Paper Code: COMP717
Artificial Intelligence
Lecturer: Ji Ruan
Assignment 1
Due Monday, 3 May 2021

Name Matt Tribble ID number 19076935

Instructions:

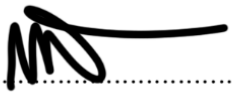
Please attach this sheet to the front of your assignment.

- This assignment contributes 50% towards your final grade. The total mark is 100.
- Choose one of the options A or B and submit an electronic copy through Blackboard before 11:59 p.m. on Monday, 3 May 2021. The submission requirement is specified at the end of each option.
- You **can team up with another student**. If you work as a team, both students should sign this page, but only one submission is needed.

The School of Computer and Mathematical Sciences regards any act of cheating including plagiarism, unauthorised collaboration and theft of another student's work most seriously. Any such act will result in a mark of zero being given for this part of the assessment and may lead to disciplinary action.

Here is a quote from Richard Feynman, a well known scientist, "The first principle is that you must not fool yourself and you are the easiest person to fool."

Please sign to signify that you understand what this means, and that the assignment is your own work.

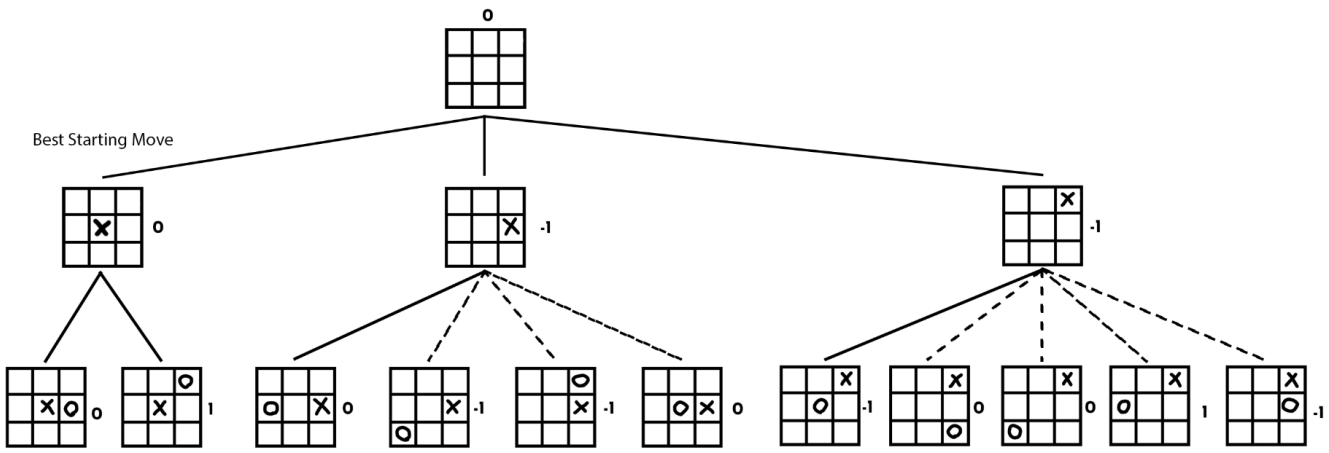
Signature: 

Task One

(a)

On the first move of the game, the player has 9 possible choices, and this decreases by 1 each move. That would logically make the total number of possible game states 9!, or 362880.
However, terminal states can be reached before 9 plays have been made, which reduces the possible number of game states. The game can be won in a minimum of 5, and a maximum of 9 plays, we can take into account this to define a more accurate guess.
There are 8 combinations (ignoring all the other spaces) that define a winning state, so if we assume that for a win in X turns, that the amount of combinations to get to a winning state is 8*3! (all the different ways of aligning the 3 in a row) * (X-3)!, then we have a new total of 321,024

(b-e)



- - - -: Nodes that would be pruned with αβ pruning

(f)

Assuming both players play perfectly, a TicTacToe game will always result in a draw. Therefore, the best opening move to make is the center tile, as that forces the game into a draw. However, if you assume that the other player will not play perfectly, then there is a better opening move, as if you play in one of the corners, and the other player does not play in the center immediately after, you can force the other player into a loss. The reason Minimax doesn't calculate this is because it will only result in a win if, and only if, the other player plays perfectly, which Minimax does not assume.

