UNIVERSITY COLLEGE DUBLIN
TRINITY COLLEGE

DISSERTATION
MAI IN ELECTRONIC & COMPUTER ENGINEERING

Developing an Artificial Intelligence to Play the Board Game Scrabble

Author:
Carl O'CONNOR

Supervisor:
Dr. Carl VOGEL

Submitted to the University of Dublin, Trinity College, May, 2015

Declaration

I, Carl O'Connor, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Wednesday 20th May, 2015

Carl O'Connor

Permission to Lend

I agree that the Library and other agents of the College may lend or copy this dissertation upon request.

Wednesday 20th May, 2015

Carl O'Connor

Summary

This project uses a data structure called a GADDAG for lexicon storage and a weighted heuristics method for word evaluation in a computer based game of Scrabble. The GADDAG structure is assessed for its speed and performance in generating words. It was found to be well suited for this purpose facilitating rapid word generation. It required quite a large amount of system memory to be constructed however which is highlighted to be a possible flaw. Weighted heuristics proved to be a fast and reliable method for move evaluation for the most part. However, it was prone to some strategic errors is judgement. An alternative method to address this is also discussed. Weighted heuristics could also be used to introduce other game elements such as realism to the game. A method for doing this using a word frequency data source is proposed.

Abstract

This project describes a Java implementation of the well known board game Scrabble. The application was written to be played by a human and computer controlled player. It takes a look back at previous attempts to implement such a system and draws on these. Lastly, an implementation of a more realistic computer controlled opponent is proposed.

Acknowledgements

I would like to thank the following people:

My supervisor, Dr. Carl Vogel, for his constant help and guidance during this project. His assistance and enthusiasm throughout the project made it both a rewarding and enjoyable experience.

My family for their continuing support over the years.

My friends for providing a distraction from college when I needed to unwind.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The aim of this project was to implement a computer controlled Artificial Intelligence (AI) to play the well known board game Scrabble.

1.2 Motivation

Although this area of research has been covered before, it has not been researched since the early 90's. Back then, computers were not as advanced and powerful as they are today. This meant computational efficiency was of primary concern and certain AI methods were not yet feasible. Thus, I chose this project with the hopes of building on previous authors work.

Scrabble is also a game enjoyed by millions of people around the world, myself included. It is a game that blends vocabulary and strategy, and makes for an interesting AI project idea.

1.3 Aims

Once my project title was selected I went about dividing the project into smaller parts and establishing my overall aims. These included:

- To investigate the different lexicon data structures, their performance and how to implement them.
- To find a way of programming a computer to play Scrabble.
  - How can a computer understand the board?
  - How does it know how and where to play moves?
  - How does it choose a good move?
- To create a Java based game of Scrabble including an AI to play against.
- To implement an AI that plays Scrabble as well as possible.
- To investigate the idea of introducing a more realistic Scrabble playing computer AI.

1.4 A Look Ahead

Chapter 2 will discuss the literature already available on this research topic and examine the current state of the art. It will go through earlier attempts at implementing a Scrabble AI and the methods they tried.

Chapter 3 discusses the system architecture for my own implementation. It goes over some of the classes that were written and their role in the system as a whole.

Chapter 4 looks at the various methods of storing a lexicon. It examines a few data structures and how suitable each is for the purpose of this project.

Chapter 5 will discuss how the computer can assess the state of the board. It goes through some of the preliminary steps that need to be carried out before a full word search can be done.

Chapter 6 discusses how the computer searches for moves. It mentions the algorithm that was implemented as well as its performance.

Chapter 7 will then discuss how the computer evaluates moves. It will discuss different methods of choosing a move given a complete list of possible moves.

Chapter 8 discusses the idea of introducing a more realistic AI to play Scrabble. It mentions how a more human-like computer player could be implemented and possible ways of evaluating its "humanness".

And finally, in **Chapter 9** I will reflect on the outcome of the project, my achievements, and some ways I think this project could be expanded upon.

Chapter 2

State of the Art

2.1 Introduction

Over the years there have been a number of computer programs developed to play Scrabble. A range of algorithms and data structures have been designed and implemented for this purpose, each with varying degrees of success. This chapter will mention some of these past implementations and the papers associated with them. It will finish by looking at MAVEN, the current state of the art in this field.

2.2 Literature Review

One of the earliest Scrabble programs released was a commercial game called MONTY released in 1983. Expert human players were able to beat MONTY consistently but said it posed a significant challenge. According to *Scrabble Players News*, MONTY was designed to use both strategic and tactical concepts [8].

Peter Turcan from the University of Reading developed an early computer based Scrabble player [18] [19]. In his implementation he stores the lexicon as a list of words in reverse order of length. For each word he decides whether and where the word can be played on the board with the given rack. Unlike many other early Scrabble programs, his move evaluation function takes simple strategic features of the board and tiles left on the rack into consideration. Most other programs at the time used a greedy algorithm for move evaluation, simply choosing the move with the highest point yield.

Stuart Shapiro of SUNY Buffalo has implemented several Scrabble programs [16]. His implementations use a tree structure of letters to represent the lexicon, where each path down the tree has an associated list of words which can be played using the letters on that path. The tree is ordered with higher scoring letters appearing higher in the tree so that higher scoring words are found first, in case a full search cannot be completed in the given time. His move evaluation takes the first move found with a score higher than some threshold.

Andrew Appel and Guy Jacobson discuss an algorithm they developed that allows for rapid move generation in Scrabble [7]. Their implementation uses a Directed Acyclic Word Graph (DAWG) to represent the lexicon. They discuss how their program analyses the board in terms of anchor squares and cross sets. A "backtracking" algorithm is proposed which generates moves from anchor squares by constructing word prefixes to the left, and then generating words by constructing corresponding suffixes to the right. Their program uses a simple evaluation function but they suggest at the possibility of improving performance by using a more sophisticated heuristics function.

Stephen Gordon developed a data structure, called a GADDAG, which can be used to reduce the amount of time taken to generate all possible word plays

when compared to using a DAWG. The structure was designed specifically for the purpose of word generation in Scrabble. He uses a word generation algorithm that is a slight variation on the one proposed by Appel and Jacobson. Gordon's representation stores a version of each word in the lexicon starting from each letter in the word. His proposed representation is five times larger in memory but allows words to be generated twice as fast compared to Appel and Jacobson [12]. He later goes on to analyse the use of different heuristics in move evaluation, and employs simple machine learning techniques to "tune" these AI parameters [11]. Lastly, he investigates the use and performance of a probabilistic search method for evaluating moves. Since the probabilistic search method is computationally expensive, he suggests combining the two methods by using a heuristic function to narrow down the number of candidate moves to be considered.

2.3 State of the Art

Last but not least, Brian Sheppard reviews his own Scrabble AI, MAVEN, currently known to be the worlds best AI Scrabble player [17]. MAVEN's game strategy is broken down into three main phases:

- **Mid Game:** from the start of the game until there are 9 or fewer tiles lift in the letterbag.
- **Pre-End Game:** when there are between 1 and 9 tiles in the letterbag.
- **End Game:** when there are no tiles left in the letterbag.

For both the mid game and pre-end game phases MAVEN uses a complicated heuristics function and probabilistic search method to choose its move. The pre-

end game also aims to produce a favorable end game scenario (i.e. a good rack leave). During the end game, each player can deduce what tiles their opponent has by looking at the board and at their own tiles. Thus, during the end game phase Scrabble becomes a game of perfect information (i.e. each player knows everything about the current state of the game). For this phase of the game MAVEN uses a B* search algorithm to predict and play out the rest of the game. Sheppard touches on some of the many factors that are taken into consideration for MAVEN's heuristic function and gives an overview of how their values were arrived at. Finally he goes through some of MAVEN's tournament successes and a brief play by play review of MAVEN's AAAI[1] 1998 match against Adam Logan. Logan was one of the top Scrabble players in the world at the time and this game is considered by some to be the best game of Scrabble ever played in a tournament or match.

2.4 Conclusions

This chapter has discussed previous attempts at implementing an AI to play Scrabble, as well as some of the methods and data structures used to achieve this. My implementation will draw from some of the ideas presented in these papers. I intend to implement Steven Gordon's GADDAG data structure for lexicon storage and a heuristics function for move evaluation. These will be discussed in greater depth later on. In the next chapter I will give an overview of the system I implemented.

[1] AAAI - Association for the Advancement of Artificial Intelligence

Chapter 3

Architecture and Design

3.1 Introduction

This project was initially intended to be an extension of two previous projects [10] [15]. Unfortunately the code that was written for these two projects has since been lost. Therefore I will be starting this project from scratch. This can be seen as either a positive or negative. On the one hand there is no inherited code for me to try and understand. Inheriting code can sometimes be a mixed blessing. Although it can provide you with a good starting point, it also requires you to follow somebody else' train of thought and to understand the decisions they made. On the other hand I am starting from scratch. When coding from scratch, like writing a book, it can be difficult to decide where to begin and how to organize ones thoughts. Writing a large program such as this can be a daunting task. However breaking the program down into more manageable parts usually helps make the overall task simpler and easier to approach. This chapter will deal with how I divided up this program into smaller components and what purpose

each of these components serves.

3.2 Java

For this project I elected to program my implementation of Scrabble using the Java programming language. Java is one of the most used programming languages in the world today and has applications across a wide range of areas, from Graphical User Interface (GUI) design to the World Wide Web (WWW). Before this project I had not undertaken a major programming project in Java. Given the popularity and widespread use of this language this seemed to be something that was lacking in my computer science knowledge. Thus, this project gave me the perfect opportunity to do so. Java is also an Object Orientated Programming (OOP) language, which means it is naturally suited to representing and describing real world objects as data abstractions. Implementing Scrabble in Java involved designing these data abstractions and how they interact with one another to form a working game. Although Java may not be as fast or efficient as other languages such as C, or C++, it should suffice for the purpose of this project.

