_{x_t} \frac{\Upsilon(x_{1:t} \,|\, a_{1:t})}{\Upsilon(x_{<t} \,|\, a_{<t})} \cdots \max_{a_{t+m}} \sum_{x_{t+m}} \frac{\Upsilon(x_{1:t+m} \,|\, a_{1:t+m})}{\Upsilon(x_{<t+m} \,|\, a_{<t+m})} \left[\sum_{i=t}^{t+m} r_i \right] \\
&= \arg\max_{a_t} \sum_{x_t} \cdots \max_{a_{t+m}} \sum_{x_{t+m}} \left[\sum_{i=t}^{t+m} r_i \right] \prod_{i=t}^{t+m} \frac{\Upsilon(x_{1:i} \,|\, a_{1:i})}{\Upsilon(x_{<i} \,|\, a_{<i})} \\
&= \arg\max_{a_t} \sum_{x_t} \cdots \max_{a_{t+m}} \sum_{x_{t+m}} \left[\sum_{i=t}^{t+m} r_i \right] \frac{\Upsilon(x_{1:t+m} \,|\, a_{1:t+m})}{\Upsilon(x_{<t} \,|\, a_{<t})} \\
&= \arg\max_{a_t} \sum_{x_t} \cdots \max_{a_{t+m}} \sum_{x_{t+m}} \left[\sum_{i=t}^{t+m} r_i \right] \Upsilon(x_{1:t+m} \,|\, a_{1:t+m}) \\
&= \arg\max_{a_t} \sum_{x_t} \cdots \max_{a_{t+m}} \sum_{x_{t+m}} \left[\sum_{i=t}^{t+m} r_i \right] \sum_{M \in C_D} 2^{-\Gamma_D(M)} \Pr(x_{1:t+m} \,|\, M, a_{1:t+m}). \qquad (20)
\end{aligned}$$

Contrast (20) now with Equation (2) which we reproduce here:

$$a_t = \arg\max_{a_t} \sum_{x_t} \ldots \max_{a_{t+m}} \sum_{x_{t+m}} \left[\sum_{i=t}^{t+m} r_i \right] \sum_{\rho \in \mathcal{M}} 2^{-K(\rho)} \rho(x_{1:t+m} \,|\, a_{1:t+m}), \qquad (21)$$

where $\mathcal{M}$ is the class of all enumerable chronological semimeasures, and $K(\rho)$ denotes the Kolmogorov complexity of ρ [Hut05]. The two expressions share a prior that enforces a bias towards simpler models. The main difference is in the subexpression describing the mixture over the model class. AIXI uses a mixture over all enumerable chronological semimeasures. This is scaled down to a mixture of prediction suffix trees in our setting. Although the model class used in AIXI is completely general, it is also incomputable. Our approximation has restricted the model class to gain the desirable computational properties of CTW. As indicated in Section 4, the model class C_D can be significantly enlarged by using predicates without sacrificing the efficient computability of mixtures.

Convergence to true environment. We show in this section that if there is a good model of the (unknown) environment in the class C_D, then CTW will 'find' it. We need the following entropy inequality.

Lemma 2 ([Hut05]). *Let $\{y_i\}$ and $\{z_i\}$ be two probability distributions, i.e. $y_i \geq 0, z_i \geq 0$, and $\sum_i y_i = \sum_i z_i = 1$. Then we have*

$$\sum_i (y_i - z_i)^2 \leq \sum_i y_i \ln \frac{y_i}{z_i}.$$

Theorem 1. *Let μ be the true environment model. The μ-expected squared difference of μ and Υ is bounded as follows. For all $n \in \mathbb{N}$, for all $a_{1:n}$,*

$$\sum_{k=1}^{n} \sum_{x_{1:k}} \mu(x_{<k} \,|\, a_{<k}) \Big(\mu(x_k \,|\, ax_{<k}a_k) - \Upsilon(x_k \,|\, ax_{<k}a_k) \Big)^2$$
$$\leq \min_{M \in C_D} \Big\{ \Gamma_D(M) \ln 2 + D_{KL}(\mu(\cdot \,|\, a_{1:n}) \,\|\, \Pr(\cdot \,|\, M, a_{1:n})) \Big\},$$

where $D_{KL}(\cdot \,\|\, \cdot)$ is the KL divergence of two distributions.

Proof. We adapt a proof from [Hut05, §5.1.3].

$$\begin{aligned}
&\sum_{k=1}^{n} \sum_{x_{1:k}} \mu(x_{<k} \,|\, a_{<k}) \Big(\mu(x_k \,|\, ax_{<k}a_k) - \Upsilon(x_k \,|\, ax_{<k}a_k) \Big)^2 \\
&= \sum_{k=1}^{n} \sum_{x_{<k}} \mu(x_{<k} \,|\, a_{<k}) \sum_{x_k} \Big(\mu(x_k \,|\, ax_{<k}a_k) - \Upsilon(x_k \,|\, ax_{<k}a_k) \Big)^2 \\
&\leq \sum_{k=1}^{n} \sum_{x_{<k}} \mu(x_{<k} \,|\, a_{<k}) \sum_{x_k} \mu(x_k \,|\, ax_{<k}a_k) \ln \frac{\mu(x_k \,|\, ax_{<k}a_k)}{\Upsilon(x_k \,|\, ax_{<k}a_k)} && \text{[by Lemma 2]} \\
&= \sum_{k=1}^{n} \sum_{x_{1:k}} \mu(x_{1:k} \,|\, a_{1:k}) \ln \frac{\mu(x_k \,|\, ax_{<k}a_k)}{\Upsilon(x_k \,|\, ax_{<k}a_k)} && \text{[by Eq. (3)]} \\
&= \sum_{k=1}^{n} \sum_{x_{1:k}} \Big(\sum_{x_{k+1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \Big) \ln \frac{\mu(x_k \,|\, ax_{<k}a_k)}{\Upsilon(x_k \,|\, ax_{<k}a_k)} && \text{[by Defn. 2]}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=1}^{n} \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \ln \frac{\mu(x_k \,|\, ax_{<k}a_k)}{\Upsilon(x_k \,|\, ax_{<k}a_k)} \\
&= \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \sum_{k=1}^{n} \ln \frac{\mu(x_k \,|\, ax_{<k}a_k)}{\Upsilon(x_k \,|\, ax_{<k}a_k)} \\
&= \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \ln \frac{\mu(x_{1:n} \,|\, a_{1:n})}{\Upsilon(x_{1:n} \,|\, a_{1:n})} && \text{[by Eq. (4)]} \\
&= \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \ln \left[\frac{\mu(x_{1:n} \,|\, a_{1:n})}{\Pr(x_{1:n} \,|\, M, a_{1:n})} \frac{\Pr(x_{1:n} \,|\, M, a_{1:n})}{\Upsilon(x_{1:n} \,|\, a_{1:n})} \right] && \text{[arbitrary } M \in C_D\text{]} \\
&= \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \ln \frac{\mu(x_{1:n} \,|\, a_{1:n})}{\Pr(x_{1:n} \,|\, M, a_{1:n})} + \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \ln \frac{\Pr(x_{1:n} \,|\, M, a_{1:n})}{\Upsilon(x_{1:n} \,|\, a_{1:n})} \\
&\leq D_{KL}(\mu(\cdot \,|\, a_{1:n}) \,||\, \Pr(\cdot \,|\, M, a_{1:n})) + \sum_{x_{1:n}} \mu(x_{1:n} \,|\, a_{1:n}) \ln \frac{\Pr(x_{1:n} \,|\, M, a_{1:n})}{2^{-\Gamma_D(M)} \Pr(x_{1:n} \,|\, M, a_{1:n})} && \text{[by Eq. (19)]} \\
&= D_{KL}(\mu(\cdot \,|\, a_{1:n}) \,||\, \Pr(\cdot \,|\, M, a_{1:n})) + \Gamma_D(M) \ln 2.
\end{aligned}
$$

