Aron Hommas, Treker Battig

05/07/17

Intro to Artificial Intelligence

AI and Chess

Chess and chess playing computers have long been a point of interest among both the computer science and chess playing communities. Computer scientists wishing to invent better and more efficiently thinking machines. Players wanting to test their mettle against the ability of a machine. Near the beginning of all this, computer chess pioneer, Claude Shannon, laid some groundwork for the field [1]. He prophesied that there would be two general strategies towards designing a computer to play chess. Strategy A is the brute force approach. This is using a minimax algorithm as many plies deep as possible in order to find the best move. Strategy B is to take a more quiescent approach that looks at only a few of the most promising moves but runs many plies. Shannon states in his paper that strategy B will likely be more effective since the processing time required for strategy A is prohibitively long in any real gameplay scenario. However, what Shannon did not consider is improvements in technology that far surpass what he deemed generous at the time and domestications to the brute force approach.

Shannon foresaw that strategy B would be the dominant style of computer chess. However, modern chess programs overwhelmingly use the strategy A approach when playing chess. There are 2 primary reasons for this. Computational power increase and tree pruning techniques. Shannon assumed that a computer might be able to consider a million moves per second [1]. To look 3 full moves ahead (a full move being 2 plies) assuming 30 or more moves per turn (30^6), the calculation would take longer than 12 minutes to compute. This is an understandably prohibitive amount of time. Since the paper, computation rates have vastly increased. In 1997, IBM’s super computer Deep Blue was capable of considering 200 million possible chess positions each second [4]. This speed increase reduces the time required to look 3 full turns ahead to just 3.6 seconds which is a much more reasonable amount of time.

The other reason that applying Shannon’s brute force strategy A has emerged dominant is because not all board states actually have to be looked at. This is due to alpha-beta pruning and other improvements such as move ordering. Alpha-beta pruning is a technique to prune nodes during a minimax search. A max node is pruned if its value is less than the current alpha value. A min node is pruned if its value is greater than the current beta value [5]. In this way all nodes representing board states that would never be played, assuming optimal play for both sides, and so have no effect on the outcome of the search are never considered. There is no disadvantage to using alpha-beta pruning; it is a strictly beneficial operation. If the very first leaf node the minimax algorithm looked at was along the branch that the minimax algorithm picked, then alpha-beta would be able to do a lot of pruning and the search would run extremely fast. Techniques such as node ordering and killer heuristics help estimate moves that should be searched first and other ways to increase the effectiveness of alpha-beta pruning [3]. Through the use of pattern recognition and other such heuristics, potential moves can be scored and sorted so that nodes with the greatest potential to allow pruning are investigated first [5].

Besides the search algorithm, the most important portion of a chess engine is the heuristic used to score the boardstate of the leaf nodes. The most basic of heuristics will provide each piece on the board a score and add the players pieces and subtract the opponent’s pieces. A long time staple set the worths as such: pawns are 1 point, knights and bishops are 3 points, rooks are 5 points, the queen is 9 points, and the king is invaluable (often implemented as a really high number). This heuristic influences the program to play a game focused on gaining material either by good trades or capitalizing on mistakes of the opponent. However, this heuristic will have no qualms about moving a queen into the corner and leaving her there. Position is a very important factor in chess. A piece, such as knight or queen, are much more valuable to have in the center of the board because of the increased board control and options provided by the piece. This suggests providing conditional piece points depending upon location. For example, the heuristic could score a queen in the middle as 9.1 while a queen in the corner is scored as 8.9. This influences the program to play a materialistic game with a preference toward positional play if there is no material difference between board states. By continuing to tinker with the heuristic further modifications and improvements can be made on the computer’s gameplay.