3.3 Java Virtual Machine

An added bonus of using Java is that it runs in the Java Virtual Machine (JVM). The JVM is a virtual computing machine that offers a standardized run-time environment for Java code to be executed in. It allows compiled Java code run on any platform that supports Java, regardless of the architecture of the underlying system. It is intended to allow developers to *"write once, run anywhere"* (WORA)

and makes applications written in Java highly portable.

Another advantage of using Java and the JVM, compared to C and C++, is that it automates system memory management. With Java the developer does not need to concern themselves with pointers or garbage collection as the JVM handles this automatically. The JVM can automatically detect when objects are no longer needed by the program and free up the memory space they occupy. The lack of garbage collection is known to be the largest source of errors for C and C++ developers [9].

3.4 System Design and Components

When writing any piece of code, especially for larger programs, it is important that the code is well structured. All code should be written in such a way that it is easy to follow and understand. In order to do this, it is important to break down the program into different components, functions and classes. Once a high level design has been drawn up, each of the individual parts can be written separately.

Figure 3.1 shows a layout of how I went about designing the architecture for this program. The core system components include:

- **GUI:** The GUI class allows the player to interface with the game. It displays the current board state, the players rack letters and the current score. The player can add letters from his or her rack to the board to make a move. The play button confirms and validates a move while the clear button returns the letters to the rack and lets the player start over. The GUI class was written to be a runnable class. In Java this essentially means it is executable in its own

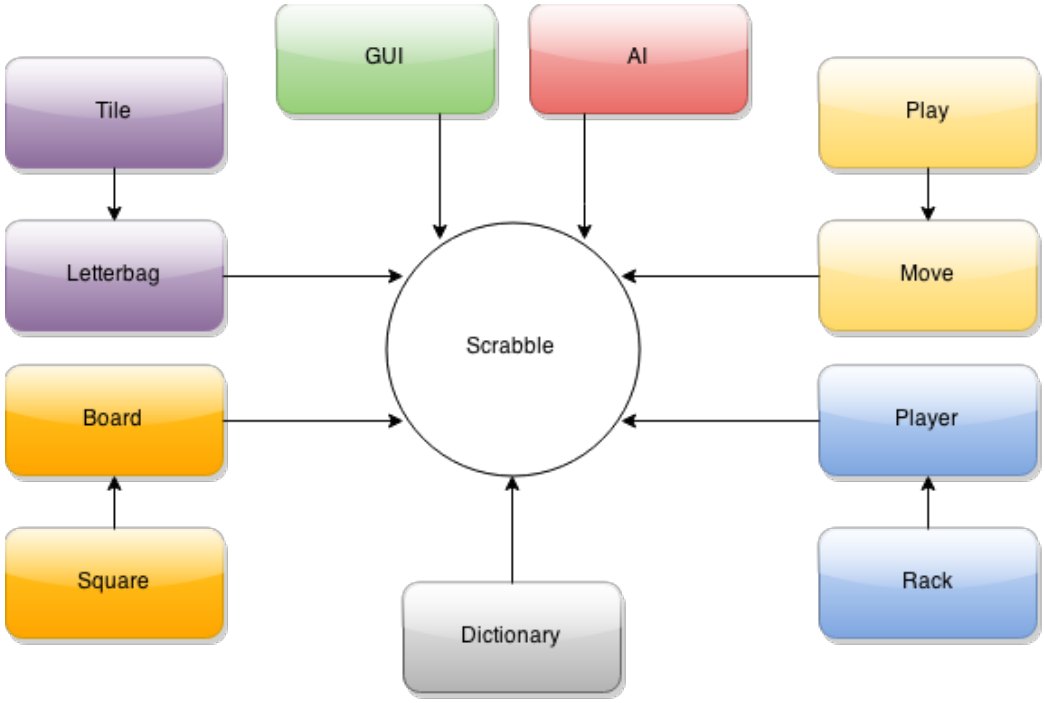

Figure 3.1: My Scrabble game system architecture.

thread. This was done so that the GUI was very responsive to user inputs giving a good user experience. Figure 3.2 shows what the GUI looks like.

- **Dictionary:** The Dictionary class stores the games dictionary. At the start of each game a dictionary object is constructed using a word list provided via a ".txt" file. The dictionary can be easily changed by swapping out this ".txt" file for another. This object is used throughout the game for validating words and finding moves for the computer to play.

- **Tile:** The Tile class describes the different letter tiles used in the game. It is implemented as an *enum* type class. This essentially restricts the values this object can take on to a set of predefined values. Instead of creating separate tile objects, this class acts more like a look-up table. The program can request the score of a tile by passing this class the character for that

letter. This saves the program from having to create multiple instances of tile objects, saving system memory, and simplifies how tiles are handled by the game. Each letter is encoded with its corresponding score. It also contains a function to pull the image for each letter tile to be used for the GUI.

- **Letterbag:** The Letterbag class is used for drawing tiles during the game. It is initialized with 100 characters according to the official letter distribution in Scrabble. Like the dictionary class, this letter distribution is read in from a ".txt" file when the program starts and can be easily swapped for another.

- **Square:** The Square class describes the individual squares on the board and the attributes associated with each. Each square has a word and letter multiplier, and X and Y board coordinates associated with it. Each square object can contain a single tile. Each square also has its own vertical and horizontal cross sets whose purpose will be explained in Chapter 5.

- **Board:** The Board class describes the game board. It is composed of 225 square objects arranged in a 15 x 15 grid. It has functions used to validate and score player moves, assess the board for anchor squares and cross sets, and play moves on the board. It is initialized by reading in an ASCII map describing the board. This ASCII map is stored as a ".txt" file and can be swapped out to play the game using different board layouts.

- **Play:** The Play class describes individual letter plays. It is made up of a single character as well as X and Y coordinates (e.g. letter 'X' at position [7, 7]). It is a very small class but simplifies much of the code associated with generating moves for the computer.

- **Move:** The Move class describes the word plays or moves that either player

makes. Each move object is made up of one or more play objects. It stores the score of that particular move and whether or not that move is a "bingo"[1].

- **Rack:** The Rack class is used to store each players letter tiles. It is a simple container class that draws tiles from the games letterbag object.

- **Player:** The Player class describes each of the players in the game. Each player object contains a name, a score and a rack object. In theory up to 4 player games could be easily implemented by extending the games player turn switching mechanic.

- **AI:** The AI class is used to allow the computer to evaluate and select which moves to play. It reads in a set of data from ".txt" files to facilitate this. The methods it uses will be discussed more in Chapter 7.

3.5 Conclusions

This chapter outlined my high level approach to designing a Java implementation of Scrabble. I described the main components and their purpose in the game. I will now take a more in depth look at some of the aspects that are specific to this project, namely lexicon storage, word generation and move evaluation.

.

[1] A bingo in Scrabble occurs when a player plays a move using all the letters on their rack. The player earns a bonus of 50 points for achieving this.

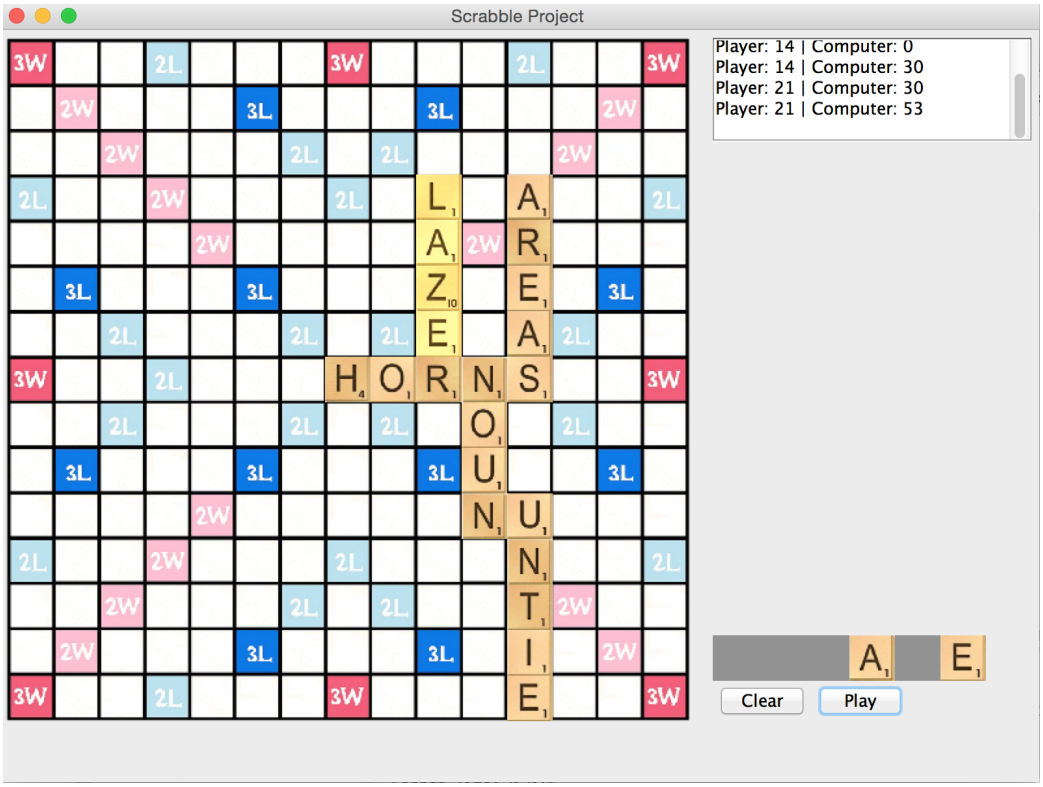

Figure 3.2: Scrabble game GUI. The letters 'L', 'A', 'Z', and 'E' have been placed on the board to be confirmed or cancelled by the player.

Chapter 4

Dictionary Storing

4.1 Introduction

One of the core elements involved in programming a word based computer game such as Scrabble is deciding how best to implement a dictionary structure. As with any decision regarding suitable data structure selection, there are multiple possible candidates to choose from, each with their own pros and cons. Lexicon storage is a topic that has been addressed multiple times in the past with many different purposes in mind. Thus, there is a wide range of data structures that have been proposed which need to be considered. This chapter will discuss some of the data structures that were considered for storing the game's dictionary.