Since the inequality holds for arbitrary $M \in C_D$, it holds for the minimising M. □

If the KL divergence between μ and the best model in C_D is finite, then Theorem 1 implies $\Upsilon(x_k \,|\, ax_{<k}a_k)$ will converge rapidly to $\mu(x_k \,|\, ax_{<k}a_k)$ for $k \to \infty$ with μ-probability 1. The contrapositive of the statement tells us that if Υ fails to predict the environment well, then there is no good model in C_D. This result provides the motivation for looking at ways of enriching the model class in Section 8.

Consistency of Predictive UCT. Let μ be the true underlying environment. We now establish the link between the expectimax value $V_\mu^m(h)$ and its estimate $\hat{V}_\Upsilon^m(h)$ computed by the Predictive UCT algorithm using Υ as the environment model.

In [KS06], the authors show that the UCT algorithm is consistent in finite horizon MDPs and derive finite sample bounds on the estimation error due to sampling. By interpreting histories as Markov states, our general agent problem reduces to a finite horizon MDP and the results of [KS06] are now directly applicable. Restating the main consistency result in our notation, we have

$$\forall \epsilon \forall h \lim_{T(h) \to \infty} \Pr\left(|V_\Upsilon^m(h) - \hat{V}_\Upsilon^m(h)| \leq \epsilon \right) = 1. \tag{22}$$

Further, the probability that a suboptimal action (with respect to $V_\Upsilon^m(\cdot)$) is picked by Predictive UCT goes to zero in the limit. Details of this analysis can be found in [KS06].

Theorem 1 above in conjunction with [Hut05, Thm.5.36] implies $V_\Upsilon^m(h) \to V_\mu^m(h)$, as long as there exists a model in the model class that approximates the unknown environment μ well. This, and the consistency (22) of the Predictive UCT algorithm, imply that $\hat{V}_\Upsilon^m(h)$ will converge to $V_\mu^m(h)$.

Domain	Aliasing	Noisy O	Noisy $\mathcal{A}$	Uninformative O
1d-maze	yes	no	no	yes
Cheese Maze	yes	no	no	no
Tiger	yes	yes	no	no
Extended Tiger	yes	yes	no	no
4×4 Grid	yes	no	no	yes
TicTacToe	no	no	no	no
Biased Rock-Paper-Scissor	no	yes	yes	no
Partially Observable Pacman	yes	no	no	no

Table 1: Domain characteristics

7 Experimental Results

In this section we evaluate our algorithm on a number of pre-existing domains. We have chosen domains that, from the agent's perspective, have noisy perceptions, partial information, and inherent stochastic elements. In particular, we will focus on learning and approximately solving some benchmark POMDPs. The planning problem (i.e. computation of the optimal policy given the full POMDP model) associated with these POMDPs were considered challenging in the mid-nineties but can now be solved easily. We stress here that our requirement of having to learn the environment model, as well as solve the planning problem, *significantly* increases the difficulty of these problems.

As we shall see, our agent achieves state-of-the-art performance in both generality (eight separate problems with different characteristics are attempted) and optimality (the agent converges to the optimal policy in seven cases, and exhibits good scaling properties in the remaining case).

Our test domains are now described in detail. Their characteristics are summarised in Table 1.

1d-maze. The 1d-maze is a simple problem from [CKL94]. The agent begins at a random, non-goal location within a 1×4 maze. There is a choice of two actions: left or right. Each action transfers the agent to the adjacent cell if it exists, otherwise it has no effect. If the agent reaches the third cell from the left, it receives a reward of 1. Otherwise it receives a reward of 0. The distinguishing feature of this problem is that the observations are *uninformative*; every observation is the same regardless of the agent's actual location.

Cheese maze. This well known problem is due to [McC96]. The agent is a mouse inside a two dimensional maze seeking a piece of cheese. The agent has to choose one of four actions: move up, down, left or right. If the agent bumps into a wall, it receives a penalty of -10. If the agent finds the cheese, it receives a reward of 10. Each movement into a free cell gives a penalty of -1. The problem is depicted graphically in Figure 5. The number in each cell represents the decimal equivalent of the four bit binary observation the mouse receives in each cell. The problem exhibits perceptual aliasing in that a single observation is potentially ambiguous.

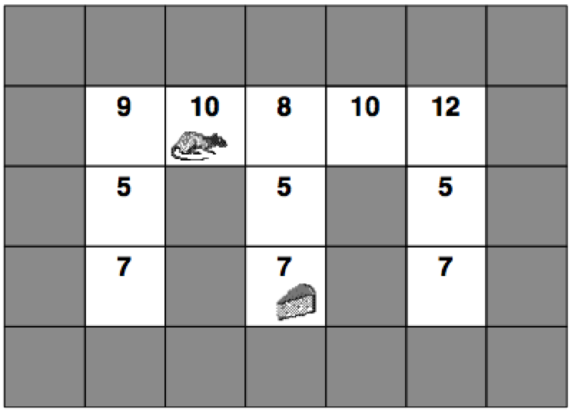

Figure 5: The cheese maze

Tiger. This is another familiar domain from [KLC95]. The environment dynamics are as follows: a tiger and a pot of gold are hidden behind one of two doors. Initially the agent starts facing both doors. The agent has a choice of one of three actions: listen, open the left door, or open the right door. If the agent opens the door hiding the tiger, it suffers a -100 penalty. If it opens the door with the pot of gold, it receives a reward of 10. If the agent performs the listen action, it receives a penalty of -1 and an observation that correctly describes where the tiger is with 0.85 probability.

Extended Tiger. The problem setting is similar to Tiger, except that now the agent begins sitting down on a chair. The actions available to the agent are: stand, listen, open the left door, and open the right door. Before an agent can successfully open one of the two doors, it must stand up. However, the listen action only provides information about the tiger's whereabouts when the agent is sitting down. Thus it is necessary for the agent to plan a more intricate series of actions before it sees the optimal solution. The reward structure is slightly modified from the simple Tiger problem, as now the agent gets a reward of 30 when finding the pot of gold.

4×4 Grid. The agent is restricted to a 4×4 grid world. It can move either up, down, right or left. If the agent moves into the bottom right corner, it receives a reward of 1, and it is randomly teleported to one of the remaining 15 cells. If it moves into any cell other than the bottom right corner cell, it receives a reward of 0. If the agent attempts to move into a non-existent cell, it remains in the same location. Like the 1d-maze, this problem is also uninformative but on a much larger scale. Although this domain is simple, it does require some subtlety on the part of the agent. The correct action depends on what the agent has tried before at previous time steps. For example, if the agent has repeatedly moved right and not received a positive reward, then the chances of it receiving a positive reward by moving down are increased.

