Artificial Intelligence COSC1127

Assignment 2 Report

Joshua Thorpe s3543345
Spencer Porteous s3539519

https://github.com/Splerge/SpenJosh-ai1901-connectfour

Search Algorithm

Our agent utilises a minimax algorithm with alpha-beta pruning to reduce the search space. Branches are terminated either at the max depth to prevent the search from taking too long, or when a game is won or drawn. If a branch is terminated at maximum depth, the algorithm will evaluate the board using the heuristics that we have implemented.

Scoring

If a board state is won or drawn, the score for that board state is calculated as ((*board.width * board.height) - move_number / 2 + 100).* This allows the algorithm to detect the "distance" from the current state to a win or loss board state, and make moves accordingly that will result in the highest scoring state. The constant of *100* is added to separate these winning scores from the board evaluation scores for pruning purposes. If the max depth is reached, the board will be evaluated using our heuristics which are described in detail below.

Algorithm Optimizations and Techniques

As our search algorithm has a default maximum complexity of O(a^b) where a is the branching factor, and b is the maximum search depth, the amount of nodes to evaluate can be very high, especially early in the game. Due to these, we implemented several optimisations.

Variable Depth Limit

As the game continues, many columns are filled up, many positions become instant losses for one of the players, and there are much less moves before the end of the game. This reduces the branching factor significantly, so the maximum depth can be increased safely while keeping the calculation time small. Around turn 27, the max depth can even be removed, as the amount of remaining positions is small enough to calculate an optimal move within a second.

Move Ordering

According to multiple strategy guides central moves are much better for increasing your chances at winning. Our algorithm reorganises the moves before the search begins to search

the middle column first and the outer column last (Fig 1.) This increases the chances of the alpha-beta window closing around optimal values early, which will decrease the amount of nodes that need to be searched because many more will be pruned away. This also means in the event of a “tie” (Two moves are scored the same), a central move will be favoured by the agent.

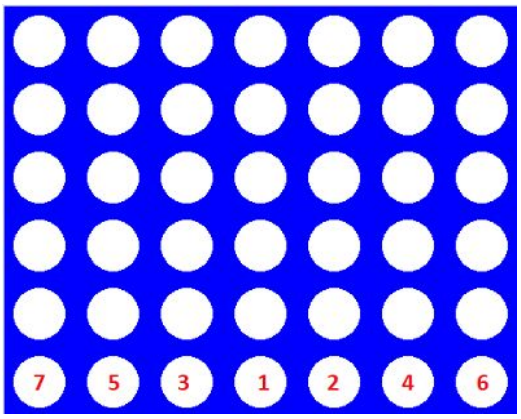

Fig 1. How the columns are ordered before the search begins.

Removing Branches That Are Considered A Loss

Before the search is applied, the agent removes any moves that will result in a loss the turn after. This causes “forced moves” (moves where the agent must go in a specific column in order to not lose) to be instant with no time taken for a search, and can potentially reduce the branching factor as well.

Memorization

Values returned from each node are hashed and stored in a lookup dictionary. This is checked before evaluating each node, greatly increasing the speed of searches that encounter duplicate nodes.

Search Algorithm Summary

This algorithm makes the agent strong and efficient and made the agent difficult to beat from a human’s point of view.
The performance is good during the midgame (moves 10-20) and endgame (moves 20-42) of the match, however it could be improved during the beginning of the match (moves < 10) as it is very slow. The branching factor can be reduced early game by removing the side columns out of the search space provided there is no tokens in them, because good agents will usually avoid the side columns in the first 8-10 moves.
If we could split the agent up across separate files, an opening book could be added - a precalculated database of every score of every board state within the first 8-10 moves. Such a database could trivialize calculation time in the early game at the cost of space.
This algorithm, when combined with the following strong heuristics, has made our agent a formidable opponent.

Heuristics

Search Heuristics

Not all the heuristics were built into the board evaluation function as this was only called if the max depth was reached. A simple check for a winning/losing state was checked upon expanding a node to ensure no further expansion was completed. The large score was returned to ensure it was more significant than any score returned by the board evaluation function and it also took into account how many moves had been played. This was due to the situation when multiple winning boards were returning the same high score, it would just pick the first winning score in the list of values but this was often a delayed win when the game could have been one much sooner.

Board Evaluation

If the maximum search depth was reached without activating any of the other heuristics then the board was evaluated and scored to determine which player was in a better position. This board evaluation was a threat based evaluation and took into account a variety of connect four strategies to return a more informed score.

Threats

A threat is generated when a player places a token in such a way that an open position can be played to connect 4 tokens and win the game. The open position could be at the end of three tokens or in the middle but this open position presents a direct threat to the opponent. If the opponent fills the threat position with their token then the threat is blocked and the fact that three tokens were connected becomes irrelevant. This is why a threat based analysis is better than simply evaluating the number of connected tokens.

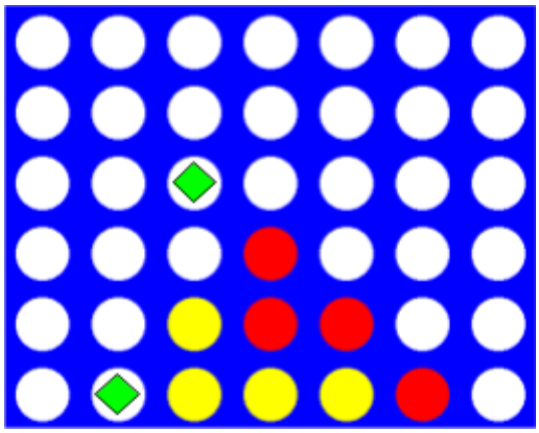

Fig 2. Shows threat positions for this board in green.

