COMS3005A Report: Congo

[Sibabalo Luqhide - 1963321] and [Phindulo Makhado - 1832463]

10 November 2021

1 Methods

1.1 Alpha-Beta Pruning

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision. The key points in Alpha-Beta Pruning are: -The Max player will only update the value of alpha,- The Min player will only update the value of beta, - While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta, - We will only pass the alpha, beta values to the child nodes. And the Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast

1.2 Advanced evaluation function

The advanced evaluation function calculates maps of what the piece can do several moves into the future. We establish ”objectives” and reward our piece if it has the potential to accomplish these objectives. We reduce our bonus for each move that it takes the piece to accomplish the objective.

1.3 Modification 3 (Negascout modification)

NegaScout is a directional search algorithm for computing the minimax value of a node in a tree. It dominates alpha-beta pruning in the sense that it will never examine a node that can be pruned by alpha-beta; however, it relies on accurate node ordering to capitalize on this advantage. NegaScout's fail-soft refinements always returns correct minimax scores at the two lowest levels, since it assumes that all horizon nodes would have the same score for the (in that case redundant) re-search. NegaScout just searches the first move with an open window, and then every move after that with a zero window, whether alpha was already improved or not. While re-searching, NegaScout uses the narrower window of score, beta, while other implementations dealing with search instability, re-search with alpha, beta. When implementing this modification, the NegaScout works best when there is a good move ordering but when the move ordering is good, With a random move ordering, NegaScout will take more time than regular alpha-beta; although it will not explore any nodes alpha-beta did not, it will have to re-search many nodes.The advantage that this modification brings to our AI, is that it dominates alpha-beta pruning in the sense that it will never examine a node that can be pruned by alpha-beta; however it relies on accurate move ordering to capitalize on this advantage.

1.4 Modification 4 (Move ordering Modification)

To improve the number of cut-offs made by the alpha-beta algorithm, the program attempts to order moves so that better moves are considered first. Moves are awarded bonuses. The moves are sorted so that those with higher scores are considered first.Move ordering uses a heuristic to decide the order in which moves are considered within (or at the fringe of) the search tree. Move ordering has its effect closer to the root of the tree, which is more relevant to the decision at hand. Furthermore, move ordering only requires computing the heuristics for a very small number of board positions. Heavy playouts require that the heuristics be evaluated for every move of the game. On the other hand, having more accurate playouts may be worth the price of completing fewer playouts in the allotted time. Heavy playouts certainly are better asymptotically, in that if the heuristic were perfect (suggesting only the single best move), only one sample would be needed from each tree node.

2 Results

Present your results here. If you have two agents (because you're working individually) compare the two agents directly by playing two games from the starting position (with each agent taking a turn to be black and white). Otherwise, you may have three agents (or more):

the minimax agent, the alpha-beta pruning agent, and the agent with all modifications implemented. Compare these agents in a round-robin tournament, where each agent plays the others twice (once as white and once as black).

The kind of game you should play depends on how you have implemented everything. If your AI can search to a fixed depth only, then you should select a depth (greater than 1) for each agent such that they take roughly the same amount of time to return a move (e.g minimax to depth 2 and alpha-beta to depth 4). However, if your agent's are always able to return a move after a fixed amount of time (e.g. you're using iterative deepening), then play your games with fixed time controls (e.g. each agent is allowed X seconds per move). Report the results similarly to the tables below, where 1 – 0 indicates a win for white, 0 – 1 a win for black, and 1/2 – 1/2 is a draw.

		Black		
		Minimax	**Alpha-Beta**	**My Agent**
White	**Minimax**	1/2–1/2	1/2 – 1/2	0 – 1
	Alpha-Beta	1 – 0	1–0	0 – 1
	My Agent	1 –0	1 – 0	1–0

Table 1: Results for tournament with e.g. 30 second time controls

3 Optional Additional Information

While we were done with implementing our modifications, our code could not run and stop by itself, this in turn means that the winning condition of the game while the AI's were playing was not being considered we could not properly examine how the condition could be well considered, but we believe that our modications implementations are working very well and are quite fast