State Generalization in UCT

by

Srinivasan Sriram

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

In this thesis, I study the problem of Monte-Carlo Planning in deterministic domains with sparse rewards. A popular algorithm in this suite, *UCT*, is studied. A new algorithm to incorporate state generalization in UCT using estimates of similar nodes and a distance metric is presented. The algorithm's correctness and asymptotic convergence to optimality under certain conditions on the domain is also shown. A second contribution in this thesis includes an algorithm for learning a *local* manifold of the state space when the state space does not have a natural distance metric and use it for state generalization in UCT. The effectiveness of the algorithm is studied by measuring its performance on multiple domains inspired by video games. Empirical evidence shows that the new algorithm is more sample efficient than UCT, especially on sparse reward games.

Acknowledgements

I would like to thank my supervisors, Dr. Michael Bowling and Dr. Erik Talvitie, for guiding me throughout this journey. Their keen insights and suggestions helped me on numerous occasions, especially during the tough times. A special thanks to Erik Talvitie for reviewing my code for performance speedups.

I would also like to thank Dr. Csaba Szepesvári for helping me with the proofs. A special thanks to Ujjwal Das Gupta, friend and colleague. I thoroughly enjoyed our discussions on research and life in general. Hope to have more in future.

I would also like to thank Dr. Rich Sutton and the RLAI lab members for the various helpful discussions. They have helped me understand RL research quickly and also shape my outlook towards research. Rich Sutton remains an inspiration for my interest in RL.

I would also like to thank my friends - Ankush, Ujjwal, Talat for the fun and support. I hope it continues in future too.

I would like to thank my parents for their unconditional love and support.

Contents

List of Figures

Chapter 1

Introduction

> One must be sane to think clearly, but one can think deeply and be quite insane.
>
> Nikola Tesla

Humans, as well as computers, are constantly involved in decision-making. In many cases, the decision-making process involves evaluating the outcomes of current as well as future decisions. This process, commonly called planning, requires a model to simulate future decisions and their outcomes without actually taking them. If the model is accurate, the outcomes of the decisions can be estimated reliably. Although an accurate model enables reliable estimation of outcomes, the problem can become hard if the search space of decisions gets very large. As the agent looks ahead into the future for measuring the outcomes, the space increases exponentially. A finite computational budget necessitates the need for efficient search.

The search process needs to be efficient in two ways:

1. It must perform a reasonably far look-ahead of the outcomes and pick the one which is most beneficial. The benefit is usually measured by the rewards the agent can possibly accrue in the future (exploitation).

2. It must also simulate decisions which are outside of the current best choices. This allows the agent to consider choices which may not look promising in the very near future but better in the long term (exploration).

Notice that the actions needed to meet the two objectives are mutually opposing. While the first objective requires looking further into the future, it limits exploring other choices in the near future which may be valuable in the long term. The second

objective of exploration limits the search from looking further in the horizon (time upto which future rewards are considered) for the choices that have been evaluated with the current horizon.

An interesting situation arises when all the decisions look equally poor in the near future, but there exists a sequence of decisions whose outcome is better than many other sequences when the length of the sequence is longer than the near future horizon. Such problems are also called sparse reward problems. One such example is the game of Freeway, where the chicken receives a reward of one only when it crosses the road completely. Identifying the best decision at the current time step would involve a distant look-ahead but the available computational budget may be insufficient to perform the look-ahead for all possible decisions in the future. Thus the search process needs to grow in areas whose outcomes look more promising, while also exploring other choices. In this thesis, I focus on such sequential decision problems. In particular, I attempt to solve this problem using a popular planning algorithm —UCT. The UCT algorithm balances look-ahead and exploration very well in general and has been successfully used in achieving master capability levels in games such as computer *Go*. However, UCT performs poorly in sparse reward problems as it spends most of its computational budget searching in regions (states) similar to those it has already seen.

One way to tackle this problem is by generalization. If actions at a particular time step lead to states similar to those that have already been encountered, the search procedure can avoid expanding the search in those states and instead focus on other states whose outcomes are unknown or unreliable yet. To achieve generalization, one needs to be able to measure similarity between states. If these states were points in Euclidean space, we could use distance between them for a similarity metric. However, if they aren't points in Euclidean space, we can *embed* them in Euclidean space and compute distances. In this work, I introduce a method for incorporating state generalization in UCT using these ideas. Specifically, I make two contributions:

1. I introduce Nearest Neighbor UCT (NN-UCT), for incorporating state generalization in UCT through a designer provided distance metric.

2. I describe how a manifold learning algorithm can be applied to provide a distance metric in NN-UCT, when a natural distance metric is unavailable.

We call this combination, the mNN-UCT algorithm.

I begin by reviewing background material (Chapter 2). Related work is presented along with the contributions in Chapters 3 and 4. All the algorithms presented here are tested on domains inspired by video games, described in Chapter 5. Finally, Chapter 6 concludes the work with directions for future research.

Chapter 2

Background

Sequential decision problems involve an agent making a sequence of decisions at discrete time steps in a known or unknown environment. The goal of the agent is to maximize its future discounted sum of rewards. The rewards may be obtained from the environment (Sutton and Barto, 1998) or intrinsically generated (Oudeyer et al., 2007) or simply the reward of gathering knowledge about the environment (Sutton et al., 2011). Sequential decision problems can be described using the Markov Decision Process (MDP) setting (Puterman, 2009). The MDP framework makes the assumption that there is a *state* variable that completely describes the environment, which is formally called the Markovian assumption. Knowledge of the state and the dynamics of the environment are sufficient to predict future rewards.

2.1 Markov Decision Processes (MDPs)

In this thesis, I focus on deterministic MDPs. Such a MDP has no randomness involved in its state transitions as well as reward function. A deterministic MDP $\mathcal{M}$ is a tuple $\langle S, A, T, \rho, \gamma \rangle$, where each one of the components are:

- S: The set of all states, with s_0 being the initial state.
- A: The set of actions available to the agent in the environment.
- $T : S \times A \rightarrow S$: The transition function returns the next state, given a state-action pair.
- $\rho : S \times A \rightarrow \mathbb{R}$: The reward function returns a scalar as reward obtained for taking an action from a state.

- $\gamma \in [0, 1]$: The discount factor acts as a tradeoff between short-term and long-term rewards.

The dynamics of the environment can be described as follows: at timestep t, the agent observes state s_t and selects an action $a_t \in A$ and receives reward $r_t = \rho(s_t, a_t)$; the next state s_{t+1} is given by the transition function $T(s_t, a_t)$. The Markov assumption of the environment can be formally described as:

$$\Pr\left(s_{t+1}, r_t | s_0, a_0, ..., s_{t-1}, a_{t-1}, s_t, a_t\right) = \Pr\left(s_{t+1}, r_t | s_t, a_t\right)$$

In other words, the next state and immediate reward is dependent only on the current state and action. The Markov assumption is also captured in the parameterization of the transition and reward function, where the arguments include only a state-action pair.

The behavior of an agent can be described by a *policy* $\pi : S \times A \to [0, 1]$, where $\pi(a|s)$ denotes the probability of taking action a in state s. We can *evaluate* a policy using its γ*-discounted expected return* defined as:

$$\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \rho\left(s_t, a_t\right) \right].$$

Now that we can evaluate a policy, a natural question is to find an *optimal* policy, $\pi^* = \arg\max_\pi \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \rho\left(s_t, a_t\right)\right] \forall s$.

2.2 Value Function

For a given deterministic MDP $\mathcal{M}$ and policy π, we can define a *state-value* function $V_\pi(s) : S \to \mathbb{R}$, as the expected return obtained when starting in state s and following policy π, i.e.,

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \rho(s_t, a_t) \mid s_0 = s \right]$$

We can also define an *state-action value function* $Q_\pi(s, a) : S \times A \to \mathbb{R}$, which is a mapping from state-action pairs to the expected return achieved by starting in state s, taking action a and following policy π thereafter, i.e.,

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \rho(s_t, a_t) | s_0 = s, a_0 = a \right] \tag{2.1}$$

The Markov property of MDPs allows us to write the state-action value function recursively, known as the *Bellman* equation :

$$Q_\pi(s,a) = \rho(s,a) + \gamma \sum_{a' \in A} \pi(a' \mid T(s,a)) Q_\pi(T(s,a), a'). \tag{2.2}$$

Thus, the state-action value function and value function are related as,

$$Q_\pi(s,a) = \rho(s,a) + \gamma V_\pi(T(s,a)).$$

The optimal policy π^* has a value function

$$Q^*(s,a) = \rho(s,a) + \gamma \max_{a' \in A} Q^*(T(s,a), a').$$

2.3 Monte-Carlo Tree Search

Now that we have described the sequential decision problem in the form of MDPs, we can now focus on planning algorithms to determine actions in the optimal policy. Recall from Chapter 1 that the search process should balance the two objectives of looking ahead into the future and exploring other actions. One of the ways to achieve this balance is by building a search tree using samples from the model. The tree consists of nodes corresponding to states and edges corresponding to actions taken from a state. These methods are called Monte-Carlo Tree Search (Browne et al., 2012) algorithms.

