Assignment 02 – PLANNING, GAMES

Chentai Yuan, Qianzhong Chen, Hao Ding

ECE448, Group 07

TA: Shuting Tao, Hanrong Zhang

April 5, 2023

Table of Contents

MP02 REPORT – PLANNING, GAMES – GROUP 07

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.

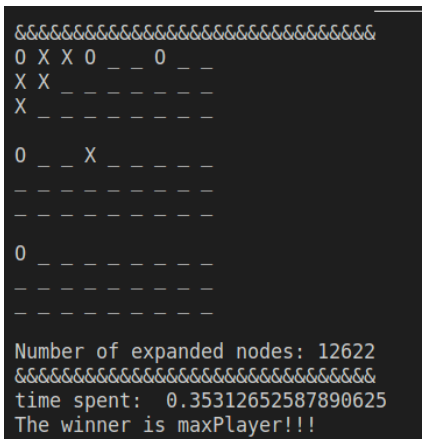
While implementing Algorithm X, some heuristics are used naturally.

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.

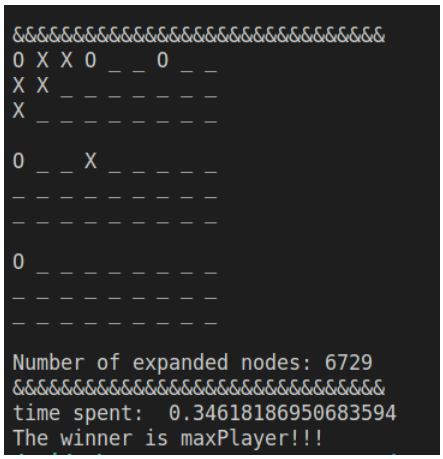
Branch and Bound: Branch and bound involves pruning parts of the search tree that are known to lead to invalid solutions, which can greatly reduce the amount of backtracking required.

Section II: Ultimate Tic-Tac-Toe

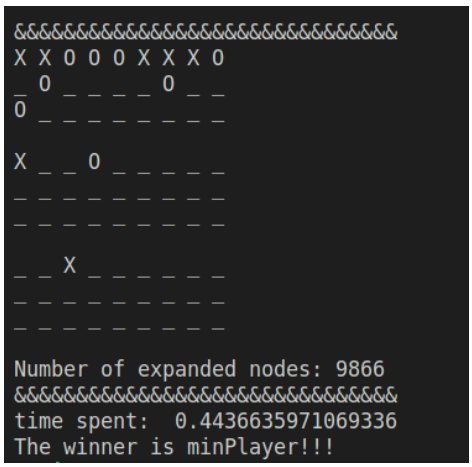
1. offensive(minimax) vs defensive(minimax), maxPlayer go first:



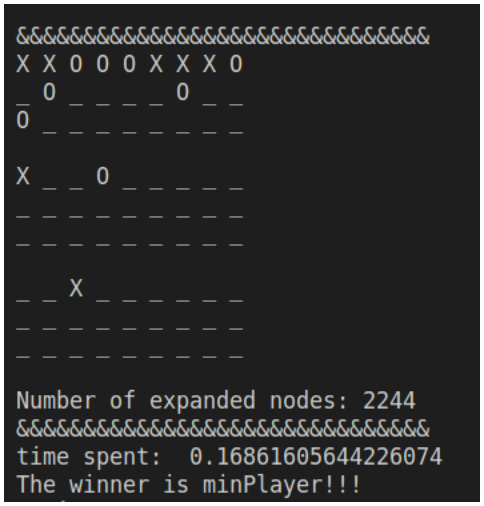
2. offensive(minimax) vs defensive(alpha-beta), maxPlayer go first:



