**Assignment 02 – PLANNING, GAMES**

***Chentai Yuan, Qianzhong Chen, Hao Ding***

***ECE448, Group 07***

***TA: Shuting Tao, Hanrong Zhang***

***March 12, 2023***

**Table of Contents**

**1. Description (Algorithm)2**

**2. Ultimate Tic-Tac-Toe 3**

**3. Offensive agent vs Your agent 4**

**4. Human vs. Your agent6**

**5. Extra Credit 7**

**6. Acknowledgments 9**

**Section Ⅰ: Description (Algorithm)**

For this exact cover problem, firstly, we find out all of the transformers of three types of blocks (domino, triomino, pentomino) by rotation and flipping. We keep a list to record all of the transformers for each shape. In total, there are 63 different pentominoes. Then, for each board, we only place single one transformer but try it for every place of the board. If no conflict occurs (no 0 is covered), we record the tilled board and the upper left coordination of the block transformer. So far, we have gained all of the elements we need to finish tilling task, if we overlap all of the boards we found before, we will find out that all of the 1s are covered. Now, we need to get rid of repetitions and pentominoes overlapping issues.

We took advantage of Algorithm X’s great performance in solving exact covering problem. For every board we found before, we changed its shape to a single row. For instance, the board was initially a (x, y) matrix, but now being transformed into (1, x\*y), every column of the row represents an entry of the board. By data processing, 1 means that this entry is tilled and 0 means no object take this position. Afterwards, we put all of the transformed rows together from top to below, forming a big matrix. Now, the exact covering problem has been simplified to selecting several rows, forming a new matrix to make sure every column has and only has one 1.

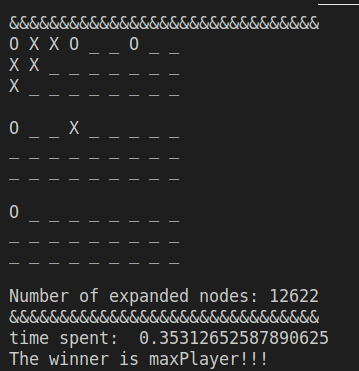
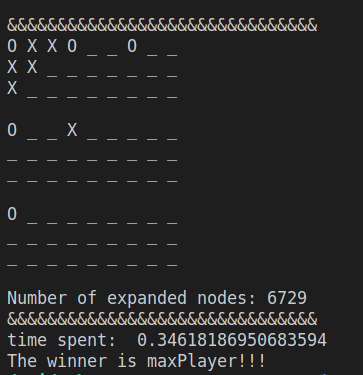
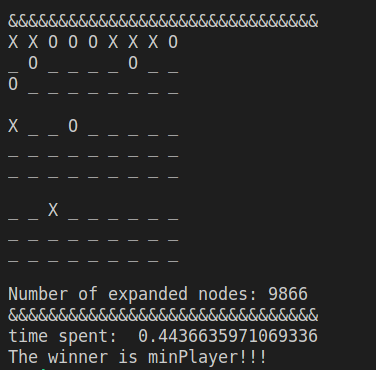
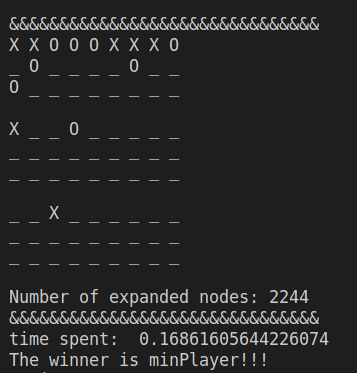
Algorithm X works in the following pattern: The algorithm selects a column in the matrix (referred to as the "pivot column") and tries to select one of the rows in that column to be included in the exact cover. If a row is selected, all other rows that contain elements in the same columns as the selected row are removed from consideration, and the process continues recursively with the reduced matrix until a valid exact cover is found. If at any point the algorithm reaches a dead end (i.e., it cannot find a valid row to select for the pivot column), it backtracks to the previous pivot column and tries a different row in that column. If there are no more rows to try in the previous pivot column, the algorithm backtracks further until it finds a pivot column where there are still rows available to try.

While implementing Algorithm X, some heuristics are used naturally.

Column Ordering: To improve performance, Algorithm X selects pivot columns in a specific order that the algorithm selects the pivot column with the fewest 1s first, and then recursively selects pivot columns in increasing order of the number of 1s in each column.