A general Monte-Carlo Tree Search (MCTS) algorithm is shown in Algorithm 1. The algorithm builds a search tree iteratively, until there is no computational budget available. This budget can be defined in terms of time or memory or a constraint on the number of iterations. When there is no more budget available, the function returns the best action from the root node. The search procedure is called from a start state (s_0). It then builds the tree, adding one node at a time. MCTS algorithms compute value functions to determine action selection at each level in the tree, eventually leading to the creation of a leaf node. Once a leaf node is created, the algorithm performs a *rollout*, which is a sequence of actions dictated by a pre-specified *default policy*. The algorithm uses the returns (discounted sum of rewards) collected from these rollouts to determine the value of the node. A formal description follows.

Let D denote the set of nodes in the tree. Note that two or more nodes can share the same state $s \in S$. Let $S(d)$ denote the state corresponding to node $d \in D$ and

$A(d)$ denote the action from the parent node, leading to node $d \in D$. Each node maintains the sum of returns obtained by rollouts starting from that node, $R(d)$, as well as the number of times it has been visited, $n(d)$. The value of the node is, $V(d) = \frac{R(d)}{n(d)}$. The action value estimate is given by $Q(d, a) = \rho(S(d), a) + \gamma * V(d')$, where d' is the child node reached by taking action a.

Algorithm 1 MCTS algorithm

function MCTS($s_0, n, depth$) ▷ n = number of rollouts, $depth$ = length of rollout
    create root node d_0 with state s_0
    while $n > 0$ **do**
        $d_l \leftarrow$ TREEPOLICY(d_0) ▷ Tree Policy algorithm dependent
        z $\leftarrow$ DEFAULTPOLICY(S(d_l),$depth$,γ)
        BACKUP(d_l,z)
        $n \leftarrow n - 1$
    end while
    return $\arg\max_{a \in A} Q(d_0, a)$
end function

function DEFAULTPOLICY($s, depth, \gamma$)
    if s is terminal or $depth == 0$ **then**
        return 0
    else
        Choose $a \in A$ uniformly at random
        $r \leftarrow \rho(s, a)$
        $s \leftarrow$ T(s, a)
        $depth \leftarrow depth - 1$
        $r \leftarrow r + \gamma$ * DEFAULTPOLICY($s, depth, \gamma$)
        return r
    end if
end function

function BACKUP(d, z)
    while d is not null **do**
        $n(d) \leftarrow n(d) + 1$
        $R(d) \leftarrow R(d) + z$
        $a \leftarrow A(d)$
        $d \leftarrow$ parent of d
        if d is not null **then**
            $z \leftarrow \rho$(S(d),a) + $\gamma * z$
        end if
    end while
end function

An MCTS algorithm consists of the following components :

1. Tree Policy - The tree policy is used to perform action selection at each level in the tree, eventually leading to the creation of a leaf node. Thus, the tree policy dictates the way the tree is grown during planning.

2. Default Policy - This is used to run a simulation, starting from the state corresponding to the leaf node, until a certain length or termination. The default policy is usually uniformly random.

3. Backup - The backup procedure updates the nodes of the tree, starting from the leaf, up to the root, along the branch chosen, with the return accumulated from the default policy. The backup procedure updates the value estimates of the nodes, which would affect the tree policy in the next iteration.

Finally, when no more simulations are run the greedy action from the root is chosen and returned. Ties are either broken based on visit counts, or randomly.

2.4 UCT - UCB on Trees

UCT (Kocsis and Szepesvári, 2006) is an MCTS algorithm that uses the UCB1 algorithm (Auer et al., 2002) for its tree policy. UCT treats the action selection problem at each sub-tree as a multi-armed bandit problem. For the multi-armed bandit problem, UCB1 tries to solve the *exploration-exploitation* dilemma by computing a score that combines the two quantities. Assuming that the payoffs of the arms are i.i.d. (identically and independently distributed), the UCB1 algorithm has a regret that grows logarithmically in the number of arm pulls. The regret is the loss incurred by not pulling the best arm in hindsight. The UCB1 aims to balance the *exploration-exploitation* dilemma by keeping track of the average rewards of the arms and picking the one with the best upper-confidence bound. The average reward of arm i is given by $\bar{X}_{i,T_i(t-1)} = \frac{1}{T_i(t-1)} \sum_{j=0}^{t-1} X_{i,T_i(j)}$, where $X_{i,j} \in [0,1]$ denotes the rewards and $T_i(t) = \sum_{j=0}^{t} \mathbb{1}(I_j = i)$ is the number of times arm i has been pulled from time 0 to t (included). The UCB1 algorithm picks the arm in the following way :

$$I_t = \underset{i \in \{1,2,..,K\}}{\arg\max} \bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)}, \qquad c_{t,s} = \sqrt{\frac{2 \ln t}{s}}. \tag{2.3}$$

The UCB1 algorithm uses the optimism under uncertainty principle. The measure of uncertainty is expressed by the exploration bonus term. Thus, if a certain arm

hasn't been pulled a sufficient number of times, its exploration bonus grows with time, forcing the arm to be pulled. This helps in improving the reliability of the estimates of less promising arms, as well as avoiding selecting a sub-optimal arm.

Algorithm 2 UCT Tree Policy

function TREEPOLICY(d)
  while d is non-terminal **do**
    if d has unexplored actions **then**
      return EXPAND(d)
    else
      $d \leftarrow$ BESTCHILD(d, C)
    end if
  end while
  return d
end function

function BESTCHILD(d, C)
  $a \leftarrow \arg\max_{a \in A} Q(d,a) + C * \sqrt{\frac{\ln \sum_{a' \in A} n(d,a')}{n(d,a)}}$
  return f(d,a)
end function

function EXPAND(d)
  Pick $a \in$ untried actions from A
  add a new child d' to d with
  S(d') $\leftarrow$ T(S(d),a)
  $f(d,a) \leftarrow d'$
  $A(d') \leftarrow a$
  $n(d') \leftarrow 0$
  $R(d') \leftarrow 0$
  return d'
end function

Algorithm 2 shows the tree policy of UCT. The UCT algorithm treats the return obtained by the Monte-Carlo simulations as i.i.d random variables. By Hoeffding's inequality, these estimates concentrate around the mean quickly at the leaf nodes. However, since the sampling probability of actions changes in the subtree, the pay-offs may change over time. But, with an appropriate exploration bonus, the change is compensated for.

In the UCT algorithm, the UCB1 algorithm is used to select actions at each level in the tree. Hence, the UCB score at node d and action a is given by:

$$Q_{ucb}(d,a) = Q(d,a) + C * \sqrt{\frac{2 \ln \sum_{a' \in A} n(d,a')}{n(d,a)}}. \tag{2.4}$$

The second term is an exploratory bonus, which encourages actions taken less frequently from a node, where C is a positive parameter, $n(d, a)$ is the number of times, action a was taken from node d and hence $n(d, a) = n(d')$, where d' is the child node reached by taking action a. After the computational budget is exhausted in building the tree, the greedy action at the root, $\arg\max_{a \in A} Q(d_0, a)$ is picked.

2.4.1 Convergence of UCT

Given a finite horizon MDP, under the assumption that the returns from the default policy are i.i.d samples, UCT converges to the optimal policy, asymptotically. The probability of choosing a suboptimal action at the root converges to 0 at a polynomial rate in the number of rollouts. In practice, UCT has been applied successfully in many games such as computer *Go*. The UCB constant (C) is the only parameter that needs to be tuned to the specific domain.

2.5 Enhancements to UCT

A few popular enhancements to UCT are described below. These enhancements are used to make UCT sample efficient in practice and have no affect on the asymptotic convergence to optimality.

1. Retaining the optimal subtree between planning steps: Once the planning budget is exhausted, UCT returns the greedy action at the root. A useful optimization is to retain the branch of the tree corresponding to the chosen action for the next planning step. This helps UCT grow its tree deeper in the next planning step, as the UCT tree for the next planning would consist of nodes in the optimal branch.

