AAAI Press Formatting Instructions for Authors Using LaTeX — A Guide

[U]niversity of Kentucky
athan.johnson@uky.edu

Abstract

In this paper I go over the process I went through to complete assignment 1, which covered the multi armed bandit problem and several solutions to it, the minimax algorithm with alpha beta pruning, and Monte Carlo tree search. For multi armed bandits I compared the results of a random agent, ϵ-greedy, UCB1, and Thompson sampling. For minimax and Monte Carlo I ran them on a game of connect four, comparing them to each other and a random agent.

Introduction

For the ϵ-greedy agent I chose to keep epsilon constant at 0.1. When run on test zero which consisted of ten different bandits with values relatively close together, it showed an average regret of 349 after 10 runs of 10,000 iterations. When it was run on test one however, which consisted of nine values of 0.2 and one of 0.9, it had an average regret score of around 743. Clearly it performs better in situations with more ambiguity, however I suspect that if I used a decaying ϵ instead of a constant one it would have performed much better here. It's worth mentioning that in test zero it outperformed UCB1 but in test one it did not. I also chose to store the history in the agent itself, updating the history of each arm at the be-ginning of each iteration of epsilon greedy so that I didn't have to iterate through history again each time. This proved to speed things up significantly. When implementing the UCB1 agent I again updated the history with the past event first thing each iteration. The UCB1 agent performed a little worse than ϵ-greedy on test zero, having an average regret of 522 on this test with many arms close to the optimal one. However, it is worth mentioning that a few of the ten tests run UCB1 came out on top. It majorly outperformed ϵ-greedy on test one with an average regret score of 211. This is three and a half times better than epsilon greedy on this test. Thompson sampling proved to be the best option in both cases. For this agent I also chose to update the history at the start to speed up the process. For test zero Thompson sampling had an average of 111, which dramatically out-performed ϵ-greedy and UCB1. It performed even better with test one, having an average score of only 33.

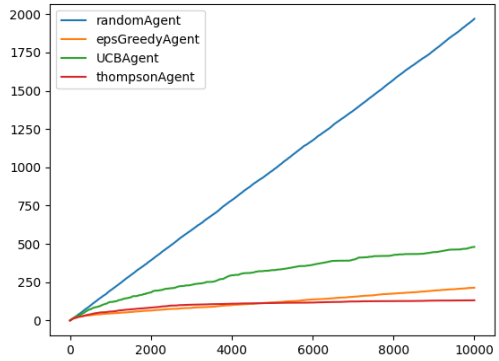

Figure 1: Here are example results for each agent compared on test zero. You can see UCB1 struggles compared to ϵ-greedy, and Thompson sampling comes out on top.

Minimax

For the minimax agent, I spent a long time to try and implement my own solution, but I couldn't get it to work correctly. I ended up choosing to use ChatGPT to do most of the code writing while I edited the code it made to work for me. The agent performs more poorly than expected, being unable to consistently win when it has three in a row, however this was consistent across both my unfinished and ChatGPT's implementation. One interesting thing to note is that it was much more consistent at stopping other three in a rows than completing its own. Instead of counting the number of steps completed I counted the depth of the search, going to a depth of 5 to stay in the limit of thirty thousand steps since 74 comes out to twenty-four hundred, which is below the max threshold of then thousand. When implementing the minimax agent, I chose to implement it with recursion and to record the actions taken and step back through them each recursive step. This greatly slows down the speed, but I was unable to get copy's deepcopy to work. I also used many simple helper functions made by ChatGPT to make the code more read-able. The whole recursive function itself has a wrapper function. Each of the three options, being random agent, minimax agent, and Monte Carlo tree search, all take

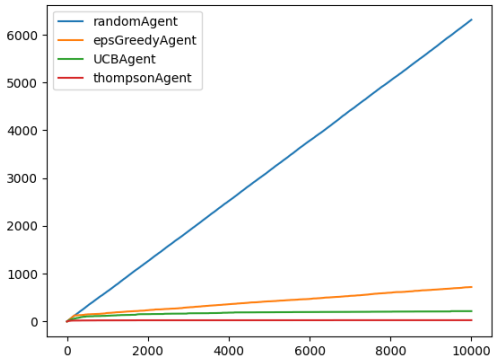

Figure 2: Here are the results for each agent compared on test one. Here Thompson dramatically outperforms all other agents, but UCB1 performs much better than the previous test while ϵ-greedy performs far worse in this test.

in the same arguments but they throw away what they don't need, this is because it is simpler to change which agents are being used without having to change any of the arguments. Against the random agent, the minimax agent one four out of five games when going first. When going second it won three, tied one, and lost one game.

Monte Carlo Tree Search

The Monte Carlo agent I also chose to use ChatGPT to help create, given its success with minimax. This time I set the max iterations to ten thousand instead of using depth. I also had a node class that was designed to be used for Monte Carlo which helped take care of the complexities of tree building. Monte Carlo runs significantly slower than minimax, however it is using all ten thou-sand nodes each iteration so this is expected. The Monte Carlo algorithm itself makes random choices until reaching the bottom of the tree. When run against the random agent it only won three out of five times when going first, but only won one game when it went second. Against minimax it performed worse, when going first it lost four and tied one game. I suspect that this poor performance may be due to a rare bug that I wasn't able to root out in the code, which occasionally results in the Monte Carlo agent choosing to place in a column that has already been filled, which the environment counts as an instant loss.

Acknowledgments

ChatGPT did much of the code writing for the Minimax and Monte Carlo tree search part.