

# WORD SOUP: ADVENTURES IN USING MATHEMATICS TO SOLVE WORD GAMES

Robin Lyster  
Student I.D.: 10459970  
Supervisor: Dr. Neil Morrison

MAY 2022  
MATH30000 - Double Project

# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
2.1	Background . . . . .	2
2.2	<i>Word Soup</i> . . . . .	3
2.3	Aims of Project . . . . .	9
<b>3</b>	<b>Simulating the game</b>	<b>14</b>
3.1	Creating the Board . . . . .	14
<b>4</b>	<b>The first algorithm</b>	<b>19</b>
4.1	The greedy solution . . . . .	19
4.2	Improving the algorithm's efficiency . . . . .	30
4.3	First set of Results . . . . .	35
4.4	Next Steps . . . . .	37
<b>5</b>	<b>Adding heuristics</b>	<b>38</b>
5.1	How humans play <i>Word Soup</i> . . . . .	38
5.2	Two-move greedy . . . . .	43
5.3	<i>X</i> -move greedy . . . . .	44
5.4	More heuristics . . . . .	46
5.5	Valuing uncommon letters . . . . .	49
5.6	Vowel to letter ratio . . . . .	52
5.7	Connectivity . . . . .	55
5.8	The heuristic model . . . . .	58
<b>6</b>	<b>Conclusion</b>	<b>61</b>
<b>7</b>	<b>Appendix</b>	<b>64</b>
7.1	Variable setup . . . . .	64
7.2	Game Setup . . . . .	66
7.3	Searching for words . . . . .	67
7.4	Heuristics . . . . .	73
7.5	Code execution . . . . .	75
7.6	<i>x</i> -move greedy algorithm . . . . .	78

# Chapter 1

## Abstract

*Word Soup* is a word game that involves selecting words on a jumbled grid of letters, and attempting to achieve a high score by choosing long words with uncommon letters and clearing the board. This project sets out to create a heuristic algorithm that can beat humans in the game *Word Soup*. By creating a simulated version of the game in Python and creating a depth-first search that could locate words and remove them from the board, I created a greedy algorithm that would select the one-turn optimal word every turn. The greedy algorithm consistently beat the amateur player, but the expert player was still scoring higher by clearing the board consistently. Then, I analyzed the differences in how the game was played by the expert player and the greedy algorithm, and looked at what heuristics could be used to improve on the greedy algorithm. After adding 3 different heuristics to the greedy algorithm which considered the composition of letters left on the board after each turn, the average score improved and the average number of letters left on the board halved in comparison to the greedy algorithm. This also marked a narrowing of the gap in performance between the algorithm and the expert player, both in score and amount of board cleared.

## Chapter 2

# Introduction

### 2.1 Background

Word games have existed for nearly as long as games themselves, and have captivated people from all walks of life. Crosswords and word searches are printed on nearly every newspaper in the English-Speaking world. Games like Scrabble and Bananagrams are staples within many families' board game collections. Even in the last few months, word games have surged in popularity thanks to *Wordle* [1], a game where you have to try and guess a five-letter word in 6 attempts, being given the correct letters and if they're correctly placed after each guess. There is also a very strong competitive element to word games, with world championships in Scrabble [2], and people attempting to get the lowest average guess number in *Wordle*. But, a group of people that happen to be perhaps most fascinated by such games may come as a surprise to many: Mathematicians. But why?

The most misleading part of many word games is in the name itself. Although games like Scrabble are on the surface about knowing and finding words, a much more integral part of the game revolves around finding patterns and optimizing a strategy. *Wordle* is much less about knowing every 5-letter word and much more about collecting as much information as possible on each guess, in order to minimise the number of potential words the solution could be [3]. Ultimately, a word game is still a game, which can be strategised and potentially solved.

In recent decades, people have also been attempting to solve games using computers. Some games, like *Noughts and Crosses*, are completely solved games, with an optimal strategy for every turn [4]. The most ideal next word can be calculated in *Wordle* using information theory, but won't necessarily beat a human player's sub-optimal guess due to the element of uncertainty. Most other games aren't yet solved or can't be, and require much more complex algorithms that won't or can't know the ideal move to make. For example, *Chess* has far, far too many moves and pieces to be solved with current technology, with there being orders of magnitude more possible games of chess than there are atoms in the universe [5]. In fact, the best *Chess* playing algorithms are neural networks that were fed with billions of games to create a "black box" that can beat all human players, but at the sacrifice of our understanding of how it does. As a direct result of this lack of human input into these neural networks, current chess Grandmasters often use assistance from neural network algorithms to inform their chess strategies, almost a reversal of how algorithms are initially informed by human strategy [6]. This suggests that the truly optimal strategy for games of such complexity lies beyond human capabilities, and I aim to explore aspects of this idea for this project.

## 2.2 *Word Soup*

*Word Soup* is a word game available on most mobile phones [7], which functions like a cross between a word search and *Candy Crush*. The player is presented with a 12 by 9 grid of semi-randomized letters, like so:

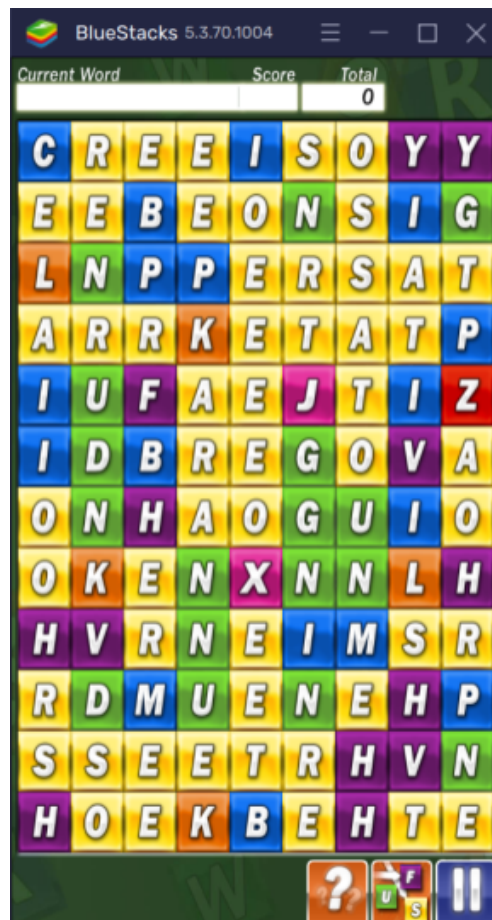


Figure 2.1: A typical Word Soup board, before playing.

Each turn involves the player eliminating one word from the grid, where each adjacent letter in the word must also connect to the next adjacently or diagonally on the board, with no letter being used twice.

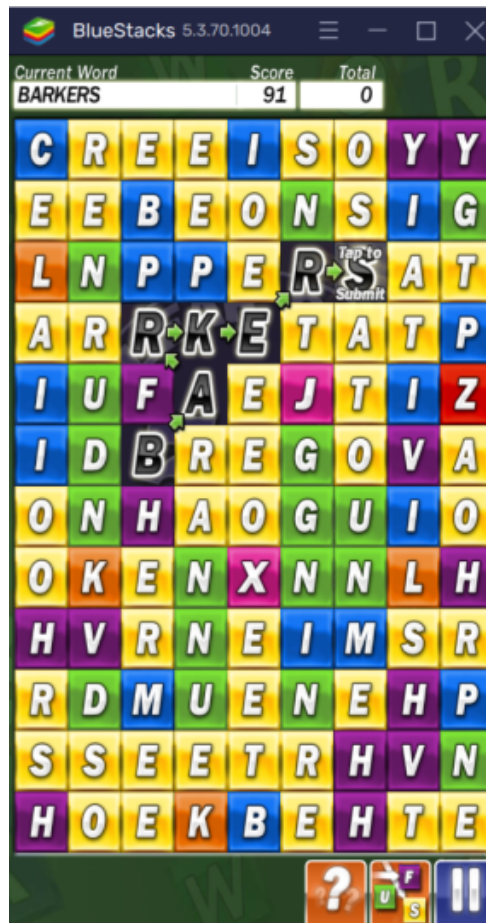


Figure 2.2: A Word Soup board with a word selected. In this case, it is the word “barkers,” which snakes around in many directions. The word must be at least three letters long!

Those letters are then eliminated from the board, and the remaining letters then “fall down” to any empty spaces below, and then slide horizontally to any empty spaces to the player’s choice of side.

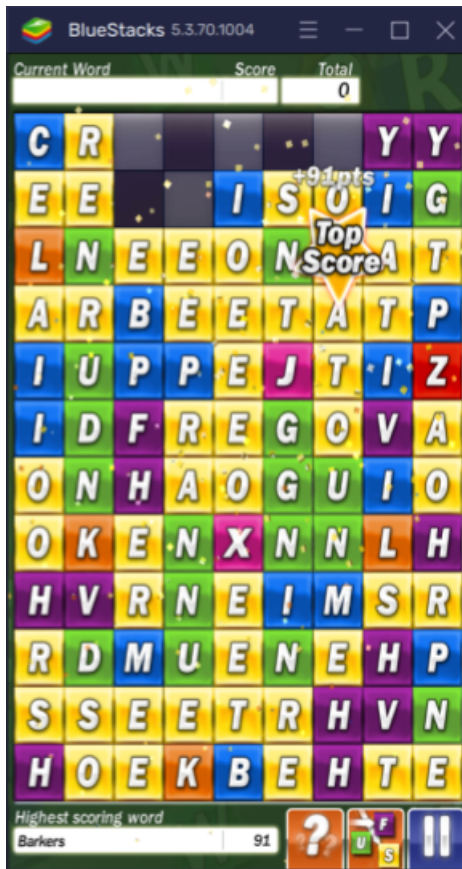


Figure 2.3: Now the letters have been eliminated, the letters above have fallen down vertically.

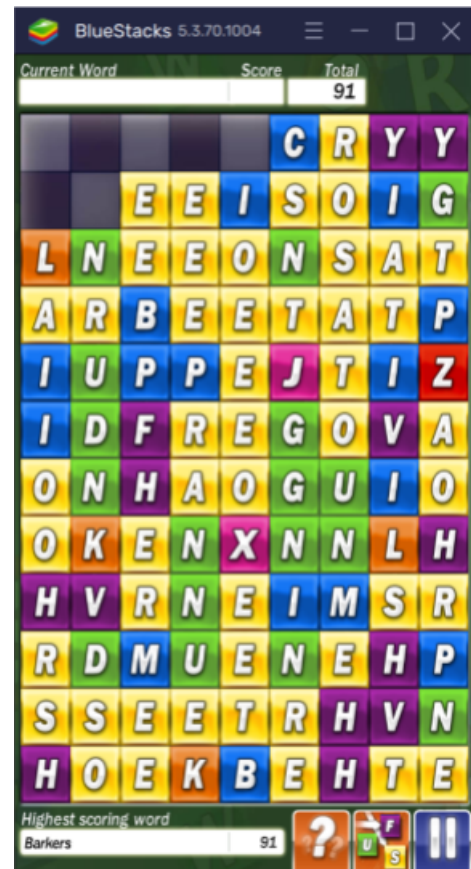


Figure 2.4: Then, the letters with empty spaces to the right slide over to fill them, leaving the updated board.

A point score is then added to a running total, which is equal to the product of the length of the word and the sum of the point scores of the letters used in the word (see Table 2.1 and 2.2).

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Point Score	1	3	3	2	1	4	2	4	3	8	5	5	3
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Point Score	2	1	3	10	1	1	1	2	4	4	8	4	10

Table 2.1: A list of all letters and their point values.



Word	Sum of letters	Length	Total Score	<i>Scrabble</i> Value
Barkers	13	7	91	13
Challenge	24	9	216	15
Set	3	3	9	3
Oratresses	10	10	100	10
Jazz	28	4	112	28
Ox	9	2	N/A (too short)	9
Quizzifications	51	15	765	37 (1 blank used)
Absentmindedness	25	16	400	N/A (too long)

Table 2.2: An example list of words, showing their point values along with their equivalent Scrabble score.

With the new, updated board, the player is now free to choose another word, which is eliminated in the same way.



Figure 2.5: After quite a few words have been selected, the board becomes smaller and concentrates in the bottom-right hand corner. Fewer words are available, and scoring becomes harder.

The game ends when the board contains no more words within it. If every letter is eliminated, 500 points are added to the total score.



Figure 2.6: Here, the last word has been selected. When removed, the game will end as no possible words remain. As there are still going to be 3 letters left, the 500 point bonus is not attained.

The objective of the game is to obtain the highest possible score before the game ends. My aim for this project is to create a heuristic algorithm that can produce a better score in a game of *Word Soup* than a human can.

## 2.3 Aims of Project

My stated aim is to create a heuristic algorithm that can best humans in *Word Soup* a majority of the time. But what is a heuristic algorithm and how good are human players?

A heuristic algorithm is one that isn't necessarily optimal, but instead one that can find a sufficiently good solution given the computing power and time at hand [8]. I am using a heuristic algorithm because, as with many games, despite there being no information withheld from the player at the beginning of the game, it is computationally infeasible to find the optimal solution. This is because the amount of computing power required to solve a game increases quicker than exponentially with the number of tiles on the initial board, and in this case, the initial board is much too large for an optimal solution to be feasible.

To elaborate, solving a game of *Word Soup* would require creating a vast tree of every possible combination of moves and finding the highest total score for all of them. There are nearly 300,000 valid English words you can play in the game, all of which need to be checked against the board for every move made. As the board gets bigger, the amount of letters that need to be checked also increases. Furthermore, letters are more connected in bigger boards than on smaller boards (as the area to perimeter ratio increases), so more words are likely to be found. This gives a rough estimation of the computing time of rigorously solving a board to be the product of the time it takes to find a word for every board size encountered in a game, which is at a minimum a factorial function. These grow quicker than exponential functions, and with an initial dictionary size so large will also take a long time even on relatively small boards. At 108 letters, solving a board would take orders of magnitude more time to solve than our lifespan.

For quantifying how good human players are at *Word Soup*, I collated 10 games from both myself (an amateur player) and an expert player [9]. Here is a breakdown of these games (see Table 2.3):

Game no.	Expert word score	Bonus	Total	Amateur word score	Bonus	Total
1	1450	0	1450	1038	0	1038
2	1594	0	1594	1062	0	1062
3	1342	500	1842	1090	0	1090
4	1410	500	1910	1120	0	1120
5	1460	500	1960	1157	0	1157
6	1512	500	2012	1187	0	1187
7	1534	500	2034	1225	0	1225
8	1626	500	2126	1244	0	1244
9	1774	500	2274	1277	0	1277
10	1785	500	2285	1309	0	1309
Average Score:		1548.7	1948.7	1170.9		1170.9

Table 2.3: A sample of 10 games from an amateur player and an expert player (the players are using different sets of 10 grids).

From the sample of 10 games, I managed an average score of 1,170.9, of which none of the games incurred a bonus score for emptying the board. On the other hand, the expert player scored an average of 1,548.7 from words scores alone, and got the 500 point bonus 80% of the time, giving a total average score of 1,948.7. This gives us some reasonable milestones to aim for when designing the algorithm, as well as some insights into where points are generally scored by human players. For example, an amateur player scores consistently lower than an expert player for word score. This is not only because an expert will be better at finding better words than an amateur, but also because an amateur player will rarely consider planning several moves ahead, and without much other strategy will try and eliminate the best word they can find each time. The better a player gets, the more strategy and forward planning is done. This includes planning a few moves ahead to make a very high-scoring word, whose score will vastly outperform that of the highest-scoring words played by the amateur. This is not in small part due to the length of the very high-scoring word being used as a multiplier in the word score. A 7-letter word will be worth only a quarter the points of a 14-letter word, assuming a similar mix of letters are used.

Furthermore, an amateur human player is unlikely to be able to clear the board and obtain a 500 point bonus, suggesting it is a difficult task that requires forward planning. This is in fact the case. Paying attention to the shape of the board and the letters remaining on it will make gaining

the 500 point bonus more likely. Nevertheless, there are many pitfalls that make it incredibly difficult to clear the board. This can be seen with some examples of unfinished boards.



Figure 2.7: A completed game of *Word Soup*, where there are still several letters left, all of which are consonants.



Figure 2.8: This completed game has a mix of consonants and vowels, but they are all spread in one long line, making them unusable.

