Artificial Intelligence (CS7IS2)

Assignment 2

Catalin Gheorghiu (22305257)

I understand that this is an individual assessment and that collaboration is not permitted. I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`. I understand that by returning this declaration with my work, I am agreeing with the above statement.

---

1 Game Implementations

No external sources were used for either Tic-Tac-Toe (TTT) or Connect 4 (C4) – I wrote both games myself and forsook a proper GUI in favour of ASCII console printing. This approach not only has the advantage of versatility but greatly shortens the following discussion on the underlying methods of the games.

The underlying data structure supporting the board of both games is a list of lists (one for each row) dubbed `squares`, with spaces marking an empty cell, `X` marking moves of the first player, and `O` marking moves of the second player. The methods interacting with the board – and thus driving the games – are as follows:

- `draw_board(squares)`: prints the current state of the game in the console, using '|' to separate rows; its output is shown in figure 1, showing an arbitrary state of a TTT and C4 game in the left and right panel, respectively.
- `reset_board(squares)`: sets all cells back to spaces, preparing the board for a new game
- `check_win(squares)`: given a board and a player, it checks if the player has enough consecutive tokens on a row, column or diagonal to win the game, in which case the winner is returned. A tie is returned if there are no more available cells without the previous condition being activated, and nothing is returned otherwise.
- `play(p_1, p_2, squares)`: given a board and two players, this core method uses the specific moving methods of various algorithms to play a game, beginning with `X` and alternating between players. The final state of the game is returned once it is determined. Players are assigned to this method using specific string codes, to be discussed when the respective algorithms they represent are introduced.
- `human_move`: the most basic type of player, it is fed to `play` by assigning '`H`' to either its first or second argument, thus allowing the user to play as either '`X`' or '`O`'. This player's method is mostly used for hands-on algorithm testing, especially if one seeks to observe the response to a specific strategy.

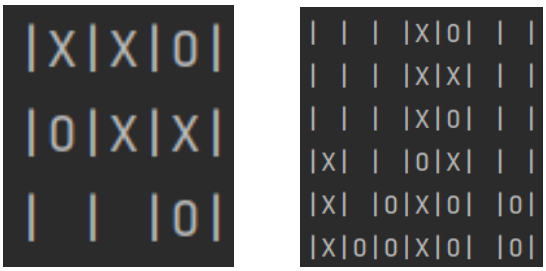

Figure 1: Example boards of TTT (left) and C4 (right).

2 Baseline Algorithm

Contained in the method `baseline_move(squares, turn)` and invoked as a player using the code 'B', this simple algorithm scans the board for valid moves and prioritises, in order, winning on the current move, blocking an opponent's winning move on the current move, and making a random move otherwise. Note that, due to the game's relative complexity, the baseline algorithm of C4 uses several helper methods to determine if a specific player (itself or the opponent) `turn` has the opportunity to win over rows, columns, or diagonals:

- `row_check_win(squares, turn)`
- `col_check_win(squares, turn)`
- `diag_check_win(squares, turn)`

For TTT, the above methods were not necessary, as the pre-terminal states are easier to identify and there is a smaller state space overall. The baseline algorithm finds a move very quickly in both games, and is used to train and/or evaluate the upcoming, more advanced algorithms.

3 Minimax + Alpha-Beta Pruning

Even ignoring practically impossible positions like a board with 9 X moves or the symmetry of certain positions, TTT would still only have $3^9 < 20000$ possible states, allowing for the terminal-test implementation of minimax to be used comfortably. This means that the recursive alternating calls between the minimising and maximising player are only stopped when a leaf of the game tree is reached. Methods `min_value` and `max_value` execute the moves that minimise the opponent's score and maximise the player's score respectively, and the terminal test is performed at the start of both methods. The maximising player receives a positive reward of +10 for winning and a negative reward of -10 for losing, while the minimising player receives a symmetric -10 reward for winning and +10 for losing. Both players receive rewards of 0 when the game is tied, ensuring that the players are trying to win and only push for ties if winning is not possible.