TicTacToe. In this domain, the agent plays repeated games of TicTacToe against an opponent who moves randomly. If the agent wins the game, it receives a reward of 2. If there is a draw, the agent receives a reward of 1. A loss penalises the agent by -2. If the agent makes an illegal move, by moving on top of an already filled square, then it receives a reward of -3. A legal move that does not end the game earns no reward.

Biased Rock-Paper-Scissor. This domain is taken from [FMRW09]. The agent repeatedly plays Rock-Paper-Scissor against an opponent that has a slight, predictable bias in its strategy. If the opponent has won a round by playing rock on the previous cycle, it will always play rock at the next timestep; otherwise it will pick an action uniformly at random. The agent's observation is the most recently chosen action of the opponent. It receives a reward of 1 for a win, 0 for a draw and -1 for a loss.

Partially Observable PacMan. This domain is a partially observable version of the classic PacMan game. The agent must navigate a 17×17 maze and eat the food pellets that are distributed across the maze. Four ghosts roam the maze. They move initially at random, until there is a Manhattan distance of 5 between them and PacMan, whereupon they will aggressively pursue PacMan for a short duration. The maze structure and game are the same as the original arcade game, however the PacMan agent is hampered by partial observability. PacMan is unaware of the maze structure and only receives a 4-bit observation describing the wall configuration at its current location. It also does not know the exact location of the ghosts, receiving only 4-bit observations indicating whether a ghost is visible (via direct line of sight) in each of the four cardinal directions. In addition, the location of the food pellets is unknown except for a 3-bit observation that indicates whether food can be smelt within a Manhattan distance of 2, 3 or 4 from PacMan's location, and another 4-bit observation indicating whether there is food in its direct line of sight. A final single bit indicates whether PacMan is under the effects of a power pill. At the start of each episode, a food pellet is placed down with probability 0.5 at every empty location on the grid. The agent receives a penalty of 1 for each movement action, a penalty of 10 for running into a wall, a reward of 10 for each food pellet eaten, a penalty of 50 if it is caught by a ghost, and a reward of 100 for collecting all the food. If multiple such events occur, then the total reward is cumulative, i.e. running into a wall and being caught would give a penalty of 60. The episode resets if the agent is caught or if it collects all the food.

Figure 6 shows a graphical representation of the partially observable PacMan domain. This problem is the largest domain we consider, with an unknown optimal policy. The main purpose of this domain is to show the scaling properties of our agent with respect to a challenging problem.

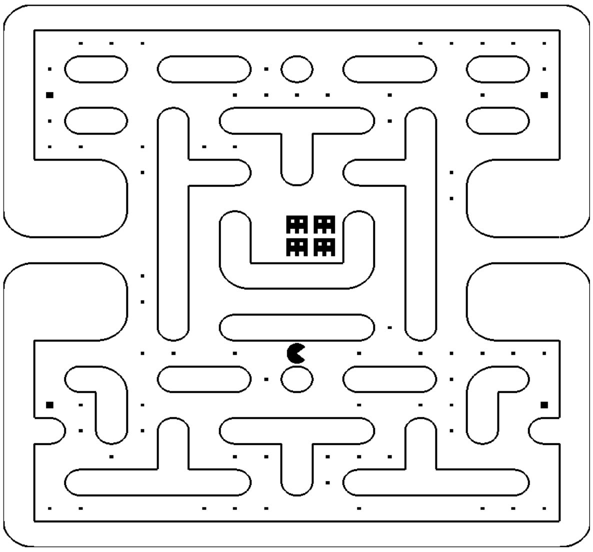

Figure 6: A screenshot (converted to b&w) of the partially observable PacMan domain

Experimental setup. Table 2 outlines the parameters used in each experiment. The sizes of the action and observation spaces are given. The $\mathcal{A}$ bits, $\mathcal{O}$ bits and $\mathcal{R}$ bits parameters specify the number of bits used to encode the action, observation and reward spaces. The context depth parameter D specifies the maximum number of most recent bits used by the action-conditional CTW prediction scheme. The search horizon is specified by the parameter m.

The experimental results are presented in terms of average reward per time step. The key factors of interest are the performance of the agent as it accumulates more real world experience, and the performance of the agent as it is given more thinking time per decision.

All experiments were performed on a dual quad-core Intel 2.53Ghz Xeon. If computational concerns could be ignored, it would be natural to make D as large as possible since CTW is robust against overfitting due to its strong bias towards simple PSTs. There are similar issues with the choice of horizon; ideally the horizon would be as large as possible if we could ignore computational concerns. In practice however, these parameters must be made much smaller for our agent to be tractable on our modest hardware. Section 7 discusses the asymptotic properties of our algorithms. Although the asymptotic behaviour is excellent (essentially linear in D and m in terms of both time and space), our prototype implementation is still pushing the boundaries of what can be done on a present day workstation. There are obvious problems if these parameters are set too small. For example, if the problem is n-Markov but we only use a $D < n$, or if the optimal policy requires planning ahead more than m steps, then we cannot expect the agent to perform optimally.

Domain	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{O}\rvert$	$\mathcal{A}$ bits	$\mathcal{O}$ bits	$\mathcal{R}$ bits	D	m
1d-maze	2	1	1	1	1	32	10
Cheese Maze	4	16	2	4	5	96	8
Tiger	3	3	2	2	7	96	5
Extended Tiger	4	3	2	3	8	96	4
4×4 Grid	4	1	2	1	1	96	12
TicTacToe	9	19683	4	18	3	64	9
Biased Rock-Paper-Scissor	3	3	2	2	2	32	4
Partial Observable Pacman	4	2^{16}	2	16	8	64	8

Table 2: Parameter Configuration

Scaling properties. Our agent has both limited thinking time and a limited amount of time to gather experience in the real world. Potentially, both of these dimensions will affect the agent's performance. This section explores what the agent's performance on different problem domains as we vary the two parameters.

Figure 7 shows the performance of the agent as it accumulates more experience. Two seconds of search time per decision was used for each experiment. The label Age for the horizontal axis refers to the number of cycles that has transpired.

Figure 8 shows the performance of the agent on each problem domain by running it with varying amounts of search. The environment model used for each experiment was learned by the agent from randomly interacting with the environment for $50'000$ timesteps, with the exception of TicTacToe which used a model built from $500'000$ timesteps. Random action selection was used for computational reasons; it allowed large amounts of experience to be gathered quickly. For each data point, the agent is run for 2000 timesteps, using the best action chosen greedily by Predictive UCT. The average reward is then calculated from the performance across these 2000 timesteps.

General discussion. In all cases, given sufficient thinking time and experience, the performance of our agent approaches optimality. Generally speaking, the agent's performance gets better as it acquires more experience and is given more search time per decision. The agent's performance on the tiger domains warrants some discussion.

