Authors
krudenko@ucsd.edu
a9dang@ucsd.edu
kdduong@ucsd.edu

CSE 150: Assignment 2

Introduction

Game-playing programs is a critical field in advancing Artificial Intelligence in other areas. Games, such as chess, checkers, tic-tac-toe, and gomoku, provide us with a problem and a set of rules that define the environment. Devising a clever AI that embodies a specially designed algorithm/solution to the game provides clues as to solving other problems in the real world. This is what makes the field of game-playing AI fundamentally interesting besides the idea that computers can serve as challenging opponents to humans.

Most game-playing programs utilize a Minimax algorithm to construct a search tree of possible states/moves. The dilemma that Minimax algorithm based game-playing programs face is the large game tree it has to search through for the best move. How can we improve the time efficiency of the AI's search for the best move while maintaining its accuracy? In this paper, we attempt to explore the implementation of the minimax algorithm, but most importantly, how the strategies of alpha beta pruning, an evaluator function for the purpose of move ordering, transposition tables, and iterative deepening improves Minimax's search time.

Description of Problems and Their Algorithms

Problem 1 delineates the need for an algorithm that allows the AI to play a game of perfect information. In other words, the AI must not only consider their goals in the decision of their moves, but the goals of their opponent as well. We chose to implement a highly popularized game-playing algorithm called Minimax. In short, the algorithm has two players called Max and Min where Max is the current player and Min is the their opponent. Max makes moves that maximize the value of the game while Min attempts to minimize the value. The values are determined by a utility function within the player class. The higher the value the more advantageous it is to the Max player.

Problem 2 asks us to improve Minimax’s search time for the best move from problem 1. We understand that the search tree for a game can be very huge. We also understand that there is no need to search down a branch that would probably not give us the best advantageous move. Therefore, it would make sense to attempt to decrease the number of elements evaluated by the minimax algorithm in its search tree. We don’t need to evaluate every node! Our solution to problem 2 involves the implementation of alpha-beta pruning in conjunction with the minimax algorithm. Alpha-beta maintains two values that represents the maximum value the max player can acquire (alpha) and the minimum value that the min player can acquire (beta) in one branch. If the value of a recently explored node becomes worse than alpha, Max will “prune” that branch. Max will stop considering the node’s children. The same goes for Min.

Problem 3 asks us to devise a new utility function for our AI that “evaluates” for the longest legal line in the current board. The AI will use this evaluator function to determine the next best move. It does this by recording the longest, unbroken streak horizontally, vertically, and diagonally for a given color (player). For oversight purposes, solving problem 3 is important for problem 4 because the evaluator function will be used to further improve our alpha-beta pruning algorithm’s search time. Sometimes pruning whole branches in the search tree is not enough to improve efficiency when the AI is dealing with large game boards. The evaluator function capitalizes on the idea of move order because it provides an estimation of the best branch to explore without attempting to search through the whole branch to the terminal state. In other words, we can search a branch that provides us with the best chances of getting the most advantageous move first.

Problem 4 was a fun problem because it requires the implementation and integration of several different strategies to improve Minimax’s search time in a totally new custom player. We decided to use alpha-beta pruning, an evaluator function, transposition tables, and iterative deepening. The first 2 strategies were already discussed above. The idea behind transposition tables is that they sacrifice memory for speed. It allows us to store states we’ve already visited. When we attempt to iterate over visited states due to the nature of iterative deepening, we don’t have to evaluate for their utility again saving search time. Iterative deepening as a strategy for this problem is powerful because as the algorithm iteratively goes deeper into their search tree it will get a more accurate estimate of the nodes. Additionally, it prevents cases where the algorithm wastefully searches too deep in a branch that does not contain the solution (a dilemma Depth First Search faces). Here's a quick explanation of how our approach works for problem 4. The algorithm iteratively searches down the search tree for the utilities of all possible moves of the current board. If the time limit or the depth limit hasn't been reached, we determine the nodes' utility via the utility function. For the latter, we determine the nodes' utility with the evaluator function. During this whole experience, the alpha-beta pruning Minimax algorithm determines the best move for the player while eliminating branches we don't need to search to improve search time. Lastly, the transposition table save the states we visited.