C4, however, has over 4.5 trillion possible states, making an exhaustive search infeasible. Applying the same terminal-test implementation of the minimax algorithm from TTT fails to make a single move in 30 minutes of running, only managing to explore a little over 67 million states in this time window for its first move. The algorithm would therefore need about 1400 days of continous running to make its first move. The `max_value` and `min_value` methods of C4 thus implement a cutoff-test based on a given game depth. Depth starts at a user-given value and decreases with every call of either method. When a depth of 0 is reached, the methods return an evaluation of the position instead of the rewards corresponding to the end of the game. I use a simple heuristic for evaluating positions, contained in the `evaluate_state` method, based on the configuration of every group of four consecutive slots on the rows, columns, or diagonals of the board:

- 5 points awarded for each group containing 2 player tokens and 2 spaces
- 10 points awarded for each group containing 3 player tokens and 1 space
- 4 points deducted for each group containing 2 opponent tokens and 2 spaces
- 8 points deducted for each group containing 3 opponent tokens and 1 space

The weights are chosen to get the player to prioritise its own victory over blocking the opponent, and to incentivise fights over empty space. The player will want to fill empty space with its own tokens, while denying the empty space around the opponent's tokens. Victory and defeat should have very high respective positive and negative rewards in order for the algorithm to prefer winning and avoid losing over playing by the heuristic. Using this reward scheme and a move of 6, the minimax player can now be pitted against other algorithms.

Alpha-Beta pruning makes the algorithm more efficient by avoiding the pursuit of sub-trees whose best outcome is inferior to that of a different sub-tree. The minimax code only needs a few lines of code worth of modifications to implement alpha-beta pruning, shown in figure 2 for both the maximising and minimising moves. The minimax player is called to play using string code '`M`' on either the first or second position of the `play` method, depending on whether the minimax player should be `X` or `O`. When making a move, this player looks at the current state of the board, solves either the entire game tree from that position for TTT or just part of the game tree (up to the given depth) for C4, and makes the best move based on the endgame rewards or evaluation heuristic, depending how far away the end of the game is.

```
# Pruning and updating alpha
if value >= beta:
    return [value, r, c]
alpha = max(alpha, value)
```

```
# Pruning and updating beta
if value <= alpha:
    return [value, r, c]
beta = min(value, beta)
```

Figure 2: Implementations of alpha-beta pruning for maximising moves (left) and minimising moves (right).

4 Tabular Reinforcement Q-Learning

The eponymous Q values are stored in a dictionary, where the keys are tuples of the form `(squares, turn, move)`. Matching this structure to the regular Q-value notation $Q(s, a)$, the board configuration `squares` gives the state s and the row and column of the next `move` in that state gives the action a. The turn is simply meant to help distinguish the keys of the two dictionaries used in the algorithm, one for the moves of an '`X`' player `qvalues_X` and one for the moves of an '`O`' player `qvalues_O`. In order to store board states, the method `hash_qargs` transforms the board into a tuple of row tuples, and in order to facilitate the retrieval of Q-values, the method `get_Q` searches for the given key in the given dictionary and initiates the Q-value of that key as 0 if it the state-action was not yet observed, or returns the Q-value otherwise. Note that the Q-values of terminal states do not appear in this dictionary because they are never updated; instead, they are assigned the corresponding reward of the game: 1 for a player win, 0.5 for a tie, and -1 for a loss in the case of TTT, and (100, 50, -100) in the case of C4. The higher magnitude of C4 rewards is meant to compensate for the higher average number of moves per game, leading to the final reward propagating through more states and getting decayed more as a result.

The method `Q_Train(squares, turn, opponent, alpha, gamma, epsilon, iterations)` trains a player specified by `turn` against the specified opponent (with current options `'baseline'` and `'minimax'`) for a number of episodes – or complete games played – specified by `iterations`. For each of the two possible players, I train the TTT Q-learning algorithm for 50 epochs of 500 episodes each, split between 400 games against the `baseline` player and 100 games against the `minimax` player. As there are far more positions in C4, I train this Q-learning algorithm for 100 epochs of 5000 episodes each, all against the `baseline` player as a better `minimax` player – such as the one with depth 6 from before – would take too long to move for training to be feasible.

