Action Schema Networks with Monte-Carlo Tree Search: The Best of Both Worlds

William Shen

A report submitted for the course
COMP3770 Individual Research Project
Supervised by: Felipe Trevizan, Sam Toyer, Sylvie Thiébaux, Lexing Xie
The Australian National University

October 2018

Except where otherwise indicated, this report is my own original work.

William Shen
26 October 2018

Acknowledgments

I would firstly like to express my sincere gratitude to my supervisors, Felipe Trevizan, Sam Toyer, Sylvie Thiébaux, and Lexing Xie. Their continuous guidance and support throughout the year has helped me appreciate and understand what research is all about. I would also like to express my gratitude to them for reviewing my report and providing continuous feedback.

I would especially like to thank Felipe and Sam for always taking the time to answer any additional questions I had in significant detail. I am now much more familiar with planning and am able to appreciate its complexities.

I am also grateful to the Australian National University for providing me with the opportunity to study at a world-class university under the National University Scholarship. I would also like to thank my friends in Australia and back home in New Zealand. Their friendship has helped me keep calm under the intense pressure I have been under, and will continue to be under until I graduate.

Finally, I would like to thank my family who have unceasingly supported my education. Without their encouragement, I would not be where I am today.

Abstract

Planning is the essential ability of an intelligent agent to solve the problem of choosing which action to take in an environment to achieve a certain goal. Planning is an extremely important field within Artificial Intelligence that has countless real-world applications, including robotics and operations research.

Monte-Carlo Tree Search (MCTS) is a state-space search algorithm for optimal decision making that relies on performing Monte-Carlo simulations to incrementally build a search tree, and estimate the values of each state. MCTS can often achieve state-of-the-art performance when combined with domain-specific knowledge. However, without this knowledge, MCTS requires a large number of simulations in order to obtain reliable estimates in the search tree.

The Action Schema Network (ASNets) [Toyer et al., 2018] is a very recent contribution in planning that uses deep learning and neural networks to learn generalized policies for planning problems. ASNets are well suited to problems where the "local knowledge of the environment can help to avoid certain traps". However, like most machine learning algorithms, an ASNet may fail to generalize to problems that it was not trained on. For example, this could be due to a poor choice of hyperparameters that lead to an undertrained or overtrained network.

This research project is concerned with investigating how we can improve upon the policy learned by an ASNet by combining it with MCTS. Our project has three key contributions. The first contribution is an ingredient-based framework for MCTS that allows us to specify different flavors of MCTS – including those which use the policy learned by an ASNet. Our second contribution is two new methods which allow us to use ASNets to perform simulations in MCTS, and hence directly affect the estimated values of states in the search tree. Our third and final contribution is two new methods for using ASNets in the selection phase of MCTS. This allows us to bias the navigation of the search space towards what an ASNet believes is promising.

Our empirical evaluation demonstrates that by combining MCTS with ASNets, we are able to 'learn' what the network did not learn during training, improve suboptimal learning, and be robust to changes in the environment or the domain. We can more reliably and effectively solve planning problems when combining MCTS with ASNets, and hence we achieve the best of both worlds.

Contents

Chapter 1

Introduction

Planning is the essential ability of an intelligent agent to solve the problem of choosing which action to take in an environment to achieve a certain goal [Geffner and Bonet, 2013]. In simplified terms, an agent needs to select a sequence of actions that bring us from the initial state to the goal state, ideally with the smallest cost possible.

Our research is concerned with combining the advantages of forward-chaining state space search through Monte-Carlo Tree Search (MCTS), with the domain-specific knowledge learned by Action Schema Networks (ASNets), a domain-independent learning algorithm. By combining MCTS and ASNets, we hope to more reliably and effectively solve planning problems, and hence achieve the best of both worlds.

1.1 Planning

We consider both classical and probabilistic planning problems. In classical planning problems, taking an action in a certain state can only lead to one outcome. On the other hand, in probabilistic planning problems, taking an action in a certain state can lead to one of many outcomes, each with a certain probability. The solution to a planning problem is a policy π, which tells us which action to take in a certain state. The optimal policy π^*, will take us to a goal state with the smallest possible cost.

Planning is an extremely important field within Artificial Intelligence that has countless real-world applications. For example, consider NASA's Mars Exploration Rovers (MERs) [Estlin et al., 2007]. Sending a radio signal to Mars takes approximately 14 minutes, so manually issuing commands to a rover and waiting for feedback is unfeasible. Thus, it is important for MERs to have some autonomous planning and scheduling capabilities, in order to navigate the Martian terrain and conduct scientific experiments.

1.2 Action Schema Networks

The Action Schema Network (ASNets) is a very recent contribution [Toyer et al., 2018] in planning that uses deep learning and neural networks to learn generalized policies for planning problems. A generalized policy is a policy that can be applied to any problem from a given planning domain. Ideally, this generalized policy is able to

reliably solve all problems in the given domain with a low cost, although this is not always feasible.

ASNets are well suited to problems where "local knowledge of the environment can help to avoid certain traps" [Toyer et al., 2018]. In such problems, an ASNet can significantly outperform traditional planners that use heuristic state space search. Moreover, a significant advantage of ASNets is that a network can be trained on a limited number of small problems, and generalize to problems of any size.

However, we cannot guarantee that an ASNet is always able reliably solve a planning problem. For example, an ASNet could fail to generalize to difficult problems that it was not trained on – an issue often encountered with machine learning algorithms. Moreover, the policy learned by an ASNet could be suboptimal, due to a poor choice of hyperparameters that has lead to an undertrained or overtrained network.

1.3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a state-space search algorithm for optimal decision making. MCTS relies on performing Monte-Carlo simulations to build a search tree and estimate the values of each state [Browne et al., 2012]. As we perform more and more of these simulations, the state estimates become more accurate. We discuss the MCTS algorithm and its variants in more depth in Section 2.3.

MCTS-based game-playing algorithms have often achieved state-of-the-art performance when paired with domain-specific knowledge. The most notable is AlphaGo [Silver et al., 2016], which infamously defeated one of the world's leading Go players Lee Se-dol in early 2016. AlphaGo enhanced MCTS by using a learned policy network as the rollout policy (which samples which actions to select) to perform simulations instead of Monte-Carlo sampling, and a learned value network to evaluate and assign value estimates to final states of simulations.

This is quite similar to what we want to achieve. Our 'policy network' is the generalized policy learned by an ASNet. We want to leverage this policy to encourage MCTS to perform search around the state space and actions an ASNet believes are promising.

However, it is important to note that ASNets is a domain-independent planning algorithm that learns domain-specific knowledge. The policy and value networks of AlphaGo have been trained specifically for learning how to play Go; hence AlphaGo is not a domain-independent game-playing algorithm.

Limitations of MCTS

One significant limitation of vanilla MCTS is that we may require a large number of simulations in order to obtain reliable estimates in the search tree. This means that it can take a long time to make a good decision, time which we do not have.

Another disadvantage of MCTS is that because simulations are random, the search may not be able to sense certain branches of the tree that will eventually lead to suboptimal outcomes, due to the random nature of how the state space is explored.

1.4 Contributions and Research Goals

The main goal of this research project is to investigate how we can improve upon the generalized policy of an ASNet by combining it with MCTS. As each has its own weaknesses, we hope that by combining both we will get the best of both worlds, and achieve more reliable results. The key contributions of this project are:

1. **An Ingredient-Based Framework for MCTS**. We introduce a framework for planning problems, extended from Trial-based Heuristic Tree Search (THTS) [Keller and Helmert, 2013], in which we can generate different flavors of MCTS by specifying a selection of ingredients. THTS is designed for solving finite horizon MDPs, while our framework solves Stochastic Shortest Path problems (SSPs) with dead ends - i.e. problems which have an indefinite horizon.

2. **Using ASNets as the rollout policy in MCTS**. We present two new rollout policies: Stochastic ASNets and Maximum ASNets. We can use these to perform simulations and improve the estimates of how promising states and actions are within the search tree. These estimated values are then used to guide the navigation of the state space.

3. **Using ASNets in the selection phase of MCTS**. We introduce two techniques, `Simple`-ASNet and `Ranked`-ASNet, that can be used to bias the selection of nodes within the explicit search tree towards those that an ASNet believes are promising. We demonstrate that these techniques are relatively robust to any misleading information provided by an ASNet, as they decay a network's influence as we apply what it has suggested more frequently.

It is worth noting that although we test our proposed methods using the generalized policies learned by an ASNet, our methods are applicable to **any** method of acquiring a generalized policy, including ROLLER [de la Rosa et al., 2011] and [Yoon et al., 2002].

1.5 Report Outline

The rest of the report follows the following structure:

- **Chapter 2 - Background and Related Work**. In this chapter, we will formalize planning as solving a Stochastic Shortest Path problem (SSP), and discuss existing algorithms used to solve planning including Monte-Carlo Tree Search (MCTS) and Upper Confidence Bounds applied to Trees (UCT). We will also describe Action Schema Networks (ASNets), a deep learning approach to planning using neural networks, which forms the basis of our research project. Finally, we give a brief discussion of related work that combines MCTS with domain-specific knowledge.

- **Chapter 3 - Combining Search with Action Schema Networks**. The main objective of this chapter is to build up a ingredient-based framework for UCT, and then discuss how we can include the knowledge that has been learned by an ASNet into the tree search. We will also outline the goals we hope to achieve by combining ASNets with search.
- **Chapter 4 - Empirical Evaluation**. We will firstly describe the configurations of UCT and ASNets we use, and then introduce the problems and domains that we evaluate our algorithms on. Next, we present the results of our experiments and discuss our findings in detail. We discover that UCT is able to account for the suboptimal learning of an ASNet. Moreover, an ASNet is able to improve plain UCT by biasing it to navigate promising parts of the search space.
- **Chapter 5 - Conclusion**. In the final chapter, we conclude the report by summarizing our contributions and discuss how we can further combine search with ASNets in future work.

Chapter 2

Background and Related Work

In this chapter, we firstly give a basic overview of planning (Section 2.1) in which we discuss the representation of planning problems, planning heuristics, and common algorithms used to solve planning problems. In Section 2.3, we will discuss Monte-Carlo Tree Search (MCTS), and variants of MCTS that have been adapted for planning.

In Section 2.4, we give a brief discussion of Action Schema Networks (ASNets) and its advantages and disadvantages over traditional planning algorithms based on heuristic search. Finally, we will consider related work that combines heuristic state space search with a generalized policy.

2.1 Planning

As we briefly mentioned in Section 1.1, outcomes of applying an action in a state are deterministic in classical planning, and stochastic in probabilistic planning. In this section, we will firstly discuss how we can model classical and probabilistic planning problems as Stochastic Shortest Path problems (SSPs) [Bertsekas and Tsitsiklis, 1991]. Finally, we briefly discuss popular heuristic search planning algorithms, and domain-independent planning heuristics.

2.1.1 Stochastic Shortest Path Problems

An SSP is a tuple $\langle S, s_0, G, A, P, C \rangle$ [Trevizan, 2013] where:

- S is the finite set of states
- $s_0 \in S$ is the initial state
- $G \subseteq S$ is the finite set of goal states
- A is the finite set of actions
- $P\colon S \times A \times S \to [0,1]$ is the transition function. $P(s' \mid a, s)$ represents the probability that we will transition to $s' \in S$ after applying action $a \in A$ in state $s \in S$.
- $C\colon S \times A \to (0, \infty)$ is the cost function. $C(s, a)$ is the immediate cost incurred when applying action $a \in A$ in state $s \in S$.

An agent's objective in an SSP is to perform a sequence of actions to move from the initial state s_0, to a goal state $s_g \in G$, with the lowest expected cost possible. At each step of the planning execution, an agent in state $s \in S$ executes an action $a \in A$, pays a cost of $C(s,a)$, and then transitions to the next state s' with a probability of $P(s' \mid a,s)$. For classical planning problems, $P(s' \mid a,s) \in \{0,1\}$.

A solution to an SSP is a policy $\pi\colon A \times S \to [0,1]$, where $\pi(a \mid s)$ represents the probability action $a \in A$ will be applied in state $s \in S$. Using this policy, we can define the state-value function $V^\pi\colon S \to [0,\infty)$ (Equation 2.1) and the action-value function $Q^\pi\colon S \times A \to [0,\infty)$ (Equation 2.2).

$$V^\pi(s) = \begin{cases} 0 & \text{if } s \in G \\ \sum_{a \in A} \pi(a \mid s) \cdot Q^\pi(s,a) & \text{otherwise} \end{cases} \tag{2.1}$$

$$Q^\pi(s,a) = C(s,a) + \sum_{s' \in S} P(s' \mid a,s) \cdot V^\pi(s') \tag{2.2}$$

Intuitively, $V^\pi(s)$ represents the expected cost to reach a goal state from state $s \in S$ under policy π. $Q^\pi(s,a)$ represents the expected cost to reach a goal state if we apply action $a \in A$ in state $s \in S$ and follow π thereafter.

An optimal policy π^*, is the policy that selects actions which minimize the expected cost of reaching a goal. For SSPs, there always exists an optimal policy that is deterministic (i.e. $\pi^*\colon S \to A$). This deterministic optimal policy can be obtained by finding the fixed-point of the state-value function V^* (Equation 2.3) known as the Bellman optimality equation [Bellman, 1954], and the action-value function as Q^* (Equation 2.4).

$$V^*(s) = \begin{cases} 0 & \text{if } s \in G \\ \min_{a \in A} Q^*(s,a) & \text{otherwise} \end{cases} \tag{2.3}$$

$$Q^*(s,a) = C(s,a) + \sum_{s' \in S} P(s' \mid a,s) \cdot V^*(s') \tag{2.4}$$

SSPs with Dead Ends

In many of the planning problems we consider, there are dead ends. A dead end represents any part of the state space in which there is no path to the goal.

A planning problem has an unavoidable dead end if the probability of reaching the goal is always less than 1 [Little and Thiébaux, 2007]. The representation of an SSP we presented above assumes that all dead ends are avoidable. One way we can handle dead ends in an SSP is by introducing a fixed dead-end penalty $D \in (0,\infty)$, and a *give-up* action [Kolobov et al., 2012].

This dead-end penalty represents the punishment incurred when we reach a state that is a dead end, and also acts as a limit to bound the maximum expected cost to reach a goal as D. Thus, if we cannot find a path to the goal with an expected cost less than D, then we can choose the *give-up* action and give up.

We may formally represent the new state-value function V^π using Equation 2.5. The action-value function Q^π remains unchanged from Equation 2.2.

$$V^\pi(s) = \begin{cases} 0 & \text{if } s \in G \\ D & \text{if } s \text{ is a dead end} \\ \min\{D, \sum_{a \in A} \pi(a \mid s) \cdot Q^\pi(s,a)\} & \text{otherwise} \end{cases} \tag{2.5}$$

2.1.2 Planning Example: Blocksworld

In planning, we generally split the description of a problem into a general domain and a concrete problem. The domain specifies the possible actions, and a problem specifies the initial state and the goal state. This is described in further detail in Section 2.4.1.

The Blocksworld domain is an example of an SSP in which the goal is to stack blocks on the table in a certain configuration using a robotic arm (see Figure 2.1).

There are only four possible actions in the original Blocksworld domain; all lead to deterministic outcomes.

1. *unstack*(x, y) - pick up block x from block y
2. *stack*(x, y) - stack block x on block y
3. *pick-up*(x) - pick block x up from the table
4. *put-down*(x) - put block x down on the table

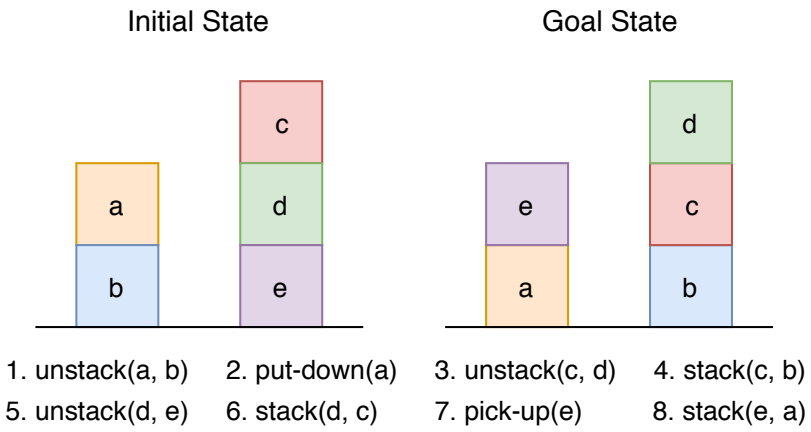

Figure 2.1: Example of a Blocksworld problem with its optimal plan

Figure 2.1 depicts an example of a concrete Blocksworld problem. The optimal plan for this problem requires us to execute eight actions. We choose Blocksworld as a running example because it is a simple and easily understood problem.

2.2 Planning as Heuristic Search

The majority of state-of-the-art probabilistic planning algorithms, such as LRTDP [Bonet and Geffner, 2003] and Anytime AO* [Bonet and Geffner, 2012], rely on heuristic search. Heuristic search planners construct a graph of state transitions, and choose the direction of the graph expansion by using a heuristic [Toyer, 2017].

2.2.1 Heuristics

A heuristic $h\colon S \to \mathsf{R}$, gives an estimate of the cost to reach the goal from a given state. By using a heuristic, heuristic search planners focus their search on promising parts of the state space.

A heuristic is said to be admissible if it always underestimates the true cost to reach the goal from a given state - i.e. $h(s) \leq V^*(s)$. The majority of heuristic search algorithms are only guaranteed to converge to the optimal solution if an admissible heuristic is used. However, admissible heuristics are often less informative than inadmissible ones [Haslum and Geffner, 2000]. Thus, heuristic search algorithms may converge to a solution (not necessarily optimal) much faster when using an inadmissible heuristic.

Most heuristics are derived through relaxing assumptions of the problem, so it is easier to solve. For example, consider the problem of navigating through the maze depicted in Figure 2.2, where the goal is to collect the seed in the bottom left-hand corner. One admissible heuristic we could use is the Manhattan distance, which calculates the distance between two points by summing the absolute differences of their Cartesian coordinates. The Manhattan distance relaxes the problem by assuming there are no obstacles in the maze, and thus underestimates the true cost of reaching a goal.

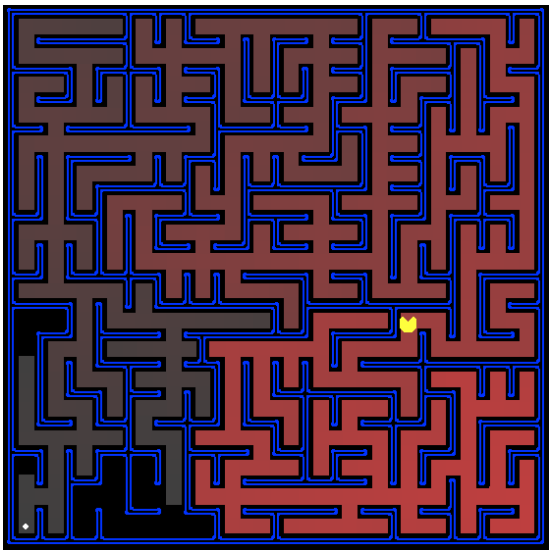

Figure 2.2: The Pacman Maze, from UC Berkeley's CS188 course page

In our research project, we will consider three domain-independent heuristics: h^{add} (inadmissible), h^{max} (admissible) [Haslum and Geffner, 2000], and the Landmark Cut heuristic (admissible) [Helmert and Domshlak, 2009]. All of these heuristics are calculated through delete relaxation, where, once a proposition is true, it is always true. The Landmark Cut (LM-Cut) heuristic is computed by summing the minimal cost of each set of action landmarks. This calculation requires computing the h^{max} heuristic for a series of relaxations of the original problem (one per set of action landmark). Thus, it can be considerably more expensive to compute than h^{max}, but generally gives much better estimates.

One potential disadvantage of these heuristics is that they were designed for deterministic planning problems. Despite the fact that we can relax a probabilistic problem into a deterministic one through determinisation, the calculation of the heuristics will ignore the true probabilities of outcomes, and will instead assume any outcome can be made true with a probability of 1 [Jimenez et al., 2006]. This can lead to very uninformative heuristic estimates, particularly for problems with many dead ends (e.g. Triangle Tireworld in Little and Thiébaux [2007]).

2.3 Monte-Carlo Tree Search

As we discussed in Section 1.3, Monte-Carlo Tree Search (MCTS) is a very simple state-space search algorithm that builds the search tree in an incremental manner by performing Monte-Carlo simulations.

The original version of MCTS approximates the true values of states and actions by performing a large number of random simulations. It then uses these estimated values to guide the navigation of the state space.

2.3.1 MCTS Algorithm

At each decision step, the MCTS algorithm incrementally builds the search tree by performing trials (also known as search iterations) until we reach some computational budget (e.g. time, memory) [Browne et al., 2012]. Each node in the search tree represents a state, and the edges between nodes represent actions.

