Artificial Intelligence I - Assignment 3
Tic Tac Toe & Connect 4 using Minimax and Tabular Q-learning

Prathamesh Sai Sankar
5th year Integrated Computer Science
saisankp@tcd.ie
19314123

Abstract

This report analyzes the performance of minimax, reinforcement learning, and a default player for playing Tic Tac Toe and Connect 4. Minimax is implemented with and without alpha-beta pruning. The default player is slightly better than a player choosing randomly since it selects a winning move or a blocking move (stopping the opponent from winning) if it exists. Tabular Q-learning is used for the reinforcement learning algorithm. The comparison between minimax, reinforcement learning and a default player can give us valuable insights into whether each approach is suitable for these games and the advantages and disadvantages of each.

1 Introduction

Algorithms can be used to play games such as Tic Tac Toe or Connect 4. The Minimax algorithm (with and without alpha-beta pruning) and reinforcement learning algorithms (such as tabular Q-learning) are examples which can be used to do so. This report will analyze the performance of both the minimax and reinforcement learning algorithms. To compare the algorithms, we can use metrics such as winning rate, average time taken to win and the average peak memory usage.

2 Game usage

To compare both algorithms, I used an open source Tic Tac Toe library in python called python-tictactoe[1] to generate a 3x3 board for Tic Tac Toe, which can be found on GitHub. Furthermore, I adapted the board in this library to be 6x7 so it can also work for Connect 4, except I changed the requirement so it needs 4 in a row to win (instead of 3). I implemented it so that the user can input the row and column they wish to mark in Tic Tac Toe, and the user can input the column they wish to mark in Connect 4 (since the row changes depending on the number of marks that exist in that column already).

3 Algorithm explanation

3.1 Minimax

The minimax algorithm is used when the player has access to the current state and any previous events, which suits Tic Tac Toe and Connect 4. The decision making from this algorithm aims to get the most optimal move for a player when assuming the opponent also plays optimally by generating a game tree where each node is a game state and edges are potential moves. Alpha-beta pruning is a technique used in the minimax algorithm which makes it more quick at iterating through all possible moves in the game tree. Therefore, the end result is the same without alpha-beta pruning, but it just takes longer. Alpha-beta pruning works by not evaluating every branch in the game tree, and instead ignores certain branches that aren't worth exploring[2]. This distinction is made by keeping track of alpha (highest value found so far) and beta (smallest value found so far).

Alpha and beta are updated as the algorithm explores the game tree, where it ignores a move if it results in a lower value compared to alpha which it knows already. In a similar fashion, it ignores a move if it results in a higher value compared to beta which it knows already. Both of these cases result in not exploring "useless" paths of the game tree because there is a better move available already or there is a better move in a different path. The pruning of these branches in the game tree is called alpha-beta pruning, it reduces the number of nodes that need to be evaluated.

3.2 Tabular Q-learning

Tabular Q-learning is a form of reinforcement learning which is inspired from MDPs (Markov Decision Processes). It aims to make the best moves it can make based on potential rewards and penalties. By iteratively making mistakes and making good moves, it is punished or rewarded based on the results of its actions. This is possible with the use of a Q-table which stores the expected rewards for each potential action for a state. These are assigned randomly initially, then the program uses it when interacting with its environment (in this case, Tic Tac Toe or Connect 4). When the program

receives a punishment or reward, the Q-table is updated to show the rewards that it received from past actions and expected rewards it can receive from future actions.

This is an iterative process which takes time, however it can learn without any context of the environment. This results in a "learn by failing" approach, similar to how a human would learn. It completes actions and keeps track of whether they were good or bad. However, the algorithm needs to explore all potential pairs of states and actions to update the Q-table which is expensive. Furthermore, it needs to balance exploration and exploitation. When exploring, the algorithm can make mistakes to learn what is a correct and incorrect move. When exploiting, the algorithm can use its knowledge to maximize the immediate reward it can get. Balancing this is a challenge!

4 Code explanation

4.1 Minimax

4.1.1 Tic Tac Toe. The `minimax_for_tic_tac_toe` function implements the minimax algorithm for Tic Tac Toe. It takes the board, alpha, beta and boolean maximize variable to determine whether to maximize or minimize the player's turn. The first condition checks if the game is over, for `result` being 1 for a win, 2 for a loss and 0 for a draw. All possible moves are evaluated in a recursive manner if it is maximizing the player's turn. It does so by iterating over all potential moves and evaluating the state of the board. This allows for the update of the `maximum_evaluation_value` (which is initially set to `100` before updating it) and `best_move` (which is initially set to `None` before updating it)[3].

All possible moves are also evaluated in a recursive manner if it is minimizing the player's turn. It also iterates over all potential moves and evaluating the state of the board. This allows for the update of the `minimum_evaluation_value` and `best_move`. For either maximizing or minimizing the player's turn, alpha-beta pruning is not completed if alpha and beta are passed in as `None` and `None`. Alpha-beta pruning is completed if alpha and beta are passed in as values such as `-math.inf` and `math.inf` respectively. The function `make_next_tic_tac_toe_move` calls the minimax algorithm in the Minimax opponent class. The move made by the minimax algorithm is used here.

4.1.2 Connect 4. The `minimax_for_connect_four` function implements the minimax algorithm for Connect 4. It takes the board, depth_limit (for depth-limited search), alpha, beta and boolean maximize variable to determine whether to maximize or minimize the player's turn. If someone has won a game, the first condition checks who exactly won, with `result` being 1 for a win, 2 for a loss and 0 for a draw. All possible moves are evaluated in a recursive manner if it is maximizing the player's turn. It does so by iterating over all potential moves and evaluating the state of the board. This allows for the update of the `maximum_evaluation_value` (which is initially set to `-math.inf` before updating it) and `math.inf` (which is initially set to `None` before updating it). We use positive and negative infinity instead of 100 and -100 from Tic Tac Toe since the game tree is deeper compared to Tic Tac Toe.

All possible moves are also evaluated in a recursive manner if it is minimizing the player's turn. It also iterates over all potential moves and evaluating the state of the board. This allows for the update of the `minimum_evaluation_value` and `best_move`. For either maximizing or minimizing the player's turn, alpha-beta pruning is not completed if alpha and beta are passed in as `None` and `None`. Alpha-beta pruning is completed if alpha and beta are passed in as values such as `-math.inf` and `math.inf` respectively. The function `make_next_connect_four_move` calls the minimax algorithm in the Minimax opponent class. The move made by the minimax algorithm is used here. A depth limit of 5 is used because it is unfeasible to complete minimax since there is a massive increase in the number of states to visit as well as the maximum length of the game[3].

