Project 2: Wordle MCTS

Ingólfur Ari Jóhannsson

T-622-ARTI Gervigreind

RU Science and Engineering

April 7, 2023

1 Introduction

Game-solving algorithms are a popular area of interest in the field of artificial intelligence. One such game that has gained popularity in recent years is the Wordle game. Wordle is a word-guessing game where the player is given a random word of a certain length and has to guess it in a limited number of tries. The game has gained a lot of popularity. The author of this short paper was interested in finding an optimal guessing path that can solve this game better than humans. This paper will outline how that attempt was made.

The goal of this paper is to implement a Monte Carlo tree search (MCTS) algorithm to solve the Wordle game as a single-player versus MCTS. The MCTS algorithm is a popular technique used in game-solving problems, especially in games where the search space is vast. It is a tree-based search algorithm that uses a combination of Monte Carlo simulations and a search tree to determine the best move to make at each turn. We focus on solving the Wordle game for word-sizes ranging from three letters to seven. We aim to show the effectiveness of the MCTS algorithm in solving the game and compare its performance with different word-sizes, parameters and exploration constants. The paper will present an explanation of the MCTS algorithm, its implementation in solving the Wordle game, and the results obtained from the implementation.

The paper's conclusion will summarize the findings and highlight the areas for improvements.

2 Methods

Monte Carlo Tree Search (MCTS) is used in games where uncertainty and partial information exist. It has been successfully applied in various games such as Go, Chess, and Poker, and is known for its efficiency and scalability. It is suitable for this papers purposes as the potential state-space for the Wordle game is quite big or circa w^n, where w is the number of words and n is the depth of search.

The algorithm is based on a tree data structure that represents all possible nodes of the game being played, and each node has it's own state.
The root node represents the initial state, and its children represent possible actions that can be taken from there, each node deeper thereafter represents a subsequent action taken, and so on until the correct word is reached. The implementation uses a changing state-space for nodes visited that adapts to the word being guessed on each time when training. The problem with this approach will be discussed in the conclusion.

The MCTS algorithm works in four stages: selection, expansion, simulation, and back-propagation. In the selection stage, the algorithm traverses the tree from the root node to an unvisited leaf node through the most promising path in the tree, based on a selection strategy. One popular strategy is the Upper Confidence Bound for Trees (UCT), which balances exploration and exploitation of the tree by selecting nodes with the highest expected reward and a high uncertainty score. Next comes the expansion stage, the algorithm adds one child node to the selected node, by checking possible actions from that node. In the simulation stage, the algorithm plays out a game from the newly added child node to a terminal state. This is done by selecting random (Monte Carlo) moves. This could be more sophisticated, such as a heuristic function that evaluates the game state and the words available.
Finally, the back-propagation stage, the simulation results are propagated back up the tree, updating the expected reward of each visited node along the path from the root to the selected leaf node. This information is used to guide future selection and expansion of nodes in the tree.

2.1 Program plan

The steps needed to make the program are the following:

#	Step Description
1	Create a collection of valid words for the game.
2	Load the Monte Carlo Tree from a pickle file if it exists, otherwise create a new empty one.
3	Prompt the user to choose to play a game type or exit.
4	If the user chooses to exit, then save the Monte Carlo Tree as a pickle file and exit the program.
5	If the user chooses to play a new game, then prompt the user to enter the word length and guess amount. If not and he wants to train jump to step 12
6	Generate a guess randomly.
7	Prompt the environment to output feedback on the guess.
8	Use the feedback to update the Monte Carlo Tree.
9	Repeat steps 6-8 until the guess is correct or the user runs out of guesses.
10	If the user chooses to quit, save the Monte Carlo Tree as a pickle file and exit the program.
11	Repeat steps 6-10 for the second player.
12	Play a training game, with both players using the same Monte Carlo Tree, to train the Tree with an incentive to win faster than the counter-party. Make the training highly customizable
13	Save the Monte Carlo Tree as a pickle file after a specified number of simulation games.
14	Prompt the user to choose to play a new game, play a simulation game or quit.
15	Repeat steps 3-13 until the user chooses to quit the program.