The behaviour of the agent in the Tiger domain varies as the amount of interaction with the environment increases. Initially, the agent avoids selecting a door, as it is too uncertain about the environment dynamics. However, as it gathers more experience, more sophisticated behaviour emerges; the agent correctly acquires multiple pieces of information before picking a door. If some of the information is contradictory, the agent gathers more information before making its decision.

The performance of the agent in the Extended Tiger domain is sensitive to the number of simulations used by Predictive UCT. As can be seen in Figure 7, two seconds of thinking time were insufficient to act optimally. As indicated by figure 8, optimal behaviour is only achieved when using a minimum of approximately $10'000$ simulations per decision.

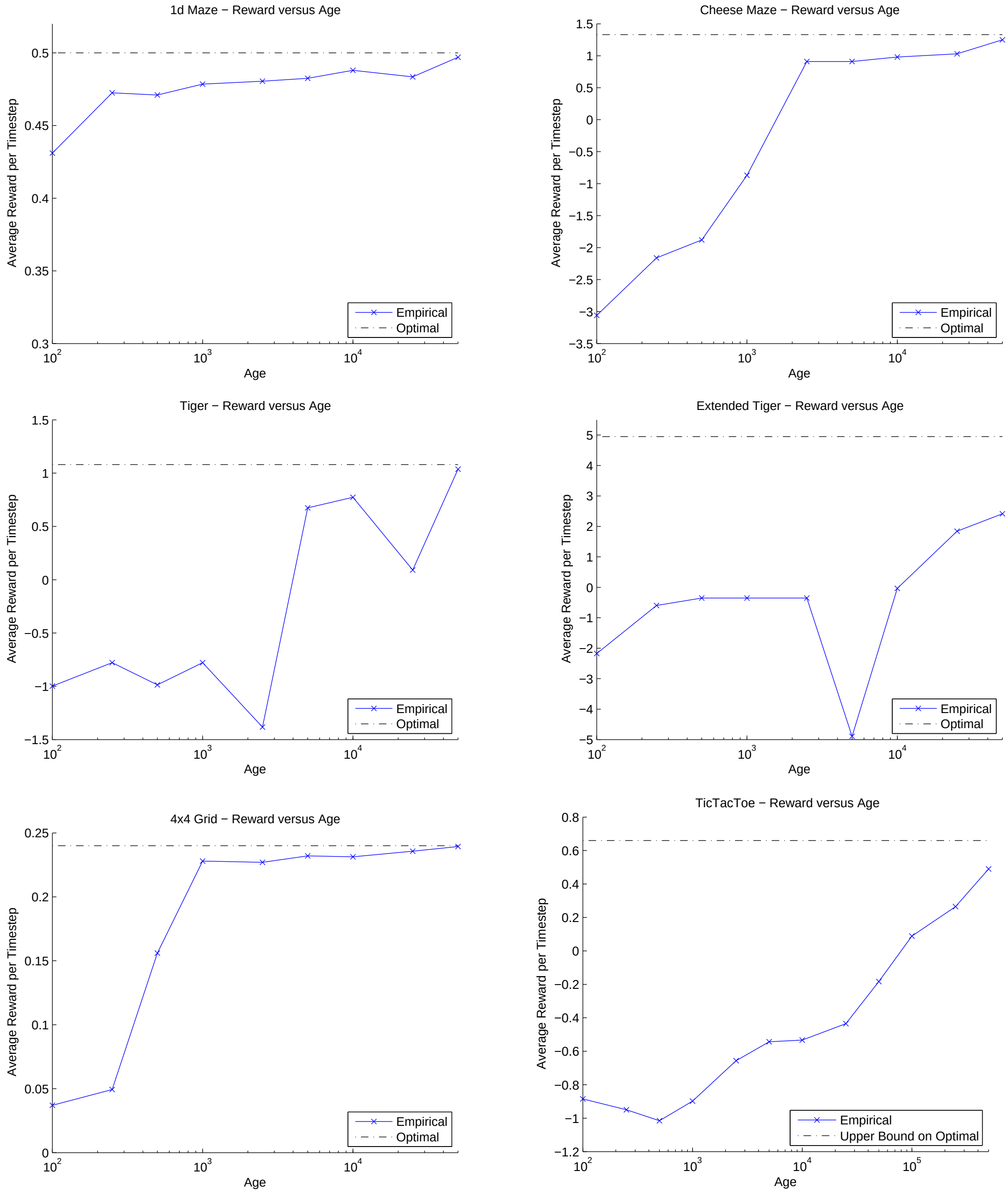

Figure 7: Average reward vs age (measured in number of cycles). Two seconds of search were used for each action.

