Martin Mao 54807690

Yang Jiao

Zeyu Su

Speedy AI for 4 in a Row

Introduction

Describe the game

Describe why we are doing this

–Define problem you are solving

–Describe your approach

Our goal is to make a fast algorithm that is still reasonable.

Performance

* Win the game
* Too slow, infeasible time

Environment

* Game board 8x8 grid

Actuator

* Placing tiles on the board

Sensor

* Board stored as matrix for evaluation

Describe GUI and how we capture this. Describe parts of GUI.

Describe Problem/Constraints

Strategic game

* Adversary in Game
* Ideal moves may be blocked

Very large branching factor of moves

* Board size of 64
* Not all moves will contribute anything

Good moves are deep, need to plan ahead

* Because of nature of the game, obvious winning moves are easy to block. AI needs to plan ahead

In implementing our algorithm, we needed to specify a value for each move. Heuristics need to be able to determine which move is more favorable. By comparing the values of several states, the heuristic will be able to evaluate what the next move should be as it seeks to move to a more favorable state and seek to win the game. We created an initial utility function that assigns values to each move, a move being the AI placing a marker of its own color onto a tile, based on the value of tiles. The value of a tile would increase based on the number of nearby tiles with the same marker placed by the AI in previous moves. This would make tiles surrounded by the same marker appear more favorable. A tile’s value would decrease however based on the number of other markers nearby. These tiles would seem to be more favorable for the opponent(s). The utility function we created is (number of own markers in a distance of 3 away on the same row, column, or diagonal) minus (number of enemy markers in a distance of 3 away on the same row, column, or diagonal).

We decided this was a good utility function to use based on the nature of the winning condition of 4 in a row. The game requires a chain of a player’s own markers to win. By making moves close in proximity to the AI’s previous moves, it will be able to better construct a chain of 4 in a row. The utility function was also restricted to only look into tiles of a distance three away because it requires only a chain of four markers to win the game. Including the current move, this will be a chain of four markers. The algorithm will have to evaluate the value of each of its children, children being the resulting board configuration after making a move. The most favorable move for the AI will have the highest positive value and the AI should make the move that will result in that state. A very favorable move for the enemy will be marked by having a very negative value.

We had to make additional adjustments to the utility function; the AI was making moves that seemed good even when there was a winning move in sight. A linear utility function did not prove ideal in all cases. We adjusted the heuristic to assign a value of 1000 for winning moves, this would guarantee that the heuristic will choose winning moves over moves that simply had a high value. The AI would also make some very bad moves that were not nearby any of its own markers. Because a tile’s value was reduced by each nearby marker of the adversary, those tiles did not appear appealing to the AI. The AI would see a very far tile as better because it had a higher value than a nearby tile that was both near its own markers and the opponent’s markers. Each tile was revised to have two values: its worth the AI alone, and its worth the adversary. The value was each state was then recalculated to be the sum of every tiles’ worth to AI minus the sum of every tiles’ worth to the adversary. Moves that would block winning chains from the adversary as well as further a chain for the AI now seemed more beneficial. These moves would increase the tile’s worth the AI and decrease its worth for the adversary, because there is one less neighboring tile for the AI to take advantage of. The caused the AI to prioritize blocking a winning move (a move that has a negative value) which previously did not look like an obvious good move for the AI. A move that helps contribute to a 4 in a row for the AI while blocking the opponent now seems better than a move that only makes a 4 in a row.

The algorithm we chose to implement for AI would use Iterative Deepening Searches with Alpha Beta Pruning. This is a depth first search that has a bounded depth. The depth value was allowed to increase or decrease based on timing requirements. We wanted to record the time for the moves of the AI to ensure that moves were completed within a reasonable amount of time. If there was extra time available in the current search, the depth value would increase and the AI would use a larger depth value for the next iteration of running depth first search. The depth value would be maintained for the duration of the game until it needed to increase or decrease again. If the AI took too long for the current move, then the depth value would decrease and the AI’s next move will search to a smaller depth. We decided a reasonable amount of time for each of the AI’s move should be within five minutes. The algorithm was allowed to adjust the depth value accordingly based on this time limit. Alpha beta pruning was implemented to help speed up searches. Each level of depth first search would alternate placing a move for the current player and a move for the opponent. The move for the player would try to maximize the values of moves possible from the current move while the opponent would attempt to minimize the values. Given that the AI will know the opponent will attempt to make the best move possible and disallow the AI from making good moves, there will be little contribution to searching a branch that the opponent wasn’t going to allow. Alpha beta pruning will allow the AI to prune away the rest of the branch and ignore the remaining children.

Good moves are possible by planning ahead and searching deeper into consecutive moves. Iterative deepening will allow the depth to fluctuate and potentially allow the AI to find better moves deeper into the tree. However the branching factor of the game is huge and searching each consecutive moves is very time consuming. Alpha beta pruning will allow the AI to prune away bad children and save time from having to traverse them. This in turn will allow the AI more time to search deeper for good moves and plan further ahead. Planning moves ahead is important because obvious winning moves can easily be blocked by a capable opponent. The opponent is making moves against the AI and is expected to disallow the AI from attempting to win.