4.2 Tabular Q-learning

4.2.1 Tic Tac Toe. The class `TabularQLearningOpponent` takes alpha (learning rate), epsilon (exploration rate), gamma (discount factor), a Q_table (in a dictionary form, mapping states to Q-values) and states (as a list). As mentioned previously, balancing exploration and exploitation is a challenge with tabular Q-learning. My implementation deals with that by generating a random number from 0 to 1, and checking if it is smaller than epsilon (exploration rate). If it is, then a random move is made (since epsilon is bigger than the random number, hence we do exploration). If it is not, a number with the highest Q-value is used instead (since epsilon is smaller than the random number, hence we do exploitation). This is done by iterating over all potential moves and calculating its Q-value. If the Q-value is greater than `maximum_q_value` (which is set to `-math.inf` initially), then it is the new `maximum_q_value` and the current move is set as the `best_move` (which is initially set to `None`)[3].

This approach balances exploration and exploitation in an interesting way which I did because I was fascinated by how efficient the algorithm would be. In addition to this, it also has a `update_epsilon` function which uses linear decay. Epsilon is reduced by a factor of 0.000001 multiplied by the episode number. A `max()` function is used, with a value of 0.3 being passed so that epsilon never goes below 0.3. With this setup, epsilon decreases gradually as the episode number increases, hence there is less exploration over time. This gives the algorithm to be explorative at the start, and become exploitative at the end. This function is called in every

episode iteration when it is being trained. The Q-learning update logic is in `q_learning_update` which is based from:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (R + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a))$$

This is completed by iterating through the list of all states and initializing the q-value of the current state as 0 if it is not present in the Q-table. Then, the equation is completed to compute `self.Q_table[state]` and the `reward` for the next iteration to calculate the Q-value of the previous state.

4.2.2 Connect 4. The same implementation from Tic Tac Toe is used for Connect 4 in the `TabularQLearningOpponent` class which takes alpha (learning rate), epsilon (exploration rate), gamma (discount factor), a Q_table (in a dictionary form, mapping states to Q-values) and states (as a list). Exploration and exploitation is balanced by generating a random number from 0 to 1, and checking if it is smaller than epsilon (exploration rate). If it is, then a random move is made (since epsilon is bigger than the random number, hence we do exploration). If it is not, a number with the highest Q-value is used instead (since epsilon is smaller than the random number, hence we do exploitation). This is done by iterating over all potential moves and calculating its Q-value. If the Q-value is greater than `maximum_q_value` (which is set to `-math.inf` initially), then it is the new `maximum_q_value` and the current move is set as the `best_move` (which is initially set to `None`).

As mentioned before, this approach balances exploration and exploitation in an interesting way which I did because I was fascinated by how efficient the algorithm would be. The same `update_epsilon` function is used when training, which is common to both Tic Tac Toe and Connect 4. The same Q-learning update logic is in also shared from the function `q_learning_update` as mentioned previously. When checking all potential moves, the only difference between Tic Tac Toe and Connect 4 here is that I push a tuple to the board to make a move in Tic Tac Toe (column and row), and I push a single number to the board to make a move in Connect 4 (column only)[3].

4.3 Design justifications

In terms of my design justifications, I wanted to take this assignment as a learning opportunity to try as much as possible, which is why I implemented linear decay for epsilon as well as more techniques to balance exploration and exploitation (based on the value of epsilon). I set the rewards for winning as 1 and rewards for losing as -1. A tie gave a reward of 0. This made the most sense to me, since disproportionate rewards were hard to justify when simpler design choices are more intuitive here. I made use of the most basic data structures I could for vital things such as a dictionary for the Q-table and a list for the states. I also decided to make each opponent have two functions inside of them, one for making a move in Tic Tac Toe and one for making a move in Connect 4. This made the code more concise and understandable compared to various files for every type of opponent and game it plays.

I reused the Tic Tac Toe board for the Connect 4 game since it was a similar layout, and I could just adjust the height (i.e. row index) based on the number of available places in that column. Taking (0,0) as the top left of the grid, an example of the logic would be that, for a grid of height 6 such as in Connect 4, if column 0 only has vertical from 1-5 (since 0 is taken), I would have to put a mark on (0, 5). Hence I put a mark on (0, `max(self.state_dict[column])`) where `self.state_dict` is a dictionary of all states in the Connect 4 board. This easy extension to the original Tic Tac Toe library was justified because it works intuitively compared to a completely different library. Since both games use the same library, the same utility functions can be shared among them. The training of the tabular Q-learning opponent is captured in a pickle file. I chose this because it is a simple binary representation which is used easily compared to other files.

For values for alpha, epsilon and gamma in the tabular Q-learning opponent, I set it with $\alpha = 0.1$, $\epsilon = 0.9$ and $\gamma = 0.9$. The learning rate α being 0.1 is commonly used, but also helps manage to update of Q-values slowly and not abruptly. The balance between exploration and exploitation is managed by setting ϵ to 0.9 which tends towards exploration, and this will be decreased by linear decay after episodes, with a minimum of 0.3 since any lower would be too exploitative. Using γ as 0.9 is commonly used because it helps manage between rewards in the future and rewards in the present.

5 Comparisons from experimentation

When running experiments, I used the default opponent (who makes moves that are slightly better than random moves), a minimax opponent, and a tabular Q-learning opponent (which uses a pickle file from a trained agent[4]). I trained the Q-learning agent for Tic Tac Toe for 5 million rounds (took around 7 hours) and Connect 4 for 1 million rounds (took around 12 hours) which are in 2 pickle files. Connect 4 took significantly more time for less rounds since it required a larger number of states and had a bigger length for the game. I tried to train for as long as I could because I wanted to push the boundaries and see how effective this opponent could get.

A key thing I learned was that the player who went first had a massive advantage. I found that when I tried a default opponent against another default opponent, the first player

was around 3 times more likely to win since they had started first. Therefore, to make experiments fair I split the numbers of rounds into 2, and swapped who makes the first move in each half. This provides a more fair opportunity for each algorithm.