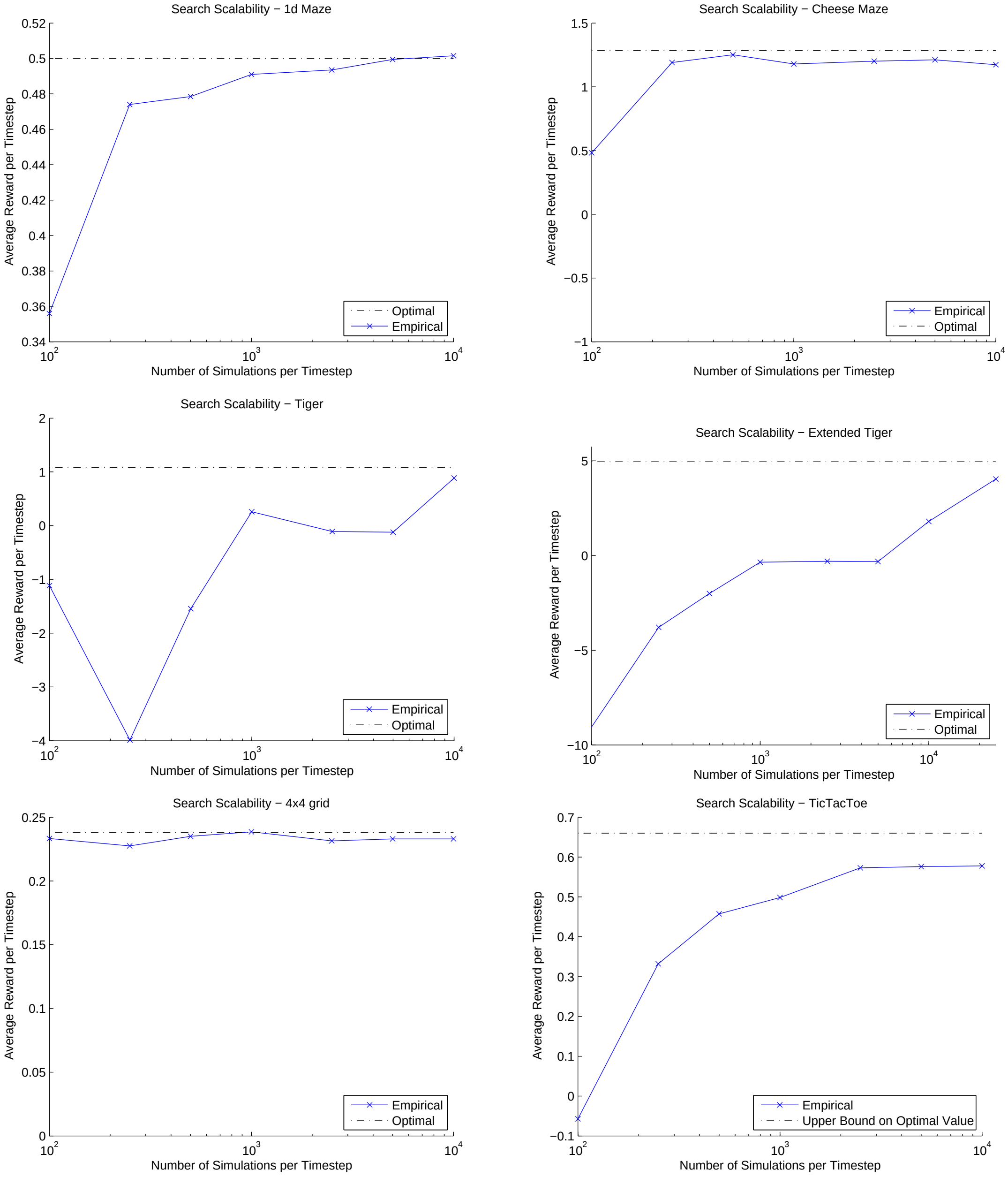

Figure 8: Average reward vs search effort (measured in terms of the number of simulations used for picking each action).

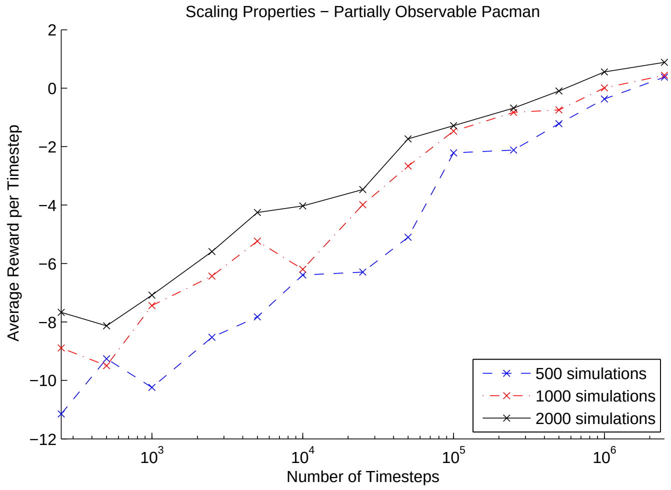

Figure 9: Scaling properties on Partially Observable Pacman

Only then does agent to understand that it is worth listening initially, then standing up, and then finally choosing the correct door according to the information it gathered whilst sitting down. With less simulations, the agent avoids picking a door. Interestingly, the performance of the agent drops after it has interacted with the world 5000 times, yet then sharply increases. At 5000 steps, the agent has overcome its aversion towards picking a door, without fully understanding the environment dynamics. This causes the agent to sometimes pick the wrong door. Further interaction refines the environment model and subsequently allows the agent to improve its performance.

Performance on a challenging domain. Above we introduced the partially observable Pacman domain. In contrast to our other domains, this is an enormous problem. Even if the underlying state space were known, the learning and planning problems would still be hard because there are more than 2^{50} states.

Figure 9 shows the scaling properties of our agent. Again, random exploration was used to build the model for computational reasons. The average reward at each data point was gathered by running the agent for 4000 timesteps, with each action being determined by Predictive UCT.

Visually, the performance of the agent was non-optimal. However, after 2.5 million cycles of interaction, the agent had managed to learn a number of important concepts. It knows not to run into walls. It knows how to seek out food from the limited information provided by its sensors. It knows how to run away and avoid chasing ghosts. The main subtlety that it hasn't learnt (after 2.5 million timesteps) is to aggressively chase down ghosts when it has eaten a red power pill. Also, its behaviour can sometimes become temporarily erratic when stuck in a long corridor with no nearby food or visible ghosts. Still, the ability to perform reasonably in a large domain, and exhibit consistent increases in performance with additional resources (experience or search time) makes us optimistic about the long-term potential of our work.

Heuristic playout function. An important parameter in Predictive UCT is the choice of the playout function. In MCTS-based methods for playing Computer Go, it is well known that adding knowledge to the playout function can dramatically improve performance [GWMT06]. One of the benefits of MCTS methods is that if the domain is known, the playout function presents a natural way to incorporate domain knowledge. In the general agent setting, it would be desirable to automatically gain some of the benefits of expert design through online learning.

If the domain is unknown, a natural baseline playout policy is one that selects between each action uniformly at random. Although this playout policy is obviously quite poor, it does make some heuristic sense: the playouts end up guiding the search toward areas that give off larger rewards without requiring a carefully planned action sequence. In Section 3, we described an intuitive method to incrementally learn a playout policy by attempting to model the real-world actions chosen by Predictive UCT. The aim of this section is to show that our heuristic approach, using a CTW-based action predictor as a playout function, can give significant improvements to Predictive UCT over the naive, uniformly random policy.

Figure 10 shows the impact of using the learned playout function on the cheese maze. (The other domains we tested exhibit similar behaviour.) Two versions of the same agent were run for $120'000$ cycles. Actions were selected using an ϵ-greedy policy: i.e. with probability ϵ the agent moved randomly, otherwise the best action according to Predictive UCT was chosen. The initial ϵ of 0.9 was decayed by multiplying by 0.999 at each time-step. A small (100 or 500) Predictive UCT simulations were used to decide on each action, to maximise the impact of the playout policy on the overall agent performance. The agent that used the self-improving playout policy learned faster and obtained a higher maximum average reward than the agent using uniform random playouts. Although the difference in average reward is small numerically, there is a qualitative difference in the performance of the agent. For example, the uniform playout policy when using 100 simulations averages approximately -1 per timestep. This is equivalent to a policy that simply runs around the maze, never finding the cheese, without ever bumping into a wall. When using 100 learned playouts however, the average reward ends up greater than zero. To achieve this, the agent must be finding the cheese, on average, in less than 11 steps every instance.

Our results demonstrate that it is both reasonable and practical for a MCTS-based general reinforcement learning agent to attempt to learn a playout function online. Our results are by no means exhaustive. The ideal action predictor may not resemble the observation/reward predictor, or it may be designed with different speed/accuracy trade-offs in mind. Online learning of playout functions for MCTS-based agents is a promising direction for future research. Building on this idea, one could also look at ways to modify the UCB policy used in Predictive UCT to automatically take advantage of learnt playout knowledge, similar to the heuristic techniques used in Computer Go [GS07].

