Henry Solberg (R4) and Ayush Sinha (Q4)
CS 440 MP2
2/20/2019

Section 1 (New version- corresponding to Henry's solve.py):
We began with the idea of using backtracking to solve the problem like a CSP. Our algorithm works by assigning values (corresponding to pentominos) to variables (corresponding to squares). We start in the bottom right corner of the board, and only consider pentominos which cover squares up and to the left of the square. This could be considered a sort of "forward checking" in that we are able before even beginning the algorithm to eliminate values from the domains of our variables. We do not, however, delete pentominos that cause collisions from domains of squares during the algorithm. This is because due to the bitwise board representation technique explained below, it is faster to simply check pentominos for collisions before placing them. If it ends up that all possible pentominos produce a collision, we of course backtrack. When we have assigned values to each variable, we are done solving the CSP.

After beginning in the bottom right, we proceed along each row one by one, right to left, bottom to top. This could be considered a "least remaining values" heuristic because squares in areas surrounded by walls and other pentominos will naturally have fewer valid options remaining for pentomino placement. We also rotate the board so that there are always more rows than columns. We decided to do this because many pentominos immediately create unfillable holes when placed near walls. Filling in the short axis first ensures that we search the area around a pentomino for unfillable squares soon after placing the pentomino. This prevents early undetected failure and drastically increases performance of the algorithm in some cases. It can be considered a form of "failure detection."

The main insight which allowed this technique to perform quickly was representing the board as simply an integer. Since integers are (obviously!) stored as sequences of zeros and ones in memory, we choose the integer corresponding to zeros for unfiled square and ones for filled squares. For example, consider this pentomino board:

```
_ _ 1
2 2 1
```

Where _s correspond to unfilled squares and 1s and 2s correspond to placed dominos. This board would be represented by the integer 15 (001111 in binary). What this permits is something very performant: a pentomino can be placed simply by using bitwise or, and we can test for collisions simply with bitwise and.

I got the idea to use integers for board representation from these two sources: https://www.cl.cam.ac.uk/~mr10/backtrk.pdf "Backtracking Algorithms in MCPL using Bit Patterns and Recursion by Martin Richards " and http://webhome.cs.uvic.ca/~ruskey/classes/Knuth4A/BacktrackChpt.pdf Chapter 3 of Frank Rusky's monograph "Combinatorial Generation." Additionally, I consulted the wikipedia pages on Algorithm X and Pentominos.

Overall, this approach bears some resemblance to the “Algorithm X” approach described below. The main difference is this algorithm does not delete subsets which cover members that have already been covered. Instead, it reduces the occurrence of overlap by covering the farthest/lowest square at each step and only considering dominos which cover exclusively nearer/higher squares than the square being considered. Additionally, the algorithm does not choose subsets based on a strict “least remaining variable” calculation, but rather the shapewise heuristic described above.

Section 1 (Old version- corresponding to Ayush’s (the original) solve.py):
We converted the problem of pentominos tiling to an exact cover problem following the below mentioned steps:

1. Construct a 2D table with one column for each pentomino and one column for each square on the board.
2. Adding rows: Choose one pentomino and one of its orientation and find one of its possible placement on the board. The row corresponding to this possibility will have ‘1’ at column corresponding to the pentomino chosen and ‘1’s at columns corresponding to the squares on board it occupies.
3. Repeat step 2 for all pentominos, all its orientations and for all possible placements on the board.
4. A subset of all these rows which has cumulatively only one ‘1’ for each column is the solution for the pentominos tiling problem.

<u>Example explaining above steps:</u>
Tile A : XX
Tile B : XX
XX
Board :
000
000 (index squares 0-5 from top-left to right-bottom)

Table:
(Columns A,B for 2 Tiles and 0-5 for squares on board)

A	B	0	1	2	3	4	5
1		1	1				

This row represents board state
XX0
000

Growing the table by adding rows following step 2 above:

Row id	A	B	0	1	2	3	4	5
a	1		1	1				
b	1			1	1			
c	1					1	1	
d	1						1	1
e	1		1			1		
f	1			1			1	
g	1				1			1
h		1	1	1		1	1	
i		1		1	1		1	1

Rows e and i together exactly cover all the columns. Hence {i,e} is one possible solution.
e i i
e i i

After converting the pentominos problem to exact cover problem, we use backtracking to find rows that exactly cover the columns. Thus, this is now a **CSP with columns as variables and rows as values along with constraints:**

1. No two rows selected must have ‘1’ in the same column
2. All columns have exactly one ‘1’

Algorithm X is known to be an effective method to solve exact cover problem. We designed our algorithm taking inspiration from algorithm X. Our method is **depth-first backtracking search with Least Remaining Values (LRV) heuristic and Forward Checking**.