The dictionary that will be used for the purpose of this project will be a copy of the Official Scrabble Players Dictionary (OSPD) downloaded from "www.puzzlers.org" [5]. This will be the default dictionary, however users can choose to use a different dictionary by swapping out the ".txt" file with one of

their choosing. This dictionary contains just under 80,000 words (79,338 to be exact) with no words greater than 8 characters in length.

The dictionary structure that will be implemented will need to serve two main functions in the game. These will be to:

1. Allow the game to check if a word played by the human player is a valid word in the game's dictionary, and
2. Enable the computer player to interact with the dictionary in some way so that it can generate a list of playable words.

As with any data structure and associated algorithms, two of the primary concerns in assessing which structure to use will be:

1. The order of complexity associated with retrieving data or interacting with the data structure, and
2. The total amount of space occupied in memory by the data structure itself.

4.2 List

Lists are one of the most common abstract data types used in computer programming. A list essentially represents a sequence of values where each value can occur more than once. If a value does occur more than once, each occurrence is considered to be a distinct item. Lists can be implemented using many concrete data structures such as linked lists and arrays.

Lists have the distinct characteristic that they can be ordered in some way. For storing a dictionary we would usually think to order a list of words in the most obvious way, alphabetically. However for a Scrabble game dictionary it may also be of use to store words based on their length or even the number of points they yield.

Such an implementation has been tried before, where a dictionary is stored as a list ordered by word length and then traversed in reverse order of length [7]. For each word the program determines whether and where the word can be played on the board. This method allows the program to impose a strict time limit on the computation time, ending the search once the time has been exceeded. By traversing the list in reverse order of length the probability of finding at least one possible play is greatly increased. This is because, for the most part, it is easier to find shorter words that can be played rather than longer ones.

Lists offer a worst case time complexity of O(N), meaning such a data structure would be quite suitable for checking if a word is valid. If the dictionary is stored alphabetically this could be improved by using a binary search method, giving a worst case time complexity of O(logN). Storing the dictionary alphabetically would mean the computer player has to search for moves by traversing from A to Z as well. This may not be ideal if a computational time limit is enforced as the computer may never get the chance to consider words towards the end of the dictionary.

4.3 Hash Table

A hash table is a data structure used to implement an associative array. It is composed of a collection of key - value pairs, where each key is unique and maps to a unique value, or a unique set of values. Hash tables use hash functions which take the input hash table keys and use them to compute the array index where the associated data can be retrieved.

These hash functions are usually implemented as some sort of arithmetic function which can be computed with O(1) time complexity. Thus, they facilitate rapid data look-up.

A hash table would be an ideal data structure to allow a Scrabble game to check if a word played by the human player is valid in the game's dictionary. The input hash key would be the candidate word, and would map to a boolean true value if the word is valid, or false if it is not.

However generating playable words is not quite as easy to facilitate with a hash table. The computer player could try entering random permutations of available rack and board letters as keys to find valid words. This exhaustive approach however would be very computationally expensive. Using between 1 - 7 letters from the rack, and at least one from the board, the number of permutations to try would be enormous (8! + 7! + ... + 3! + 2! = 46,232+ keys). Things would get even more complicated when two or more letters from the board are to be used, or if even more stringent board conditions are taken into consideration.

Another potential option could be to use hash keys that are made up of readily available letters, and which map to a list of words that can be played using these letters. The keys of available letters would be made up of the computer play-

ers rack letters and available board letters. Although this method would drastically reduce the computational cost of generating words it would involve storing a lot of duplicate words, especially smaller words which can be played with a large number of different rack letter combinations. E.g. the word “CAT” can be played with the racks “CATABCD”, “CATABCE” and countless other combinations. Each of these combinations would result in another “CAT” entry somewhere in the data structure which would result in an enormous waste of system memory.

4.4 Trie

A trie is an ordered tree data structure which is usually used to store strings. Unlike most other tree data structures, such as Binary Search Trees (BST), none of the nodes that make up the trie actually store the value associated with that node. Instead, each nodes position in the tree structure is what determines its associated value. All the descendants of a node share a common prefix of the string associated with that node. The root node of the tree is associated with an empty string.

The trie is a popular data structure for storing large dictionaries, and is used mainly for spell checking programs and other Natural Language Processing (NLP) applications. Like other tree based data structures, each node in the trie contains information about each of the other nodes that can be reached within one hop of that node. For the purpose of storing strings, or a dictionary, words are broken down into individual characters. These characters are represented by a transition from one particular node in the data structure to another. Thus, words can be constructed by traversing the trie and concatenating all of the transition letters between each of the nodes that were visited previously. When using a trie

for a spell checking program it would make sense to include a boolean value at each node to indicate whether or not the string constructed in arriving at that node is a valid dictionary word.

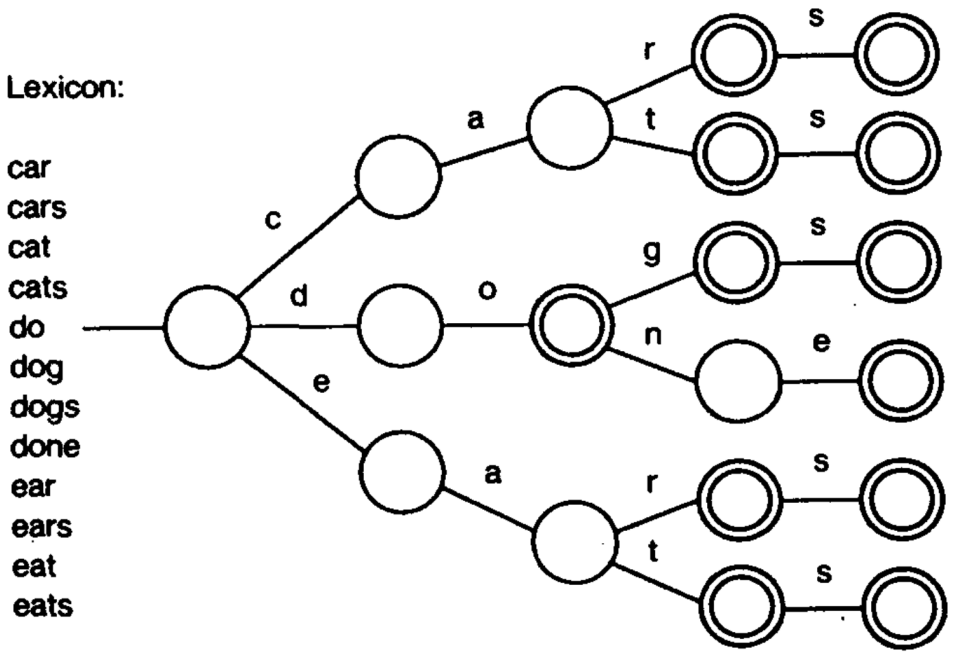

Figure 4.1: Trie structure for the given lexicon. Circled nodes indicate word endings. Source [7].

Since checking whether a word is valid in the dictionary would take m hops, where m is the length of the query word, the time complexity to query a trie in this fashion would be O(m). This is independent of the number of node elements in the trie, and returns searches faster than searching through a binary tree.

For example, consider a dictionary with words of length up to 8 characters. In the worst case scenario such a tree structure could contain up to $n = 26^8 = 208,827,064,576$ nodes, and a binary search would take $log_2 n \approx 37.6$ iterations

to check an 8 letter word. Checking the same 8 letter word using a trie would require just 8 iterations.

As well as facilitating rapid word checking, a trie is also naturally well suited to generating words given a set of letters. This makes it a suitable data structure to store a dictionary for a game like Scrabble. Figure 4.1 shows a trie structure for a small sample dictionary. Circled nodes indicate a word ending at that node. The structure can be traversed starting from the root node to construct words. Each transition between nodes is done via a letter. Thus, if a player does not have a certain letter they cannot continue along certain branches from their current node. This deterministic approach to word construction proves much more efficient than the "trial and error" attempts using lists or hash tables.

4.4.1 DAWG

A Directed Acyclic Word Graph (DAWG), or sometimes just Directed Acyclic Graph (DAG), is a data structure that is very similar to a trie. Unlike a trie however, each node in a DAG can be reached by multiple paths. This has the distinct advantage of allowing many redundant nodes to be removed. In this regard, a DAG can be considered to be a condensed version of a trie, and offers enormous space savings in memory over a trie. Figure 4.2 shows the structure of a DAWG. Like the trie, circled nodes indicate a word ending. Appel and Jacobson state that using this data structure to hold their lexicon reduced its size from 780 KB, as a trie, to 175 KB, as a DAWG, equating to a $\approx 77.6\%$ compression [7]. Checking and finding words using this data structure is done in much the same way as with a trie. Thus, it performs the same a as trie but requires less memory to be stored.

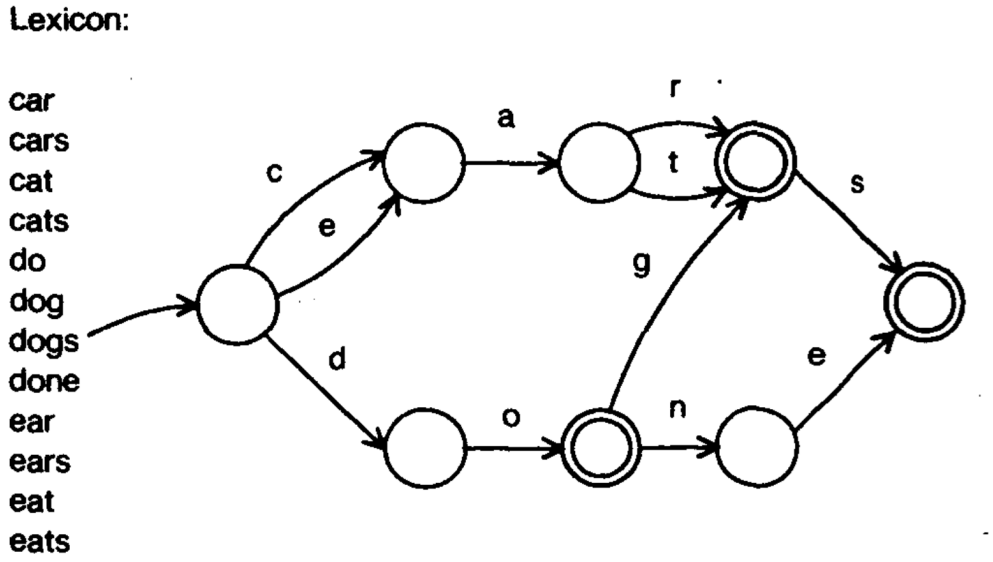

Figure 4.2: DAWG structure for the given lexicon. Circled nodes indicate word endings. Source [7].