Once the computational budget is reached, MCTS will return the action that gave the best estimated value. MCTS is an example of an **anytime optimal** algorithm [Dean and Boddy, 1988]. An anytime algorithm can return a non-random action even if it is interrupted. An anytime optimal algorithm will eventually return the optimal action given enough time.

In each trial, MCTS follows four phases (as depicted in Figure 2.3):

1. **Selection** - MCTS recursively selects nodes in the tree using a child selection policy until a leaf node in the explicit tree is encountered.
2. **Expansion** - one or more child nodes of the leaf node are created in the explicit search tree according to the available actions.

3. **Simulation** - a simulation of the game or scenario is run from one of the new child nodes to completion.

4. **Backpropagation** - the result of this simulation is backpropagated through the selected nodes in the tree to update their estimated values. The updated estimates affect the child selection policy in future trials.

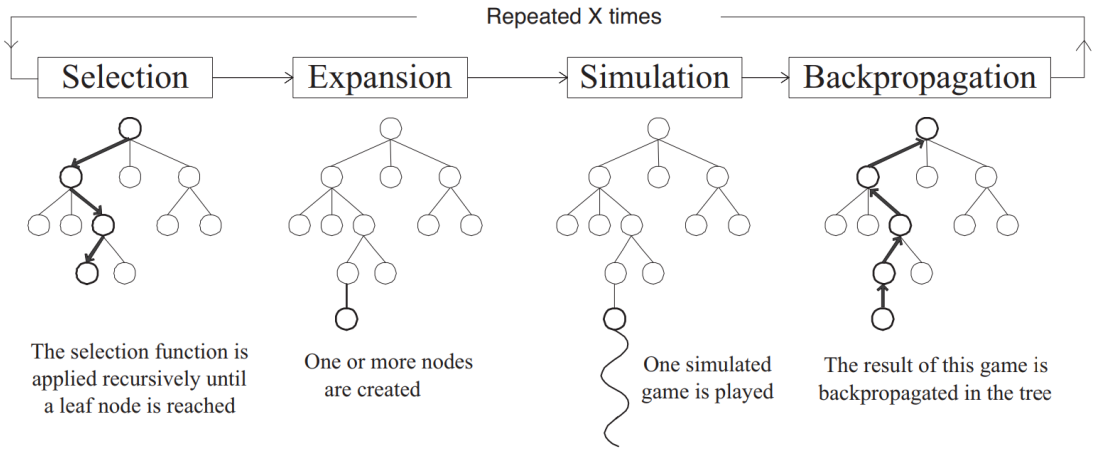

Figure 2.3: Phases of MCTS [Chaslot et al., 2008]

One major problem in MCTS is how we should choose the child selection policy such that it balances the trade-off between exploration and exploitation. That is, the exploitation of nodes it believes have optimal state estimates, and the exploration of nodes that currently have sub-optimal state estimates, but may turn out to be superior in the long term [Browne et al., 2012].

It must be noted that the full benefits of MCTS are not realized unless we adapt the basic algorithm for the problems we are tackling (planning in our case) [Domshlak and Feldman, 2013]. Thus, we will discuss MCTS-based algorithms for planning in Sections 2.3.3 and 2.3.4.

2.3.2 Upper Confidence Bounds applied to Trees

Upper Confidence Bounds applied to Trees (UCT) [Kocsis and Szepesvári, 2006] is a variation of MCTS that addresses the trade-off between exploration and exploitation in the selection phase by using the Upper Confidence Bound 1 term (UCB1).

UCT treats the choice of a child node as a multi-armed bandit problem, in which we must allocate our computational budget between the child nodes to minimize our expected cost (in the context of SSPs).

In UCT, the child node c of a parent node p that maximizes the UCB1 term (Equation 2.6) is selected:

$$UCB(p,c) = \underbrace{B \cdot \sqrt{\frac{\log n_p}{n_c}}}_{exploration} - \underbrace{V(c)}_{exploitation} \tag{2.6}$$

Where n_p represents the number of times the parent p has been visited, n_c represents the number of times the child c has been visited, and $V(c)$ represents the current state-value estimate for the child c. B is the bias term which allows to scale the trade-off between exploration and exploitation. B is commonly set to $\sqrt{2}$.

We can see that if a child node is visited, the contribution of the exploration term will decrease as both n_p and n_c increase by 1. However, if a different child node $c' \neq c$ is visited, then n_p increases by 1 and the exploration term increases for c. Thus, it is clear how UCT can balance the trade-off between exploration and exploitation.

Kocsis and Szepesvári [2006] proved that UCT will converge to the optimal policy given an infinite number of simulations. We will give a further description and analysis of the UCB1 term in Section 3.5.

Problems with Vanilla MCTS and UCT

Vanilla MCTS and UCT are domain-independent search algorithms that rely on performing a large number of random simulations in order to get good estimates of states and actions.

However, in planning we have domain-independent heuristics that estimate the cost of reaching a goal. These heuristics would give us much faster and hopefully more accurate estimates compared to those calculated by performing a large number of simulations. By using these heuristics in MCTS or UCT, we can more efficiently guide the search to more promising parts of the search space and perform more trials with the limited computational budget we are given.

Another problem with vanilla MCTS and UCT is that they do not consider the true transition probabilities $P\colon S \times A \times S \to [0,1]$ provided by an SSP when back-propagating information up the search tree. Instead, they estimate these probabilities from performing a large number of simulations. We will discuss this issue in much more detail in Section 3.3.2.

2.3.3 PROST

PROST is a domain-independent probabilistic planning algorithm based on UCT [Keller and Eyerich, 2012]. PROST introduces many improvements to UCT that have been tailored for probabilistic planning including Q-value initialization, search depth limitation, and the removal of superfluous actions.

It should be noted that PROST is used to solve finite horizon MDPs. Thus, some of the improvements such as reward locks as discussed by Keller and Eyerich, are not directly applicable to SSPs.

To represent search nodes, PROST adopts decision nodes to represent a state, and chance nodes to represent a state and an action. This allows us to support

probabilistic planning problems where there may be many outcomes from applying an action in a given state. We discuss this representation of search nodes in detail in Section 3.2.1.

Q-Value Initialization

We have highlighted the fact that vanilla UCT must perform an extremely large number of random simulations in order to converge to the optimal policy.

Q-value initialization helps us overcome the need for these simulations, as it assigns quality estimates to the unvisited children (chance nodes) of a decision node in the search tree [Keller and Eyerich, 2012]. By assigning these estimates, we roughly know which child chance node, and hence action is more promising in a given state.

PROST uses an iterative deepening search to estimate the Q-values, while we will use the domain-independent planning heuristics we previously introduced in Section 2.2.1.

Thus, Q-value initialization can significantly assist PROST in the navigation of the search tree, and will help it converge to the optimal policy much faster than vanilla UCT.

Search Depth Limitation

In planning problems, immediate decisions at the top of the search tree have a much larger influence on the expected cost of solving a problem than decisions in the future [Keller and Eyerich, 2012]. This is especially the case for SSPs where taking an inappropriate action could lead to a dead end with a very high probability.

On the other hand, this may not be the case in games such as Chess, where we may have to plan ahead for a certain move in the future, and hence may need to increase the search depth limit.

Thus, the search space near the root of the tree should be thoroughly explored for probabilistic planning problems. We can do this by limiting the search depth, which we will refer to as the trial length.

By limiting the trial length, we effectively do a limited lookahead search and as a result, can perform a much larger number of trials in the limited computational budget we are given. Ideally, this allows PROST to converge to the optimal policy in a smaller number of trials.

2.3.4 Trial-based Heuristic Tree Search

Trial-based Heuristic Tree Search (THTS) is an ingredient-based algorithmic framework that allows us to express MCTS, Dynamic Programming, and Heuristic Search based planning algorithms [Keller and Helmert, 2013]. We are mainly concerned with the MCTS and UCT algorithms that can be specified using THTS.

In a THTS algorithm, we must specify five ingredients: action selection, backup function, heuristic function, outcome selection and the trial length. We discuss these ingredients and a slightly modified version of the THTS algorithm in detail in Chapter

3. Note that like PROST, THTS is also aimed at solving reward-based finite horizon MDPs.

Using these ingredients, Keller and Helmert [2013] create three new algorithms, all of which provide superior theoretical properties over UCT: MaxUCT, DP-UCT and UCT*. MaxUCT combines Monte-Carlo backups with Full Bellman backups to build estimates based on the best partial solution graph. DP-UCT uses the probabilities provided by the transition function of the planning problem to directly backpropagate estimates by performing Bellman backups. Finally, UCT* is an extension to DP-UCT in which the trial length is set to 0, hence focusing the exploration of the state space on states closer to the root of the search tree. The advantages of these algorithms should become clearer after we discuss backup functions in Section 3.3.2.

Keller and Helmert found that UCT* outperformed both MaxUCT and DP-UCT because of its stronger theoretical properties. We discovered through our experiments that this was also the case for our planning problems.

2.4 Action Schema Networks

The Action Schema Network (ASNets) is a neural network architecture that exploits deep learning techniques in order to learn generalized policies for probabilistic planning problems [Toyer et al., 2018]. Recall, generalized policies are policies that can be applied to any problem in the given domain.

In this section, we will discuss of the intuition behind an ASNet, its architecture, and how a network is trained. We limit our discussion to be relatively brief, as the main goal of this research project is to use an existing policy learned by an ASNet.

2.4.1 Intuition and Architecture

What is an Action Schema?

In the Probabilistic Planning Domain Definition Language (PPDDL), which is commonly used to specify probabilistic planning problems, we must split the description of a problem into a general domain and a concrete problem [Younes and Littman, 2004]. The domain description can be considered a general template that allows us to describe concrete problems. In this domain description, we must specify the predicates, the **action schemas**, and the cost function [Toyer et al., 2018].

An action schema allows us to represent an action through the preconditions that must be satisfied before it can be executed, and the (probabilistic) effects of the action once it is executed. [Russell and Norvig, 2009]. An example of an action schema for the *unstack* action in Blocksworld is shown below.

```
(:action unstack
 :parameters (?x - block ?y - block)
 :precondition (and (on ?x ?y) (clear ?x) (handempty))
 :effect (and
```

```
    (holding ?x) (not (handempty))
    (clear ?y) (not (clear ?x))
    (not (on ?x ?y)))))
% Taken from the benchmarks provided by pyperplan
```

In the context of classical and probabilistic planning, predicates are boolean-valued functions that tell us whether certain statements are true. In the Blocksworld action schema given above, `(on ?x ?y)` and `(holding ?x)` are examples of predicates that tell us whether block x is on block y, and if the robotic arm is holding block x respectively.

When specifying a concrete problem in PPDDL, we list the objects in the problem, and the propositions (grounded predicates) that must hold in the initial state and in the goal state. Since we have propositions in a concrete problem, we can ground an action schema to get all the possible ground actions in the planning problem.

Network Structure

An ASNet consists of alternating action layers and proposition layers (Figure 2.4). The first and last layer of an ASNet are always action layers.

An action layer is composed of a single action module for each ground action. Similarly, a proposition layer is composed of a single proposition module for each ground proposition [Toyer et al., 2018]. An action module in one layer is connected to a proposition module in the next layer only if the ground proposition is directly related to the action. That is, if the ground proposition appears in the preconditions or effects of a ground action. Similarly, a proposition module in one layer is connected to an action module in the next layer only if the ground proposition appears in the preconditions or effects of the relevant ground action.

These sparse connections between the modules in the layers ensure that only the relevant action modules are connected to a proposition module, and vice-versa. This is vital for the weight sharing scheme, which we describe below. As such, we can see how ASNets can exploit the relational structure of planning problems.

Note, that the input to the first layer of an ASNet is a binary vector representing the truth-value of all propositions, while the output of each action module in the final layer represent the policy $\pi(a \mid s)$.

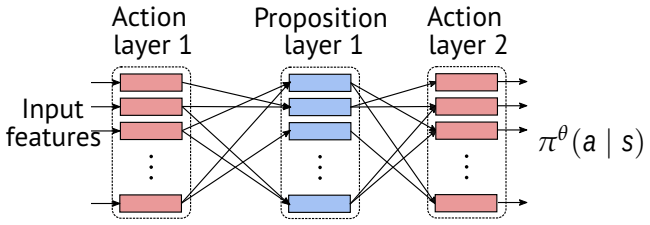

Figure 2.4: ASNet with 1 hidden layer [Toyer et al., 2018]

Weight Sharing

One major advantage of an ASNet is its weight sharing scheme, which "allows a network to be applied to any problem in a given planning domain" [Toyer et al., 2018].

Since all ground actions instantiated from the same action schema will have the same 'structure', we can use this structure to share the same set of weights between their corresponding action modules in a single action layer. Similarly, weights are shared between proposition modules in a single proposition layer that correspond to the same predicate. It is easy to see that by learning a set of common weights for each action schema and predicate, we can scale an ASNet to a problem of any size.

The weight sharing and network structure allow ASNets to achieve state-of-the-art performance on several planning problems, as it can learn knowledge of the environment from very small problems, and scale this knowledge up to problems of any size.

2.4.2 Heuristic Feature for Expressiveness

Since the action modules and proposition modules in an ASNet are sparsely connected, one limitation of an ASNet is its limited receptive field which may cause it to behave analogously to lookahead search. That is, an ASNet is unable to support long chains of reasoning.

One way Toyer et al. has suggested to overcome this problem is to supply the network with additional features obtained from a domain-independent planning heuristic such as LM-Cut (see Section 2.2.1). This allows an ASNet to 'see' beyond its fixed receptive field.

2.4.3 Training an ASNet

The weights θ of an ASNet are learned through a sophisticated training algorithm that alternates between guided exploration and supervised learning [Toyer et al., 2018].

In the guided exploration stage, we firstly sample trajectories from the policy π^θ an ASNet has learned so far, and from the policy envelopes for the optimal policy π^*, which has been calculated by a teacher such as LRTDP. We then store the states encountered along these trajectories in a state memory.

Next, in the supervised learning stage, we sample selections of states from the state memory into several mini-batches, and optimize the cross-entropy loss using the Adam optimizer, which updates θ in the direction that decreases the loss.

We continue training the network by alternating between the guided exploration and supervised learning stages until we have reached a pre-defined maximum number of iterations (epochs), a time limit, or have satisfied a performance-based early-stopping condition (e.g. 100% success rate of reaching the goal for a fixed number of iterations).

2.4.4 Pros and Cons of ASNets

One major advantage of ASNets is that it learns a generalized policy. Due to the weight sharing scheme, the set of weights θ learned by an ASNet can be applied to any problem from the domain the network was trained on. That is, even though a new network needs to be instantiated for each new problem, this network uses the learned weights θ

This means that an ASNet can be trained on a small number of problems, and evaluated on any number of different problems without the requirement to retrain the network. This is in comparison to traditional planning algorithms which must solve each problem separately.

As we have previously discussed in Section 1.2, ASNets are well-suited to problems where learning some knowledge of the local environment can help us avoid common traps. As such, we can train an ASNet to learn 'tricks' from problems with a small number of objects, and generalize and apply these 'tricks' to problems with any number of objects. An example of such a domain is Triangle Tireworld [Little and Thiébaux, 2007], which we discuss in Section 4.2.4.

However, like most machine learning algorithms, an ASNet could fail to generalize to new problems it was not trained on. Perhaps this could be because each problem has its own 'trick'. If this were the case, then an ASNet would not be able to model all of these 'tricks' due to its limited modelling capacity.

We could increase the number of hidden layers and units to remedy this problem, but finding a good choice of these hyperparameters is very difficult. Moreover, a poor choice of hyperparameters can lead to a sub-optimally trained network, or a network that has overfitted to the problems that it was trained on.

2.5 Related Work

Heuristic search algorithms are a well understood and researched field for solving planning problems. State-of-the-art probabilistic planning algorithms include Labeled Real Time Dynamic Programming (LRTDP) [Geffner and Bonet, 2013], Anytime AO* [Bonet and Geffner, 2012] and variants of UCT [Keller and Eyerich, 2012].

On the other hand, deep learning is still a very new technique for solving planning problems. Nevertheless, Groshev et al. [2018] use convolutional neural networks to learn generalized reactive policies for classical planning problems by representing them as images. However, this requires hand-engineered mappings from the description of the problem (usually in PDDL [Ghallab et al., 1998], the original deterministic version of PPDDL) to an image. Thus, their deep neural network planning pipeline is domain-dependent, unlike ASNets.

As far as we are aware, combining UCT with a generalized policy has not been done before. Moreover, our framework for combining UCT with ASNets is domain-independent. This represents a significant advantage because there is no need to hand-engineer features for each planning domain.

It is also important to note that our contributions are not limited to MCTS and UCT. Our contributions can be used in many other search algorithms. For example, we could use ASNets as a simulation function (described in Section 3.4) to guide the trajectories selected in LRTDP. Moreover, if a better algorithm for learning a generalized policy is invented, our contributions are still applicable as they are not coupled tightly to ASNets.

2.6 Summary

In this chapter, we firstly introduced classical and probabilistic planning as the problem of solving an SSP. We then gave a brief discussion of planning as heuristic search and described domain-independent planning heuristics.

In Section 2.2, we analyzed MCTS in detail and introduced UCT, which uses the UCB1 term to balance the trade-off between exploration and exploitation. We then discussed UCT in the context of planning, and described PROST and THTS.

Finally, we introduced ASNets and how its neural network architecture exploits the relational structure of planning problems. Moreover, we introduced the procedure used to train an ASNet, and discussed the advantages and disadvantages of ASNets. We finished this chapter by briefly exploring related work in the field of heuristic search and deep learning for planning.

In the next chapter, we will introduce our framework for combining UCT with ASNets.

Chapter 3

Combining Search with Action Schema Networks

ASNets are "well-suited to problems where the local knowledge of the environment can help avoid common traps" [Toyer et al., 2018]. That is, ASNets are effective when there is some trick that allows us to easily solve the problem and avoid any common traps. However, it is much more difficult for ASNets to generalize to solving problems that may not contain a certain trick, or may contain too many tricks (e.g. each problem in the training set has its own trick). Moreover, ASNets are not robust to changes in the domain. For example, an ASNet does not take the probability of non-deterministic actions into account after it has been trained, so the learned policy will not change if the probabilities within a domain are later altered.

In Section 2.3, we introduced Monte-Carlo Tree Search and UCT as domain-independent search algorithms for solving planning problems. A major disadvantage of vanilla MCTS-based search algorithms is that they do not employ any planning specific knowledge, such as using a heuristic, $h\colon S \to \mathsf{R}$, for calculating a state-value estimate, $V(s)$. Instead, vanilla MCTS-based searched algorithms rely on a large number of simulations (rollouts) to estimate this state-value. Of course, this can lead to increased search times, and an increased number of rollouts required for MCTS to find a solution, especially if random simulations are used.

PROST [Keller and Eyerich, 2012] and THTS [Keller and Helmert, 2013] attempt to circumvent these issues by using the Q-value initialization of unvisited search nodes. By doing so, they remove the strict requirement of doing simulations, as these Q-value estimates essentially tell us which action is more suitable, and hence should be considered at each decision point.

We develop a ingredient-based UCT framework, similar to THTS, that allows us to exploit the learned policy of an ASNet. By using the local knowledge of the environment, we hope to more efficiently navigate the state-space of the problem during search by biasing the simulations and action selection of UCT. Moreover, we hope to overcome the limitations of ASNets, as we will further discuss in Section 3.1.

We should keep in mind that ASNets is a domain independent planning algorithm despite the fact that an ASNet learns local knowledge of the environment. Thus, our pipeline of combining UCT with ASNets is also domain independent.

3.1 Goals

With the potential issues of plain UCT and ASNets in mind, we aim to achieve the following goals in combining UCT with ASNets.

3.1.1 Learn what we have not learned

As encountered in many machine learning algorithms, the capability of an ASNet to generalize beyond the distribution of problems it has encountered during training time can be very limited. An ASNet trained on a very specific problem within a domain could fail to generalize to new problems with slight modifications.

As a concrete example, consider Blocksworld (Section 2.1.2) and the task of stacking n blocks initially all on the table into a single tower, and the reverse (unstacking) as shown in Figure 3.1. We could train an ASNet to solve the relatively trivial task of unstacking the tower of n blocks and placing each block on the table. However, this ASNet would fail completely when evaluated on stacking the n blocks into a single tower, as it has never encountered the idea of stacking blocks on top of each other, only unstacking. In this scenario, the problems in the training set are clearly not representative of the problems the network is being evaluated on.