Our algorithm can be broken down in following steps:

1. Choose column C with least number of ‘1’s (LRV to choose variable)
2. Choose topmost row R with ‘1’ in chosen column (value chosen from ordered domain)
3. Delete all columns where R contains ‘1’, and delete all rows which have ‘1’ in deleted columns (Forward Checking)
4. Go to step 1 with this reduced table until one of the following occurs:
    a. All rows (values) are deleted but columns (variables) are still remaining: This means solution doesn’t exist for current choices and hence algorithm backtracks to last variable-value assignment.
    b. Al rows and all columns are deleted: This means a solution is found. The variable-value assignments are returned.

Section 2:
As instructed, here are the four games, with the player who takes the first move listed first.

Game 1: Max (minimax) vs Min (minimax). Winner: 1 (max) in 9 turns.

```
O _ X O _ _ O X _
_ X _ _ _ _ _ _ _
X _ _ _ _ _ _ _ _

_ _ _ X _ O _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
```

Expanded nodes: [793, 775, 688, 742, 588, 651, 635, 668, 400]

As expected, expanded nodes are high since both players are using minimax with no pruning. Although it may look like O has handed X a win here, this is a correct application of the predefined agent rules. When O plays in the top left of the second game board, it is because O knows X can always hold O to a node O evaluates at -120. Ending the game also has a value of -120, so O plays in the top left square (since ties go to topmost and then leftmost squares).

Game 2: Max (minimax) vs Min (alphabeta). Winner: 1 (max) in 9 turns.

```
O _ X O _ _ O X _
_ X _ _ _ _ _ _ _
X _ _ _ _ _ _ _ _

_ _ _ X _ O _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
```

Expanded nodes: [793, 296, 688, 293, 588, 265, 635, 301, 400]

As expected, the result is the same, but the min player evaluates fewer nodes because it is using alpha beta pruning.

Game 3: Min (minimax) vs Max (alphabeta). Winner: -1 (min) in 13 turns.

```
X _ O X _ X X O _
_ O _ _ _ _ _ O _
O _ _ _ _ _ _ _ _

_ _ _ O _ X _ O _
_ _ _ _ _ X _ _ _
_ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
```

Expanded nodes: [793, 296, 688, 293, 588, 246, 635, 344, 531, 273, 621, 272, 359]

Again the players play according to their predefined rules and this time max evaluates fewer nodes due to alpha beta pruning.

Game 4: Min (alphabeta) vs Max (alphabeta). Winner: -1 (min) in 13 turns.

```
X _ O X _ X X O _
_ O _ _ _ _ _ O _
O _ _ _ _ _ _ _ _

_ _ _ O _ X _ O _
_ _ _ _ _ X _ _ _
```

```
_ _ _ _ _ _ _ _ _


_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
```

Expanded nodes: [240, 296, 255, 293, 219, 246, 253, 344, 206, 273, 285, 272, 170]

As expected, the same result as the previous game with fewer nodes evaluated for min.

Section 3:

Simple agent:

It is very easy to design an evaluation function which beats the predefined offensive agent more than 80% of the time. Since the offensive agent does not have a concept of losing, it will often give its opponent opportunities to win. Any agent which capitalizes on these opportunities will perform well. We implemented a simple agent which evaluates winning positions as 1, losing positions as -1, and all other positions as 0. (Or, if it is the minimizer, winning as -1 and losing as 1.)

A random playthrough of 60 games vs. this simple agent results in the simple agent winning 90% of the time. One common theme is that the offensive agent would gain a prevention, and then after that point most states would simply look like value 100 (the value of one prevention). So, the offensive agent would simply play in the upper-leftmost square each time until the simple agent saw an opportunity to win. There are many such opportunities for simple agent to win early, since offensive agent does not have a concept of losing. Only when games go long and the possibility of a 4-turn-deep victory arises can this simple agent lose.