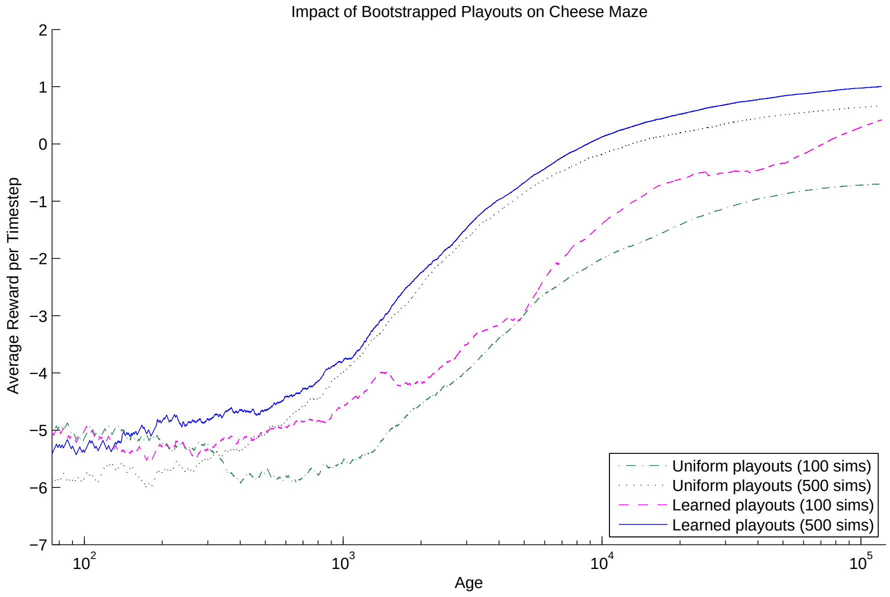

Figure 10: Impact of learned playout function on performance

Computational considerations. If an agent has interacted with the world for t cycles, using a context tree with depth D, there is at most $O(tD\log(|\mathcal{O}||\mathcal{R}|))$ nodes in the context tree. In practice, unless the environment is very noisy, only a subset of the 2^D possible contexts will be created. In our experiments, no more than a gigabyte of memory was required to store the entire environment model. The time complexity of CTW is also impressive: $O(D)$ to generate a single bit, and $O(Dm\log(|\mathcal{O}||\mathcal{R}|))$ to generate the m observation/reward pairs needed to perform a single Predictive UCT simulation.

Predicate CTW. This section gives an example of how Predicate CTW can be used to incorporate domain knowledge that drastically simplifies the agent's learning task. We saw earlier in Figure 7 that the dynamics of TicTacToe required a large amount of training examples for CTW to correctly predict the environment dynamics. Essentially, the main difficulty for the first hundred thousand steps was avoiding making illegal moves. In this experiment, the set of predicates that define CTW was augmented with a predicate that indicated whether the last move by the agent was legal. As one would expect, the agent using this augmented predicate set quickly learnt to play according to the game rules. Figure 11 shows how a small but carefully chosen piece of domain knowledge can have a significant impact on the agent's performance.

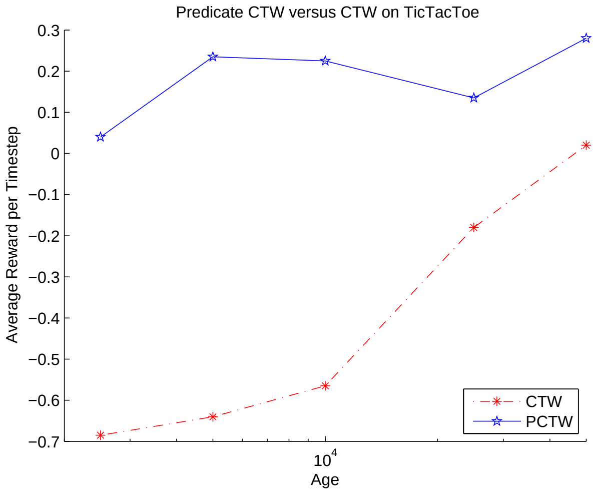

Figure 11: Impact of domain knowledge, using 1000 Predictive UCT simulations.

8 Discussion

We discuss some related and future work in this section. The headings reflect the general area of the literature in which those work can be found.

Algorithmic Information Theory. There have been several attempts at studying the computational properties of AIXI. In [Hut02], an asymptotically optimal algorithm is proposed that, in parallel, picks and runs the fastest program from an enumeration of provably correct programs for any given well-defined problem. A similar construction that runs all programs of length less than l and time less than t per cycle and picks the best output (in the sense of maximizing a provable lower bound for the true value) results in the optimal time bounded AIXItl agent [Hut05, Chp.7]. Like Levin search [Lev73], such algorithms are not practical in general but can in some cases be applied successfully; see e.g. [Sch97, SZW97, Sch03, Sch04].

In tiny domains, universal learning is computationally feasible with brute-force search. In [PH06] the behaviour of AIXI is compared with a universal predicting-with-expert-advice algorithm [PH05] in repeated 2×2 matrix games and shows they exhibit different behaviour.

A Monte Carlo algorithm is proposed in [Pan08] that samples programs according to their algorithmic probability as a way of approximating Solomonoff's universal prior. A closely related algorithm is that of speed prior sampling [Sch02]. It remains an open question whether algorithms that sample from the space of general Turing machines can be made to work in practical problems.

General Reinforcement Learning. We move on next to a discussion of related work in the general RL literature. An early and influential work is the Utile Suffix Memory

(USM) algorithm by McCallum [McC96]. USM uses a suffix tree to partition the agent's history space into distinct states, one for each leaf in the suffix tree. Associated with each state/leaf is a Q-value, which is updated incrementally from experience like in Q-learning [WD92]. The history-partitioning suffix tree is grown in an incremental fashion, starting from a single leaf node in the beginning. A leaf in the suffix tree is split when the history sequences that fall into the leaf are shown to exhibit statistically different Q-values. The USM algorithm works well for a number of tasks but could not deal effectively with noisy environments. Several extensions of USM to deal with noisy environments are investigated in [SB04, Sha07]. USM and their extensions are usually well-motivated but lack formal performance guarantees.

The work closest to ours in the general RL literature is the BLHT algorithm described in [SHL97, SH99]. As in the present work, Suematsu et al. use prediction suffix trees as the model class but their suffix trees are defined at the symbol level (like in USM) as opposed to the bit level at which we operate. Another difference is that BLHT uses the maximum a posteriori (MAP) model to predict the future at any one time whereas we use a mixture of models. Having said that, the actual data structure and algorithm used in [SHL97, SH99] to efficiently compute the MAP model bears close resemblance to CTW, and their algorithm may indeed be a general form of the context tree maximising algorithm [VW95]. In their experiments, Suematsu et al. chose to use a uniform prior over the tree models even though their algorithm would work with an Ockham prior like that given in Equation (20). It is also worth noting that our use of a Bayesian mixture admits a much stronger convergence result compared to what can be proved for BLHT. For control, BLHT uses an (unspecified) dynamic programming based algorithm.

The active LZ algorithm [FMRW09] is also similar in spirit to our work. It combines a Lempel-Ziv [ZL77] based prediction scheme with dynamic programming for control to produce an agent that is provably optimal if the environment is n-Markov, for some arbitrary n. They introduced and evaluated the performance of their agent on the (n-Markov) biased Rock-Paper-Scissor domain. We ran our agent on the same domain, using action-conditional CTW, 10000 Predictive UCT simulations and a uniform playout policy. Figure 12 shows our results overlayed with their reported results. Though it is difficult to compare implementations, it is clear that our agent has reached optimal performance using vastly less (at least two orders of magnitude) experience.