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

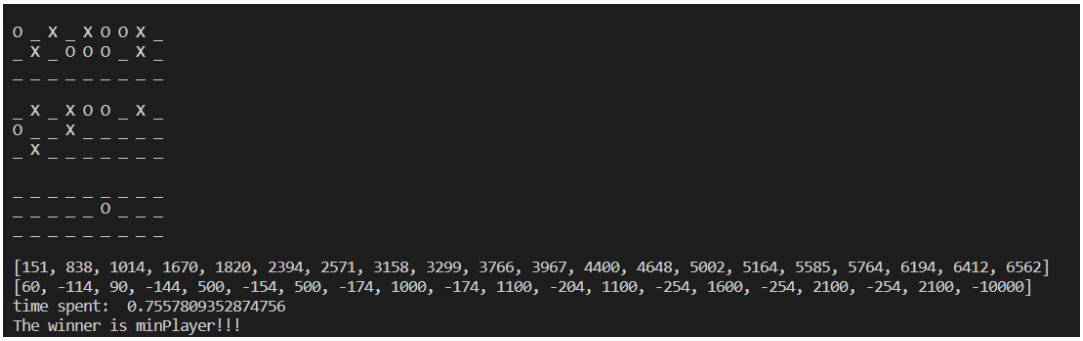
In my evaluation function, I added an evaluation rule 4 as calculating the distance bonus between the pieces on the board. The base points of distance will be 10, while the points will get smaller as the agent decides to put its piece (`score += (10 - max(max_distances))`). Finally, a smaller distance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

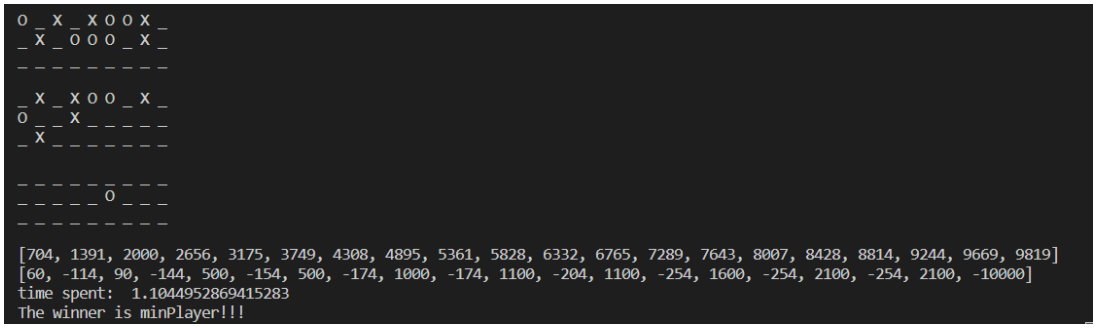
Table.1 Results of 20 games Offensive agent vs Your agent

Game No.	Player & Opponent	Winner	Winning Time	Expanded Nodes
1	Alpha-Beta vs. My Minimax	minPlayer	0.761769295	[151, 838, 1014, 1670, 1820, 2394, 2571, 3158, 3299, 3766, 3967, 4400, 4648, 5002, 5164, 5585, 5764, 6194, 6412, **6562**]
2	Alpha-Beta vs. My Minimax	minPlayer	0.75843668	
3	Alpha-Beta vs. My Minimax	minPlayer	0.773959398	
4	Alpha-Beta vs. My Minimax	minPlayer	0.749777555	
5	Alpha-Beta vs. My Minimax	minPlayer	0.759175777	
6	Minimax vs. My Minimax	minPlayer	1.085407734	[704, 1391, 2000, 2656, 3175, 3749, 4308, 4895, 5361, 5828, 6332, 6765, 7289, 7643, 8007, 8428, 8814, 9244, 9669, **9819**]
7	Minimax vs. My Minimax	minPlayer	1.071468353	
8	Minimax vs. My Minimax	minPlayer	1.097765446	
9	Minimax vs. My Minimax	minPlayer	1.07721591	
10	Minimax vs. My Minimax	minPlayer	1.07738781	
11	Alpha-Beta vs. My Alpha-Beta	minPlayer	0.454843283	[151, 359, 535, 742, 892, 1080, 1257, 1481, 1622, 1795, 1996, 2183, 2431, 2572, 2734, 2960, 3139, 3361, 3579, **3663**]
12	Alpha-Beta vs. My Alpha-Beta	minPlayer	0.440228224	
13	Alpha-Beta vs. My Alpha-Beta	minPlayer	0.437424421	
14	Alpha-Beta vs. My Alpha-Beta	minPlayer	0.443815947	
15	Alpha-Beta vs. My Alpha-Beta	minPlayer	0.442094088	
16	Minimax vs. My Alpha-Beta	minPlayer	0.760994434	[704, 912, 1521, 1728, 2247, 2435, 2994, 3218, 3684, 3857, 4361, 4548, 5072, 5213, 5577, 5803, 6189, 6411, 6836, **6920**]
17	Minimax vs. My Alpha-Beta	minPlayer	0.772672653	
18	Minimax vs. My Alpha-Beta	minPlayer	0.761615038	
19	Minimax vs. My Alpha-Beta	minPlayer	0.762463331	
20	Minimax vs. My Alpha-Beta	minPlayer	0.766946793	