Some boards end up with the wrong combination of letters left to form words. This may be due to a lack of balance between the number of vowels and consonants remaining, or perhaps that the letters are too spread out to allow for choice of direction, meaning only words existing on a straight line path can be taken. Sometimes, it's just the presence of an annoying “Q” with an absent “U” that prevents a full clear.

This helps inform my further aim: not only to outperform human players, but also to deduce a good strategy for playing the game that will not only get high-scoring words, but even attempt to set up board clears, and maybe even use the results of running and tweaking this heuristic algorithm to figure out strategy that even human players may be able to learn from.

## Chapter 3

# Simulating the game

### 3.1 Creating the Board

To attempt to find a computerized solution to *Word Soup*, I first need to build a simulation of the game on which I can run algorithms. For this, I will be using Python, and will be coding all algorithms in Python as well.

First, a board has to be created. The easiest way to store a two-dimensional board which can then be manipulated later is by creating a list of lists, with each row being stored as a list of individual characters (one for each column within the row) which itself is part of a list of rows. Then, for each letter space in the grid, a random letter is picked from a string of characters containing an amount of each letter in rough proportion to their frequency in the English dictionary, and is assigned to that space. I chose the frequency within the English dictionary instead of the English language because this more closely resembles the distribution of letters on the board generated by the *Word Soup* app, which makes sense as all words should theoretically be given equal weighting, as the frequency of a word in the English language has nothing to do with the game. Also, for the purposes of recreating the game, I am using the *Collins* Scrabble dictionary [10], which contains the most comprehensive collection of English words of any dictionary that doesn't also include acronyms or proper nouns (which aren't allowed in the game). We don't know the full list of words in *Word*



*Soup's* internal dictionary, but by doing some spot checks on obscure words we can surmise that it is at least as extensive as Collins, except for the non-playable two letter words. This creates a pseudo-randomly generated board:

```
[ 'i', 'h', 'f', 'c', 'a', 'a', 'u', 'o', 'o' ]
[ 'a', 'l', 't', 'e', 'a', 'n', 't', 'k', 's' ]
[ 'n', 'e', 'u', 'r', 'm', 'e', 'e', 'a', 'i' ]
[ 'r', 'v', 'y', 'u', 'a', 'r', 'a', 'a', 'v' ]
[ 'f', 'l', 't', 't', 'e', 'b', 'i', 'v', 't' ]
[ 'i', 'l', 'j', 'i', 'o', 'o', 'd', 'e', 'n' ]
[ 'n', 'r', 'h', 'd', 'h', 'a', 'k', 't', 'e' ]
[ 'i', 't', 'n', 'a', 'h', 'b', 's', 't', 't' ]
[ 'i', 'e', 'a', 'p', 't', 's', 'n', 'e', 'w' ]
[ 'm', 't', 'o', 'n', 'a', 'c', 's', 'l', 'u' ]
[ 'o', 'n', 'i', 'r', 'a', 'u', 'h', 'r', 'e' ]
[ 'j', 'r', 'h', 'c', 'e', 't', 'r', 'n', 'o' ]
```

Figure 3.1: An example of a simulated Word Soup board, generated by my code. The outermost square brackets have been removed for easier viewing.

The next function necessary to simulate the game is to remove a word from the board and calculate the score. This can be further broken down into steps that can be executed by code:

1: Store a string of characters within the board by referencing their positions within the grid. There are many ways to do this, and storing a list of positions by row and column seems like a tempting option. But, when considering the context of finding a word, it is much easier to instead store a list of characters, and also store a second list of equal length containing numbers corresponding to the occurrence of each of that letter you have to skip through on the board to find that letter. For example, the lists `['t', 'h', 'e']` and `['4', '0', '5']` would indicate that this string of characters is composed of the 5th 't', the 1st 'h' and the 6th 'e' on the board, counting left to right, top to bottom (note that number order on a list begins at 0 in Python).

```
methanation 231 [[1, 7, 6, 1, 9, 4, 10, 10, 7, 4, 6],
[1, 7, 6, 1, 9, 4, 10, 10, 7, 4, 7]]
```

Figure 3.2: The two pathways that the word “methanation” can take on the grid in Fig. 2.1. The word has a score of 231, which is the length of the word (11) multiplied by the sum of the letter scores (21).

2: Remove the specified characters from the grid. This can be done by iterating through the board once for each character in the string, finding the specified occurrence of that character, and adding a marker to the character (see Fig. 2.3) (if you delete the character outright, the ordering of all subsequent occurrences of that letter on the board will now have the wrong occurrence number). Then, remove every marked character on the board (see Fig. 2.4).

```
methanation 231
['i', 'h', 'f', 'c', 'a', 'a', 'u', 'o', 'o']
['a', 'l', 't', 'e', 'a', 'n', 't', 'k', 's']
['n', 'e', 'u', 'r', 'm', 'e', 'e', 'a', 'i']
['r', 'v', 'y', 'u', 'a', 'r', 'a', 'a', 'v']
['f', 'l', 't', 't', 'e', 'b', 'i', 'v', 't']
['i', 'l', 'j', 'i', 'o', 'o', 'd', 'e', 'n']
['n', 'r', 'h-', 'd', 'h', 'a', 'k', 't', 'e']
['i', 't-', 'n-', 'a-', 'h', 'b', 's', 't', 't']
['i', 'e-', 'a-', 'p', 't', 's', 'n', 'e', 'w']
['m-', 't-', 'o-', 'n-', 'a', 'c', 's', 'l', 'u']
['o', 'n', 'i-', 'r', 'a', 'u', 'h', 'r', 'e']
['j', 'r', 'h', 'c', 'e', 't', 'r', 'n', 'o']
```

Figure 3.3: The letters on the first pathway have been marked with red dashes.

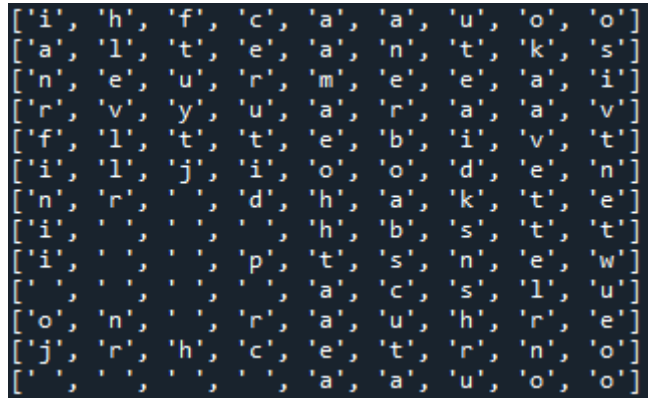


Figure 3.4: The dashed letters have now been removed entirely.

3: Apply “gravity” to each character, by iterating through each column in the grid and shifting empty spaces to the top (see Fig. 2.5). Then, repeat with rows to apply the same technique with “sideways gravity,” leaving a board with the specified characters removed and the remaining characters moved appropriately (see Fig. 2.6).

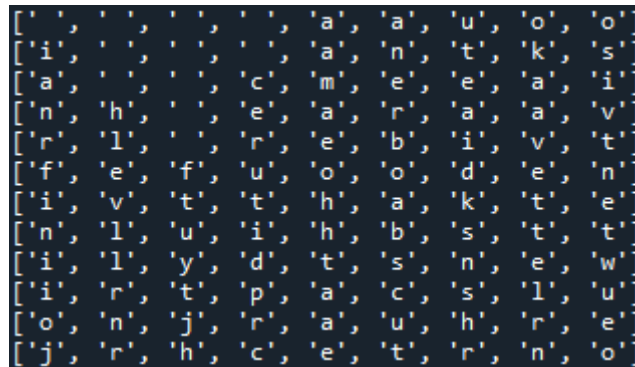


Figure 3.5: The board after applying downwards gravity.

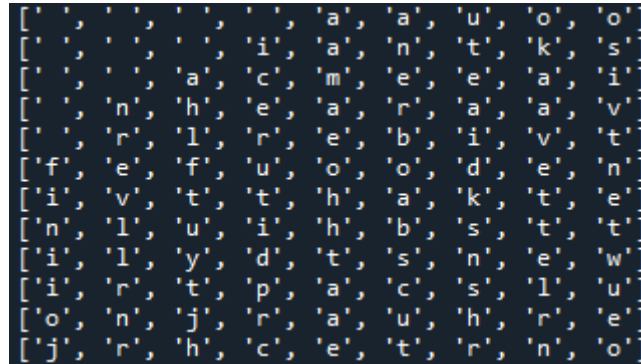


Figure 3.6: The board after applying sideways gravity.

4: Calculate the score of the word that's been removed, by looking up the score of each letter in the string of characters, and then multiplying the sum by the length of the string of characters. This can be added to a running score total.

These are the basic necessary functions required to build an algorithm that can find a solution to the game. Both of these functions have computing complexities linearly proportional to either the size of the grid or the length of the word and only have to be applied a one time for each word selected per turn throughout the solving process, so there is little need for optimisation. The bulk of the computing time, and also the vast majority of the solving process, lies in the next task: finding out what the best word to next remove from a board is.

## Chapter 4

# The first algorithm

### 4.1 The greedy solution

Every function covered so far has one specifically-defined goal. The next one, choosing a word to remove, is much the opposite. On an unsolved board, there are often thousands of unique words that are playable, and each word may show up multiple times on the board in different places. Choosing what is the optimal word to remove is where all the strategy lies, both for a human and for a computational algorithm. What letters are left on the board and what shape the remaining letters form differ vastly depending on the word removed, and can have serious consequences later on in the game. Removing a high-scoring word from the board may make finding further high-scoring words much more difficult later on, resulting in a lower overall score than if a lower-scoring word is chosen. This is very hard to predict, and has a few different root causes. For example, many long words use proportionally a lot of vowels, which leaves fewer on the board later. A more specific example is that a word might be close to but not quite adjacent to a suffix (like “-ing” for verbs and “s” or “-es” for plural nouns), which could be connected by removal of a different word, allowing for a higher score over 2 turns. But, what would happen if you always chose the highest-scoring available word on the board every time, without considering what remains? This is called a greedy algorithm, as the locally optimum solution is always taken (which in this case is considered as the word found with the highest word score).

This is also the first time we take advantage of an instance of a computer being superior to a human, as a human player can't check every word in the dictionary every time they wish to take a turn, while a computer, given a sufficiently optimal algorithm, is easily able to within a reasonable amount of time. Creating this algorithm is also a useful springboard to applying the later heuristics necessary to optimise the end score, as each word could have a "weighting" applied to it, changing the score to account for potential future score.

This makes clear the next task, creating an algorithm that will be applied more times than any other in the whole process: checking whether a given word is on the board. To find the highest-scoring word on the board, the whole dictionary has to be iterated through, each time checking whether the word exists on the board, and the highest-scoring word then has to be identified amongst them. This means that checking whether a word is on the board will be done thousands of times each turn, making it the bulk of computing time and the most primed for time optimisation. But, how can a computer check whether a word is on the board?

To find a working algorithm to do this, it is best to start by breaking down the thought processes applied by a human player. By playing the game myself, the way I naturally found myself checking for words is by stringing letters together that look like parts of words, and then affixing letters to the start and end until a word is created. This involves lots of "brain heuristics," which aren't well defined and can't realistically be translated into code. What "looks like" part of a word? Where do you progress from here to create a full word? These tasks are necessary for a human in order to eliminate the vast majority of non-words, but a computer doesn't necessarily have the same limitations in computing. Instead, I decided to incorporate stringing letters together in a more tangible form for an algorithm: a directed graph.

A given word must start with the first letter of the word, and then connects to the second letter, and so on until the last letter, which is connected to the penultimate letter. How can this be used to find a word on a board? One way this could be done is with a breadth-first search, which is

carried out as follows:

1: Find the first instance on the board of the first letter of the word being searched for (see Fig. 4.1).

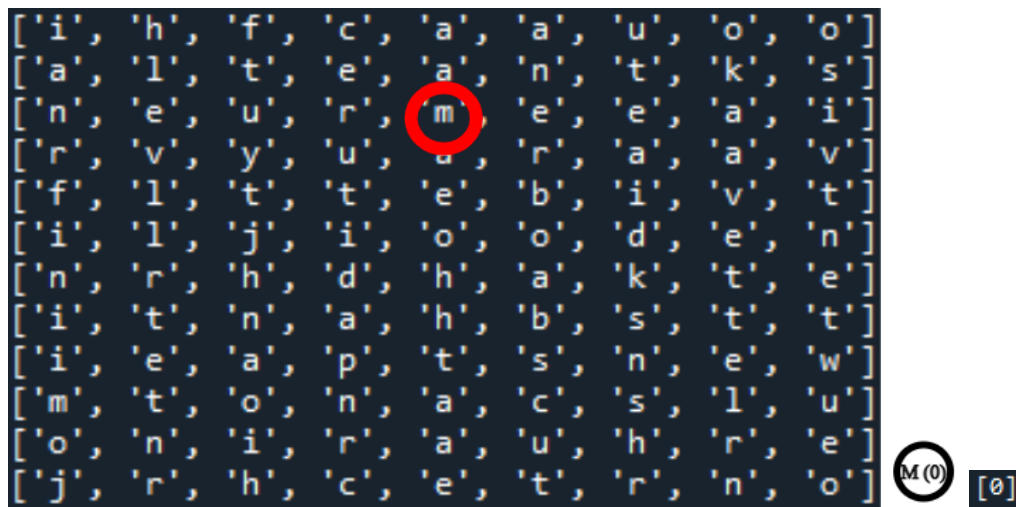


Figure 4.1: The first letter of the example word, “methanation,” has been encircled in red on the board. Also shown is the current directed graph and how it is stored in the Python code.

2: Search each letter surrounding this first letter, and if any of them happen to be the second letter of the word, note down the position of every occurrence of this letter. This is equivalent to finding a root of a directed graph, and finding all branches out of the root that connect to the wanted nodes (see Fig. 4.2).



Figure 4.2: The second letter (E) has been found in two locations adjacent to M, and have been added to the list of pathways. Note that each split creates a new stored route for each possible pathway.

3: Repeat this process for each new node, but for the next letter of the word. This will then have generated all possible paths from the first to the second letter, and then from all of those second letters to all possible third letters, which when used successively can find all possible paths from the root letter to the third letter of the word (see Fig. 4.3).



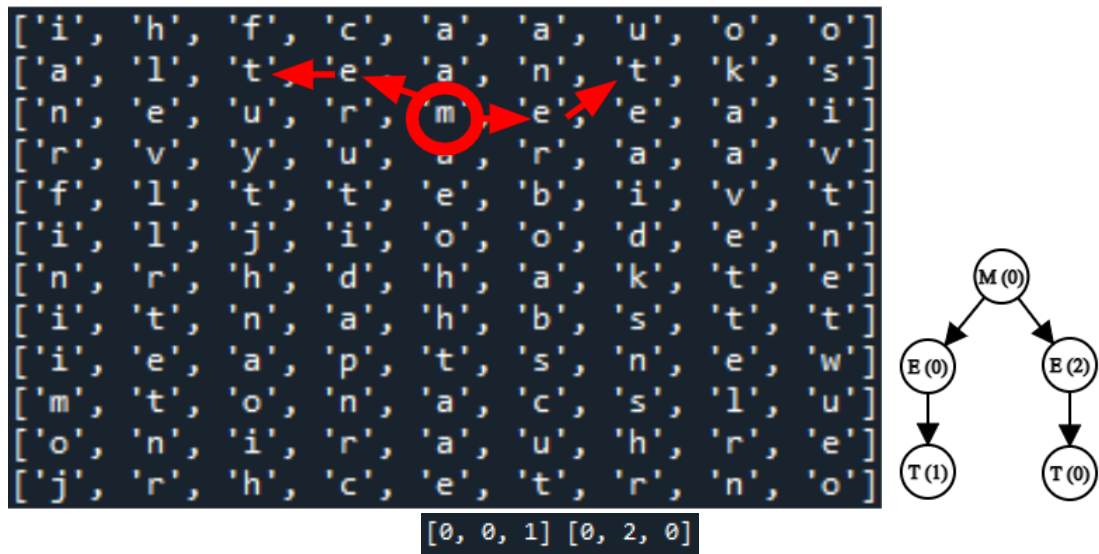


Figure 4.3: The third letter, T, has been found once adjacent to both Es. Since no new splits in the diagram are found, the number of routes stored remains the same.