2. Transposition tables: Transposition tables (Childs et al., 2008) are lookup tables to use values of state-action pairs that have already been encountered in the tree. This allows statistics to be shared across nodes with the same state. When memory is at a premium, care must be taken in ensuring that all the child nodes are considered for action selection when only some of them are stored in the table.

3. After-state trick (Méhat and Cazenave, 2010): Instead of storing statistics (returns, visit counts) as state-action pairs in the edges, we can store the statistics

in the nodes corresponding to the after-state. This helps in the exploration being guided towards promising after-states, instead of relying on the counts of the actions being taken from a state.

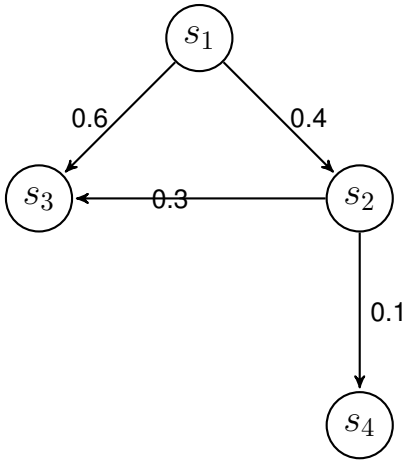
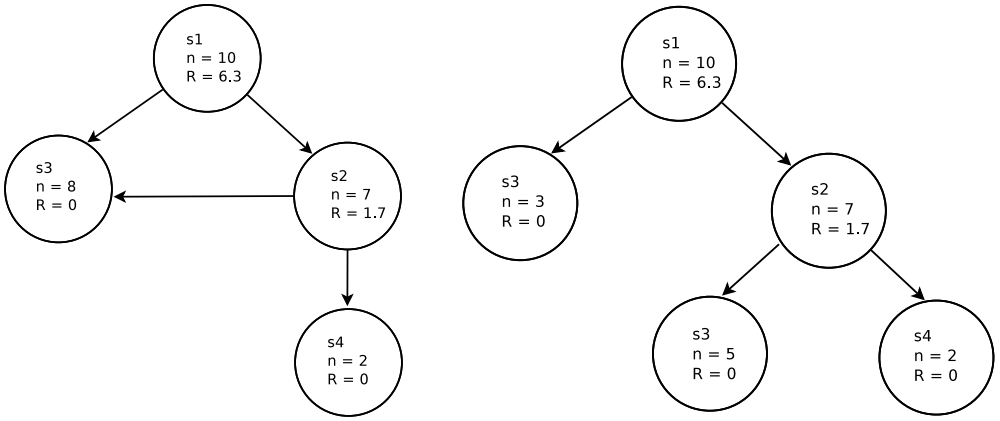


Figure 2.1: DAG MDP State Diagram



(a) UCT with Transpositions (b) UCT without Transpositions

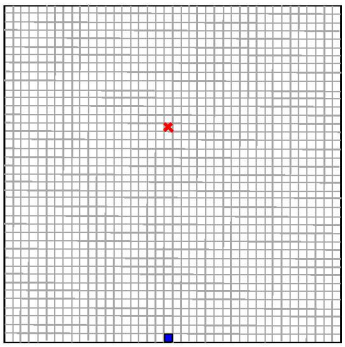
Figure 2.2: UCT Implementations

Figure 2.1 shows the state diagram of an MDP whose state transitions form a Directed Acyclic Graph (DAG) with states s_3 and s_4 being terminal states. There is a transposition at state s_3 as it is reachable from both states s_1 and s_3. Figure 2.2 shows the UCT tree with and without transpositions, after 10 iterations. The after-state trick along with transpositions turns the UCT tree in to a DAG with returns and visit counts stored in the nodes, rather than the edges. Notice that the visit count of state s_3 is higher than one of its parents s_2 as it is visited from both the root (s_1) as well as state s_2. Without transpositions, two different nodes for the same state s_3 are created. The sum

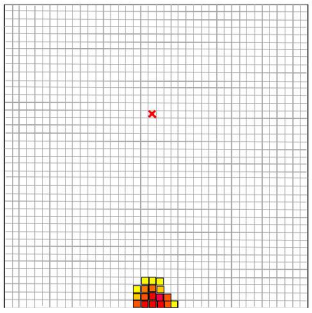
of the returns and visit counts of the two nodes sum up to the total visit count of the state.

Chapter 3

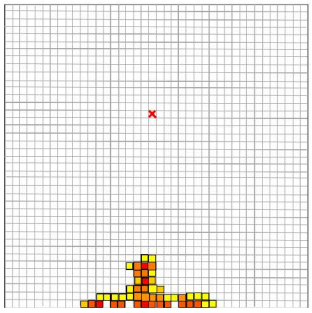
State Generalization in UCT



(a) Grid World



(b) Heat map of UCT after 100 rollouts



(c) Heat map of NN-UCT after 100 rollouts

UCT has been successfully used in many sequential decision problems and was able to achieve *master* level in computer *Go*. Its balance of exploration-exploitation using confidence bounds is built on the sound theory of regret minimization.

However, UCT performs poorly when the rewards are sparse. Consider the grid world domain shown in Figure 3.1(a). The agent starts at the solid square in blue. The goal state is marked by a red X and the agent receives a reward of $+1$ on reaching the goal state, leading to episode termination. The agent has 4 actions available - move UP, DOWN, LEFT and RIGHT. In the absence of reward from rollouts, UCT essentially follows random behavior. For a reasonably distant goal state, a uniform random policy is unlikely to observe any reward. Figure 3.1(b) shows the heat map of states visited by UCT after 100 rollouts from the start state. Due to its random rollout policy, UCT keeps visiting the same or nearby states. As a result, UCT needs a large number of rollouts or very long rollouts to see reward, and therefore to make a good decision.

An alternative approach would be to systematically explore the state space, collecting returns from states that are different from those already visited. However,

performing an exhaustive search of the state space would be impractical. We want to retain the strengths of Monte-Carlo planning in using samples, while still achieving efficient exploration. I aim to exploit similarity between states and augment UCT with this information. Using this information, we can *generalize* to new states. The state generalization can be incorporated in the tree policy of UCT, specifically in the UCB score computed for action selection in the tree policy. By doing so, UCT can now grow its tree towards new and dissimilar states, thereby helping UCT see possible reward in distant future states. This would help UCT choose the optimal action from the start state, that helps it obtain reward(s) in future.

3.1 NN-UCT

Even if we haven't visited a state before, a good estimate of the node's statistics can be obtained from the statistics of other *similar* nodes in the tree, thus making better use of samples. A natural way to incorporate this form of generalization is to combine the statistics of similar nodes and use it to compute the UCB score. One way to achieve this is to compute a nearest neighbor estimate of the return and visit counts for each node d, as given by :

$$R_{nn}(d) = \sum_{d' \in D} K(S(d), S(d')) * R(d')$$

$$n_{nn}(d) = \sum_{d' \in D} K(S(d), S(d')) * n(d')$$

$$V_{nn}(d) = \frac{R_{nn}(d)}{n_{nn}(d)}$$

where $K(s, s')$ is a kernel function that measures similarity between states s and s'. A popular choice for the kernel function is the Gaussian kernel function, $K(s, s') = \exp\left(\frac{\|s-s'\|_2^2}{\sigma^2}\right)$, where $s, s' \in \mathbb{R}^d$, σ is the Gaussian width parameter which controls the degree of generalization. As $\sigma \to 0$, the nearest neighbor estimate approaches the Monte-Carlo (MC) estimate of all of the transpositions. If generalization is turned off, we can still reap the benefit of using transposition tables, described in section 2.5, without additional memory but with additional computational cost for every run of the tree policy. The computational cost is linear in the number of nodes in the UCT tree for every level at which the tree policy is executed. Although transposition tables cost less in combining statistics of nodes with transpositions, our approach helps us to introduce higher degrees of generalization.