5.1 How do your algorithms compare to each other when playing against default opponent in Tic Tac Toe

5.1.1 Default vs Minimax (no alpha-beta pruning). Without alpha-beta pruning, minimax is really slow. This is because it has to iterate through all possible moves in the game tree instead of skipping useless ones. From manual experimentation, it took around 6 minutes just to finish 1 game. I completed 10 rounds since it was very slow, but it was evident from all attempts that minimax still always wins or ties against the default opponent:

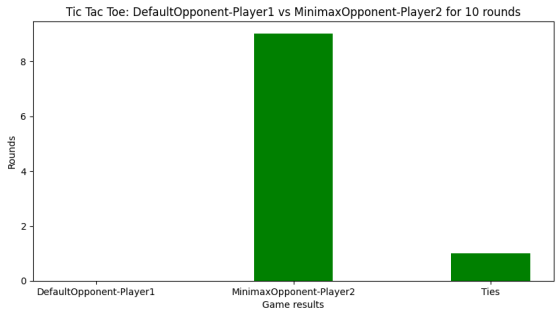

Figure 1. Default vs Minimax (without alpha-beta pruning) for 10 rounds.

	Wins	Average time taken	Average max memory
Default	0	0.0214s	10365.30 bytes
M (WABP)	9	405.5520s	40818.10 bytes
Ties	1	-	-

Table 1. Default vs Minimax (without alpha-beta pruning) for 10 rounds.

The data shows that the default player always loses or ties, but it uses a lot less memory and time compared to minimax. This makes sense because minimax has to iterate through all possible moves in the game tree which uses a lot of memory and time, but makes up for it by always winning against the default opponent. In short, default opponent always loses or ties against minimax without alpha-beta pruning, but it uses considerably less time and memory.

5.1.2 Default vs Minimax (with alpha-beta pruning). With alpha-beta pruning, minimax is much faster. This is because it does not have to iterate through all possible moves in the game tree, hence skipping useless ones. I was able to complete 150 rounds since it was not as slow as before, and it once again evident that minimax still always wins or ties against the default opponent:

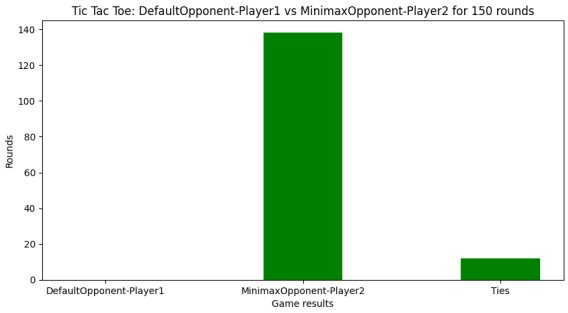

Figure 2. Default vs Minimax (alpha-beta pruning) for 150 rounds.

	Wins	Average time taken	Average max memory
Default	0	0.0203s	9024.21 bytes
M (ABP)	138	14.7227s	37001.52 bytes
Ties	12	-	-

Table 2. Default vs Minimax (alpha-beta pruning) for 150 rounds.

The default player always loses or ties again, but uses a lot less memory and time compared to minimax. Minimax with alpha-beta pruning uses less time and less memory compared to minimax without alpha-beta pruning. However, the end result is the same, since alpha-beta pruning just skips the useless paths in the game tree. Therefore, it makes sense to use alpha-beta pruning when possible.

5.1.3 Default vs Tabular Q-learning. Tabular Q-learning does not always win with the default opponent, unlike minimax. However, tabular Q-learning is much quicker and uses less memory. Tabular Q-learning is much faster, so I was able to run it for 1000 rounds. It slightly beats the default opponent, which makes it feasible. It is better than randomly choosing moves, and it takes less memory and time than the default approach!

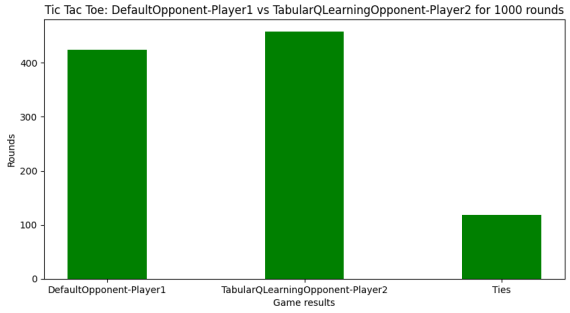

Figure 3. Default vs Tabular Q-learning for 1000 rounds.

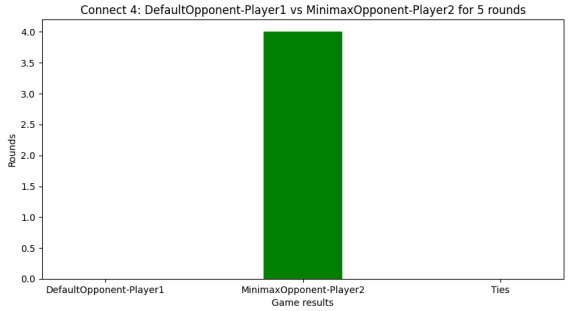

Figure 4. Default vs Minimax (without alpha-beta pruning) for 5 rounds.

	Wins	Average time taken	Average max memory
Default	424	0.0242s	10938.31 bytes
TQL	457	0.0008s	6526.643 bytes
Ties	119	-	-

Table 3. Default vs Tabular Q-learning for 1000 rounds.

	Wins	Average time taken	Average max memory
Default	0	0.1274s	27438.40 bytes
M (WABP)	5	157.1316s	85259.40 bytes
Ties	0	-	-

Table 4. Default vs Minimax (without alpha-beta pruning) for 5 rounds.

The experiment above shows that tabular Q-learning is slightly better than the default opponent, but does tie or lose with the default opponent a minority of times. This means that it is not a massively accurate opponent, but it is a computationally cheaper way of choosing moves that are better than making random moves.

5.1.4 Conclusion. For Tic Tac Toe, it is evident that using alpha-beta pruning is best to save time, since they both yield the same results. It is also evident that minimax is very slow in comparison to tabular Q-learning, however it always wins against the default opponent. Therefore, minimax with alpha-beta pruning is best for use cases when time and memory is not an issue, but the winning rate is important. On the other hand, tabular Q-learning is best for use cases when time and memory are constrained, but the winning rate only needs to be better than choosing randomly.

5.2 How do your algorithms compare to each other when playing against default opponent in Connect 4

5.2.1 Default vs Minimax (no alpha-beta pruning). Minimax with no alpha-beta pruning always wins compared to the default opponent. It takes a lot more memory and time, but it seems to always win. Because it takes a lot of time, I experimented across 5 rounds which is shown below:

It is evident that the default opponent will never win against the minimax opponent, but it takes considerably less time and memory when it is trying to defend itself.