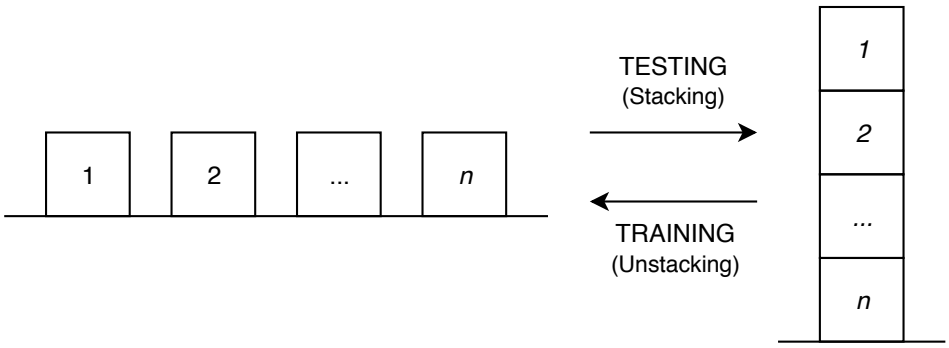

Figure 3.1: Stack and Unstack Blocksworld

We would expect an algorithm such as UCT to solve stacking blocks into a single tower relatively easily, given a sufficient number of simulations. Thus, we are interested in how we can combine UCT with ASNets to overcome the problem-specific bias ASNet has learned. Ideally, we hope UCT will be robust to any misleading information an ASNet provides, but also exploit an ASNet if the learned policy is informative.

3.1.2 Improve suboptimal learning

Neural networks can often be very difficult to train, especially due to the extremely large number of parameters that must be learned and the hyperparameters that can be tuned. A shallow ASNet with a limited number of hidden units could fail to learn very specific details and 'tricks' for solving a problem due to its limited feature representation.

Moreover, the loss while training an ASNet could fail to converge to a minima during training, due to the limited number of epochs, small learning rate, etc. In such a scenario, the network would be undertrained and could fail to reliably solve both the problems it was trained on and evaluated on.

On the other hand, we could overfit an ASNet on the training data we provide it, and hence overtrain the network. As a result, this ASNet could fail to generalize when evaluated on new, never-before-seen problems.

By combining UCT with what has already already been learned by an ASNet, we can more efficiently guide the search to what ASNets believes are the promising parts of the search space. Moreover, UCT should be relatively robust to any bad suggestions that ASNets makes, given that UCT balances the trade-off between exploration and exploitation.

3.1.3 Robust to changes in the environment or domain

We cannot always guarantee the problems that we evaluate on a trained ASNet are similar, or originate from the exact same environment that the network was trained on. Although an ASNet may have learned general information about a domain from the problems it was trained on, we may not expect it to perform well on problems from a slightly modified environment or domain.

Consider a dead-end version of the original Blocksworld domain, Exploding Blocksworld, in which blocks or the table could be exploded with certain probabilities when blocks are put down on top of them. A good policy to Exploding Blocksworld avoids exploding important objects - i.e. the blocks that form the goal state or the table. Exploding Blocksworld is discussed in detail in Section 4.2.2.

Consider an ASNet trained on Exploding Blocksworld, where the probability of exploding the table or a block when putting down a block is 0.1. This network would be more willing to take risks than an ASNet that has been trained with an exploding probability of 0.5, as the probability of exploding the table or an important block and entering a dead-end state and incurring the dead-end penalty is minimal. However, if we evaluated this trained ASNet on an environment where the exploding probability has been changed from 0.1 to 0.5, we would expect it to fail to reach a goal a majority of times as the network is not reactive to the changed probability. Clearly, we would expect the ASNet that was trained with an exploding probability of 0.5 to more reliably solve problems with an exploding probability of 0.5.

By combining UCT with ASNets in such a scenario, we hope to use the general information about the domain learnt by the ASNet to help guide the search and avoid the problems discussed above. However, we should also take care to control the influence of the network such that we do not just blindly follow what it has suggested.

3.2 General UCT Framework

We now describe an general ingredient-based framework in which we can combine different ingredients to generate different flavors of UCT. This framework is very similar to Trial-Based Heuristic Tree Search (THTS) [Keller and Helmert, 2013], apart from some changes to support Stochastic Shortest Path problems (SSPs) instead of finite horizon MDPs, and the introduction of dead-ends and the 'simulation function' to perform rollouts as described in vanilla MCTS. Note that our framework is focused solely on specifying UCT-based search algorithms.

Using our framework, we can specify the DP-UCT, Min-UCT and UCT* algorithms seen in THTS, as well as new flavours of UCT that make use of the learned policy of an ASNet. We describe the algorithms we can specify using our UCT framework in depth in Section 3.7.

3.2.1 Representation of Search Nodes

We adopt the same representation of alternating decision nodes and chance nodes in our search tree as used commonly in decision trees, and as seen in PROST and THTS.

A decision node n_d is a tuple $\langle s, N^k, V^k, \{n_1, \ldots n_m\}\rangle$, where $s \in S$ is the state, $N^k \in \mathsf{Z}^+$ is the number of visits to the node in the first k trials, $V^k \in \mathsf{R}$ is the state-value estimate based on the first k trials, and $\{n_1, \ldots n_m\}$ are the successor nodes (i.e. children) of n_d.

A chance node n_c is a tuple $\langle s, a, N^k, Q^k, \{n_1, \ldots n_m\}\rangle$. The only changes compared to a decision node are the $a \in A$, representing the action this chance node corresponds to, and Q^k, the action-value estimate based on the first k trials.

Assume, that we are able to access the value of attributes within tuples through their name - e.g. $s(n)$ represents the state of a node n, $V^k(n_d)$ represents the state-value estimate of a decision node n_d, $a(n_c)$ represents the action of a chance node n_c.

Let $S(n)$ represent the successor nodes $\{n_1, \ldots n_m\}$ of a search node n. Note that a node must be expanded for its successor nodes $S(n)$ to be initialized in the search tree. Clearly, it must be true that for every chance node n_c, $\sum_{n_d \in S(n_c)} P(s(n_d) \mid a(n_c), s(n_c)) = 1$.

Initially, our search tree contains a single decision node n_d with $s(n_d) = s_0$, representing the initial state of our problem. At each step of the planning execution process, the root node represents the current state of the planning problem.

Thus, a decision node n_d represents a state in which we must make a decision by selecting an action among the successor chance nodes, $S(n_d)$. A chance node n_c represents a state and an action in which the successor decision nodes, $S(n_c)$, represent the possible outcomes of applying the action $a(n_c)$ in the given state $s(n_c)$. For a classical planning problem, where all outcomes are deterministic, there can only be one child decision node for each chance node. In contrast, for a probabilistic planning problem, there can be any number of child decision nodes for each chance node.

3.2.2 Algorithm

UCT is described as an online planning algorithm, as it interleaves planning with execution. At each planning step, we give UCT a time constraint under which it continuously performs trials. We also introduce another constraint, the maximum number of trials per step, which takes priority over the time limit and allows UCT to complete the planning execution sooner when we are following a direct path to the goal. Once the time constraint or maximum number of trials per step is reached, we select the chance node n_c from the children of the root decision node that has the highest action-value estimate, $Q^k(s, a)$, and apply its action $a(n_c)$.

We provide the pseudocode of the algorithm we now describe in Algorithms 1 and 2. A trial in our framework can we specified under four distinct phases of MCTS (2.3.1): the Selection, Expansion, Simulation and Backup phase.

Selection Phase

As described in THTS, in the selection phase we traverse the explicit nodes in the search tree by alternating between **action selection** for decision nodes, and **outcome selection** for chance nodes until we reach an unexpanded decision node n_d. n_d is called the tip node of the trial.

Expansion Phase

In the expansion phase, we expand n_d and optionally initialize the Q-values of the child chance nodes, $S(n_d)$. This **Q-Value initialization** is optional as it can be computationally expensive. Calculating an estimated Q-Value requires calculating a weighted sum over the successor decision nodes for each chance node - i.e. $Q^k(n_c) = c(s(n_c), a(n_c)) + \sum_{n_d \in S(n_c)} P(s(n_d) \mid a(n_c), s(n_c)) \cdot V'(s(n_d))$, where $V'(s)$ is a state-value estimate for the state s. This state-value estimate could be taken from a planning **heuristic function**, such as h^{add}.

In Q-value initialization for the children of a decision node n_d, we calculate an estimate $Q(s(n_c), a(n_c))$ for each child chance node n_c and then back-propagate these Q-values to the parent node n_d. Next, we apply action selection to n_d to select a chance node n_c, and sample an outcome using the outcome selection to get the new tip node of the trial. We do this so we can immediately and effectively use the Q-values which were expensive to compute.

Simulation Phase

Now, in contrast to THTS algorithms, which would then continuously switch back between the selection and expansion phase until the trial length is reached, we transition to the simulation phase.

In the simulation phase, we perform a simulation (also known as playout and rollout) of the planning problem from the state $s(n_d)$ until we reach a terminal or dead-end state, or exceed the trial length. In vanilla MCTS algorithms, we do not

initialize the explicit search nodes in the tree that correspond to a playout. However, in our framework, we provide the option to create these **explicit nodes**. This allows our framework to model the algorithms seen in THTS, where explicit search nodes are created in its 'simulation phase' until the trial length is reached. However, creating explicit nodes requires much more memory and is not beneficial for SSPs in most situations as they are for finite horizon MDPs.

We use the **simulation function** to choose which action to take in a given state in the simulation phase and sample the next state according to the transition function $P(\cdot \mid a, s)$. If we have completed a simulation without reaching a goal or dead-end state, we add a heuristic estimate $h(s')$ to the rollout cost using the **heuristic function**, based on the final rollout state s'. If s' is a dead-end, then we set the rollout cost to be the dead end penalty D.

Following the steps above, if the trial length is set to 0, we do not perform any simulations and simply take a heuristic estimate for the tip node of the trial, or D if the tip node represents a dead-end.

Vanilla MCTS algorithms use a random simulation function, where each action has the same probability of being selected. Several simulations from a state s, essentially allow us to calculate a state-value estimate $V^k(s)$. However, calculating an accurate $V^k(s)$ using random simulations requires an extremely large number of simulations, so it is often more time-efficient to just assign a heuristic estimate $h(s)$. It should be noted that for some problems, such as CosaNostra Pizza as described in Section 4.2.3, the heuristic could be uninformative and hence misguide the search. In such a scenario, random simulations could give a better estimate of $V^k(s)$ than a heuristic h.

Backup Phase

After the simulation phase, we must propagate the information we have gained from the current trial back up in the search tree. We use the **backup function** to update the state-value estimate $V^k(n_d)$ for decision nodes and the action-value estimate $Q^k(n_c)$ for chance nodes. We do this by propagating the information we gained during the simulation in reverse order through the nodes in the trial path, by continuously applying the **backup function** for each node until we reach the root node of the search tree.

Summary

We have shown a framework under which many configurations of UCT can be implemented by specifying six key ingredients: action selection, outcome selection, simulation function, heuristic function, trial length, and backup function. We discuss these ingredients in depth in Section 3.3 and show the flavours of UCT that can be specified using these ingredients in Section 3.7.

3.2.3 UCT Framework Pseudocode

Algorithm 1 Performing trials and selecting action using UCT

```
1:  procedure Select-Action(rootNode, timeLimit)
2:      trialCount ← 0
3:      while ¬timeLimitReached(timeLimit) ∧ ¬maxTrialsReached(trialCount) do
4:          Do-Trial(rootNode)
5:          trialCount ← trialCount + 1
6:      bestChild ← arg max_{child∈rootNode.children} child.QValue
7:      return bestChild.action          ▷ Return action of child with highest Q-value

8:  procedure Do-Trial(rootNode)
9:      selectedNode ← rootNode
10:     ▷ 1. Selection phase
11:     while selectedNode.isExpanded() do
12:         if selectedNode is a DecisionNode then
13:             selectedNode ← Action-Selection(selectedNode)
14:         else if selectedNode is a ChanceNode then
15:             selectedNode ← Outcome-Selection(selectedNode)
16:
17:     ▷ 2. Expansion phase
18:     Expand-Node(selectedNode)
19:     if initialiseQValues then
20:         Initialize-Q-Values(selectedNode)
21:         selectedNode ← Action-Selection(selectedNode)
22:         selectedNode ← Outcome-Selection(selectedNode)
23:         Expand-Node(selectedNode)
24:         ▷ Immediately select 'best' ChanceNode, sample outcome and expand.
25:
26:     ▷ 3. Simulation Phase
27:     futureCost = Simulate(selectedNode)
28:
29:     ▷ 4. Backup Phase
30:     for node ∈ Reversed(trialPath) do
31:         Backup-Function(node, futureCost)
```

Pseudocode continued on next page.

Algorithm 2 Performing trials and selecting action using UCT (continued)

```
32: procedure Initialize-Q-Values(parentNode)
33:     for child ∈ parentNode.children do
34:         Expand-Node(child)                          ▷ Expand node to get successors
35:         child.QValue ← Q-Value(child.state, child.action)
36:     Backup-Function(selectedNode)                   ▷ Propagate Q-values to parent

37: procedure Simulate(rolloutNode)
38:     rolloutState ← rolloutNode.state
39:     rolloutCost ← 0
40:     rolloutLength ← 0
41:     ▷ Loop while trial length is not exceeded, and not in terminal/dead-end state
42:     while ¬Simulation-Stop-Criterion(rolloutState, rolloutLength) do
43:         action ← Simulation-Function(rolloutState)
44:         rolloutState, stepCost ← Sample-Outcome(rolloutState, action)
45:         rolloutCost ← rolloutCost + stepCost
46:         rolloutLength ← rolloutLength + 1
47:
48:     ▷ Handle final rollout state
49:     if Is-Goal(rolloutState) then
50:         return rolloutCost
51:     else if Is-Dead-End(rolloutState) then
52:         return Dead-End-Penalty
53:     else if Is-Terminal-State(rolloutState) then
54:         ▷ SSPs formally do not have terminal states, so this should never happen.
55:     else
56:         ▷ Trial length exceeded, add heuristic estimate
57:         return rolloutCost + Heuristic-Function(rolloutState)
```

3.3 Ingredients

Now, we introduce and describe the selections for the ingredients required to specify common UCT flavors. We will discuss how to incorporate ASNets into these ingredients later in Sections 3.4 and 3.5.

3.3.1 Action Selection

Vanilla UCT [Kocsis and Szepesvári, 2006] balances exploration between nodes that have not been visited frequently, and exploitation of nodes that are known to be good by choosing the action that maximizes the Upper Confidence Bound (UCB1) formula during action selection (Equation 3.1). In the action selection step, UCT selects the action of the child node of the decision node that maximizes the UCB1 term - i.e. $\mathit{arg\,max}_{n_c \in S(n_d)} \text{UCB1}(n_d, n_c)$.

$$\text{UCB1}(n_d, n_c) = \underbrace{B \cdot \sqrt{\frac{\log C^k(n_d)}{C^k(n_c)}}}_{\mathit{exploration}} - \underbrace{Q^k(n_c)}_{\mathit{exploitation}} \tag{3.1}$$

We set $\text{UCB1}(n_d, n_c) = \infty$ if $C^k(n_c) = 0$ to force the exploration of chance nodes that have not been visited before. We call B the bias term which can be adjusted to adjust the trade-off between exploration and exploitation. Clearly, lower values of B value exploitation. Conversely, higher values of B value exploitation.

UCB1 is a formula used to minimize regret in a multi-armed bandit problems [Auer et al., 2002], and is said to implement optimism in the face of uncertainty. As shown by the exploration component, the confidence bound grows logarithmically with the total number of visits to a decision node, and shrinks with the number of visits to a chance node (and hence the number of times the action $a(n_c)$ has been tried). Thus, UCB1 is able to maintain a balance between exploration and exploitation.

One can also non-trivially prove that UCB1 never stops exploring [Kocsis and Szepesvári, 2006]. Essentially, for a decision node n_d, selecting a child chance node n_c with action $a(n_c)$ will decrease the exploration term for n_c, but will increase the exploration term for all other child chance nodes $S(n_d) \setminus n_c$ [Gusmão and Raiko, 2012].

3.3.2 Backup Function

The backup function is used to propagate the information we gained during a trial up the search tree by updating the state-value estimates $V^k(n_d)$ and the action-value estimates $Q^k(n_c)$. Here, we will outline the backup functions as discussed in THTS, which can be used to specify multiple flavors of UCT.

In our discussion, we only consider partial backup functions. A partial backup function on a chance node only requires one of its children to be explicated and visited (hence implying there is a state-value estimate for that child) in the tree. On the other hand, full backup functions require all children of a chance node to be explicated

and visited. For us to consider full backup functions would require all children of a chance node to be initialized with some heuristic estimate. This would require state-value initialization for all decision nodes which would incur large computation overheads.

Let D represent the fixed dead-end penalty. Recall, a dead end is any state from which a goal state can no longer be reached. Let $P(n_d \mid n_c) = P(s(n_d) \mid a(n_c), s(n_c))$, representing the probability of transitioning to $s(n_d)$ from $s(n_c)$ after applying action $a(n_c)$. Let $c(n_c) = c(s(n_c), a(n_c))$, representing the cost of applying $a(n_c)$ in state $s(n_d)$.

Monte-Carlo Backups

Monte-Carlo backups [Kocsis and Szepesvári, 2006] take a weighted average over the state-values and action-values of the children of a search node, based on the number of visits to a child node. Child nodes with a high number of visits are given more weighting in this sum.

Thus, using Monte-Carlo backups to backup information at an end of a trial simply extends the current average with the latest sampled value [Keller and Helmert, 2013].

$$V^k(n_d) = \begin{cases} 0 & \text{if } s(n_d) \text{ is a goal} \\ D & \text{if } s(n_d) \text{ is a dead end} \\ \min\left\{D, \frac{\sum_{n_c \in S(n_d)} C^k(n_c) \cdot Q^k(n_c)}{C^k(n_d)}\right\} & \text{otherwise} \end{cases} \tag{3.2}$$

$$Q^k(n_c) = c(n_c) + \frac{\sum_{n_d \in S(n_c)} C^k(n_d) \cdot V^k(n_d)}{C^k(n_c)}$$

A major disadvantage of using Monte-Carlo backups is that it does not consider the provided probabilities of outcomes of a chance node $P(\cdot \mid a(n_c), s(n_c)$ that are known a priori. Instead, Monte-Carlo backups try to 'learn' this probability, as $C^k(n_d)/C^k(n_c) \to P(n_d \mid n_c)$ as $k \to \infty$ [Keller and Helmert, 2013]. In most practical situations, learning a good estimate of $P(n_d \mid n_c)$ can take hundreds of thousands or even millions of simulations. Moreover, since the state-value estimate for a decision node n_d is also calculated using a weighted average over all child nodes, an action $a(n_c)$ in a chance node n_c that leads to a very high cost (and hence should not be chosen) can bias the updated $V^k(n_d)$ disproportionately if there has only been a small number of simulations.

Consider a sub-tree of a search tree, as represented in Figure 3.2. Taking action a_1 leads directly to a goal while taking action a_2 leads directly to a dead end with a fixed dead-end penalty of D. Obviously, we would never choose a_2 in such a scenario. However, if both leaf decision nodes were visited only once, then $V^k(n_d) = \frac{1 \cdot 0 + 1 \cdot D}{2} = \frac{D}{2}$ for the root decision node in the sub-tree, n_d. Clearly, by using Monte-Carlo backups, we can disproportionately bias action selection in future rollouts due to the high perceived cost of n_d.

Given an action selection ingredient, such as UCB1, that never stops exploring, we can prove that Monte-Carlo backups will eventually converge to the optimal state-values and action-values, as $\lim_{k\to\infty} \frac{C^k(n_c^*)}{C^k(n_d)} \to 1$ [Keller and Helmert, 2013], where n_c^* represents the chance node with the optimal action $\pi^*(s)$. This proof, however, is not trivial [Kocsis and Szepesvári, 2006].

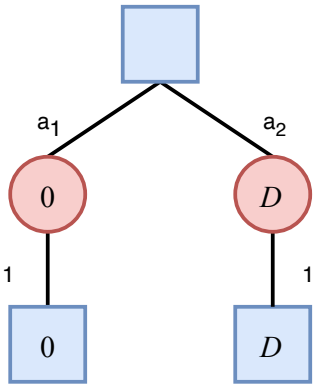

Figure 3.2: Subtree where Monte-Carlo Backups are not ideal

Minimum Monte-Carlo Backups

Minimum Monte-Carlo backups (called Maximum Monte-Carlo in Keller and Helmert [2013] as they deal with rewards, not costs) solves the issue of disproportionately biasing $V^k(n_d)$ that we encounter in plain Monte-Carlo backups by simply selecting the action-value of the child chance node n_c that gives the lowest action-value $Q^k(n_c)$. However, the probabilities of outcomes of a chance node are still being estimated instead of using their known values in the backups. We can solve this issue using Bellman Backups, which we describe next.

$$V^k(n_d) = \begin{cases} 0 & \text{if } s(n_d) \text{ is a goal} \\ D & \text{if } s(n_d) \text{ is a dead end} \\ \min\left\{D, \min_{n_c \in S(n_d)} Q^k(n_c)\right\} & \text{otherwise} \end{cases} \tag{3.3}$$

$$Q^k(n_c) = c(n_c) + \frac{\sum_{n_d \in S(n_c)} C^k(n_d) \cdot V^k(n_d)}{C^k(n_c)}$$

Keller and Helmert [2013] have proven that Minimum Monte-Carlo backups will converge to the optimal state-values and action-values, using the same reasoning as for plain Monte-Carlo backups.

