Assignment 2: Planning, Games

University of Illinois at Urbana-Champaign
Spring 2019 CS 440/ECE 448

Heting Fu	hfu6	Section Q
Yuhao Liu	yliu167	Section Q
Yuhao Min	ymin6	Section Q

Submitted on: February 25th, 2019

Section I: CSP

The pentomino problems can be considered as an Exact Cover Problem where all the different placement of each pentomino tile at different orientation is a subset of the entire board. In other words, we can choose some of the pentomino tiles from the subset and cover the entire board.

A mathematic representation can be a matrix where each row represents a subset. Each column of the matrix represents a particular cell in the matrix. Values can be assigned to the matrix to identify the existence of tiles in the cell of the board. In this problem, "0" was chosen to represent an empty cell, while positive integer was used to represent the occupied cell.

In addition, more columns need to be added to specify the tile identity since they can only be used once. A "1" was added at the ith column of the matrix. An illustration of the matrix is shown in the following figure.

Represent the cell occupied by the tile					Represent the identity of the tile		
1	1		0	0	1		0
0	1	…	0	0	1	…	0
0	0		0	0	1		0
…					…		
0	0	…	12	12	0	…	1

Table 1. An illustration of the problem representation.

To populate the matrix, all orientations of a tile are explored. For each orientation, all possible position of the tile is tested. The cells that are occupied by the possible placement are marked with a positive integer, and the entire board is flattened. The identity of the tiles is appended at the end of the placement 1-d array. This procedure is performed for all possible placement. For the 6x10 board, there are in total of 72 columns and 2057 rows.

To find a collection of the subset to satisfy the problem, we need to find rows in the matrix such that in the new matrix formed by stacking the rows, each column has only one positive integer. Algorithm X, proposed by Knuth, is effective in solving the problem. This is a recursive, back-tracking algorithm. Each time the function is called:

It first selects a column with the smallest number of positive integers. This column is deleted.

For every positive integer element in the column, it will delete the row of such element and add the row index to the solution.

For each positive integer in the delete row, the column such element is in will be deleted.

Lastly, rows contain a positive integer element in the deleted column will also be deleted.

The recursive call terminates successfully when there is nothing to be deleted.

To back-track, the algorithm will simply "undo" the deletion and choose another row with a positive integer to delete (step 2).

Each time the algorithm is run, constraints are applied in order to yield the solution. The deletion operation in the algorithm is essentially applying the constraints to reduce the number of possible solutions. Once a row is chosen (step 1), all other rows intersecting (have positive integer element) with the chosen row should not be in the solution. Therefore all such rows are deleted, and the size of the matrix is now smaller.

The heuristic used in the algorithm is the number of positive integers. The column with the smallest number of positive integers is chosen first since essentially there will be fewer possible outcomes. In other words, the back-tracking tree structure is smaller, making the running time faster.

A dancing link (DLX) data structure was used to represent the sparse matrix. This is a double-linked, four-directional, circular mesh of nodes. The manipulation of the pointers at each node made the operation very easy and fast.

Section II: Ultimate Tic-Tac-Toe Predefined Agent

Game Number	Board Position	Number of Node	Final winner
1	Figure.1	8367	maxPlayer
2	Figure.2	5977	minPlayer
3	Figure.3	6346	maxPlayer
4	Figure.4	6346	maxPlayer

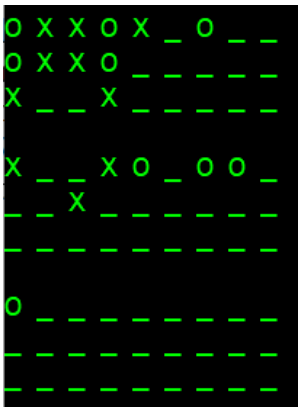

Figure 1. Offensive(minimax) vs defensive(minimax)