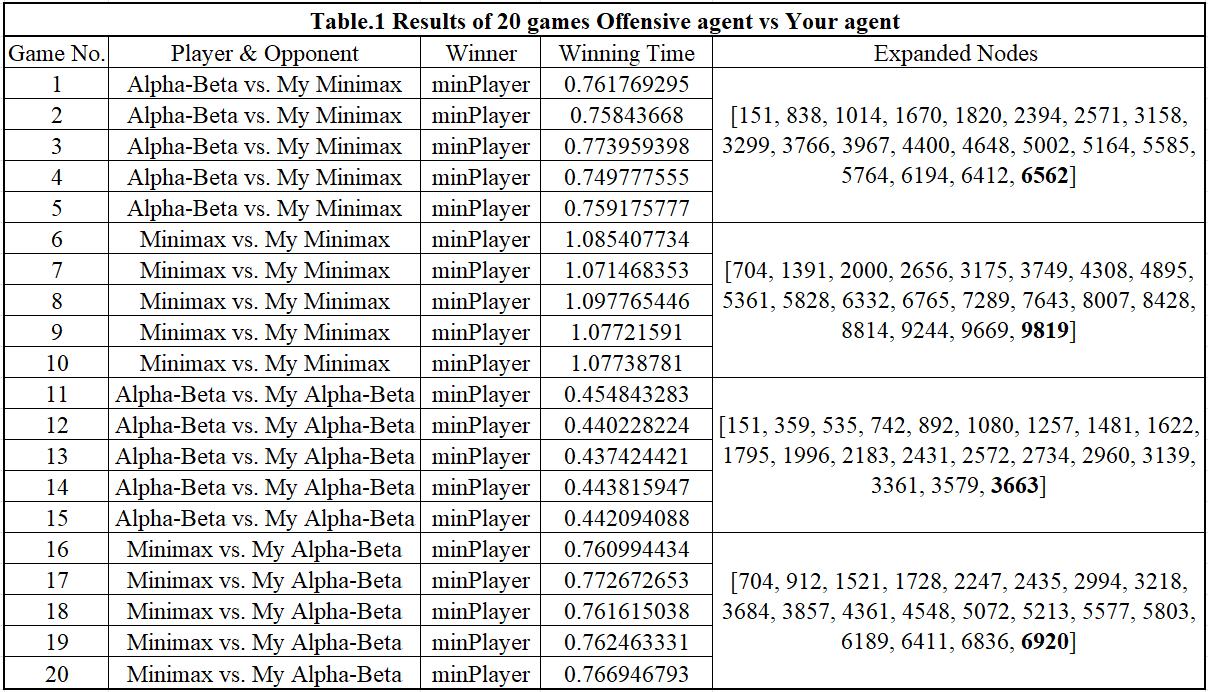
Branch and Bound: Branch and bound involves pruning parts of the search tree that are known to lead to invalid solutions, which can greatly reduce the amount of backtracking required.

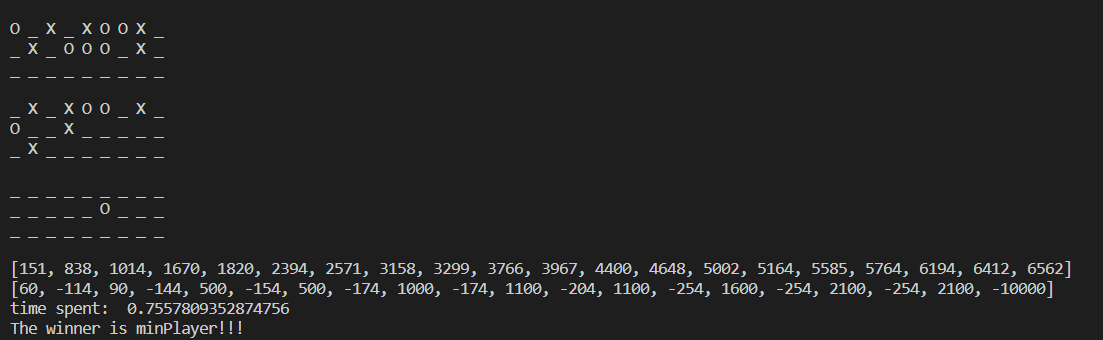
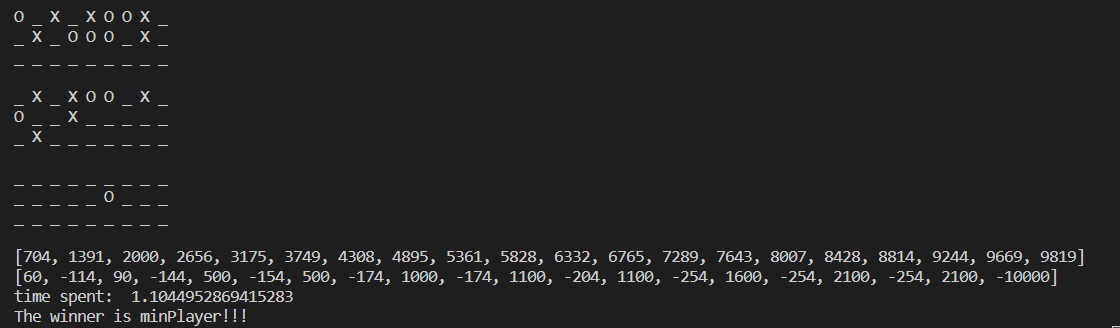
**Section Ⅱ: Ultimate Tic-Tac-Toe**