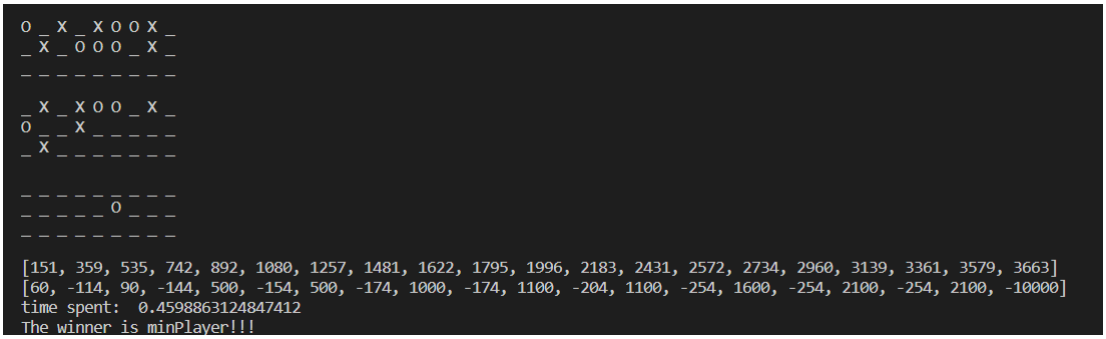
1. Offensive agent (Alpha-Beta) vs. Defensive agent (my Minimax):



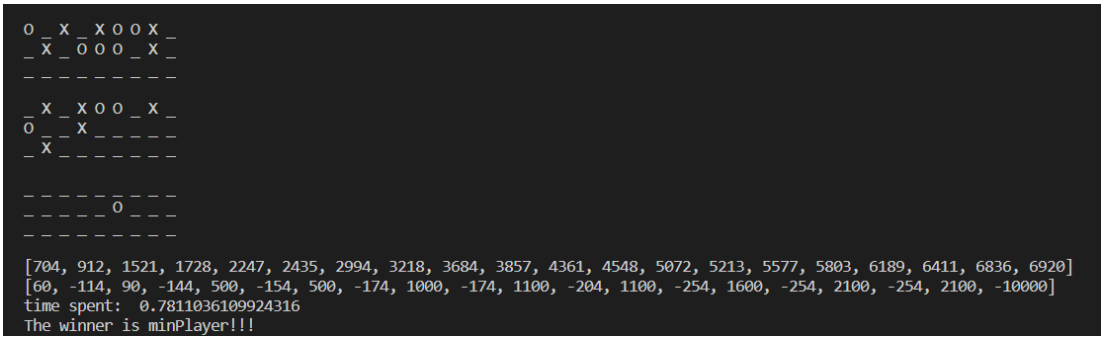
2. Offensive agent (Minimax) vs. Defensive agent (my Minimax):



3. Offensive 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

I have played at total 10 games with our agent， and only won 3 times. The percentage of winning time of our agent is about 30%. According to my observation, there are some advantages and disadvantages of our defined evaluation functions:

