Martin Mao 54807690

Yang Jiao

Zeyu Su

Smart AI for 4 in a Row

Introduction

Describe the game

Describe why we are doing this

–Define problem you are solving

–Describe your approach

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.

Describe Algorithm

Implement an Iterative Deepening Search Tree with Alpha Beta Pruning

* Maintain a variable called depth
* and increase/decrease depth based on timing requirements
  + If there is extra time in the current search, then increase depth for next iteration, and run depth first search again.
  + If this search is taking too long, then decrease depth. The next move will use a search with a lower depth
* Alpha beta pruning to speed up searches
  + Ignore searching the rest of the children in a branch if this branch is bad
  + Maintain a beta and alpha value in searching. If beta <= alpha, we know the adversary will attempt a very good move. Do not bother to expand the rest
  + Gives algorithm more time, allowing for a larger depth

This allows the algorithm to search for good moves and plan ahead

Not make very obvious moves and also consider that the opponnent is making moves against them

Start with this DFS search with alpha beta pruning

Then if there’s extra time, redo search with depth = depth + 1

Else decrease depth for next turn

Tile location of the first move doesn’t matter too much

* + We can save time by starting the algorithm with a low depth value
  + Depth will increase on successive moves through

Describe speed enhancements to algorithm

Even with Alpha Beta Pruning, the algorithm is too slow when we want Iterative Deepening to increase the depth more

Not all tiles are good to search

* Avoid branching on all of 64 tiles
* Ignore illegal moves
* Restrict branching to tiles near existing tiles. A distance of 4 away (since 4 moves needed to win)

Even if some tiles are valid for expansion. Do not bother to expand if the tile’s value is too low.

* Algorithm will look all values of available tiles and determine a threshold value.
* Tiles with values below the threshold will not be expanded.

Reduce branching factor and speed up algorithm. This gives it more time to evaluate better branches to a larger depth

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
  + Algorithm sometimes lost if the first move was bad. We gave algorithm a low starting depth value and it would occasionally pick a bad first move

Compared to Decent AI

* Against our original AI that compared tile values of both itself and the adversary, the algorithm was very good
  + This was because the original AI had a higher chance of wasting bad moves

Compared to Players

* Compared to players, the algorithm would win sometimes
* It was very good at making plans that led to it’s 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 if the player went first
  + AI did not make good blocking moves that could stop players early on

Describe timing issues

Timing

* Very fast if depth does not increase too much
* Speeds up of reducing branch size allowed more depth within reasonable time
* 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?

Conclusion

Going first has a huge advantage

Early in the game, large depth has little deciding factor

* Tiles mostly have relatively same values

Large depth matters more later in the game

* Tile values will differ more as the game progresses
* The best moves are not as obvious

Alpha beta pruning and branch reduction are necessary

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