Our agent:

This simple agent can be improved with ideas from offensive agent. In addition to not considering losing states, we found another problem with offensive agent is that if it gains a prevention or a two-in-a-row on any board, and then is asked to play on a board where it cannot gain another prevention or two-in-a-row, it will simply play on the top-left most square whereas intuitively a corner may be preferred, since nodes are no longer evaluated according to the third rule. We had the agent evaluate local boards separately so that if the second rule is applied on one board, the third rule could still be applied to another board if the second rule is not applicable locally. With these modifications, we have not been able to find a starting local board/starting player combination where our agent was defeated by the offensive agent.
Here are some example games:
Example 1:
My agent starts
starting board: 7
MyHeuristic vs. Offensive. Winner: -1 in 11 turns. (-1 is our agent)

```
X _ O _ _ _ X _ O
_ O _ _ _ _ X _ _
O _ _ _ _ _ _ _ _

_ _ _ _ _ X _ _ _
_ _ _ _ _ _ _ _ _
O _ _ _ _ _ _ _ _

X _ _ O _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
Expanded nodes: [240, 296, 255, 293, 223, 268, 249, 218, 282, 266, 156]
```

This is a basic example of our agent taking advantage of offensive agent's weakness: it allows its opponents winning moves, especially after rule 3 is no longer used. In this case, X had no hope of gaining another two in a row or a prevention in the next 3 moves, and so allowing O to end the game was the same as playing on.

Example 2:

```
Offensive agent starts
starting board: 8
MyHeuristic vs. Offensive.  Winner: -1 in 18 turns.
O _ X _ X O O X _
_ X _ _ O _ _ X _
O _ _ O _ _ _ _ _

_ _ _ _ O O _ _ _
_ _ _ X _ _ _ _ _
O _ _ _ _ _ _ _ _

X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
Expanded nodes: [240, 296, 255, 293, 219, 276, 245, 343, 198, 304, 259, 250, 223, 327, 261,
233, 225, 179]
```

Although games do not usually go long, this is among the longer games played between our agent and offensive agent. In this case, O did not immediately gain two-in-a-row on the upper left board, because X did first, and then did not send O there in order to avoid losing its two-in-a-row. So, X spent more turns getting two-in-a-rows before giving O a win in the top-middle board.

Example 3:

```
My agent starts
starting board: 4
MyHeuristic vs. Offensive.  Winner: -1 in 11 turns.
X _ O _ _ _ X _ O
_ O _ _ _ _ X _ _
O _ _ _ _ _ _ _ _

_ _ _ O _ X _ _ _
_ _ _ _ _ _ _ _ _
O _ _ _ _ _ _ _ _

X _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
[240, 296, 255, 293, 223, 246, 265, 229, 299, 275, 167]
```

This game illustrates the offensive agent's strong tendency to give its opponents wins in the upper-left board after switching to rule 2, since it defaults to playing in the upper left when it cannot find a leaf node that improves on its rule 2 evaluation.

Example 4:

```
Offensive agent starts
starting board: 2
MyHeuristic vs. Offensive.  Winner: -1 in 14 turns.
O _ X _ _ _ X _ O
O X _ _ _ _ X _ _
O _ _ _ _ O _ _ _

O _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
_ _ _ O _ _ _ _ _

X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
[240, 295, 256, 265, 231, 294, 216, 277, 268, 185, 235, 335, 238, 147]
```

Another example of the dynamics described above, with the twist that X played in the upper left corner of the lower left board and gave O a win because doing so allowed it to get another two-in-a-row (although it meant sacrificing a two-in-a-row on the upper left, so still it did not improve its rule 2 evaluation).

Monte Carlo Tree Search:
Just for practice, we also implemented a much “higher-powered” (but slower) heuristic: pure monte-carlo game search. This heuristic operates by playing out random games and evaluating the node based on the results of those games. Our agent plays out 20 games at each node. Since 20 is larger than the number of children of game nodes, our agent is likely to evaluate nodes with winning moves available highly. (One out of however many children the node has will immediately result in victory, bumping up the evaluation for that node.) For the same reason, it evaluates losing positions poorly. In dense boards, it also favors nodes with plays that force the opponent to allow it a victory, since the limited number of children make it likely that this situation will be represented in one or more of the 20 random playthroughs. This effectively allows the agent to “look deeper” than the 3 moves other agents are allowed.