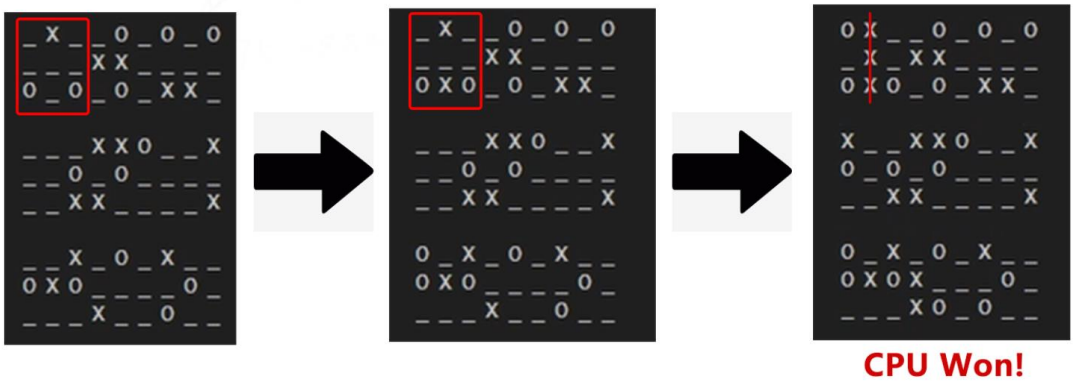
Advantages:

1. No mistakes.

   Unlike me (humans) who sometimes make mistakes during the game and allow the opponent to win easily, our agent does not make low-level mistakes, but instead persists until it reaches a deadlock or achieves victory every time.

2. Balancing offense and defense.

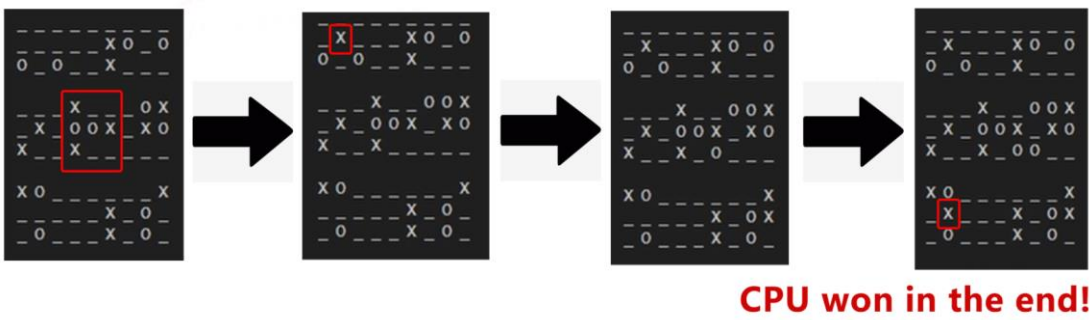
   Whenever possible, our agent tries to form a two-in-a-row while also blocking the third spot of my two-in-a-row, which often allows it to turn the tide of the game.



   As shown in the diagram above, it is the agent's turn to place an "X" in the local board highlighted in red. There are many ways to form a two-in-a-row, but it chooses the one that simultaneously blocks the third spot of my two-in-a-row, causing my offense to fail in this local board. In the end, it wins the game.

