Monte Carlo Hierarchical Model Learning

Jacob Menashe and Peter Stone
The University of Texas at Austin
Austin, Texas
{jmenashe,pstone}@cs.utexas.edu

ABSTRACT

Reinforcement learning (RL) is a well-established paradigm for enabling autonomous agents to learn from experience. To enable RL to scale to any but the smallest domains, it is necessary to make use of abstraction and generalization of the state-action space, for example with a factored representation. However, to make effective use of such a representation, it is necessary to determine which state variables are relevant in which situations. In this work, we introduce T-UCT, a novel model-based RL approach for learning and exploiting the dynamics of structured hierarchical environments. When learning the dynamics while acting, a partial or inaccurate model may do more harm than good. T-UCT uses graph-based planning and Monte Carlo simulations to exploit models that may be incomplete or inaccurate, allowing it to both maximize cumulative rewards and ignore trajectories that are unlikely to succeed. T-UCT incorporates new experiences in the form of more accurate plans that span a greater area of the state space. T-UCT is fully implemented and compared empirically against B-VISA, the best known prior approach to the same problem. We show that T-UCT learns hierarchical models with fewer samples than B-VISA and that this effect is magnified at deeper levels of hierarchical complexity.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Intelligent agents

General Terms

Algorithms

Keywords

Single and multi-agent learning techniques; Reinforcement Learning; Factored Domains; Model Learning; Hierarchical Skill Learning; Monte Carlo Methods

1. INTRODUCTION

Suppose you are tasked with driving to a new supermarket downtown. At short notice you might be able to come up with some simple instructions, such as "head south for approximately 3 miles." Before actually making the trip you could consult a map and look at a couple of possible routes, then settle on the route that seems the best given the distance, the time of day, etc. Finally you try out your selected route and use this new experience to help you plan better in the future.

In further detail, the process of planning your actions is divided into distinct phases. The first phase is target selection, in which you decide on the supermarket as your destination. The next is a rough planning phase, in which you select a high-level action sequence to consider: "head south for 3 miles." For the third phase you then simulate the experience of navigating to your target by looking at a map and planning out the specific roads you'll be taking. Finally you execute an action sequence by following your planned route to the new supermarket.

In this work we introduce an implementation of this approach to model-based planning, namely *Transition-based Upper Confidence Bounds for Trees*, or T-UCT. We draw from the widely successful UCT algorithm [6] by extending it for use with action sequences rather than primitive actions. This extension allows us to make long-term, compound planning decisions that respect both the intermediate reward and transition dynamics of a given environment. T-UCT selects targets to explore novel areas of the state space, performs randomized depth-first graph search for rough planning, and then uses UCT to carry out Monte Carlo simulations. Finally, T-UCT executes the best plan derived from this process to explore the environment.

In Section 2 we begin with some background on our models and describe B-VISA, a successful earlier approach to the problem of learning models in hierarchical domains. In Section 3 we describe the T-UCT algorithm. In Section 4 we apply T-UCT to the task of learning model structures while simultaneously using these structures for exploration. Finally we conclude in Section 5.

2. BACKGROUND

This section presents in detail the background necessary to understand our novel T-UCT algorithm. Though all the concepts summarized here exist in prior work, a full understanding of the specific assumptions and notations is necessary for understanding the remainder of the paper. Specifically, in Section 2.1 we introduce Markov Decision Processes and their Factored variants, followed by a discussion of compact models for these domains in Section 2.2. In Section 2.3 we discuss the means of analyzing these models to predict the dynamics of factored domains. In Section 2.4 we exam-

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May 4–8, 2015, Istanbul, Turkey.*
Copyright © 2015, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

ine learning domain models from experience, and finally in Section 2.5 we discuss Bootstrapping with VISA, the best known approach for the problem of learning and exploiting hierarchical domains.

2.1 (Factored) Markov Decision Processes

A *Markov Decision Process (MDP)* is a task defined by a tuple $\langle S, A, P, R\rangle$ of states S, actions A, a transition function P and a reward function R. Here we define P and R such that $P(s, a, s') = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$ is the probability that action a in state s at time t will result in a transition to state s' at time $t+1$, and $R(s, a, s') \in \mathbb{R}$ is the reward for transitioning from s to s' under action a. An MDP is assumed to satisfy the Markov property which guarantees that the distribution of the successor state $\Pr\{s_{t+1}\}$ depends only on s_t and a_t. To solve an MDP means to find an optimal policy $\pi : S \to A$ that maximizes expected cumulative discounted reward. When the functions P and R of an MDP are unknown, it is possible to learn an optimal policy through experimentation with Reinforcment Learning (RL) techniques.

RL research focuses on the question of how to quickly and accurately compute a policy that solves an MDP. A simple RL approach to this problem is to construct an agent which learns a function $Q : S \times A \to \mathbb{R}$ that assigns values to each state-action pair (s, a) in the given MDP. Specifically, the Q function estimates the cumulative future discounted reward expected when executing action a in state s and acting optimally thereafter. Once an accurate value function is learned the optimal policy can be computed with dynamic programming. As the state-action space increases in size, however, so does the complexity of Q. If we are to efficiently navigate large domains it is therefore necessary to apply further assumptions to the MDP framework. This can be accomplished by factoring the state space to create a *Factored MDP* (FMDP). A Factored MDP is an MDP whose state space can be represented as the cross product of orthogonal state variables: $S = S_1 \times S_2 \times \cdots S_n$. Thus an FMDP is defined by a tuple $\langle \{S_i\}_{i=1}^n, A, P, R\rangle$.