Note that the nearest neighbor estimate adds bias, which could prevent the guarantee of continual exploration of all actions. Hence, asymptotically, we want the returns and visit count estimates to be closer to the unbiased MC estimates. One solution is to employ a decay scheme, that shrinks the Gaussian widths of the nodes as planning continues. In section 3.1.1, we discuss one such scheme for reducing σ smoothly over time. The new UCB score is computed as :

$$Q_{ucb}(d,a) = Q_{nn}(d,a) + C * \sqrt{\frac{2 \ln \sum_{a' \in A} n_{nn}(d,a')}{n_{nn}(d,a)}}$$

where $Q_{nn}(d,a) = \rho(S(d),a) + \gamma * V_{nn}(d')$, where d' is the child node of node d when action a is taken from state $S(d)$. The quantity $n_{nn}(d,a) \equiv n_{nn}(d')$ is taken from the child node d'. Thus, the nearest neighbor estimate of the number of times action a was taken from state $S(d)$ is obtained from the number of state visits to the after-state $S(d')$. In our algorithm, unlike UCT, $\sum_{a \in A} n_{nn}(d,a) \neq n_{nn}(d)$. Thus, neither the actual visit count ($n(d)$), nor the nearest neighbor estimate ($n_{nn}(d)$) of the parent node is used in the exploration term. Hence, only the estimates of the child nodes (after-states, as described in section 2.5) are used for action selection and actions leading to after-states similar to the ones already explored are not preferred. Figure 3.1(c) shows the heatmap of states visited by NN-UCT on the grid world domain after 100 rollouts from the start state where each state is given by a pair $(i,j), 0 \leq i,j < 40$, and a Gaussian kernel with $\sigma = 100$ is used with a decay scheme (explained in the next section) that uses $\beta = 0.9$. NN-UCT directs tree to be grown in areas farther from states that have been encountered already. In the absence of reward, with ties bro- ken using visit counts, the action picked at the root would be one that leads to a state where rollouts could see non-zero reward.

3.1.1 Decay Scheme

Given the initial Gaussian kernel width σ, we want to develop a scheme that shrinks the Gaussian widths of nodes such that the value estimates and visit counts are closer to the MC estimate, asymptotically. We use a simple decay rate, β, where $0 < \beta < 1$. In this scheme, we shrink the Gaussian width of the child nodes whenever the tree policy is executed at the parent node. Thus, the Gaussian width of a node d' is given by

$$\sigma(d') = \sigma * \beta^{n(d)},$$

where $n(d)$ is the count of the visits to the parent node d. The Gaussian width of the root node is shrunk after each rollout. This scheme ensures that if erroneous generalization has starved visits to d', it will eventually be explored once its parent has been visited sufficiently often. The next section formalizes this intuition.

3.2 Theoretical Analysis

For our theoretical analysis, we assume that the deterministic MDP is episodic and the maximum episode length is finite. We also assume for the moment that the underlying state transition graph is a DAG. We consider transpositions and use the values and visit counts of the after-states. One way of handling this in UCT is to create only one node for each transposition as shown in Figure 2.2(a), making the UCT tree also a DAG. The root of a DAG is the node which has a path to every other node in the DAG. Leaf nodes in the DAG correspond to terminal states, and do not have a path to any other node in the DAG. Another way of handling transpositions while retaining the tree structure of UCT (Figure 2.2(b)) is to combine the statistics of all the transpositions. The NN-UCT algorithm achieves this while computing the nearest neighbor estimates using the kernel function. We also assume that the kernel (similarity) function converges to the Kronecker delta after a (possibly) large but finite number of rollouts, i.e.

$$K(S(d), S(d')) = \begin{cases} 1 : S(d) == S(d') \\ 0 : otherwise \end{cases}$$

Note that even after the kernel function evaluates to the Kronecker delta function, NN-UCT still accounts for transpositions in the search tree. We want to show that the NN-UCT algorithm chooses the optimal action at the root considering all rewards until termination, as the number of rollouts goes to infinity.

Let U denote the UCT DAG. A set of nodes τ of U spans a *proper* subgraph of U if τ satisfies all of the following:

1. The root and at least one leaf node of U is in τ.

2. For every non-leaf node in τ, there is a path of U through nodes only in τ to reach some leaf node in τ.

For a node $d \in \tau$, we define $child_\tau(d)$ as the set of nodes such that $d' \in child_\tau(d)$ if node d' is a child of node d in U.

For a proper subgraph τ and a node $d \in \tau$, define

$$V_\tau(d) = \max_{d' \in child_\tau(d)} \rho(S(d), A(d')) + \gamma * V_\tau(d')$$

Lemma 1. *Let τ denote the set of nodes in the UCT DAG that are visited infinitely many times. Then τ is a directed acyclic subgraph, and for all nodes in τ, the nearest neighbor estimates converge to the τ-value of the node.*

Proof. Firstly, the root is visited infinitely often and hence is in τ. The set τ also includes at least one leaf node since each rollout ends at a leaf node and there are finitely many leaf nodes in U. The value of the leaf node(s) in τ converges trivially to its τ-value, which is also its value.

It also holds that every non-leaf node in τ has a path in τ to a leaf node in τ, since there are finitely many nodes in τ. Hence τ is a proper subgraph of U.

Now that the nearest neighbor estimates at the leaf node(s) in τ converge to their τ-values, by induction, all other nodes in τ also converge to their τ-values. □

Lemma 2. *τ is the whole DAG U.*

Proof. We prove by contradiction. Let $d \in \tau$ be a node such that d' is a child of d in U, with $S(d') = T(S(d), a')$ but d' is not in τ. Let $t_{d'}$ denote the last time (index of a rollout) d' was visited. Consider any child d" of d with $S(d") = T(S(d), a")$ such that $d" \in \tau$. Since d" is visited infinitely often, $Q_{ucb}(d, a")$ converges to $\rho(S(d), a") + \gamma * V_\tau(d")$ since $\lim_{t\to\infty} \frac{\ln\left(\sum_{\tilde{a}\in A} n_{nn}(d,\tilde{a})\right)}{n_{nn}(d,a")} = 0$. Hence for all children of d in τ, UCB estimates converge to a finite value. On the other hand, since d' is visited only finitely many times but d is visited infinitely often, $Q_{ucb}(d, a')$ grows unbounded. This is because the denominator in the bonus term of $Q_{ucb}(d, a')$ converges to a finite visit count $n(d')$, while the numerator grows unbounded. Hence, there exists a time after $t_{d'}$ such that $Q_{ucb}(d, a') > \max_{d' \in child_\tau(d), A(d')=a} Q_{ucb}(d, a)$, contradicting that $t_{d'}$ was the last time node d' was visited. □

We now have the main result.

Theorem 1. *For sufficiently many rollouts, the action whose value estimate is the largest at the root is the optimal action.*

Proof. By Lemma 2, the NN-UCT algorithm visits all the nodes in the DAG, infinitely often. By Lemma 1, the values of all nodes converge to their true values. Since there are finitely many actions, the result follows. □

A couple of important points regarding the proof are noted here:
Firstly, in the above proof, we assume a general kernel function that converges to the Kronecker delta after a finite number of rollouts. Our choice of the gaussian kernel function with an exponential decay seems to work in practice while performing a finite number of rollouts, but it may be interesting to see if an appropriate fast decay scheme for the gaussian kernel function is also sufficient to retain optimality in the limit when the number of rollouts goes to infinity.

Secondly, the above proof of asymptotic convergence of NN-UCT assumes that the transition graph of the deterministic MDP is a DAG, but it can be extended to deterministic MDPs in general if we ignore transpositions. For example, when the deterministic MDP includes loops, the NN-UCT algorithm can still be made to converge to optimality in the limit by a slight modification of the kernel function. Forcing $K\left(S(d), S(d')\right) \rightarrow 0$ for $d \neq d'$ after sufficiently many but finite number of rollouts, ignores transpositions, but ensures that the value estimate of node d converges to its true value estimate in the limit.

3.3 Related Work

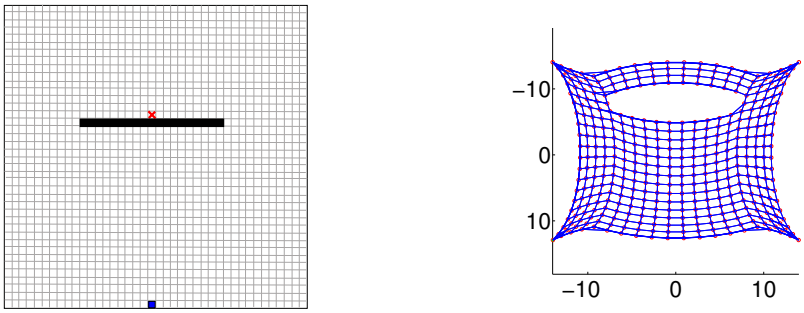
We review transposition tables, described in section 2.5. Transposition tables offer a basic form of generalization by sharing statistics of nodes in the UCT tree having the same state. Our algorithm with $\sigma \approx 0$ closely resembles UCT with transposition tables. Although transposition tables help speed up search, they provide limited exploration benefit as they can only recognize state equality. In large state spaces, the likelihood of encountering transpositions is low, and hence transposition tables may be less useful. In Chapter 5, this will be seen in even our relatively small experimental domains, with more aggressive generalization improving on basic transposition tables.