I use the `epsilon` parameter to implement the ε-greedy variation of the Q-learning algorithm. By starting with a high value (0.8 for TTT and 1 for C4 in my case) and decaying it exponentially with a rate of 0.95 for TTT and 0.8 for C4 per epoch, a high proportion of exploration can be guaranteed for roughly the first half of training epochs, with the emphasis shifting to action based on Q-values in the second half after some guiding principles for the game were established through mostly random movements. I opted for a higher initial ε for C4 to cover as many states as possible early on, and for a higher decay rate to compensate for doing 10 times more episodes per epoch compared to TTT.

Keeping `gamma` fixed at 0.9 to prioritise the long-term gains over the short-term, the only parameter left to discuss is the learning rate `alpha`. This value also starts high, at 0.8 for TTT and 0.9 for C4, to then decay at a rate of 0.95 for TTT and 0.9 for C4 per epoch. This ensures that historic rewards are not very impactful in the beginning – when a high proportion of moves is random – but highly relevant towards the latter epochs, when a move yielding new rewards should meaningfully change the player's strategy. Decaying both the learning rate α and random move threshold ε should improve the algorithm's accuracy, as it has been empirically observed in the past for most RL applications.

After training, the number of unique Q-states in the Q-table of each game/player combination, as well as the average number of new states per training game, can be seen in table 1. It is noteworthy that significantly more states are observed when the Q-player goes second, likely due to the randomness of the initial moves when the baseline player goes first. The more complex the game is, the more this initial randomness incites completely different reactions, as seen by the much higher number of Q-states observed for C4 as an 'O' player compared to an 'X' player. The comparative complexity of C4 is also obvious from the massively different number of unique Q-states observed for both players compared to TTT; it will be interesting to see how this difference in the sheer number of observed states translates into performance against the various opponents assessed in the next section.

Table 1: Unique Q-states assessed by Q-Learning player as both first and second mover in the two games. Average number of new states per game played in brackets.

	Unique Q-States Assessed	
Game	Playing as X	Playing as O
Tic-Tac-Toe	4,433 (0.17)	4,899 (0.19)
Connect 4	7,378,698 (14.75)	11,110,631 (22.22)

5 Performances Against Baseline

For both minimax and q-learning, table 2 shows a player's outcomes against the baseline algorithm, both when playing first and when playing second. As TTT games are significantly faster, 1000 trials are performed for all player-algorithm combinations in order to get a more accurate representation of averages. This was not possible for C4, mainly due to the minimax algorithm taking twice as long when playing as X (84 VS 140 ms) and 9 times as long when playing as O (20 VS 178 ms) compared to TTT. As a result, only 100 trials of C4 are shown – still enough to get a general idea of its performance.

Table 2: Outcomes of a minimax and q-learning player against the baseline algorithm, across positions and games. 1000 TTT trials, 100 C4 trials.

		Minimax		Q-Learning	
		X Player	O Player	X Player	O Player
Tic-Tac-Toe	W	875	343	737	430
	L	0	0	36	16
	T	125	657	227	554
Connect 4	W	100	99	70	68
	L	0	0	30	32
	T	0	1	0	0

It is already worth noting that the minimax algorithm takes less time to move when playing as O in TTT, and while playing as X in C4. The result from TTT is theoretically sound, as there are more games to consider when going first, the game tree is more expansive, and updating all the branches would take longer. The different outcome for C4 might be determined by how easily an efficient X player can win – after all, while TTT can theoretically be drawn every time, the X player in C4 can theoretically force wins every time. After a short number of moves, a minimax X player may reach a point where alpha-beta pruning removes most alternatives and

directs the game to a fast win against the mostly random-moving baseline.

The aforementioned efficiency of minimax is clear from the results in table 2 as well. It never loses, and when playing C4, it wins every time as X and only ties 1% of the time as O. Conversely, a tie is very common in TTT, especially when the baseline goes first. This falls in line with the previous assertion about the comparative ease of drawing in TTT, seeing as even a mostly random player can tie 66% of games when going second and slightly over 10% of games when going first. Nevertheless, judging by the 87% winrate as the X player, minimax clearly dominates the baseline.

