MP2: Project Report

Members:
Jhwang51 (448, Q3), awdisho2 (440Q3) , sskim16 (448, Q3)
CS 440 Artificial Intelligence
Professor Hasegawa-Johnson
27 February 2019

Section 1:

CSP: We would like a description of your representation of the problem as well as an explanation of the algorithm (as well as heuristics) that you used to find a solution.

Algorithm:

We decided to implement algorithm X in order to solve this problem. Algorithm X is a fast algorithm that was made to solve the exact_cover problem where rows are chosen from a matrix so all of the chosen rows together contain a 1 in every column. This problem is very applicable to other problems such as the pentomino problem as long as you can properly set up a matrix to pass into the algorithm. Once passed in, a column is selected with the least 1's . For every one in the column, rows and columns are deleted and the function is then called recursively. If it needs to backtrack, it will try a different one from the column to base the deletions on. Once the columns are empty, it returns the solution and stops further recursion in our implementation.

In order to create the matrix, we found all board locations that all orientations of all pentominoes could be placed. With this information, we could create the rows of the matrix. The first x parts of the row being the 2D representation of the board after adding the pent and the next y being which pent it was. This works well for the problem as our pents all can only be used once and cannot overlap. So having the pents and board spaces represented by ones causes them to be disabled from being chosen in future steps of the algorithm.

Receiving the data from the finished goal was difficult as algorithm x only returns which rows were chosen. To circumvent this, we added an extra column at the end for the row numbers of all of the rows on the original matrix. That way even after being heavily altered, the board placement and pent chosen can be retained.

Heuristics:

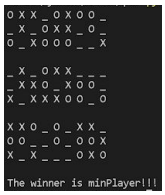
The heuristics we had were that at any given iteration of the algorithm, we always choose the column from the matrix with the least 1's as this will have the least branches(the ones are our branch factor in this case). This behaves as an LRV. We also instantly stop the algorithm if any of the columns have no ones left as this means that a spot on the board cannot be filled anymore. This forward checks and prevents wasting time looking at solutions that cannot be had.

Section 2:
Ultimate Tic-Tac-Toe: for each of the four games of predefined agents, report the final game board positions, number of expanded nodes, and the final winner.

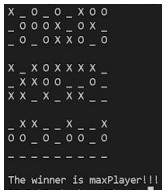
For our implementation, we were unable to successfully implement ultimate tic tac toe. Although we were able to receive outputs, we are certain that there are bugs in the code and that the output is incorrect. We would greatly appreciate if we could receive feedback on where exactly we went wrong in the code.

What we have to show is the provided code and the following output for one game of uttt using 'X' player first and minimax for both the offensive and defensive agents:

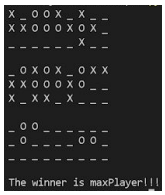
Start Local Board 0:



Start Local Board 1:

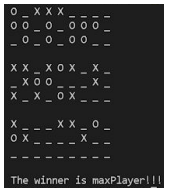


Start Local Board 2:

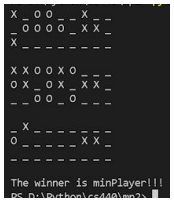


Start Local Board: 3:

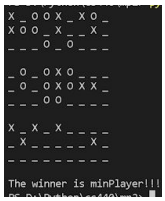
Start Local Board 4:



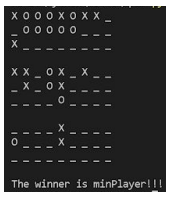
Start Local Board 5:



Start Local Board 6:



Start Local Board 7:



Start Local Board 8:

Section 3:
Ultimate Tic-Tac-Toe: for at least 20 games of offensive agent vs your agent, explain your formulation and advantages of evalution function. Report the percentage of winning time and number of expanded nodes for each game. Report 3-5 representative final game boards that show the advantage of your evaluation function vs predefined offensive evaluation function. If your own defined agent fails to beat the predefined agent, explain why that happened.

For this section, we were unable to complete our own offensive agent. Therefore, instead of leaving it blank, we have provided an explanation of our current implementation of the predefined minimax algorithm.

playGamePredefinedAgent

1. Check first player
2. Save initial max and min players
3. For loop for 81 moves (max number of positions)
4. First get the local board and find all possible board placements
5. Next, check whether you are running the minimax or alphabeta algo
6. For all the valid moves, run the minimax algo and play the game 3 moves into the future, with currentPlayer, opponentPlayer, and currentPlayer again
7. Get the best value and the best move, then place it onto the board
8. Check the local board for a winner, if so break
9. Update the new localBoardIndex
10. Switch the min max players to simulate the two agents playing against each other

Minimax

1. Check the board for a win, if so return 10000 if is maxPlayer
2. Get all next moves, if no moves evaluatePredefined
3. Run recursively on the function until max depth is reached

Evaluate Predefined
Get the value of the current position based on the predefined rules.

Section 4:
Ultimate Tic-Tac-Toe: for at least 10 games of human vs your agent, discuss your observations, including the percentage of winning time, the advantages or disadvantages of your defined evaluation functions. Report 3-5 representative final game boards that show the advantages or disadvantages of your evaluation function.

We did not complete this section.

Section 5:
We think that our code for the CSP is worthy of extra credit for how fast it runs. After several days trying to make it work, we were able to come out with a very fast implementation and some of the coding quirks that had to be done in order to make Algorithm X work cleanly for pentomino were quite difficult.

Statement of Contribution:

Jason - Wrote initial version of Part 1: CSP with two version, using for loop and recursive.
I also wrote Part 2: UTTT for the agents although I was unable to complete it.
For the report, I wrote the sections for UTTT.
I have decided to submit the project and report although it is not complete in order to focus on the exam tomorrow.

We would greatly appreciate if we could discuss with you about this MP. Thanks

Sam - I contributed to the original CSP that we did, using the following heuristics:
LRV - finding the pentomino with the least amount of position that would fit on the board, and using that pentomino as the piece to be placed on the board
LCA - finding the position with the least amount of pentominos that would fit on that position and choosing that position to place the LRV pentomino
Forward Checking - finding islands of 0's after the pentomino was placed and if there was an island then immediately enter backtracing

We loop through the available positions per pent updating LRV on every iteration and updating our tried list with the pent just used, then we move on to the next pent.

Although it seemed like a good idea at the time, this was not the implementation that we went with. When implemented it would run ~30s for the 6x10 board but for the remaining boards it was very slow. It is a shame because these heuristics were recommended during office hours and we spent days trying to optimize these heuristics for our problem to no avail. Luckily Tyler was able to implement the Algorithm X for the CSP and it solved the problem very quickly.

Tyler- created algorithm X CSP, helped with report