2.2 DBNs and CPTs

An FMDP can be modeled compactly as a set of Dynamic Bayesian Networks (DBNs) with a single DBN corresponding to each action in the FMDP [2]. Here a DBN is specified as a two-layer directed bipartite graph with nodes corresponding to state variables and edges corresponding to dependence under the specified action. An example is given in Figure 1. The edges always move forward in time, indicating that the state s_{t+1} depends only on s_t. Since we are analyzing depdencies between variables under different actions, the dependencies must move forward in time. This forward-directed dependence follows from our assumption of a Markov environment. In the figure we see that all state variables depend on themselves, and additionally that S_i depends on both on itself and S_j.

Each DBN can be encoded as a set of Conditional Probability Trees (CPTs), one for each variable $S_i \in S$. Figure 1 gives an example of such a CPT. Each internal node in the tree is a *refinement* on a particular variable, indicating that the expected effects of an action depend on that variable. For example, the node S_j in the CPT indicates that the effects of a_2 on S_i depend on S_j when $S_i = 1$. Each leaf in the CPT is predicated on a single assignment of state variables

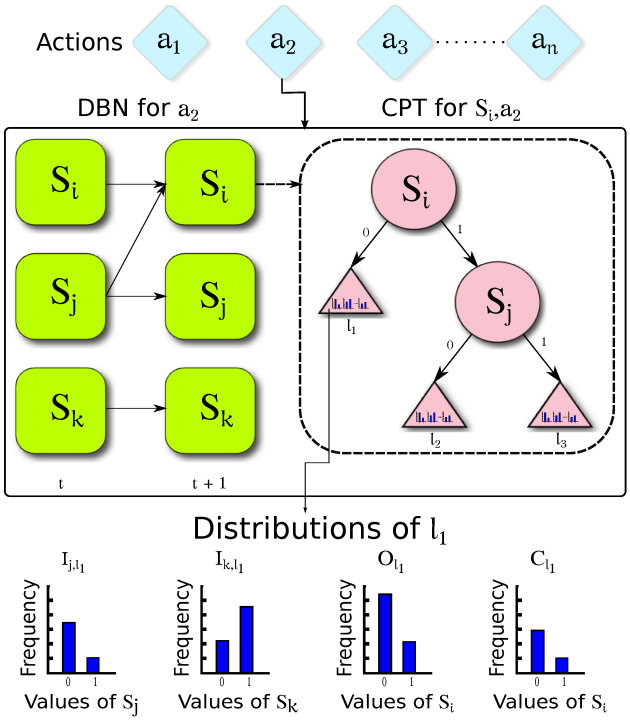

Figure 1: *Top:* All actions available in the environment. *Left:* A DBN representing the effects of action a_2 on state variables S_i, S_j, S_k. *Right:* The CPT encoding transition dynamics for state variable S_i under action a_2, with leaves labeled with distributions. *Bottom:* The distributions of values that are tracked at leaf l_1 (see Section 2.3).

to values which can be recovered by traversing the path from the leaf to the root. The *context* of a leaf l is the assignment of values (edges) to state variables (nodes) on the path from the leaf to the root. For instance, the context of l_2 is $\{S_i = 1, S_j = 0\}$.

2.3 Model Dynamics

Each CPT is used to store (s_t, a, S_i^{t+1}) tuples[1]. When the agent takes action a in state s_t and observes the resulting state s_{t+1}, we first look up the set of CPTs representing the DBN for action a. In each CPT for state variable S_i and action a we find the leaf whose context matches s_t and then add (s_t, a, S_i^{t+1}) to the leaf. Finally we analyze the dataset of each leaf l to produce distributions O_l, C_l, $\{I_{j,l}\}_{j \le n}$ for j such that S_j is not in the context of l. Each $I_{j,l}$ represents an **input distribution**: values of state variable S_j in state s_t for leaf l. We ignore the variables already in the context of l since their value assignments are fixed for all data stored in l. O_l is the **output distribution**: values of S_i in state s_{t+1}. Finally, C_l is the **change distribution**: values of S_i in state s_{t+1} that are different from those of s_t. The purpose of the input distributions is to track the states from which

[1]Prior work uses the tuple (s_t, a, s_{t+1}), however the purpose of the latter element is to model the distribution of a single state variable for each CPT. We therefore restrict the result state s_{t+1} to the value of the state variable that is being modeled for the given CPT.

all data samples have been gathered, allowing the agent to seek novel states for exploration. The purpose of the output distribution and change distribution is to model the effects that the CPT's action has on the CPT's state variable.

We now provide a simplified example of what these distributions might look like using the CPT from Figure 1. We use the notation $\{S_i = v_1 : x, S_i = v_2 : y\}$ to represent a histogram of values encountered for S_i where x instances of value v_1 and y instances of value v_2 have been encountered. Suppose we execute action a_2 for the first time in the state $s_t = \{S_i = 0, S_j = 1, S_k = 0\}$ and observe the successor state $s_{t+1} = \{S_i = 1, S_j = 1, S_k = 1\}$. We then add the tuple $(s_t, a_2, \{S_i = 1\})$ to our CPT at the leaf whose context matches s_t. The first refinement in the CPT for S_i is on the state variable S_i. Since $S_i = 0$ in state s_t, we move to the leaf with $S_i = 0$, namely l_1. We then update each distribution modeled by l_1. Beginning with the input distributions, we set $I_{j,l_1} = \{S_j = 0 : 0, S_j = 1 : 1\}$, $I_{k,l_1} = \{S_k = 0 : 1, S_k = 1 : 0\}$. We then update the output and change distributions: $O_{l_1} = \{S_i = 0 : 0, S_i = 1 : 1\}$ and $C_{l_1} = \{S_i = 0 : 0, S_i = 1 : 1\}$. Now suppose we arrive in state $s_{t+k} = s_t$ and execute a_2 again, except this time the action has no effect. Then our distributions become $I_{j,l_1} = \{S_j = 0 : 0, S_j = 1 : 2\}$, $I_{k,l_1} = \{S_k = 0 : 2, S_k = 1 : 0\}, O_{l_1} = \{S_i = 0 : 1, S_i = 1 : 1\}$ and $C_{l_1} = \{S_i = 0 : 0, S_i = 1 : 1\}$.