Task Two

For this task (and Task Three), the program(s) where developed as a .NET Framework Console Application, written in C#.

(a)

Data Structures

For a Minimax Algorithm, you need to generate a tree of game states, and as such a tree data structure needed to be designed. In this implementation, there are 3 core classes, the tree, it's nodes, and the game state itself.

The **state** is an interface that all of the game states can inherit from, as they all have an evaluation function, and terminal states. Any other specifics of the game state are not needed for the tree generation.

A **node** has 4 key parameters:

- The state it's holding
- It's inherent value (determined by the evaluation function of the state)
- The move that lead to this node
- The children of the node

The **tree** holds the root node (which then references the child nodes that make up the tree), and the function that generates the game tree.

As the only difference between a complete tree search and the depth limited search is the depth at which it evaluates the state and stops generating further, the complete tree search is evaluated by setting the search depth to the maximum possible search depth of the game, which is 9 for TicTacToe, and the starting number of chips for the take away game.

Functions

Evaluation

The goal of the Take Away game is to force the other player into taking chips with n+1 number of chips left on the table, where n is the amount that any player can remove, the evaluation function of the game can be written as:

Eval(s):

    if (s) is loss return -10
    if (s) is win return 10
    if X(s) + 1 % n = 0 and is not agent's turn return 5
    if X(s) + 1 % n = 0 and is agent's turn return- 5

Where X(s) is the number of chips on the board for any state (s). This weights moves that lead to the agent having the player on multiples of n + 1.

Minimax

Once the game tree has been generated, the Minimax function takes the terminal values of the tree, and propagates them back upwards to set the values of every non terminal node. This is done as a recursive function, which takes the root node, the search depth, and which player it is acting upon as arguments, and will:

- Return the value of the node if it is a terminal node.
- If it is acting for the agent's turn, in which it is trying to maximise the value, it will iterate over each child of the current node it is searching, and set the value of the current node to the maximum of the minimax of the children, called with a decrement of the depth, and the flipped value of which turn it is acting for. It will also return this value.
- If it is acting for the player's turn, in which it is trying to minimise the value, in which it will do a similar function to the prior condition, but instead of setting and returning the maximum value, it sets and returns the minimum value.

Once run, the tree iterates over the children of the root node at depth 1, and will set the best play possible from the node with the highest value.

Tree Generation

In order to generate the tree, a recursive function creates nodes down to either the terminal state, or the depth limit.

The function takes a node, the current turn, and the current search depth as arguments, and it will iterate over each of the spaces on the board, and provided it is empty, i.e. a move can be made there, it will either:

- Return a node with a value determined by the evaluation function, if the state of the node is terminal or the depth has reached 0
- Generate a new node with the current move in the loop, if the state of the node is not terminal, and then add a child to the current node that is generated by calling the function again, and passing the newly generated node as the argument, alongside the flipped turn and a decremented depth

Resource Consumption & Win/Loss/Draw Rate

TicTacToe

If the player it is playing against plays perfectly, both the full tree search, and the depth limited search will always force a draw, regardless of the depth limit. Even at a depth limit of 1, if the first player plays in a corner, which is the best move to make if your opponent does not play perfectly, it will respond by playing in the center, which is the only play that forces that game into a draw. Therefore, the full tree and depth limited search will always result in the same outcome, although the individual moves they make may be different.