4: Repeat step 3, iterating through to the end of the word, creating a directed graph containing all possible routes from the first letter to the last letter (see Fig. 4.4).

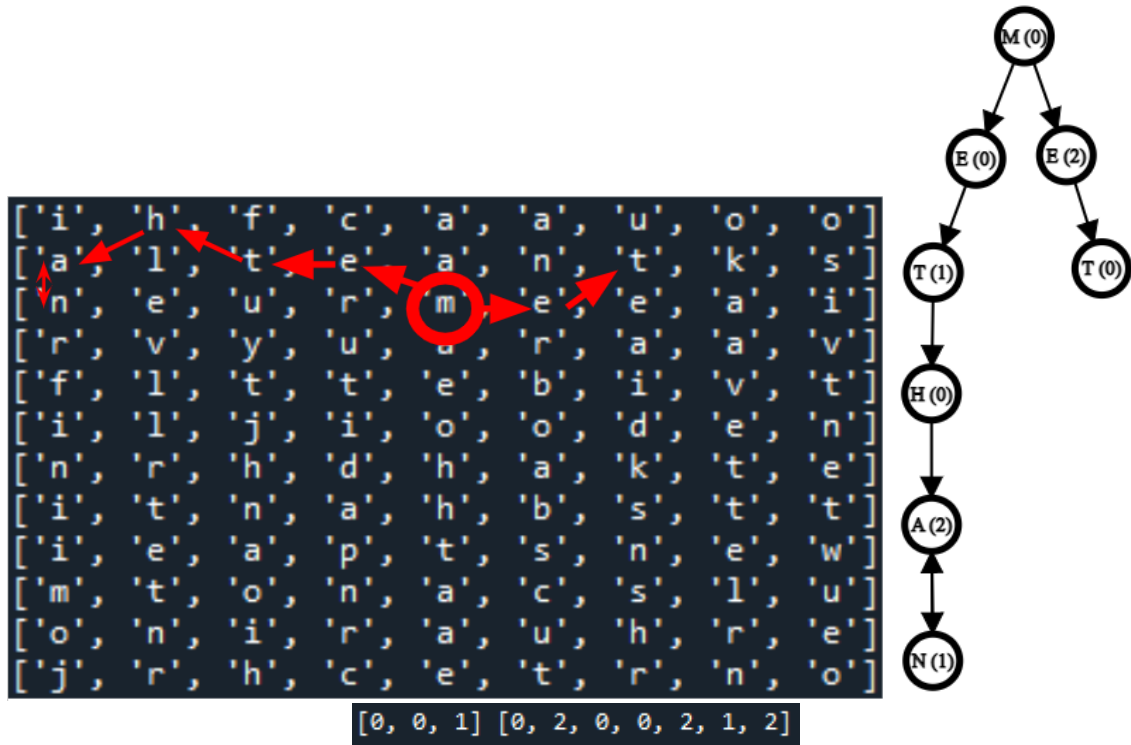


Figure 4.4: This is the finished directed graph for the first instance of M on the board. Note that it is not a simple graph, as the 7th letter in the 2nd route is the same location as the 5th letter on that route. Also note that no full path to the end of the word is found, so the first instance of M is a “dead end.” The next steps will use the graph created from the second instance of M found.

5: The directed graph will inevitably contain many branches which end partway through the word, as not every letter will necessarily be connected to the next one on the board. So, eliminate all pathways from the root which don’t end at the final letter. This can be done by deleting any stored pathway of length shorter than the number of connections between the letters in the word, as they cannot contain the full word within the pathway, leaving every pathway which ends at the final letter. Also eliminate all cyclical pathways (in the case of the board, this is any pathway that passes through a location on the board twice, as this is not allowed). You will be left with a list of all pathways from the first letter to the last, starting from the first instance of the first letter on the board (see Fig. 4.7). This list is very likely to be empty, as the chances of a letter connecting to the next is small, and the chances of every connection between letters in the word existing in connection

on the board is roughly proportional to the chance of one connection occurring raised to the power of the number of connections between letters in the word being searched. In fact, the chances of an arbitrary pathway of length  $n$  being an actual word gets vanishingly small for longer words (see Fig. 4.5):

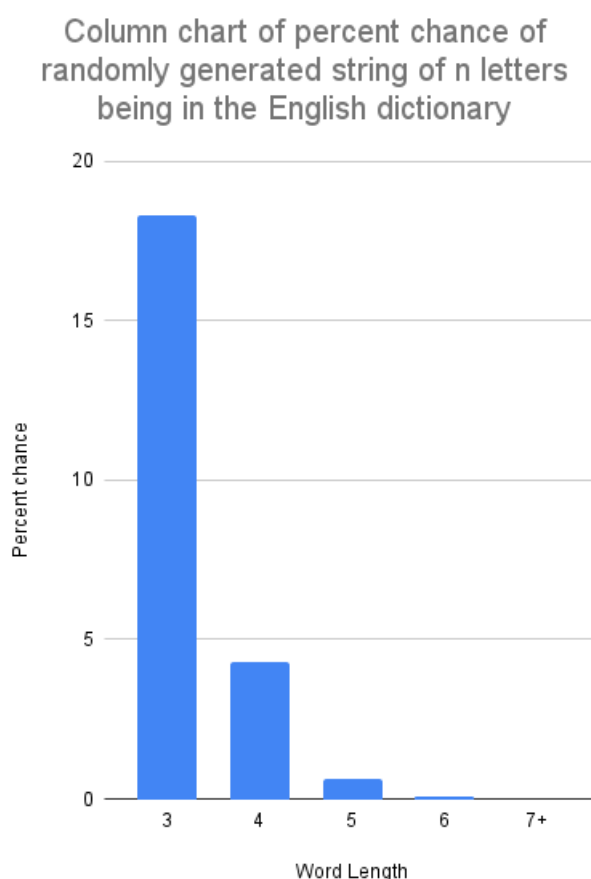


Figure 4.5: As the graph shows, an arbitrary string of length 7 or more letters is so unlikely to be an actual word, it doesn't even show up on the chart. Even though there are millions of possible different paths of length 7 and above on the grid, only a tiny proportion will end up being words.

Furthermore, there may be multiple occurrences of the first letter of the word in the grid (see Fig. 4.6).

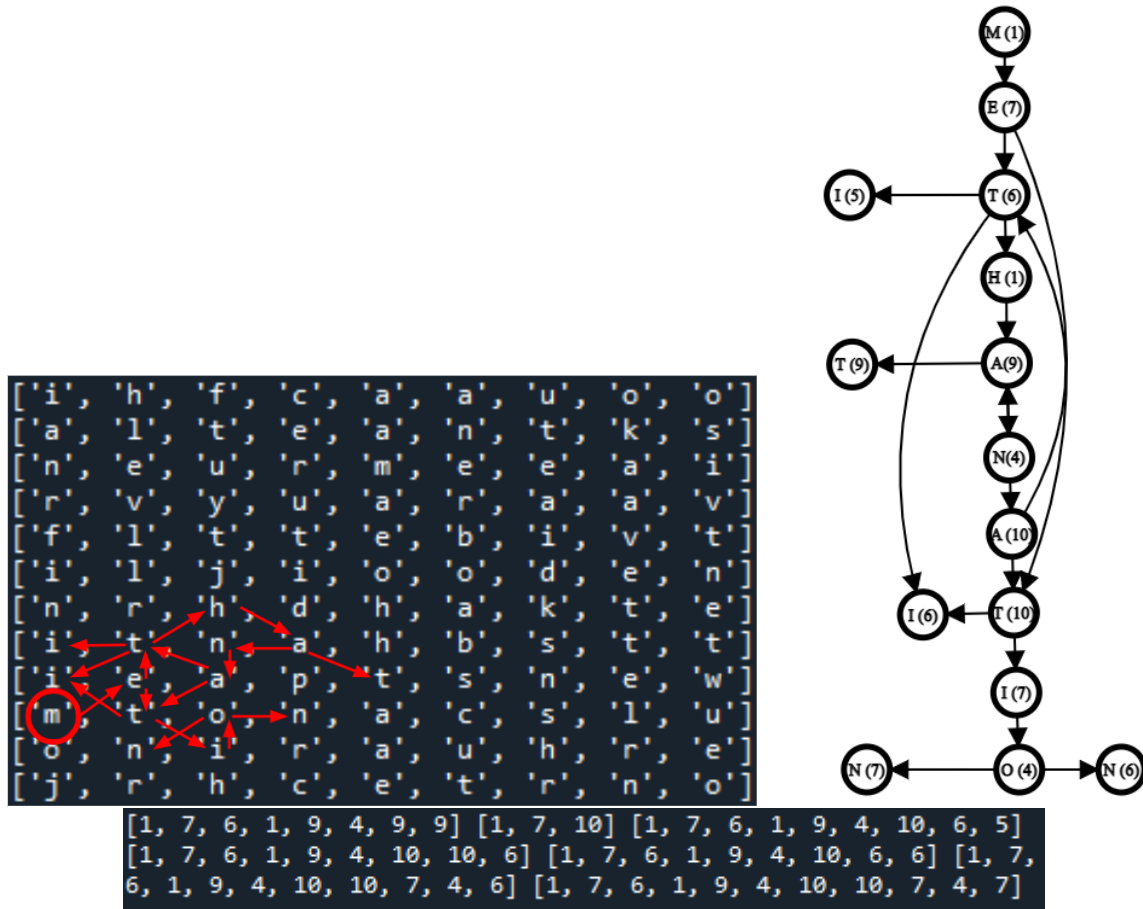


Figure 4.6: The full directed graph formed from the 2<sup>nd</sup> instance of M on the board. This contains two full pathways containing the whole word, but also contains five dead ends, three of which also contain certain locations on the board twice. These have to be removed in two stages, as there are routes that are incomplete but don't contain repeats but also routes that are complete but do contain repeats. It is also important to note that removing a different path for the word will result in a different grid on the next move.

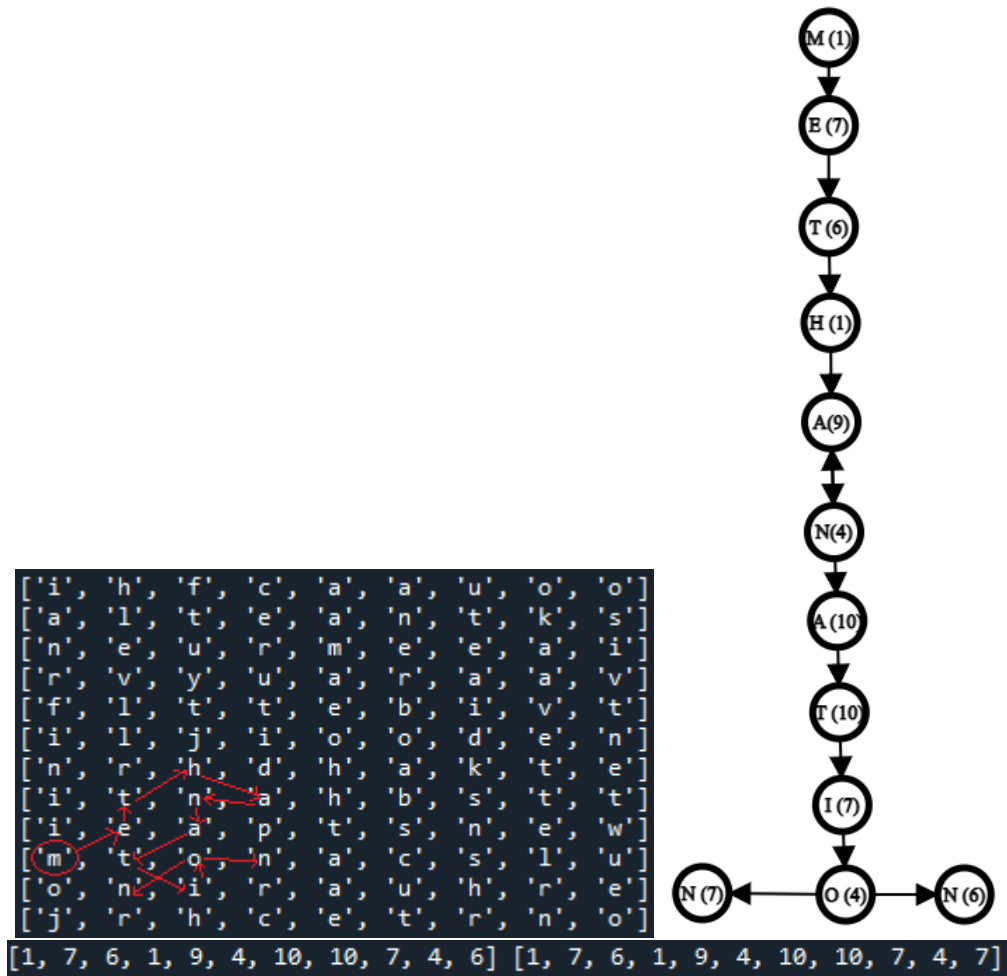
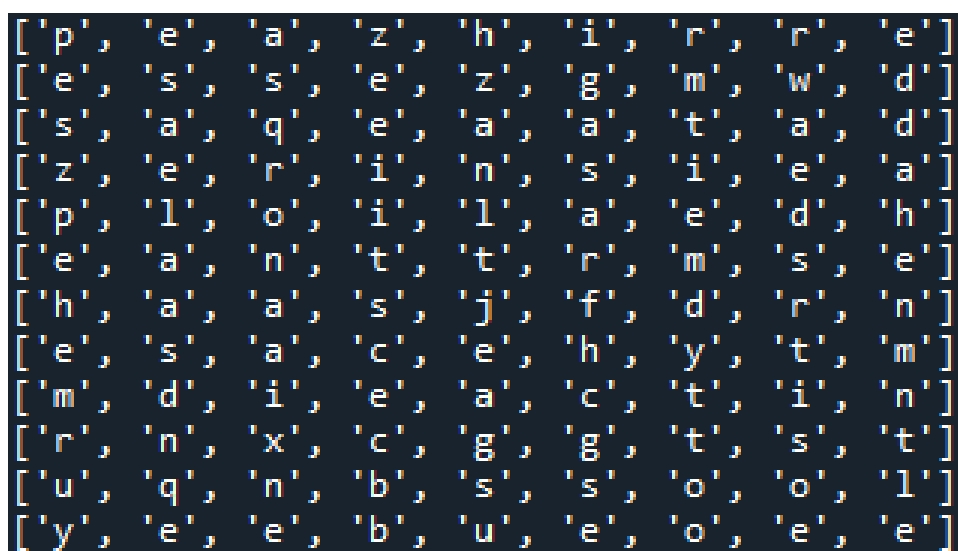


Figure 4.7: After running through all stored pathways and removing incomplete and repeating ones, you are left with every instance of the word “methanation” on the board, along with their pathways on the board.

6: Repeat steps 1 through 5 for every occurrence of the first letter within the grid. After this, every existing occurrence of the word being searched within the grid will have been located and stored as a pathway. Very conveniently, this pathway can be stored in the same format as required to remove a word from the board, allowing for easy removal from the board, if that particular pathway is considered the optimal move.

This whole process, although rather lengthy to do by hand, can be done computationally in a under a millisecond. This is essential, as it has to be done once for every word in the dictionary to find every possible word on the board. The next step of the greedy algorithm is to use the function that checks the score of a word, and assign a score to every word that exists on the board, and then order each word by decreasing score. The first word on this list is the local optimum move as its score is the highest, and can be removed from the board, with the score being added to the running total. This is the “greedy” choice, as it makes no attempt to look further ahead or strategize in any way. If you repeat this process on each subsequent board until the algorithm cannot find any word on the board, you will have completed a greedy algorithm method of solving a game of *Word Soup*! The following is an example runthrough of an arbitrary board (see Table 4.1):



'p'	'e'	'a'	'z'	'h'	'i'	'r'	'r'	'e'
'e'	's'	's'	'e'	'z'	'g'	'm'	'w'	'd'
's'	'a'	'q'	'e'	'a'	'a'	't'	'a'	'd'
'z'	'e'	'r'	'i'	'n'	's'	'i'	'e'	'a'
'p'	'l'	'o'	'i'	'l'	'a'	'e'	'd'	'h'
'e'	'a'	'n'	't'	't'	'r'	'm'	's'	'e'
'h'	'a'	'a'	's'	'j'	'f'	'd'	'r'	'n'
'e'	's'	'a'	'c'	'e'	'h'	'y'	't'	'm'
'm'	'd'	'i'	'e'	'a'	'c'	't'	'i'	'n'
'r'	'n'	'x'	'c'	'g'	'g'	't'	's'	't'
'u'	'q'	'n'	'b'	's'	's'	'o'	'o'	'l'
'y'	'e'	'e'	'b'	'u'	'e'	'o'	'e'	'e'

Figure 4.8: The starting board.