5.2.2 Default vs Minimax (alpha-beta pruning). Since minimax with alpha-beta pruning is a lot faster, I experimented with 50 rounds against the default opponent. When I increased the number of rounds (because it was computationally feasible now), the default opponent was able to win once.

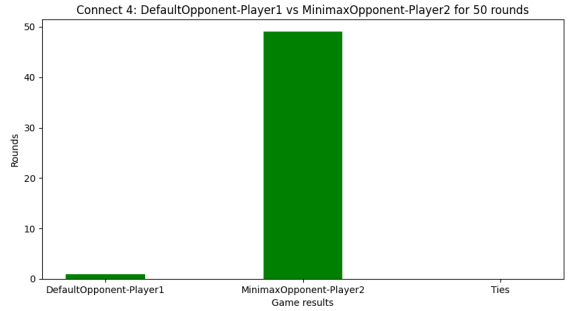

Figure 5. Default vs Minimax (alpha-beta pruning) for 50 rounds.

	Wins	Average time taken	Average max memory
Default	1	0.1258s	28984.80 bytes
M (ABP)	49	12.7208s	83102.54 bytes
Ties	0	-	-

Table 5. Default vs Minimax (alpha-beta pruning) for 50 rounds.

However, It is evident that minimax with alpha-beta pruning wins almost all the time. It is valuable to note that the alpha-beta pruning made minimax use considerably less time and memory compared to without alpha-beta pruning.

5.2.3 Default vs Tabular Q-learning. Tabular Q-learning is much quicker in comparison to the rest of the algorithms, so I tried 1000 rounds with the default opponent. It is evident that the tabular Q-learning opponent actually does worse than the default opponent.

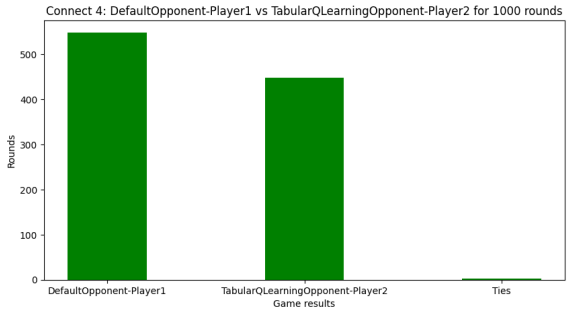

Figure 6. Default vs Tabular Q-Learning for 1000 rounds.

	Wins	Average time taken	Average max memory
Default	548	0.2369s	42695.86 bytes
TQL	449	0.0068s	30747.40 bytes
Ties	3	-	-

Table 6. Default vs Tabular Q-Learning for 1000 rounds.

This is an interesting observation, because we know that tabular q-learning was feasible for Tic Tac Toe but here it is evident that it is not feasible for Connect 4. This makes sense, since Tic Tac Toe has a smaller number of states to visit compared to Connect 4 (3x3=9 states vs 6x7=42 states).

5.2.4 Conclusion. From my experiments with Connect 4, it seems that Minimax is a feasible approach since it almost always wins against the default opponent, however it takes much more time and memory. Tabular Q-learning is not feasible for Connect 4 since it has more states to visit, and will require a lot more rounds when training to have a better accuracy. In terms of Minimax, using alpha-beta pruning is still the best option to reduce the memory and time required to win.

5.3 How do your algorithms compare to each other when playing against default opponent overall

Overall, almost all of the algorithms win against the default opponent on both Tic Tac Toe and Connect 4. The only algorithm that did not defeat the default was tabular Q-learning when playing Connect 4. It seems that the higher number of states and longer length of the game is difficult to train for using Q-learning. It might be feasible with more rounds when training, but using minimax with alpha-beta pruning is much more accurate (which is worth it even if you need to spend more memory and time to get that!).

5.4 How do your algorithms compare to each other when playing against each other in Tic Tac Toe

5.4.1 Minimax with alpha-beta pruning vs without. In terms of winning, minimax with alpha-beta pruning was the same accuracy of as minimax without alpha-beta pruning. This makes sense, since they both always have the same result, but alpha-beta pruning ignores useless paths in the game tree. In this experiment, they seem to always have a tie.

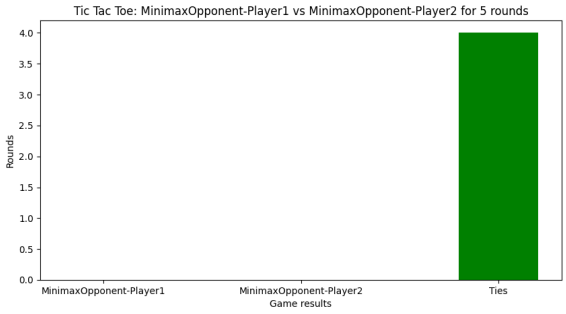

Figure 7. Minimax (alpha-beta pruning) vs Minimax (without alpha-beta pruning) for 5 rounds.

	Wins	Average time taken	Average max memory
M (ABP)	0	10.6443s	25565.60 bytes
M (WABP)	0	313.3520s	34204.00 bytes
Ties	5	-	-

Table 7. Minimax (alpha-beta pruning) vs Minimax (without alpha-beta pruning) for 5 rounds.

Once again, minimax with alpha-beta pruning is a lot quicker and uses much less memory when playing Tic Tac Toe. They are equal in terms of accuracy, but in terms of time taken and memory usage, it is clear that alpha-beta pruning is more efficient.

5.4.2 Minimax (alpha-beta pruning) vs Tabular Q-learning. For Tic Tac Toe, minimax with alpha-beta pruning is a lot slower than tabular Q-learning. Minimax also takes a lot more time compared to tabular Q-learning. However, Minimax still wins or ties against tabular Q-learning when it was ran for 50 rounds:

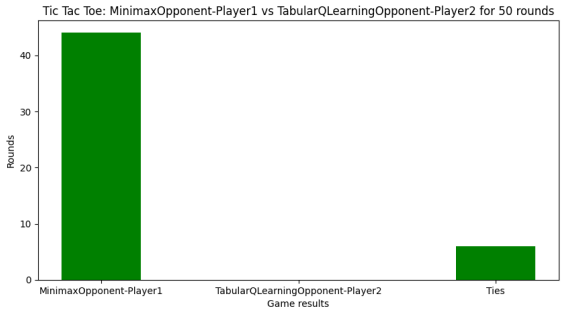

Figure 8. Minimax (alpha-beta pruning) vs Tabular Q-learning for 50 rounds.