4.4.2 GADDAG

Although the DAG already offers an extremely rapid way to both check and generate words from a given lexicon, it only facilitates word generation from left to right. The GADDAG data structure can be thought of as a two-way DAG, hence the name, and allows more flexibility in the way words are constructed.

It was first proposed by Stephen Gordon in a 1994 paper and was developed specifically to facilitate rapid word generation in Scrabble. Gordon argues that a DAG is not the most efficient lexicon representation for generating Scrabble moves [12]. This is because in Scrabble, a word is played by "hooking" one or more of its letters onto the words already played on the board, not just the first letter. Thus, the GADDAG encodes a bidirectional path starting from each letter

in each word of the lexicon. Using this method, each word has n valid paths which can be traversed to find it, where n is the number of letters in the word. It allows for words to be constructed starting from any of the letters in that word, not just the first. This results in a data structure that is nearly five times as large as a DAG for the same lexicon but is capable of generating moves at more the twice the speed. This time - space trade-off is justified by the decreasing cost of computer memory as well as the extensive use of move generation in more advanced board analysis techniques that will be discussed in Chapter 5. Figure 4.3 shows the 4 ways the word "CARE" is encoded in the GADDAG structure. Each node in this data structure contains an array of letters than can be used to traverse to a child node as well as an array of letters that can be used to end a word at that node.

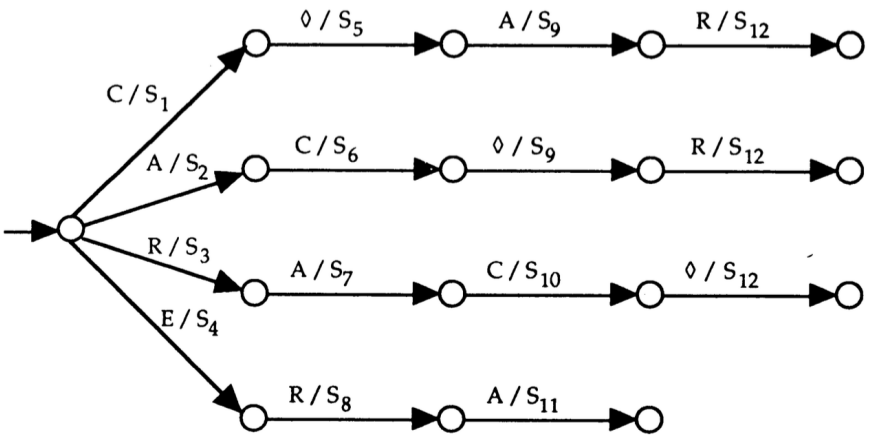

Figure 4.3: Un-minimized GADDAG structure for the word 'CARE'. Source [12].

The GADDAG facilitates word checking at the same speed as the DAG, with O(m) complexity, where m is the length of the query word. It also facilitates word generation that is more akin to how human players tend to find words. Typically expert human players tend to try and group letters into common prefixes and suffixes, and shuffle the remaining letters until they find a word. Generating words using the GADDAG structure follows a very similar method. Words are

constructed by traversing common prefixes first. Prefixes are stored backwards as they are read from right to left. A delimiting character is used to signify the end of a valid prefix. For the purpose of this project the "@" character was used as the delimiter. For each valid prefix found, the corresponding suffixes are traversed until complete words are generated. Using this method results in a complicated recursive function but allows for extremely rapid move generation.

Like the trie data structure, common prefix and suffix paths in the GADDAG can be merged to give large memory savings. Gordon provides pseudo-code for constructing and minimizing his GADDAG data structure shown in Figure 4.5 [12]. Figure 4.4 shows the resulting minimized GADDAG data structure.

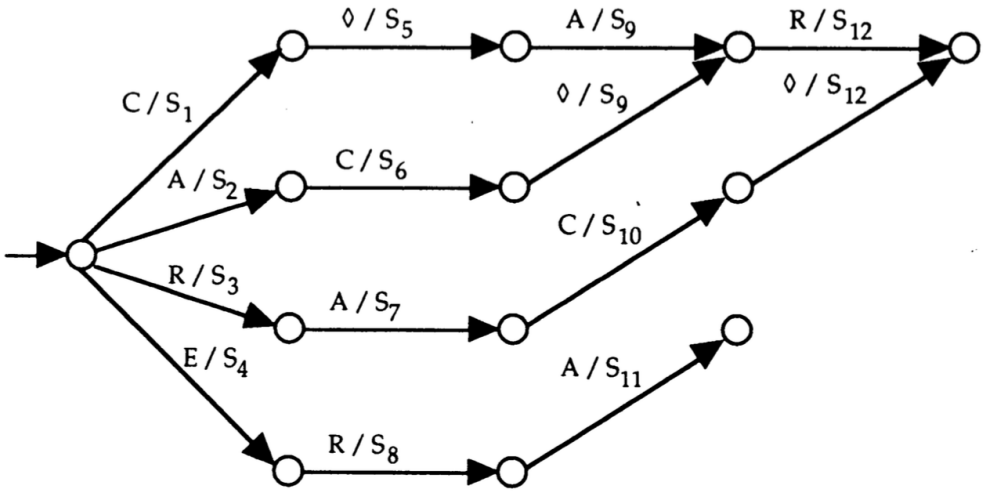

Figure 4.4: Minimized GADDAG structure for the word 'CARE'. Source [12].

Some run-time statistics were recorded for the Java implementation of this data structure that was written for this project. This was run on a 2014 MacBook Pro.

```
FOR each word a1a2…an in the lexicon:
      st ← initialState                          {create path for anan-1…a1:}
      FOR i FROM n DOWNTO 3
           AddArc(st, ai)
      AddFinalArc(st, a2, a1)

      st ← initialState                          {create path for an-1…a1◊an:}
      FOR i FROM n-1 DOWNTO 1
           AddArc(st, ai)
      AddFinalArc(st, ◊, an)

      FOR m FROM n-2 DOWNTO 1      {partially minimize the remaining paths:}
           forceSt ← st
           st ← initialState
           FOR i FROM m DOWNTO 1
                AddArc(st, ai)
           AddArc(st, ◊)
           ForceArc(st, am+1, forceSt)

   AddArc(st, ch) adds an arc from st for ch (if one does not already exist)
      and resets st to the node this arc leads to.

   AddFinalArc(st, c1, c2) adds an arc from st for c1 (if one does not
      already exist) and adds c2 to the letter set on this arc.

   ForceArc(st, ch, fst) adds an arc from st for ch to fst (an error occurs
      if an arc from st for ch already exists going to any other state).
```

Figure 4.5: Pseudo-code for GADDAG construction and minimization. Source [12].

Average Build Time	0.651 seconds
Number of Nodes	2,512,662
Size in Memory	38.6 Mb

Table 4.1: GADDAG construction statistics.

4.5 Conclusions

In this chapter I have discussed some potential ways of storing a lexicon. I assessed each based on its advantages and disadvantages and how suitable each was for this project. Ultimately I decided to use the GADDAG data structure presented by Gordon since it was specifically developed for Scrabble and offers rapid word generation [12]. In next few chapters I will discuss how the computer can use this data structure to generate a complete list of possible moves.

Chapter 5

Assessing the Board

5.1 Introduction

One of the most important parts in creating an AI to play Scrabble, or any game for that matter, is giving it the ability to properly assess the current game state. This is something that we as humans can often take for granted. When a normal human player looks at a Scrabble board it is easy for us to identify which squares we can try to play a word from or which letters can be played legally on which squares. This however is not so easy for a computer to understand. When provided with any sort of information, a computer needs to be given a set of strict instructions to go about using this information in some useful way. To solve this problem, it is necessary to provide the computer with a set of logical or mathematical guidelines to help it assess the current state of the board. Only then it can go about making and selecting its play. Two important characteristics of the board state that the computer will need to be able to compute are:

1. The location of anchor squares on the board, and

2. The vertical and horizontal cross sets of each board square.

5.2 Anchor Squares

Anchor squares are simply the squares on the board which can be used to "hook" onto an already existing word and make a play. Figure 5.1 shows an example of a board with anchor squares highlighted. Logically we can define a square as being a valid anchor square if:

1. It doesn't currently have a tile placed on it, and

2. One or more of its neighboring squares does have a tile placed on it.

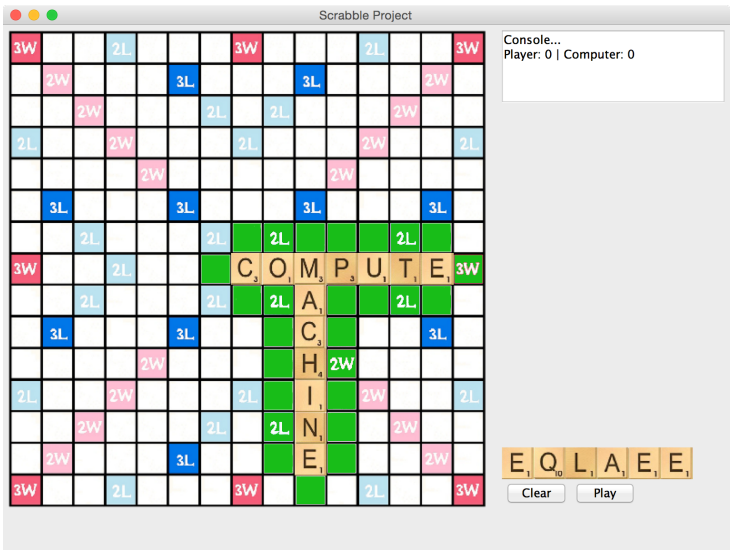

Figure 5.1: Board with anchor squares shown in green.