Word:	Score:
sensationless	260
mediatize	198
hazardry	216
energize	168
heartpeas	135
cinque	120
snitches	120
faggots	105
leptomes	120
excambed	168
jowar	80
lear	40
dah	27
Total:	1757

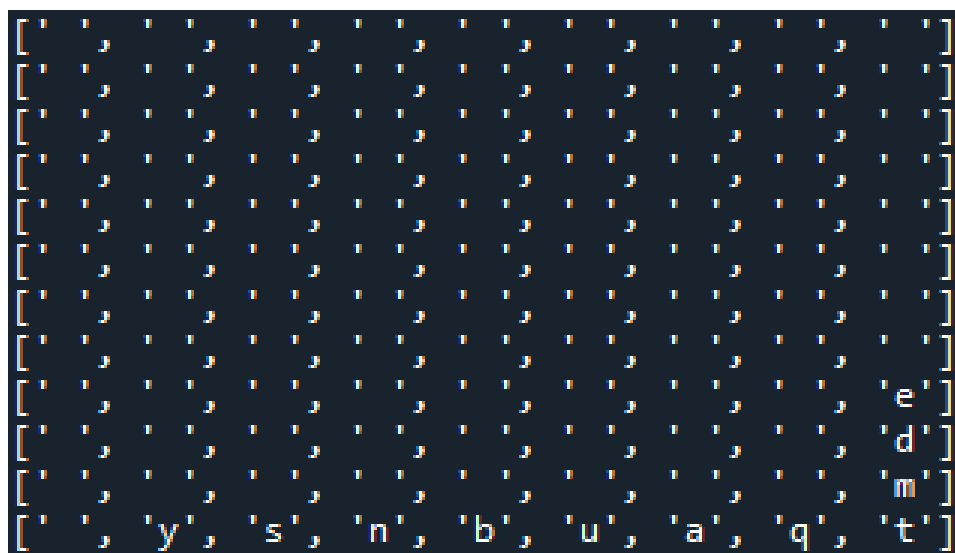
Table 4.1: An example game of *Word Soup* using the greedy algorithm.

Figure 4.9: The finished board, with no words of length 3 or longer left. Note that the letters all lie on the perimeter. This observation suggests that a heuristic attempting to prevent this could be employed later, to prevent such a scenario occurring.

## 4.2 Improving the algorithm's efficiency

Running this process with a loaded list of all words in the *Collins* Scrabble dictionary does solve the game, but still is a lengthy process (see Fig. 4.10) that has plenty of room for optimisation. This algorithm has a complexity proportional to the square of the length of the word, so it is necessary to minimize the amount of times this algorithm is executed.

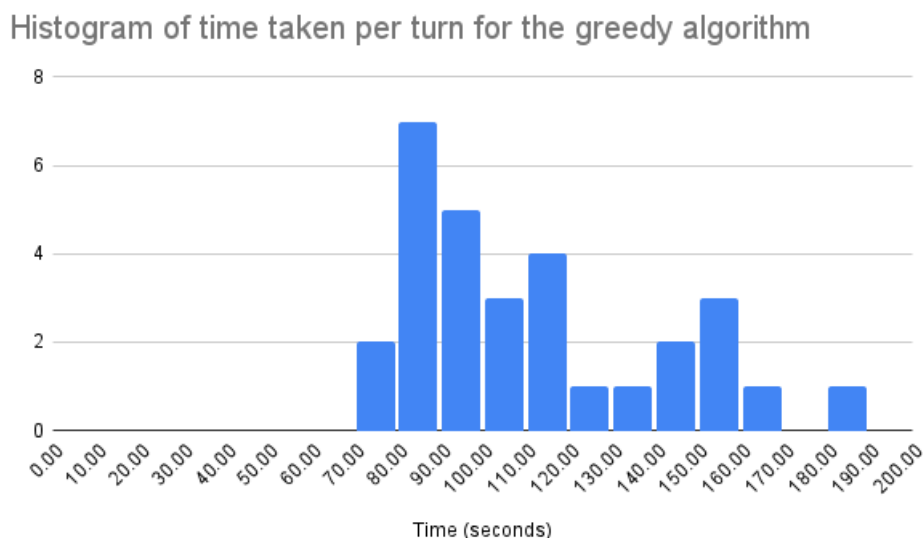


Figure 4.10: A histogram of time taken per turn with the greedy algorithm with no improvements made to efficiency. The  $y$ -axis is the number of recorded instances of a turn taking the specified time interval. Due to the sheer length of time it took for each turn, I only ran 2 games for this algorithm.

The first method of doing this involves a simple reordering of the greedy algorithm: Instead of iterating through the whole dictionary and sorting every word found to obtain the highest-scoring word on the board, simply sort the whole dictionary by score at the very beginning, and then remove the first word found within the dictionary, which will be by definition the highest-scoring word on the board (see Table 4.2). This, considerably reduces the amount of times the breadth-first search has to be done (see Fig. 4.11), while only adding on the amount of time it takes to sort the dictionary (which can then be saved as a text file, so doesn't need to be more every game). This makes



very significant reductions at the beginning of the game, when high-value words are more common. The algorithm only needs to search through a few words before finding one which is on the board. Towards the end of each game, the time reductions are much less significant, as shorter words of lower value are present, and the algorithm has to search through most of the dictionary to find a word that exists on the board.

Word rank	Alphabetized	Score	Sorted by points	Score
1	aah	18	quizzifications	765
2	aahed	45	tranquilizingly	765
3	aahing	91	zygophyllaceous	735
4	aahs	28	quinquennially	728
5	aal	21	oxyphenbutazone	720
...	...	...	...	...
279,425	zythums	175	tit	9
279,426	zyzzyva	201	toe	9
279,427	zyzzyvas	352	too	9
279,428	zzz	90	tor	9
279,429	zzzs	124	tot	9

Table 4.2: A table of the five first and last words of the alphabetized and point sorted dictionaries, and their point values. Many words tie in point value, and are sub-sorted alphabetically.

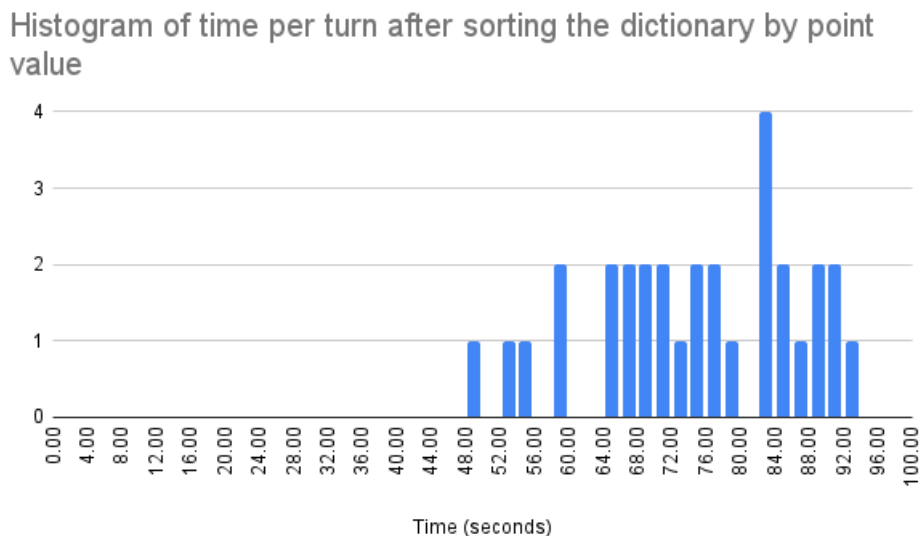


Figure 4.11: A histogram of time taken per turn with a sorted dictionary. The  $y$ -axis is the number of recorded instances of a turn taking the specified time interval over 2 games. This is already nearly twice as quick as the first algorithm!

Another optimisation may also seem obvious to a human: if a letter in the word doesn't exist on the board, then the word cannot exist on the board. So, by searching through the board and noting any letters not present, you can skip any words in the dictionary containing any of those letters altogether. For example, if there was no letter "K" on the board, you could safely refrain from checking the 8.06% (see Table 4.3) of letters containing the letter "K" saving time. This does rid some possibilities, but in practice doesn't save much time at the beginning of the game, as almost every letter is present on the board for a significant proportion of the game, and the letters most likely not on the board are the least frequent letters, so oftentimes only a few words are skipped in the dictionary. What does remove a significant number of words, though, is pairs of letters not on the board. Typically there are a much higher proportion of pairs of letters absent from the board, and by checking a word for any pairs absent from the board is much more likely to eliminate the word. This also removes the need to check for missing single letters, as by definition a missing single letter won't show up in a pair of letters.

Letter	Occurrence in dictionary (%)	Rank	Pair of letters	Occurrence in dictionary (%)
E	69.43	1	ES	19.20
S	61.71	2	IN	17.82
I	60.00	3	ER	17.36
A	54.95	4	TI	12.41
R	52.25	5	TE	10.66
N	48.52	6	NG	10.65
T	47.60	7	AT	10.60
O	46.43	8	IS	10.55
L	39.32	9	RE	10.31
C	31.47	10	ON	10.25
U	27.19	11	ST	9.41
D	27.03	12	ED	9.40
P	24.14	...		
M	23.58	603	QH	0.00072
G	22.84	604	QF	0.00072
H	20.87	605	QE	0.00072
B	15.57	606	PZ	0.00072
Y	14.07	607	JT	0.00072
F	9.68	608	JP	0.00072
V	8.09	609	JD	0.00072
K	8.06	610	GV	0.00072
W	6.71	611	FV	0.00072
Z	4.08	612	VT	0.00036
X	2.57	613	VH	0.00036
Q	1.52	614	RX	0.00036
J	1.50	615-676	—	0

Table 4.3: A sorted list of all letters by frequency in the English dictionary, along with the most and least frequent pairs of letters.

This, in practice, saves checking the majority of words, cutting the amount of times you have to check words drastically, while only adding a linear-complexity check to the start of each turn. This makes the greedy algorithm work in a fraction of the time it works in without the optimisation (see Fig. 4.12). Unlike the first optimisation check, this is the most effective when the board is small, as more pairs of letters are missing, meaning that the proportion of the dictionary considered is much, much smaller.

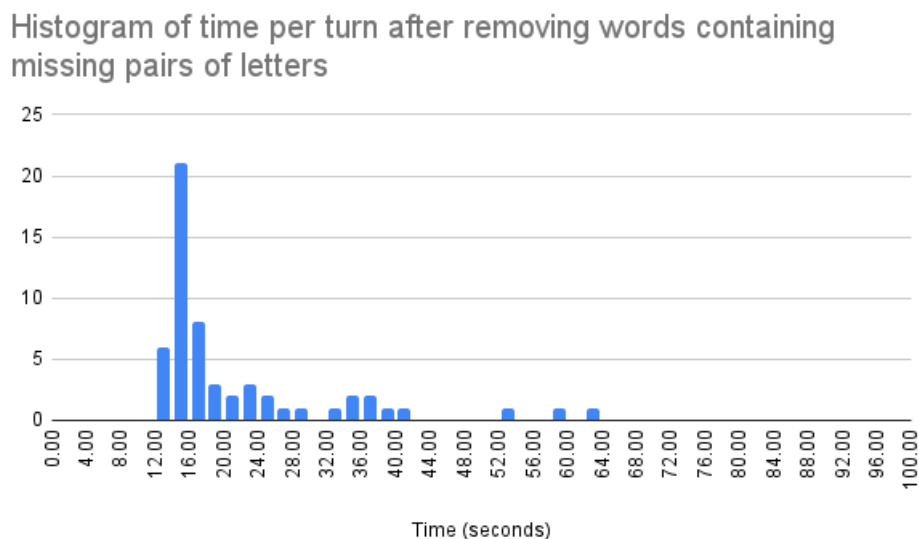


Figure 4.12: A histogram of time taken per turn after skipping over words containing missing pairs of letters. The  $y$ -axis is the number of recorded instances of a turn taking the specified time interval over 3 games. This is a bigger improvement than the sorted dictionary approach, but still takes a rather long time for the first few goes of each game.

A combination of these two methods reduces redundant calculations in the algorithm both at the beginning and the end of the game, saving time throughout the whole game (see Fig. 3.13). Although it seems counter-intuitive, the algorithm is now slightly faster at the beginning of the game, when the board is full. But, if you ran the greedy algorithm on an arbitrarily large board, you could expect nearly every word to show up, meaning that the sorted dictionary would almost instantly find a word.

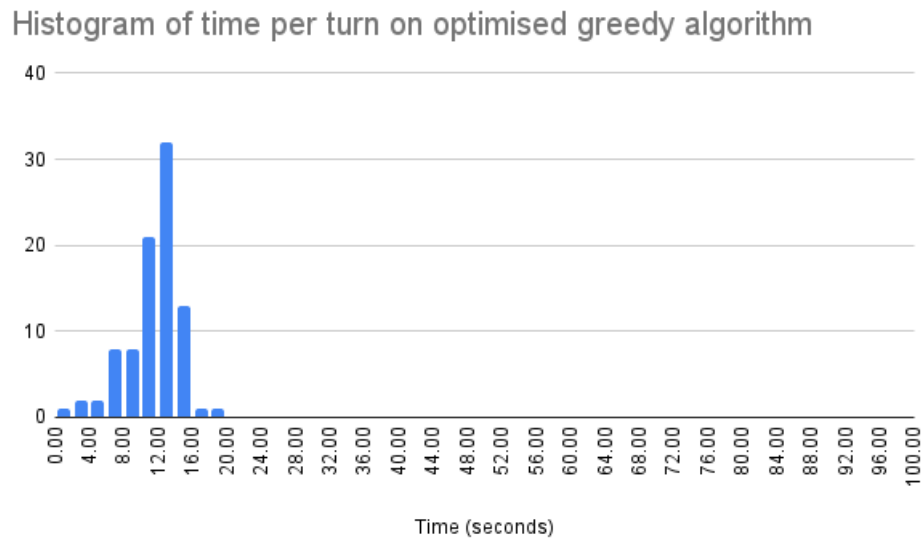


Figure 4.13: A histogram of time taken per turn with the optimised greedy algorithm. The  $y$ -axis is the number of recorded instances of a turn taking the specified time interval over 5 games. Using both optimisation techniques, the algorithm is about 10 times faster than the first version of the greedy algorithm!

### 4.3 First set of Results

Now that we have our first algorithm that can complete a game, we can have our first comparison with human players. Running the greedy algorithm on the same 10 boards the expert player solved gives us the following results (see Table 4.4):

Game no.	Amateur	Expert	Bonus?	Greedy algorithm	Beat expert?	With bonus?
1	1038	1460	500	1780	yes	no
2	1062	1410	500	1752	yes	no
3	1090	1785	500	1759	no	no
4	1120	1512	500	1791	yes	no
5	1157	1594	0	1542	no	no
6	1187	1450	0	1748	yes	yes
7	1225	1626	500	1494	no	no
8	1244	1534	500	1843	yes	no
9	1277	1342	500	1688	yes	no
10	1309	1774	500	1851	yes	no
Average:	1170.9	1548.7	1948.7	1724.8		

Table 4.4: The results table for 10 games played by the greedy algorithm, compared with the amateur and expert player. Note that the 10 games played by the expert and the greedy algorithm are identical.

This is a very promising start! The algorithm managed to achieve a higher score than the amateur player's best score on even its worst game (the amateur's best score was 1,309 points compared to the greedy algorithm's worst performance of 1,494 points), so I can confidently say that the greedy algorithm plays above the level of most human players.