These distributions allow us to query our model to answer four main questions:

1. How can we quickly learn the dynamics of our domain?
2. What changes might occur after executing a?
3. What state s_{t+1} is expected after executing a in s_t?
4. How can we plan a path from state s_t to state s_{t+k}?

Question 1 is answered by computing each leaf's entropy gain as a function of its $\{I_{j,l}\}$. This computation is described in detail in Section II.F of the previous work by Vigorito and Barto [10]. Question 2 is answered by searching each leaf for each CPT in the DBN for a. Each leaf l with a non-empty distribution C_l encodes a change in state for its CPT's associated state variable S_i. Question 3 is answered with O_l: we can predict s_{t+1} under action a by sampling from the distributions of the leaves corresponding to s_t for each CPT in the DBN for a.

To answer Question 4 we introduce transitions. A *transition* on a state variable S_i is a tuple $\langle S_i, c, a, v, p\rangle$ where c is a context, a is an action, v is a value assignment to S_i and p is the probability of success. A transition $t = \langle S_i, c, a, v, p\rangle$ denotes the statement, "When action a is taken in context c the state variable S_i will change to value v with probability p." These are similar to exits, which are introduced by Hengst [3] and used by VISA for identifying options. Transitions give rise to a transition graph and allow us to reduce the problem of navigating a domain with complex dynamics to simple graph planning.

To identify the transitions encoded in our model we start by iterating through all of the leaves in all of the CPTs. If a leaf's action results in a change to the state variable it models in that leaf's context, then we extract a single transition from that leaf for each value that S_i can change to. For example, suppose after executing some action a 20 times in state s we observe the value S_i changed from 0 to 1 with $p = .5$, from 0 to 2 with $p = .3$, and did not change with $p = .2$. If we look at the leaf corresponding to s in the CPT for S_i and a we would find the change distribution $C_l = \{S_i = 0 : 0, S_i = 1 : 10, S_i = 2 : 6\}$ and 20 total recorded datapoints. We could then extract two transitions for S_i with their contexts set to the context c_l of the leaf l. The transitions would be represented as $\langle S_i, c_l, a, 1, .5\rangle$ and $\langle S_i, c_l, a, 2, .3\rangle$. Thus these transitions give us model-based descriptions of the prerequisites and effects for the primitive action a.

2.4 Model Learning

We seek to find a way to learn environment dynamics from experience in deeply hierarchical stochastic domains with specific interest in scalability and sample efficiency. We expect model inaccuracies to be consistently present and that some environment dynamics may be entirely missing from the model. Our ideal algorithm therefore has the following properties:

- **Sample Efficiency** - We wish to learn and exploit model dynamics in as few timesteps as possible.
- **Linear Scaling** - Planning time should scale linearly with the number of state variables and actions and the hierarchical depth of the domain.
- **Partial Modeling** - Partial models should be usable for planning.

Given a DBN/CPT model for a Factored MDP it is possible to compute optimal policies for any particular reward function [1]. However the problem of computing an optimal policy now becomes that of obtaining an accurate model. For small domains, this may be feasible (albeit tedious) to perform by hand. For larger domains or domains with unknown dynamics it may not be possible without some form of automation. Thus model-learning algorithms such as that developed by Jonsson and Barto [5] seek to explore an MDP for the purpose of obtaining representative data samples. These samples are then leveraged to build the DBN/CPT network that compactly models the dynamics of the underlying FMDP. As the stochasticity of a domain increases, the number of samples needed to accurately model its dynamics increases as well.

For complex hierarchical domains it can be difficult to obtain the samples needed for modeling high-level dynamics. If some state variable S_i is only affected by actions taken from within a particular context, this creates a dependency in which that context must be achieved in order to collect data for S_i. One can imagine building multiple levels of these dependencies into a hierarchy. In such circumstances it may be intractable to collect data for the deeper levels of the hierarchy under model-free RL, or with using flat models to learn the environment dynamics.

In this work we explore methods for efficiently learning high-dimensional, deeply-nested hierarchical environments. The general approach can be broken down into two subproblems. First, we need a model that compactly represents environment dynamics in a factored domain and can be refined as new data becomes available. The DBN/CPT model outlined in Section 2.1 achieves compactness through exploiting the separability of FMDP state variables. Jonsson and Barto [5] furthermore provide a technique for iteratively refining DBN/CPT models as data is gathered. By assuming that each state variable is independent of the other variables in the domain under each action, and by only adding CPT refinements when this assumption is proven false, we

can achieve a logspace reduction in model complexity. For instance, a domain with 30 state variables and 25 actions will have $2^{30} \cdot 25 \approx 25$ billion state-action pairs, but we can model such a domain with only $30 \cdot 25 = 750$ CPTs.

The second subproblem is that of data gathering. As discussed by Jonsson and Barto [5], it can be beneficial for an agent to gather data according to entropy measurements on its input distributions. For example, if we've seen 50 samples with $S_i = 0$ and 2 samples with $S_i = 1$, then our distribution $0 : 50, 1 : 2$ has very low entropy. To increase the entropy of this distribution we would need to collect more samples when $S_i = 1$. In this way maximizing entropy gain can push an agent to explore novel data.