Predictive state representations (PSRs) [LSS02, SJR04] maintain predictions of future experience. Formally, a PSR is a probability distribution over the agents future experience, given its past experience. A subset of these predictions, the core tests, provide a sufficient statistic for all future experience. PSRs provide a Markov state representation, can represent and track the agents state in partially observable environments, and provide a complete model of the worlds dynamics. Unfortunately, exact representations of state are impractical in large domains, and some form of approximation is typically required. There is considerable interest in PSRs but there are at present still no satisfactory learning and discovery algorithms for PSRs.

Temporal-difference networks [ST04] are a form of predictive state representation in which the agent's state is approximated by abstract predictions. These can be predic-

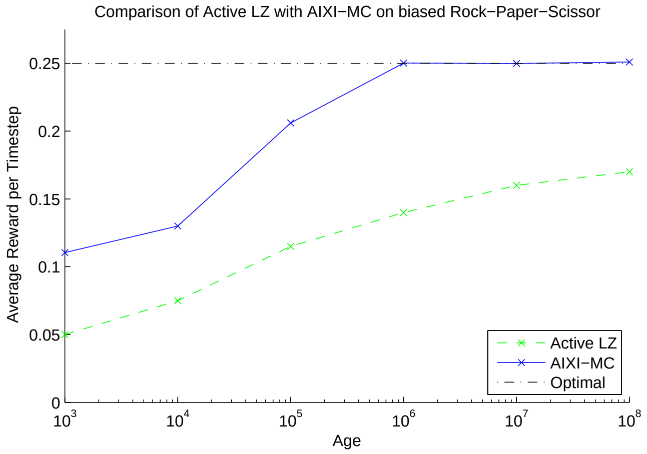

Figure 12: Comparison between AIXI-MC (using action-conditional CTW, 10k Predictive UCT simulations and uniform playouts) and the Active-LZ algorithm.

tions about future observations, but also predictions about future predictions. This set of interconnected predictions is known as the *question network*. Temporal-difference networks learn an approximate model of the worlds dynamics: given the current predictions, the agents action, and an observation vector, they provide new predictions for the next time-step. The parameters of the model, known as the *answer network*, are updated after each time-step by temporal-difference learning. Some promising recent results applying TD-Networks for prediction (but not control) to small POMDPs have been reported in [Mak09].

Model Learning and CTW. Bayesian model averaging is a well-studied technique in statistics and machine learning [HMRV99, Bun92, OH95, CGM98]. There is a nice connection between CTW, Buntine's tree-smoothing algorithm [Bun92], Winnow-style online learning [Lit88, LW94], and boosting [FS97]. The key idea behind Lemma 1 appears in [Bun92, Lemma 6.5.1]. The same technique is used in [HS97] to implement an efficient version of the $P(\beta)$ online learning algorithm [CBFH$^+$93] as a way of avoiding the problematic post-pruning step in decision-tree induction [BFOS84]. [PS99] then builds on that work to implement an efficient version of the Hedge algorithm [FS97] for constructing mixtures of the larger class of edge-based (as opposed to node-based) prunings of a tree. The algorithm in [PS99] can be used in conjunction with the predicate CTW idea to enlarge our agent's model class.

There are several noteworthy ways the basic CTW algorithm can be extended. The finite depth limit on the context tree can be removed [Wil94] without increasing the asymptotic space overhead of the algorithm. We chose to avoid this extension however due to the asymptotic time complexity increase of generating a symbol from linear in the context depth to linear in the number of observed symbols. CTW has also been extended to general non-binary alphabets, and the state-of-the-art seems to be the DE-CTW algorithm [BEYY04, BEY06]. We opted not to use DE-CTW for several reasons. Firstly, DE-CTW is not a strictly online algorithm: a preprocessing phase is required to compute a way of decomposing the alphabets. Secondly, what is computed by DE-CTW isn't really a Bayesian mixture and this is an unnecessary deviation from the theory of AIXI. Lastly, most of the effects of decomposing alphabets can in fact be realised using the predicate CTW extension.

Future work. Our experimental results have been restricted to problems of modest size. Future work will attempt to apply the algorithms presented here to more challenging domains.

The biggest limitation of our current agent is the restricted model class. Prediction suffix trees are simplistic models, inadequate to compactly represent something as simple as the rules of TicTacToe. Furthermore, the strong emphasis placed by CTW on temporally recent symbols is appropriate for only a subset of interesting real-world problems. The aim of the Predicate CTW extension is to relax this restriction somewhat, yet keep the desirable computational properties of CTW. As these predicates are arbitrary boolean functions on the agent's history, they have the power to represent more complicated pieces of information that are useful to an agent in terms of making sensible predictions. Domain knowledge can be encoded in the form of user-supplied predicates, which seems essential for our agent to have any realistic chance of scaling to problems with real-world visual or audio data. Given a large model class $\mathcal{P}$, the main learning problem in predicate CTW is in the identification of a small subset $\mathcal{P}'$ of $\mathcal{P}$ that is relevant to the current environment. This is a major unsolved problem in our setup and we think a suitable application of the Minimum Message Length principle [Wal05] along the lines of [Hut09b] would shed much light on the key issues.

Furthermore, the performance of our agent is dependent on the amount of thinking time allowed at each time step. A crucial property of Predictive UCT is that it is naturally parallel. A prototype parallel implementation of Predictive UCT has been completed, with promising scaling results using between 4 and 8 processing cores. We are confident that further improvements to our prototype implementation will allow us to solve problems where the amount of search, rather than the agent's predictive power, is the main performance bottleneck. Continuing advances in computer hardware will no doubt help address this issue as well.

9 Conclusion

The main contribution of the paper is the extension and synthesis of two key results from online MDP planning (UCT) and information theory/machine learning (CTW) in the design of an agent that directly and scalably approximates the AIXI ideal. This is an important result. Although well established theoretically, it has previously been unclear whether AIXI could motivate the design of practical, yet theoretically well-founded algorithms. Our work answers this question strongly in the affirmative: empirically, our AIXI approximation achieves state-of-the-art performance and theoretically, we can provide some characterisation of the type of environments we expect our agent to handle.

To develop this approximation, we introduced two key algorithms:

- Predictive UCT- a histories-as-states expectimax approximation algorithm;
- action-conditional CTW - an agent-specific generalisation of the CTW algorithm.

Furthermore, we demonstrated that our approach opens a number of future research areas:

- incorporating background knowledge through the predicate CTW extension;
- the possibility of constructing self-improving heuristic playout policies.

Although we are a long way away from being able to construct a truly powerful general agent, the future looks promising. We hope this work generates further interest from the broader artificial intelligence community in both AIXI and general reinforcement learning agents.

Acknowledgements. The authors thank Alan Blair, Bernhard Hengst, John Lloyd, Hassan Mahmud, Malcolm Ryan, Scott Sanner, and William Uther for many helpful discussions. NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

[Aue02] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2002.

[BD98] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[BEY06] Ron Begleiter and Ran El-Yaniv. Superior guarantees for sequential prediction and lossless compression via alphabet decomposition. *Journal of Machine Learning Research*, 7:379–411, 2006.