The greedy algorithm can also, on average, score 176 more points on word scores than the expert human player, and outscored the expert player 7 times out of 10 on word score alone. Where the algorithm falls short, as with the amateur player, is with clearing the board and gaining an extra 500 points. The greedy algorithm, fitting with its name, fails to take into account future moves, and as a result leaves itself with a poorly-shaped board with difficult to use letters (an example would be Fig. 3.9). As a result, the expert player still wins 9 out of the 10 games. From this, it is clear that focusing on clearing the board is crucial in improving the greedy algorithm. This would also need to be done without sacrificing the word score we get with the greedy algorithm, as the average word score achieved is currently better than even the expert players.

## 4.4 Next Steps

The greedy algorithm, along with the improvements, can execute within a reasonable amount of time and produce a good score. However, this is not necessarily the optimal solution. The theoretically optimal solution would require finding every possible first move, and then finding every possible second move by checking every possible second word playable in every different remaining board, and so on until every possible route towards an endgame has been exhausted. Take this example: if there are 1,000 words playable on the first turn (from inspection, most starting boards yield about 1,500 and 2,000 starting words), and another 1,000 on the second and third turns, the algorithm would already have to check for the best word on 1,000,000 different boards with 2 words removed just to find the optimal 3-move solution. Furthermore, at each move the board can be shifted to the left and right. This has a complexity of order roughly proportional to the number of words available on a typical board to the power of the typical number of moves in a game, all multiplied by two to the power of the number of moves after the first move (to account for the shifting the board to each side). It is very clear that the amount of computing power required to find the one (or joint, if there are multiple solutions which all score the same) optimal solution to a game of *Word Soup* would take dozens of orders of magnitude more time to compute than is feasible in the context of solving a word game (or any context, for that matter).

So far, we have a greedy algorithm that can solve a game of *Word Soup* consistently better than an amateur player and beat the word score of an expert player more often than not, but one that can't clear the board well, leaving a jumble of obscure letters in a poorly-connected arrowhead at the lower-right hand of the board. It is also clear that just improving efficiency will not be sufficient to find an optimal solution. Instead, it is necessary to dispense with the idea of finding the global optimum solution, and introduce some heuristics into the algorithm in order to attempt to clear the board.

## Chapter 5

# Adding heuristics

### 5.1 How humans play *Word Soup*

Analyzing how humans play *Word Soup* may help gain useful insights into how the current algorithm differs in play style from expert players, and may help find methods for improving the algorithm. This includes analyzing the spread of word scores and where points are mostly scored. In Figure 5.1 we show the distribution of expert human scores over 10 sample games, for which full statistics were available. The expert player scored primarily low-scoring words, with a few high-scoring words and nearly invariably one very-high scoring word (above 300 points) per game.



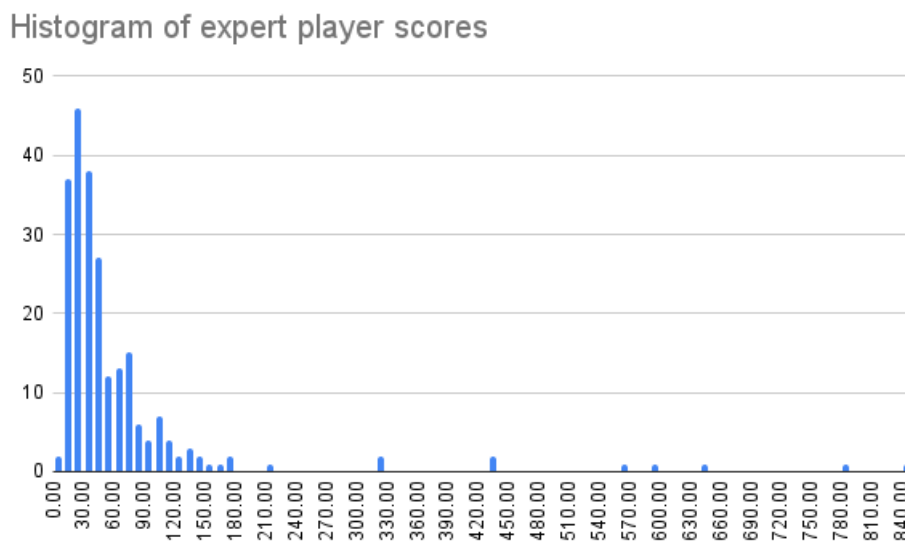


Figure 5.1: A histogram of word scores made by an expert player over 10 games. The  $y$ -axis is the count of word with scores in that range.

In Figure 5.2, we show the distribution greedy algorithm scores for a sample 46 games for which full statistics were available. The distribution is surprisingly even, with a high-scoring word of 200 points being just as likely to occur as a relatively low-scoring word of 40 points. The tail at the upper end is much shorter and thinner in comparison, showing the greedy algorithm's lack of ability to find very high-scoring words.

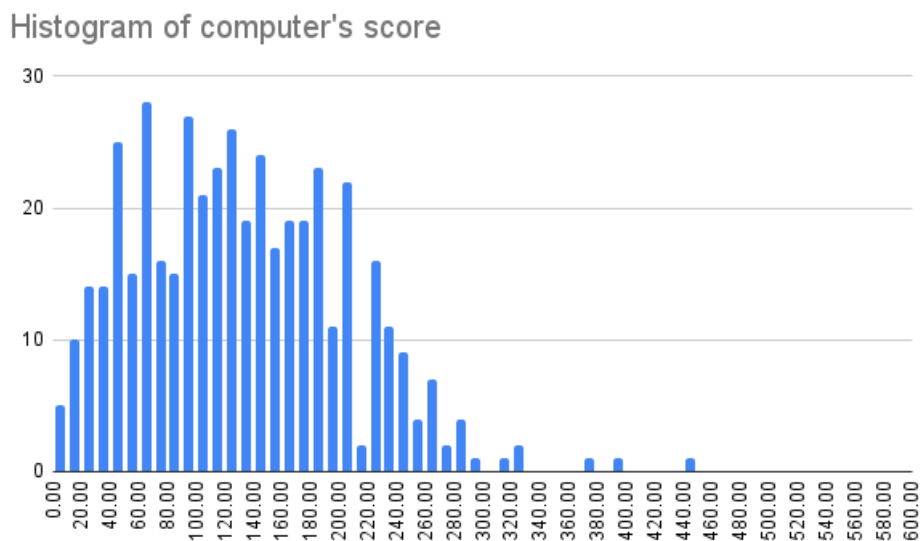


Figure 5.2: A histogram of word scores made by the greedy algorithm over 46 games. The  $y$ -axis is the count of word with scores in that range.

These histograms make very clear the main difference in play style from humans and the greedy algorithm. While both the expert player and the greedy algorithm score more lower-scoring words than higher-scoring words, the expert player scored proportionally many more very low-scoring words, and the greedy algorithm dominates in the 100-300 point range. Taken at face value, this would seem to suggest that the greedy algorithm is the better player. But, this is ignoring the few outlying high scores that the expert played. While the greedy algorithm seldom scores over 300 points, the expert player can consistently score one or more words with a point value above 300, and up to 600 points. This is where strategy comes into play. An expert player, able to think ahead, will sacrifice points in the short term in order to line up a very high-scoring word in a few rounds in the future. The greedy algorithm by definition will always opt to score more points on the current turn, meaning that very high-scoring words are unlikely to occur by happenstance. In other words, the expert player has deliberately reached for a subset of the solution space which is inaccessible to the greedy algorithm: extremely long, very high-scoring words.

This alone is only a partial explanation of the abundance of low-scoring words on behalf of the

expert player. The second half of the explanation is made clear when you analyze the points scored at different stages of the game (see Fig. 5.3 and 5.4).



Figure 5.3: A scatter plot of word scores over percentage of letters used, when done by an expert human player.

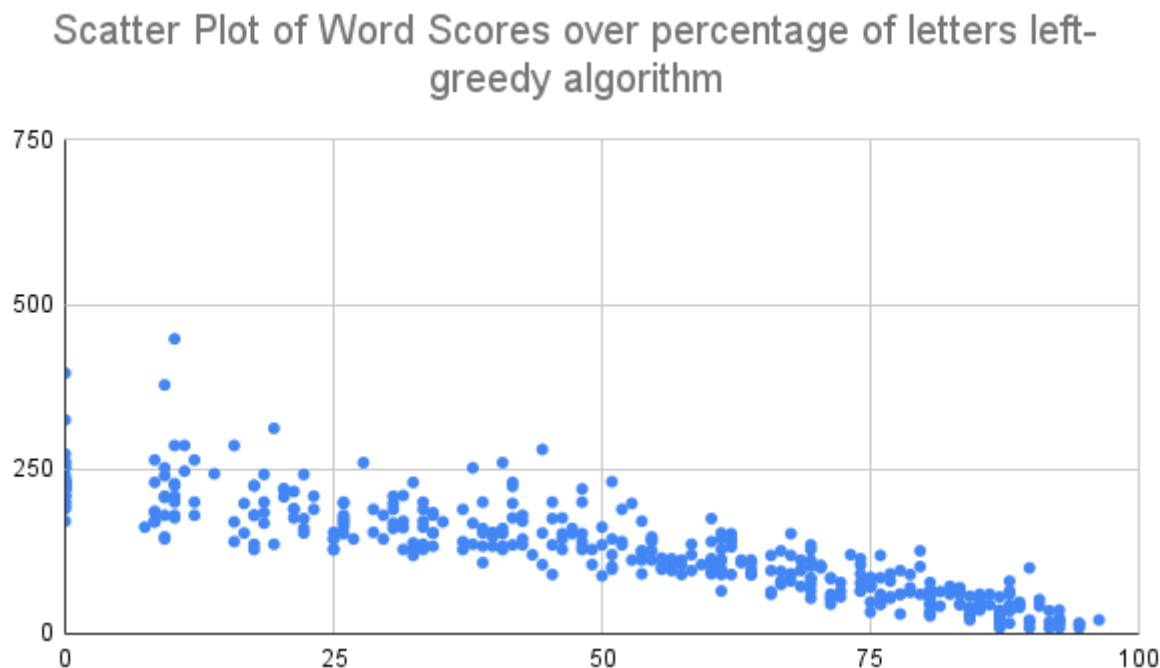


Figure 5.4: A scatter plot of word scores over percentage of letters used, when done by the greedy algorithm.

As expected, the greedy algorithm performs best at the beginning of each game, when the most letters are available and more words are likely to occur. It is also observable that the longest words are found at the beginning of the game, as the data points are further spread out (although as differing word lengths from different game cause turns to overlap, it is more difficult to tell how far the trend continues).

Like the greedy algorithm, the expert player tends to score more points early on, as the board has more letters and therefore more opportunities to score. But, the second big difference is made clear at the endgame. When roughly one quarter of letters are left, the expert player stops finding high-scoring words altogether, and instead focuses on clearing the board. No amount of points gained from words in the last part of the game could ever compare to the 500 scored by clearing the board. Again, the greedy algorithm has no concept of future moves, so will only ever clear the

board if it gets lucky.

This rather clearly shows where the biggest improvements in score can be made: sacrificing a few letters in order to execute a very high-scoring word, and managing to clear the board. But, how feasible are these to execute computationally? Being able to bring together all the letters for a very high-scoring word is almost intrinsically a human ability; it requires spotting patterns, and aiming towards achieving a goal with no tangible metric of the best way to achieve it, both of which are abilities very difficult to describe computationally. On the other hand, clearing the board is a more tangible goal for an algorithm. If attention is paid to what letters remain on the board and what shape the board takes, much less luck is involved in clearing the board.

## 5.2 Two-move greedy

The most direct opportunity for scoring more points on a game of *Word Soup* is to search more than one move ahead. Instead of simply choosing the best word given the existing board, you can instead perform a depth-first search for words on future boards. This works as follows:

- 1: Starting with the current board, find the highest scoring word available by applying the greedy algorithm. Create a new, imaginary board with this word removed from the board, and store its point value.
- 2: On this imaginary board, find the best available word and add this to the point value gained from the previous board.
- 3: Move back to the current board, and continue searching down the list of words until you find the second highest scoring word. Replace the previous imaginary board with one of the current board with the second best word removed and store a variable of its point value.
- 4: Repeat step 2 with the new imaginary board. If the sum total of the second best word on

the current board and the best word on the new imaginary board is higher than the total found in step 2, replace the value found in step 2 with this new, higher total.

5: Repeat step 3 and 4 for as many turns as is feasible within a reasonable computing time, or until all possible combinations of 2 words have been checked.

6. Remove the highest-scoring combination of two words, which is guaranteed to have a combined score higher than or equal to two iterations of the one-step greedy algorithm. This is because the first two steps perform the same actions as iterating the one-step greedy algorithm twice, and choosing a different pair of words is only performed if they score a higher total.

Although this method requires more computing time than the one-step greedy algorithm, you can restrict how many words it checks from the current board to limit its computing time to a linear multiple of the one-step greedy algorithm (a polynomial of order 1). This does mean that not every combination of two words is checked, but will search the vast majority of pairs of words most likely to score the highest. It should also be noted that although this will with 100% certainty produce a higher or equal score over two moves than the one-move greedy algorithm, this doesn't necessarily guarantee it will find a better score over the whole game. This is because a better combination of two moves will leave a different set of letters left, which may in turn have a worse scoring potential later on in the game. This makes it worth considering generalising 2-move greedy to  $x$ -move greedy, which would search for the best combination of  $x$  words in  $x$  moves.

### 5.3 $X$ -move greedy

It seems that the next sensible step may be to try a three-move greedy algorithm, or perhaps four. This can be done by altering the two-move algorithm to search  $x$ -layers deep when performing the depth-first search algorithm, using an array of imaginary boards instead of just one, and carefully tracking through the dictionary at each layer to ensure every combination of words is checked. This could theoretically be extrapolated until the board is empty, but again the amount of computing

power becomes an issue. Because you only have to search for the best word on the lowest layer of the tree you're searching (as you're not searching any further down, the second-best word doesn't present any immediate advantages as far as this iteration of the algorithm is concerned), two-move greedy will only takes a linear multiple of one move-greedy, but three-move greedy will require a polynomial of degree 2 times longer than one-move greedy to perform. With thousands of words to search through for each step of the algorithm early on in the game, this is entirely infeasible without many restrictions. Without restrictions, assuming a mean time of 10 seconds to search for 1 word and approximately 1,000 words on the board towards the beginning of the game, searching 3 moves deep would take about 116 days.

One such restriction includes searching through a reduced dictionary size. This may mean excluding some words containing certain letters and not searching for long words. This presents a high likelihood of missing out on the highest-scoring combination of words, but is much, much quicker in finding a solution. This makes the algorithm potentially less effective than one-move greedy early on in the game, but becomes much more useful on small boards, where long words are nigh-on nonexistent and several letters are missing altogether.

Here is an example of a board in which  $x$ -move clears the board, where the greedy algorithm would instead fall short (see Fig. 5.5):

```
['e', 'r', 'f']
['t', 'y', 'i']
['a', 'l', 'i']
[[1, 3990], [0, 0], [0, 0]] 65 ['treyf', '', '']
[[4, 11455], [1, 16554], [0, 0]] 80 ['alif', 'tyer', '']
[[21, 16838], [5, 19018], [0, 19896]] 563 ['fil', 'tay', 'ire']
['e', 'r', 'f']
['t', 'y', 'i']
['a', 'l', 'i']
```

Figure 5.5: An example of  $x$ -move greedy solving a 3 by 3 board.

In this example, the board was cleared by choosing the 21st highest-scoring word on the first move, followed by the 5th highest-scoring word on the second move, followed by clearing the board. The

first dead-end found in the depth-first search is identical to running the greedy algorithm, and only found one word, leaving four letters on the board. Then, another dead-end was found with a higher total score, which left one letter on the board and scored more total points (this is coincidentally what a 2-move greedy algorithm would have found, showing an example of its improved performance to greedy in some cases). Finally, a dead-end was found which used all 9 letters on the board over 3 moves, which scored fewer points in word scores but achieved the 500-point bonus, vastly outscoring the other two solutions. This makes it clear that  $x$ -move greedy shows real promise in clearing the board when only a few letters remain. Unfortunately, the  $x$ -move greedy code failed to execute most of the time so I was unable to implement it in any further algorithms.

## 5.4 More heuristics

To consider which heuristics to use, it is useful to first consider a few properties of how points are scored, and where they could be scored. To do this, I decided to analyze the results of several runs of the greedy algorithm.