3. Reduce opponent's score.

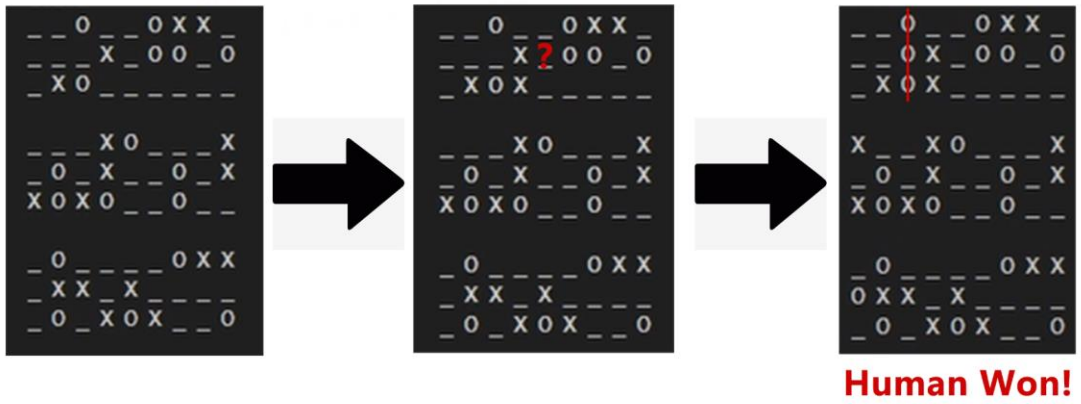
   Our defined evaluation functions strive to reduce the opponent's score, which can be summarized as "wasting the opponent's moves, thereby indirectly increasing one's own advantage".



This strategy is particularly evident in the above game. As shown in the first figure, the central local board is already meaningless to me (O), yet our agent (X) showed a strong tendency to let me make moves in the center local board, wasting many of my opportunities and causing me to lose the game.

Disadvantage:

In the game, I didn't find any big flaw in our defined evaluation functions. Judging from the win rate, our agent's skills surpass mine, and I must be highly cautious every time I play against it. However, sometimes our agent exposes its "short-sighted" flaw, that is, it cannot predict possible future outcomes after many moves.



As shown in the first figure above, in the case where the agent (X) plays first, the wrong choice led to its defeat in two rounds. However, placing X at the question mark position would somewhat delay the arrival of defeat, and even give it a chance to turn the tables.

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:

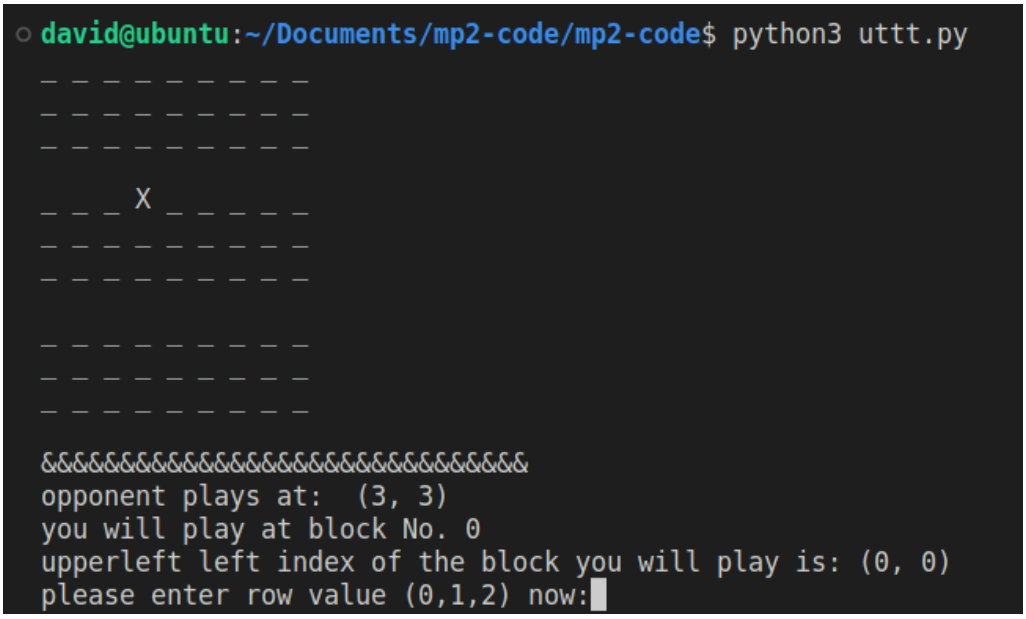
1. Open lines: If a player has two of their pieces on an "open line" (An open line is a row, column, or diagonal that contains only empty cells), they can win the game in the next move. Therefore, an evaluation function could assign a higher value to boards with open lines that are occupied by the maximizing 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 uttt.py and 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”