Playing Gomoku with Minimax Alpha-Beta pruning algorithm

Gonçalo Figueira(ID:180583128)

g.g.c.figueira@se18.qmul.ac.uk

Heuristic function

The estimation of the current board state focused on the sets (ie. consecutive pieces) that each player had on the board in the different 4 possible directions (vertical, horizontal, left-right and right-left diagonal) (see *getScore* method). Depending on the set length and in the number of open-ends of each set, a score was assigned empirically (see table below). In addition, the score was a function of the player's turn, since a set can be much more valuable if it's on a player's turn or not (eg. a set of 4 with 1-open end in the opponent's turn is not that valuable as it can be easily blocked). Note that a zero score was assigned to sets with no open ends. First, the score was computed for the max player, f_{max}, and for the min player, f_{min}, by summing all the points of the sets of the correspondent player. Finally, the global score was computed as the difference between the two: $f_{global} = f_{max} - f_{min}$ (*getGlobalScore*). If a terminal condition was found (set of 5), the maximum value was returned (20x10^9 for max player or -20x10^9 for min player). Therefore, the final state estimation ranged from -20x10^9 and 20x10^9, where a positive or negative value reflected an advantage for either max or min player, respectively. The max and min values of the evaluation function were used to signal a terminal condition (*isGameOver*) inside the max and min function (win, lose or draw). When detecting the length of the set in a given direction, the program had to take into account that multiple sets might be present in that direction and make sure that they were detected separately (*EvaluateHorizontal* example).

SET		TURN	
Consecutive	**Open ends**	**Me**	**Other**
5	**-**	2000000000	
4	**2**	100000000	50000000
	1	100000000	50
	0	0	0
3	**2**	10000	50
	1	7	5
	0	0	0
2	**2**	7	7
	1	3	3
	0	0	0
1	**2**	2	2
	1	1	1
	0	0	0

Move search

The search space can be very large, especially in the beginning of the game (64 possible moves). In order to improve the efficiency of the minimax search (which depends heavily on the branching factor), we tried to eliminate moves of no real value. We reduced the number of successors by considering only the empty cells near the proximity (within the 4 possible directions) of the pieces already positioned on the board (*getFiltMoves*). This modification lead to a huge reduction of the search space and far less branches at each level of search process. As a consequence, the new agent was much stronger and beat the previous alpha-beta version and the vanilla minimax every time.

To further improve alpha-beta pruning, I tried sorting the possible moves based on the distance to the player's pieces (ie. locations 1 case away from player's pieces were put first than positions near to the opponent's pieces), with the idea of increasing the likelihood of finding better moves first and therefore pruning more sub-branches in the search process. However, this extra computation did not improve the performance and were not considered for the final implementation (only locations one case away from any piece on the board were considered).The center of the board (3,3) was picked when the agent was the starting player in the game. This move increased the chance of dominating the center of the board and of generating more threatening configurations (set of 3s with 2 open-ends and sets of 4).

Iterative deepening search

Despite using pruning methods and limiting the number of successors, the search space will still be very large and will grow exponentially until an end state has been met. The standard alpha-beta pruning algorithm used depth-first search to find the best move at a given depth cut-off, however, this might not be optimal, since better moves that have not been evaluated might be located at shallower levels of the search tree. This was observed for example in near terminal conditions (for example closing a set of 4), were the agent was searching deeper and making erroneous moves, where the optimal solution was at a shallower level. In order to improve this, iterative deepening was implemented by calling the minimax with increasing max depth cut-off inside a for-loop (*line 57*).

Fixed/dynamic search cut-off

By experimenting with agents with increasing cut-offs playing against each other, it was found that players with higher cut-offs were stronger. Furthermore, I experimented using a time-based cut-off, which would terminate the search if a time limit was found (9.9s). This modification allowed the agent to search deeper in certain moves (especially in the end of the game, since there are fewer options), which made the performance increase. In the iterative deepening procedure, if the time constrain was reached in a middle of a search at a particular depth, the recursive call was stopped and the move found at the previous depth was kept.

Forced moves

I decided to hard-code a finishing move to guarantee that the agent finished the game when it had the opportunity. Likewise, after evaluating a possible winning move, it also checked if an opponent's winning move could be avoided (*line 116*).

Future work

- To exploit further IDS, a look-up table could have been generated to save the state and minimax value, so when we have to evaluate the same state again, instead of expanding the whole subtree, we could much quicker just take the value from the tables saved in memory. This would allow to go much deeper in the tree, as the values in depth *d-1* would be saved for the next search *d*.
- Use the evaluation score at shallower levels to indicate if a searching should be performed from left-to-right or right-to-left (therefore leading to more pruning).
- Speed-up computation using bit-board state representation.