The most glaring result after dozens of simulated games is that the game always ended before all the letters on the board had been used. As a result, the 500 point bonus was almost never applied. It follows that if there was a different algorithm that could reliably clear the whole board and sacrifice fewer than 500 points during the game in comparison to the greedy algorithm, it would be more optimal.

Another method that could be useful is working backwards; by backtracking from a “dead-end” solution and trying all alternative pathways, working backwards until a solution removing all letters is found (which would be similar in result to using an  $x$ -move deep algorithm when there are only a few letters left).

The picture is also made more complete by analyzing the letters left in the final stages in a typical game, and their locations. The remaining letters are laden with cumbersome consonants like “V” and “Q,” which occur in very few short words, and form an arrowhead shape towards the bottom-right corner. Short words disproportionately contain fewer common letters and more even vowel to



letter ratios, and a reduction in the average number of letters that the average letter is adjacent to vastly decreases the amount of pathways that can be forged into words.

This suggests that the following heuristics may be worth paying attention to:

- 1: Preferring words that keep the vowel to letter ratio as close to ideal as possible.
- 2: Valuing words that use uncommon letters more highly than words composed entirely of common letters, even more so than the scoring system initially does.
- 3: Emphasizing using words that keep the remaining letters in a “good shape.” One metric that could quantify this is the average number of letters surrounding each letter.

Another property that repeated games show is that the majority of points scored from words are scored towards the beginning of the game. For an algorithm that doesn’t look several moves ahead, a larger board will generally yield more words, and words with higher word scores (see Fig. 5.6).

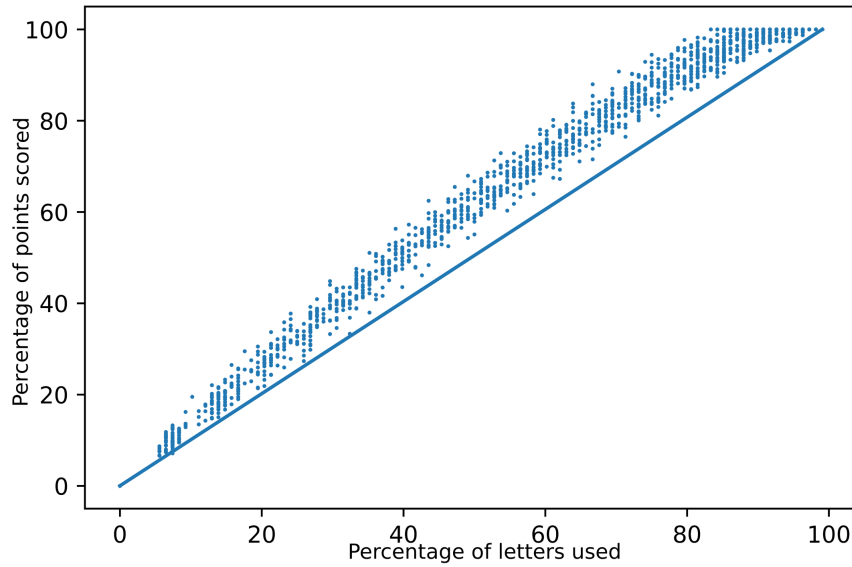


Figure 5.6: A graph showing the percentage of points scored vs a game over the percentage of letters used in the game over 71 simulations, as well as the line  $y = x$ . It is clear that the score per letter towards the beginning of the game is higher than at the end, indicating that towards the end of the game, scoring points may be less important.

Not only are half of the points in the greedy algorithm’s games scored with the first 40% of letters, but there are, on average, 24 letters of the original 108 left on the board when the greedy algorithm has equalled the total word score of the expert player. This presents an idea: perhaps focusing on locally optimal words is a useful method at the beginning of the game, but preserving the grid and attempting to clear the whole board is more important towards the end, implying a “change in tactics” at a crossover point.

If the algorithm can typically score all of the points from words it needs plus extra, it gives us room to lose a few points of the highest-scoring words in order to allow for a more usable board later on.

## 5.5 Valuing uncommon letters

In order to figure out which letters are the least common, I ran a letter frequency analysis on a dictionary of 3 to 5 letter words. This is because those are the words that are most likely to exist on a small board of fewer than 25 letters. Here are the results (see Table 5.1):

Letter	Frequency in dictionary of 3-5 letter words
A	9.35%
B	2.59%
C	2.99%
D	3.79%
E	10.02%
F	1.87%
G	2.66%
H	2.78%
I	5.80%
J	0.52%
K	2.48%
L	5.08%
M	3.14%
N	4.49%
O	7.17%
P	3.23%
Q	0.16%
R	6.06%
S	9.62%
T	5.04%
U	4.01%
V	1.08%
W	1.79%
X	0.46%
Y	3.10%
Z	0.72%

Table 5.1: A table showing the frequency of each letter in all 3-5 letter words the English dictionary.

This is rather similar to most letters' frequency in English prose, with some exceptions. "T" and "H" are lower down on the list as they tend to show up mostly in common words, like "the." But, as this is a dictionary frequency analysis and not an analysis of wider literature, so all words are treated with equal weight. On the other hand, "K" and "S," which show up in less common words more often (as plural words are less common than their singular counterparts, and "K" isn't in any

of the 50 most common words).

Next, I re-weighted the point values of each letter in order to heavily favour the letters that show up the least. To do this, I took each word's original word point value and multiplied it by the inverse of its frequency to generate its new word point value. There are now some massive point disparities, with "Q" being worth 625 times the value of "E." But, this is necessary in order to get rid of it as soon as possible, because the presence of the letter "Q" near the end of a game practically guarantees that the bonus won't be claimed. I then re-sorted the dictionary by the new letter weightings and ran the greedy algorithm using that dictionary (see Table 5.2).

Letter	Old point value	New point value
A	1	1.07
B	3	11.58
C	3	10.03
D	2	5.28
E	1	1.00
F	4	21.39
G	2	7.52
H	4	14.39
I	3	5.17
J	8	153.85
K	5	20.16
L	5	9.84
M	3	9.55
N	2	4.45
O	1	1.39
P	3	9.29
Q	10	625.00
R	1	1.65
S	1	1.04
T	1	1.98
U	2	4.99
V	4	37.04
W	4	22.35
X	8	173.91
Y	4	12.90
Z	10	138.89

Table 5.2: A table showing the adjusted point values of each letter.

With these new point values, I created a new, reordered dictionary that prioritizes words with difficult letters (see Table 5.3):

Word rank	Sorted by points	Score	Prioritising uncommon letters	Weighted score
1	quizzifications	765	quinquivalences	20103.70
2	tranquilizingly	765	quinquevalences	20041.08
3	zygophyllaceous	735	quinquagenarian	19455.88
4	quinquennially	728	quinquivalency	18915.57
5	oxyphenbutazone	720	quinquevalency	18857.13
	...	...	...	...
279,425	tit	9	eas	9.32
279,426	toe	9	sae	9.32
279,427	too	9	sea	9.32
279,428	tor	9	ess	9.23
279,429	tot	9	see	9.11

Table 5.3: A table showing the English dictionary, sorted by point value and adjusted point value.

Now, when the greedy algorithm is run, it will automatically take into account the new word scores and favour those. Ideally, this will take out the most difficult letters to create words with and leave us with fewer letters on the board at the end of each game. Here is a score comparison of a sample of 10 games done with the greedy algorithm and 10 games with the new, revised dictionary (see Table 5.4):

Without change:		With reordered dictionary:	
Score	Letters left	Score	Letters left
1715	7	1675	19
1785	15	1829	7
1863	4	1665	8
1884	8	1684	13
1727	3	1680	5
1771	13	1556	6
2128	6	1777	14
1708	8	1861	2
1860	2	1732	1
1761	3	1685	9
Average:			
1820.2	6.9	1714.4	8.4

Table 5.4: The results table for the greedy algorithm with and without the reordered dictionary.

These results, at face value, are rather disappointing, with no improvement in average number of letters left on the board and a statistically significant ( $P = 0.0431$ ) decline in average score. This can be partly explained by the over-valuing of rare letters being so extreme that even relatively short words containing them, like “qadi” or “qat” will be favoured over longer, higher-scoring words. Overall, this slightly reduces the net overall true point score over a game. But, this heuristic is just one building block that will make up the final model, without paying attention to the shape of the board or the vowel to letter ratio. As a result, a non-improvement isn’t necessarily a bad thing. Nevertheless, these results do suggest that this re-ordering of words might be too drastic, leaving a lot of common letters at the end.

## 5.6 Vowel to letter ratio

If you analyse the frequency of vowels in all words in English (including “y”, which can be interpreted as both a vowel and a consonant but for the purposes of this exercise is very valuable as a vowel, with words like “dry” containing no other vowels), you get the following results (see Fig. 5.7):

### Histogram of Vowel Ratio in all Words

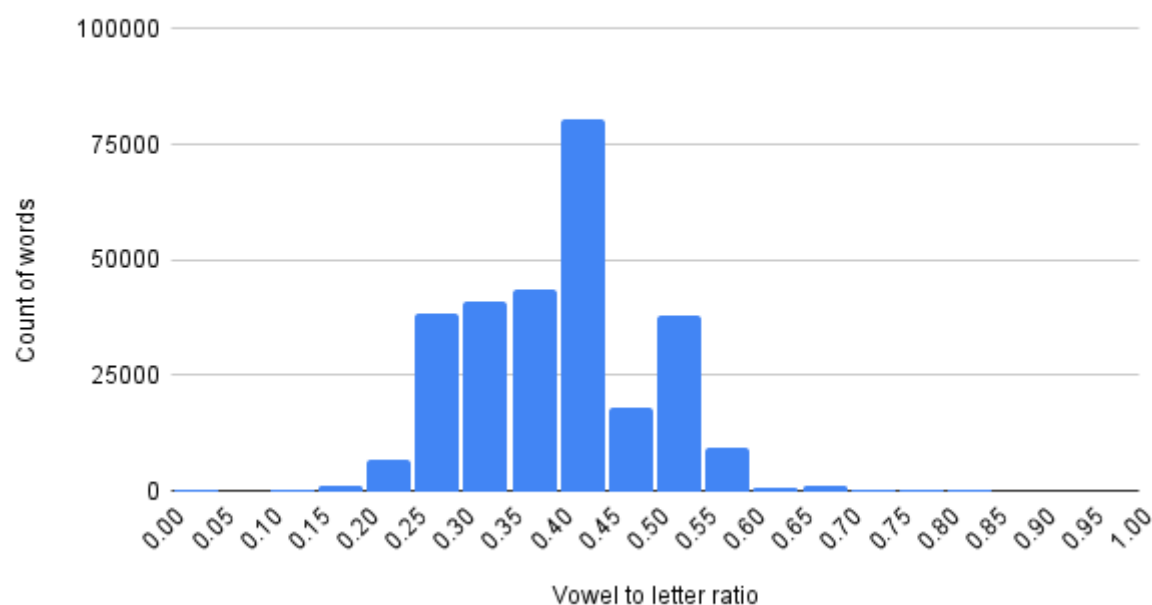


Figure 5.7: A histogram of vowel ratios in all words

The histogram shows that most words have a vowel ratio (the number of vowels divided by the total length of the word) between 0.25 and 0.5 inclusive, with a mean of 0.384. To maximise the number of words likely to exist on the board, the vowel ratio should ideally be kept within these ranges. To do so, I calculated the vowel ratio of the 10 highest-scoring words on each grid, calculated the vowel ratio of the remaining letters on the board, and created a modified score for each word that favours words that are close to the mean vowel ratio of all words. Using the *distfit* python package, I tested various distributions against the vowel ratio values, and found the normal distribution to be the closest fit, having a location parameter of 0.395 and a standard deviation of 0.087 (see Fig. 5.8).

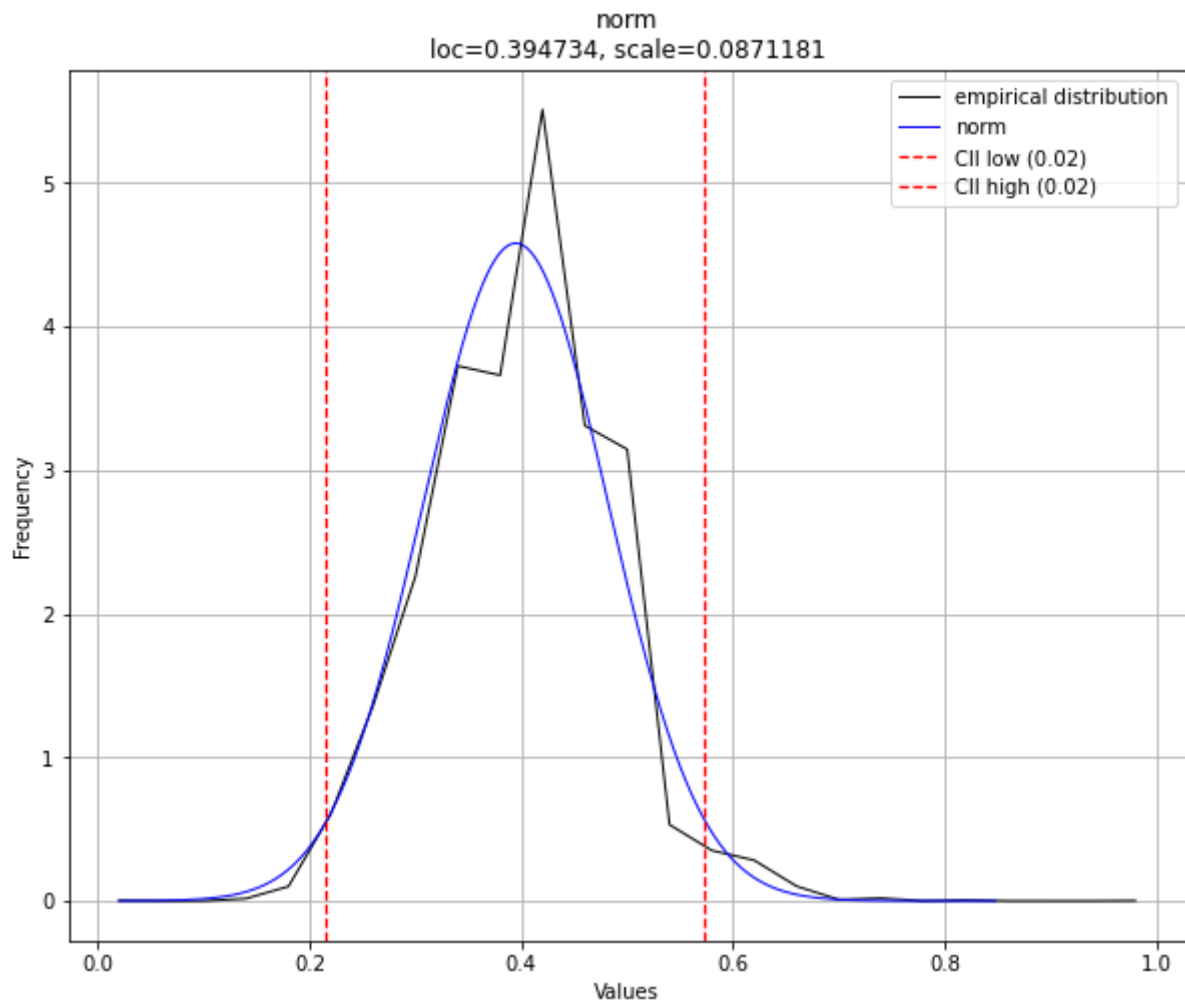


Figure 5.8: The closest distribution for the vowel ratio data

The new modified score is simply the product of the score with modified letter values and the frequency value obtained when you enter the vowel ratio of the letters left into the normal distribution closest to fitting the distribution of vowel ratio of all words. Here are the results of 10 test games (see Table 5.5):



Without change:		With vowel ratios:	
Score	Letters left	Score	Letters left
1715	7	1736	11
1785	15	1882	4
1863	4	1846	5
1884	8	1785	8
1727	3	1822	5
1771	13	1662	6
2128	6	1963	5
1708	8	1723	7
1860	2	1714	6
1761	3	2059	7
Average:			
1820.2	6.9	1819.2	6.4

Table 5.5: The results table for the greedy algorithm with and without adjusting word scores for vowel ratio.