5.3 Cross-Sets

In Scrabble, when a player plays a horizontal word, that word must also form valid words with any letters on the board it "hooks" onto vertically. The opposite is true for vertical words.

Cross sets can be defined as the sets of individual letters which can be placed on a square to complete a valid word on the board. Each square on the board has a vertical and horizontal cross set, which contain all the letters which can be played to complete words vertically and horizontally, respectively. When playing a horizontal word, from left to right, the computer needs to consider the vertical cross sets of any squares it tries to place a tile on. And the opposite is true when playing a vertical word.

Pre-computing and updating these cross sets makes word generation faster and simpler. Computing cross sets involves traversing the dictionary using the letters already on the board to see which letters could be used to append them and complete a word. After each players turn, the computer only needs to update the cross sets of any new anchor squares generated as a result of the previous play. Any board squares that already have a tile on them should have empty cross sets, (i.e. no letters can be played on them), while any squares that are not anchor squares should have full cross sets, (i.e. any letter can be validly played on them). There are three conditions that need to be accounted for when computing cross sets:

1. When a vertical word can be appended with a letter to create a valid vertical word.

2. When a horizontal word can be appended with a letter to create a valid

horizontal word.

3. When a letter can be placed between two existing words, either vertically or horizontally, bridging them and creating a new valid word.

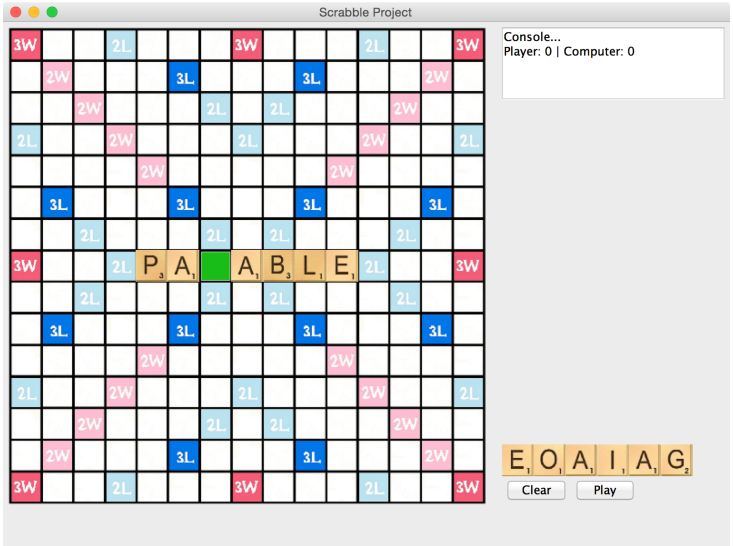

Figure 5.2: Example of cross sets.

Figure 5.2 shows an example of what is meant by cross sets. The green square on the board is an anchor square, so horizontal and vertical cross sets are calculated for it. The vertical cross set will be full since there are no tiles directly above or below the square. Thus, any letters can be played to form vertical words. The horizontal cross set however will contain just two letters, “R” and “Y”. According to the game dictionary, there are the only two letters that can be placed between the words “PA” and “ABLE” to complete valid words, “PARABLE” and “PAYABLE”.

5.4 Conclusions

This chapter has discussed some of the preliminary functions that need to be carried out before the computer can begin searching for moves in Scrabble. It explained how two key parameters: anchor squares and cross sets can be identified. The next chapter will discuss how words are found once these two parameters have been calculated.

Chapter 6

Word Generation

6.1 Introduction

Generating a list of all possible plays given a certain board state and set of rack letters is one of the core parts to any Scrabble playing computer program. Unlike humans, computers need a strict set of instructions to follow in order to carry out even the simplest of tasks. Thus, it can be a very complicated task to program or show a computer how to find playable words in Scrabble. However, computers have the advantage that, once shown how to find words, they will carry this out both quickly and flawlessly, never missing a possible word. This chapter discusses the algorithm that was used in this project to achieve this task.

6.2 Word Finding Algorithm

Gordon provides the following pseudo-code for his word generation algorithm:

```
Gen(pos, word, rack, arc):                {pos = offset from anchor square}
   IF a letter, L, is already on this square THEN
      GoOn(pos, L, word, rack, NextArc(arc, L), arc)
   ELSE IF letters remain on the rack THEN
      FOR each letter on the rack, L, allowed on this square
         GoOn(pos, L, word, rack - L, NextArc(arc, L), arc)
      IF the rack contains a BLANK THEN
         FOR each letter the BLANK could be, L, allowed on this square
            GoOn(pos, L, word, rack - BLANK, NextArc(arc, L), arc)

GoOn(pos, L, word, rack, NewArc, OldArc):
   IF pos ≤ 0 THEN                                          {moving left:}
      word ← L ‖ word
      IF L on OldArc & no letter directly left THEN RecordPlay
      IF NewArc ≠ 0 THEN
         IF room to the left THEN Gen(pos-1, word, rack, NewArc)
         NewArc ← NextArc(NewArc, ◊)                      {shift direction:}
         IF NewArc ≠ 0, no letter directly left & room to the right THEN
            Gen(1, word, rack, NewArc)
   ELSE IF pos > 0 THEN                                     {moving right:}
      word ← word ‖ L
      IF L on OldArc & no letter directly right THEN RecordPlay
      IF NewArc ≠ 0 & room to the right THEN
         Gen(pos+1, word, rack, NewArc)
```

Figure 6.1: Pseudo-code for GADDAG word generation algorithm. Source [12].

This algorithm was implemented as two sets of Java functions. One set found horizontal words while the other set found vertical ones. Both functions were called for each anchor square on the board and appended any moves found to a common list.

Figure 6.2 shows four different ways the word “CARE” can be played off the word “ABLE”. If we look at example b) we see that the algorithm starts by hooking onto the already existing “A” in “ABLE”. Then the letter “C” is extended to the left as part of a prefix. Next the computer finds the delimiting character, “@”, indicating that “CA” is a valid prefix. Next the computer starts to construct the suffix starting from the right hand side of the letter “A” by placing the letters “R” and then “E”. The node in the GADDAG that corresponds to the final letter “E” will have “E” as a valid word ending letter indicating the end of the word “CARE”.

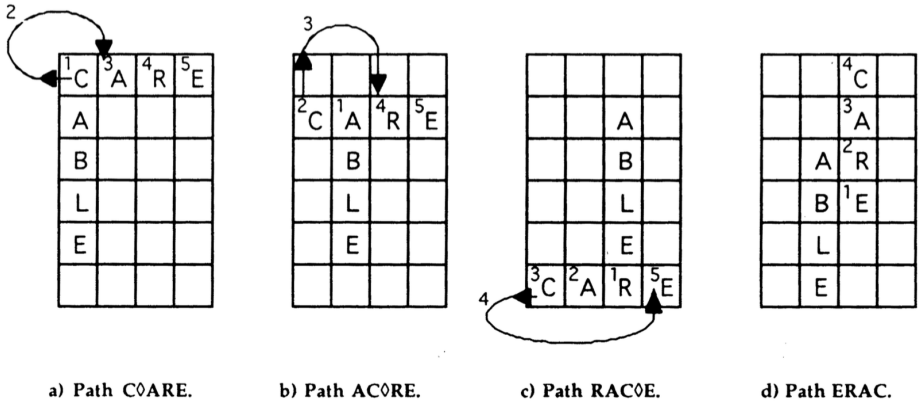

Figure 6.2: Example of four ways the word “CARE” can be played off the word “ABLE” using the GADDAG data structure. Source [12].

6.3 Unit Testing

Software testing is the process of validating and verifying that a program or piece of code works as expected. It is a key part of the software development process and is useful in detecting bugs in the code. It is usually carried out by executing the program, or part of the program, and examining the output. Usually if the output conforms with some expected result this is an indication that the code works correctly.

Unit testing is a method of software testing that involves writing test cases for each component, or group of components, that makes up the program. It is a useful way of automating much of the software testing process and allows the program to be easily validated when small changes are made. A testing methodology based on unit testing was adopted for this project. Specific test cases were written for most of the components and major functions that made up the overall program. However, given the enormous sample space involved in testing a game of Scrabble (up to 100 tiles arranged onto 225 board squares) it is difficult to think

of and write test cases for all the situations that could occur. Unit testing was carried out mainly for the functions that computed cross sets and generated playable words. In order to reduce the sample space to something more manageable short test dictionaries of no more than 5 words were used when testing. Ideally for a program such as this, in order to guarantee the program works correctly, some form of beta testing would be carried out. However carrying out such testing is beyond the scope of this project.

6.4 Speed and Performance

Gordon's work has already showed that the GADDAG data structure can be used to facilitate much faster word generation over typical methods, including the DAG [12]. In a large part this is due to the fact that using this method eliminates words which cannot be played as soon as possible. Typical methods, such as using lists or hash tables, take each word and attempt to play it in each board position. This results in a lot of computation time wasted in trying to play words that can't actually be played. Gordon's method attacks the problem in a different manner, assessing the board for what letters can be played legally in what positions. For each letter Gordon's algorithm tries to play, it first checks the cross sets of the board square and the GADDAG to see if it is a valid branch. If one of these checks fails the branch of the search is eliminated immediately.

Using Big O Notation we can attempt to describe the performance or complexity of Gordon's algorithm. This is typically done by assuming the worst case scenario and expressing the complexity of this case. Normally the search algorithm will finish searching a branch early if one of the following conditions are met:

- The next letter cannot be placed because it is not in the next squares corresponding cross set.
- The next letter cannot be placed because it is not a valid branch of the current GADDAG node.
- There is no room on the board to continue constructing a word.