Identifying threats

A winning position can be achieved in the following three orientations: horizontal, diagonal and vertical. So threats need to be identified in these three orientations

Analysing threats

Once a list of threats of each orientation are generated further game logic was applied to increase the heuristics strength. Counting the number of threats was a simple scoring method trialled and resulted in an intermediate level agent. When analysing the board scores, an identical threat position that was identified by both the horizontal and diagonal threat analysis would increase the score for a board.

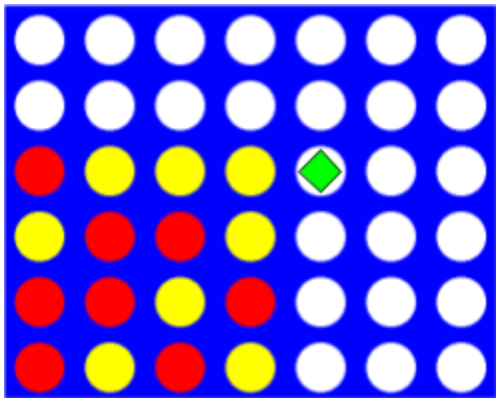

Fig 3. Shows redundant double threat on same position

The above board has a threat at the green position, but if a yellow token is placed in this position it doesn't matter that yellow wins horizontally and diagonally as only one of these threats on this position was necessary. So to increase the strength of the agent, the score should not increase if two different orientation threats are generated on the same position. To ensure this the three lists from each orientation of threats could be analysed for duplicates or the use of the python set data type could be used. The python set data type natively stores a set of unique unordered values. The identified threats don't need to be ordered or treated over much but do need to be unique and checking if a value in the set exists needs to be fast. For these reasons the python set was chosen over the list as it performs faster and requires less code for this particular application.

Once duplicate threats are removed more bad or invalid threats can be removed.

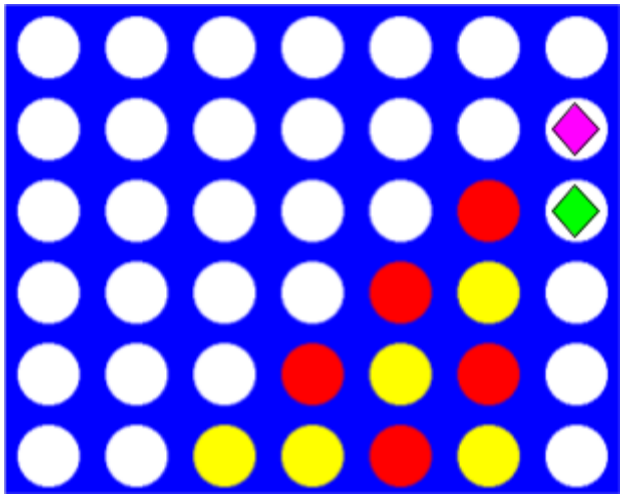

Fig 4. Bad threat by red as it is directly above yellows threat

If a threat is generated one position above an opposing threat this is also a redundant threat. In the figure above, if yellow plays in the green position then they win and the game is over before red ever gets to play pink, and if red blocks yellow's green threat then the very next move, yellow will have the opportunity to block red's pink threat and hence it is pointless. For this reason the heuristic removed these bad threats from affecting the score.

Once all bad threats are removed, the good threats can still be further analysed and more strategies applied to create a more informed heuristic.

A threat generated at the top of a board is not as significant as a threat generated at the bottom of a board as it is much less likely to ever be played. As the board fills from the bottom up, a threat at the bottom of the board could win a game much earlier and should be scored better.

Furthermore, the threat analysis of the odd, even strategy made the agent significantly stronger. If threats are generated on the lower rows of the board, the opponent is blocked from playing tokens in the columns that the threat exists in. In advanced gameplay often the board will become segmented, fill up and will come down to each player being forced to play a token in the column that the threat exists. In these situations the importance of the row that the threat is generated in is significant. For player one, they should aim to generate threats on odd rows starting at row one from the bottom, and player two should generate threats on even rows as these are the most likely rows that each player will end up with tokens in when the board is full and they are forced to play the column with the threat.

This strategy is critical when top players compete against each other, as it essentially comes down to if player one produces an odd threat lower down the board to player twos even threat, they will win and vise versa.

Heuristic Summary

The threat based and winning state heuristic require a reasonable search depth to work effectively. If the search depth is small then the heuristic didn't return any scores for the first few moves of the game. The simple central heuristic added very small values to boards with tokens in the middle of a board and this helped the agent early on until the threats heuristic started finding boards with threat features. More advanced strategies to identify which player holds the Zugzwang (a situation in which the current player forces opponent to make a disadvantageous move) could make the agent even stronger in early game play or if search depth is small.

Contributions	
Joshua Thorpe s3543345	● Report writing ● Code peer reviewing ● Testing ● Board evaluation code ● Threat based heuristic ● Minimax speed optimizations ● Transpose tables ● Alpha beta pruning
Spencer Porteous s3539519	● Report writing ● Code peer reviewing ● Testing ● Get_move and minimax search algorithm code ● Faster board tools and move generators ● Winning state heuristic integration into search algorithm ● Transpose tables ● Alpha beta pruning ● Variable depth function