Like with the reordered dictionary, there is no improvement in average score or letters left on the board. But, the same reasoning applies here as with the reordered dictionary: it should work better when used in combination with the other heuristics.

## 5.7 Connectivity

The connectivity of a grid can be defined in many ways, and depending on what exactly you want to encourage and discourage, different metrics can be used. For a game of *Word Soup*, you specifically want to minimise letters that only connect to one or two others, as they don't allow for potential word paths to fork: words including a letter connected to only 2 others will only be able to enter and leave it forwards or backwards. You also want to encourage situations where letters are connected to as many other letters as possible: if a letter is connected to other letters in all 8 directions, then other letters around it will border many letters themselves due to bordering each other.

To create a connectivity score for a board, I summed the square of the number of neighbours that each remaining letter is connected to, and divided by the number of letters remaining on the board. This ensures that words both keep as many squares very connected, but also heavily discourages

trailing tails of letters connected to just one or two others (see Fig. 5.9 and 5.10).



Figure 5.9: A well-connected square grid, with an average squared connectivity score of 30.75.



Figure 5.10: A poorly-connected two-tailed grid, with an average squared connectivity score of 7.6.

This also means that square board shapes are favoured over rectangular ones, especially on smaller grids (see Fig. 5.11).

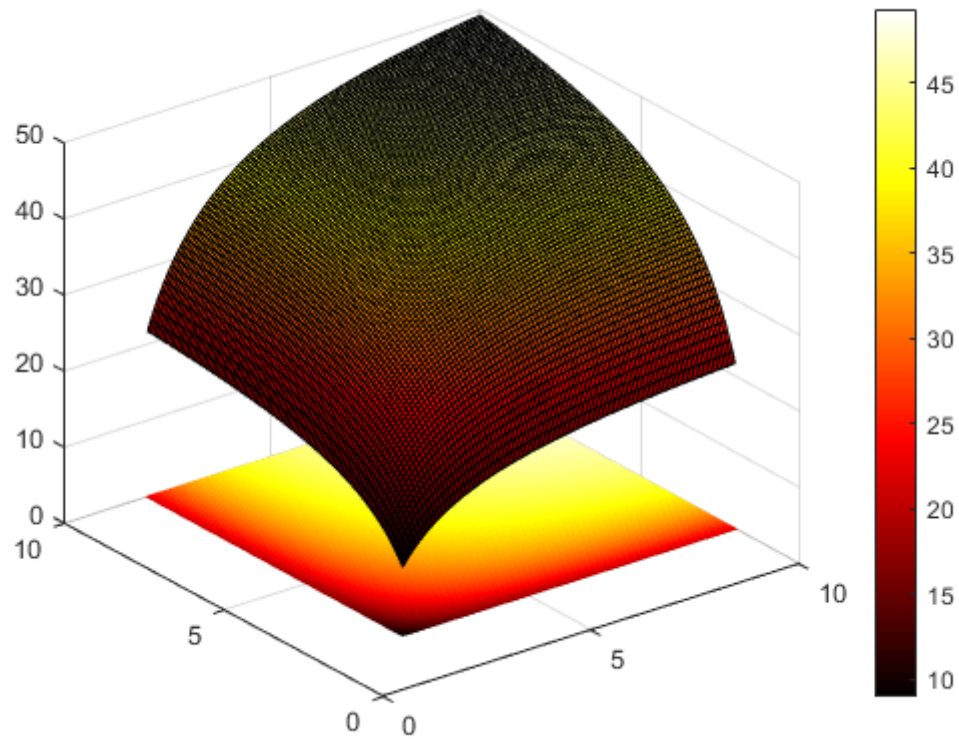


Figure 5.11: The average squared connectivity score of a rectangle of length  $x$  and  $y$ . The highest connectivity scores all fall along the line  $y = x$ , representing a square.

Running the greedy algorithm with word scores being altered for the connectivity of the remaining board for 10 games leaves us with the following results (see Table 5.6):

Without change:		With connectivity:	
Score	Letters left	Score	Letters left
1715	7	1750	2
1785	15	1834	10
1863	4	1650	5
1884	8	2316	0
1727	3	1705	1
1771	13	1718	5
2128	6	1724	11
1708	8	1799	9
1860	2	1720	7
1761	3	1705	6
Average:			
1820.2	6.9	1792.1	5.6

Table 5.6: The table of results comparing the greedy algorithm with and without adjusting word score for connectivity.

This is starting to look slightly promising, with one successful board clear! This is indicated by the “0” value in the “Letters left” column for the connectivity algorithm. Despite scoring  $2316 - 500 = 1816$  points on word scores, less than the 1834 scored on the second game, the game which cleared the board clearly outperformed every other game overall. Again there is no statistically significant change in score or letters left, there are a few boards which got incredibly close to clearing, indicating a step in the right direction. Nevertheless, this is just one heuristic of three, all of which should support each other. Perhaps, the final score could be weighted by the “distance to bonus” in order to create a combined metric to measure how close to achieving the bonus each algorithm is.

## 5.8 The heuristic model

By combining all three heuristic measures, we arrive at the heuristic algorithm. This algorithm works through the altered dictionary favouring rare letters, and finds the first 20 words on the board. Then, it takes their actual word scores and creates a modified score for each of them, taking into account the vowel to consonant ratio and the shape of the remaining letters. Then, it chooses the word with the highest modified score, and uses that as its turn. I chose to look at the first 20 words, as searching for more would counter the speed-increasing measures applied earlier, and the

best modified score almost never came from outside the first 20 found words in the first place. I ran this algorithm for 25 games, and compared it with 25 runs of the original greedy algorithm. Here are the results (see Table 5.7):

Greedy algorithm:		Heuristic algorithm:	
Score	Letters left	Score	Letters left
1715	7	2006	5
1785	15	1742	3
1863	4	1936	3
1884	8	1799	11
1727	3	1771	3
1771	13	1842	1
2128	6	1795	2
1708	8	1859	5
1860	2	1749	6
1761	3	1975	3
1869	2	2026	5
2078	9	1963	6
1796	4	2015	4
1871	10	1975	3
1749	5	1800	3
2159	4	1757	5
1796	11	1843	5
1960	3	1815	2
1783	1	1931	5
1839	13	1914	2
1700	5	1778	5
1606	15	1742	5
1766	13	2038	7
1770	12	2260	0
1621	10	1832	3
Average:			
1822.6	7.44	1886.52	4.08

Table 5.7: The results table comparing the performance of the greedy algorithm with the heuristic algorithm.

These results show a dramatic improvement in performance, which emphasizes the importance of combining all the heuristics. Not only has the average score improved slightly, the number of letters left has almost halved in a very statistically significant way ( $P = 0.0013$ ). Although only one board clear occurred, an average of only four letters remained on the board, with more than 5 letters remaining occurring 16% of the time, compared to 56% of the time with the greedy algorithm. This

means that most of the time, the board didn't end up with long tails of less useful letters, hence the heuristic algorithm is more efficient in using the available contents of the grid.

In comparison to the expert player, the heuristic algorithm performed much more closely than the greedy algorithm. The average score attained, 1886.52, is impressively close to the expert player's average score of 1948.7. The average number of letters left, 4.08, is closer to the expert player's average of 1.2 than it is to the greedy algorithm. This is impressive not only because of the improvement in performance, but also in the details of why the performance improved. The heuristic algorithm ends up playing the game in a more similar way to the expert player, paying attention to both what letters are being played and what letters are being left. Although this isn't manifested in very high-scoring words and board clears, it is clear in how high-scoring words are still played often, but with more attention paid as to which high-scoring word is the most advantageous to play. This plays off the strengths of lots of computing power, to produce an algorithm that can rival an expert player in performance, whilst taking no longer to play the game.

## Chapter 6

# Conclusion

My objective for this project was to design a heuristic algorithm that can beat human players in games of *Word Soup*. To this end, I have managed to program a simulation of the game in Python, from which to build and test potential algorithms. On this, I have created a breadth-first search algorithm to be able to determine if a word is on a board, store its locations in an array, and remove the word from the board in the same way as the app version. Using this and a dictionary, I then created a greedy algorithm that would repeatedly select the best word on the board until no more moves were possible. This greedy algorithm already managed to beat the amateur's performance consistently, but struggled to match the performance of the expert player. This was owing to the fact that the greedy algorithm, fitting with its name, failed to look ahead, resulting in it never attaining the 500 point bonus from clearing the board. To combat this, I employed a combination of three heuristics, all of which aim to focus not only on what word is being played, but also what letters are left on the board, and their arrangement. These heuristics didn't improve upon the greedy algorithm's performance when added in isolation, but when used together managed to attain a higher score than the greedy algorithm and leave a lower average number of letters on the board at the end of the game (even clearing the board on occasion). This heuristic algorithm performed in a more similar fashion to the expert player, but didn't manage to score a higher average.

One method that I briefly outlined but never integrated into the heuristic algorithm is the 2-move

and  $x$ -move greedy approach to playing the game, which would both take unrealistically long on large boards. If I were to create an algorithm that would improve upon the heuristic algorithm I designed, my next step would be to implement  $x$ -move deep once the board contained just a handful of letters, with the express aim of finding a combination of words that would completely clear the board. This wouldn't improve the word scores achieved at all due to the change in tactics only being used at the very end, but would significantly increase the proportion of games in which the board is fully cleared. If this algorithm managed to clear the board just 20% of the time, it would be able to outscore the expert player's average score. With the  $x$ -move deep algorithm I coded, there were already examples of board clearing on small grids, when the code executed correctly.

Another method I didn't integrate into the heuristic algorithm is the option to shift the board's horizontal gravity from right to left at any point in the game. Early in the game, this would allow for some more possibilities to score high-scoring words, and at the end of the game would allow for more opportunities to attempt to clear the board. It would also nearly double the execution time of the algorithm, as with every board you would have to check the same board shifted over to the other side. If I were to implement this, I would combine its use with  $x$ -move to increase its chances of clearing the board without overly increasing the algorithm's run time. There are, of course, completely different approaches to programming an algorithm to play *Word Soup*. Taking a page from the expert player, an algorithm could focus on attempting to build incredibly long, very high-scoring words once or twice a game. This is infeasible by searching far down an  $x$ -move tree without computing power far beyond feasibility, but perhaps by choosing words that, after being played, group together very long prefixes and suffixes (like "under-" and "-ication") and attempt to build very long words. Then, after successfully playing a very high-scoring word, the algorithm could switch tactics to attempt to clear the board. Alternatively, instead of designing an algorithm, an AI could be used instead. By feeding it millions of games and giving it a goal to maximize score, it may manage to eventually beat an expert player. This would come at the cost of knowing how the AI is attempting to solve the board, but if successful, could potentially help inform human players how to improve their own tactics.



Heuristic optimisation is useful not only in unsolved games such as *Word Soup*, but actually is applicable to real-world problems. One such problem is the “Travelling salesmen problem,” which aims to find the shortest route that connects  $n$  different locations. This problem is NP-hard, and its complexity increases with the factorial of the number of the sites visited. For a large  $n$ , computing the optimal solution is infeasible. As a result, heuristics are used. One such example that reduces the problem to polynomial complexity is its own greedy algorithm (also called the “nearest neighbour method” that attempts to minimize total distance (just like how the *Word Soup* greedy algorithm attempts to maximize score) [11]. By choosing a starting location and always choosing the single closest point not yet on the path, the problem is now only of order  $O(n^2)$ . This sacrifices the optimal solution, which rarely takes the form of the greedy solution, whilst arriving at a good solution. This has actual real-world applications, such as finding the best routes for a postal worker to take whilst delivering parcels. Once scaled to a multinational corporation such as UPS, small improvements using heuristics can save millions of dollars in time and fuel [12].

Whilst the techniques used in my heuristic algorithm to play *Word Soup* are commonplace in other heuristic algorithms already, I believe this project underlines the wide-ranging applications of using heuristics in a broad range of applications. The very same ideas applied to attempt to beat experts at a word game can be used in real life, to find close solutions to problems that would otherwise be infeasible to solve perfectly, and potentially make amazing efficiency improvements in all walks of our lives.

## Chapter 7

# Appendix

In this appendix, I include the key functions I used within the code written in the course of the project.

### 7.1 Variable setup

---

```
1 import random
2 import math
3 from copy import deepcopy
4 board_width = 9
5 board_height = 12
6 #The alphabet with letters proportioned
7 letters_proportioned = "
    aaaaaaaabcccddeeeeeeeeeeffgghhhhhhiiiiijklmmmnnnnnnn
    \ooooooooppqqrrrrrrssssstttttttttuuuvwxxyz"
8 #The actual point value of each letter
9 points = [("a",1),("b",3),("c",3),("d",2),("e",1),("f",4),("g",2),("h",4),("i",3),("j",8),("k",5),("l",5),("m",3),("n",2),("o",1),("p",3),("q",10),("r",1),("s",1),("t",1),("u
```

```

    ",2),("v",4),("w",4),("x",8),("y",4),("z",10)]
10 #The revised point value of each letter
11 new_points = [("a",1.06951871657754),("b",11.5830115830116)
    ,("c",10.03344482),("d",5.277044855),("e",0.998003992),("
    f",21.3903743315508),("g",7.518796992),("h"
    ,14.3884892086331),("i",5.172413793),("j",153.8461538),("
    k",20.16129032),("l",9.842519685),("m",9.554140127),("n"
    ,4.454342984),("o",1.394700139),("p",9.287925697),("q"
    ,625),("r",1.650165017),("s",1.03950104),("t"
    ,1.98412698412698),("u",4.987531172),("v",37.03703704),("
    w",22.34636872),("x",173.913043478261),("y"
    ,12.9032258064516),("z",138.8888889)]
12 letters = ["a","b","c","d","e","f","g","h","i","j","k","l","
    m","n","o","p","q","r","s","t","u","v","w","x","y","z"]
13 works = []
14 number_left_array = []
15 letters_left_percentage = []
16 points_percentage = []
17 points_p = [0]
18 letters_p = [board_height*board_width]
19 board = []
20 best_score_tree = []
21 best_word = ""
22 current_word = "zzzzzz"
23 best_score = 0
24 found_word = 0
25 final_score = 0
26 board_finished = 0

```

---

## 7.2 Game Setup

---

```
1  #Uses the altered dictionary
2  def open_file(filename = "altered_points_2.txt"):
3      file = open(filename, "r")
4      contents = file.read()
5      global dictionary
6      dictionary = contents.splitlines()
7      file.close()
8
9  def sort_dictionary(dictionary):
10     i = 0
11     for word in dictionary:
12         word = word.lower()
13         dictionary[i] = dictionary[i].lower()
14         point = 0
15         dictionary[i] = [dictionary[i]]
16         for letter in word:
17             for item in points:
18                 if item[0] == letter:
19                     point += item[1]
20         point *= len(word)
21         dictionary[i].append(point)
22         i+=1
23     dictionary.sort(key=lambda s: s[1], reverse=True)
24
25  def remove_short_words(dictionary):
26     for i in range(len(dictionary)):
27         dictionary[i] = dictionary[i][0]
```

```
28     remove_list = []
29     for word in dictionary:
30         if len(word) <= 2:
31             remove_list.append(word)
32     for word in remove_list:
33         dictionary.remove(word)
34
35 def create_board():
36     global board
37     global letters_left
38     board = []
39     for i in range(board_height):
40         board.append([])
41     #Creates the board
42     #start_time = time.perf_counter()
43     for i in range(board_height):
44         for j in range(board_width):
45             board[i].append(random.choice(
46                 letters_proportioned))
47     letters_left = board_height*board_width
```

---

### 7.3 Searching for words

---

```
1 def double_letters_missing(letters = letters, board = board)
   :
2     global double_letters_not_on_board
3     double_letters_not_on_board = []
4     for i in range(26):
```

```

5         for j in range(26):
6             double_letters_not_on_board.append(letters[i]+
              letters[j])
7     for i in range(len(board)):
8         for j in range(len(board[i])):
9             for k in range(8):
10                down = 0
11                right = 0
12                if k < 3:
13                    down = -1
14                if k > 4:
15                    down = 1
16                if k in (0, 3, 5):
17                    right = -1
18                if k in (2, 4, 7):
19                    right = 1
20                if 0<=i+down<=len(board)-1 and 0<=j+right<=
              len(board[i])-1:
21                    if board[i][j]+board[i+down][j+right] in
              double_letters_not_on_board:
22                        double_letters_not_on_board.remove(
              board[i][j]+board[i+down][j+right
              ])
23
24 def check_word(current_word = current_word, board = board,
              found_word = 0, best_score = best_score, best_word =
              best_word, best_score_tree = best_score_tree, works =
              works):
25     global point