If we assume none of these conditions are met the algorithm's time complexity will be maximized. Under these conditions the search will be able to use all 7 rack letters in any permutation, (in reality this will produce words that do not actually exist). This will result in 7! = 5,040 full branch searches. I have not taken letter tiles already on the board into consideration to somewhat simplify the situation. Thus, we can say the complexity is proportional to R!, where R is the maximum number of letters in a players rack (typically 7, however some variations of Scrabble can use 6 or even 8). This search is carried out for each anchor square, A, currently on the board. The value of A varies wildly during the game. Thus, the overall time complexity of carrying out a full word search can be expressed as:

$$O(AR!)$$

where:

A = # of anchor squares.

R = # of tiles in a players rack.

Using Gordon's algorithm the word generation statistics shown in Table 6.1 were collected:

Average Search Time	26.67 milli-seconds
Average Number of Moves Found	304 moves
Average Highest Move Score	34.6 points

Table 6.1: Word Generation Statistics.

6.5 Conclusions

This chapter has discussed the algorithm that was implemented to allow the computer to generate a list of possible moves. It analysed the performance of this implementation and explained how some unit testing was carried out. In the next chapter I will talk about how the computer selects a move from this list of possible moves.

Chapter 7

Move Evaluation

7.1 Introduction

A computer controlled Scrabble AI has a distinct advantage over any human player in that it will never miss a possible word or bingo. It has been identified that vocabulary is one of, if not the biggest contributing factor to a players average score [17]. Even expert level players are prone to missing or overlooking a word which can cost them points.

Playing the highest scoring possible move all the time will beat most non-expert Scrabble players. However total command of the lexicon, like an AI would possess, without some sort of strategy is not enough for expert play [11]. Thus, to achieve expert level play, a Scrabble AI needs to have an intelligent strategy when it comes to move selection. There are two main methods which are typically used to evaluate moves in Scrabble. These are:

1. A heuristic function, and

2. Probabilistic search

Many expert players agree that some of the most important factors to consider when evaluating a move include:

- Word score
- Rack leave
- Board leave

7.2 Heuristic Function

For the most part, Scrabble is a game of incomplete information. That is, at most stages in the game, neither player knows what tiles or potential moves the opposite player has at their disposal. Similarly, neither player knows what tiles will be drawn from the letter bag next. This element of randomness exists in the game until the final few moves, known as the "end game" phase, when there are no more tiles to be drawn. At this point each player can deduce what tiles their opponent has by looking at the board and their own tiles.

Given this, in a game of Scrabble it often is difficult to say with absolute certainty that one move is better than another. Just because one move offers a higher immediate point yield does not mean it will do so in the long run. Often in Scrabble, playing lower scoring words will actually result in a higher relative point return over a number of turns.

A heuristic function is a method that uses heuristics to rank alternatives in order to decide which option to follow. In computer science, a heuristic is a kind of algorithm that uses loosely applicable information to provide a "good" solution to a problem. Although a heuristic might provide a good solution there is no guarantee that the solution is correct. It can be thought of as a general "rule of thumb" or an "educated guess" to a problem. Heuristics are useful in situations where finding an optimal solution is impossible or impractical, or where a pretty good solution will suffice. Thus, this method lends itself well to evaluating Scrabble moves.

Heuristics is something that expert Scrabble players also take into consideration when deciding on their next move. For example, consider the case where a player has the "Q" tile on their rack. As the joint highest scoring tile (tied with "Z" with 10 points), this tile has the potential to create high scoring plays. However, playing the "Q" tile without a "U" can prove difficult given the small number of "Q" words without the "QU" combination. The question they are faced with then is, Do I get rid of or hold onto the "Q" tile? Heuristics can be calculated and used to help give some insight into a "good" solution to this problem.

7.2.1 Weighted Heuristics

Stephen Gordon has addressed the idea of using weighted heuristics for move evaluation in Scrabble [11]. Using this method involves evaluating a players rack leave by assigning numerical values to each of the letters, and duplicate letters, left behind after a move. In his paper Gordon includes suggested weights to be applied to each instance of a letter and how to calculate the overall heuristic of a rack leave. He also mentions a method to incorporate weights to address the idea

of a good vowel - consonant[1] mix for each rack.

Letter	Heuristic1	Heuristic2		Letter	Heuristic1	Heuristic2	
A	+0.5	+1.0	−3.0	B	−3.5	−3.5	−3.0
C	−0.5	−0.5	−3.5	D	−1.0	0.0	−2.5
E	+4.0	+4.0	−2.5	F	−3.0	−2.0	−2.0
G	−3.5	−2.0	−2.5	H	+0.5	+0.5	−3.5
I	−1.5	−0.5	−4.0	J	−2.5	−3.0	
K	−1.5	−2.5		L	−1.5	−1.0	−2.0
M	−0.5	−1.0	−2.0	N	0.0	+0.5	−2.5
O	−2.5	−1.5	−3.5	P	−1.5	−1.5	−2.5
Q	−11.5	−11.5		R	+1.0	+1.5	−3.5
S	+7.5	+7.5	−4.0	T	−1.0	0.0	−2.5
U	−4.5	−3.0	−3.0	V	−6.5	−5.5	−3.5
W	−4.0	−4.0	−4.5	X	+3.5	+3.5	
Y	−2.5	−2.0	−4.5	Z	+3.0	+2.0	
BLANK	+24.5	+24.5	−15.0				

Figure 7.1: Two sample sets of heuristics for Scrabble rack leave evaluation. Source [11].

Figure 7.1 shows two sets of heuristics suggested by Gordon [11]. Heuristic2 was used for the purpose of this project as the authors results suggested they performed better. This heuristic has a value associated with each letter and each duplicate letter left on a players rack after a move. The value associated with duplicate letters is reapplied for each duplication. For example the rack leave “II-ISS” would be valued as:

$$-0.5 + (-0.5 - 4.0) + (-0.5 - 4.0 - 4.0) + 7.5 + (7.5 - 4.0) = -2.5$$

Figure 7.2 shows the vowel - consonant ratio values that were suggested [11]. This value is simply added to the value obtained using Heuristic2. For example the rack leave “IIISS” has a vowel - consonant ratio of 3:2. The value associated with this ratio is 1 giving an overall rack leave score of -2.5 + 1.0 = -1.5.

[1]the vowel - consonant mix is the idea that having too large or small a ratio of vowels to consonants in a rack leave can hinder a player in making future moves. Many experts agree that players should aim to have a ratio close to 1:1 or slightly more in favor of consonants at 3:2.

		CONSONANTS						
		0	1	2	3	4	5	6
V	0	0	0	−1	−2	−3	−4	−5
O	1	−1	1	1	0	−1	−2	
W	2	−2	0	2	2	1		
E	3	−3	−1	1	3			
L	4	−4	−2	0				
S	5	−5	−3					
	6	−6						

Figure 7.2: Suggested Vowel - Consonant ratio table. Source [11].

Although a weighted heuristics method provides a good solution to move evaluation, it is still prone to strategic mistakes or making serious errors in judgement. This is because weighted heuristics can only provide a "general" good solution. It cannot account for positional factors related to the current state of the board. What would normally be a good move may not be so in certain situations. Consider the example shown in 7.3. Playing the word "COMPUTE" may yield a high point return. However it also leaves an opening to append this with a vertical word containing an "R", making "COMPUTER". This would result in the opponent getting two triple word scoring words in one move as shown in 7.4.

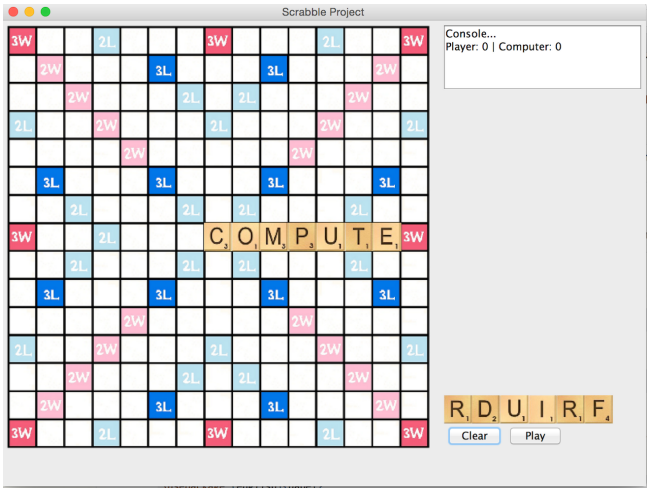

Figure 7.3: Weighted heuristics leaving a good opening for an opposition move.

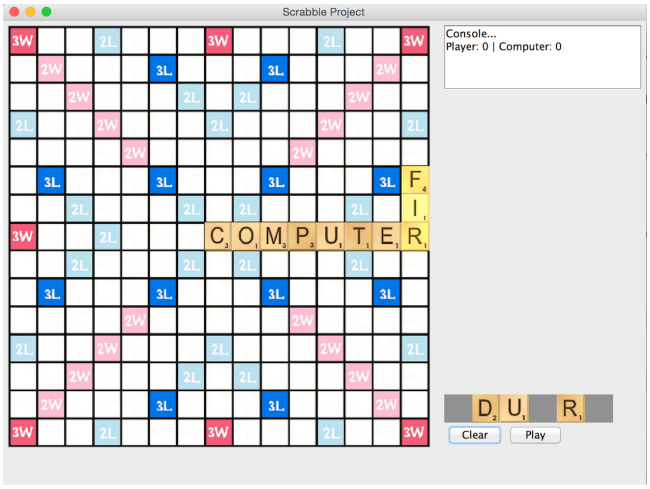

Figure 7.4: Counter move.

7.2.2 Machine Learning Algorithms

Machine learning is a field of computer science that is associated with AI, and is also closely related to the field of computational statistics. It primarily deals with the development and study of algorithms that can learn from and make predictions based on some sample data. Such algorithms can develop a model of a system given data inputs in order to make future decisions or predictions.

In order to use a weighted heuristics method for word evaluation, we first need some way of obtaining values for these heuristics. Machine learning offers a way of doing this by allowing optimum values for each heuristic to be "learned" by the computer.