This is obviously a simplification of how the game will be implemented, but it gives the general idea of how the flow of the project will be.

3 Results

During the implementation process, I encountered several challenges that impacted the performance of the algorithm. One significant challenge was the size of the English dictionary data-set for all of the word sizes, this was way too large for the training I was able to do, making it not effective against a human player. Additionally, testing the tree against an opposite agent that was also playing semi-optimally was not wise in hindsight as it was difficult for the tree to adjust to unexpected player moves. This means that when a player does something unexpected the tree only has weak responses as it has not frequented that node often. This is not due to overfitting since this was the case regardless of how greedy the tree was made to be with the exploration constant. There was also an attempt to train the tree on mixed C constants as it was possible to change it between runs.

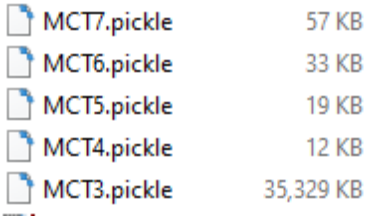

Figure 1: Extensively trained MCT of word size 3 with a mix of different exploration constants, trained upwards of 100.000 times.

Despite these challenges, I was able to successfully implement the MCTS algorithm. But the implementation is only suitable for optimal-play, and is dead in the water against anything unexpected. This is purely the fault of the approach chosen for this project, when the tree is traversed, the character limitations are not carried though the nodes, so the contextual information when utilizing the trained tree needs to be improved for the tree to be more effective. The implementation works, the program is able to make a very logical tree when you look at smaller word pools (like the demo files provide). Utilizing the tree though is not performing adequately. Overall, this project has been a valuable learning experience in implementing AI algorithms and working through challenges that arise during the implementation process. It has also highlighted the importance of practical experience in understanding the limitations of algorithms and the need for efficient algorithms that can address real-world problems, there were a lot of situations where I initially programmed an inefficient solution and had to think outside the box to improve performance.

What is impressive, is how the training seems to reach a stable value, it does not slow down the more you train it. Which is something that was not expected to happen.

