Comparing Uniform, On-Policy, and MCTS Approaches in Tabular Reinforcement Learning

Cooper Golemme

April 7, 2025

1 Motivation

In reinforcement learning, there is a great split between *model-based* and *model-free* reinforcement learning. Model based reinforcement learning models the environment of the agent as an MDP and seeks to learn from the environment by exploring and updating its action value estimates. Model-free methods rely on planning, where the agent experiences a simulated episode from the environment and updates its policy before performing new episode generation. Both models prove to do well for tasks that are particularly suited to their strengths, but hope for a more general model lies in combining these two forms of learning. The goal of combining these methods is to create an agent that plan, act, and learn in its environment without human intervention. In this paper, we will explore different tabular based methods for agent planning and learning. We will explore trajectory sampling methods and a decision-time planning algorithm that combines many concepts of reinforcement learning to surprising results.

2 Introduction

To begin our discussion of tabular methods for agent planning and learning, we need to introduce a few key topics.

2.1 Trajectory Sampling

Trajectory sampling is the idea that we can distribute updates by sampling the state or state-action space according to some distribution. This is in contrast to the classic approach from dynamic programming, which sweeps through the entire state or state-action space and updates each state or state-action once per sweep. One obvious benefit of trajectory sampling is the computation time. Instead of updating all states in sweeping fashion, we update states as we encounter them along a trajectory. In effect, trajectory sampling tends to focus updates on regions of the state space that appear more frequently in practice.

It's like studying for a test from the teacher's study guide rather than indiscriminately reading the textbook.

The book gives the example of chess. Theoretically, the pieces could appear in very strange configurations on the board. Think all pawns move up 2 spaces on both sides. Such a configuration is valid, but in practice, this state and others like it rarely if ever come up in actual games. Instead of sweeping updates over the whole state-space which would include updates to states like these, trajectory sampling allows the agent to focus on refining its policy for states that are more likely to occur.

However, trajectory sampling is not perfect. It can cause the same parts of the state-space to be updated continuously, not effectively exploring the environment, leading to suboptimal solutions. Rare states are also a problem with trajectory sampling. Because the agent hast encountered the state much, suboptimal or unknown random behavior can occur. On possible solution which we will explore is decision time planning. When we encounter rare states, we process the environment in real time and assess possible actions multiple steps into the future before deciding on an action. We will go into more detail about this kind of planning in the next section.

2.2 Decision Time Planning

Up to this point, the kind of planning that we have discussed has been background planning. Background planning is done beforehand, usually through trajectory sampling and policy shaping, to inform table entries or function approximation parameters that the agent then uses at decision time. At decision time, all the agent has to do is look up in some table to determine the optimal action.

Unlike background planning approaches, decision-time planning focuses on a particular state, generating sample trajectories off the state, looking many states and rewards into the future before deciding what to do at the current state. This type of planning is well suited to tasks that don't require immediate feedback as it shifts the computation time to be in real time as the agent is interacting with the environment.

One important component to any decision time planning algorithms are *rollout-algorithms*. Rollout algorithms are based on Monte Carlo control algorithms applied to simulated trajectories that begin at the current environment state. These algorithms estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action then follow a given policy. Unlike Monte Carlo control algorithms, the goal of rollout algorithms is to produce Monte Carlo estimates of action values only for each current state and for given policy, rather than estimating a complete action-value function q_π for a given policy π. Rollout algorithms aim to improve the rollout policy rather than find an optimal policy. While, these policies aren't typically considered learning in the most literal sense because they do not retain information about their policy only inform the next action to take, they nonetheless provide a simple and powerful way to perform action selection.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search(MCTS) is a popular algorithm to make optimal decisions in games where future outcomes can be uncertain and complex but can be simulated [2]. Like many RL algorithms, it attempts to balance exploration of new moves and exploitation of known moves. It has previously been used in state of the art reinforcement learning algorithms to play complex games like Chess, Go and Atari [1].