State abstraction using homomorphisms in UCT (Jiang et al., 2014) adds another layer of generalization, by grouping states with the same transition and reward distributions. The homomorphism groups states together but like transposition tables, they do not have a degree of similarity, only a hard notion of equivalence. In the case of the grid world domain in Figure 3.1(a), the only homomorphism is the grid itself. Thus, homomorphisms offer no benefit over transposition tables in this case.

Another approach to generalization in UCT is Rapid Action Value Estimation (RAVE) (Gelly and Silver, 2011). RAVE uses the *All-Moves-As-First* (AMAF) heuristic which states that the value of an action is independent of when it is taken, i.e. if an action is profitable at a subtree $\tau(s)$, it is profitable immediately at root s as well. RAVE combines the Monte-Carlo estimate with the AMAF estimate, $Q_{rave}(s,a) = (1-\beta) * Q(s,a) + \beta * Q_{AMAF}(s,a)$, where $0 < \beta < 1$ is a weight parameter that decreases with increasing number of rollouts. The RAVE estimate is used in place of $Q(s,a)$ for the tree policy. RAVE has been shown to be successful in Computer Go and has also been applied in continuous action spaces (Couetoux et al., 2011). In that case, the RAVE estimate for action a is obtained by computing a nearest-neighbor estimate of the returns in the subtree where action a was taken. However, the AMAF heuristic does not modify the exploration bonus. Thus, when no rewards have been encountered, its exploration strategy suffers from the same problems as UCT.

Chapter 4

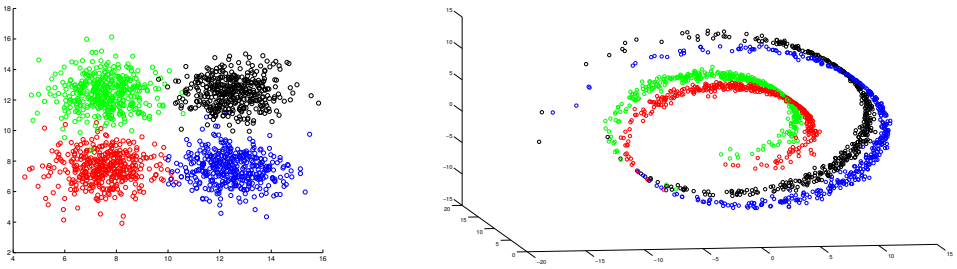
State Generalization on Complex Domains



(a) Grid World with Wall (b) Manifold Embedding of 20 × 20 grid with wall

Figure 4.1: Grid World with Wall

In the grid world example shown in Figure 3.1(a), the Euclidean distance between two points on the grid was a good choice for a distance metric. However, in general, the grid coordinates may not be the best representation. For example, consider a grid world with a wall as shown in Figure 4.1(a). In this case, the distance between grid coordinates is not a good distance metric, as states on either side of the wall, are farther apart than those on the same side. Furthermore, not all state representations have a natural metric space for defining similarity. High dimensional state spaces pose an additional problem of tractably computing a distance metric. Hence, we need a state representation that captures the underlying topology of the environment, that is low dimensional, allowing fast computation of distances between states. Low dimensional manifolds are a natural choice that satisfy all the requirements.



(a) 2D Data (b) Swiss Roll

Figure 4.2: 2D data and its 3D mapping

4.1 Manifolds

A manifold $\mathcal{M} \subset \mathbb{R}^m$ is a set of points such that the Euclidean distance between neighboring points on the manifold reflects the true distances between those points. Figure 4.2(a) shows two dimensional data from four gaussian distributions. The data was mapped to three dimensions using the mapping $\langle x, y \rangle \rightarrow \langle x * \cos(x), y, x * \sin(x) \rangle$ to result in a Swiss roll shape shown in Figure 4.2(b). The 3D data on a Swiss roll is said to lie on a 2D manifold. Note that the euclidean distance between nearby points in the Swiss roll data is not exactly the same as the Euclidean distance between nearby points in the two dimensional data, but approximately equal.

A manifold embedding refers to the representation of the points in the data in Euclidean space. Figure 4.1(b) shows an example of a 2D manifold embedding of a 20×20 grid, with a wall. The intersection of edges represents points on the grid, with the length of the edges representing the approximate distance between the points on the grid. Notice that embedding has made the points on either side of the wall to be farther apart than other points. The absence of a direct edge between these points captures the presence of the wall.

4.2 Manifold Learning

Manifold learning typically involves learning a low-dimensional representation, subject to distance preserving constraints. Manifold learning algorithms (Ma and Fu, 2012) rely on the true distances between neighboring states, while approximating other distances. Euclidean distance between points on the manifold can then serve as a distance metric for measuring similarity of states. Thus, the similarity measure needed in our NN-UCT algorithm, can be obtained by embedding the

states on a manifold.

4.2.1 Isomap

Isomap (Tenenbaum et al., 2000) is a popular manifold learning algorithm that is typically used on high dimensional data. The input is a distance matrix. The distance matrix is constructed in the following way: The distance between a point and any of its k nearest neighbors is equal to the Euclidean distance or the actual distance between them. The other pairwise distances are computed using the shortest paths between them. The rest of the steps is the same as Metric Dimensional Scaling (MDS), which is to compute an embedding that preserves the pairwise distances. Algorithm 3 shows the steps involved in learning a manifold from a transition graph.

Algorithm 3 Isomap algorithm

function ISOMAP(G) ▷ $G = \langle V, E \rangle$ is the state transition graph
  Construct distance matrix D from G such that
  for $u, v \in V$ **do**
    if $\langle u, v \rangle \in E$ **then**
      $D(u, v) = 1$
    else
      $D(u, v)$ = length of shortest path from $u \rightarrow v$
    end if
  end for
  $n = |V|$
  $H = I_n - \frac{1}{n} 1 1^T$ ▷ Construct Centering matrix
  $Dsq = \frac{1}{2} * H * D^2 * H^T$ ▷ Center the squared Distance Matrix
  Compute top k eigen-vectors of Dsq . Choose k by heuristic
  return Embedding as the top k eigen-vectors of Dsq
end function

The computational complexity of most manifold learning algorithms, including Isomap, is $\mathrm{O}(m^3)$, where m is the number of points needed to be embedded. This cost is due to the distance matrix construction and subsequent eigen-decomposition.

It is important to note that MDS (and hence Isomap) assumes that the distance matrix is symmetric. Such an assumption may not be true in certain domains (e.g when time elapsed is part of the state). Other manifold learning algorithms such as Graph Laplacian can be applied on directed graphs that do not impose this constraint. In this thesis, I used Isomap for illustrative purpose to show that manifolds

learnt from state transition graph of MDPs can be useful for planning.

It is also noteworthy to point out the criteria used for choosing k- the top eigenvectors for the embedding. One such approach is to threshold the eigenvalues corresponding to the eigenvectors. Another approach is to measure the reconstruction error. In this case, the reconstruction error corresponds to the error between reconstructing the pairwise distances on the manifold and the real distances in high dimensional space. However, this involves additional computation cost of $\mathrm{O}(m^2)$ for each dimension to threshold. In this thesis, I used the former approach of picking the eigenvectors by thresholding based on eigenvalues.

4.2.2 Local Manifolds

In my problem setting, each state is a point to be embedded, making cubic complexity infeasible for large problems. One way to deal with the high computational cost of embedding states on a manifold is to embed only those states that may be encountered in the near future. As we are operating in the planning setting, with access to a model, we can sample the states reachable in the near future and construct a manifold that is local to the current state.

In our work, at every decision step, we learn a local manifold of the deterministic MDP from the sample transitions encountered during a short breadth-first walk of the state space, starting from the current state. The state transition graph from the BFS walk is the input to the Isomap algorithm outlined in Algorithm 3. The output of Isomap is an embedding of the states in Euclidean space.

During UCT planning, we may encounter states that lie outside the local manifold. This situation is referred to as the out-of-sample problem and is more likely to occur as the UCT tree grows larger. If we were to embed these states on a manifold, the BFS walk needed from the start state would get deeper, resulting in a possibly exponential increase in the number of states to be embedded.