Since all routes lead to victory or loss on nodes which are already won or lost, the agent also has all the advantage of the simple agent in that lost nodes will be evaluated minimally and won nodes maximally.

Here are some representative games:

Example game 1:
Offensive agent starts
starting board: 2
MCTSHeuristic vs. Offensive. Winner: -1 in 10 turns.

```
O_X X__ XO_
_O_ ___ _O_
__O ___ ___

___ X_X ___
___ ___ ___
___ ___ ___

___ ___ ___
___ ___ ___
___ ___ ___
```

This game illustrates our agent’s ability to capitalize on offensive agent’s weakness: it will always take winning moves when offensive agent gives them to it. This game also shows that our agent favors plays which make two in a row. This is because random playthroughs on boards with two in a row will be more likely to result in victory when the third square gets randomly filled in. This is of course more likely than both a second AND third square getting filled in, so our agent favors nodes with two in a row.

Example Game 2:
MCTS agent starts

```
starting board: 2
MCTSHeuristic vs. Offensive.  Winner: -1 in 11 turns.
X__ ___ O__
OOO O__ _O_
___ ___ ___

X__ XX_ __X
___ ___ ___
___ ___ ___

___ ___ ___
___ ___ ___
___ ___ ___
```

Another illustrative game for capitalizing on offensive agent's weakness. Here, offensive agent gave our agent a winning move in order to get two in a row.

Example Game 3:
MCTS agent starts

```
starting board: 1
MCTSHeuristic vs. Offensive.  Winner: -1 in 23 turns.
OOO X_X XO_
__O ___ XO_
__X __O ___

O__ X__ __X
___ ___ ___
X__ X__ ___

___ ___ X_O
O__ ___ _O_
O_X ___ ___
```

This board shows our agent's performance in somewhat longer games. The agent's favor for two in a row is again illustrated here.

Example Game 4:
MCTS agent starts

```
MCTSHeuristic vs. Offensive.  Winner: 1 in 34 turns.
O_X X_O _OO
__X ___ _O_
O__ O_X XXX
```

```
_XX XXO ___
__O ___ OX_
___ _O_ _X_

_O_ ___ O__
X__ OXO _O_
XO_ __X ___
```

We actually had to play substantially more than 20 games in order to find one in which the monte carlo agent lost. In this case, we can see that the game went fairly long. As discussed above in the context of the simple win/lose agent, this is necessarily the case since the agent losing requires a four-deep forced loss. Since the agent's heuristic is fundamentally stochastic, it may sometimes miss important features of a node if routes exploiting that feature are not randomly selected, which is what happened here.

Section 4:
In general ultimate tic tac toe strategy, in seems important to maintain "control" of as many boards as possible, in that you want to be able to place a piece in the corresponding square without allowing your opponent to win, and you do not want your opponent to be able to place a piece in the corresponding square without allowing you to win. This will allow you to keep placing pieces and eventually your opponent will be forced to place a piece in a square that leads to a board you "control."

When you send your opponent to such a board, this is a killing move, but since our designed agents only look 3 moves deep, they will not consider giving you such a move to be a losing strategy. (Because they still get another safe move before losing, and their evaluation only goes up to that safe move.) So, in order to beat computer opponents which only look 3 moves deep, we adopt the strategy of maintaining "control" and waiting until a killing move appears.

Of course, our agent is designed so that it also maintains "control" as much as possible by getting two-in-a-row and preventions. This is an advantage the agent has- it fights for control and never makes sloppy mistakes. So, our hope is to fight for control for long enough to allow killing moves to appear, and then use our ability to sense killing moves deeper than our opponent does in order to win.

In ten games, the human and designed agent split them 5-5. (Human wins 50%). It could be argued that the human's losses were mainly due to "silly mistakes" such as entering a wrong number. However, sometimes the agent was simply better at control- see example 2 below.

Example 1:

Player first, start board: 4

```
O _ X _ O O O _ _
_ O X _ _ _ _ _ _
X _ _ _ X X X X X

X X _ X _ _ _ _ _
O _ O X _ O X _ _
O _ _ _ _ O X O O

_ O O _ X _ O _ O
X X _ O _ _ _ X X
_ _ _ O _ _ _ _ _
```

6
Human vs. designed. Winner: 1 in 37 turns.

Clearly, this game went longer than games involving offensive agent. To beat our agent, we simply look ahead more than 3 moves. This will allow us to make “killing moves” the agent could not see by sending the agent to a board where it is forced to make a play that allows us to win (the leaf where we win was 4 levels away when the agent was evaluating its previous move.)