MCTS is executed when the agent encounters new state to select a new action for the given state. Below, we show the four phases of MCTS and explain them in detail.

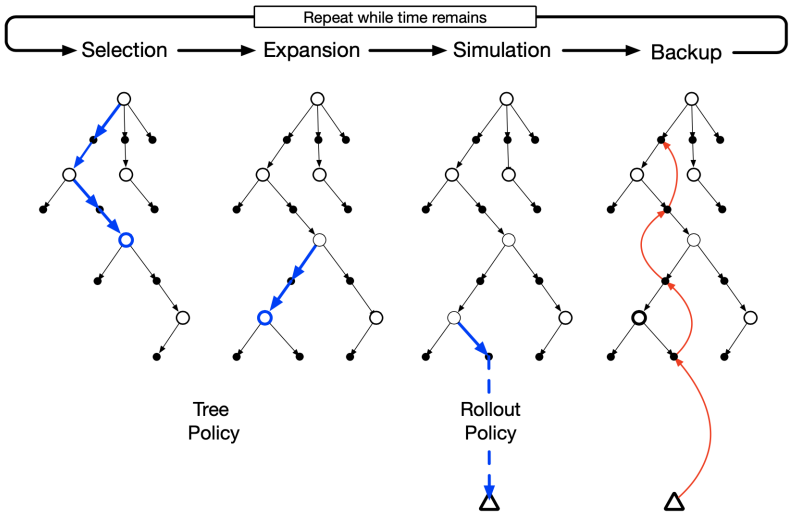

Figure 1: Figure showing the phases of Monte Carlo Tree Search: Selection, Expansion, Simulation, and Backup. We select actions according to a *tree policy*, expand the tree with new actions, follow a *rollout policy* when we reach a leaf, and update our action value estimates up propagating returns back up the tree.

Each iteration of MCTS consists of four steps:

1. **Selection.** During selection, we start from the current position at the root node of the tree. We then traverse the tree by repeatedly selecting child nodes until we reach a leaf node. We traverse by following a *tree policy*. In our application, we will select nodes according to the Upper Confidence Bound (UCB) formula. We need to shape the UCB formula for this context. The general formula for a given node i looks like:

$$UCB_i = \text{win rate of node } i + C\sqrt{\frac{\ln(\text{parent visits})}{\text{visits to node } i}}$$

2. **Expansion.** During expansion, the tree is expanded from the selected leaf node by adding one or more child nodes. This corresponds to adding new actions for the agent to explore in the future.

3. **Simulation.** From the selected node or the nodes added in expansion, we simulate a complete episode according to the rollout policy. From this, we get a Monte Carlo trial, where we first select actions in the tree according to the UCB policy and then generate an episode outside of the tree by the roll out policy.

4. **Backup.** From this generated trial in the simulation step, we have some reward. In the backup step, the reward from this trial is backed up the tree to update (in the case where we simulated from selected node in step 1) or initialize (from expanded node in step 2) the action values in the tree. For the actions generated beyond the tree, no values are updated or stored. A visualization of this is shown in Figure 1.

With MCTS, we continue these 4 steps starting at the tree's root to some time horizon T or until some other threshold is reached. When MCTS finishes, an action from the root node is selected which either: (1) has the highest visit count, (2) has the highest action value estimate, or (3) some combination of the two. After selecting this action, the agent the proceeds to the next action, performing MCTS over again to select a new action. We will experiment with different ways to select this action to best fit our application to playing chess.

3 Experimental Design

To evaluate and compare different planning methods, we will create various Markov Decision Processes (MDPs) with adjustable parameters to test our methods and our hypotheses. We will then implement our planning methods in these environments and compare them against eachother and themselves using simple metrics. In all of the experiments, we use the average reward value as a way to compare.