Since a computer can only plan as many moves ahead as it is capable of calculating, it is very difficult for a computer to make good opening moves. This can be partially remedied by storing a database of book openings. This trades significant amounts of memory to have access to opening plays that have been thoroughly researched outside of the current game. This allows the computer to compete with a human player’s research and prior experience in the beginning stages of the game. Due to the branching factor of chess, it is not feasible to store openings too deep into the game. Similar databases can be kept for end games. When entering an endgame scenario, there is often already a side that has “won” if play commences perfectly. However, these solutions can run so deep that the 50 move rule may produce a draw before a side can force a checkmate! Storing endgame bases counteracts the calculation time, but can require significant amounts of memory. Many 6 piece end game variations can take more than 4 GB of memory in a compressed state [6]. Thus the usefulness of external endgame evaluation is limited and often databases containing only 5 or fewer piece endgames are used.

An interesting facet to consider is how the algorithmic move search reveals itself in gameplay. In other words, how does a computer act that is different from humans. The most noticeable aspect of computer play is that a computer does not make any horrendous blunders. This is a benefit provided by the exhaustive search algorithm. This trait on its own wins many games. Even in high level games human players make great mistakes, however, in games between humans these mistakes are often made due to distractions and so are not noticed. This is not true against a computer. A computer may not play every other move as well as a human, but a computer will never miss a mistake that a human player makes [2].

The other big difference between human and computer players, is that computer players have solid long term plans. A computer’s plan is only to get the best possible board state n plies deep, where n is the depth the computer is capable of reaching. This results in a possibly shortsighted gamplay and transient plans that change from move to move. Early opponents of computers were able to capitalize on this weakness by playing a slow positional game with distant move benefit beyond the computer’s horizon of computation. This weakness is now largely sealed due the sheer depth that computers are now capable of calculating.

Our project uses iterative deepening minimax as its primary search strategy. The heuristic is a simple one that calculates the value of the board state by adding together the board’s piece values. Using minimax on its own is an incredibly sluggish process. Looking only 4 plies deep can take over 2 minutes to calculate. A significant amount of time was spent working on various optimizations to improve the speed of computation. Algorithmic improvements implemented include alpha-beta pruning and node ordering.

Alpha-beta pruning is used to cut off significant portions of the search tree in order to speed up the searching process. In addition to this, node ordering is used to estimate which nodes are best during the search. By doing this, it is more likely that larger portions of the search tree get pruned earlier in the searching process.

Additionally, there have been software optimizations that have been made such as distributing calculations over multiple threads. At first, a 4 depth search would often take over 2 minutes in the during mid game. After implementing the optimizations, the same depth search in the middle of a game often clocks in under 5 seconds. This allows the engine to perform more than 20 times faster with optimizations.

I would like to continue playtesting the engine and making improvements to its play after this project is completed. Even with the engine in its current state, it has been a weird experience being defeated by it. I can beat it, but it takes more effort and thought than I would like to admit. It would be an interesting continuation of this project to run it against other engines of known strength to get a feel for where its elo rating might be.

References

[1] C. E. Shannon, “Programming a Computer for Playing Chess,” *Philosophical Magazine,* vol 41. no. 314. Mar, 1950.

[2] F. Hapgood, “Computer Chess Bad-Human Chess Worse,” *New Scientist,* vol 23, no. 30, pp. 827-830. Dec. 1982.

[3] M. Sakuta, et al., “Application of the Killer-tree Heuristic and the Lambda-Search Method to Lines of Action,” *Information Sciences,* vol. 154, no. 3-4, pp. 141-155.

[4] IBM100, Deep Blue: Overview [Online] Available FTP: www-03.ibm.com Directory: ibm/history/ibm100/us/en/icons/deepblue

[5] S. Russell, P. Norvig, “Adversarial Search,” in Artificial Intelligence: a Modern Approach, 3rd ed. New Jersey: Pearson Education, Inc., 2010, ch. 5, sec. 3, 167-171.

[6] Endgame Tablebases Online [Online] Available FTP: kirill-kryukov.com Directory: chess/tablebases-online