	Wins	Average time taken	Average max memory
TQL	0	0.0008s	5370.10 bytes
M (ABP)	44	14.2432s	37208.72 bytes
Ties	6	-	-

Table 8. Minimax (alpha-beta pruning) vs Tabular Q-learning for 50 rounds.

Tabular Q-learning did not win against minimax with alpha-beta pruning at all. It was only able to tie a minority of the time, but it required a fraction of the time and memory usage of minimax.

5.4.3 Conclusion. It is evident that minimax with and without alpha-beta pruning have the same results in terms of win rate, but alpha-beta pruning requires only around 3% of the original time and around 73% of the original memory. Q-learning is not as good at Tic Tac Toe as minimax with alpha-beta pruning, which concludes that minimax with alpha-beta pruning is the best option for Tic Tac Toe. It is the most efficient, but requires the most computational resources. Tabular Q-learning requires less memory, less time, and is still more efficient than the default opponent as mentioned previously.

5.5 How do your algorithms compare to each other when playing against each other in Connect 4

5.5.1 Minimax with alpha-beta pruning vs without. Similar to Tic Tac Toe, minimax with alpha-beta pruning is still the same accuracy as minimax without alpha-beta pruning when playing Connect 4. Once again, minimax has lower time and memory usage with alpha-beta pruning compared to without.

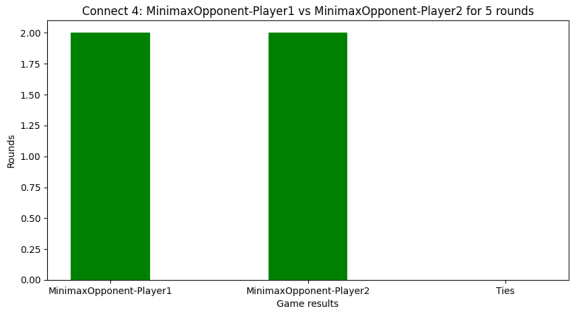

Figure 9. Minimax (alpha-beta pruning) vs Minimax (without alpha-beta pruning) for 5 rounds.

	Wins	Average time taken	Average max memory
M (ABP)	2	17.3038s	137056.60 bytes
M (WABP)	2	171.6612s	150768.20 bytes
Ties	0	-	-

Table 9. Minimax (alpha-beta pruning) vs Minimax (without alpha-beta pruning) for 5 rounds.

This time, both minimax opponents had the same number of wins. It shows 2 and 2 for 5 rounds because the requested amount of rounds is divided by two, and converted to an integer. This number is used to run the two opponents against each other as player one and player two, but while swapping who goes first in each half of the total number of rounds.

When this experiment was conducted with Tic Tac Toe, both minimax opponents had no wins but simply had a tie each time. This proves that the added complexity from Connect 4 (compared to Tic Tac Toe in terms of length of the game and the states to visit) makes the minimax opponents more susceptible to losing. However, since they both result in the same output, the number of wins from each minimax opponent is the same.

5.5.2 Minimax (alpha-beta pruning) vs Tabular Q-learning. When running minimax with alpha-beta pruning against a tabular Q-learning opponent, the minimax wins almost every time. This is shown in an experiment with 50 rounds below:

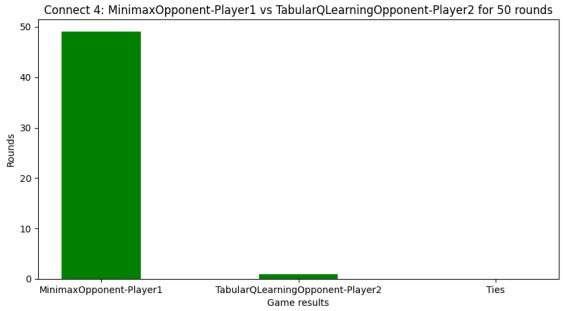

Figure 10. Minimax (alpha-beta pruning) vs Tabular Q-learning for 50 rounds.

	Wins	Average time taken	Average max memory
TQL	1	0.0036s	18247.92 bytes
M (ABP)	49	9.6850s	73299.02 bytes
Ties	0	-	-

Table 10. Minimax (alpha-beta pruning) vs Tabular Q-learning for 50 rounds.

It is evident that tabular q-learning is not able to handle the complexity of connect 4 in a feasible way. When running more than 50 rounds, it crashes my computer, making me unable to gather data, which shows that the extra states to visit and extra length in the game make it more difficult to make the tabular Q-learning opponent win consistently.

5.6 How do your algorithms compare to each other when playing against each other overall

The tabular q-learning struggles to beat minimax overall. Tabular q-learning is able to beat the default opponent in Tic Tac Toe, but is unable to beat any other opponent. This shows that tabular q-learning is a good option for cases where time and memory are constrained heavily but the outcome just has to be better than random moves for a problem with a small number of states to visit (such as Tic Tac Toe). When accuracy is a requirement and there are a lot of states to visit, minimax with alpha-beta pruning is the best option. It beats every opponent possible, with the caveat that it requires a lot of memory since it has to traverse the game tree to find an optimal move. Avoiding alpha-beta pruning is not a good idea, since the end result from the opponent is the same with alpha-beta pruning but it requires a fraction of the initial memory and time.

6 Conclusion

From my experiments, I can conclude that tabular Q-learning performs well for games with a low number of states such as Tic Tac Tow. It requires less memory than the default opponent, and was able to beat it in terms of the winning rate. However, tabular Q-learning struggles with games that have a high number of game states such as Connect 4. This is because the number of rounds required when training to encounter all possible states becomes very high. Therefore, the time complexity of the algorithm becomes a lot worse.

The minimax algorithm performs well across games with a low number of states and a high number of states. It has a better winning rate compared to tabular Q-learning. However, minimax requires a lot more memory and time since it has to iterate across many paths in the game tree. When using minimax, alpha-beta pruning reduces the high memory and time requirement on this regard, while having the same winning rate.

To conclude, tabular Q-learning is good for Tic Tac Toe since there is a low number of states (making it able to beat the default opponent). However, minimax is better for a more general approach to solving games with a high or low amount of states. When using minimax, alpha-beta pruning is the most efficient method of using minimax since it reduces the time and memory required by many magnitudes. Minimax is good for competing against both the tabular Q-learning opponent as well as the default opponent since it wins almost every time.