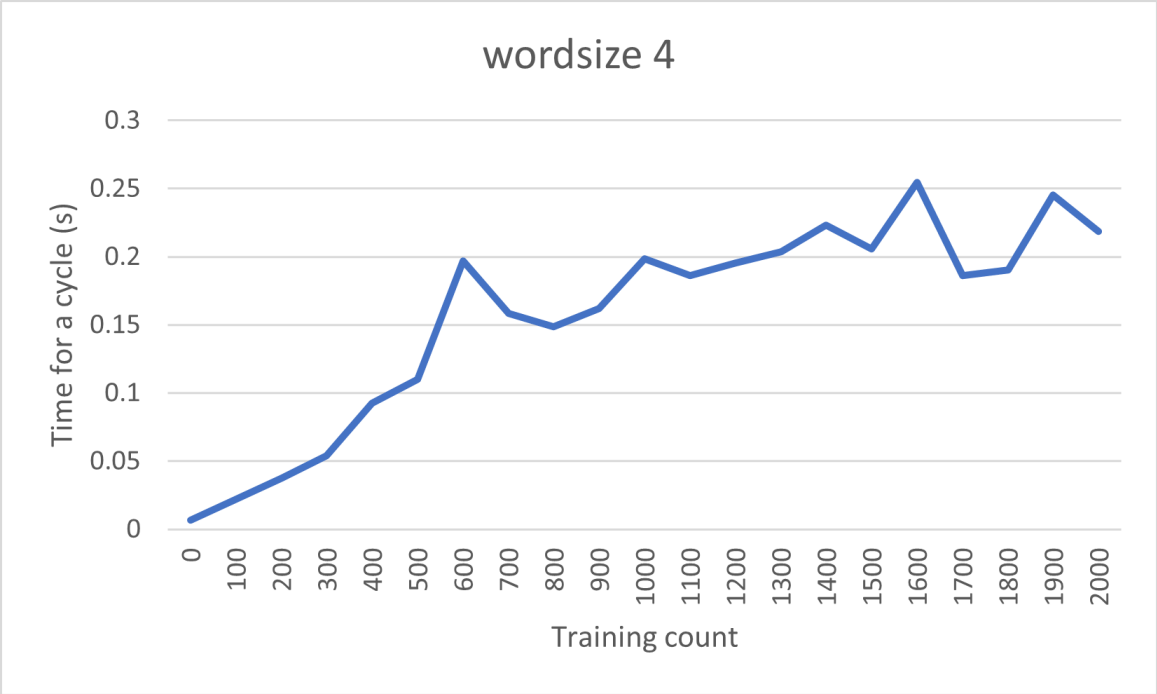

Figure 2: Trained MCT of word size 4 from 0 training runs to 2000 training runs. The times are measured for a single run at the measurement interval 100.

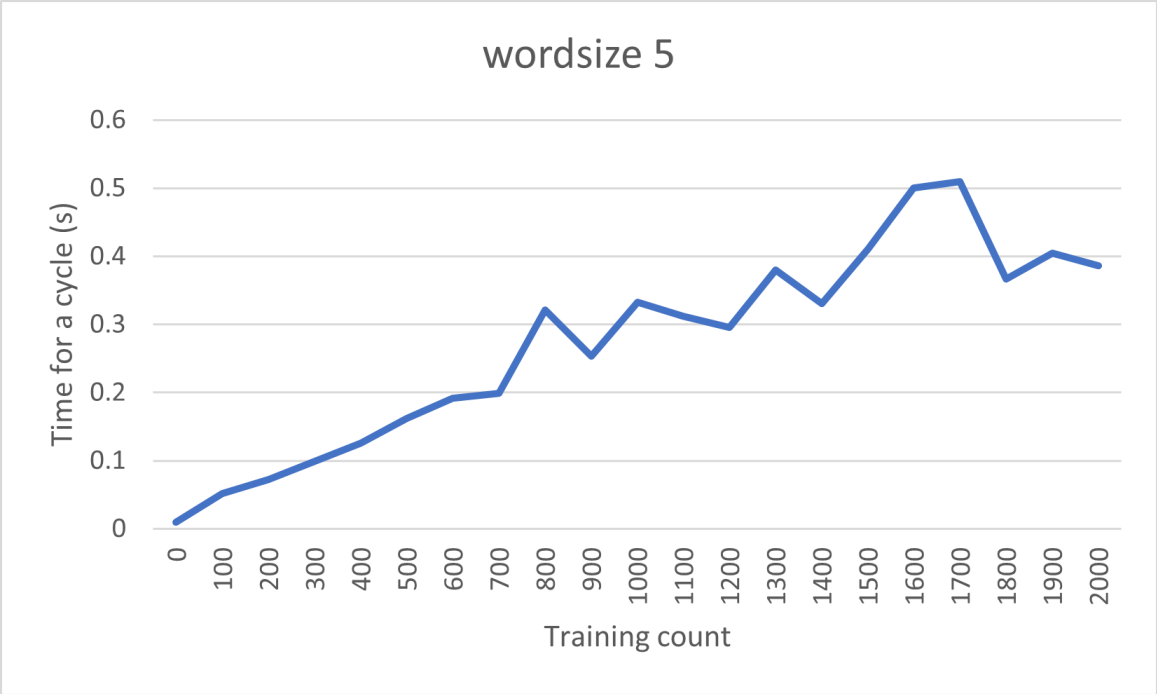

Figure 3: Trained MCT of word size 5 from 0 training runs to 2000 training runs. The times are measured for a single run at the measurement interval 100.