Example 2:
Player first, starting board 1

```
O _ X X O _ O _ X
O _ O X _ X _ X _
X _ _ O O _ O _ X

X X _ _ O O X X _
O _ X _ _ X O O _
O O _ X X _ _ _ O

_ X O O _ _ _ _ O
X X O X O _ _ O _
_ _ _ _ X O _ X X
```

Human vs. designed. Winner: -1 in 46 turns.

I made a mistake in the middle of the game in which I wasted a move without creating or stopping a two-in-a-row threat. For this reason, the designed agent got ahead of me and was able to find a 3-deep killing move before I did.

Example 3:

Player first, starting board: 4

```
O O _ _ X O _ _ X
_ _ X _ X _ O _ _
X O X _ X _ O _ X

X _ _ X _ X _ _ _
X O _ O _ _ _ _ _
_ _ _ _ _ _ O _ _

X O O O _ _ O O _
_ _ _ _ _ _ _ _ _
X _ _ _ X _ _ _ _
```

The “killing move” was to force O to play in the upper left board, which only had four spots remaining, each of which led to O losing. I was a bit lucky in that I controlled nearly only the boards I needed to win, so this game wasn’t very long.

Bonus example, human vs. monte-carlo:
Player first, starting board: 1

```
O _ X X X X _ X X
O O _ O O _ _ O _
_ _ X _ O _ _ _ O

O O _ X O O _ _ _
X X _ _ X _ X _ X
_ O _ X _ O O _ _

_ _ _ _ X _ X _ O
_ X _ _ _ O X _ O
O _ X _ X _ _ _ _
```

Human vs. designed. Winner: 1 in 39 turns.

Although the monte carlo agent has the potential to “see” further than 3 moves ahead, it is also susceptible to random error when fighting for control of boards, making it ultimately easier to beat than the designed agent. Of course, the monte carlo agent could theoretically be improved to be better than any human, and converges to optimal play as more playthroughs are allocated per leaf node.

Extra Credit:
We implemented the longer version of ultimate tic tac toe in which a player must achieve 3-in-a-row on 3 local boards, themselves in a row, to win. We also took the opportunity to

implement agents which lie between the default agents and the monte carlo agents in terms of power.

For this game, our offensive agent and defensive agent were both given awareness of winning and losing moves. Other than this, the offensive agent favored the following board characteristics in this order: 1) Having more 3 in a row 2) The enemy having less 3 in a row 3) Having 2 in a row 4) The enemy having less 2 in a row. And the defensive agent, predictably, favored these: 1) The enemy having less 3 in a row 2) Having more 3 in a row 3) The enemy having less two in a row 4) Having 2 in a row.

We played 10 games of offensive vs. defensive using alpha beta pruning at depth 3. In our 10 games, offensive won 5 times, defensive won 4 times, and there was 1 draw. All games went for more than 40 turns, with the draw being the longest at 55 turns. The reason for the long game lengths is that both players have win/lose built in to their evaluation functions, and so will not make moves that result in unforced losses within 3 turns.

Game 1:

```
X O _ _ X O O O X
X _ _ _ X _ X X O
X _ O _ X _ O X O

O _ O _ O _ X _ O
O X _ _ O X X _ O
O O _ _ O _ X _ _

_ _ O X O X O O X
X X X X X X O X O
O _ _ _ O O O X X
```

[240, 287, 255, 293, 265, 280, 305, 277, 314, 313, 304, 301, 306, 312, 242, 276, 235, 297, 233, 235, 263, 193, 227, 211, 184, 178, 167, 170, 141, 206, 187, 275, 2307, 389, 234, 228, 397, 235, 157, 325, 326, 76, 54, 100, 78, 126, 46, 379, 173, 111, 71, 42, 20, 8, 1]
EC: Min first, starting board 2. Winner: 0 in 55 turns.

The list above is the nodes explored per move in this game. You can see the nodes explored dropping and finally ending at 1 when few legal moves remain. This game ended in a draw, and the upper-right board went to neither player. This is perhaps an example of O's defensive tendencies.