7 Video demo

My video demo for this project can be viewed on Youtube at https://youtu.be/q-rjDTo_1tw or on Google Drive at https://drive.google.com/file/d/1pSzgFOpyk4C6qbEyF6kZuhPRh_dhn0a6/view?usp=sharing. The project contains a file called `readme.txt` that contains instructions on how the code can be run. The project needs to be run in a regular terminal for the input/output to be working correctly. All necessary packages are installed in the virtual environment ("venv" folder) for Python 3.9.

8 References

[1]: python-tictactoe (Python library for board)
[2]: Alpha-beta pruning with depth limit
[3]: Inspiration for code with reinforcement learning
[4]: Using agents in reinforcement learning

9 Appendix

9.1 Minimax implementation

```
import math


class MinimaxOpponent:
    def __init__(self, alpha, beta):
        self.alpha = alpha
```

```
        self.beta = beta

    @staticmethod
    def minimax_for_tic_tac_toe(self,
    tictactoe_board, alpha, beta, maximize):
        result = tictactoe_board.result()
        if result == 1:
            return 1, None
        if result == 2:
            return -1, None
        elif result == 0:
            return 0, None
        if maximize:
            best_move = None
            maximum_evaluation_value = -100
            for possible_move in
            tictactoe_board.possible_moves():
                temporary_tictactoe_board =
                tictactoe_board.copy()
                temporary_tictactoe_board.push(
                possible_move)
                evaluation =
                self.minimax_for_tic_tac_toe(
                self, temporary_tictactoe_board,
                alpha, beta, False)[0]
                if evaluation >
                maximum_evaluation_value:
                    best_move = possible_move
                    maximum_evaluation_value =
                    evaluation
                if alpha is not None:
                    alpha = max(alpha,
                    maximum_evaluation_value)
                    if alpha >= beta:
                        break
            return maximum_evaluation_value,
            best_move
        elif not maximize:
            best_move = None
            minimum_evaluation_value = 100
            for possible_move in
            tictactoe_board.possible_moves():
                temporary_tictactoe_board =
                tictactoe_board.copy()
                temporary_tictactoe_board.push(
                possible_move)
                evaluation =
                self.minimax_for_tic_tac_toe(
                self, temporary_tictactoe_board,
                alpha, beta, True)[0]
                if evaluation <
                minimum_evaluation_value:
                    best_move = possible_move
                    minimum_evaluation_value =
                    evaluation
                if beta is not None:
                    beta = min(beta,
                    minimum_evaluation_value)
                    if alpha >= beta:
                        break
            return minimum_evaluation_value,
            best_move

    @staticmethod
    def make_next_tic_tac_toe_move(**kwargs):
        self = kwargs.get('self', None)
        board = kwargs.get('board', None)
        _, move =
        self.minimax_for_tic_tac_toe(self,
        board, self.alpha, self.beta,
        board.turn == 1)
        return move


    @staticmethod
    def minimax_for_connect_four(self,
    connect_four_board, depth_limit, alpha,
    beta, maximize):
        result = connect_four_board.result()
        if (connect_four_board.has_won(1) or
        connect_four_board.has_won(2)) or
        depth_limit == 0:
            if result == 1:
                return 1, None
            if result == 2:
                return -1, None
            elif result == 0:
                return 0, None
            return 0, None
        if maximize:
            best_move = None
            maximum_evaluation_value = -math.inf
            for possible_move in
            connect_four_board.possible_moves():
                temporary_connect_four_board =
                connect_four_board.copy()
                temporary_connect_four_board
                .x_in_a_row = 4
                temporary_connect_four_board
                .push(possible_move)
                reward =
                self.minimax_for_connect_four(
```

```
                self,
                temporary_connect_four_board,
                depth_limit - 1, alpha, beta,
                False)[0]
                if reward >
                maximum_evaluation_value:
                    best_move = possible_move
                    maximum_evaluation_value =
                    reward
                if alpha is not None:
                    alpha = max(alpha,
                    maximum_evaluation_value)
                    if alpha >= beta:
                        break
            return maximum_evaluation_value,
            best_move
        elif not maximize:
            best_move = None
            minimum_evaluation_value = math.inf
            for possible_move in
            connect_four_board.possible_moves():
                temporary_connect_four_board =
                connect_four_board.copy()
                temporary_connect_four_board
                .x_in_a_row = 4
                temporary_connect_four_board
                .push(possible_move)
                reward =
                self.minimax_for_connect_four(
                self,
                temporary_connect_four_board,
                depth_limit - 1, alpha, beta,
                True)[0]
                if reward <
                minimum_evaluation_value:
                    best_move = possible_move
                    minimum_evaluation_value =
                    reward
                if beta is not None:
                    beta = min(beta,
                    minimum_evaluation_value)
                    if alpha >= beta:
                        break
            return minimum_evaluation_value,
            best_move

    @staticmethod
    def make_next_connect_four_move(**kwargs):
        self = kwargs.get('self', None)
        board = kwargs.get('board', None)
        _, move =
        self.minimax_for_connect_four(self,
        board, 5, self.alpha, self.beta,
        board.turn == 1)
        return move
```