In addition to seeking out good data it is necessary to have the means to reach new areas of the state space. In a hierarchical domain this task is non-trivial and may not be possible without some model of the dynamics. To navigate a domain we therefore need a way to plan our trajectory through the state space. Thus the data gathering problem is one of active learning (selecting new areas of the state space) and path planning (navigating a complex environment).

2.5 B-VISA

Previous work by Vigorito and Barto [10] addresses the path planning component using the VISA algorithm [4] and the Options framework [8]. An *option* consists of a policy π, an initiation set I, and a termination condition β. When an agent enters a state $s \in I$ it can use the option by executing its policy π until it reaches some state s' such that $\beta(s') = 1$. Termination is generally assumed to be guaranteed, however in practical settings one may simply limit the number of timesteps that an option may be executed for. Options are used to formalize the notion of distinct skills in RL. As an agent explores the FMDP and models the dynamics, it leverages its model to create options that allow it to move to areas of the state space that might otherwise be difficult to reach. Initially each option only maps states to primitive actions; however as the agent builds a library of options it begins to use these alongside primitives when computing new options. This process of bootstrapping options that are built with VISA creates a skill hierarchy that mirrors the structural hierarchy of the FMDP. In this way an agent is able to reach contexts at higher levels in the hierarchy and collect data at these contexts to further refine its model.

Bootstrapping with VISA (B-VISA) provides an effective way to analyze and make use of the DBN/CPT model. This approach used by Vigorito and Barto [10] suffers from drawbacks, however. The algorithm relies on entropy gain to decide exploration points in the state space, but it does not consider intermediate rewards accumulated when navigating toward an exploration target. This work also employs Structured Value Iteration (SVI) [1] as the central planning algorithm both for computing option policies and for general exploration. SVI uses a set of CPTs along with a reward tree to compute an optimal policy tree - the algorithm essentially applies value iteration to trees. SVI scales superlinearly with the size of the state-action space, and when options are provided to SVI as part of the bootstrapping process, this increases the computation time for option policies and thus computation time increases as model dynamics are learned. It is possible to improve the performance of SVI by heuristically restricting which options and primitives are available for computing a particular policy, however we know of no robust method of filtering options in this way. When SVI is used for bootstrapping with option policies, it also becomes necessary to compute the effect distributions for each of these policies, which is similarly time-consuming. Finally, SVI relies on having complete information on supplied environment dynamics - if an environment has been partially learned, it may not be possible to construct a valid policy even in well-understood areas of the environment.

In this paper we extend the work of Vigorito and Barto [10] by using UCT as the data gathering mechanism used to quickly explore and learn a hierarchical FMDP. As we describe in the next section, our primary contribution is the replacement of the Options framework and SVI's exact planning approach with Monte Carlo planning to consider intermediate and dynamic rewards and to perform effective planning in partially learned hierarchical environments.

3. MODEL LEARNING WITH MONTE CARLO METHODS

Monte Carlo methods offer a promising alternative to exact planning techniques by relying on randomized simulations that can be effective with partial or inaccurate models and large state spaces. Rather than computing a precise policy that accounts for all possible states from a model that is assumed to be perfect, we can instead focus on simulated experiences based on our immediate needs and the incomplete information available to us. UCT is an instantiation of this approach that effectively manages the exploration-exploitation tradeoff in considering both known rewards and novel actions to affect the agent's behavior [6]. UCT involves repeatedly simulating actions from the current state by sampling successive states from some available state-action model. After performing a number of simulations, the action with the greatest expected reward (i.e., the action which performs best in simulation) is taken by the agent. In this section we describe T-UCT, an extension of UCT, which we use to iteratively learn the CPT refinements of hierarchical domains while leveraging these structures for more effective exploration.

3.1 Extending UCT with T-UCT

To apply UCT to hierarchical domains we need to make proper use of the information provided by our models. Simply following a path of discounted entropy gain can lead an agent to execute individual actions that improve entropy gain without moving to hard-to-reach areas of the state space. UCT as described by Kocsis et al. [6] applies only to primitive actions and does not take complex action planning into account. We therefore extend this approach by using transitions in place of primitive actions as the basis for agent trajectory planning. We call this method Transition UCT (T-UCT). T-UCT is similar to the recent H-UCT algorithm developed by Vien and Toussaint [9]. However, while H-UCT requires an action hierarchy to be provided to the agent, T-UCT computes hierarchical plans directly from its learned model. Pseudocode for T-UCT is given in Algorithm 1.

T-UCT proceeds in four distinct phases:

1. A *target selection* phase (line 5), in which the agent uses entropy gain to evaluate and randomly select a number of target contexts from the state space.

Algorithm 1 Transition UCT search algorithm.

```
1: function T-UCTSearch(State s)
2:     br ← −∞, T ← OrderTargets()
3:     Initialize Q(s, a) to 0 for all s, a
4:     repeat
5:         t ← SelectTarget(T)
6:         p ← PlanRoughPath(s, t)
7:         repeat
8:             s′, r, steps ← SimulatePath(s,t,p)
9:         until enough samples for accurate Q
10:        r̄ = r/steps
11:        if r̄ > br then
12:            bp ← p                          //Select new best path
13:            br ← r̄                          //Select new best reward
14:        end if
15:    until NumSuccesses() ≥ M or T = ∅
16:    return bp
17: end function
```

2. A *rough planning* phase (line 6), in which the agent computes a high-level plan for how to navigate from a source to a target.
3. A *simulation* phase (line 8), in which the agent uses its knowledge of the domain to simulate different trajectories from the source to the target. Each iteration improves the accuracy of the Q table and thus the quality of the simulated path with respect to the reward signal.
4. An *execution* phase (Algorithm 5), in which the agent selects a single trajectory based on the performance in the simulations from the previous step.