Game 2:

```
X O O _ X O _ _ X
X _ O _ X O O _ X
```

```
_ _ O _ O O _ O X

O X O O _ _ X _ _
_ X _ _ O X X _ O
O _ _ _ _ O X _ _

X X X X O _ _ X _
O _ _ _ _ _ _ X _
O _ _ X _ _ _ X O
```

[240, 287, 255, 294, 274, 315, 273, 268, 306, 294, 305, 282, 309, 295, 245, 308, 244, 229, 263, 182, 232, 182, 243, 234, 232, 237, 178, 189, 183, 161, 139, 170, 104, 142, 85, 231, 266, 2244, 188, 356, 92, 159]
EC: Min first, starting board 3. Winner: 1 in 42 turns.

The shortest game of the ten. It can be seen that X quickly established 2-in-a-row-threats, and eventually O was forced to allow X to win.

Game 3:

```
O X O X O X X _ O
X X O _ _ _ X _ _
_ O O X _ _ X _ _

O O _ O _ O X _ X
O X _ _ O X O X O
X O _ O _ _ X _ _

X _ O _ _ _ X O _
_ X O _ _ X X _ _
O _ O _ _ X X _ _
```

[240, 287, 255, 294, 267, 316, 274, 293, 270, 303, 300, 300, 282, 288, 279, 232, 213, 246, 176, 225, 188, 208, 180, 225, 155, 228, 178, 190, 126, 193, 230, 173, 136, 128, 156, 130, 371, 247, 672, 1077, 317, 93, 1002, 56, 68]
EC: Max first, starting board 8. Winner: 1 in 45 turns.

All games start with the players not repeating boards for several turns. This is because both players assign some value to preventing their opponent from achieving 2 in a row. The 16th move of this game is interesting because it showcases the difference between the two agents:

```
>>> printGameBoard(gameBoards[15])

O X _ _ O X _ _ O
```

```
_ _ _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _
O X _ _ O X _ _ O
_ _ _ _ _ _ X _ _

X _ _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _
O _ _ _ _ _ _ _ _
```

```
>>> printGameBoard(gameBoards[16])

O X _ _ O X _ _ O
_ _ _ _ _ _ X _ _
_ O _ _ _ _ _ _ _

_ _ _ _ _ _ _ _ _
O X _ _ O X _ _ O
_ _ _ _ _ _ X _ _

X _ _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _
O _ _ _ _ _ _ _ _
```

It can be seen that whereas X would have taken the two-in-a-row on the upper-leftmost board, O favored the bottom-middle square, so that X would not be able to gain two-in-a-row on the next turn.

Game 4:

```
O X _ _ O X _ _ O
_ O _ _ O _ X _ O
_ X O _ O _ X _ O

_ _ _ X _ _ X X _
O X O _ X O X _ X
O _ _ _ _ X O X X

O _ _ X X _ O X X
X _ O _ _ O _ _ O
X _ _ _ O _ _ _ O
```

[240, 287, 255, 293, 267, 314, 274, 292, 293, 268, 321, 269, 339, 230, 291, 231, 265, 296, 291, 237, 259, 208, 222, 308, 215, 255, 361, 2214, 755, 456, 508, 393, 2001, 349, 228, 314, 224, 235, 190, 1351, 122, 1163]
EC: Max first, starting board 4. Winner: -1 in 42 turns.

An example of the defensive agent winning. You can see the jumps in nodes evaluated quite clearly. These correspond to when agents were given an "unrestricted move" since their opponent's last move sent them to an already claimed board.

Statement of Contribution:

Henry and Ayush both developed their own versions of every part of the MP except the extra credit (which was developed by Henry). We decided to submit Ayush's code for part 1 because it aligned more closely with concepts taught in the course (Henry's solution used a bitwise board representation and "snaking" search style). The final version of the predefined agent and alphabeta/minimax search was written by Henry, whereas the submitted designed agent was written by Ayush. Henry wrote the MCTS and human vs. designed code. Ayush wrote section 1 of the report, whereas other parts were written by Henry, but checked by Ayush.

For resubmission: Due to concerns about Ayush's code bearing similarities to Algorithm X code published online, we have resubmitted this MP using Henry's approach to part 1. Henry rewrote section 1 of the report.