3.1 Trajectory Sampling Experiments

3.1.1 Branching Factor of 1

In this experiment, each MDP consisted of 10,000 states with a branching factor of 1, simulating sparse transitions to isolate the impact of planning efficiency. Two key update strategies were compared: (1) uniform updates, where state-action pairs are selected uniformly at random for value iteration, and (2) on-policy updates, where updates prioritize states encountered under the current ϵ-greedy policy ($\epsilon = 0.1$).

To account for task variability, results were averaged over 200 independent MDPs. Performance was measured by tracking the value of the start state under

the greedy policy, with convergence compared against an optimal baseline derived from value iteration. We used a discount factor ($\gamma = 0.9$) to ensure future rewards were not too heavily weighted, and computation time was quantified in terms of expected updates. We hope that this experimental framework isolates the trade-offs between exploration breadth (uniform) and policy-directed efficiency (on-policy), providing insights into scalable planning in large state spaces.

3.1.2 Branching Factor of 3

After evaluating uniform and on-policy updates with a MDP branching factor of 1, we will retry the experiment now with a branching factor of 3. We hope that this will show the limitations with on-policy updates. We hope to show that the on-policy updates more quickly update the value estimate for the starting state, but in the long term, converge to a less optimal solution than the uniform updates. We will run both of these experiments in the same environment described above, with 10,000 states and an epsilon greedy policy, so that we can directly compare the results.

3.2 Decision Time Planning Experiments

We will then shift our focus from trajectory sampling to decision time planning. We will create an environment that allows for a simple implementation of MCTS, so we can experiment with different parameters.

3.2.1 Environment

We will create an MDP that is organized as a tree. The root state of the tree is the starting point and each internal node has a fixed number of children determined by a **branching factor**. The leaves of the tree indicate terminal states and have no children. The leaves are the tree are located at a specified **depth** parameter.

To model transitions between states, we will have actions and rewards associated with each node in the tree. When we are at a given node in the tree, we can choose between **branching factor** number of actions. Each action leads deterministically to one of the children of the node we are at. When we perform an action, the reward we receive is sampled randomly from a uniform distribution from $[0, 10]$

3.2.2 Experiments

In these experiments, we create an agent that uses MCTS to plan at decision time guiding the agent through our tree structured MDP. We will not explicitly compare MCTS to the on-policy or uniform updates, instead focusing on how MCTS performs by measuring the cumulative reward that the agent receives over an episode. In these experiments, we will consider various parameters and assess their importance to MCTS

1. **MCTS Iterations per Decision**: We will experiment with varying the number of iterations that MCTS performs each time it needs to decide on an action to take. We expect that more iterations per decision will yield higher episode returns because it allows MCTS to search deeper and more thoroughly through the action space.

2. **Rollout Depth**: We will experiment with varying the rollout depth, which tells the agent how many actions we should simulate in the simulation phase of MCTS. Theoretically, longer rollout depth should capture longer term effects of decision making and thus improve decision quality.

3. **Exploration Constant**: We will experiment with the exploration constant in the UCB formula, which balances exploring new actions and exploiting high value known actions.

Additionally, we will change the MDP environment. We will test MCTS with different depth and branching factors to assess how the optimal values for the variables we enumerated above will change based on the MDP environment.

4 Results

4.1 Experiment 1 Results

The first experiment that we ran was to compare the on-policy to uniform updates in an MDP environment with branching factor 1. With this, we hoped to show that the on-policy updates lead to faster planning initially, but sub-optimal planning in the long run.