We discuss these phases in order.

3.2 Target Selection

The agent begins randomly selecting a set of targets to evaluate under its selection criterion, in this case entropy gain. First, the agent iterates through all the leaves in all the CPTs in its model and computes the expected entropy gain that would arise from executing each leaf's action in that leaf's context. This general approach is described by Jonsson and Barto [5]. From information theory we have that the entropy H of a discrete distribution P is

$$H(P) = -\sum_i P(x_i) \log P(x_i)$$

To compute the entropy gain of executing an action in a leaf's context, we take the difference in entropy for all of the leaf's I distributions as a result of executing its associated action. Letting an input distribution I' of l be the distribution I after executing l's action a in context c, and letting $H_l = \sum_{I \in l} H(I)$ and $H'_l = \sum_{I' \in l} H(I')$, the expected entropy gain is $H'_l - H_l$.

The agent then orders these leaves by their entropy gain in decreasing order and randomly selects from this list until it has simulated some pre-determined number M of successful path traversals. Here a path traversal is considered successful if the agent's state matches the selected target context at the end of the simulation; that is, the simulation doesn't encounter any errors due to cycles, dead ends, or excessively long traversals. Before each traversal, the agent selects a target from the top of the list with probability p, from the second position with $p/2$, from the third position with $p/4$, and so on. In our implementation we use $p = .5$. Each item that is skipped over is popped off of the list.

This randomized selection approach allows the agent to select targets with high expected entropy gain often, but the randomization also ensures that alternative targets will be evaluated as well. The benefit of this approach, versus only taking the top M targets, is that targets with low rankings and high intermediate entropy gain (cumulative gain observed while transitioning to a leaf's context) have a chance of being evaluated.

Once a target has been selected we proceed to the next phase of T-UCT: rough path planning.

3.3 Rough Path Planning

The purpose of the rough planning phase is for an agent to answer two questions. First, given a possibly inaccurate model of an environment's transition dynamics, is it possible to traverse from a source state to a target state? And if so, what possible paths can we try out? These questions are answered simultaneously through a series of randomized depth-first searches through a transition graph constructed from the agent's transition model. Algorithm 2 gives the pseudocode for this process.

Algorithm 2 Rough path planning algorithm.

```
1: function PlanRoughPath(SourceContext sc, Target-
   Context tc)
2:     pa ←EmptyPath()
3:     c_d ←MEA(sc, tc)                        //Means Ends Analysis
4:     for all S_i in StateVariables(c_d) do
5:         T_i ←ComputeTransitionGraph(c_d, S_i)
6:         P ← ∅
7:         for 1 to M do                       //M is configurable
8:             p ←RandomDepthFirstSearch(sc, tc, T_i)
9:             Push(P, p)
10:        end for
11:        T_i ←MergePathsIntoGraph(P)
12:        pa[S_i] = T_i
13:    end for
14:    return pa
15: end function
```

Given some starting state and a target state the agent begins by computing a means-ends analysis [7] to produce a difference context c_d in line 3. This context is a tuple of tuples: each entry is a pair of values v_s, v_t for a state variable S_i and denotes the agent's wish to change its state from $S_i = v_s$ to $S_i = v_t$. The agent considers each entry in isolation and constructs a directed graph $\mathcal{T}_i$ of all possible transitions for the state variable S_i in line 5. Vertices in $\mathcal{T}_i$ are defined by the values in the domain of S_i, and edges are defined by transitions taken from a model. For instance, a transition that causes $S_i = 0$ to change to $S_i = 6$ would be represented as an edge from the graph vertex for 0 to the graph vertex for 6. In line 8 the agent performs multiple randomized depth-first searches from v_s to v_t to produce a set of possible paths between these two value assignments. The agent then merges these paths into a subgraph T_i of $\mathcal{T}_i$ in line 11. An example of $\mathcal{T}_i$ and its subgraph T_i can be seen in Figure 2. Finally in line 12 the path is indexed by the state variable it describes.

As an example, consider the transitions t_1, t_2, t_3, t_4 indicated in Figure 2. The agent begins its randomized depth

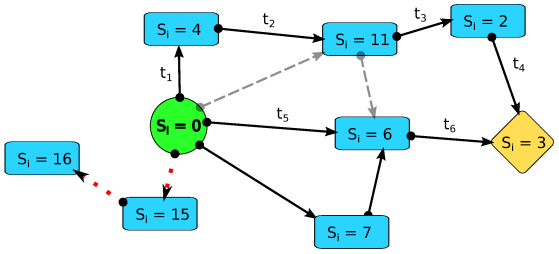

Figure 2: A full transition graph $\mathcal{T}_i$ (all edges) and its randomized subgraph T_i (solid edges), as described in Section 3.3. T_i represents a set of possible paths for S_i from a source value $S_i = 0$ to a target value $S_i = 3$. The dotted lines represent a dead end. The dashed lines represent transitions in $\mathcal{T}_i \setminus T_i$.

first search by looking at all transitions that alter state variable S_i and whose contexts match our source context's value assignment $S_i = 0$. t_1 and t_5 are examples of such transitions. If we randomly select t_1, we "follow" this transition by looking at its modeled effect of setting S_i to the value 4. We then look at all transitions available to us that alter S_i from 4 to some other value. In this case t_2 is the only transition available. We repeat this process of moving to new nodes and randomly selecting from the available transitions until we reach the target value assignment $S_i = 3$. On our first navigation we might therefore find the path t_1, t_2, t_3, t_4. On another round, we might find t_5, t_6. We repeat this process and remember the transitions from paths that successfully move from our source to our target value assignment. In Figure 2 this is represented by all of the solid edges. This collection of transitions defines a subgraph $T_i \subseteq \mathcal{T}_i$.