Unfortunately, we weren't adventurous, so we didn't explore other approaches. We are the “if it ain't broke, don't try to fix it” folks.

Maximum Number of Empty Squares for Which the Agents Can Play

Minimax: 12 squares – 60 seconds for first turn
Alpha-Beta: 12 squares - 20 seconds for first turn
16 squares – 60 seconds for first turn
Custom Agent: 16 squares – 50 seconds for first turn with depth of 8

Irrational Decisions
The custom agent does some irrational moves sometimes against
5x5 board with K = 4. We believe the reason for the irrational moves involves evaulator function. It does not complete consider moves that blocks the win of the opponent. We feel that adding more attention to this heuristic calculation will improve the efficacy of the AI's ability to determine the best move. This is problem that we would love to tackle in the future, and we hope to publish our findings soon!

VS	3	4	5	6	7	8
3	X	10:0:0	0:10:0	0:0:10	0:0:10	0:0:10
4	10:0:0	X	0:10:0	0:0:10	0:0:10	0:0:10
5	10:0:0	10:0:0	X	0:0:10	0:10:0	0:0:10
6	10:0:0	10:0:0	10:0:0	X	0:10:0	0:10:0
7	10:0:0	10:0:0	10:0:0	0:10:0	X	0:10:0
8	10:0:0	10:0:0	10:0:0	0:10:0	0:10:0	X

In the presented table numbers are the maximum depth of iterative deepening. The stats of the games are presented in a way of Wins:Draws:Losses. In column*row proportion gives us the score when column has the first turn, for example 5x4 score is 10:0:0 which means that 5

boated 4 all 10 times when had the first move. Opposite to that 4x5 gives us 0:10:0, which means that 4 came to a draw with 5 all 10 times when 4 had the first turn.
Games between strong AIs (with big depth) such as 8vs7 showed that game can be played to a draw with relatively big number of plies known for every turn. When strong AI meets a weak one then the weak one uses evaluation function much longer than the strong AI, which already can reach terminal states at the end of the game. Thus AI with small deepening can be trying to reach a local maximum when stronger already knows which moves will leas him to the win.
As you can also see the first move plays a big role to detect a winner because it limits possible winning outcomes for the second player. For example 4vs5 ended with 10 draws, while 5vs4 ended with 10 wins by 5.

Author Contributions

Kevin Duong

I worked on problems 1-3, testing on problem 4, and a good amount of this write up. The things I have learned in this assignment would be the different approach to puzzles and problems when more than one side is involved like the last assignment; this assignment has two sides. I have learned more of Python this assignment, as well as a different way to look at Minimax in a game and how to store and search visited nodes. I am curious what more than two sides would be like.

Andrew Dang

I worked on problems 1, 3, 4, and the write up. This assignment has taught me quite a lot about how game algorithms work. I wished we were able to finish this assignment earlier before the midterm because it would probably help me a bunch on that one midterm problem! What I took away from this assignment is the strategies used to make our AI faster and smarter. For example, alpha-beta pruning helps our AI search for the best move faster by ignoring whole branches that would have been wasteful to search otherwise. Iterative deepening helps keep search at shallow depths so that if the best move is closer to the root, we could find it faster. A depth first search might end up taking our AI too deep in one branch that probably doesn't have the solution. Transposition tables trade space for speed. It allows our AI to be aware of states it has already searched so it doesn't have to spend time calculating the utility of those moves again! I'm still trying to understand how we can make our AI smarter, and I'm pretty sure most of this falls upon how we evaluate value in the moves.

Kirill Rudenko

I worked on problems 1-4 implementing different variations of minimax, alpha-beta, and custom player. I also worked on the write up! What I took away from that assignment is that increasing depth of search in iterative deepening doing problem our agent becomes really smart and outperforms other agents with smaller deepening. Using of a transposition table in alpha beta and in the custom player really increase the time our program is running. It happens because there are a lot of moves that can lead to the same terminal state, and saving this state is increasing the speed when utility will be obtained by the root.