Bellman Backups

Bellman backups [Keller and Helmert, 2013] take probabilities of outcomes into consideration when backing up action-value estimates. When updating the action-value

estimate $Q^k(n_c)$ for a chance node n_c, a weighted sum is taken over the state-value estimates $V^k(n_d)$ for each child decision node n_d based on the probability $P(n_d \mid n_c)$. Clearly, the state-value estimate $V^k(n_d)$ of a child decision node is given a higher weighting in the sum if the probability $P(n_d \mid n_c)$ is higher.

For updating state-value estimates, the action-value of the best child chance node is selected, as we have already encountered in Minimum Monte-Carlo backups.

$$V^k(n_d) = \begin{cases} 0 & \text{if } s(n_d) \text{ is a goal} \\ D & \text{if } s(n_d) \text{ is a dead end} \\ \min\left\{D, \min_{n_c \in S(n_d)} Q^k(n_c)\right\} & \text{otherwise} \end{cases} \tag{3.4}$$

$$Q^k(n_c) = c(n_c) + \frac{\sum_{n_d \in S(n_c)} P(n_d \mid n_c) \cdot V^k(n_d)}{P^k(n_c)}$$

Here, $P^k(n_c) = \sum_{n_d \in S(n_c)} P(n_d \mid n_c)$, representing the sum of the probabilities of all child decision nodes of n_d that have been explicated and visited. Full Bellman Backups assume all child nodes are explicated and initialized, and thus effectively disregards the normalizing term $P^k(n_c)$ in Equation 3.4 as $P^k(n_c) = 1$.

Bellman backups are derived directly from the Bellman optimality equation, as we have already encountered in Equation 2.3. Thus, a flavor of UCT using Bellman backups is anytime optimal given a correct selection of ingredients that will ensure the whole search tree is eventually explored.

3.3.3 Heuristic Function

As described in Section 2.2.1, heuristics for planning are domain-independent. A heuristic $h\colon S \to \mathsf{R}$, gives us an estimate of how much it costs to reach a goal from the given state s.

In this report, we consider the following heuristics: *zero*, h^{add}, h^{max}, *LM-Cut* (see Section 2.2.1 for details). *zero* is the trivial heuristic which simply maps all states to an estimate of 0.

When choosing a heuristic function, we must keep the cost of computing the heuristic in-mind, and whether it is admissible or not. Computing LM-Cuts is expensive and hence we would expect the number of playouts that can be performed in a second to be significantly less than if h^{add} was used.

As described in Section 3.2.2, we use the heuristic in Q-Value initialization by taking a weighted sum over successor decision nodes, and after the simulation phase as an estimate of the remaining cost after the playout (if any) to reach a goal. That is, once a simulation has been completed, we add a heuristic estimate to the cost of the rollout and then use the backup function to propagate these values up the search tree. By doing so, we have an estimate of the cost of reaching a goal node from the tip node of a trial.

Unfortunately, the heuristic is not always a good estimate for the cost of reaching the goal from a given state and can sometimes be misleading. Heuristics based on

the delete relaxation, such as h^{add}, assume that once a proposition is true, it is always true. This assumption can lead to uninformative heuristic values as we encountered in the CosaNostra Pizza experiment, which will be discussed in Section 4.2.3.

3.3.4 Outcome Selection

Outcome selection deals with selecting the outcome after applying an action. In terms of our search node representation, outcome selection deals with choosing a decision node among the children of a chance node. We only support Monte-Carlo sampling in our framework, in which we randomly sample child decision nodes based on their probabilities $P(n_d \mid n_c)$.

Although we only provide one selection for this ingredient, we maintain it in our UCT framework for future work. As Keller and Helmert mention, the only search based algorithm that does not use Monte-Carlo sampling is Anytime AO* [Bonet and Geffner, 2012]. Instead, Anytime AO* deals with AND/OR graphs, and chooses the outcome that has the 'biggest potential impact' in the partial graph.

3.3.5 Simulation Function

As we describe in the general UCT algorithm presented in 3.2.2, the simulation function is used to choose which action to take in a given state in the simulation phase where we perform playouts.

Several simulations with a decision node n_d in the trial path, effectively act as an indicator to how promising a search node is, and hence represent an estimate for $V^k(n_d)$ once the information gained from each simulation is backpropagated through the search tree.

Traditional MCTS-based algorithms use a random simulation function, in which each available action in the state has the same probability of being selected. However, this is not very suitable for SSPs as we can continuously loop around a set of states and never reach a goal state. Moreover, using a random simulation function requires an extremely large number of simulations to obtain a good estimate for state-values and action-values within the search tree.

Because of this, the simulation phase in MCTS-based algorithms for planning is often neglected and replaced by a heuristic estimate. In our UCT framework, this is equivalent to setting the trial length to be 0. In this case, we finish a trial once we expand the tip node of the trial, calculate a heuristic estimate for the state of the tip node, and backup that estimate.

However, there can be situations where the heuristic function is uninformative and thus misguides the search. In such a scenario, it may be more productive to use a random simulation function to calculate estimates rather than using the heuristic.

The simulations would be much more effective if we had a simulation function that uses domain-specific knowledge to guide the search. Thus, we may include ASNets in our simulation functions, and exploit the knowledge of the local environment that

an ASNet has learned. We discuss using ASNets as a simulation function in Section 3.4.

For the sake of posterity and the ability to model the algorithms presented in THTS, we also provide a heuristic-guided simulation function. This simulation function selects the action with the highest $Q(s, a)$, as calculated by the **heuristic function**, at each step in the simulation.

Choosing a Backup Function when using a Simulation Function

Ideally, Bellman backups should not be used with a simulation function unless we can guarantee that the simulations will always provide a good estimate of the state-values and action-values within the search tree. To understand why this is the case, let us consider random simulations.

A single random simulation from a decision node n_d clearly does not give a good estimate of the state-value $V^k(n_d)$. However, as we do more and more of these simulations and backpropagate the results of these simulations, $V^k(n_d)$ will slowly converge to the optimal state-value.

Thus, when performing a backup on a chance node n_c, we should weight the sum by the number of times a child decision node n_d has been visited $C^k(n_d)$, not its transition probability $P(n_d \mid n_c)$. This is especially true when using random simulations, as the simulations are highly susceptible to random noise.

Therefore, when using a simulation function that is not very informative and does not use any knowledge of the problem or the environment, we should consider using either Monte-Carlo or Min-Monte-Carlo backups.

3.3.6 Trial Length

Given that we are dealing with planning problems which may have loops and dead ends, this means that trials or simulations could take infinite steps to reach a goal. Thus, we must introduce the trial length, which bounds how many steps can be applied in the simulation phase.

Moreover, the trial length allows us to adjust the lookahead capability of a UCT algorithm. By setting the trial length to be very small, we can focus the search on nodes closer to the root node, much like breadth-first search [Keller and Helmert, 2013]. For our experiments, we found that setting the trial length to be 0, and hence not performing any simulations gave the best results for domains where the heuristic function is informative, and hence provides a good estimate of the cost to reach the goal from the current state.

3.4 Using ASNets as a Simulation Function

An ASNet learns a policy $\pi\colon A \times S \to [0,1]$, where $\pi(a \mid s)$ represents the probability action a is applied in state s. Using this policy, we can bias the simulation function to navigate parts of the search space that the trained ASNet believes is promising.

We introduce two simulation functions that make use of a trained ASNet: Stochastic ASNets and Maximum ASNets. As their names suggest, Stochastic ASNets simply samples from the probability distribution given by the policy to select an action at each simulation step, while Maximum ASNets selects the action a with the highest probability - i.e. $a \leftarrow \mathit{arg\,max}_{a \in A(s)} \pi(a \mid s)$.

Using domain-specific knowledge in the simulation phase of MCTS is often called using heavy playouts, while using a random simulation function is known as soft playouts. Heavy playouts are called 'heavy' because they often require some computation overhead. In our case, this is evaluating the current state using an ASNet to get $\pi(s)$. Ideally, heavy playouts will focus the search process by using domain-specific knowledge.

It is important to note that although an ASNet has learned domain-specific knowledge, ASNets is a domain-independent planning algorithm.

When is it worth using ASNets as a simulation function?

Given that an ASNet has learned some useful features or tricks about the environment or domain of the problem that we are trying to solve, we ideally should expect an ASNet-based simulation function to guide the search process in UCT much better than if we were to use a random simulation function.

In turn, this will lead to a smaller number of simulations required to obtain good state-value and action-value estimates in the search tree. Thus, using UCT with ASNets as a simulation function can help the search converge to the optimal policy in a much smaller number of trials than if random simulations were used, despite the increased computational cost of evaluating the learned policy of an ASNet.

Moreover, as UCT balances the trade-off between exploration and exploitation through UCB1, our algorithm will still perform search and can determine whether the trial paths an ASNet has suggested are ideal. That is, by combining UCT with ASNets as a simulation function, we are able to improve suboptimal learning and stay robust to any changes in the environment.

When is it not worth using ASNets as a simulation function?

Using ASNets as a simulation function may not be very robust if the learned policy is misleading and hence uninformative. Robustness indicates how well UCT can overcome the bad information provided by an ASNet within the time limit we give UCT at each planning step.

As the navigation of the search space is heavily influenced by the state-value and action-value estimates we obtain from completing a large number of simulations,

UCT with ASNets could require a significantly large number of simulations in order to converge to the optimal policy in comparison to random simulations.

However, given the correct selection of ingredients that guarantees that we will never stop exploring, a flavor of UCT which uses a ASNet-based simulation function will nevertheless converge to the optimal policy, and is hence anytime optimal.

Since there is no way to reduce the influence of an ASNet in Stochastic and Maximum ASNets as more and more simulations are completed, we introduce **mixed simulation functions** which introduce random noise into ASNet-based simulations.

A mixed simulation function takes two different simulation functions, and mixes the action selection at each step of the simulation between the two simulation functions with a certain probability. That is, both simulation functions may be used in a single simulation. For example, we could create a mixed simulation function that uses random simulations 30% of the time, and stochastically samples an action from the learned policy of an ASNet 70% of the time.

It is clear that mixing random simulations with ASNet-based simulations can help navigate the search process away from any misinformation an ASNet provides us. Thus, by introducing mixed simulation functions, we can moderately combat any misleading actions that ASNets may suggest and ensure robustness of our search. However, in such a scenario, it would probably be more effective to just use plain UCT and disregard the policy learned by an ASNet.

How should you choose between using Stochastic ASNets and Maximum ASNets?

Of course, Maximum ASNets will completely bias the simulations towards what an ASNet believes is the best action for a given state. If the probability distribution returned by the learned policy of the ASNet is extremely skewed towards a single action, then we could say that this ASNet is 'confident' in its decision to take this action. Thus, in such a scenario, it may be beneficial to use Maximum ASNets.

On the other hand, if the probability distribution is relatively uniform, then using Stochastic ASNets as a simulation function would more likely be the better choice, as the ASNet may be uncertain and not very 'confident' in which action it should take in the given state. However, this could also be because each action leads to a goal with a similar cost, though this is not usually the case.

Thus, to determine which ASNet-based simulation function to use, we should consider the planning domain and the specific problems we are trying to solve within that domain, and whether an ASNet alone is able to solve these problems reliably. Moreover, we should also consider how well the ASNet has been trained and how this is reflected in the probability distribution returned by the policy.

Summary

We introduced two new simulation functions based on ASNets: Stochastic ASNets and Maximum ASNets. By using these simulation functions, we can guide the navigation of the search tree to what ASNets believes are promising parts of the

search space, and ideally minimize the number of simulations required for UCT to converge to the optimal policy.

Moreover, given that UCT still performs search, we are able to improve any suboptimal learning of an ASNet and combat any changes in the environment or the domain. Thus, we get the best of both worlds.

3.5 Using ASNets in UCB1

As we described in Section 3.3.1, UCB1 allows us to balance the trade-off between exploration of actions in the search tree that have not been applied often, and exploitation of actions that we already know have good action-value estimates based on the previous trials.

By including an ASNet's influence within UCB1, we hope to further bias the action selection towards what ASNets believes are promising actions in a given state, whilst maintaining the fundamental balance between exploration and exploitation.

This will help us achieve the goals we previously defined in Section 3.1: learn what we have not learned, improve suboptimal learning, and be robust to changes in the environment or domain.

3.5.1 Simple ASNet Action Selection

Let $\pi(n_c) = \pi(a(n_c) \mid s(n_c))$. In the action selection step, we select the child chance node n_c of a decision node n_d that maximizes $\texttt{Simple-ASNet}(n_d, n_c)$:

$$\begin{aligned} \texttt{Simple-ASNet}(n_d, n_c) &= \frac{M \cdot \pi(n_c)}{C^k(n_c)} + \text{UCB1}(n_d, n_c) \\ &= \underbrace{\frac{M \cdot \pi(n_c)}{C^k(n_c)} + B \cdot \sqrt{\frac{\log C^k(n_d)}{C^k(n_c)}}}_{\textit{exploration}} - \underbrace{Q^k(n_c)}_{\textit{exploitation}} \end{aligned} \tag{3.5}$$

Where M is a parameter which we call the **influence constant** and, similar to UCB1, if a child chance node n_c has not been visited before (i.e. $C^k(n_c) = 0$), we set $\texttt{Simple-ASNet}(n_d, n_c) = \infty$ to force its exploration.

Intuition

We have added a new 'reward' term to UCB1, $M \cdot \pi(n_c)/C^k(n_c)$. This term essentially forces UCB1 to explore the actions that ASNets wants to exploit, since $\pi(n_c) \in [0, 1]$. That is, we force the exploitation of ASNets for exploration.

M is the influence constant which allows us to scale the influence of ASNets within the action selection. Clearly, a higher value of M increases the influence of an ASNet in the whole Simple-ASNet term, while a lower value of M decreases the influence of an ASNet. In practice, M should be empirically selected based on

the problem we are dealing with, and how well the ASNet has learned to solve that problem.

We divide $M \cdot \pi(n_c)$ by $C^k(n_c)$ so that the influence of an ASNet diminishes in the Simple-ASNet term as an action is applied more often. Hence, as we show later, the whole Simple-ASNet term eventually converges to the optimal action-value $Q^*(n_c)$ as an action is applied infinitely often, given a correct selection of ingredients that guarantees the whole search tree is explored.

Therefore, we expect that Simple-ASNet action selection will be robust to any bad information provided by the policy of a trained ASNet. Obviously, the higher the value of M, the more trials we require to combat the misleading policy provided by an ASNet.

Analysis

We now provide a sketch of a proof that Simple-ASNet action selection will eventually converge to the optimal policy and thus select the optimal action.

It suffices to show that the ASNet term $M \cdot \pi(n_c)/C^k(n_c)$, converges faster to 0 than the UCB1 exploration term $B \cdot \sqrt{\log C^k(n_d)/C^k(n_c)}$, as $C^k(n_c) \to \infty$. This is trivial to prove as $C^k(n_c)$ is linear in the denominator of the ASNet term, while it is a square root in the denominator of the UCB1 exploration term. Thus, Simple-ASNet action selection decomposes into UCB1 as $C^k(n_c) \to \infty$.

Now, we can generalize and use the proof presented by Kocsis and Szepesvári [2006] to show that Simple-ASNet action selection will converge to the optimal action.

3.5.2 Ranked ASNet Action Selection

One pitfall of using Simple-ASNet action selection is that all child chance nodes must be visited at least once before we truly consider the policy learned by an ASNet, as Simple-ASNet$(n_d, n_c) = \infty$ if $C^k(n_c) = 0$.

Ideally, we should be able to use the knowledge learned by an ASNet to select the order in which unvisited chance nodes are explored during action selection. Thus, we introduce Ranked-ASNet action selection, an extension to Simple-ASNet action selection.

$$\text{Ranked-ASNet}(n_d, n_c) = \begin{cases} \text{Simple-ASNet}(n_d, n_c) & \text{if } \forall n'_c \in S(n_d),\, C^k(n'_c) > 0 \\ -\infty & \text{if } C^k(n_c) > 0 \\ \pi(n_c) & \text{otherwise} \end{cases} \tag{3.6}$$

The first condition stipulates that all chance nodes are selected and visited at least once before Simple-ASNet action selection is used. Now, unvisited child chance nodes are visited in order of their probability within the probability distribution of the policy π learned by an ASNet. That is, the order in which the children are visited is determined by each child's rank in the policy.

Intuition

Given that an ASNet has learned some useful knowledge of the environment and which action to apply at each step, we ideally would expect Ranked-ASNet action selection to require a smaller number of trials to converge to the optimal action in comparison with Simple-ASNet action selection.

To understand why this is the case, consider a decision node n_d with 10 child chance nodes, $n_{c_1}, \ldots, n_{c_{10}}$. Let us assume that $\pi(n_{c_1}) = 0.6$, $\pi(n_{c_2}) = 0.32$ and $\pi(n_c) = 0.01$ for all other chance nodes, $n_{c_3}, \ldots, n_{c_{10}}$. Assume that n_{c_2} represents the optimal chance node while $n_{c_3}, \ldots, n_{c_{10}}$ represent chance nodes with trial paths which may end up in a dead end after a few planning steps with an extremely high probability.

In this scenario, we obviously do not want to select $n_{c_3}, \ldots, n_{c_{10}}$ for exploration before n_{c_1} and n_{c_2} as this would likely represent wasted search time with very little information gained. Ranked-ASNet action selection will select n_{c_1} first, and then n_{c_2} the optimal chance node. This is clearly more ideal than choosing child chance nodes at random.

Although the chance nodes $n_{c_3}, \ldots, n_{c_{10}}$ will eventually be visited, it is much better to focus the initial stages of the search on what is believed to be the promising parts of the search space. By doing so, we gain and propagate the most valuable information up the search tree in the limited time we are given.

On the other hand, if the policy learned by an ASNet is misleading and uninformative, then Ranked-ASNet may not be as robust as Simple-ASNet. If the optimal action has a very low probability in the policy, then we require an increased number of trials to converge to this optimum due to the 'ranking' of actions.

Analysis

The proof of optimality shown for Simple-ASNet action selection holds for Ranked-ASNet action selection, as we can guarantee that all child chance nodes will eventually be explored.

3.5.3 Summary

We have demonstrated how we can incorporate the policy learned by an ASNet into the action selection ingredient, by introducing a new term in the UCB1 formula that forces the exploitation of ASNets for exploration in the search tree.

We have also shown how we can decay the influence of an ASNet as we apply and visit a chance node it suggests more frequently, and sketched a proof showing that Simple-ASNet will eventually select the optimal action.

Finally, we introduced Ranked-ASNet action selection, a simple extension to Simple-ASNet action selection that selects unvisited chance nodes according to their corresponding ranking in the probability distribution given by the learned policy.

With these new ASNet-influenced action selection ingredients in mind, we can hopefully overcome the issues we encounter when using either ASNets or UCT alone,

and achieve the goals we discussed in Section 3.1. By combining ASNets with UCT, we evidently get the best of both worlds.

3.6 ASNets as a Simulation Function versus ASNets in UCB1

Using ASNets in a simulation function effectively allows us to calculate a state-value estimate $V^k(n_d)$ for the tip node in a trial n_d. Thus, the state-value and action-value estimates within the search tree are directly derived from what the ASNet suggested during the simulation phase.

On the other hand, using ASNets during action selection by incorporating it within UCB1 allows us to balance the trade-off between the exploration and exploitation of explicit nodes in the search tree. Thus, the state-value and action-value estimates are not immediately derived from what an ASNet suggests during action selection, unless we combine ASNet action selection with ASNet-based simulations.

Moreover, `Simple`-ASNet and `Ranked`-ASNet action selection are more robust to any misleading information an ASNet has learned. Since we can decrease the influence of ASNets as we apply an action the network has suggested more frequently, we will eventually explore actions that may have a small probability $\pi(a \mid s)$, but are in-fact optimal. Although this is theoretically also the case when using ASNets as a simulation function, we may require hundreds of thousands or even millions more trials in comparison to achieve this.

Our experiments show that ASNets should only be used as a simulation function in problems where the domain-independent planning heuristic is misleading and thus gives uninformative state-value estimates, as we have previously discussed in Sections 3.3.3 and 3.3.5. In most scenarios, using ASNets in action selection is more ideal for the reasons previously mentioned.

3.7 Ingredient Configurations

In this section, we demonstrate some of the flavors of UCT we can specify using our general UCT framework. In practice, the selection of ingredients we choose when using ASNets with UCT depends on how well an ASNet can solve the problem we are dealing with, and how informative the learned policy is.

3.7.1 Available Ingredients

- **Action Selection**: UCB1, `Simple`-ASNet, `Ranked`-ASNet
- **Backup Function**: Monte-Carlo, Minimum Monte-Carlo, (Partial) Bellman
- **Heuristic Function**: *zero*, h^{add}, h^{max}, *LM-Cut*
- **Outcome Selection**: Monte-Carlo
- **Simulation Function**: Random, Heuristic-Guided, Stochastic ASNet, Maximum ASNet, Mixed Random with ASNet
- **Trial Length**: any positive integer

Optional features: Q-value initialization of children of a tip node, explicit nodes in the search tree for the simulation phase.