After the initial implementation of our algorithm, the algorithm was still very considerably slow. Alpha beta pruning did create a large time reduction for each search. Compared to only having depth first search alone, alpha beta pruning was able to considerably decrease algorithm time by removing expansion of child nodes that were not optimal. However we were still receiving moves taking over five minutes at depth three. Iterative Deepening could not increase depth past three because it was hitting the timing restriction we set for moves taking over five minutes. We wanted to be able to allow depth to increase further through iterative deepening and decided to enhance the algorithm and possibly reduce the branching factor involved in the game. In order to reduce expansion from 64 possible tiles, we first looked into the obvious candidates of illegal moves. A move couldn’t be placed onto a previous move so we were able to reduce the list of possible candidates for expansion. We also wanted to apply the restriction that only viable tiles with good values gain should be candidates for the next move. Because it requires a chain of four moves in a row to win the game, there would be little gain from placing a move that did not build up a chain. Out of the possible 64 tiles that could be expanded, we then limited children to only contain the tiles that were a distance of three away from existing tiles. Including the current existing marker on the board, it would only require three additional markers to create a chain of four and win the game. This enhancement was able to speed up the algorithm significantly in the early stages of the game. Each successive move was able to increase depth further until there became a significant amount of existing markers on the board. The beginning stages of the game will have the possibility of tiles reduced significantly from 64, since there are so few available tiles on the board. However towards the later stages of the game, the branching factor would approach 64 again because the remaining tiles will be viable for being distance three away from an existing marker. The branching factor of searches will eventually become huge again and make searches take significantly long.

Reducing the size of potential children for each branch would give massive speed ups for our algorithm searches. Instead of having to search the entire board of 64 tiles, we reduced the branching factor to a fraction of this. The potential tiles to expand on each level of search are limited to tiles that are distance three away from existing markers. A branching factor of 64 will lead to significant timing hindrances because depth first search is exponential on 64 based on the depth. Reducing the branching factor from 64 will allow depth to potentially be bigger and still meeting timing requirement of searches. With the algorithm speed enhancement, the early portion of a game would allow the AI to search beyond depth two or three and search into depth four and five. However as the game progressed further and each player has placed a significant amount of markers onto the board, more tiles became eligible for being a distance three away from an existing marker. Long games would thus negatively hurt our algorithm because the branching factor of each move would approach 64 again. The depth of each search would slowly drop to three or even two in cases where the AI could not achieve an early victory or loss. In the rare case where the game went extremely long and there number of available spaces of the board has extremely diminished, the AI would be able to make deep searches again because there are few legal moves left on the board. Our speed up enhancements to the algorithm could thus only help the algorithm make wiser moves with deeper planning in the earlier stages of the game. Later stages of the game had to suffer from a huge branching factor and a reduction to the depth of searches.

**Describe issues of winning**

Compared to Bad AI

* Algorithm will always win against very bad AI, such as Random AI
* It will also usually win against AI using only depth of 0
  + Depth 0 AI will not expand moves, but just find max of tile values and commit to max
  + Algorithm sometimes lost if made obvious winning chains. The depth 0 AI was smart enough to block the AI and become in the lead

Compared to Decent AI

* Against our original AI with the original utility function, the algorithm was very good
  + This was because the original AI would make occasional very bad moves

Compared to Players

* Compared to players, the algorithm would win sometimes, not often
* It was very good at making plans that led to its own victory
  + Moves that allowed multiple possible winning moves
* However it was bad at stopping Player’s strategy of making good plans
  + It can be easy to beat by players who plan ahead
  + AI did not make good blocking moves that could stop players early on

3) compare with other algorithm using resources: compare with brute force and with most optimal algorithm

-how does it compare to dumb AI and how compare to AI that always makes best move (forsight)

1) evaluate algorithm: how good is it? is complete? is complex? space take, time take

-compare to other AI to determine how good is (percentage of win, relative to speed)

-make some claims and try to back it up about our AI

Describe timing issues

Timing

* Very fast if depth does not increase too much
  + Without alpha beta pruning, going beyond depth 2 would take over a minute
  + Without speed ups or pruning, depth 2 took less than a minute, but depth 3 and more would take too long
* Speeds up of reducing branch size allowed more depth within reasonable time
* Algorithm was able to increase to depth 4 or 5
* Removing speed ups would have made a high depth infeasible
* 64 possible tiles was a huge branching factor

Space

* Storing the data was compact
* Did not search into infinity and create memory issues

–Describe/analyze properties

–Describe/analyze experimental results

–Make observations

Problems/Improvements with the Algorithm?

Although the algorithm was very good at trying to win

* Very bad at stopping an opponent from winning if the opponent is going for a not obvious winning strategy plan
  + This could have been fixed by making our utility function more complicated
  + Trying to win appears better than trying to block, didn’t block often
* Human players can search deeper for very good plans and strategies
  + AI was limited to search all strategies to the same depth
  + Searching a bad strategy takes time and the algorithm will not increase depth value when time is wasted

Early in the game, large depth has little deciding factor

* Values do not fluctuate a lot when few pieces on board

Large depth matters more later in the game

* Tile values will differ more as the game progresses
* The best moves are not as obvious
* Allowing depth to fluctuate and potentially increase will help the AI make smarter moves when needed

Alpha beta pruning and branch reduction are necessary

* Branching factor of game is huge
* AI’s moves will take too long when depth increases

Having deep searches is not enough

* Utility function that is too simple makes okay and very good moves seem indifferent
* Making obvious winning moves wasn’t very good. The other player could easily block those moves. Whoever when we increase the depth of the searches, the AI would choose to rather make more tricky moves that allowed multiple winning moves. Being able to plan moves ahead has a large advantage.
* Bad utility function will cause deep searches to be meaningless. If the utility function is not very good, then the good and better moves will have the same values. The AI will think they will have both just as good as the other and will not make the better move. The heuristic needs to be improved to better differentiate between what’s a good move and what is better or else having deep searches will not be as effective.