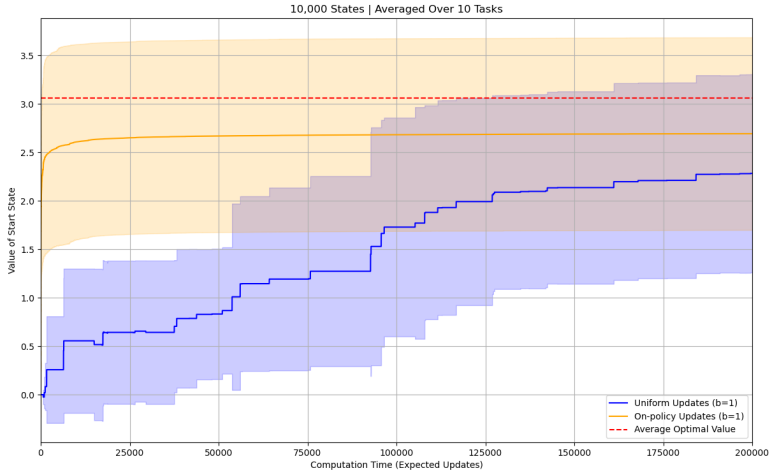

Figure 2: Figure showing the estimated value of the start state after a given number of computations for the uniform and on-policy updates in the branching factor 1 environment.

From the results of the experiment, shown in Figure 2, we observe that it is indeed the case that the on-policy updates learns quicker initially, as evidenced by the sharp jump in the orange colored line at the start. We did not, after 200,000 updates, observe that the uniform update line crosses the on-policy update line, which would indicate that the on-policy updates results in sub-optimal planning in the long run. When we extrapolate the curve, however, it seems reasonable that it would eventually converge to the same value as the on-policy or even surpass.

The results make sense with what we understand about the experimental setup. When the agent updates its policy, while following the same policy, it is likely to find some optimal path very quickly due to the low computational effort of updating values along a single policy, but in the long run, it could potentially converge to some suboptimal solution due to it not thoroughly exploring the environment outside of its policy. This effect should become more pronounced with a greater branching factor, which is what we will test in the next experiment.

4.2 Experiment 2 Results

In the second experiment, we compared the on-policy and uniform updates in a new MDP environment – this time, one with a branching factor of 3. We hoped to see experimentally the on-policy initially perform better than the uniform, but for the uniform update to quickly pass the on-policy update method in terms of the value of the start state. We believed this because with a higher

branching factor, there are are more states that are available to the agent after it acts, meaning there is a greater state-space to explore, which the on-policy update method will not do as well at as the uniform update method.

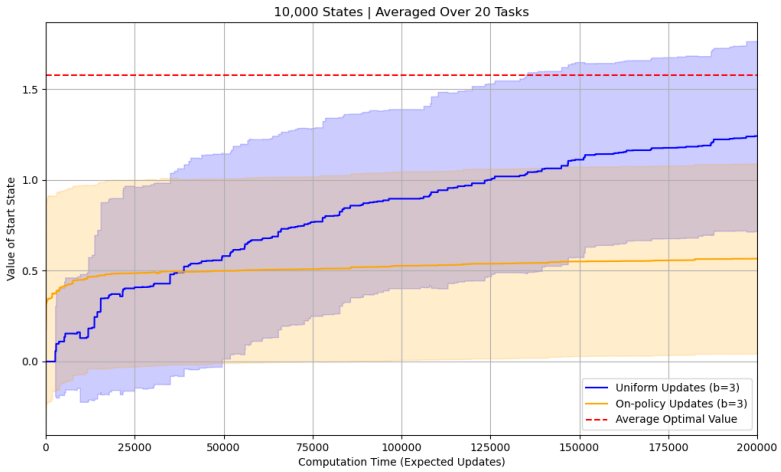

Figure 3: Figure showing the estimated value of the start state after a given number of computations for the uniform and on-policy updates in the branching factor 3 environment.

The results of the experiment support our hypothesis. With a higher branching factor, we still observe the on-policy update method performing better initially, yet relatively quickly (after about 40000 updates) the uniform update method surpasses the on-policy.

4.3 Experiment 3 Results

The results of the experiments roughly align with our intuition. The results of varying the number of iterations per decision, rollout depth and exploration constant are shown below. We tested across different environments to see if we noticed the same trends when varying the different parameters.