3.7.2 Flavors of UCT

For our experiments, we use UCT* as the baseline flavor of UCT. From here onwards, plain UCT and UCT* is synonymous.

We assume that all flavors of UCT use Monte-Carlo outcome selection, as that is the only option we provide. Moreover, any heuristic function can be used with each flavor of UCT. Our experiments showed that using h^{add} helped the tree search converge to a good solution much faster than if h^{max} or *LM-Cut* was used.

We now specify the main flavors of UCT we will consider in our experiments in Table 3.1.

	Action Selection	Backup Function	Simulation Function	Trial Length
UCT/UCT*	UCB1	Bellman	None	0
Rollout-based UCT	UCB1	(Min) Monte-Carlo	Random	Problem dependent
UCT + Simple ASNets	Simple ASNets	Bellman	None	0
UCT + Ranked ASNets	Ranked ASNets	Bellman	None	0
UCT + Stochastic ASNets	UCB1	Depends on $\pi(a \mid s)$	Stochastic ASNets	Problem dependent
UCT + Maximum ASNets	UCB1	Depends on $\pi(a \mid s)$	Maximum ASNets	Problem dependent

Table 3.1: Examples of Ingredient Configurations

3.8 Summary

In this chapter, we discussed the goals we hope to achieve in combining UCT with ASNets, and defined a general ingredient-based framework for UCT that is similar to THTS [Keller and Helmert, 2013]. Using this framework, we then analyzed how different selections of ingredients can influence the search of UCT.

Next, we introduced ASNets as a simulation function through Stochastic ASNets and Maximum ASNets. We argued that you should consider using ASNets as a simulation function if the heuristic estimates are uninformative and an ASNet has learned some useful knowledge about the environment. In such a situation, we can combine the advantages of both UCT and ASNets together to get the best of both worlds.

We then introduced Simple-ASNet action selection, an extension to UCB1 that leverages the policy learned by an ASNet. We discussed how the influence of an ASNet in Simple-ASNet will decay as an action is applied more often, and gave a sketch of a proof that Simple-ASNet will eventually converge to the optimal policy and thus select the optimal action. Next, we discussed the potential pitfalls of Simple-ASNet action selection and introduced Ranked-ASNet action selection, which selects unexpanded child nodes according to their ranking within the policy, and uses Simple-ASNet otherwise.

By incorporating ASNets into UCB1, we can directly bias the actions selected by UCT for the exploration of actions an ASNet wishes to exploit. Hence, we can obtain the best of both worlds.

Finally, we argued the differences between using ASNets as a simulation function and within action selection in the UCB1 term, and presented some flavors of UCT that can be specified using our general framework.

In the next chapter, we will begin discussing the experimental configuration, and the domains and problems we will be evaluating our algorithms on.

Chapter 4

Empirical Evaluation

In this chapter, we will present the results of our experiments in combining UCT with ASNets. Firstly, we will describe the common configurations that we will use to run UCT, and to train an ASNet.

Next, in Section 4.2, we will review and analyze the problems and domains we evaluate our algorithms on. Finally, we give a detailed discussion the results of our experiments in Section 4.3

4.1 Experimental Setup

All ASNet and UCT experiments were run on an Amazon Web Services EC2 `c5.4x large` instance with an Intel Xeon Platinum 8000 series processor. In total, this instance has 16 virtual CPUs and 32GB of memory.

Each experiment was limited to one CPU core with a maximum turbo clock speed of 3.5 GHz. We did not place any restrictions on the amount of memory an experiment used.

4.1.1 UCT Configuration

The baseline UCT configuration was UCT* (as described in Section 3.7), with h^{add} as the heuristic function and the UCB1 bias B set to $\sqrt{2}$. We use h^{add} because in our experiments, it allowed UCT to converge to a good solution in a reasonable time.

Recall, UCT* completes a trial when an unexpanded decision node is reached, and a heuristic estimate for the given state is calculated and backed up the search tree. This allows us to focus the search on shallower parts of the search space, much like breadth-first search [Keller and Helmert, 2013].

For all problems with dead ends, we enable Q-value initialization, as it helps us avoid selecting a chance node for exploration as it may lead to a dead end. We did not enable Q-value initialization for problems without dead ends because estimating Q-values is computationally expensive, and not beneficial in comparison to the number of trials that could have been performed in the same time frame.

We have already presented the main flavors we will consider in Table 3.1. The flavor of UCT we use with ASNets is dependent on the domain and problem we are

tackling, and whether an ASNet has learned a good policy in such a scenario.

Unless otherwise specified, we gave UCT 10 seconds to do trials at each planning step, and limited the maximum number of trials at 10,000 per planning step. Moreover, we set the dead end penalty to be 500. We gave each planning round a maximum time of 1 hour, and a maximum of 100 planning steps. We ran 30 rounds per planner for each experiment.

4.1.2 ASNet Configuration

We use the same ASNet hyperparameters as described by Toyer et al. [2018] to train a network. To summarize, we used an ASNet with "three action layers and two proposition layers, with a hidden representation size of 16 for each internal action and proposition module", and enabled *LM-Cut* heuristic features.

We trained the network with an Adam optimizer using a batch size of 128, learning rate of 0.0001, and a dropout of 0.25. We imposed a strict two hour time limit to train the network, though in most situations, the network finished training within one hour.

We trained an ASNet using an LRTDP-based teacher that used *LM-Cut* as the heuristic to compute optimal policies. We only report the time taken to solve each problem for the final results for an ASNet, and hence do not include the training time.

4.2 Domains and Problems

We evaluate our algorithms on a wide-variety of domains, aimed at showing that combining UCT with ASNets can give us the best of both worlds.

4.2.1 Stack Blocksworld

We introduced the deterministic Blocksworld domain in Section 2.1.2. As we have already discussed in the 'Learn what we have not learned' goal in Section 3.1.1, an ASNet trained to unstack blocks from a single tower and put them all down on the table, would fail completely to then stack these blocks into a single tower. This is because the network never learned how to stack blocks on top of each other during the training phase, as all the training problems were focused on unstack blocks from a single tower (see Figure 3.1). Thus, the training set is not representative of the test set.

This represents a worst-case scenario for ASNets. On the other hand, stacking blocks into a single tower is a relatively easy problem for UCT. However, as the number of blocks in the problem increases, UCT will begin to struggle as the number of states and actions increase exponentially.

Our goal is to show that by combining UCT with ASNets, we can overcome the misleading information returned by the policy learned by the ASNet. Hopefully, we will be able to achieve similar performance to that achieved when using UCT alone.

4.2.2 Exploding Blocksworld

Exploding Blocksworld is an extension to the original Blocksworld domain that incorporates probabilities and dead ends. In Exploding Blocksworld, putting down a block can detonate and destroy the block or the table it was put down on, with a probability of 10% and 40% respectively. Once a block is exploded, we can no longer use the block as it 'disappears' from the environment.

Thus, a good policy to an Exploding Blocksworld problem avoids placing a block down on the table, or down on another block that is required for the goal state. However, it is possible to construct a problem where we must perform one of these actions at some point during the planning execution. Thus, Exploding Blocksworld problems can have unavoidable dead ends.

It is very difficult for an ASNet to reliably learn to solve Exploding Blocksworld problems. Firstly, each problem could have its own 'trick' in order to avoid dead ends as much as possible, and reach the goal with the smallest cost. Given the limited modelling capacity of the neural network, it is very difficult to learn a generalized policy that characterizes all these tricks. Moreover, since each problem can have its own 'trick', it is very likely that an ASNet will overfit to the problems it has been trained on, and hence would fail to generalize to new problems it has not seen before.

We hope that by combining ASNets with UCT, we can exploit the limited knowledge and 'tricks' learned by an ASNet to help navigate the search space, and solve problems more reliably at a lower cost. In other words, we aim to 'learn what we have not learned' and 'improve suboptimal learning' (see Section 3.1 on goals).

4.2.3 CosaNostra Pizza

The CosaNostra Pizza planning problem was first introduced by Toyer et al. [2018]. The objective of CosaNostra Pizza is to safely deliver a pizza from the pizza shop to the waiting customer, and then return to the shop. There is only one two-way road between the pizza shop and the customer (see Figure 4.1). On this road, there are a series of toll booths.

At each toll booth, we can choose to either pay the toll operator or drive straight through the toll booth without paying. By driving straight through the toll booth, we save a time step, but the operator becomes angry. Angry operators will drop the toll gate on you and crush your car with a probability of 50% when you next pass through their toll booth. An angry operator will stay angry, even if you try to pay the toll.

In CosaNostra Pizza, we must pass through each toll booth at least twice as there is only one two-way road. Hence, the optimal policy is to pay the toll operators when we are travelling to the customer to ensure a safe return, but to avoid paying the operator on our trip back from the customer to the pizza shop [Toyer et al., 2018].

CosaNostra Pizza is an example of a problem with avoidable dead ends. That is, by ensuring that we pay the toll operator on our trip to the customer, we can guarantee that we can avoid any dead ends on the trip back. An ASNet is able to learn this 'trick', and reliably solves a CosaNostra Pizza problem with any number of

Figure 4.1: Two-Way CosaNostra Pizza

toll-booths, even when the network has been trained on problems with only 1-5 toll booths.

Analysis

CosaNostra Pizza was designed to be extremely difficult for heuristic search planners that use determinising heuristics. As determinising heuristics assume that all outcomes of an action can be made true, they fail to consider the 50% probability of reaching a dead end when a toll operator is not paid, because such heuristics can simply select the most convenient outcome. Thus, the heuristic will not reward states in which a toll-operator has in-fact been paid, as it fails to comprehend the advantage of paying the operator and avoiding a potential dead end.

The heuristics we are considering (h^{add}, h^{max}, and $LM\text{-}Cut$) are calculated through delete relaxation, where once a proposition is true it stays true. In the context of CosaNostra Pizza, this means that once we visit a location, we are always at the location. Thus, the heuristic will suggest to never pay the toll operator, as it believes that the agent is simultaneously at the shop and at the customer after delivering the pizza.

Hence, 'learning' to pay the toll operator on the trip to deliver the pizza to the customer requires extremely long reasoning chains. By performing a limited depth search using a determinising heuristic, a heuristic search planning algorithm will avoid paying a toll operator because it takes an extra step with no improvement in the heuristic. Such an algorithm fails to take into account what could happen far in the future if a toll operator is not paid now.

It is therefore not surprising to discover that plain UCT fails to solve CosaNostra Pizza for problems with more than 4 toll booths. However, by combining UCT with ASNets, we hope to vastly improve the ability of UCT to reliably solve CosaNostra Pizza for problems with a larger number of toll booths by exploiting the optimal policy learned by the ASNet.

One-Way CosaNostra Pizza

The problems described above assumed a single two-way road between the pizza shop and the customer (Figure 4.1). This immediately lead to the uninformative heuristic values we discussed.

However, if we were to convert the road into a one-way road (Figure 4.2), such that the path to and back from the customer are different, then the heuristic estimate will evidently be informative again because there are no longer any dead ends which can be encountered.

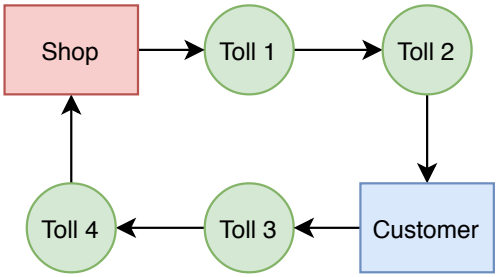

Figure 4.2: One-Way CosaNostra Pizza

Clearly, the optimal policy for one-way CosaNostra Pizza is to never pay the toll operator, as you can only pass through the same toll booth at most once. Although one-way CosaNostra is a relatively easy problem to solve, we use it to demonstrate how we can improve the suboptimal policy learned by an ASNet trained on one-way CosaNostra Pizza (that has learned the trick to always pay the toll-operator before delivering the pizza), by combining the network with search. That is, we want to show that combining UCT and ASNets is robust to changes in the environment or domain.

4.2.4 Triangle Tireworld

The objective in Triangle Tireworld [Little and Thiébaux, 2007] is to navigate a car between the start and end location through a series of locations connected by one-way roads that are arranged in the shape of a triangle (see Figure 4.3).

For each action to move the car from one location to another, there is a probability of 50% that you will get a flat tire. However, there are spare tires at certain locations (represented by black circles in Figure 4.3) which you can use to replace the flat tire and continue navigating. If we have a flat tire, and there is no spare tire at the current location, then we have reached a dead end.

Analysis

In a Triangle Tireworld problem, the tires are arranged in a way such that the optimal policy navigates along the outside edge of the triangle (the path that follows all the black circles in Figure 4.3), and hence reaches the goal with a probability of 1 [Toyer et al., 2018]. This optimal policy also represents the longest path to the goal.

The shortest path to the goal follows the inside edge of the triangle, where no location has a spare tire. However, the probability of a getting a flat tire and reaching a dead end is $1 - 0.5^n$, where n is the number of locations between the initial and goal location on this shortest path. Clearly, this path is not ideal as we will incur the dead end penalty with a probability of $1 - 0.5^n$.

Triangle Tireworld was designed to mislead determinisation-based heuristics. It does so by making the cheapest path in the determinisation to be the path with the smallest probability of reaching the goal. Despite this, both ASNets and plain UCT

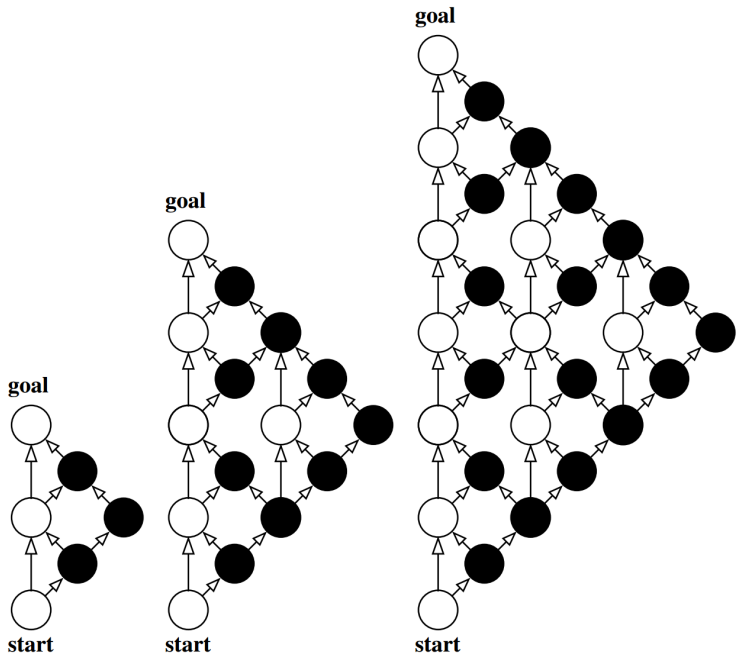

Figure 4.3: The first three Triangle Tireworld problems [Little and Thiébaux, 2007]

are able to learn the 'trick' to always follow the outside edge of the triangle network even when using a determinising heuristic, and thus avoid dead ends and incurring the dead end penalty.

By combining ASNets with UCT in Triangle Tireworld, we aim to understand how using a learned policy can affect the navigation of the search space, and hence the number of nodes initialized and expanded within the search tree. We also hope to demonstrate that combining both algorithms maintains robustness, and does not introduce any new noise into the planning procedure.

4.3 Results

In this section, we will present and analyze the results of our experiments in detail. In general, we found that UCT is robust to the bad information provided by a policy learned by an ASNet if we use `Simple`-ASNet or `Ranked`-ASNet and select the influence constant M appropriately.

Moreover, by combining ASNets with search, we can be robust to any changes in the environment or domain, and solve problems more reliably and with a lower cost. Finally, by using ASNets as a simulation function, we can vastly improve the performance of UCT to solve problems in domains where the heuristic is uninformative or misleading (e.g. two-way CosaNostra Pizza).

Thus, by combining ASNets with MCTS, we can get the best of both worlds.

Experiment Result Tables

The numerical results for our experiments are presented in Tables 4.1, 4.2, 4.3, 4.4, and 4.5 for Stack Blocksworld, Exploding Blocksworld, CosaNostra Pizza, one-way CosaNostra Pizza, and Triangle Tireworld respectively. Each cell in the tables represents the following metrics in order:

1. **Coverage**: the number of successful runs of ASNets/UCT that reached a goal.
2. **Mean Cost**: the mean cost to reach a goal and the 95% confidence interval.
3. **Mean Time**: the mean time to reach a goal and the 95% confidence interval.

It is important to note that the cost and the time of runs of ASNets or UCT that did not reach a goal are **not** included in the mean cost and mean time.

4.3.1 Stack Blocksworld

We train an ASNet on problems with 2-10 blocks, where the objective is to unstack blocks in a single tower and put each block on the table (depicted in Figure 3.1). We evaluate ASNets and UCT on problems with 5-20 blocks, where the objective is to stack blocks that are initially all on the table into a single tower.

As we have previously explained in Section 3.1.1 and 4.2.1, this ASNet will completely fail to stack blocks into a single tower, as it has only learnt how to unstack blocks. This is clearly reflected in the numerical results achieved by ASNets as presented in Table 4.1, where ASNets achieves a coverage of 0% for all test problems.

To demonstrate how UCT can combat the misleading information provided by the policy learned by an ASNet, we use `Simple`-ASNet action selection with the influence constant M set to 10, 50 and 100. We allocate each planning step $n/2$ seconds for all runs of UCT, where n is the number of blocks in the problem.

We do not run experiments that use ASNets as a simulation function for the Stack Blocksworld experiment. Using an ASNet-based simulation functions would result in completely misleading state-value and action-value estimates in the search tree, and hence UCT would achieve near-zero coverage. In fact, using random simulations would be much more effective.

Analysis of Results

ASNets is unable to solve Stack Blocksworld for any of the problems for the reasons we described above. UCT, however, is able to reliably achieve near-full coverage for all problems up to 20 blocks. Although not depicted in Table 4.1, the coverage for UCT decayed exponentially for problems with more than 20 blocks. Evidently, this is because the number of reachable states and actions increases exponentially with the number of blocks in the problem.

In general, as we increase M, the coverage of UCT with `Simple`-ASNet action selection decays earlier as the problem size increases (see Figure 4.4). This is not unexpected, as by increasing M, we increasingly 'push' the UCB1 term to select

actions that an ASNet wishes to exploit, and hence misguide the navigation of the search space. The actions that the ASNet wish to exploit in our case, are those that will unstack a partially-built tower - this is clearly not ideal.

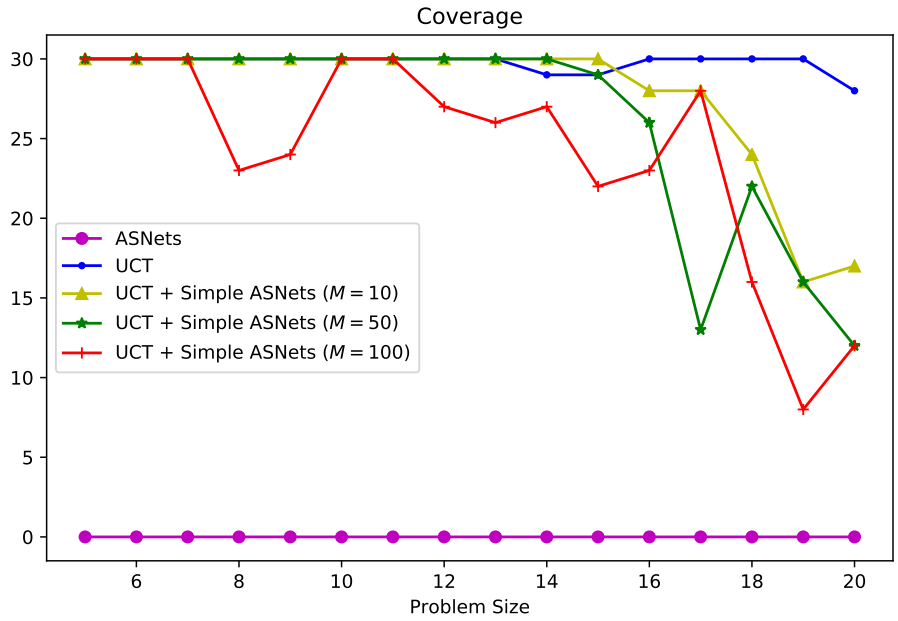

Figure 4.4: Coverage for Stack Blocksworld

Nevertheless, UCT with `Simple`-ASNet action selection is able to achieve near-full coverage for problems with up to 17 blocks for $M = 10$, 15 blocks for $M = 50$, and approximately 11 blocks for $M = 100$. Thus, as we increase M, it is evidently more difficult for UCT to account for the misleading actions that the ASNet is suggesting in the limited time we give each planning step. Note, that due to the noise of UCT, UCT with `Simple`-ASNet action selection achieves a higher coverage for 17 blocks when $M = 100$, than when $M = 50$. We would expect the noise to even out as we increase the number of rounds we evaluate our algorithms on.