As the state tree for TicTacToe is not that large, it does not require a lot of either time or memory to create, store, and minimax. The largest game tree, that being from a blank board with a maximum

search depth of 9 (which is the maximum depth any game of TicTacToe can get to), takes 1 second to generate and search through, and only has to allocate 80MB to store. As the search depth decreases, the time

Take Away

The full tree search of Take Away will always find the perfect solution to the game, which is, if the removal amount is at or above a multiple of the starting number of chips, the starting player, if they play perfectly. Therefore, it has a 100% of winning, if it is the winning player, and if it is the losing player, it will set itself to be the winning player, if the other player does not play perfectly. Otherwise, regardless of what it played, it would lose.

The depth limited search operates differently, as it cannot see to the final play. Once it can see the final play, it will force the other player into a losing state, but if the number of chips is too large for the search depth to accurately search through, it is possible for the player to force the agent into a losing state, provided they play perfectly.

At smaller numbers of staring chips (X), the full tree search and depth limited search are very similar in terms of the amount of time required to search through the tree, however at larger values of X, the search time increases exponentially [1], and for any search depth d, the search time will be equal to the full tree search for the game state where X=d.

(b)

Program Design

As the user facing component of this program is still text based, the game state has to be represented in a text form. As all of the game states being implemented act as a grid, or as a number, this does not impact the readability of the game state in a large way.

The user can adjust any of the settings for the algorithms in a settings menu, accessed from the main menu where the user selects which game they want to play. In that menu, the user selects the game they want to adjust settings for, and then can change the search depth, the parameters of the game, and the type of algorithm the agent will use.

Each game is stored as its own class, alongside a class used to store and generate the game state tree for that game. The node data structure is abstracted to be used for each game, but each game has their own tree class, as each tree needs to be generated in a different way, based on the rules of the game.

Each game acts as a loop, in which it waits for the user's input, then makes a play based on that input, before generating the best play based on the selected settings for the computer, making that play, and then looping back. If at any point the game's state is deemed to be terminal, the loop breaks, and the result of the game is displayed.

[1] See part (c), Performance Experiments

(c)

Performance Experiments

There are 2 key variables that contribute to the performance of the Minimax algorithm for the take away game, the search depth, and the amount of chips a player can remove on each turn.

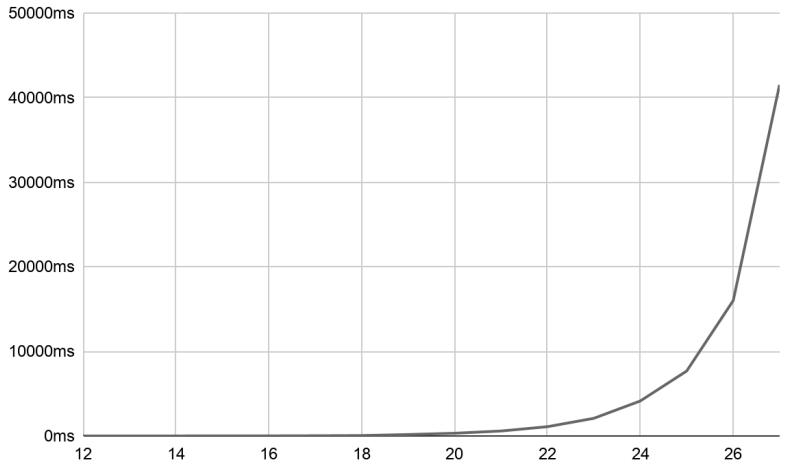
The following tests were run on a AMD Ryzen 2600, and .NET console applications have a maximum available memory pool of 4GB. The following results may vary on other hardware configurations.

Search Depth

The search depth can be any value between 1 and the number of chips that start on the table, as the game tree's depth is bound by the number of chips currently on the table, given the highest depth possible is each player taking 1 chip at a time. To test how the depth affected the search time, the agent was run against itself, incrementing the number of chips and the search depth by 1 after every run, and recording the time it took to search for the first move, which is the most complex when searching the entire game tree. The maximum number of chips that can be removed for these tests was 3.