[BEYY04] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

[BFOS84] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Chapman & Hall, 1984.

[Bun92] Wray L. Buntine. *A Theory of Learning Classification Rules*. PhD thesis, University of Technology Sydney, 1992.

[CBFH$^+$93] Nicolò Cesa-Bianchi, Yoav Freund, David P. Helmbold, David Haussler, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. In *Proc. 25th Annual ACM Symposium on the Theory of Computing*, pages 382–391, 1993.

[CGM98] Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. Bayesian CART model search. *Journal of the American Statistical Association*, 93:935–960, 1998.

[Chr92] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 183–188, 1992.

[CKL94] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *AAAI*, pages 1023–1028, 1994.

[CWH08] Guillaume M. Chaslot, Mark H. Winands, and H. Jaap Herik. Parallel monte-carlo tree search. In *Proceedings of the 6th International Conference on Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer-Verlag.

[CWU$^+$08] G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3), 2008.

[FB08] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[FMRW09] Vivek F. Farias, Ciamac C. Moallemi, Benjamin Van Roy, and Tsachy Weissman. Universal reinforcement learning. *IEEE Transactions on Information Theory*, 2009. To appear.

[FS97] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[GS07] S. Gelly and D. Silver. Combining online and offline learning in UCT. In *Proceedings of the 17th International Conference on Machine Learning*, pages 273–280, 2007.

[GW06] Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS Workshop on On-line trading of Exploration and Exploitation*, 2006.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[HJ06] Michael P. Holmes and Charles Lee Isbell Jr. Looping suffix tree-based inference of partially observable hidden state. In *ICML*, pages 409–416, 2006.

[HMRV99] Jennifer A. Hoeting, David Madigan, Adrian Raftery, and Chris T. Volinsky. Bayesian model averaging: A tutorial. *Statistical Science*, 14(4):382–417, 1999.

[HS97] David P. Helmbold and Robert E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Machine Learning*, 27(1):51–68, 1997.

[Hut02] Marcus Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science.*, 13(3):431–443, 2002.

[Hut05] Marcus Hutter. *Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability*. Springer, 2005.

[Hut09a] Marcus Hutter. Feature dynamic Bayesian networks. In *AGI*, pages 67–73, 2009.

[Hut09b] Marcus Hutter. Feature Markov decision processes. In *AGI*, pages 61–66, 2009.

[KLC95] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1995.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.

[KT81] R.E. Krichevsky and V.K. Trofimov. The performance of universal coding. *IEEE Transactions on Information Theory*, IT-27:199–207, 1981.

[KW01] Stefan Kramer and Gerhard Widmer. Inducing classification and regression trees in first order logic. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 6. Springer, 2001.

[Leg08] Shane Legg. *Machine Super Intelligence*. PhD thesis, Department of Informatics, University of Lugano, 2008.

[Lev73] Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9:265–266, 1973.

[LH07] Shane Legg and Marcus Hutter. Universal intelligence: A definition of machine intelligence. *Minds and Machines*, 17(4):391–444, 2007.

[Lit88] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[Llo03] John W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Springer, 2003.

[LN07] John W. Lloyd and Kee Siong Ng. Learning modal theories. In *Proceedings of the 16th International Conference on Inductive Logic Programming*, LNAI 4455, pages 320–334, 2007.

[LSS02] Michael Littman, Richard Sutton, and Satinder Singh. Predictive representations of state. In *NIPS*, pages 1555–1561, 2002.

[LV08] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, third edition, 2008.

[LW94] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–261, 1994.

[Mak09] Takaki Makino. Proto-predictive representation of states with simple recurrent temporal-difference networks. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 697–704, New York, NY, USA, 2009. ACM.

[McC96] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, 1996.

[Ng05] Kee Siong Ng. *Learning Comprehensible Theories from Structured Data*. PhD thesis, The Australian National University, 2005.

[OH95] Jonathan J. Oliver and David J. Hand. On pruning and averaging decision trees. In *ICML*, pages 231–241, 1995.

[Pan08] Sergey Pankov. A computational approximation to the AIXI model. In *AGI*, pages 256–267, 2008.

[PH05] Jan Poland and Marcus Hutter. Defensive universal learning with experts. In *Proc. 16th International Conf. on Algorithmic Learning Theory*, volume LNAI 3734, pages 356–370. Springer, 2005.

[PH06] Jan Poland and Marcus Hutter. Universal learning of repeated matrix games. Technical Report 18-05, IDSIA, 2006.

[PS99] Fernando C. Pereira and Yoram Singer. An efficient extension to mixture techniques for prediction and decision trees. *Machine Learning*, 36(3):183–199, 1999.

[RPPCD08] Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-Draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32:663–704, 2008.

[RST96] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2):117–150, 1996.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[SB04] Guy Shani and Ronen Brafman. Resolving perceptual aliasing in the presence of noisy sensors. In *NIPS*, 2004.

[Sch97] Jürgen Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873, 1997.

[Sch02] Jürgen Schmidhuber. The speed prior: A new simplicity measure yielding near-optimal computable predictions. In *Proc. 15th Annual Conf. on Computational Learning Theory*, pages 216–228, 2002.

[Sch03] Jürgen Schmidhuber. Bias-optimal incremental problem solving. In *Advances in Neural Information Processing Systems 15*, pages 1571–1578. MIT Press, 2003.

[Sch04] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004.

[SH99] Nobuo Suematsu and Akira Hayashi. A reinforcement learning algorithm in partially observable environments using short-term memory. In *NIPS*, pages 1059–1065, 1999.

[Sha07] Guy Shani. *Learning and Solving Partially Observable Markov Decision Processes*. PhD thesis, Ben-Gurion University of the Negev, 2007.

[SHL97] Nobuo Suematsu, Akira Hayashi, and Shigang Li. A Bayesian approach to model learning in non-Markovian environment. In *ICML*, pages 349–357, 1997.

[SJR04] Satinder Singh, Michael James, and Matthew Rudary. Predictive state representations: A new theory for modeling dynamical systems. In *UAI*, pages 512–519, 2004.

[Sol64] Ray J. Solomonoff. A formal theory of inductive inference: Parts 1 and 2. *Information and Control*, 7:1–22 and 224–254, 1964.

[ST04] Richard S. Sutton and Brian Tanner. Temporal-difference networks. In *NIPS*, 2004.

[SZW97] J. Schmidhuber, J. Zhao, and M. A. Wiering. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130, 1997.

[VW95] Paul A.J. Volf and Frans M.J. Willems. A study of the context tree maximizing method. In *16th Symposium on Information Theory in the Benelux*, pages 3–9, 1995.

[Wal05] Christopher S. Wallace. *Statistical and Inductive Inference by Minimum Message Length*. Springer, 2005.

[WD92] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[Wil94] Frans M. J. Willems. The context-tree weighting method: Extensions. *IEEE Transactions on Information Theory*, 44:792–798, 1994.

[WST95] Frans M.J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41:653–664, 1995.

[WST97] Frans Willems, Yuri Shtarkov, and Tjalling Tjalkens. Reflections on “The Context Tree Weighting Method: Basic properties”. *Newsletter of the IEEE Information Theory Society*, 1997.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.