```

```

26     global acc_point
27     point = 0
28     acc_point = 0
29     global list_of_paths
30     list_of_paths = []
31     on_board = 1
32     for i in range(len(current_word)-1):
33         if current_word[i]+current_word[i+1] in
           double_letters_not_on_board:
34             on_board = 0
35     if on_board == 1:
36         locations = []
37         for i in range(len(current_word)):
38             locations.append([current_word[i],[]])
39             for j in range(len(board)):
40                 for k in range(len(board[j])):
41                     if board[j][k] == current_word[i]:
42                         locations[i][1].append([j,k])
43     tree = []
44     for i in range(len(current_word)-1):
45         tree.append([i,[]])
46         for j in range(len(locations[i][1])):
47             for k in range(len(locations[i+1][1])):
48                 if abs(locations[i][1][j][0]-locations[i
+1][1][k][0])<=1 and abs(locations[i
][1][j][1]-locations[i+1][1][k][1])
<=1 and not ((locations[i][1][j][0]-
locations[i+1][1][k][0]) == 0 and
abs(locations[i][1][j][1]-locations[i

```

```

+1][1][k][1]) == 0):
49         tree[i][1].append([j,k])
50     for i in tree[0][1]:
51         list_of_paths.append(i)
52     for i in range(1, len(tree)):
53         for k in range(len(list_of_paths)):
54             for j in range(len(tree[i][1])):
55                 try:
56                     if list_of_paths[k][i] == tree[i
                    ][1][j][0]:
57                         if len(list_of_paths[k]) == i+1:
58                             list_of_paths[k].append(tree
                                [i][1][j][1])
59                         elif len(list_of_paths[k]) == i
                            +2:
60                             new_path = list(
                                list_of_paths[k])
61                             new_path[-1] = tree[i][1][j
                                ][1]
62                             list_of_paths.append(
                                new_path)
63                 except IndexError:
64                     continue
65     remove_list = []
66     for i in range(len(list_of_paths)):
67         if len(list_of_paths[i]) != len(current_word):
68             remove_list.append(list_of_paths[i])
69     for i in range(len(remove_list)):
70         if remove_list[i] in list_of_paths:

```



```

71         list_of_paths.remove(remove_list[i])
72     for k in range(len(current_word)-2):
73         for i in range(len(current_word)-(k+2)):
74             if current_word[i] == current_word[i+k+2]:
75                 for j in range(len(list_of_paths)):
76                     if list_of_paths[j][i] ==
77                        list_of_paths[j][i+k+2]:
78                         remove_list.append(list_of_paths
79                            [j])
80     for i in range(len(remove_list)):
81         if remove_list[i] in list_of_paths:
82             list_of_paths.remove(remove_list[i])
83 if list_of_paths != []:
84     for letter in current_word:
85         if change_method == 0:
86             temp_iter = 0
87             for item in new_points:
88                 temp_iter += 1
89                 if item[0] == letter:
90                     point += item[1]
91                     acc_point += points[temp_iter-1][1]
92             elif change_method == 1:
93                 for item in points:
94                     if item[0] == letter:
95                         point += item[1]
96 point *= len(current_word)
97 acc_point *= len(current_word)
98 works.append([current_word, point, list_of_paths])

```

```

98 def remove_word(best_word = best_word, best_score_tree =
    best_score_tree, board = board):
99     for i in range(len(best_word)):
100         letter = best_word[i]
101         order = best_score_tree[0][i] + 1
102         occurrence_number = 0
103         for j in range(len(board)):
104             for k in range(len(board[j])):
105                 if letter in board[j][k]:
106                     occurrence_number += 1
107                 if occurrence_number == order:
108                     board[j][k] += '_'
109                     occurrence_number += 1
110     for i in range(len(board)):
111         for j in range(len(board[i])):
112             if '_' in board[i][j]:
113                 board[i][j] = ' '
114     column_number = 0
115     while column_number < len(board[0]):
116         for i in range(len(board)):
117             if board[len(board) - i - 1][column_number] == '
                ':
118                 number_up = 0
119                 above_letter = ' '
120                 while above_letter == ' ' and number_up <=
                    len(board) - 2 - i:
121                     above_letter = board[len(board) - i -
                        number_up - 2][column_number]
122                     number_up += 1

```

```

123         board[len(board) - i - 1][column_number] =
            above_letter
124         if above_letter != " ":
125             board[len(board) - i - 1 - number_up][
                column_number] = ' '
126         column_number += 1
127     for i in range(len(board)):
128         for j in range(len(board[i])):
129             if board[i][len(board[i]) - j - 1] == ' ':
130                 number_up = 0
131                 above_letter = ' '
132                 while above_letter == ' ' and number_up <=
                    len(board[i]) - 2 - j:
133                     above_letter = board[i][len(board[i]) -
                        j - number_up - 2]
134                     number_up += 1
135                 board[i][len(board[i]) - j - 1] =
                    above_letter
136             if above_letter != " ":
137                 board[i][len(board[i]) - j - 1 -
                    number_up] = ' '

```

---

## 7.4 Heuristics

---

```

1 def find_vowel_ratio(board = board):
2     global vowel_ratio
3     vowel_ratio = 0
4     for i in range(len(board)):

```

```

5         for j in range(len(board[i])):
6             if board[i][j] == "a" or board[i][j] == "e" or
              board[i][j] == "i" or board[i][j] == "o" or
              board[i][j] == "u" or board[i][j] == "y":
7                 vowel_ratio += 1
8         vowel_ratio = vowel_ratio/(0.01+(letters_left-len(
              current_word)))
9         return vowel_ratio
10
11 def check_connectivity(board = board):
12     global connectivity
13     connectivity = 0.0
14     for i in range(len(board)):
15         for j in range(len(board[i])):
16             if board[i][j] in letters:
17                 temp_connect = 0
18                 for k in range(8):
19                     down = 0
20                     right = 0
21                     if k < 3:
22                         down = -1
23                     if k > 4:
24                         down = 1
25                     if k in (0, 3, 5):
26                         right = -1
27                     if k in (2, 4, 7):
28                         right = 1
29                     if 0<=i+down<=len(board)-1 and 0<=j+
                        right<=len(board[i])-1:

```

```
30         if board[i][j] in letters:
31             temp_connect += 1
32             connectivity += temp_connect**2
33     connectivity = connectivity/(0.01+(letters_left-len(
        current_word)))
34     return connectivity
```

---

## 7.5 Code execution

---

```
1 open_file()
2 sort_dictionary(dictionary)
3 remove_short_words(dictionary)
4 create_board()
5 find_vowel_ratio()
6
7 while board_finished == 0:
8     for i in board:
9         print(i)
10    double_letters_missing(board = board)
11    best_score = 0
12    acc_best_score = 0
13    best_word = ""
14    best_score_tree = []
15    found_word = 0
16    iterate = -1
17    potentials = []
18    while found_word <= 20:
19        iterate += 1
```

```

20         try:
21             current_word = dictionary[iterate]
22         except IndexError:
23             found_word = 21
24             break
25     check_word(current_word = dictionary[iterate], board
                = board, found_word = found_word, best_score =
                best_score, best_word = best_word,
                best_score_tree = best_score_tree, works = works)
26     if point > 0:
27         imaginary_board = deepcopy(board)
28         remove_word(best_word = current_word,
                     best_score_tree = list_of_paths, board =
                     imaginary_board)
29         find_vowel_ratio(board = imaginary_board)
30         check_connectivity(board = imaginary_board)
31         if change_method == 0:
32             modified_score = (connectivity)*acc_point
33                             *((1/(0.0871*(2*math.pi)**0.5))*math.e
34                             **(-0.5*((vowel_ratio-0.395)/0.087)**2))
35             potentials.append([current_word,
36                             modified_score, acc_point, list_of_paths
37                             ])
38         else:
39             modified_score = (connectivity)*point
40                             *((1/(0.0871*(2*math.pi)**0.5))*math.e
41                             **(-0.5*((vowel_ratio-0.395)/0.087)**2))
42             potentials.append([current_word,
43                             modified_score, point, list_of_paths])

```

```
37         if letters_left - len(current_word) in [1,2]:
38             potentials[found_word][1] *= 0.1
39             found_word += 1
40     potentials.sort(key=lambda s: s[1], reverse=True)
41     try:
42         x = potentials[0][2]
43     except:
44         board_finished = 1
45     if board_finished == 0:
46         if change_method == 0:
47             print(potentials[0][0], potentials[0][2])
48             final_score += potentials[0][2]
49         else:
50             print(potentials[0][0], potentials[0][2])
51             final_score += potentials[0][2]
52     points_p.append(final_score)
53     letters_left -= len(potentials[0][0])
54     letters_p.append(letters_left)
55     imaginary_board = deepcopy(board)
56     remove_word(best_word = potentials[0][0],
57                 best_score_tree = potentials[0][3], board = board
58                 )
59     number_left_array.append([letters_left, final_score])
60     print(final_score)
61     print(number_left_array)
```

---

## 7.6 *x*-move greedy algorithm

Unfortunately, this code is faulty and only worked on occasion, so couldn't be incorporated into the heuristic algorithm.

---

```
1 total_checks = 0
2 final_score = 0
3 board_finished = 0
4 number_left_on_board = board_height * board_width
5 bonus = 0
6 max_depth = [50,50,1]
7 max_depth_incise = max_depth
8 number_deep = len(max_depth)
9 while board_finished == 0:
10     if number_left_on_board <= 12:
11         get_dict("3_5_sorted_scrabble_words_points.txt")
12         max_depth = [50,50,1]
13         number_deep = len(max_depth)
14     for i in board:
15         print(i)
16     complete = 0
17     board_finished = 0
18     double_letters_missing(board = board)
19     depth = 0
20     end_of_depth = 0
21     best_iteration = []
22     best_score = 0
23     best_word_list = []
24     best_score_tree = []
25     current_board_tree = []
```



```

26     current_score = []
27     current_score_tree = []
28     current_word = ""
29     current_word_tree = []
30     iteration = []
31     for i in range(number_deep):
32         if i == 0:
33             current_board_tree.append(deepcopy(board))
34         else:
35             current_board_tree.append([])
36             current_score.append(0)
37             current_score_tree.append([])
38             current_word_tree.append("")
39             iteration.append([0,0])
40     while complete == 0:
41         current_word = dictionary[iteration[depth][1]]
42         check_word(current_word = dictionary[iteration[depth
43             ][1]], board = deepcopy(current_board_tree[depth
44             ]), works = works)
45         if works != []:
46             current_score[depth] = point
47             current_score_tree[depth] = deepcopy(
48                 list_of_paths)
49             current_word_tree[depth] = current_word
50             if sum(len(s) for s in current_word_tree) ==
51                 number_left_on_board:
52                 bonus = 500
53                 if (sum(current_score) + bonus) > best_score
54                     :

```

```

50         best_score = sum(current_score) + bonus
51         best_iteration = deepcopy(iteration)
52         best_word_list = deepcopy(
53             current_word_tree)
54         best_score_tree = deepcopy(
55             current_score_tree)
56         print(best_iteration, best_score,
57             best_word_list)
58         for i in range(number_deep):
59             for j in range(4):
60                 print(current_board_tree[i][
61                     j])
62             bonus = 0
63         if depth!= number_deep - 1:
64             current_board_tree[depth+1] = remove_word(
65                 best_word = current_word, best_score_tree
66                 = [current_score_tree[depth][0]], board
67                 = deepcopy(current_board_tree[depth]))
68         else:
69             end_of_depth = 1
70             if (sum(current_score) + bonus) > best_score
71                 :
72                 best_score = sum(current_score) + bonus
73                 best_iteration = deepcopy(iteration)
74                 best_word_list = deepcopy(
75                     current_word_tree)
76                 best_score_tree = deepcopy(
77                     current_score_tree)
78                 print(best_iteration, best_score,

```

```

        best_word_list)
69     iteration[depth][1] += 1
70     if iteration[0][0] == max_depth[0] or iteration
        [0][1] >= len(dictionary):
71         complete = 1
72     elif iteration[depth][0] == max_depth[depth]:
73         iteration[depth] = [0,0]
74         depth -= 1
75     elif iteration[depth][1] == len(dictionary)-1:
76         for i in range(number_deep - depth):
77             iteration[depth + i] = [0,0]
78             current_board_tree[depth+i] = []
79             current_word_tree[depth+i] = ""
80             current_score_tree[depth+i] = []
81             current_score[depth+i] = 0
82         if depth <= 0:
83             complete = 1
84         if (sum(current_score) + bonus) > best_score:
85             best_score = sum(current_score) + bonus
86             best_iteration = deepcopy(iteration)
87             best_word_list = deepcopy(current_word_tree)
88             best_score_tree = deepcopy(
                current_score_tree)
89             print(best_iteration, best_score,
                best_word_list)
90         depth -= 1
91     elif works != []:
92         iteration[depth][0] += 1
93     if end_of_depth == 0:

```

```
94         depth += 1
95     works = []
96     end_of_depth = 0
97     bonus = 0
98     if best_word_list == []:
99         board_finished = 1
100     else:
101         for i in range(number_deep):
102             if best_word_list[i] != "":
103                 board = remove_word(best_word =
                                     best_word_list[i], best_score_tree = [
                                     best_score_tree[i][0]], board = board)
104             number_left_on_board -= sum(len(s) for s in
                                     best_word_list)
105             final_score += best_score
106         print(best_score, "\n", best_iteration)
107
108 print(final_score)
```

---

# Bibliography

- [1] Wardle, J. (2022). *Wordle*, <https://www.nytimes.com/games/wordle/index.html>.
- [2] Contributors to Wikimedia projects. (2005, July 13). World Scrabble Championship - Wikipedia, the free encyclopedia. [https://en.wikipedia.org/wiki/World\\_Scrabble\\_Championship](https://en.wikipedia.org/wiki/World_Scrabble_Championship)
- [3] 3Blue1Brown. (2022, February 6). *Solving Wordle using information theory* [Video]. YouTube. <https://www.youtube.com/watch?v=v68zYyaEmEA>
- [4] Munroe, R. (2010, December 10). *Tic-Tac-Toe*. xkcd. <https://xkcd.com/832/>
- [5] Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314), 256–275. <https://doi.org/10.1080/14786445008521796>. Page 4.
- [6] Sadler, M., Regan, N., Kasparov, G. (2019). *Game Changer: AlphaZero’s Groundbreaking Chess Strategies and the Promise of AI*. Continental Sales, Inc.
- [7] Fuzzy Bug Interactive Ltd. (2015). *Word Soup*. <http://www.wordsoup.com/>
- [8] Kenny, V., Nathal, M., Saldana, S. (2014, May 25). *Heuristic algorithms - optimization*. Northwestern University Open Text Book on Process Optimization. [https://optimization.mccormick.northwestern.edu/index.php/Heuristic\\_algorithms#:~:text=A%20heuristic%20algorithm%20is%20one,a%20class%20of%20decision%20problems](https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms#:~:text=A%20heuristic%20algorithm%20is%20one,a%20class%20of%20decision%20problems).
- [9] Morrison, N. (2021, October 7) *Game data from 10 games of Word Soup by expert player*.

- [10] Dictionaries, C. C. (2019). *Collins Official Scrabble Words: The Official, Comprehensive Wordlist for Scrabble*. HarperCollins Publishers Limited.
- [11] Ma, S. (2020, January 2). *Solving the Travelling Salesman Problem for deliveries*. Delivering Happiness – A Blog for Delivery Businesses. <https://blog.routific.com/travelling-salesman-problem>
- [12] López, E. E., Palazuelos, M. T. (2019). Application of a Heuristic to Reduce Fuel Consumption for the Traveling Salesman Problem. In *Operations Management for Social Good (pp. 501–508)*. Springer International Publishing. [https://doi.org/10.1007/978-3-030-23816-2\\_49](https://doi.org/10.1007/978-3-030-23816-2_49)