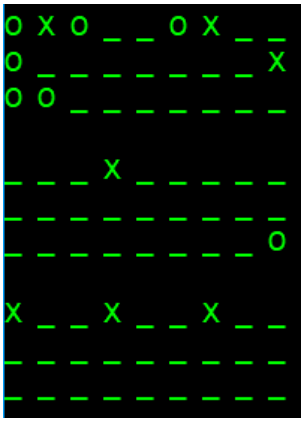

Figure 2. Offensive(minimax) vs defensive(alpha-beta)

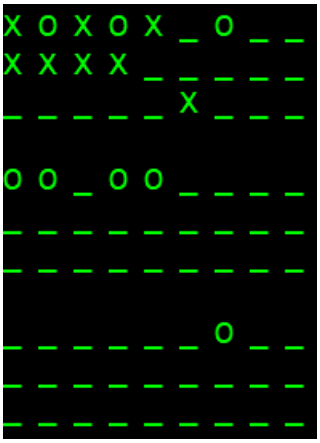

Figure 3. Offensive(alpha-beta) vs defensive(minimax)

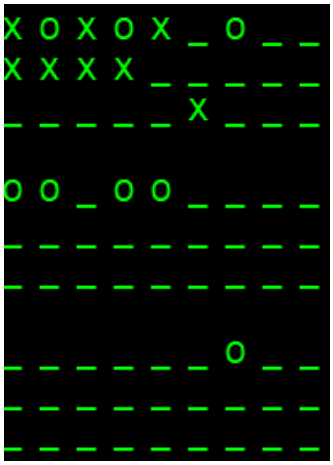

Figure 4. Offensive(alpha-beta) vs defensive(alpha-beta)

Section III: Ultimate Tic-Tac-Toe with Designed Agent

We found out the predefined defensive and offensive agents are both only evaluating their own scores based on mostly the status on the board position they occupied, meaning they consider less about how opponent's current status could affect how the game will turn out in the future. So, based on the offensive agent we add couple of new rules for determining the final score, one rule is that if our agent occupies a center, we increment the score by 100. And then if the opponent has a two-in-row we subtract 200 points out of the total score for each opponent two-in-row.

In this way we actually can take into account the opponent's move, and this gives us a way to prevent the offensive agent to do certain moves for example that can easily form a two-in-row.

Game Number	Board Position	Number of Node	Final winner
1	Figure.5	9460	minPlayer
2	Figure.6	5612	minPlayer
3	Figure.7	4621	maxPlayer
4	Figure.8	6511	maxPlayer

We played the game with the predefined offensive agent 20 rounds and get the final winning rate of **60%** which is less than the expected 80% so we picked 4 of the games as example to show what happened during the playoff. The **offensive agent is "X"** and our **designed agent is "O"**.

We picked 2 winning games as shown in Figure.5 and Figure.6. And 2 losing game as shown in Figure.7 and Figure.8.

Since we have reached 60% of the winning rate which is a little less than 80% of the winning ratio but still has an overall expected result to win over the predefined offensive agent.

We suspected there are a couple of reasons that the win ratio does not meet with the expected ratio. First, we think that the evaluation of the agent taking over an arbitrary center of a local board should not have a relatively high score over other. We tweaked that for some smaller values, but the result didn't seem to change. Second is that we didn't use the limited horizon algorithm to help detect some move that at first may have higher score, but the next opponent move may make this high score move meaning less.

So, for future improvements we can first add the limited horizon mechanism to see the forth level and if there are certain moves made on the 4th level that extremely favor the player or the opponent, then we can make certain move towards or avoid that consequence. Second is to fine tune the evaluation function to favor the designed agent.