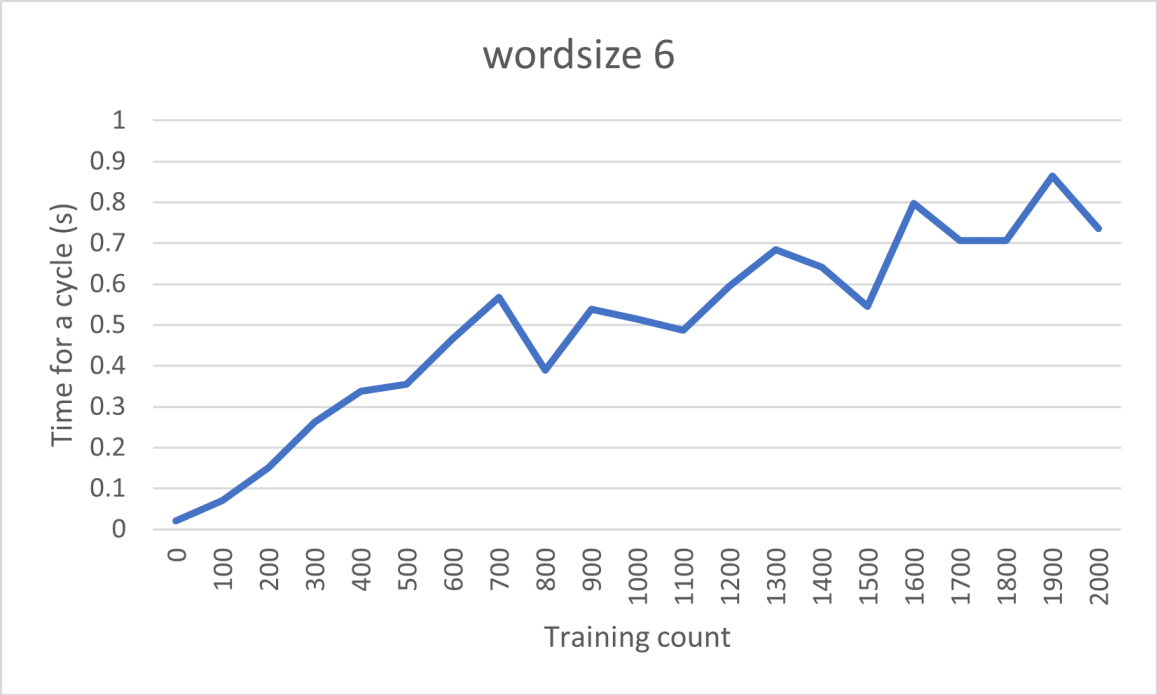

Figure 4: Trained MCT of word size 6 from 0 training runs to 2000 training runs. The times are measured for a single run at the measurement interval 100.