We address the out-of-sample problem in the following way. We generate the approximate embeddings of these states by applying a translation operator learnt for each action in the action set. An operator is a function that takes as input the manifold embedding of the current state and action, and outputs the manifold embedding (approximate) of the next state. For example, if x denotes the manifold embedding of a state s, and action k was taken from s to obtain state s' which was not encountered during the BFS walk, the operator for action k is given by:

$f(x,k) = x + b_k$, where b_k is a vector that denotes the average translation offset for action k. We learn this offset from the manifold embeddings of the states encountered during the BFS walk for the local manifold. Let T denote the set of tuples $\langle x, k, y \rangle$, such that y is the manifold embedding of the state obtained by taking action $k \in A$ from the state with manifold embedding x. We learn a simple translation offset (b_k) for this action by :

$$b_k = \frac{1}{|T|} \sum_{t \in T, t = \langle x,k,y \rangle} (y - x).$$

We found the simple translation operator to be useful in all the domains tested in Chapter 5. Applying more sophisticated solutions to solve the out-of-sample problem is a direction for future work.

Our use of local manifolds ensures good local coverage of the state space using the BFS walk and along with operators limits complexity by not embedding the entire space. In chapter 5, we evaluate the choice of using local manifolds over global manifolds on grid world domains and find them to offer similar benefits.

4.3 mNN-UCT

We combine all the steps to obtain NN-UCT with manifolds, referred to as mNN-UCT. This algorithm uses the manifold embedding as the state representation to compute the kernel function, needed in NN-UCT. During planning, for nodes whose states have already been encountered during the BFS walk, their manifold embedding is computed from the Isomap algorithm described above. If a state outside of the BFS walk is encountered, its manifold embedding is computed by applying the translation operator to the state from which the most recent action was taken. This enables the UCT tree to grow deeper than the BFS walk.

4.3.1 Discussion

While we show empirically (Chapter 5) that state generalization helps UCT, the improvement comes at an additional computational cost. In our algorithm, the cost is due to manifold learning and computing the nearest neighbor estimates. The cost from manifold learning is controlled by size of the BFS walk. While it would be useful to have a large size of the BFS walk to ensure a high coverage of the state space, the number of states to be embedded will increase exponentially

in the depth of the BFS tree. Recall from section 4.2.1 that the computational cost of most manifold learning algorithms is $\mathrm{O}(m^3)$, where m is the number of states to be embedded. Thus, increasing the size of the BFS walk results in a dramatic increase in the cost for manifold learning. Our choice of using local manifolds constructed from a small BFS tree, along with operators, does not limit UCT from growing deeper. Although operators only approximate the embeddings of states outside of the BFS walk, the error is likely to be small within the finite horizon of UCT planning. In all our experiments, the number of calls to the operator during planning totaled less than 20% of the planning steps. Thus, the approximation error due to the operator does not increase significantly.

The cost of computing the nearest neighbor estimate is controlled by the number of rollouts because each rollout adds one node. Experimental results (Chapter 5) show that mNN-UCT may need fewer rollouts than UCT to achieve good performance. This limits the cost of computing nearest neighbor estimates and may in itself justify the additional overhead.

4.4 Related Work

Learning manifolds in MDPs was previously done in proto-value functions (PVFs) (Mahadevan and Maggioni, 2007). Proto-value functions are used to learn a low dimensional representation from a transition graph generated by following a fixed policy which is subsequently improved using policy iteration. PVFs assume that a random walk of the state space is sufficient to construct basis functions that reflect the underlying topology. However, in many domains with large state space, such as video games, a policy consisting of random actions does not provide a sufficient view of the state space. For example, in the grid world with a wall, such a policy is unlikely to make the agent move close to the wall. Additionally, it may be computationally infeasible to embed all of the states encountered by following the policy. The high cost of manifold learning limits the number of states that can be embedded.

The idea of using local homomorphisms (Jiang et al., 2014) as an approximate abstraction was an inspiration for the development of using local manifolds. In their work, they construct a homomorphism of the MDP *locally*. They do so by finding a homomorphism from sample UCT trajectories. Since UCT trajectories

are limited up to a finite depth in practice, the homomorphism obtained is local to the current state. Nevertheless, they show empirically that UCT can benefit from these local homomorphisms. The homomorphism construction procedure relies on the policy followed (UCB in this case). But as we have shown earlier, UCT's policy is poor when rewards are sparse. Hence, as mentioned in section 3.3, homomorphisms offer no benefit over transposition tables in sparse reward domains.

Chapter 5

Experiments

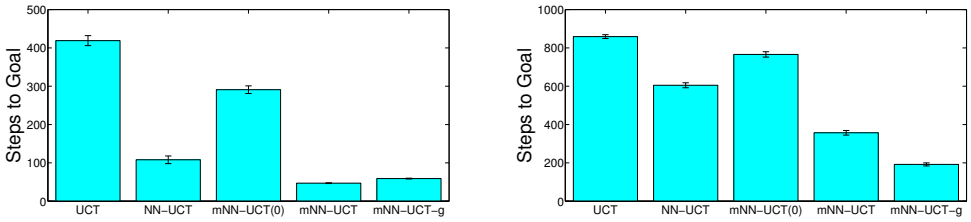
Now that we have described the algorithms, it is time to evaluate them in practice. Video games are a natural choice which can be modelled as sequential decision problems. We can evaluate the algorithms using the score obtained in these games. Additionally, the sparse reward scenario occurs naturally in some games. In the next section, we describe the setup and evaluation methodology.

5.1 Experiment Setup

As the focus of this thesis is on using UCT for planning, details of its implementation are described. In my UCT implementation, whenever a new node is created, the default policy (random actions) was run for a constant length of length 50, rather than out to a fixed horizon. This helped prevent finite horizon effects from affecting generalization. The optimal branch of the UCT tree and the BFS tree was retained across planning steps. In all of the experiments, a wide range $\{10^{-5}, 10^{-4}, .., 10^{3}, 10^{4}\}$ was tested for the UCB parameter with 500 trials performed for each parameter setting.

5.2 Grid World Domains

We first validate our intuitions about the impact of state generalization in UCT by evaluating the algorithms on the two grid world domains shown in Figures 3.1(a) and 4.1. An episode terminates when the agent reaches the goal state, or 1000 time-steps have elapsed, whichever is shorter. We report the number of steps taken to reach the goal state, given a fixed number of samples for planning. For the UCT algorithms with generalization, the Gaussian kernel function was used. The

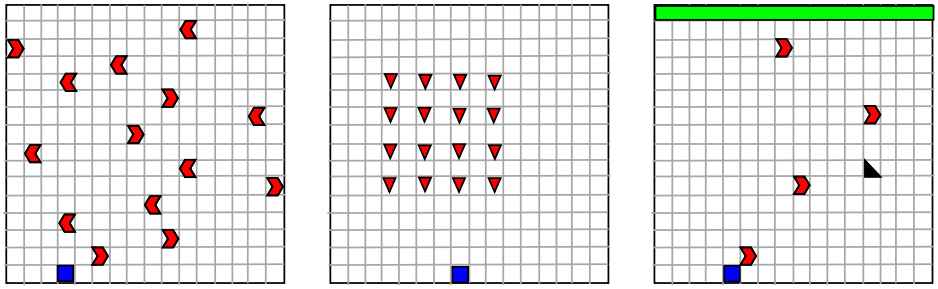


(a) Grid (b) Grid with Wall

Figure 5.1: Performance on Grid World Domains

initial Gaussian width σ, was chosen amongst $\{10, 100, 1000, 10000\}$ and the decay rate β amongst $\{0.1, 0.5, 0.9, 0.99\}$, and the best result is reported. The Euclidean distance was used for the distance metric in the kernel function. For the mNN-UCT algorithm, the BFS walk included at most 400 states for each planning step. The mNN-UCT algorithm with $\sigma = 10^{-6}$, referred to as mNN-UCT(0), effectively only shares statistics between nodes with the same state. This provides a difference over sharing values between nodes representing identical states. mNN-UCT-g is the mNN-UCT algorithm using a global instead of a local manifold, where a full BFS walk is used to construct the manifold. This allows us to evaluate the effectiveness of local manifolds.

Figure 5.1 summarizes the results of our grid world experiments. The mean number of steps (fewer steps being better), with standard error bars is shown. Sharing statistics between nodes with exactly the same states (mNN-UCT(0)) improves UCT slightly. Adding more widespread generalization to UCT improves the performance dramatically in both the domains, with the NN-UCT algorithm reaching the goal state in less than half the number of steps taken by UCT in the first domain. The best performing σ for the mNN-UCT and mNN-UCT-g algorithms was substantially greater than 0 (100), suggesting that generalizing over nearby states in the neighborhood of the manifold is useful. The NN-UCT algorithm, which uses the Euclidean distance metric between grid coordinates, does not perform as well in the second domain. The distance metric is not as useful since states on either side of the wall are farther apart than the Euclidean distance indicates. The mNN-UCT algorithm does not suffer from this problem as much, as it better captures the topology of the state space. In both the domains, using local manifolds rather than global manifolds does not substantially affect planning performance.