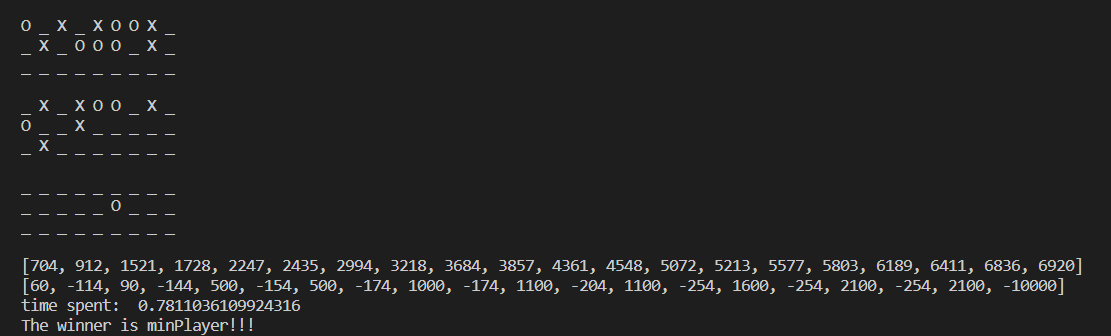
1. offensive(minimax) vs defensive(minimax), maxPlayer go first:
2. offensive(minimax) vs defensive(alpha-beta), maxPlayer go first:
3. offensive(alpha-beta) vs defensive(minimax), minPlayer go first:
4. offensive(alpha-beta) vs defensive(alpha-beta), minPlayer go first:

**Section Ⅲ: Offensive agent vs Your agent**

In my evaluation function, I added an evaluation rule 4 as calculating the distance bonus between the pieces on the board. The base points of distance will be 10, while the points will get smaller as the agent decides to put its piece (score += (10 - max(max\_distances))). Finally, a smaller distance will lead to larger bonus points. That evaluation rule will incline the agent to put their piece near to the exciting piece.

The final results for 20 games are shown in Table.1 below. The winner will always be my agent.

1. Offensive agent (Alpha-Beta) vs. Defensive agent (my Minimax):
2. Offensive agent (Minimax) vs. Defensive agent (my Minimax):
3. A picture containing graphical user interface

   Description automatically generatedOffensive agent (Alpha-Beta) vs. Defensive agent (my Alpha-Beta):
4. Offensive agent (Minimax) vs. Defensive agent (my Alpha-Beta):

**Section Ⅳ: Human vs. Your agent**

**Section Ⅴ: Offensive agent vs Your agent**

**Discussion1: Improvements of *Rule 2***

When implementing the evaluation function of Rule2, we finalized the function:

def RuleTwo(self, player\_cur, player\_opp)

So that Rule 2, for each unblocked two-in-a-row, increment the utility score by 500 and for each prevention, increment the utility score by 100, is optimized to fit with both “empty” and “opponent” in one function.

**Discussion2: Other possible advanced evaluation functions**

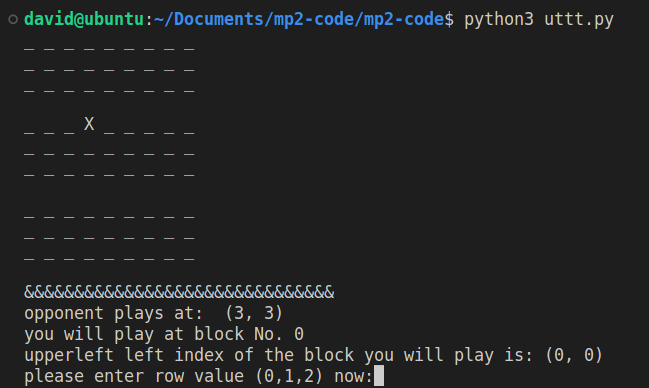
We finally decided to implement the evaluation function “distance bonus” mentioned above in Section3. And we also come up with several possible advanced evaluation function:

1. Open lines: If a player has two of their pieces on an “open line” (An open line is a row, column, or diagonal that contains only empty cells), they can win the game in the next move. Therefore, an evaluation function could assign a higher value to boards with open lines that are occupied by the maximizing player and a lower value to boards with open lines occupied by the minimizing player. For our function, the value is 500 pts. We can try to make the value adjustable for each step.

2. Symmetry: As we searched on the internet, one trick to winning Tic Tac Toe is using symmetry. For example, if the maximizing player has two pieces on opposite corners of the board, they can force a win if the minimizing player makes a mistake. Therefore, an evaluation function could assign a higher value to boards with this particular symmetry on left & right and diagonally.

3. Winning patterns: We can control the agent aiming at putting the piece as some specific patterns to win games easier.

**Discussion3: Human player interface**

We optimize the interface better shown below.

**ACKNOWLEGEMENTS**

Statement of Contribution:

Qianzhong Chen developed the CSP algorithm and developed the uttt.py individually and his code is submitted as solve.py and uttt\_cqz.py. He is responsible for Section#1 and Section#2 of the report.

Chentai Yuan implemented the Assignment of uttt.py and wrote the report paper for section#3 and section#5. And he also helps to debug.

Hao Ding implemented the human agent and helped with debugging. He wrote the report paper for section#4.

**REFERENCE**

1. " Assignment 2: Planning, Games - ECE448 Spring 2023 Assignment#2 Manual."
2. Wikipedia on Pentomino game: “https://en.wikipedia.org/wiki/Pentomino”