Arnav Khera

Gordon McCulloh

Assignment 3 for CSE 415, Spring 2021, University of Washington

Deterministic Simplified Backgammon Agent (DSBG)

* Gordon completed the structure of the code including all subfunctions and the minimax algorithm. He also modified until it passed all but one of the test cases. Arnav edited the code by pointing out an error where Gordon forgot to clear the number of states and cutoff nodes in the useAlphaBetaPruning( ) function, which fixed the error in the final test case where the program was expanding too many nodes.
* The static evaluation function works by first counting the number of positions away from the goal state each checker is for both adversaries, then summing the total number of positions and counting them against each player. In theory, both players want fewer checkers left on the board, and they both want their own checkers closer to the goal state. We liken this to a one-dimensional Manhattan distance. Then, the static evaluation function looks at one player’s checkers and counts the number of its adversary’s checkers within twelve positions (i.e. two die rolls). These are all potential checkers that could bear off the former’s checkers, and they are counted against the first player’s score. Since bearing off represents a very poor outcome for the player on the receiving end, these bearing off checkers are weighted threefold. Next, the static evaluation function takes advantage of the built-in board state by counting the number of checkers each player has on the bar and counting these against them. The bar is particularly undesirable, so checkers on the bar are weighted by 25. Lastly, the static evaluation function counts the number of checkers each player has off the board and weights these 50-fold in favor of each player, because more checkers off the board is necessary for winning the game.
* For alpha-beta pruning, we stored the best moves in a dictionary. The ordering of successors was automatically completed by the pruning algorithm and cleared with each move. It was important to define the number of states kept by the minimax search algorithm in order to count the number of cutoff states.
* In order to implement this code, we felt it best to create another subfunction which creates a list of all legal moves according to the rules of Backgammon, because this made it easy to expand each node and explore legal outcomes without expanding pass states. To do this, it was key to count winning states and only add one pass state per search.

Stochastic Simplified Backgammon Agent (SSBG)

* Arnav completed the structure of this code and carried out the modification of the DSBG code to completion.
* In order to implement the expectiminimax algorithm, the same structure of the minimax algorithm was used, but the expected value of each die roll (for two dice) was backtraced to 1/36 for each (die1, die2) combination with replacement. This gave the expected value when evaluating forward in the state space.

Partnership Retrospective

* Gordon: I struggled at first with thinking about how plyLeft would operate in expectimax, but Arnav cleared up how to understand the pseudocode given in class for me. I also learned the importance of having a second set of eyes on a long program, because I submitted by dbsg code 13 times before Arnav quickly found the missing reset.
* Arnav:

Optional Insights

* We found it interesting that different weight systems for the heuristic function guaranteed at least a 10% monotonic increase for each improved state. This only resulted with a high-enough weighting of checkers on the bar, which showed that designing a good heuristic is iterative and highly problem-dependent. This required a bit more knowledge of the game than expected.