(a) Freeway (b) Space Invaders (c) Seaquest

Figure 5.2: Games Domains

5.3 Game Domains

We also test our algorithm on 3 domains inspired by video games. They are single player games played on a 16×16 pixel screen. The initial configuration of the games is shown in Figure 5.2. All the games last for 256 time-steps or until the agent loses all of its lives, whichever is shorter. In each domain the agent is depicted by the blue square.

1. Freeway : Cars are depicted by red chevrons. Their positions on each row and direction of movement is set randomly at the start. On reaching the end of the board, they reappear back on the opposite side. The agent has 3 lives. On collision, the agent loses a life and its position is reset to its start position. The agent has 3 actions: NO-OP (do nothing), UP and DOWN. The agent receives a reward of $+1$ on reaching the topmost row after which the agent's position is reset to the start state. The maximum possible score is 16 in the absence of cars, although with cars, the optimal score is likely lower. The complete state of the game is specified by the time elapsed, number of lives left, location of the agent, cars and their direction of movement

2. Space Invaders : The aliens are depicted by red triangles. These aliens fire bullets vertically downward, at fixed intervals of 4 time-steps. The aliens move from left to right. They change directions if the rightmost (or leftmost) alien reaches the end of the screen. The agent loses its only life if it gets hit by a bullet from the alien. The following actions are available to the agent : NO-OP, FIRE, LEFT, RIGHT, LEFT-FIRE (fire bullet and move left) and similarly RIGHT-FIRE (fire bullet and move right). The maximum possible score is 32 in the absence of the alien's bullets, with a realizable score likely lower. The

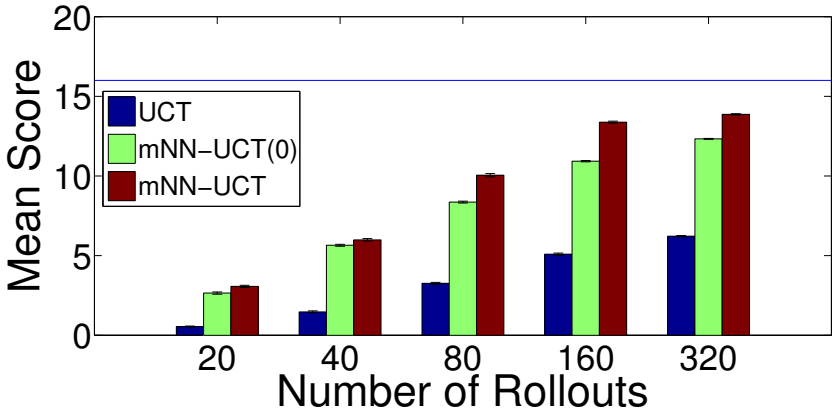
state of the game is specified by the time elapsed, location of the agent and its bullet, locations of the aliens and their bullets.

3. Seaquest : Fish depicted by red chevrons, move from left to right and reappear back on reaching the end of the screen. The diver, indicated by a black triangle, appears at either end, alternately. A diver appears when the current diver is collected by the agent or if the current diver reaches the end of the screen. The agent receives a reward of +1 if it collects the diver and drops it at the drop point (top row, indicated in green). The agent loses its only life on collision with a fish, or on reaching the drop point without the diver. The actions available to the agent are NO-OP, UP, DOWN, LEFT and RIGHT. The maximum possible score is 8, as it is impossible to capture any 2 successive divers and drop them at the drop point. The state of the game is completely specified by the time elapsed, location of the agent, fish and divers.

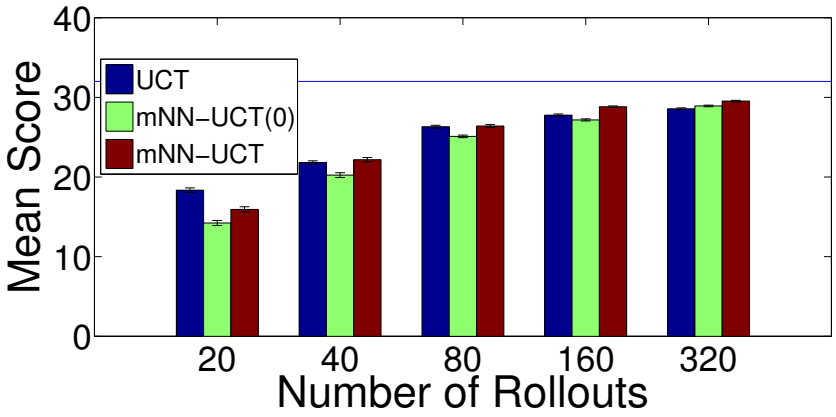
5.3.1 Performance Comparison.

We compare mNN-UCT with UCT on the domains described above. We again evaluate the effect of generalization by running our algorithm with $\sigma = 10^{-6}$, referred to as mNN-UCT(0). We ensure that all of the algorithms are given the same number of samples per planning step. We vary the number of rollouts from $\{20, 40, 80, 160, 320\}$. The discount factor was set at 0.99. We fix the other parameters for the mNN-UCT algorithm with $\sigma = 100$, and $\beta = 0.9$. We learn a local manifold for each planning step from a BFS tree comprising of at most 400 states. For each of the domains, we measure the mean score obtained per episode. The standard errors are indicated as error bars. The optimistic maximum score possible is indicated by a blue line in all the plots. Note that it may not be possible to achieve this score in all domains.

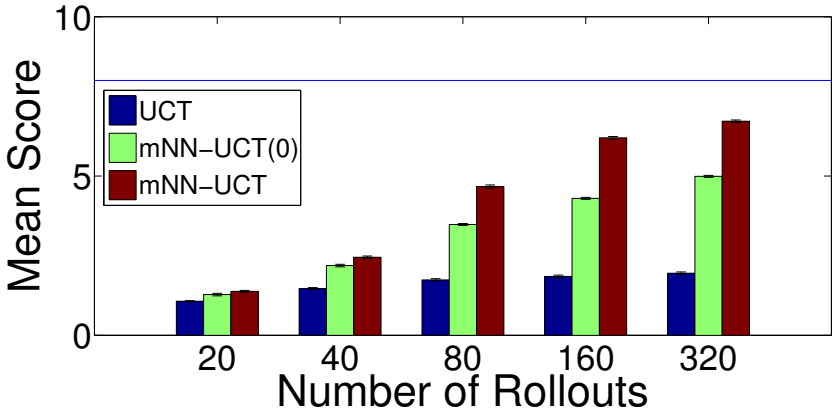
Figure 5.3 shows the performance of the 3 algorithms against the number of rollouts. The results are reported for the parameters which produced the best mean score obtained by performing 500 trials. mNN-UCT outperforms UCT, especially in Freeway and Seaquest. Both these games require the agent to perform a specific sequence of actions in order to obtain reward. UCT's rollout policy is sufficient to achieve a high score in Space Invaders as the game does not have sparse rewards to the same degree as the other domains. Generalization using manifolds provides only a marginal improvement in the Space Invaders game.