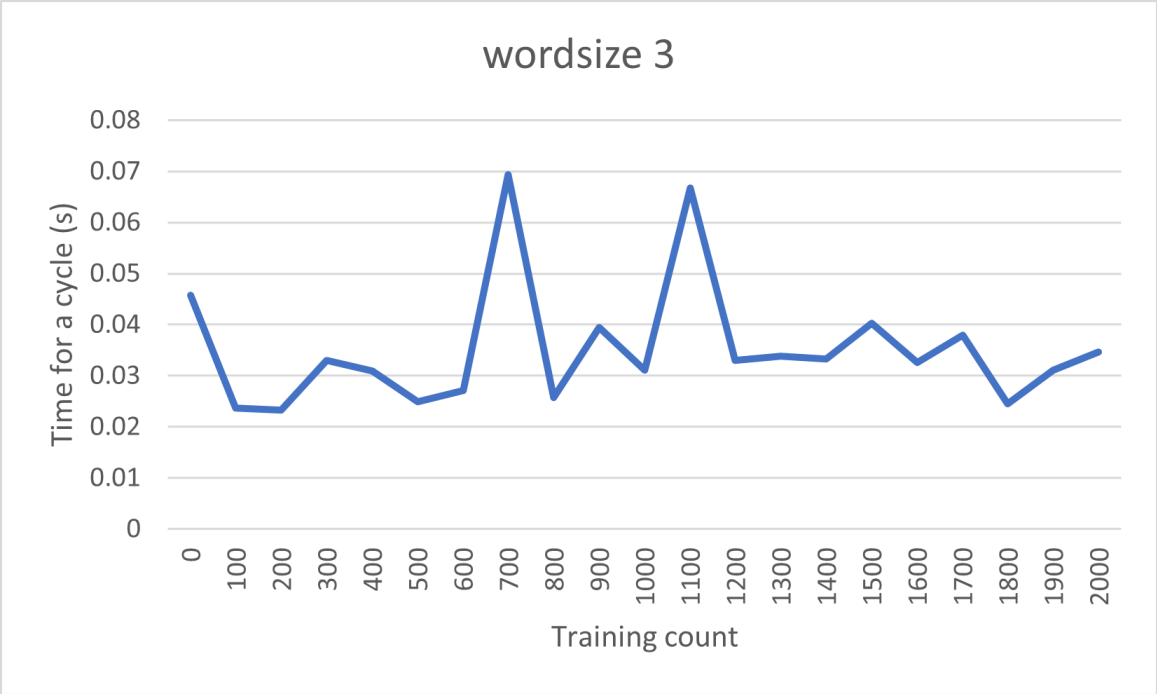

Figure 5: Trained MCT of word size 3 from over 100.000 training runs, with the same method measuring 2000 additional training runs. The times are measured for a single run at the measurement interval 100.

Here in the figure aboce it is clear that at higher training counts the time it takes to run 1 additional training cycle reaches a limit and stabilizes.

The author would like to point out that if you run the files named "$3a - demo.txt$" and "$3v - demo.txt$" you will get much better results, these files contain dummy words that showcase how the project performs on smaller data-sets, you only need to change these filenames to "$3a.txt$" and "$3v.txt$" respectively, and run the **return resettree()** line in Wordle.py line 122 to reset the pickle files. Then you need to train the tree a little bit (5 second run for example), after this is done the tree is good and practical.

4 Conclusion

In summary, the implementation of the Monte Carlo Tree Search (MCTS) algorithm for the Wordle game has been a challenging yet rewarding project that has provided valuable practical experience in implementing AI algorithms. While the current implementation has some limitations, I am sure that it can be improved by incorporating the following possible enhancements:

- Enhanced Heuristic Function: Use parent words and their feedback values and incorporate those into the heuristic, on how we cut out the words for the space of the current node. For example if we get this feedback "Cc–" and "C-c-" from parent nodes, we should be able to make the assumption that the last letter is the correct one.
- Refined Feedback Mechanism: The feedback mechanism used in this project was crucial, but it can also be improved upon. For instance, we can assign a score to each word in the word-bank based on the frequency of the letters in the words, as well as how well-positioned those letters are. This would also act as a heuristic function, enabling the algorithm to guess the most commonly structured words more effectively.
- Optimized Exploration Constant: Determining the optimal value of the exploration constant can be challenging, but it is crucial for the algorithm's effectiveness. With a larger tree, we can collect more statistics

to find the sweet spot of the constant that yields the best results. In this project, the tree was trained for only around 100.000 runs at most, resulting in pickle files that were excessively large.

That being said, I'm pretty sure that my approach to this problem was wrong and not suitable for this game. Lets talk about why, the nature of the state-space. This game is very fluid, the possible words at each node change depending on what word is correct at this moment. But the MCTS has static nodes, that is when we explore in the tree the added node stays there, but the word-bank you can guess at that node is changing. This changing of the states means that every node has a child of all other words at all nodes in all depths, so the state-space does in fact not get smaller but by 1 word for each depth you go.

Overall, this project has provided a valuable learning experience, and the possible enhancements listed above can be explored to further improve the effectiveness of the MCTS algorithm for the Wordle game.