The Q-learning player is not as profficient at C4, winning 70% of the time as X and slightly over 30% of the time as O. The complementarity of these values implies that the Q-learning player is about as good as the baseline, which was shown to be clearly inferior to minimax in the previous paragraph. It is interesting to note how, in spite of the O player Q-value table being so much larger than the X player table, the Q-learning algorithm performs significantly worse going second, while minimax performed very similarly. This once again attests to the impact and overall advantage of the X player in C4, who can easily bring games to previously unseen states and beat the reinforcement learning agent. The Q-learning agent performs much better in TTT – it does slightly worse than minimax when going first, only winning 73% of the time and even losing sometimes, but it outperforms minimax when going second, winning 10% more often. This is likely because the algorithm got close to converging, and an effective strategy was found to create unwinnable double-threat positions that the baseline player could not deal with.

To make an overall comparison of the algorithms, Q-learning moves much faster (less than a millisecond per turn) than minimax, especially in C4, at the cost of playing C4 substantially worse and TTT as an X player slightly worse. While the q-learning O player loses against the baseline 70% of the time in C4, the fact that it can win more consistently at TTT than minimax makes it a viable choice for the latter game.

6 Peformances Against Each Other

When playing TTT, both algorithms are proficient enough to tie against each other every time. Every game plays out exactly the same – with the final states seen in figure 3 depending on who goes first – due to both algorithms essentially having scripted reactions to the different board states the other player puts before them. This is because the Q-learning player has a fixed ordering of the Q-state values, while the minimax player always reacts the same to the Q-player's moves following its game tree. It is interesting to note that the two algorithms play differently, as the final state differs depending on who goes first; this attests to the high likelihood of drawing TTT, as there are clearly multiple different optimal strategies that still lead to a draw.

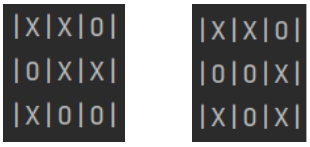

Figure 3: Repeated TTT final game states when the Q-player goes first against minimax (left) and when the Q-player goes second (right).

When it comes to C4, however, the observed gap in performance against the baseline translates to the minimax player never losing, regardless of going first or second. As seen in figure 4, the game goes on for much longer when the Q-learning player goes first, matching this players superior performance against the baseline when going first compared to the instances when it went second. The gap between the players may be diminished if the Q-player trained against a minimax opponent rather than a baseline, however such training would be very computationally expensive, especially since while the minimax player would take a stable amount of time to investigate the game up to a fixed depth from every state, the increasing number of states the Q-learner would have to process as training games against minimax got increasingly longer would likely continue to hamper its performance in a head-to-head matchup.

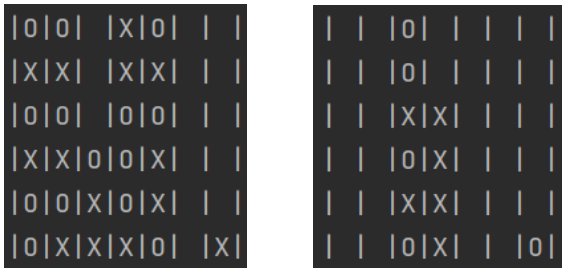

Figure 4: Repeated C4 final game states when the Q-player goes first against minimax (left) and when the Q-player goes second (right).

Comparing the algorithms overall, depth-limited minimax comes out on top since it ties every TTT game while winning every C4 game. While state space reductions and other heuristics based on the human understanding of the game's strategy may improve the performance of the Q-learner, the outcome of C4 appears to be dependant enough on the short term to be consistently won by a minimax player only looking 6 moves ahead. Reinforcement Q-learning would come out on top at the limit however, as training a stronger Q-player would likely take less time than the projected 1400 days per move needed for an alpha-beta pruning minimax player to play the unbeatable terminal-test variation of the algorithm. The fact that a Q-player takes very little time to move once trained gives it a significant advantage over minimax if there are enough resources available to train it well.