Gordon carried out a simple form of machine learning to arrive at his weights:

1. Estimate reasonable values for each weight.
2. Find the optimal value for each weight by:
    (a) Adjusting the weights in increments of 2.0 until performance in 1,000 games is optimized against a greedy opponent.
    (b) Repeating (a) with increments of 0.5 over 10,000 games.

7.3 Probabilistic Search

Probabilistic search is a method for Scrabble move evaluation that attempts to address some of the limitations of weighted heuristics.

A probabilistic search works by taking the current game state and carrying out simulations to try and predict likely outcomes over the next few moves. It can be used to try and predict how the game will play out over the next N moves and estimate the point distribution associated with different candidate moves. As the depth of the search increases however, the computation time grows exponentially. Thus, this method can only really be applied practically over a small number of iterations.

Below is a brief outline of how such a method would be carried out:

1. Find and chose "C" candidate move to be considered for play. Moves can be chosen using a greedy algorithm, heuristics or some other method.

2. For each move, generate "S" random scenarios. Each scenario consists of a random rack for the opponent and a random draw of tiles to replace the letters used by each candidate move.

3. Play out each candidate move in its respective scenario by playing the candidate move. Then have the opponent play its "best" move and have the first player play its best "comeback" move.

This method can be continued up to a depth of N plays. When the search has completed, each branch ending will have a value associated with it to represent the predicted relative point distribution. This value is calculated as, Score = Player move - Opponent comeback move + Player move... Each candidate move is evaluated based on the average return of its branch in the search tree. By using this method on a large enough sample space it is intended that positional factors such as the one discussed in section 7.2.1 can be taken into account and avoided.

7.4 Weighted Heuristics and Probabilistic Search

I have already mentioned that weighted heuristics as a method for move evaluation can be prone to strategic errors. I have also explained how carrying out large probabilistic searches isn't a very viable option due to its large computational complexity. The idea of combining the two methods has been mentioned in other papers [7]. In theory, combining the two methods should result in a move evaluation function that is both quick and more reliable (i.e. less prone to strategic errors than weighted heuristics on its own). Gordon carried out a brief comparison of the two methods [11]. His results showed that weighted heuristics outperformed probabilistic search but suggested that combining the two in some way could be fruitful. He suggests using heuristics to narrow down the search space of a full probabilistic search. How and when to combine the two could be an interesting starting point for future research.

7.5 Conclusions

In this section I have discussed two of the main methods that can be used for move evaluation in Scrabble. I mention the advantages and disadvantages of each and how they could be combined to create an improved evaluation function. In the next chapter I will discuss the idea of augmenting the AI to create one that behaves more realistically or human-like.

.

Chapter 8

Artificial Intelligence and Realism

8.1 Introduction

MAVEN famously showed that computers had reached the point where they could out perform even the worlds best human players at games such as Scrabble [17]. In 1997, IBM's supercomputer "Deep Blue" went on to defeat the reigning world chess champion, Garry Kasparov, under standard tournament conditions [13]. At this point it had become clear that computers had matched, and even surpassed human intellectual capabilities, and many people debated whether Deep Blue had achieved true artificial intelligence.

In recent years there has been a growing emphasis put on the development of robotics and AI programs that behave more like humans. In AI for games, this involves developing programs that don't necessarily perform at the optimum level and are even sometimes prone to human-like "errors" in judgement. Implementing such behaviour in a computer program adds many new elements of

complexity. The computer still needs to be capable of performing the task at hand optimally while also introducing poor decision making in some way. This can be implemented easily by introducing an element of randomness to the decision making process. However human errors are not necessarily random in nature, thus this method, while it does incorporate apparent 'errors', may still not be considered to be human-like.

Playing against an optimally performing AI may pose an interesting challenge for expert level Scrabble players. However, for most average or casual Scrabble players, such an opponent would be a daunting experience. More importantly, playing against a Scrabble AI or expert Scrabble player would be quite unlike playing against another casual player. In tournament level Scrabble it has been identified that [11]:

> *"To the casual observer, the most apparent characteristic of tournament play is the many obscure words that are played (e.g. OE, QAT and ZEMSTVO)."*

Similarly, simply playing the highest scoring possible move each turn will consistently beat most non-expert Scrabble players [11]. Thus, the idea of introducing AI realism into this project attempts to create a Scrabble AI that is more on par with a typical, non-expert level, player.

8.2 The Turing Test

How human like an AI program behaves is an entirely subjective measure. That is, it cannot be defined, measured or quantified by some objective test. The Turing

test was first proposed by Alan Turing in his 1950 paper "Computing Machinery and Intelligence" [20]. It is a test of a machines ability to exhibit intelligent behaviour which is equivalent to, or indistinguishable from, that of a human. Figure 8.1 illustrates how the Turing test is carried out. The test is normally interpreted as follows:

- The test requires 2 test subjects, A and B, and a human interrogator. Of the 2 test subjects, one is human and the other is a computer.
- The interrogator, C, is not told which test subject is human or computer. The test subjects and interrogator are all separated and can only communicate through written communication.
- The interrogator is tasked with determining, through a line of questioning, which test subject is human and which is the computer. If the interrogator cannot reliably tell the computer from the human, up to 70% of the time, the computer is said to have passed the test.

8.3 BotPrize Competition

A modern day version of the Turing test has been sponsored by video games developer 2K Games since 2008. The "BotPrize" competition challenges contestants to create a computer-controlled bot to compete in the video game "Unreal Tournament 2004" (UT2004) [2]. UT2004 is a first-person shooter game that has quickly become an established platform for AI academic research and simulation.

In the competition, bots and human players (who also act as judges) take part in multiple rounds of combat. As well as playing the game, the judges must

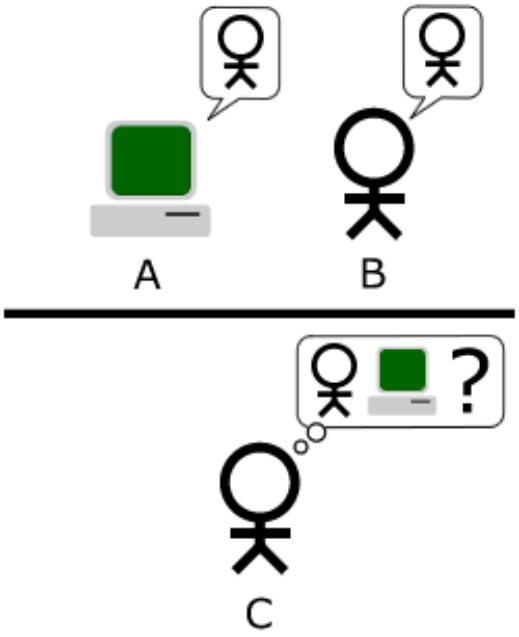

Figure 8.1: The “standard interpretation” of the Turing Test, in which player C, the interrogator, is given the task of trying to determine which player A or B is a computer and which is a human. The interrogator is limited to using the responses to written questions to make the determination.

try and guess which opponents are human, and which are not. To win the prize, a bot has to be indistinguishable from a human player more than 50% of the time. In 2012 two teams managed to achieve this and split the winning prize of $7000 [3].

A common tactic used to create more human-like behaviour in AI applications is to directly mimic observed human behaviour. This can be done using both real time mimicking and replicating past behaviour. Some of the teams that won the BotPrize used this simple tactic to achieve a more realistic AI.

8.4 A Realistic Scrabble AI

As mentioned previously, measuring how realistically a computer controlled opponent plays a game of Scrabble is a very subjective measure. An interesting way of judging such an AI would be to carry out an experiment similar to the Turing test or BotPrize competition. A human Scrabble player, preferably a non-expert player, would be pitted against another player and a bot in two games of Scrabble. This person would also be tasked with judging which test subject is human and which is a bot. They would have no indication which is human or bot and would judge this solely on the basis of each opponents move choice.

When it comes to actually developing a more realistic AI for Scrabble however, mimicking the human player in real time is not really a valid option. Thus, any attempt to copy human behaviour will have to be based on past data. Ideally one would want the computer to favor playing more commonly used, or well known words, over some of the higher scoring obscure words it has in its dictionary. In order to achieve this, the AI needs access to a source of data that it can use to determine which words are more well known than others to humans. The ideal data source for this would be a database of word frequency statistics drawn from real games of Scrabble played by actual human players. Unfortunately, no such database exists today, and gathering these statistics is way beyond the scope of this project. There do however exist two databases of English word frequencies obtained from other sources. These are:

1. The Corpus of Contemporary American English [6], and
2. Google's N-Gram Viewer.

The Corpus of Contemporary American English is a web database con-

taining detailed word frequency data from the "Corpus of Historical American English", the "British National Corpus", and the "Corpus of American Soap Operas". Access to this database however requires payment. Therefore, for the purpose of this project, I will be using Google's N-Gram Viewer which is free and open source.

Incorporating word frequencies in the move evaluation function is quite a simple process. It is simply a matter of including another parameter in the heuristic function already used by the AI. The word frequency of each candidate move is read from a ".txt" file and normalised by dividing by each by the highest word count of all the candidate words. This can then be multiplied by a multiplier, M, to adjust the weight or importance given to this "realism" parameter. Selecting an ideal weight for this multiplier is tricky. We don't want it so high that it overshadows all other heuristics. But we also don't want it so small that it has no impact on the outcome of word evaluation. The current range of the heuristic function is between -37, for a rack leave of "QWWYYV", and +46.5, for a rack leave of "ES*XZA"[1]. Thus, for the purpose of this project, I think a multiplier of around 40, and no more than 80, is a good starting value. Ideally this value could be modified and experimented with by the player.

8.5 Google's N-Gram Viewer

Google's N-Gram Viewer is a database which contains the frequency of occurrence of different n-grams[2] found in printed sources taken from 1800 to 2012.

[1] * indicates a blank tile.

[2] An n-gram is a contiguous sequence taken from text or speech. This project is only concerned with 1-grams, or individual words.