Search Depth	Time (ms)
12	0
13	1
14	1
15	6
16	8
17	25
18	49
19	168
20	319
21	583
22	1093
23	2084
24	4132
25	7679
26	15992
27	41527

From this data, we can conclude that an increase in the search depth results in roughly two to three times the search time, which aligns with the expected increase, as Minimas has a time complexity of O(b^d), of which we are increasing d, with a b of 3, resulting in an exponential increase as demonstrated.



A search depth beyond 27 requires more memory to store than is allocated for a .NET console application (4GB), and so 27 is the maximum search depth that can be calculated, at least for a chip removal of 3.

Limit of Chip Removal

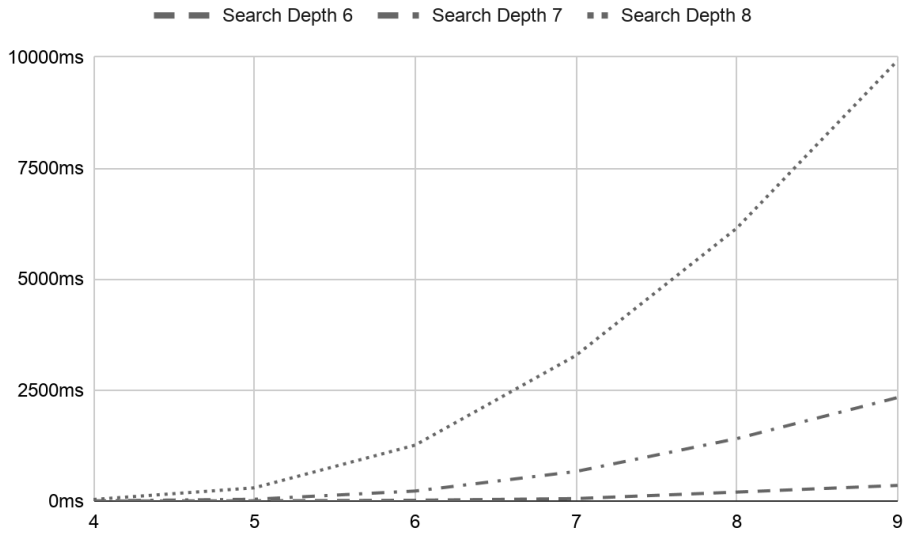
The amount of chips each player moves increases the number of branches possible at any given node. For these tests, a similar setup was used, in which the algorithm was played against itself, but this time it iterated over the removal limit, as well as the search depth. The amount of cards starting on the table was left at 20, and a subset of the data has been recorded.

With this data, we can conclude that an increase in the number of chips that can be removed (and therefore the branching factor of the game tree) results in a roughly polynomic increase in the search time.

This is exactly the distribution that would be expected from the given time complexity of Minimax, as of O(b^d), this test is increasing the b for a given d, and the rate at which increasing b increases the search time also increases for an increase of d.

Effectively, the higher the search depth, the faster increasing the removal limit increases the search time.
However, if either variable is static, increasing the removal limit increases the search time less than the search depth.

Search Depth	Removal Limit	Time (ms)
6	3	0
	4	0
	5	5
	6	18
	7	56
	8	202
	9	352
7	3	0
	4	4
	5	41
	6	226
	7	666
	8	1407
	9	2332
8	3	1
	4	36
	5	298
	6	1260
	7	3280
	8	6148
	9	9925



Task Three

Game Design

The game chosen to implement for this task was Connect 4. In Connect 4, you have a 7x6 grid of tiles, and 2 players, each with a stack of different coloured chips (in this implementation, the different players are represented in the same way as TicTacToe, with O's and X's). Each player takes turns to play, and can play in any of the 7 columns (provided it is not full), in which they "drop" one of their chips, which land on the lowest tile in that column that does not have a chip in it already. The game ends when either player makes a row of 4 of their coloured chips, in any direction (horizontal, vertical, or diagonal), which is a winning state, or when the grid is full, which is a tie state. Therefore, the goal of each player is to make a row of 4 without allowing the other player to do the same.