Figure 4.5 depicts the mean number of expanded nodes in the search tree, for the runs of UCT where a goal is reached. We can observe that in general, the higher M is in `Simple`-ASNet, the higher the number of nodes that are expanded by UCT. This is not surprising - as M increases, UCT requires more trials and hence more expanded nodes in order to converge to the optimal policy. The ASNet term within `Simple`-ASNet action selection will only decrease as the action an ASNet has suggested is applied more often.

It is intriguing that we see a slight decrease in the coverage (Figure 4.4) and a large increase in the mean number of nodes (Figure 4.5) expanded for UCT with `Simple`-ASNet action selection with $M = 100$ for 8 and 9 blocks. We believe this is due to the noise of UCT and poor guidance during the runs of UCT. In another run of the same experiment, we were only able to achieve a coverage of 5/30 for 8 blocks and 6/30 for 9 blocks – this suggests that the noise of UCT can affect the results

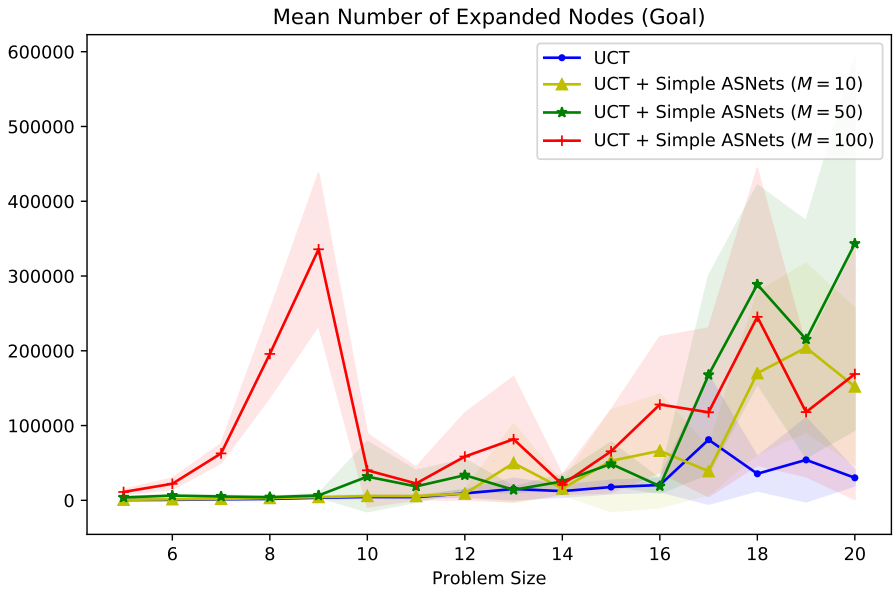

Figure 4.5: Mean and 95% CI for the number of expanded nodes for Stack Blocksworld when a goal is reached

significantly if we are unlucky.

Summary

Through the Stack Blocksworld experiment, we have shown how we can 'learn what we have not learned' by correcting the bad actions an ASNet suggests, and help ASNets stay robust to changes to the environment. We have demonstrated that as we increase M in `Simple`-ASNet action selection, the coverage decreases and the number of nodes expanded increases. UCT with `Simple`-ASNet action selection is relatively robust to the misleading information provided by an ASNet, as we had theorized in Section 3.5.

	ASNets	UCT	UCT + Simple ASNets (M=10)	UCT + Simple ASNets (M=50)	UCT + Simple ASNets (M=100)
blocks-10-stack	0/30	30/30 20.07 ± 0.93 52.18 ± 3.63s	30/30 20.8 ± 0.67 63.25 ± 2.98s	30/30 23.07 ± 4.08 91.67 ± 20.75s	30/30 23.0 ± 4.48 95.04 ± 22.42s
blocks-11-stack	0/30	30/30 21.93 ± 0.8 88.59 ± 4.09s	30/30 22.8 ± 0.8 100.38 ± 4.25s	30/30 23.47 ± 1.87 93.61 ± 11.83s	30/30 23.33 ± 1.81 103.22 ± 9.97s
blocks-12-stack	0/30	30/30 24.07 ± 0.75 104.27 ± 4.44s	30/30 25.2 ± 0.95 93.55 ± 6.56s	30/30 26.07 ± 1.1 98.05 ± 5.84s	27/30 28.0 ± 4.29 137.25 ± 25.77s
blocks-13-stack	0/30	30/30 27.07 ± 1.19 128.56 ± 7.17s	30/30 30.13 ± 3.32 137.15 ± 19.6s	30/30 27.73 ± 0.73 143.14 ± 5.22s	26/30 32.08 ± 6.82 166.24 ± 45.16s
blocks-14-stack	0/30	29/30 28.34 ± 0.98 141.95 ± 6.26s	30/30 29.93 ± 1.42 142.01 ± 12.66s	30/30 29.8 ± 0.91 173.02 ± 21.0s	27/30 28.81 ± 0.96 153.1 ± 8.21s
blocks-15-stack	0/30	29/30 31.38 ± 0.93 170.22 ± 6.09s	30/30 34.27 ± 3.93 157.36 ± 24.75s	29/30 33.79 ± 2.73 195.85 ± 28.7s	22/30 32.55 ± 2.34 149.77 ± 17.56s
blocks-16-stack	0/30	30/30 33.47 ± 1.19 179.83 ± 12.86s	28/30 37.07 ± 4.71 181.98 ± 32.48s	26/30 33.62 ± 0.89 180.11 ± 25.69s	23/30 39.65 ± 5.09 243.09 ± 42.56s
blocks-17-stack	0/30	30/30 38.27 ± 4.5 240.04 ± 36.58s	28/30 38.43 ± 3.72 208.72 ± 35.47s	13/30 46.92 ± 10.32 353.73 ± 98.78s	28/30 38.93 ± 5.03 209.74 ± 36.92s
blocks-18-stack	0/30	30/30 38.6 ± 1.37 193.56 ± 16.04s	24/30 49.17 ± 6.41 295.86 ± 53.89s	22/30 57.64 ± 8.51 451.47 ± 77.67s	16/30 49.12 ± 10.16 358.39 ± 92.2s
blocks-19-stack	0/30	30/30 40.0 ± 2.03 197.12 ± 13.65s	16/30 54.5 ± 9.59 381.26 ± 91.74s	16/30 53.0 ± 9.48 379.12 ± 78.66s	8/30 44.75 ± 4.46 334.33 ± 47.94s
blocks-20-stack	0/30	28/30 42.14 ± 0.89 214.24 ± 8.11s	17/30 52.59 ± 7.15 355.33 ± 64.69s	12/30 60.67 ± 13.74 434.53 ± 128.91s	12/30 46.33 ± 6.24 296.3 ± 51.93s

Table 4.1: Results for Stack Blocksworld. We have not included the results for 5 to 9 blocks as they are essentially identical.

4.3.2 Exploding Blocksworld

Training the ASNet

Selecting a training set for Exploding Blocksworld is extremely difficult, as some problems may be absolutely trivial while others are impossibly hard. It is unfeasible to train an ASNet with extremely difficult problems, as a single training iteration could take more than an hour. Thus, we selected a variety of easy, medium and relatively difficult problems by measuring how long LRTDP took to solve each problem, and ignoring problems that took more then 2 minutes to solve. The final training set consists of 15 problems selected from:

- The International Probabilistic Planning Competition (IPPC) 2006 (5 problems from warm-up, 5 problems from competition final).
- Problems randomly generated by Felipe Trevizan, that have a probability of 1 to reach the goal (3 problems).
- Problems randomly generated by myself, that do not have a probability of 1 to reach the goal (2 problems).

We increased the training time for this network to 5 hours, as calculating the optimal policy for these problems is very expensive during the guided exploration phase of training. Moreover, it should be noted that training the network required approximately 30GB of memory.

Experiment

We evaluate ASNets and UCT on the first ten Exploding Blocksworld problems (`p01-p10`) from the IPPC 2008 competition. We increased the UCB1 bias to 4, and set the maximum number of trials to 30,000 in order to promote more exploration.

To combine UCT with ASNets, we use `Ranked`-ASNet action selection. Recall that `Ranked`-ASNet selects unvisited chance nodes based on their 'ranking' within the policy learned by an ASNet. This allows us to explore actions that the ASNet believes are promising first. Hopefully, this will lead to more accurate state-value and action-values estimates in the search tree, given the limited time to perform trials at each planning step.

We use `Ranked`-ASNet over `Simple`-ASNet action selection because the initial results we achieved for the former were more encouraging. We theorize this is because the policy learned by the ASNet is generally informative, and hence assigns higher probabilities to actions that are truly promising.

Analysis of Results

Table 4.2 presents the numerical results for this experiments. Note, that the coverage for Exploding Blocksworld is an approximation of the true probability of reaching the goal. Since we only run each algorithm 30 times, the results are susceptible to chance.

As our training problems are most likely not representative of the IPPC 2008 problems we are evaluating our algorithms on, the policy learned by the ASNet is suboptimal. For example, the ASNet is unable to reach the goal in any of the experiments we ran for `ex_bw_6_p04` and `ex_bw_9_p07`. On the other hand, UCT alone is able to more reliably solve the majority of problems, as shown by the higher coverage in comparison to ASNets in Table 4.2.

Moreover, we are able to achieve the performance of plain UCT at a minimum when combining UCT with ASNets through `Ranked`-ASN`et` action selection, even if ASNet achieved a coverage of 0%.

However, for certain configurations of UCT with `Ranked`-ASN`et`, we were able to improve upon all other configurations. For `ex_bw_10_p08`, UCT with `Ranked`-ASN`et` and $M = 50$ achieves a coverage of 10/30, while all other configurations of UCT are only able to achieve a coverage of around 4/30. Despite that fact that the ASNet achieves a coverage of 0/30 in this experiment, the general knowledge learned by the ASNet helps us navigate the search tree more effectively and efficiently, even if the suggestions provided by the ASNet are not completely optimal. The same reasoning applies to the results for `ex_bw_6_p04`, where UCT with `Ranked`-ASN`et` and $M = 50$ achieves a higher coverage than all other configurations.

In general, the results in Table 4.2 suggest that the influence constant M should be empirically selected based on the domain we are trying to solve, or automatically tuned during the search. Since the knowledge learned by an ASNet can be misleading for some problems, and informative for others, we need to find a good balance for the influence constant M. If M is too small, then we do not exploit the knowledge of an ASNet enough, while if M is too big, we follow the suboptimal actions suggested by an ASNet too often. An example of this is `ex_bw_10_p08`, where $M = 50$ achieves a good balance between exploiting the ASNet too little ($M = 10$) and too much ($M = 100$).

Summary

Through the Exploding Blocksworld experiment, we have demonstrated that we can exploit the policy learned by an ASNet to achieve more promising results than plain UCT, even if this policy is suboptimal.

Since the ASNet has learned 'tricks' from the training problems, some of these tricks may be applied to the problems we are evaluating our algorithms on, while some cannot. Because of this, we must empirically select the influence constant M when combining UCT with an ASNet-influenced action selection ingredient. The local knowledge learned by an ASNet may only be beneficial in some problems, while detrimental in others.

Thus, this experiment has demonstrated that we can achieve all the goals we presented in Section 3.1: learn what we have not learned, improve suboptimal learning, and be robust to changes in the environment. It is clear that by combining ASNets with UCT, we can achieve the best of both worlds.

	ASNets	UCT	UCT + Ranked ASNet (M=10)	UCT + Ranked ASNet (M=50)	UCT + Ranked ASNet (M=100)
ex_bw_5_p01	16/30 8.0 ± 0.0 0.18 ± 0.14s	26/30 10.92 ± 0.52 102.51 ± 5.24s	25/30 10.96 ± 0.48 100.21 ± 6.01s	23/30 11.04 ± 0.58 94.17 ± 6.51s	25/30 11.04 ± 0.48 105.26 ± 4.83s
ex_bw_5_p02	10/30 12.0 ± 0.0 0.17 ± 0.01s	9/30 18.22 ± 1.62 175.01 ± 16.24s	6/30 17.0 ± 3.45 164.77 ± 34.89s	10/30 17.6 ± 2.85 166.29 ± 27.91s	12/30 17.33 ± 2.44 167.75 ± 24.5s
ex_bw_6_p03	6/30 10.0 ± 0.0 0.2 ± 0.04s	13/30 25.23 ± 8.86 222.27 ± 88.77s	11/30 30.0 ± 13.64 280.25 ± 135.07s	14/30 35.71 ± 7.87 352.14 ± 78.66s	14/30 28.43 ± 6.54 259.18 ± 65.16s
ex_bw_6_p04	0/30	11/30 14.55 ± 0.63 136.46 ± 6.75s	10/30 14.4 ± 0.6 125.74 ± 11.93s	15/30 14.4 ± 0.46 123.06 ± 5.75s	10/30 14.6 ± 0.69 126.61 ± 6.41s
ex_bw_7_p05	30/30 6.0 ± 0.0 0.19 ± 0.07s	30/30 6.13 ± 0.19 36.51 ± 2.4s	30/30 6.0 ± 0.0 38.11 ± 1.17s	30/30 6.0 ± 0.0 38.85 ± 1.15s	30/30 6.0 ± 0.0 39.41 ± 1.08s
ex_bw_8_p06	19/30 12.0 ± 0.0 0.42 ± 0.12s	28/30 13.93 ± 0.8 132.36 ± 8.11s	25/30 13.6 ± 0.83 113.56 ± 8.11s	27/30 13.33 ± 0.76 127.69 ± 7.59s	29/30 13.38 ± 0.74 111.66 ± 7.15s
ex_bw_9_p07	0/30	30/30 13.0 ± 0.73 107.11 ± 6.95s	30/30 12.07 ± 0.14 116.36 ± 1.4s	30/30 12.07 ± 0.14 102.57 ± 1.38s	30/30 12.33 ± 0.28 103.56 ± 3.16s
ex_bw_10_p08	0/30	5/30 36.4 ± 5.09 335.87 ± 54.56s	4/30 35.0 ± 7.58 340.82 ± 75.18s	10/30 38.6 ± 0.97 374.93 ± 12.01s	4/30 36.5 ± 9.14 344.06 ± 93.88s
ex_bw_11_p09	0/30	0/30	0/30	0/30	0/30
ex_bw_12_p10	0/30	0/30	0/30	0/30	0/30

Table 4.2: Results for Exploding Blocksworld. Note: ex_bw_7_p05 is a trivial problem.

4.3.3 CosaNostra Pizza

As we have already discussed in Section 4.2.3, an ASNet is able to learn the trick to pay the toll operators when we are travelling to the customer to deliver the pizza, and avoid paying the tolls on the trip back to the pizza shop. We argued that this requires very long chains of reasoning, which is why UCT is expected to struggle significantly. Moreover, we analyzed why the heuristic estimates are uninformative and provide little value to help us more efficiently navigate the search space.

We train an ASNet on problems with 1-5 toll-booths, and evaluate UCT and ASNets on problems with 2 to 15 toll booths. In this experiment, we will consider using ASNets as both a simulation function (Stochastic and Maximum ASNets), and in the UCB1 term for action selection (`Simple`-ASNet and `Ranked`-ASNet with $M = 100$). The results of our experiments are presented in Table 4.3.

The optimal policy for CosaNostra Pizza takes $3n + 4$ steps, where n is the number of toll booths in the problem. We set the trial length when using ASNets as a simulation function to be $\lfloor 1.25 \cdot (3n + 4) \rfloor$. The 25% increase over the length of the optimal policy allows us to give some leeway and deeper simulations which is beneficial for Stochastic ASNets. We use Bellman backups as the policy learned by the ASNet is informative, and hence gives accurate state-value estimates.

Analysis

UCT alone is only able to achieve full coverage for the problems with 2 and 3 toll booths. On the other hand, ASNets is able to achieve full coverage for all problems. Using ASNets in the action selection ingredient through `Simple`-ASNet or `Ranked`-ASNet with the influence constant $M = 100$, yields minimal improvements over plain UCT (see Figure 4.6): we are only additionally able to achieve full coverage for the problem with four toll booths.

To understand why this is the case, consider Figure 4.7, where we show the mean number of expanded nodes when a goal is reached. Although `Simple`-ASNet and `Ranked`-ASNet will guide the action selection to the optimal action, UCT will still significantly explore other parts of the search space. Moreover, due to the uninformative heuristic, UCT does not know whether an action is good or not unless the action has been applied in a trajectory to the goal state. This requires an exponential increase in the number of trials and expanded nodes as the problem size increases, as depicted in Figure 4.7.

However, we are able to much more reliably solve CosaNostra Pizza problems when using ASNets as a simulation function. Since an ASNet is able to learn the optimal policy, using ASNets as a simulation function allow us to obtain much better state-value estimates for nodes in the search tree than those provided by a heuristic. It is easy to see that when we use Maximum ASNets, the state-value $V^*(n_d)$ for the tip node of a trial n_d obtained from the simulation is optimal (assuming a sufficiently large trial length).

UCT with Maximum ASNets as the simulation function is able to achieve full coverage for all the problems. Since Maximum ASNets will always provide us with a

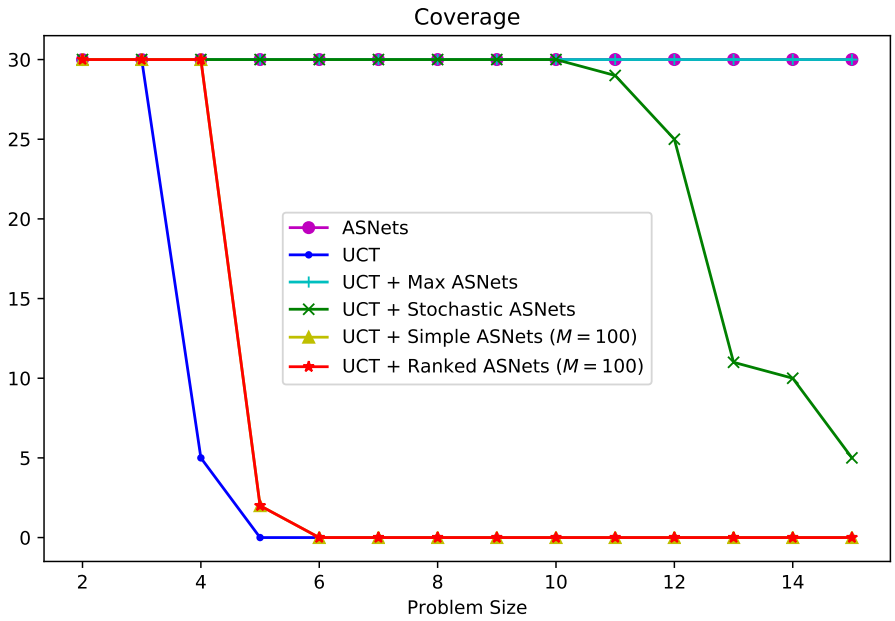

Figure 4.6: Coverage for CosaNostra Pizza. The line for UCT + Ranked ASNets occludes the line for UCT + Simple ASNets, and the line for UCT + Max ASNets occludes the plain ASNets line.

path directly to the goal, UCT will do some search and then decide that the actions suggested through Maximum ASNet simulations are the best it can achieve, and begins exploiting that path. That is, the UCB1 term is effectively dominated by the exploitation term $-Q^k(n_c)$ (see Section 3.3.1). These statements are clearly reflected through the number of expanded nodes for UCT + Maximum ASNets in Figure 4.7.

Similar reasoning applies to UCT with Stochastic ASNets as the simulation function. However, we see an exponential decay in the coverage as the problem size increases above 10 toll booths. The reason for this is because as the problem size increases, the probability of obtaining a path that leads directly to the goal decreases. Hence, UCT cannot default to the path the ASNet has provided it, as this path may not exist in the search tree. This is reflected through the increased number of expanded nodes for UCT + Stochastic ASNets in Figure 4.7.

Summary

We have shown how using ASNets in UCB1 through `Simple-ASNet` or `Ranked-ASNet` action selection can only provide marginal improvements over plain UCT when the number of reachable states increases exponentially with the problem size, and the estimates provided by the heuristic are uninformative.