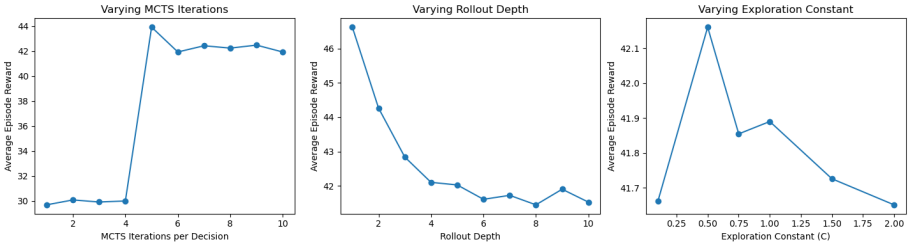

Figure 4: Comparing parameters for MCTS in our tree MDP environment with depth 9 and branching factor 4.

We noticed that when we increased the number of iterations of MCTS per decision, we experienced higher average episode returns. This makes sense. If at each step we take more time planning, we expect that we will have better results. This is what we noticed from experimentation.

The rollout depth at first appears counter intuitive. We would expect that increasing the rollout depth would allow for MCTS to have better results because it is able to plan further into the future, but we noticed the opposite trend; the shorter the rollout depth the higher the average episode return. One possible explanation is that the environment has a high discount factor ($\gamma = 0.9$), meaning that the most informative rewards are those obtained in the near future rather than far into the future. A shorter rollout might capture these more reliably without experiencing diminishing returns of far-future rewards.

The exploration constant experiment makes sense with intuition. We expect that a "good" exploration rate is somewhere between either extreme. We want to balance exploration and exploitation. Perhaps with higher branching factors it may be more beneficial to explore more, but for this experiment, around $C = 0.5$ appears to be the best.

We also hope that if we were to change the environment by changing the depth and the branching factor, that we would observe similar trends. In the next figure, we tested in an environment with depth 5 and branching factor 10.

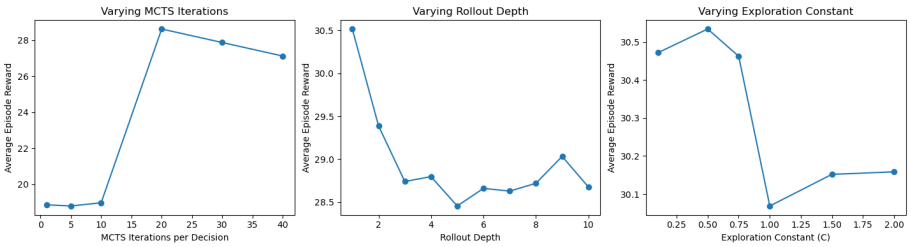

Figure 5: Comparing parameters for MCTS in our tree MDP environment with depth 5 and branching factor 10.

Figure 5 and Figure 4 are quite similar across all three parameters. Increas-

ing the number of iterations per decision increases the average reward in both, the optimal rollout depth is quite small in both, and a good exploration constant is about 0.5. This is what we expected to see because we believed that the factors that contributed to Figure 4 are similar to 5. Changing the environment slightly should not change the underlying trend we observe.

5 Conclusion

Through comprehensive experiments using tabular MDPs, we demonstrated that while on-policy updates can quickly improve performance in the short term by focusing on frequently encountered states, they risk converging to suboptimal policies when exploration is limited. In contrast, uniform updates offer more robust long-term performance through broader state coverage, but with slower initial improvements. We then shifted our focus to decision-time planning using Monte Carlo Tree Search (MCTS). We revealed that increasing iterations generally enhances performance, but optimal results hinge on carefully tuning parameters such as rollout depth and the exploration constant. Notably, the experiments suggest that shorter rollouts can be more effective in environments with high discount factors, emphasizing the importance of maximizing near-term rewards in such environments. These insights not only validate our experimental design but also underscore the potential benefits of integrating diverse planning approaches to develop more versatile and resilient learning agents.

References

[1] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1995.