The advantage of working with this randomized subgraph T_i, versus the full transition graph, is that every node in T_i is on a path from v_s to v_t, and furthermore any choice of directed edge moves us along one of these paths. Thus if we pass this graph to our simulator (see Section 3.4) we can avoid exhaustively searching the state space and hoping to land on our target.

3.4 Simulation

The agent begins with the set of rough paths (i.e. transition graphs) constructed in the previous step, one for each state variable. It is now necessary to simulate traversals on these paths to evaluate their benefit in terms of both feasibility and computed reward. The rough path planning algorithm ignores some important information, such as dependences between state variables, reward functions, and transition probabilities. Simulation allows the agent to evaluate the constructed graphs tractably while taking all of these factors into account.

Algorithm 3 shows the pseudocode for T-UCT simulations. These simulations are similar to those described for UCT [6] with special handling for transitions and action sequences. Step 6 uses UCT's standard `UCTSelect` function, selecting from a set of transitions T rather than a set of primitive actions.

The rough path that is generated is broken into subpaths for individual state variables (see line 12 of Algorithm 2). This rough path is passed to Algorithm 3 in line 8 of Algorithm 1 as well as recursively in line 10 of Algorithm 3 itself. Since it is necessary to simulate these subpaths in some order, T-UCT uses a directed acyclic[2] causal graph to topologically order state variables. The causal graph is encoded by the DBN: for each pair of state variables S_i, S_j, we say that S_i depends on S_j if any of the CPTs associated with S_i have refinements on state variable S_j. In this case we add a directed edge from S_j to S_i in the causal graph of our domain, indicating that the value of S_j is significant in determining the effect some action has on S_i.

The agent simulates variable subpaths in reverse order of each variable's depth in the causal graph, beginning with deeply nested variables and working its way up to the roots. The intuition here is that altering a deeply nested variable that is directly or indirectly dependent on other variable settings will require changes in those variables as well, whereas altering a root variable would not require changing any other variables. So the agent begins simulating with action sequences that have widespread effects on its state, and ends with action sequences whose effects are more localized. When the target context has been reached the agent calls Algorithm 4 in line 17.

Algorithm 3 T-UCT path simulation algorithm.

```
1: function SimulatePath(SourceContext sc, TargetContext tc, Path p)
2:     S ← ∅, s' ← sc, c_d ←MEA(sc, tc)        //Means Ends Analysis
3:     for all v in GetTopologicalVariables(c_d) do
4:         while sc[v] ≠ tc[v] do
5:             T ← GetTransitions(sc, p[v])
6:             t ← UCTSelect(sc, T)
7:             c_t ← Context(t)
8:             a_t ← Action(t)
9:             p' ← PlanRoughPath(sc, c_t)
10:            sc, r_t, n_t ← SimulatePath(sc, c_t, p')
11:            s', r ← SimulateAction(sc, a_t)
12:            r ← r_t + γ^{n_t} · r
13:            Push(S, (s, a_t, r, n_t + 1))
14:            sc ← s'
15:        end while
16:    end for
17:    r, n ← UpdateQ(S)
18:    return s', r, n
19: end function
```

Algorithm 4 is a modification of the standard temporal difference update used in Q-Learning [11] that considers sequences of actions to compute cumulative discounted reward. State-action values (Q) are computed in reverse order, beginning with the last action in the sequence and ending with the first. Each item in $\mathcal{S}$ represents execution of a transition, which consists of all steps necessary to reach the transition's context as well as the transition's final action. In line 5 the agent accounts for cumulative discounted reward for successive states and actions. In line 6 the agent updates its Q function based on the learning rate α. In line 7 the agent takes care to update the discount factor γ' such that the reward is discounted for each step taken per transition. Finally in line 8 the agent estimates the cumulative reward r based on the observed reward r' and the maximal reward from the state-value function.

[2]As with VISA, an acyclic causal graph is not required, but will improve performance. VISA handles cycles by merging state variables into strongly-connected components. T-UCT simulations are simply more likely to succeed with an accurate topological ordering.

Algorithm 4 T-UCT Q Update for Action Sequences.

function UPDATEQ(StateActionRewardStack $\mathcal{S}$)
  $\gamma' \leftarrow 1, r \leftarrow 0, n \leftarrow 0$
  while $\mathcal{S} \neq \emptyset$ **do**
    $s, a, r', n' \leftarrow$Pop($\mathcal{S}$)
    $r' \leftarrow r' + \gamma' r$
    $Q(s,a) \leftarrow \alpha r' + (1-\alpha) Q(s,a)$
    $\gamma' \leftarrow \gamma' \cdot \gamma^{n'}$
    $r \leftarrow \lambda r' + (1-\lambda) \arg\max_{a'} Q(s,a')$
  end while
  return r, n
end function

3.5 Execution

The final step of T-UCT is to execute an action sequence based on the best computed path. The path p that is returned by Algorithm 1 takes the form of a stack of state-action pairs. Each action is paired with the state that is expected to follow as a result of executing the action. At each timestep t the agent pops the top pair s, a off of the stack, executes a, and observes the successor state s_{t+1}. If s and s_{t+1} match then the agent continues to the next timestep. If no change is observed from s_t to s_{t+1}, the agent executes a again up to a maximum of 10 times. If s_{t+1} and s don't match then the agent has diverged from the expected path and thus restarts Algorithm 1. Pseudocode for the execution phase is given in Algorithm 5.