Users can use Google's online viewer to graph the frequency of different words and phrases over a given time period [1]. The viewer was initially based on Google Books, a collection of over 30 million books and magazines scanned and digitized by Google, spanning hundreds of languages. The N-Gram Viewer's database was compiled using about 8 million of these, or 6% of books ever published [14].

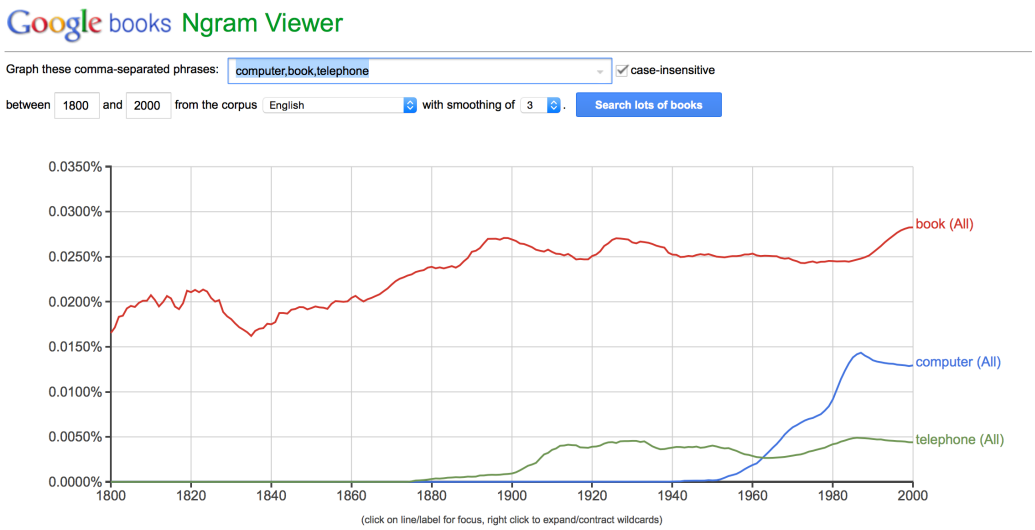

Figure 8.2: A sample query using Google's N-Gram Viewer. The graph shows the frequency of occurrence of the words, "book", "computer" and "telephone" between the years 1800 and 2000.

The justification for using this database as a source for creating a more realistic AI in Scrabble is based on a simple assumption: Words that appear more often in books and magazines are more likely to be commonly known by people. This seems like a simple and obvious enough assumption and provides a good starting point to try and create an AI that favors more well known words. Languages are also known to evolve and change over time. Thus, to get a modern day representation of the English language, only sources published from 1950 onwards were used in this project.

Queries to Google's N-Gram Viewer can be automated in a script using a HTTP "GET" request. However in doing so, only 12 queries can made at a time, and an IP is blocked out for 5 minutes if it sends too many requests within a that time interval. Thus, the easiest and quickest way of extracting the relevant word data for this project was to download the entire library of 1-gram data from [4]. Google offers its entire alphabetised n-gram database for download so that users can conduct their own large scale language research on a local machine. Downloading the entire alphabetical, A - Z, 1-gram library comprised about 25Gb of raw text data. This was parsed for relevant words using a Python script and condensed into a small text file with a word count for each word in the given Scrabble dictionary. Some words in the Scrabble dictionary did not appear in the database under the specified search and were given a count of -1 to signify this. This same procedure could be easily carried out using different languages at a later date.

8.6 Conclusions

In this chapter I have discussed the idea of creating a Scrabble program that plays more like its human counterpart. A method of implementing and evaluating a more realistic AI was proposed. Evaluating the effectiveness of this method could be an interesting future project. In the final chapter I will reflect on the work I have done and some of my results and findings.

Chapter 9

Conclusions

9.1 Introduction

Over the past eight chapters I have discussed how I implemented a computer controlled Scrabble AI. In this chapter I will reflect on the work done, my own achievements, and discuss some ways the this project could be expanded or improved.

9.2 Results and Findings

I implemented my Scrabble game using the GADDAG data structure developed by Steven Gordon [12]. For move evaluation I implemented a simple heuristics function and borrowed some of his suggested values. This resulted in a Scrabble program that was both extremely quick and performed well. Table 4.1 shows some statistics of how fast the program was as well as its average point yield. For the

most part, this implementation beat me in Scrabble quite comfortably.

9.3 Achievements

Overall I have achieved much of what I set out to do at the start of this project. I have successfully implemented a Scrabble game in Java and developed a computer controlled AI to play against. Over the course of the project I have learned a lot about software development, algorithms and data structures. I had very little experience with Java before undertaking this project. Doing this project has forced me to expand my knowledge of Java which I think will benefit me greatly. I have also learned quite a lot about Scrabble strategy and believe I have improved my own game.

9.4 Critical Analysis

Although my implementation worked quite well there were however some drawbacks with it. When given a much larger tournament Scrabble dictionary, containing some 270,000 words, the algorithm to construct the GADDAG quickly ran out of system memory and threw an "OutOfMemoryError" error. This could be fixed by explicitly assigning the program more than the default amount of memory to run. It would appear that although the constructed GADDAG data structure is quite small, the amount of memory required for "bookkeeping" of temporary variables during its construction is rather large. A typical Java memory heap is 256 MB in Eclipse. Increasing this to 512 MB solved the problem. However, since system memory is a finite resource, this is not a good fix in general. Perhaps

the algorithm can be modified to be more memory efficient?

9.5 Future Work

Ideally I would have liked to implement and experiment with some of the more advanced ideas I have discussed in this report. Unfortunately there was not sufficient time to do so. This project however lays the ground work for some interesting future research and experimentation. There are numerous ways this project can be built upon or expanded, some of which I will mention now.

9.5.1 Probabilistic Search

Implementing and experimenting with probabilistic search as a method of move evaluation would be an interesting area for future work. Investigating how and when to use this method would be a good starting point for any future projects. Combining this method with weighted heuristics could also produce some interesting results.

9.5.2 Tuning Heuristic Weights

The heuristic weights used in this project were taken from a previous papers findings [11]. These weights were found to be optimal using increments of 0.5. However, perhaps calculating weights using increments of 0.1 or smaller would produce more optimal weights. It stands to reason that the higher the precision of the weights, the more accurately they would reflect the true optimal weight values.

9.5.3 Game Functionality

The overall game functionality could be improved and expanded. Given the time constraints of this project, some features that would normally be present in Scrabble were omitted. These include features such as blank tiles and swapping tiles. The game could also be expanded to allow up to 4 players, including multiple AI controlled players.

9.5.4 Web Based and Mobile Platform

Introducing a web based version of this program could be part of a future project, similar to the one done previously [15] [10]. Given the speed and efficiency of the program it could also be implemented on a mobile platform.

9.5.5 Assessing and Improving Realism

A method of making the AI more realistic has been proposed in this project. Assessing this method would require some type of subjective testing which could be part of a future project. Similarly gathering an alternative source of word frequencies could be a part of this. Gathering statistics from games between human players to improve realism would be an interesting idea and could be efficiently done if players could play online as per the previous suggestion.

9.6 Summary

The outcome of this project was a Java implementation of Scrabble that can be played by a computer controlled AI. The resulting program was both fast and chose moves strategically well. A method and justification for improving realism in this AI was outlined. This method would require further testing to evaluate its success. Overall this project implemented some of the current state of the art in Scrabble AI, while also exploring a new element of realism within this field.

Bibliography

[1] Google's n-gram viewer. *https://books.google.com/ngrams*.

[2] The 2k botprize: Can computers play like people? *http://botprize.org*, Verified Jan. 2015.

[3] Artificially intelligent game bots pass the turing test on turing's centenary. *http://news.utexas.edu/2012/09/26/artificially-intelligent-game-bots-pass-the-turing-test-on-turings-centenary*, Verified Feb. 2015.

[4] Google's n-gram viewer raw database. *http://storage.googleapis.com/books/ngrams/books/datasetsv2.html*, Verified Mar. 2015.

[5] Official scrabble players dictionary source. *http://www.puzzlers.org/pub/wordlists/ospd.txt*, Verified Feb. 2015.

[6] Word frequency data: Corpus of contemporary american english. *http://www.wordfrequency.info*, Verified Feb. 2015.

[7] A. W. Appel and G. J. Jacobson. The world's fastest scrabble program. *Communications of the ACM*, 31(5):572–578, 1988.

[8] J. Cosma and D. Jackson. Introducing monty plays scrabble. *Scrabble PIayers News,(June 1983)*, pages 7–10, 1983.

[9] D. . Deitel. *Java, How to Program (4th edition).* Prentice Hall International, Inc., 2002.

[10] S. Gibbons. An extension of corpus based scrabble. *Final Year Project, Trinity College Dublin*, 2003.

[11] S. Gordon. A comparison between probabilistic search and weighted heuristics in a game with incomplete information. In *AAAI Fall 1993 Symposium on Games: Playing and Learning, AAAI Press Technical Report FS9302, Menlo Park, CA*, 1993.

[12] S. A. Gordon. A faster scrabble move generation algorithm. *Software: Practice and Experience*, 24(2):219–232, 1994.

[13] D. King. Kasparov vs. deeper blue: The ultimate man vs. machine challenge. 1997.

[14] Y. Lin, J.-B. Michel, E. L. Aiden, J. Orwant, W. Brockman, and S. Petrov. Syntactic annotations for the google books ngram corpus. In *Proceedings of the ACL 2012 system demonstrations*, pages 169–174. Association for Computational Linguistics, 2012.

[15] T. S. Lopez. Corpus based scrabble. *Final Year Project, Trinity College Dublin*, 2002.

[16] S. C. Shapiro and H. R. Smith. *A Scrabble crossword game-playing program*. Springer, 1988.

[17] B. Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241–275, 2002.

[18] P. Turcan. Computer scrabble. *SIGArt Newsletter*, 76:16–17, 1981.

[19] P. Turcan. A competitive scrabble program. *SIGArt Newsletter*, 80:104109, 1982.

[20] A. M. Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.