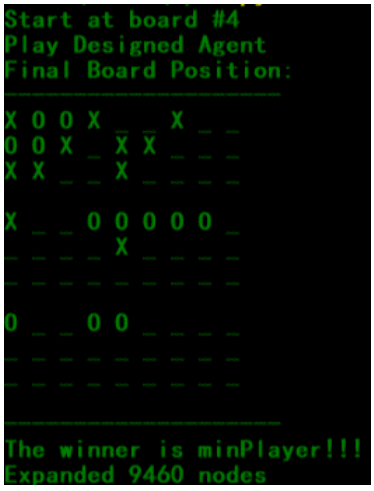

Figure 5. Designed agent vs offensive agent final board I

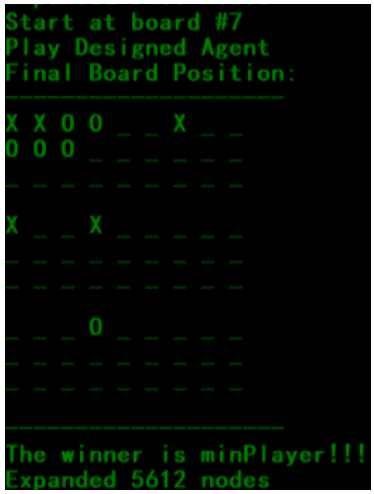

Figure 6. Designed agent vs offensive agent final board II

Figure 7. Designed agent vs offensive agent final board III

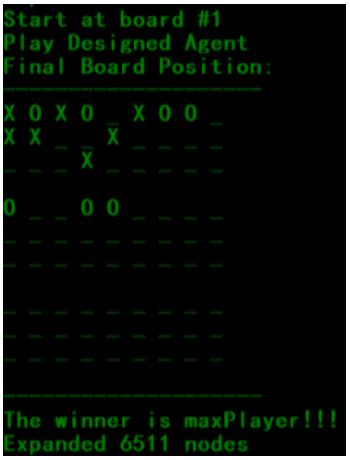

Figure 8. Designed agent vs offensive agent final board IV

Section IV: Ultimate Tic-Tac-Toe Designed Agent vs Human

For all games, the human player won. The evaluation function follows similar heuristic as that in predefined. It can be effective in stopping a computer-like opponent that follow unchanged rules. However, when played against human, since the top-left corner of an empty board is always preferred, the agent chooses it, allowing human to draw in the first local board. This is evident in Figures 9-12, as all winning are in the first local board.

An improvement of the agent can be made such that the agent place at random when facing an empty board.

Game Number	Board Position	Final winner
1	Figure.9	Human
2	Figure.10	Human
3	Figure.11	Human
4	Figure.12	Human

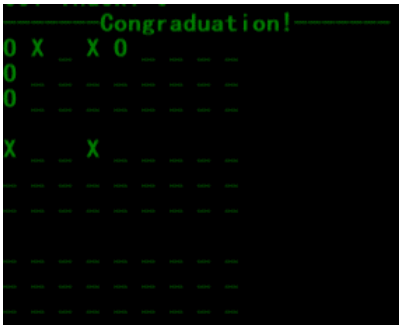

Figure 9. Human vs designed agent

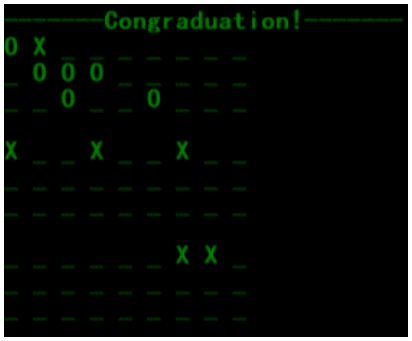

Figure 10. Human vs designed agent

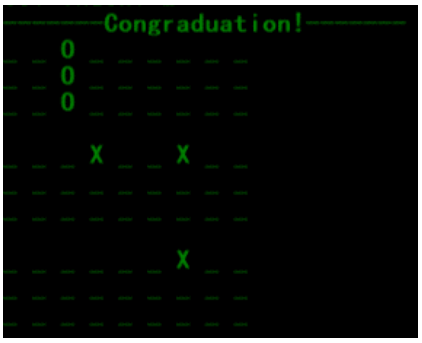

Figure 11. Human vs designed agent

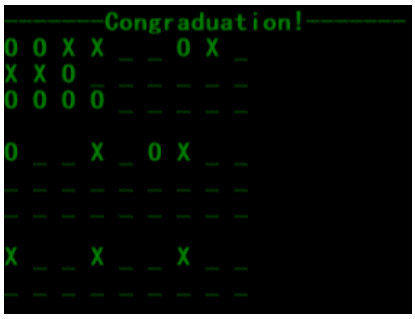

Figure 12. Human vs designed agent

Extra Credit:

From our 10 trials, the offensive and defensive players happen to win equal times. The offensive agent is the same as the designed agent we implemented against the predefined offensive agent. And the defensive agent is the offensive agent with utility score's signs flipped.