Algorithm 5 T-UCT Execution Algorithm.

function EXECUTEPATH(State s_t, Path p, MaxTries N)
  if $p = \emptyset$ **or** $n \geq N$ **then** *//Path empty or failed*
    $p \leftarrow$T-UCTSearch(s_t), $n \leftarrow 0$
  end if
  $s, a \leftarrow$Pop(p)
  $s_{t+1} \leftarrow$TakeAction(a)
  if IsMatch(s_t, s_{t+1}) and $n < N$ **then**
    $n \leftarrow n + 1$ *//Increment counter*
    Push($p, (s, a)$) *//Retry action*
  else if not IsMatch(s, s_{t+1}) **then**
    $p \leftarrow$T-UCTSearch(s_{t+1}), $n \leftarrow 0$ *//Find new path*
  end if
  ExecutePath(s_{t+1}, p, N)
end function

4. EXPERIMENTS

In the following experiments we compare T-UCT with B-VISA and UCT for the purpose of successfully modeling environment dynamics. Each experiment is supported with independent t-tests which show that T-UCT outperforms both B-VISA and UCT with $p < .001$. We evaluate performance first on the light box domain [10] in Section 4.1 and then the random lights domain in Section 4.2.

4.1 The Light Box Domain

In this experiment we provide an empirical comparison of our own T-UCT against our implementation of B-VISA. The purpose of this experiment is both to show the effectiveness of T-UCT and the baseline performance of our implementation. Our environment is the light box domain: a collection of 20 lights and 20 toggle actions in which a light can only be toggled if its dependencies are met, and each action has a 10% chance of having no effect. The causal graph for this domain is given in Figure 3. The causal graph indicates the dynamics of the environment. For instance, light 17 is dependent on lights 11 and 12, and thus light 17 can only be toggled "on" if lights 11 and 12 are already on. Additionally if the agent attempts to toggle a light whose dependencies aren't met, the entire domain is reset to "off".

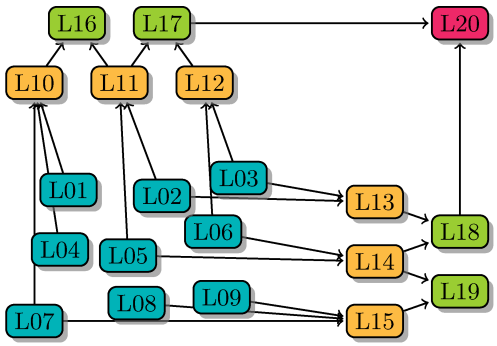

Figure 3: The light box domain introduced by Vigorito and Barto [10] represented as a causal graph. Directed edges represent dependencies.

Lights in this domain are arranged hierarchically, and lower levels of the hierarchy must be understood before an agent can reliably manipulate lights in higher levels. Generally an agent will not be able to reliably toggle lights in upper levels of the hierarchy until it understands their dynamics. For example, the agent will often fail toggling light 17 until it learns that light 17 depends on lights 11 and 12 being on, as well as the dependencies for lights 11 and 12.

The ground truth model of the light box domain contains 424 refinements. 400 of these (one for each CPT) are *reflexive* refinements: a CPT for action a and state variable S_i always depends on S_i. The additional 24 refinements stem from dependencies between state variables in the causal graph, and are indicated by the 24 directed edges of Figure 3.

We note that some clarifications are necessary to stick to this number of refinements. The reset mechanics in particular have the possibility of effecting new refinements in the ground truth model. For example, if light 18 is off, and an agent attempts to toggle light 20, all lights will turn off. Thus, all lights are dependent on light 18. To avoid these refinements, when a reset is triggered, we pass the tuple (s_t, a, s_t) instead of (s_t, a, s_{t+1}) to the agent. Thus the agent learns that toggling a light with unmet dependencies has no effect on the state, but must still handle the added difficulty of having the domain reset. Secondly we note that if a light is already on, then no dependencies need be met to toggle the light off. These two details restrict the domain's ground truth model to 424 total refinements.

As shown in Figure 4 we see that T-UCT is able to find all refinements in approximately half the number of timesteps required by B-VISA. T-UCT performs particlarly well in comparison with B-VISA when learning deeper levels of the model hierarchy. We attribute this improvement to the fact that T-UCT considers intermediate entropy gain when executing sequences of actions, rather than picking a target state and only considering the entropy gain at that state. Action sequences become longer at deeper levels of the hierarchy, and thus the agent benefits most from T-UCT's intermediate reward consideration at these levels. Our next experiment demonstrates this advantage more clearly.

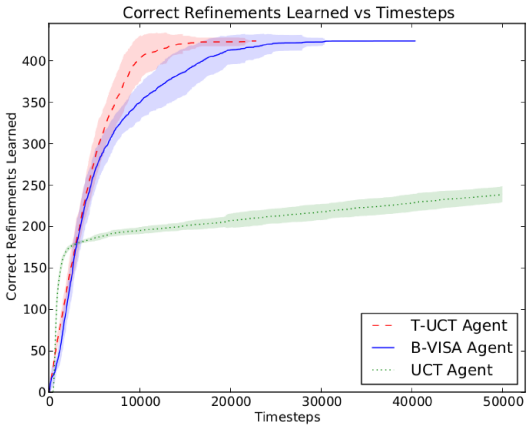

Figure 4: A comparison of T-UCT, B-VISA, and UCT agents on the light box domain [10]. The data show the number of timesteps required for the agents to learn all of the 424 CPT refinements needed to model the domain, averaged over 25 trials. Shaded regions represent standard error.