9.2 Tabular Q-Learning implementation

```
import pickle

import numpy as np


class TabularQLearningOpponent:
    def __init__(self, alpha, epsilon, gamma):
        self.alpha = alpha
        self.epsilon = epsilon
        self.gamma = gamma
        self.Q_table = {}
        self.states = []

    def load_policy(self):
        loaded = False
        while not loaded:
            pickle_file_name = input('What is
            the name of the pickle file from
            training your q-agent: ')
            try:
                pickle_file =
                open(pickle_file_name, 'rb')
                self.Q_table =
                pickle.load(pickle_file)
                pickle_file.close()
                loaded = True
            except FileNotFoundError:
                print(f'Pickle file named
                {pickle_file_name} not found.
                Try again')

    def update_states_as_empty(self):
        self.states = []

    def save_policy(self, pickle_file_name):
        pickle_file = open(pickle_file_name,
        'wb')
        pickle.dump(self.Q_table, pickle_file)
        pickle_file.close()

    def set_state_of_agent(self, state):
        self.states.append(state)

    def q_learning_update(self, reward):
        for state in reversed(self.states):
```

```
            if self.Q_table.get(state) is None:
                self.Q_table[state] = 0
            self.Q_table[state] += self.alpha *
            (self.gamma * reward -
            self.Q_table[state])
            reward = self.Q_table[state]

    def update_epsilon(self, episode):
        self.epsilon = max(0.3, self.epsilon -
        0.000001 * episode)

    @staticmethod
    def board(tictactoe_or_connect_four_board):
        return
        str(tictactoe_or_connect_four_board
        .board.flatten())

    @staticmethod
    def make_next_tic_tac_toe_move(**kwargs):
        move = None
        self = kwargs.get('self', None)
        possible_board_moves =
        kwargs.get('positions', None)
        board = kwargs.get('board', None)
        if np.random.uniform(0, 1) <=
        self.epsilon:
            move =
            possible_board_moves[np.random
            .choice(len(possible_board_moves))]
        else:
            max_q_value = float('-inf')
            for possible_move in
            possible_board_moves:
                upcoming_tictoctoe_board =
                board.copy()
                upcoming_tictoctoe_board
                .push(tuple(possible_move))
                upcoming_tictoctoe_board_state
                = self.board(
                upcoming_tictoctoe_board)
                if self.Q_table.get(
                upcoming_tictoctoe_board_state)
                is None:
                    q_value = 0
                else:
                    q_value = self.Q_table.get(
                upcoming_tictoctoe_board_state)
                if q_value >= max_q_value:
                    max_q_value = q_value
                    move = possible_move
        return move

    @staticmethod
    def make_next_connect_four_move(**kwargs):
        best_move = None
        self = kwargs.get('self', None)
        possible_board_moves =
        kwargs.get('positions', None)
        board = kwargs.get('board', None)
        if np.random.uniform(0, 1) <=
        self.epsilon:
            best_move = np.random.choice(
            board.possible_moves())
        else:
            maximum_q_value = float('-inf')
            for possible_move in
            possible_board_moves:
                upcoming_connect_four_board =
                board.copy()
                upcoming_connect_four_board
                .x_in_a_row = 4
                upcoming_connect_four_board
                .push(possible_move)
              upcoming_connect_four_board_state
                = self.board(
                upcoming_connect_four_board)
                if self.Q_table.get(
             upcoming_connect_four_board_state)
                is None:
                    q_value = 0
                else:
                    q_value = self.Q_table.get(
                    upcoming_connect_four_board
                    _state)
                if q_value >= maximum_q_value:
                    maximum_q_value = q_value
                    best_move = possible_move
        return best_move
```

9.3 Tic Tac Toe implementation

```
import time
import tracemalloc
from IPython.core.display_functions import
clear_output
from tictactoe import Board
from tqdm import tqdm


class TicTacToe:
    def __init__(self, first_player,
    second_player):
        self.first_player = first_player
        self.second_player = second_player
        self.board = Board(dimensions=(3, 3))
```

```
        self.board_as_string = None

    def tictactoe_board(self):
        self.board_as_string =
        str(self.board.board.flatten())
        return self.board_as_string

    def reset_board(self):
        self.board = Board()
        self.board_as_string = None

    def set_learning_reward(self):
        result = self.board.result()
        if result == 1:
         self.first_player.q_learning_update(1)
            self.second_player.
            q_learning_update(-1)
        if result == 2:
            self.first_player.
            q_learning_update(-1)
            self.second_player.
            q_learning_update(1)
        else:
         self.first_player.q_learning_update(0)
            self.second_player.
            q_learning_update(0)

    def play(self):
        winner = None
        first_player_time_taken = 0
        first_player_max_memory_used = 0
        second_player_time_taken = 0
        second_player_max_memory_used = 0
        print(self.board)
        while True:
            positions =
            self.board.possible_moves()
            print(f"{self.first_player
            .__class__.__name__}'s turn
            (marking as X)")
            start_time = time.time()
            tracemalloc.start()
            try:
                first_player_action =
                self.first_player
                .make_next_tic_tac_toe_move(
                self=self.first_player,
                positions=positions,
                board=self.board)
               self.board.push(first_player_action)
            except (ValueError, IndexError):
                print("That place is taken
                already or you are out of
                bounds. Try again.")
                continue
            end_time = time.time()
            first_player_max_memory_used =
            first_player_max_memory_used +
            (tracemalloc.get_traced_memory()[1])
            tracemalloc.stop()
            first_player_time_taken =
            first_player_time_taken +
            (end_time - start_time)
            clear_output()
            print(self.board)
            winner = self.board.result()
            if winner is not None:
                if winner == 1:
                    print(self.first_player
                    .__class__.__name__, "has
                    won.")
                else:
                    print("tie!")
                self.reset_board()
                break
            else:
                positions =
                self.board.possible_moves()
                print(f"{self.second_player
                .__class__.__name__}'s turn
                (marking as 0)")
                start_time = time.time()
                tracemalloc.start()
                try:
                    second_player_action =
                    self.second_player
                   .make_next_tic_tac_toe_move(
                    self=self.second_player,
                    positions=positions,
                    board=self.board)
                    self.board.push(
                    second_player_action)
               except (ValueError, IndexError):
                    print("That place is taken
                    already or you are out of
                    bounds. Try again.")
                    continue
                end_time = time.time()
                second_player_max_memory_used =
                second_player_max_memory_used
                +
            (tracemalloc.get_traced_memory()[1])
                tracemalloc.stop()
```

```
                second_player_time_taken =
                second_player_time_taken +
                (end_time - start_time)
                clear_output()
                print(self.board)
                winner = self.board.result()
                if winner is not None:
                    if winner == 2:
                       print(self.second_player
                        .__class__.__name__,
                        "has won.")
                    else:
                        print("It is a tie.")
                    self.reset_board()
                    break
        return winner, first_player_time_taken,
        second_player_time_taken,
        first_player_max_memory_used,
        second_player_max_memory_used

    def train(self, rounds):
        for i in tqdm(range(rounds)):
            self.first_player.update_epsilon(i)
           self.second_player.update_epsilon(i)
            while True:
                positions =
                self.board.possible_moves()
                first_player_action =
                self.first_player
                .make_next_tic_tac_toe_move(
                self=self.first_player,
                positions=positions,
                board=self.board)
           self.board.push(first_player_action)
                board_as_string =
                self.tictactoe_board()
                self.first_player
           .set_state_of_agent(board_as_string)
                winner = self.board.result()
                if winner is not None:
                    self.set_learning_reward()
                    self.first_player
                    .update_states_as_empty()
                    self.second_player
                    .update_states_as_empty()
                    self.reset_board()
                    break
                else:
                    positions =
                    self.board.possible_moves()
                    second_player_action =
                    self.second_player
                   .make_next_tic_tac_toe_move(
                    self=self.second_player,
                    positions=positions,
                    board=self.board)
                    self.board.push(
                    second_player_action)
                    board_as_string =
                    self.tictactoe_board()
                    self.second_player
                    .set_state_of_agent(
                    board_as_string)
                   winner = self.board.result()
                    if winner is not None:
                     self.set_learning_reward()
                        self.first_player
                      .update_states_as_empty()
                        self.second_player
                      .update_states_as_empty()
                        self.reset_board()
                        break
```