We demonstrated how we can combat this issue by using ASNets as a simulation function, which allows us to obtain much more accurate state-value and action-value estimates in the search tree, and hence more efficiently explore the search space and

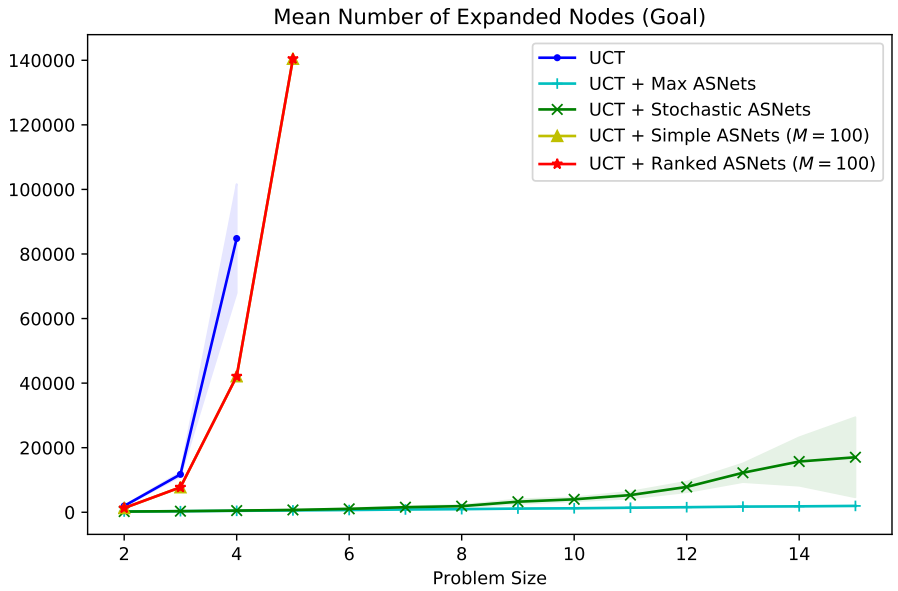

Figure 4.7: Mean and 95% CI for the number of expanded nodes for CosaNostra Pizza when a goal is reached. Only UCT + Max ASNets achieves full coverage.

find the optimal policy.

	ASNets	UCT	UCT + Max ASNets	UCT + Stochastic ASNets	UCT + Simple ASNet (M=100)	UCT + Ranked ASNet (M=100)
cosanostra-n2	30/30 10.0 ± 0.0 0.09 ± 0.05s	30/30 10.0 ± 0.0 15.81 ± 1.09s	30/30 10.0 ± 0.0 25.83 ± 0.19s	30/30 10.73 ± 0.37 27.61 ± 1.47s	30/30 10.0 ± 0.0 73.13 ± 4.48s	30/30 10.0 ± 0.0 79.44 ± 0.74s
cosanostra-n3	30/30 13.0 ± 0.0 0.07 ± 0.03s	30/30 13.0 ± 0.0 36.57 ± 0.3s	30/30 13.0 ± 0.0 38.74 ± 0.35s	30/30 14.13 ± 0.42 42.93 ± 2.11s	30/30 13.0 ± 0.0 80.28 ± 1.34s	30/30 13.0 ± 0.0 108.35 ± 0.2s
cosanostra-n4	30/30 16.0 ± 0.0 0.15 ± 0.05s	5/30 17.6 ± 1.11 64.95 ± 7.16s	30/30 16.0 ± 0.0 54.51 ± 0.19s	30/30 18.13 ± 0.52 64.7 ± 3.17s	30/30 16.0 ± 0.0 104.45 ± 2.38s	30/30 16.0 ± 0.0 124.41 ± 7.27s
cosanostra-n5	30/30 19.0 ± 0.0 0.18 ± 0.05s	0/30	30/30 19.0 ± 0.0 73.31 ± 0.21s	30/30 22.0 ± 0.61 88.25 ± 4.08s	2/30 19.0 ± 0.0 119.55 ± 6.71s	1/30 27.0 245.67s
cosanostra-n6	30/30 22.0 ± 0.0 0.22 ± 0.05s	0/30	30/30 22.0 ± 0.0 93.41 ± 0.47s	30/30 25.6 ± 0.82 114.86 ± 6.54s	0/30	0/30
cosanostra-n7	30/30 25.0 ± 0.0 0.26 ± 0.05s	0/30	30/30 25.0 ± 0.0 121.02 ± 1.45s	30/30 29.53 ± 0.81 147.25 ± 7.32s	0/30	0/30
cosanostra-n8	30/30 28.0 ± 0.0 0.31 ± 0.05s	0/30	30/30 28.0 ± 0.0 89.11 ± 1.15s	30/30 34.33 ± 0.9 116.88 ± 5.38s	0/30	0/30
cosanostra-n9	30/30 31.0 ± 0.0 0.36 ± 0.05s	0/30	30/30 31.0 ± 0.0 102.9 ± 1.45s	30/30 37.33 ± 0.62 136.62 ± 4.38s	0/30	0/30
cosanostra-n10	30/30 34.0 ± 0.0 0.29 ± 0.06s	0/30	30/30 34.0 ± 0.0 197.99 ± 0.86s	30/30 41.6 ± 0.91 240.02 ± 19.3s	0/30	0/30
cosanostra-n11	30/30 37.0 ± 0.0 0.3 ± 0.03s	0/30	30/30 37.0 ± 0.0 226.04 ± 0.99s	29/30 45.69 ± 1.04 287.56 ± 17.49s	0/30	0/30
cosanostra-n12	30/30 40.0 ± 0.0 0.36 ± 0.02s	0/30	30/30 40.0 ± 0.0 255.1 ± 0.78s	25/30 50.32 ± 1.3 322.44 ± 27.56s	0/30	0/30
cosanostra-n13	30/30 43.0 ± 0.0 0.65 ± 0.05s	0/30	30/30 43.0 ± 0.0 285.2 ± 0.74s	11/30 56.09 ± 2.03 350.76 ± 54.76s	0/30	0/30
cosanostra-n14	30/30 46.0 ± 0.0 0.72 ± 0.05s	0/30	30/30 46.0 ± 0.0 314.55 ± 0.77s	10/30 58.0 ± 2.94 338.77 ± 68.49s	0/30	0/30
cosanostra-n15	30/30 49.0 ± 0.0 0.8 ± 0.05s	0/30	30/30 49.0 ± 0.0 345.66 ± 1.49s	5/30 64.6 ± 3.24 452.19 ± 107.4s	0/30	0/30

Table 4.3: Results for CosaNostra Pizza

4.3.4 One-Way CosaNostra Pizza

In one-way CosaNostra Pizza, there is only one path which may be followed to travel from the pizza shop to the customer, and back (Figure 4.2). The optimal policy in one-way CosaNostra Pizza is to never pay the toll operator, as you will only ever encounter the same operator once. Thus, the number of steps required to reach the goal state following the optimal policy is $n + 4$, where n is the number of toll booths.

We use the same ASNet we trained in two-way CosaNostra Pizza (see Section 4.3.3) for this experiment. Clearly, the policy learned by this ASNet is suboptimal (Figure 4.8), as it will always choose to pay the toll operator when travelling to the customer to deliver the pizza on a two-way road. In fact, we found that this ASNet would always suggest to pay the toll operator even if the pizza is already delivered in one-way CosaNostra Pizza.

We evaluate our algorithms on problems with 2 to 15 toll booths. We will consider using ASNets as a simulation function (Stochastic and Maximum ASNets) and within the UCB1 term (`Simple-ASNet` with $M = 10$ and $M = 50$).

We use the same trial length of $\lfloor 1.25 \cdot (3n + 4) \rfloor$ we used in the two-way CosaNostra experiment (4.3.3), and use Bellman backups.

The numerical results of this experiment can be found in Table 4.4.

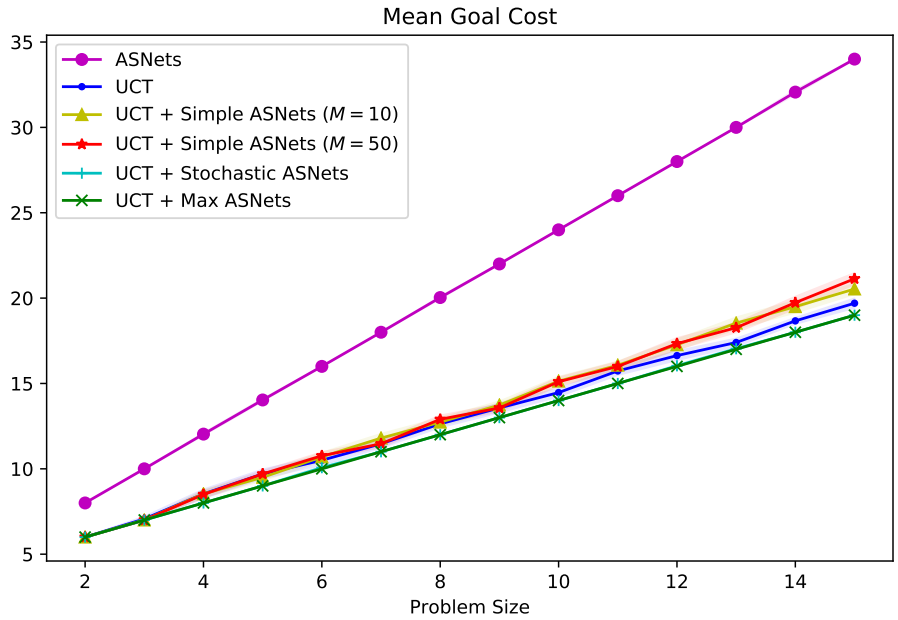

Figure 4.8: Mean and 95% CI for the goal cost for one-way CosaNostra Pizza

Analysis of Results

One-Way CosaNostra Pizza is relatively trivial, with both ASNets and plain UCT achieving full coverage for all problems. We have already explained that the policy

learned by the ASNet is suboptimal, because it was trained on two-way CosaNostra Pizza. Similarly, UCT does not always follow the optimal policy, opting to pay the toll operator very occasionally (see the confidence intervals in Table 4.4). The reason why this is the case is because UCT may have found a path to the goal, and decided that it is no longer worth exploring other actions and begins exploiting that path.

When we use `Simple-ASNet` action selection with the influence constant $M = 10$ and $M = 50$, we achieve almost identical results to plain UCT. UCT is able to account for the change in the environment of the problem, despite the fact the ASNet will always suggest to pay the toll with $\pi(\text{pay-operator} \mid s) \approx 0.999...$.

It is also interesting to compare the number of expanded nodes when using `Simple-ASNets` with different values of M, and against plain UCT (Figure 4.9). Once we incorporate ASNets into UCB1, we increase the number of nodes that must be expanded in the search process. This is not surprising, as an ASNet will always favour paying the toll operator, and hence UCT must apply more search to discover that paying the operator is in fact, not the optimal action. The same reasoning applies to why more nodes are expanded as we increase the influence constant, M.

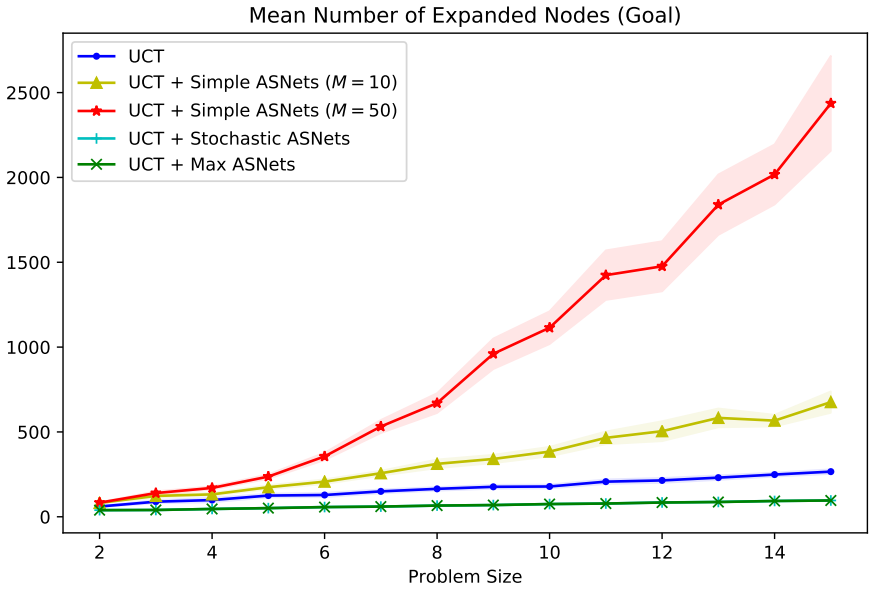

Figure 4.9: Mean and 95% CI for the number of expanded nodes for one-way CosaNostra Pizza when a goal is reached

When combining UCT with ASNets as a simulation function, we almost always follow the optimal policy (see Table 4.4). To understand why this is the case, consider the state-value estimate $V^k(n_d)$ obtained from a simulation for a tip node of the trial n_d. As the ASNet will always suggest to pay the toll operator, the state-value estimate $V^k(n_d)$ is always bounded above by the optimal state-value for one-way CosaNostra Pizza, no matter if we use Stochastic or Maximum ASNets (this is guaranteed by our trial length). That is, $V^*(n_d) \leq V^k(n_d) - t$, where t is the number of toll booths on the path from the current state back to the pizza shop.

Thus, the simulations provide much more meaningful estimates of the cost to reach the goal than a heuristic, and UCT is quickly guided to the goal by the simulation function. We can confirm this by observing that the number of expanded nodes when using ASNets as a simulation function is minimized (Figure 4.9).

Summary

Thus, our experiments for one-way CosaNostra Pizza have shown that combining UCT with ASNets can help us learn what we have not learned. That is, learning to not pay the toll-operator if we encounter a one-way road, but to pay the operator if we encounter a two-way road, as we may encounter the toll-operator again in the future.

We have also shown that combining UCT with ASNets allows us to improve the suboptimal policy which resulted from the changed environment. Although the policy learned by the ASNet is not optimal for one-way CosaNostra Pizza, the state-value estimates obtained from using ASNets as a simulation function provide much more accurate estimates of the true cost to reach the goal in comparison to the estimates provided by a planning heuristic. Altogether, this helps UCT converge to the optimal policy in a much smaller number of trials.

	ASNets	UCT	UCT + Simple ASNets (M=10)	UCT + Simple ASNets (M=50)	UCT + Stochastic ASNets	UCT + Max ASNets
cosanostra-n2-one-way	30/30 8.0 ± 0.0 0.07 ± 0.05s	30/30 6.0 ± 0.0 11.35 ± 0.05s	30/30 6.0 ± 0.0 12.89 ± 0.06s	30/30 6.0 ± 0.0 13.0 ± 0.06s	30/30 6.0 ± 0.0 11.74 ± 0.13s	30/30 6.0 ± 0.0 11.77 ± 0.07s
cosanostra-n3-one-way	30/30 10.0 ± 0.0 0.09 ± 0.05s	30/30 7.07 ± 0.14 14.13 ± 0.43s	30/30 7.0 ± 0.0 16.13 ± 0.11s	30/30 7.0 ± 0.0 16.35 ± 0.12s	30/30 7.0 ± 0.0 14.65 ± 0.14s	30/30 7.0 ± 0.0 14.66 ± 0.17s
cosanostra-n4-one-way	30/30 12.03 ± 0.07 0.07 ± 0.03s	30/30 8.53 ± 0.25 18.71 ± 0.86s	30/30 8.53 ± 0.21 21.77 ± 0.89s	30/30 8.5 ± 0.19 21.89 ± 0.83s	30/30 8.0 ± 0.0 11.15 ± 0.13s	30/30 8.0 ± 0.0 17.65 ± 0.1s
cosanostra-n5-one-way	30/30 14.03 ± 0.07 0.08 ± 0.03s	30/30 9.7 ± 0.26 22.53 ± 0.97s	30/30 9.5 ± 0.19 25.2 ± 0.79s	30/30 9.7 ± 0.17 26.73 ± 0.74s	30/30 9.0 ± 0.0 13.22 ± 0.17s	30/30 9.0 ± 0.0 21.27 ± 0.2s
cosanostra-n6-one-way	30/30 16.0 ± 0.0 0.14 ± 0.05s	30/30 10.5 ± 0.25 25.69 ± 1.09s	30/30 10.7 ± 0.24 30.61 ± 1.13s	30/30 10.77 ± 0.21 31.1 ± 1.48s	30/30 10.07 ± 0.14 18.99 ± 2.21s	30/30 10.0 ± 0.0 24.62 ± 0.11s
cosanostra-n7-one-way	30/30 18.0 ± 0.0 0.16 ± 0.05s	30/30 11.47 ± 0.23 28.77 ± 0.97s	30/30 11.8 ± 0.25 36.08 ± 1.31s	30/30 11.47 ± 0.19 33.6 ± 1.36s	30/30 11.0 ± 0.0 29.09 ± 0.37s	30/30 11.0 ± 0.0 28.67 ± 0.3s
cosanostra-n8-one-way	30/30 20.03 ± 0.07 0.19 ± 0.05s	30/30 12.63 ± 0.21 21.66 ± 0.67s	30/30 12.73 ± 0.24 25.01 ± 0.91s	30/30 12.9 ± 0.23 26.04 ± 0.9s	30/30 12.0 ± 0.0 32.84 ± 0.52s	30/30 12.0 ± 0.0 32.21 ± 0.1s
cosanostra-n9-one-way	30/30 22.0 ± 0.0 0.16 ± 0.05s	30/30 13.57 ± 0.27 23.91 ± 1.02s	30/30 13.73 ± 0.22 28.04 ± 0.94s	30/30 13.57 ± 0.21 28.52 ± 0.84s	30/30 13.0 ± 0.0 33.29 ± 2.7s	30/30 13.0 ± 0.0 36.93 ± 0.17s
cosanostra-n10-one-way	30/30 24.0 ± 0.0 0.18 ± 0.05s	30/30 14.47 ± 0.23 29.37 ± 3.44s	30/30 15.13 ± 0.25 35.41 ± 4.07s	30/30 15.1 ± 0.25 36.32 ± 4.32s	30/30 14.0 ± 0.0 39.78 ± 1.75s	30/30 14.0 ± 0.0 41.02 ± 0.14s
cosanostra-n11-one-way	30/30 26.0 ± 0.0 0.27 ± 0.04s	30/30 15.73 ± 0.24 32.99 ± 3.58s	30/30 16.07 ± 0.22 40.04 ± 4.82s	30/30 16.0 ± 0.26 40.3 ± 4.73s	30/30 15.0 ± 0.0 33.49 ± 3.48s	30/30 15.0 ± 0.0 45.92 ± 0.18s
cosanostra-n12-one-way	30/30 28.0 ± 0.0 0.31 ± 0.04s	30/30 16.63 ± 0.27 36.62 ± 4.68s	30/30 17.27 ± 0.31 45.2 ± 6.02s	30/30 17.33 ± 0.33 46.82 ± 6.76s	30/30 16.03 ± 0.07 31.68 ± 0.46s	30/30 16.0 ± 0.0 51.36 ± 0.29s
cosanostra-n13-one-way	30/30 30.0 ± 0.0 0.32 ± 0.04s	30/30 17.4 ± 0.23 40.38 ± 5.18s	30/30 18.53 ± 0.34 51.05 ± 6.78s	30/30 18.27 ± 0.28 52.03 ± 7.07s	30/30 17.03 ± 0.07 33.65 ± 0.61s	30/30 17.0 ± 0.0 40.57 ± 4.26s
cosanostra-n14-one-way	30/30 32.07 ± 0.14 0.37 ± 0.04s	30/30 18.67 ± 0.18 45.03 ± 5.99s	30/30 19.5 ± 0.35 56.9 ± 8.17s	30/30 19.73 ± 0.35 59.09 ± 8.22s	30/30 18.0 ± 0.0 52.88 ± 5.16s	30/30 18.0 ± 0.0 32.56 ± 0.34s
cosanostra-n15-one-way	30/30 34.0 ± 0.0 0.4 ± 0.04s	30/30 19.7 ± 0.28 49.83 ± 6.61s	30/30 20.53 ± 0.31 63.4 ± 8.7s	30/30 21.13 ± 0.34 65.15 ± 8.24s	30/30 19.0 ± 0.0 69.18 ± 0.74s	30/30 19.0 ± 0.0 38.7 ± 3.71s

Table 4.4: Results for One-Way CosaNostra Pizza

4.3.5 Triangle Tireworld

The optimal policy in Triangle Tireworld is to follow the outer edge of the triangle navigation map, such that each location contains a spare tire (see Figure 4.3). This path ensures that we avoid all dead ends and reach the goal with a probability of 1. A problem of size n in Triangle Tireworld has $(n+1)(2n+1)$ locations, and takes approximately $6n$ steps to reach the goal state if we follow the optimal policy [Little and Thiébaux, 2007].

We train an ASNet on problems of size 1-3, and evaluate UCT and ASNets on problems of size 1-10. We consider using ASNets both as a simulation function (Stochastic and Maximum ASNets) and through UCB1 (`Simple-ASNets` with $M = 10$). We set the trial length to be $\lfloor 1.25 \cdot 6n \rfloor$ when a simulation function is used. The extra 25% over the length of the optimal policy gives some leeway which is beneficial to Stochastic ASNets. The numerical results for this experiment are presented in Table 4.5.

Analysis of Results

As we discussed in Section 4.2.4, both UCT and ASNets are able to learn the trick of following the outer edge of the triangle, despite the fact that the heuristic biases the search towards the shortest path to the goal location which has an extremely high probability of failure. Both UCT and ASNets achieve 100% coverage, as seen in Table 4.5.