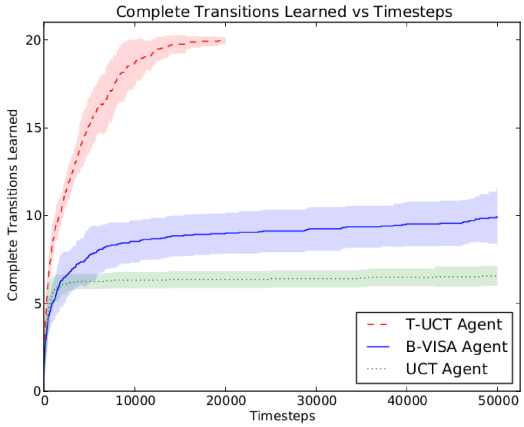

Figure 5: A comparison of T-UCT, B-VISA and UCT agents on the random light domain. The graph shows the number of correct transitions identified by each agent out of the 20 available, averaged over 25 trials with a maximum of 50,000 timesteps. Shaded regions represent standard error.

4.2 The Random Light Domain

Our next experiment was designed to test the effectiveness of T-UCT at navigating deeper hierarchies. We randomly generated a domain with similar rules to the light box domain and selected a domain with a greater number of hierarchical levels. A causal graph is shown in Figure 6.

Similar to the light box domain, the random light domain is comprised of binary variables that can be switched on or off if their dependencies are met. If a toggle action is taken on a variable with unmet dependencies, the entire domain is reset to off and the tuple (s_t, a, s_t) is recorded by the agent. Additionally we initialize every CPT with a reflexive refinement at the root: each CPT for state variable S_i and action a depends on S_i. The purpose of this heuristic is

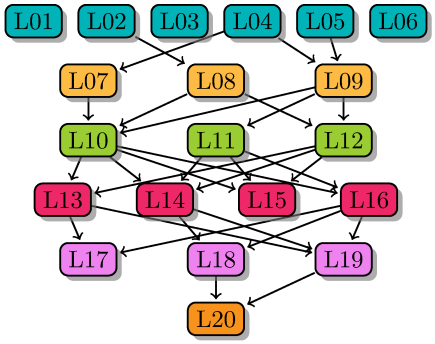

Figure 6: A randomly generated analogue of the light box domain represented as a causal graph. Directed edges represent dependencies.

to focus our evaluation on the agent's ability to learn non-trivial transition dynamics.

The results in Figure 5 show that T-UCT is able to effectively navigate through a deeply hierarchical domain and quickly learn the environment dynamics. In all cases T-UCT is able to solve the domain in less than 21,000 timesteps. T-UCT's consideration of complete entropy gain along an action sequence, rather than B-VISA's focus on entropy gain at endpoints, allows the agent to make better use of its actions.

5. CONCLUSION AND FUTURE WORK

We have presented a new model-based planning algorithm for navigating hierarchical domains and maximizing expected rewards. In our test domains T-UCT's sample efficiency consistently outperformed B-VISA, and performed particularly well with deeper levels of hierarchy. T-UCT meets two of our three desiderata from Section 2.4 through its use of partial models and its sample efficiency.

T-UCT was designed with linear scaling in mind, however both T-UCT and B-VISA rely on tracking the effects of all actions on all state variables. This results in $O(n^2)$ processing time and memory usage for domains on n variables and n actions. Ideally a model-learning solution would be capable of ignoring particular state-action pairs that are unlikely to be related. Avoiding unnecessary computation in this manner is thus a promising direction for future work.

While we did not perform experiments based on extrinsic, stochastic rewards it is conceivable that model-based path planning would allow an agent to more easily exploit an environment's reward dynamics by enabling the agent to reach high-reward areas of the state space that might not otherwise be accessible. T-UCT's effectiveness stems from its ability to maximize cumulative reward and thus the source of the reward, intrinsic or extrinsic, is irrelevant. Nonetheless, one area of improvement for T-UCT would be to balance exploiting extrinsic rewards with intrinsic (entropy-based) rewards for the purpose of model learning.

Acknowledgements

This work has taken place in the Learning Agents Research Group (LARG) at the Artificial Intelligence Laboratory, The University of Texas at Austin. LARG research is supported in part by grants from the National Science Foundation (CNS-1330072, CNS-1305287), ONR (21C184-01), AFOSR (FA8750-14-1-0070, FA9550-14-1-0087), and Yujin Robot.

REFERENCES

[1] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

[2] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational intelligence*, 5(2):142–150, 1989.

[3] B. Hengst. Discovering hierarchy in reinforcement learning with hexq. In *ICML*, volume 2, pages 243–250, 2002.

[4] A. Jonsson and A. Barto. A causal approach to hierarchical decomposition of factored mdps. In *Proceedings of the 22nd international conference on Machine learning*, pages 401–408. ACM, 2005.

[5] A. Jonsson and A. Barto. Active learning of dynamic bayesian networks in markov decision processes. In *Abstraction, Reformulation, and Approximation*, pages 273–284. Springer, 2007.

[6] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[7] A. Newell, H. A. Simon, et al. *Human problem solving*, volume 104. Prentice-Hall Englewood Cliffs, NJ, 1972.

[8] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[9] N. A. Vien and M. Toussaint. Hierarchical monte-carlo planning. 2015.

[10] C. M. Vigorito and A. G. Barto. Intrinsically motivated hierarchical skill learning in structured environments. *IEEE Transactions on Autonomous Mental Development*, 2(2):132–143, June 2010.

[11] C. J. C. H. Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge, 1989.