Evaluation Function

As the goal of Connect for is to make rows of 4, the evaluation function of the game state is weighted accordingly, and can be written as follows:

$$Eval(s) = X_1(s) + 2X_2(s) + 3X_3(s) - (O_1(s) + 2O_2(s) + 3O_3(s))$$

Where A_n is, for every group of 4 tiles that can lead to a win, the amount of unbroken rows of n chips of type A, that do not have any chips of the opposite type in that group. Any winning state has an evaluation of 100, and a losing state has an evaluation of -100.

This evaluation function weights chains of chips higher than single chips, but it also ignores any column, row or diagonal that can no longer be used to win the game for either player, as once a chain is broken, for either player, that chain is no longer relevant to getting that game to a terminal state.

Search Algorithm

As Alpha Beta Pruning can be seen as a subset of the Minimax algorithm, that benefits by eliminating the requirement of searching every branch of a tree, it has been implemented not as another search algorithm to find the value of a given node, but rather as a process during the game tree's creation.

When a node is created, it's value is determined from the Minimax algorithm, and that value is compared to the alpha and beta values, which are updated based on the current turn at that node in the tree. If a >= b, it can be assumed that the rest of the moves at that current node are not better than the one that was just checked, so the node ends the generation of the children early.

This allows branches that are pruned to not need to generate the rest of the game tree, which improves performance substantially more than if the algorithm was just not checking the value of pruned branches, as that results in an undetectable change in performance.

Resource Consumption & Win/Loss/Draw Rate

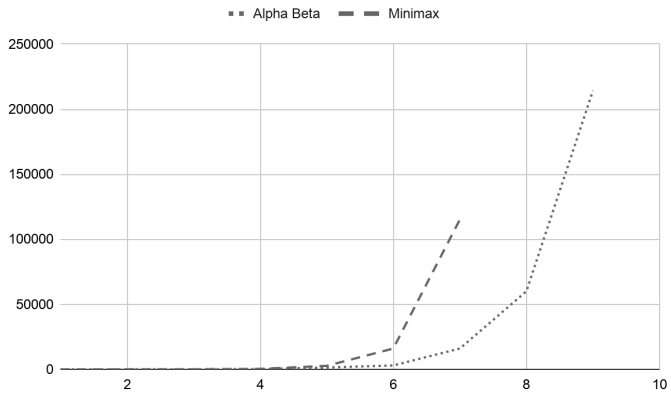
Below is a table of the (approximate) resources used for both Alpha Beta Pruning and the Minimax algorithms searching for the optimal move from an empty board.

Alpha Beta Pruning		
Search Depth	Time (ms)	Memory
1	1	N/A
2	1	N/A
3	1	N/A
4	6	1MB
5	40	4MB
6	200	14MB
7	534	30MB
8	3400	150MB
9	13000	540MB
10	47978	2GB
11	DNF	4GB

Minimax		
Search Depth	Time (ms)	Memory
1	3	N/A
2	3	N/A
3	4	N/A
4	9	1MB
5	70	6MB
6	530	23MB
7	3900	190MB
8	27490	1.3GB
9	DNF	4GB

As Alpha Beta Pruning has no effect on the minimax value of the root node, the success rate of both algorithms are identical, as their plays for any given game state will be the same, given the same search depth.

Therefore, the only benefit of the alpha beta pruning algorithm is the saved time and memory for a given search depth. Alpha Beta Pruning decreases the amount of search time for a given search depth, as such decreases the rate at which increasing the search depth increases the search time. However, as the increase is still exponential, the benefit only realistically helps for a small number of extra search depths.



The Alpha Beta Pruning algorithm reduces the time of the search for a given search depth down to roughly what that search time would be for the Minimax algorithm at a search depth 2 less than the search depth of Alpha Beta. Effectively, it increases the maximum possible search depth by 2.