Luke Doukakis - 011847368
Professor Markus Eger
CS 4200.01

Lab 2: Monte Carlo Tree Search - Report

Implementation

To implement the tree for the MCTSPlayer, I created a simple tree object that links together nodes which contain action sequences, expected values, and references to their parent and children nodes. The MCTS_N loop within the get_action method of MCTSPlayer works with an instance of this tree, traversing and expanding it with each iteration.

The tree traversal and expansion is done by keeping a list of expansion candidates (nodes which have not yet been expanded), and chooses the candidate with the highest expected value. This node is then expanded, creating a new node for each of the game state's possible actions. Each new node will inherit its parent's action sequence, with our new action tacked on, and then rolled out. The rollout for the new node is done by a modification of the RolloutPlayer which I called SequencePlayer: a player that performs actions according to the sequence of actions it is provided, then acts randomly when it finishes that sequence. This way, when a node deep down in the tree is to be rolled out, a game could be simulated with all the appropriate actions taken until after that point, then followed by the random rollout strategy to produce an expected result. After the children are rolled out, their expected results are back-propagated up the tree through parent referencing, the children themselves are added to the expansion candidates, and the loop is repeated once more for MCTS_N iterations.

After MCTS_N iterations are complete, an action is then chosen by comparing the expected values of the all the root node's children. Each of these expected values should have been tempered by the back-propagations of all its descendents, so that they represent an average of all the possible scores from games explored by that branch of the tree. The action corresponding to the highest expected value is selected and returned by the get_action method.

Performance

The MCTS agent in general performs better than TimidPlayer but worse than BasicStrategyPlayer. For the default deck type, I was not too surprised to find that it didn't stack up to the tried and true basic strategy, getting an average score of -0.37 to basic's -0.09. Where I expected it to do better than basic strategy was in the differing deck types. MCTS is a strategy that should allow for adaptation to differing conditions, and my implementation seemed to do so reasonably well, but not as well as I wanted. For -d high, it performed well, averaging at .25 to

basic's -.57. This makes sense, as since card values are higher, the game ends in fewer moves, so the MCTS agent will have explored more game states relative to the total. In -d low, the MCTS also did well. Even though the agent could not explore as many game states relative to the total, it still came through against the basic strategy because it was able to adapt to the changing conditions. In -d even, -d odd, and -d random however, MCTS was handily defeated by the basic strategy.

MCTS_N

I was glad to see that the higher the MCTS_N value was, the better the agent scored on most deck types. This showed me that my implementation of the tree traversal, expansion, and back-propagation was working to come up with a better action decision than not doing so. Again though, the -d even and -d odd deck types threw me for a loop, where my lower valued MCTS_N agent outperformed the higher one. The graph below shows the performance of each of the agents I tested: MCTS where MCTS_N = 1000, MCTS where MCTS_N = 10, basic, and timid.