Game Number	Board Position	Final winner
1	Figure.13	minPlayer
2	Figure.14	maxPlayer
3	Figure.15	maxPlayer
4	Figure.16	minPlayer

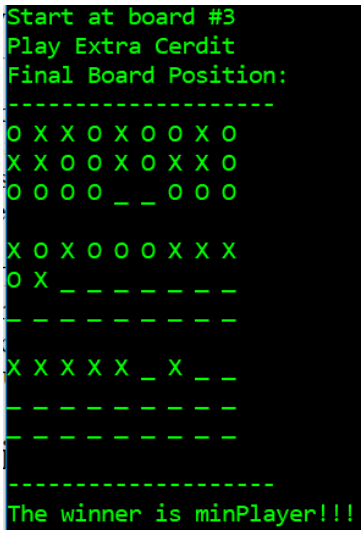

Figure 10. Extra credit final board position I

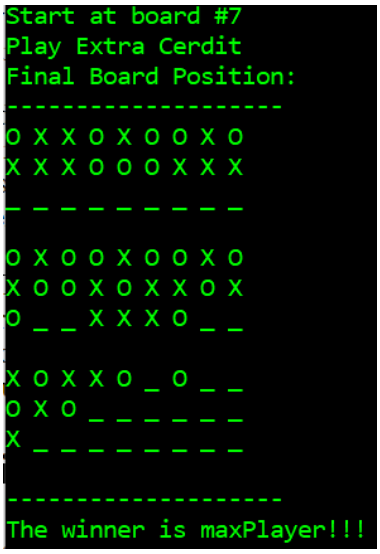

Figure 11. Extra credit final board position II

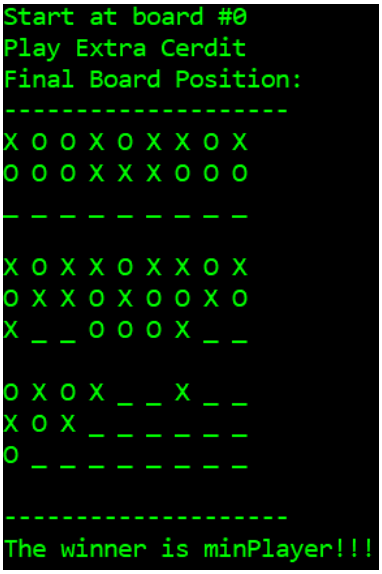

Figure 12. Extra credit final board position III

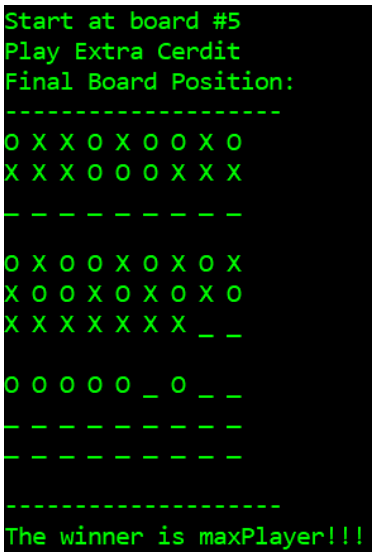

Figure 13. Extra credit final board position IV

Statement of Contribution:

All members mostly did all of the algorithm on our own different versions and compared the result to ensure the result is as required. But for CSP algorithm we used Yuhao Min's code. For section 2.2 we use currently using Heting Fu and Yuhao Liu's version of code.

References:

1. Knuth, Donald E. "Dancing links." arXiv preprint cs/0011047 (2000).
2. Kapanowski, Andrzej. "Python for education: the exact cover problem." arXiv preprint arXiv:1010.5890 (2010).