(a) Freeway

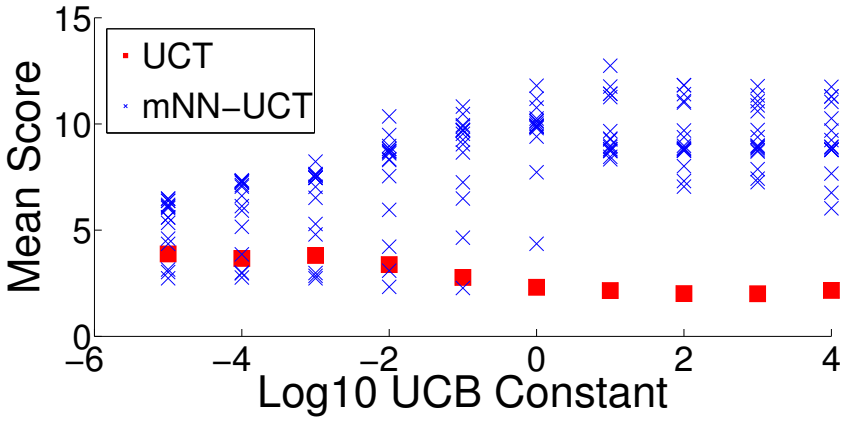


(b) Space Invaders

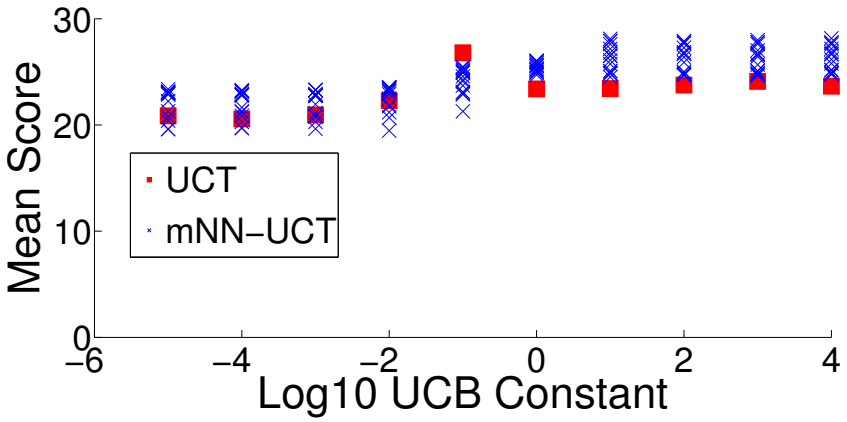


(c) Seaquest

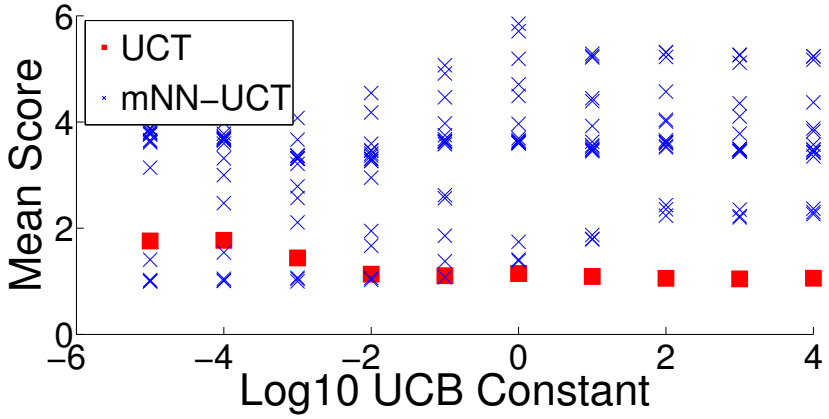
Figure 5.3: Performance comparison between mNN-UCT and UCT on Game Domains



(a) Freeway



(b) Space Invaders



(c) Seaquest

Figure 5.4: Parameter sensitivity of mNN-UCT algorithm on Game Domains

5.3.2 Parameter Sensitivity.

Compared to UCT, the mNN-UCT algorithm has additional parameters such as the depth of the BFS walk, and Isomap parameter (choosing the top k eigenvalues) and the gaussian kernel width (σ) and the rate of decay (β). Of these, σ and β together control the degree of generalization in the UCT algorithm. While an extensive sweep of all the parameters is desirable, the main objective of this thesis is to demonstrate the effectiveness of state generalization in UCT. Hence, we evaluate the sensitivity of the algorithm with respect to only these parameters (σ and β), keeping the rest fixed. Our choice for the rest of the parameters were set keeping in mind that they do not affect the objective. For example, the depth of the BFS walk was set so that it does not provide a full view of the state space, allowing the test of generalization possible along with the use of the algorithm in large state spaces. In this experiment, we fix the number of rollouts at 100. We vary σ from $\{10, 100, 1000, 10000\}$ and β in $\{0.1, 0.5.0.9, 0.99\}$. For each of the parameter settings, we measure the performance of the mNN-UCT algorithm and compare it to UCT using various settings of the UCB parameter.

Scatter plots are shown in Figure 5.4 where each blue x and red square represents the mean score per episode of the mNN-UCT and UCT algorithms respectively for a specific parameter setting. We see that the mNN-UCT algorithm performs better than UCT for a wide range of parameter settings in all the domains. Using widespread generalization achieves significantly higher score on sparse reward games such as Freeway and Seaquest, while the improvement is modest when rewards are easy to obtain as in the game Space Invaders.

5.4 Summary

State generalization helps when rewards are hard to obtain. As we have seen through the heat maps and experimental results, generalization helps UCT to spend its rollouts more efficiently by visiting states different from those it has already visited. On domains whose state space does not have a natural distance metric, we showed a method to learn a local manifold of the state transition graph to use Euclidean distances on the manifold as a distance metric. In addition, the manifolds learnt were found to be low dimensional (< 10 dimensions). This not only helps computing the kernel function faster but provides a simple feature space to distin-

guish states.

The experimental results demonstrate that manifolds act as a useful distance metric for measuring similarity between states. In addition, the algorithm is not very sensitive to the parameters controlling the degree of generalization in these domains.

Chapter 6

Conclusion

In this thesis, I presented a new algorithm that incorporates state generalization in UCT by using nearest neighbor estimates of the action-value and the visit counts in the tree policy. The primary motivation came from the need for efficient exploration in domains where rewards are sparse. UCT with no generalization performs poorly in such domains. In domains whose states do not have a natural distance metric I presented an algorithm that learns a local manifold of the state space that gives us a Euclidean representation of the states in the near future. I demonstrated the benefit of learning such local manifolds on video game domains. Empirically, we see that the algorithm substantially outperformed UCT with no generalization, especially in environments with sparse rewards.

Through this work, an interesting observation can be made. State generalization has largely been used to handle learning in large state spaces. For example in reinforcement learning (Sutton and Barto, 1998), value function approximation techniques are used to generalize to new unseen states. This work has shown that an additional benefit of state generalization is efficient exploration. Empirically, we have shown that incorporating state generalization results in significant performance improvement in sparse reward domains. This improvement was due to better exploration. Although Monte-Carlo methods incorporate exploration in their search process (e.g. confidence bounds in UCT), it is not sufficient when rewards are sparse. Thus, through this work, we have augmented UCT's exploration strategy by adding state generalization.

It is also useful to revisit the way state generalization was incorporated in UCT. The search tree was used not just for action selection, but also to learn a state representation of the states encountered during planning. The choice of using only a

fraction of the states for learning the representation enabled planning to continue unrestricted. Our choice of using a short BFS walk ensured complete coverage of the state space in the near future. It would be interesting to see if the trajectories of Monte-Carlo planning could be used to learn a state representation. The benefit of this approach would be that it would make the process of learning the representation sample efficient as samples would be shared for planning and learning the representation. As a result, there is an opportunity to learn a representation when the search tree grows deep. A challenge, however, would be to ensure the adequate coverage of the state space so that the representation captures the topology of the state space well.

It would also be interesting to see how this algorithm performs on games with extremely sparse rewards. For example, when the horizon of UCT planning is not long enough to see any reward, it would be interesting to see the actions chosen by the algorithm. Our intuition indicates that actions which leads to more unseen states would be picked, causing the agent to take an exploratory action in the real world. Whether such a behavior would result in the agent finally observing rewards, remains to be seen.

Bibliography

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of Monte Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.

Childs, B. E., Brodeur, J. H., and Kocsis, L. (2008). Transpositions and move groups in Monte Carlo tree search. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 389–395. IEEE.

Couetoux, A., Milone, M., Brendel, M., Doghmen, H., Sebag, M., Teytaud, O., et al. (2011). Continuous rapid action value estimates. In *The 3rd Asian Conference on Machine Learning (ACML2011)*, volume 20, pages 19–31.

Gelly, S. and Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875.

Jiang, N., Singh, S., and Lewis, R. (2014). Improving uct planning via approximate homomorphisms. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 1289–1296. International Foundation for Autonomous Agents and Multiagent Systems.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer.

Ma, Y. and Fu, Y. (2012). *Manifold learning theory and applications*. CRC Press.

Mahadevan, S. and Maggioni, M. (2007). Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8(2169-2231):16.

Méhat, J. and Cazenave, T. (2010). Combining uct and nested Monte Carlo search for single-player general game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):271–277.

Oudeyer, P.-Y., Kaplan, F., and Hafner, V. V. (2007). Intrinsic motivation systems for autonomous mental development. *Evolutionary Computation, IEEE Transactions on*, 11(2):265–286.

Puterman, M. L. (2009). *Markov decision processes: discrete stochastic dynamic programming*, volume 414. John Wiley & Sons.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT Press.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., and Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768. International Foundation for Autonomous Agents and Multiagent Systems.

Tenenbaum, J. B., De Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.