9.4 Connect 4 implementation

```
import time
import tracemalloc
from IPython.core.display_functions import
clear_output
from tictactoe import Board
from tqdm import tqdm


class ConnectFour:
    def __init__(self, first_player,
    second_player):
        self.first_player = first_player
        self.second_player = second_player
        self.board =
        ConnectFourBoard(dimensions=(6, 7))
        self.board_as_string = None
        self.board.x_in_a_row = 4

    def connect_four_board(self):
        self.board_as_string =
        str(self.board.board.flatten())
        return self.board_as_string

    def set_learning_reward(self):
        result = self.board.result()
        if result == 1:
         self.first_player.q_learning_update(1)
            self.first_player
            .q_learning_update(-1)
        if result == 2:
```

```
            self.first_player.q_learning_update(1)
            self.first_player
            .q_learning_update(-1)
        else:
            self.first_player.q_learning_update(0)
            self.first_player.q_learning_update(0)

    def reset_board(self):
        self.board =
        ConnectFourBoard(dimensions=(6, 7))
        self.board.x_in_a_row = 4
        self.board_as_string = None

    def play(self):
        winner = None
        first_player_time_taken = 0
        first_player_max_memory_used = 0
        second_player_time_taken = 0
        second_player_max_memory_used = 0
        print(self.board)
        while True:
            positions =
            self.board.possible_moves()
            print(f"{self.first_player
            .__class__.__name__}'s turn
            (marking as X)")
            start_time = time.time()
            tracemalloc.start()
            try:
                move = self.first_player
                .make_next_connect_four_move(
                self=self.first_player,
                positions=positions,
                board=self.board)
                self.board.push(move)
            except ValueError or IndexError:
                print("That column is filled.
                Try another column.")
                continue
            end_time = time.time()
            first_player_max_memory_used =
            first_player_max_memory_used +
            (tracemalloc.get_traced_memory()[1])
            tracemalloc.stop()
            first_player_time_taken =
            first_player_time_taken +
            (end_time - start_time)
            clear_output()
            print(self.board)
            winner = self.board.result()
            if winner is not None:
                if winner == 1:
                    print(f'{self.first_player
                    .__class__.__name__} has
                    won.')
                else:
                    print("It is a tie.")
                self.reset_board()
                break
            else:
                positions =
                self.board.possible_moves()
                print(f"{self.second_player
                .__class__.__name__}'s turn
                (marking as 0)")
                start_time = time.time()
                tracemalloc.start()
                try:
                    move = self.second_player
                    .make_next_connect_four_move(
                    self=self.second_player,
                    positions=positions,
                    board=self.board)
                    self.board.push(move)
                except ValueError or IndexError:
                    print("That column is filled.
                    Try another column.")
                    continue
                end_time = time.time()
                second_player_max_memory_used =
                second_player_max_memory_used +
                (
                    tracemalloc
                    .get_traced_memory()[1])
                tracemalloc.stop()
                second_player_time_taken =
                second_player_time_taken +
                (end_time - start_time)
                clear_output()
                print(self.board)
                winner = self.board.result()
                if winner is not None:
                    if winner == 2:
                    print(f'{self.second_player
                        .__class__.__name__} has
                        won.')
                    else:
                        print("It is a tie.")
                    self.reset_board()
                    break
        return winner, first_player_time_taken,
        second_player_time_taken,
        first_player_max_memory_used,
        second_player_max_memory_used

    def train(self, iterations):
```

```
        for i in tqdm(range(iterations)):
            self.first_player.update_epsilon(i)
            self.second_player.update_epsilon(i)
            while True:
                positions =
                self.board.possible_moves()
                first_player_action = self
                .first_player
                .make_next_connect_four_move(
                self=self.first_player,
                positions=positions,
                board=self.board)
            self.board.push(first_player_action)
                board_as_string =
                self.connect_four_board()
                self.first_player
            .set_state_of_agent(board_as_string)

                winner = self.board.result()
                if winner is not None:
                    self.set_learning_reward()
                    self.first_player
                    .update_states_as_empty()
                    self.second_player
                    .update_states_as_empty()
                    self.reset_board()
                    break

                else:
                    positions =
                    self.board.possible_moves()
                    second_player_action =
                    self.second_player
                  .make_next_connect_four_move(
                    self=self.second_player,
                    positions=positions,
                    board=self.board)
                    self.board.push(
                    second_player_action)
                    board_as_string =
                    self.connect_four_board()
                    self.second_player
                    .set_state_of_agent(
                    board_as_string)

                  winner = self.board.result()
                   if winner is not None:
                    self.set_learning_reward()
                       self.first_player
                     .update_states_as_empty()
                       self.second_player
                     .update_states_as_empty()
                       self.reset_board()
                       break


class ConnectFourBoard(Board):
    def __init__(self, dimensions):
        self.state_dict = dict()
        super().__init__(dimensions,
        x_in_a_row=4)

    def push(self, column):
        for possible_move in
        super().possible_moves():
            self.state_dict[possible_move[0]] =
            []
        for possible_move in
        super().possible_moves():
            self.state_dict[possible_move[0]]
            .append(possible_move[1])
        super().push((column,
        max(self.state_dict[column])))

    def copy(self):
        board =
        ConnectFourBoard(self.dimensions)
        board.state_dict = self.state_dict
        board.board = self.board.copy()
        board.turn = self.turn
        return board

    def possible_moves(self):
        possible_moves_list = []
        for possible_move in
        super().possible_moves():
            if possible_move[0] not in
            possible_moves_list:
                possible_moves_list.append(
                possible_move[0])
        return possible_moves_list
```