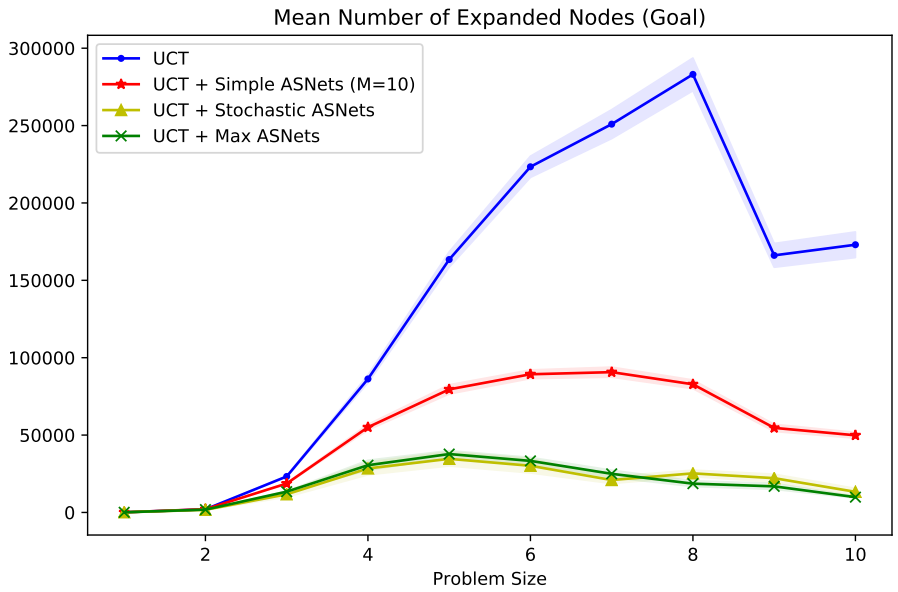

Figure 4.10: Mean and 95% CI for the number of expanded nodes for Triangle Tireworld when a goal is reached. Note: all flavors of UCT achieve full coverage, except for UCT + Stochastic ASNets (see Table 4.5).

As we might expect, UCT with the Maximum ASNet simulation function and

UCT with Simple-ASNet action selection also give us 100% coverage. However, using Maximum ASNets allows us to converge to the optimal policy much faster than using Simple-ASNet action selection, as depicted by the significantly decreased number of expanded nodes in Figure 4.10. This is because using Maximum ASNets as a simulation function allows us to obtain the optimal state-value for the tip node of a trial. This is much more informative than a heuristic which suggests the shortest path to the goal.

It is also interesting to note that using Simple-ASNet for action selection yields a much lower number of expanded nodes than when plain UCT is used. This is because Simple-ASNet will immediately guide us away from any actions that take us to a state that does not have a spare tire, while UCT has to discover that information through a local search of the environment.

However, if we use Stochastic ASNets as the simulation function instead of Maximum ASNets, the coverage decreases exponentially as the problem size increases. The reason why this is the case is because as the size of the problem increases, the number of action modules and proposition modules in an ASNet also increases exponentially. As a result, evaluating the policy of an ASNet is more computationally expensive. Because of this, we can only perform a very limited number of simulations (less than 100 in many cases) in the 10 seconds UCT is given at each planning step.

This means that we cannot always reliably find the safest path to the goal which follows the outer edge of the triangle, as we do not do enough trials to counter the probabilities experienced with stochastic sampling. As such, the state-value and action-value estimates in the search tree could be misleading, and suggest that we move to a location that does not have a spare tire and hence could lead to a dead end with a probability of 50%. However, by increasing the time limit per planning step, we expect that UCT with Stochastic ASNets will achieve higher coverage. We do not encounter this problem when using Maximum ASNets, as we are always following the safest possible path in a simulation from the given state of the tip node of the trial.

Summary

We have discussed how we can maintain the robustness of UCT and ASNets by combining both of them together through ASNets as a simulation function, and ASNets in UCB1. We have demonstrated that we should consider using ASNets as a simulation function if it provides us with accurate estimates of the remaining cost to reach a goal (previously discussed in Section 3.4).

We also discussed the potential pitfalls of using ASNets as a simulation function when it is computationally expensive to evaluate a state to get the policy. In such a scenario, we should consider either increasing the time we give to each planning step, or use the policy learned by ASNet in action selection instead. Using ASNets in action selection means that we only need to call an ASNet once in every trial (unless ASNets is also used as a simulation function).

	ASNets	UCT	UCT + Simple ASNets (M=10)	UCT + Stochastic ASNets	UCT + Max ASNets
triangle-tire-1	30/30 5.53 ± 0.29 0.02 ± 0.01s	30/30 5.43 ± 0.29 17.9 ± 0.98s	30/30 5.27 ± 0.35 10.1 ± 0.64s	30/30 5.2 ± 0.41 17.98 ± 1.48s	30/30 5.43 ± 0.38 18.2 ± 1.2s
triangle-tire-2	30/30 11.3 ± 0.45 0.05 ± 0.01s	30/30 12.13 ± 0.44 59.35 ± 2.04s	30/30 11.37 ± 0.46 64.7 ± 2.59s	29/30 11.83 ± 0.61 62.66 ± 3.55s	30/30 11.83 ± 0.51 62.57 ± 2.58s
triangle-tire-3	30/30 17.2 ± 0.56 0.11 ± 0.01s	30/30 17.23 ± 0.59 71.51 ± 2.76s	30/30 16.67 ± 0.57 77.4 ± 3.19s	24/30 17.17 ± 0.97 111.51 ± 10.86s	30/30 17.33 ± 0.59 115.11 ± 3.86s
triangle-tire-4	30/30 23.37 ± 0.66 0.39 ± 0.02s	30/30 23.37 ± 0.9 130.47 ± 4.83s	30/30 24.03 ± 0.81 146.19 ± 4.7s	21/30 23.71 ± 0.87 179.47 ± 9.36s	30/30 23.67 ± 0.77 166.59 ± 10.55s
triangle-tire-5	30/30 28.87 ± 0.81 0.69 ± 0.03s	30/30 29.6 ± 0.86 193.14 ± 5.81s	30/30 29.07 ± 0.89 199.53 ± 5.71s	20/30 29.75 ± 1.07 201.87 ± 13.08s	30/30 30.27 ± 0.92 198.58 ± 6.71s
triangle-tire-6	30/30 34.87 ± 0.94 1.13 ± 0.04s	30/30 36.43 ± 0.84 261.78 ± 6.13s	30/30 35.8 ± 0.77 266.53 ± 6.05s	14/30 36.71 ± 1.31 282.38 ± 16.03s	30/30 36.13 ± 0.94 260.87 ± 7.07s
triangle-tire-7	30/30 40.77 ± 0.91 1.76 ± 0.04s	30/30 42.77 ± 1.02 325.49 ± 8.48s	30/30 42.8 ± 0.85 338.59 ± 7.6s	13/30 42.92 ± 1.55 377.02 ± 15.52s	30/30 42.3 ± 1.17 341.2 ± 13.96s
triangle-tire-8	30/30 46.83 ± 1.12 2.57 ± 0.06s	30/30 49.37 ± 1.08 393.69 ± 9.28s	30/30 48.13 ± 1.09 393.25 ± 9.06s	12/30 48.83 ± 1.6 393.72 ± 13.1s	30/30 48.57 ± 0.8 422.81 ± 10.31s
triangle-tire-9	30/30 52.93 ± 1.27 3.75 ± 0.08s	30/30 55.73 ± 1.26 503.52 ± 11.77s	30/30 55.33 ± 1.0 502.31 ± 9.53s	10/30 55.9 ± 1.33 468.68 ± 11.5s	30/30 55.6 ± 1.14 497.67 ± 13.17s
triangle-tire-10	30/30 59.0 ± 1.11 5.15 ± 0.11s	30/30 62.13 ± 0.93 569.71 ± 8.67s	30/30 60.93 ± 0.86 560.39 ± 8.54s	6/30 61.67 ± 3.49 569.18 ± 46.65s	30/30 61.8 ± 0.98 592.03 ± 9.64s

Table 4.5: Results for Triangle Tireworld

4.3.6 Experiment Summary

We have described and analyzed the experimental results of our proposed methods to combine UCT with ASNets in detail.

In the Stack Blocksworld experiment, we demonstrated that using an ASNet-influenced action selection ingredients allows us to be robust to any misleading information provided by an ASNet, as the influence of an ASNet is scaled down as we apply what the network has suggested more often.

The Exploding Blocksworld experiment showed us how we can exploit the policy learned by an ASNet to achieve better results than plain UCT even though this policy was suboptimal. We argued that it is important to select the influence constant M empirically based on the planning problem we are dealing with.

In the two-way CosaNostra Pizza experiment, we found that using ASNets as a simulation function allowed us obtain more accurate state-value and action-value estimates in the search tree, and hence increase the coverage. On the other hand, UCT with `Simple`-ASNet or `Ranked`-ASNet action selection only lead to slightly improved results over plain UCT. We argued that this was because of the misleading estimates provided by the planning heuristic, and the state space which increases exponentially with the number of toll booths.

The one-way CosaNostra Pizza experiment demonstrated how we can account for changes in the environment by using search. We used the ASNet trained for two-way CosaNostra Pizza that always opts to pay the toll operator, and showed that using either ASNets as a simulation function or within UCB1 allowed us to achieve 100% coverage with the optimal policy on most occasions.

Finally, the Triangle Tireworld experiment allowed us to explore the robustness of combining ASNets and UCT when both already achieve good performance. We demonstrated that by using ASNet in UCB1, we were able to decrease the number of nodes expanded in the search tree as we do not need to apply actions that we know are inherently bad. Moreover, we argued that it may not be worthwhile using ASNets as a simulation function if we can only perform a very limited number of trials in the time limit per planning step.

Thus, by combining ASNets with MCTS, we can learn what we have not learned, improve suboptimal learning and be robust to changes in the environment and domain. As such, we get the best of both worlds.

Chapter 5

Conclusion

In this final chapter of the report, we will summarize the contributions we have presented, and the results we were able to achieve. We will then discuss promising directions for future work.

5.1 Contributions

The main goal of this research project was to investigate how we could improve upon the generalized policy of an ASNet by combining it with UCT. The key contributions we made were:

1. **An Ingredient-Based Framework for UCT**.

   We introduced an ingredient-based framework, extended from THTS, from which we were able to generate different flavors of UCT including those that exploited the generalized policy learned by an ASNet.

2. **Using ASNets as the Simulation Function in UCT**.

   We introduced Stochastic ASNets and Maximum ASNets as simulation functions which can help us obtain much more accurate state-value estimates than those computed by a domain-independent planning heuristic.

   We also discussed when it is worth and not worth using ASNets as a simulation function, and confirmed these hypotheses in our experiments which showed that UCT would converge to the optimal policy much faster if the knowledge learned by an ASNet is informative.

3. **Using ASNets in UCB1**.

   We introduced two new action selection ingredients, `Simple`-ASNet and `Ranked`-ASNet, both extensions of UCB1. By introducing the new ASNet term, we can force UCB1 to explore actions that an ASNet wants to exploit, as we 'reward' actions that have a higher probability in the policy learned by the ASNet.

   Moreover, the new ASNet term allows us to decay the influence of an ASNet over time as we apply the action it has suggested more often. This means that UCT is robust to any misleading information provided by an ASNet, as

we demonstrated in our Stack Blocksworld experiments. This is in contrast to Stochastic and Maximum ASNets, where UCT may not be robust to any misleading state-value estimates obtained through the ASNet-based simulations.

We also provided a sketch of a proof that shows UCT with `Simple`-ASNet or `Ranked`-ASNet action selection converges to the optimal policy, and discussed when we should use one over the other.

4. **Empirical Evaluation**.

   We firstly gave a detailed description of the wide variety of domains we evaluated our algorithms on. We then analyzed the results of our experiments in significant detail, and showed that combining UCT and ASNets allows us to achieve the goals we described in Section 3.1, and hence obtain the best of both worlds.

5.2 Future Work

5.2.1 Improved Algorithmic Efficiency

The code for our UCT framework could be optimized to reduce the computational time required to perform a single trial. As a result, we would be able to complete more trials in the limited time we give UCT at each planning step, and ideally achieve better experimental results.

Another possible improvement would be to support running trials in parallel by using an asynchronous variant of MCTS [Silver et al., 2017]. This would allow us to distribute the workload across several CPUs and ultimately reduce the time required to converge to an optimal solution.

Rollout-based UCT also benefits from a state-value cache, in which past state-value estimates may be used to provide even more accurate estimates in future rollouts (also known as simulations) [Kocsis and Szepesvári, 2006]. Hence, we may potentially speed up the convergence of the state-value and action-value estimates to the true optimal values.

5.2.2 Mixed Simulation Functions

We introduced the mixed simulation function as a potential way to overcome the misleading state-value estimates obtained when using the Stochastic or Maximum ASNet simulation functions. By mixing random simulations with ASNet-based simulations, we are able to introduce more noise which may help us overcome the bad information provided by the policy learned by an ASNet.

We did not investigate mixed simulation functions because selecting a good level of mixing between random simulations and ASNet-based simulations proved to be very difficult. Ideally, this level of mixing should be learned or determined automatically by judging how useful the knowledge learned by an ASNet is for solving

our planning problem. It would be interesting to see how robust mixed simulation functions are in comparison to ASNet-based action selection.

5.2.3 Interleaving Planning with Learning

Leapfrogging is a strategy that allows us iteratively improve the performance of a learning algorithm by using the learning algorithm itself as a teacher.

Consider the Exploding Blocksworld domain. We could initially train an ASNet using the procedure described in Section 2.4.3 on problems that are relatively easy. Then, using the policy learned by the ASNet, we could run several rounds of UCT with ASNets on slightly more difficult problems. Using the trial paths accumulated from running UCT with ASNets, we then continue training the ASNet and hence further improve the learned policy. We can repeat this process for many iterations, and slowly increase the difficulty of the problems.

This would hopefully allow an ASNet to learn more useful knowledge about good patterns of actions in Exploding Blocksworld, and successfully handle problems with differing levels of difficulty.

Thus, UCT with ASNets effectively acts as a 'teacher' for training an ASNet. It is clear that we can incrementally improve the policy learned by an ASNet by interleaving planning with learning.

AlphaZero has successfully applied policy iteration through self-play to improve the performance of its deep neural network [Silver et al., 2017]. Junyent et al. [2018] use lookahead trees to learn compact policies for improving width-based planning.

5.3 Concluding Remarks

We have introduced several methods which can be used to combine generalized policies with UCT. Although we test our proposed methods using the generalized policies learned by an ASNet, our methods are applicable to any method of acquiring a generalized policy such as ROLLER [de la Rosa et al., 2011] and [Yoon et al., 2002].

Nevertheless, we have shown that combining the generalized policy learned by ASNets with UCT allows us to achieve the best of both worlds.

List of Figures

List of Tables

Bibliography

Auer, P.; Cesa-Bianchi, N.; and Fischer, P., 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47, 2 (May 2002), 235–256. (cited on page 27)

Bellman, R., 1954. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60, 6 (11 1954), 503–515. https://projecteuclid.org:443/euclid.bams/1183519147. (cited on page 6)

Bertsekas, D. P. and Tsitsiklis, J. N., 1991. An Analysis of Stochastic Shortest Path Problems. *Math. Oper. Res.*, 16, 3 (Aug. 1991), 580–595. doi:10.1287/moor.16.3.580. http://dx.doi.org/10.1287/moor.16.3.580. (cited on page 5)

Bonet, B. and Geffner, H., 2003. Labeled RTDP: Improving the Convergence of Real-time Dynamic Programming. In *Proceedings of the Thirteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'03 (Trento, Italy, 2003), 12–21. AAAI Press. http://dl.acm.org/citation.cfm?id=3036969.3036972. (cited on page 8)

Bonet, B. and Geffner, H., 2012. Action Selection for MDPs: Anytime AO* Versus UCT. In *AAAI*. (cited on pages 8, 16, and 31)

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P. I.; Tavener, S.; Perez, D.; Samothrakis, S.; Colton, S.; and et al., 2012. A survey of Monte Carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*, (2012). (cited on pages 2, 9, and 10)

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P., 2008. Monte-Carlo Tree Search: A New Framework for Game AI. In *AIIDE*. (cited on pages 10 and 71)

de la Rosa, T.; Celorrio, S. J.; Fuentetaja, R.; and Borrajo, D., 2011. Scaling up Heuristic Planning with Relational Decision Trees. *J. Artif. Intell. Res.*, 40 (2011), 767–813. (cited on pages 3 and 69)

Dean, T. L. and Boddy, M. S., 1988. An analysis of time-dependent planning. In *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988.*, 49–54. http://www.aaai.org/Library/AAAI/1988/aaai88-009.php. (cited on page 9)

Domshlak, C. and Feldman, Z., 2013. To UCT, or not to UCT? (position paper). In *SOCS*. (cited on page 10)

Estlin, T.; Gaines, D.; Chouinard, C.; Castano, R.; Bornstein, B.; Judd, M.; Nesnas, I.; and Anderson, R., 2007. Increased Mars Rover Autonomy using AI Planning, Scheduling and Execution. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 4911–4918. doi:10.1109/ROBOT.2007.364236. (cited on page 1)

Geffner, H. and Bonet, B., 2013. *A Concise Introduction to Models and Methods for Automated Planning: Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, 1st edn. ISBN 1608459691, 9781608459698. (cited on pages 1 and 16)

Ghallab, M.; Howe, A.; Knoblock, C.; Mcdermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D., 1998. PDDL—The Planning Domain Definition Language. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212. (cited on page 16)

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P., 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17782/16931. (cited on page 16)

Gusmão, A. and Raiko, T., 2012. Towards Generalizing the Success of Monte-Carlo Tree Search Beyond the Game of Go. In *Proceedings of the 20th European Conference on Artificial Intelligence*, ECAI'12 (Montpellier, France, 2012), 384–389. IOS Press, Amsterdam, The Netherlands, The Netherlands. (cited on page 27)

Haslum, P. and Geffner, H., 2000. Admissible Heuristics for Optimal Planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AIPS'00 (Breckenridge, CO, USA, 2000), 140–149. AAAI Press. http://dl.acm.org/citation.cfm?id=3090475.3090491. (cited on pages 8 and 9)

Helmert, M. and Domshlak, C., 2009. LM-Cut: Optimal Planning with the Landmark-Cut Heuristic. (cited on page 9)

Jimenez, S.; Coles, A.; and Smith, A., 2006. Planning in probabilistic domains using a deterministic numeric planner. In *25th Workshop of the UK Planning and Scheduling Special Interest Group*. https://strathprints.strath.ac.uk/3154/. (cited on page 9)

Junyent, M.; Jonsson, A.; and Gómez, V., 2018. Improving width-based planning with compact policies. *CoRR*, abs/1806.05898 (2018). (cited on page 69)

Keller, T. and Eyerich, P., 2012. PROST: Probabilistic Planning Based on UCT. In *ICAPS*. (cited on pages 11, 12, 16, and 19)

Keller, T. and Helmert, M., 2013. Trial-Based Heuristic Tree Search for Finite Horizon MDPs. In *ICAPS*. (cited on pages 3, 12, 13, 19, 22, 28, 29, 31, 32, 40, and 41)

Kocsis, L. and Szepesvári, C., 2006. Bandit based Monte-Carlo Planning. In *In: ECML-06. Number 4212 in LNCS*, 282–293. Springer. (cited on pages 10, 11, 27, 28, 29, 36, and 68)

Kolobov, A.; Mausam; and Weld, D., 2012. A Theory of Goal-Oriented MDPs with Dead Ends. In *Proceedings of the Twenty-Eighth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-12)*, 438–447. AUAI Press, Corvallis, Oregon. (cited on page 6)

Little, I. and Thiébaux, S., 2007. Probabilistic planning vs. Replanning. In *In ICAPS Workshop on IPC: Past, Present and Future*. (cited on pages 6, 9, 16, 45, 46, 62, and 71)

Russell, S. and Norvig, P., 2009. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. ISBN 0136042597, 9780136042594. (cited on page 13)

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D., 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529, 7587 (Jan. 2016), 484–489. doi:10.1038/nature16961. (cited on page 2)

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D., 2017. Mastering the game of Go without human knowledge. *Nature*, 550 (Oct. 2017), 354–. http://dx.doi.org/10.1038/nature24270. (cited on pages 68 and 69)

Toyer, S., 2017. *Generalised Policies for Probabilistic Planning with Deep Learning*. Bachelor's thesis, Australian National University. (cited on page 8)

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L., 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *AAAI Conference on Artificial Intelligence (AAAI)*. (cited on pages v, 1, 2, 13, 14, 15, 19, 42, 43, 45, and 71)

Trevizan, F., 2013. *Short-Sighted Probabilistic Planning*. Ph.D. thesis, Carnegie Mellon University. (cited on page 5)

Yoon, S.; Fern, A.; and Givan, R., 2002. Inductive Policy Selection for First-order MDPs. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02 (Alberta, Canada, 2002), 568–576. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. http://dl.acm.org/citation.cfm?id=2073876.2073944. (cited on pages 3 and 69)

Younes, H. L. S. and Littman, M. L., 2004. PPDDL 1.0 : An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. (cited on page 13)