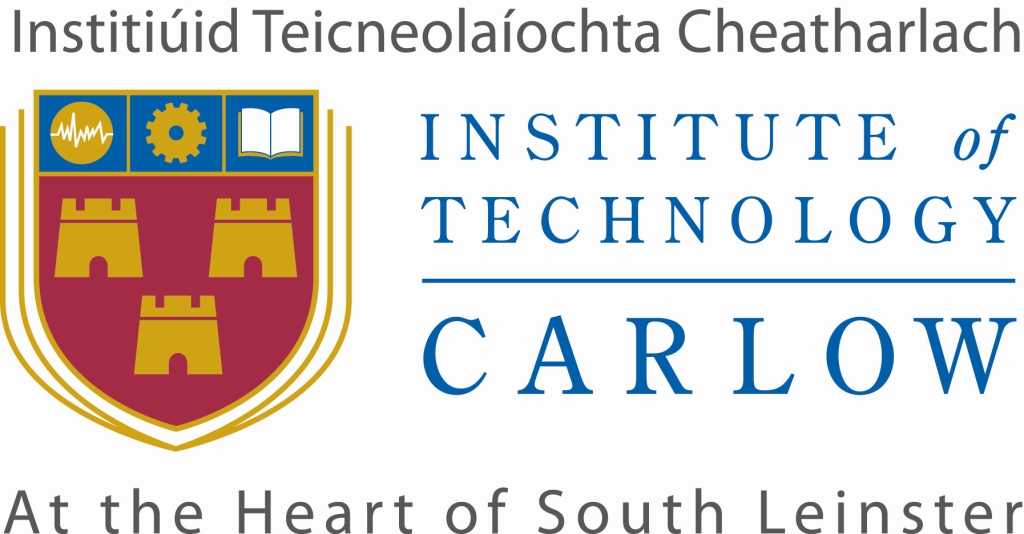
|  |
| --- |
| Institute of Technology Carlow, Kilkenny Road, Carlow, Co. Carlow |
| Scrabble Engine |
| Final Report |



Authors: Keith Byrne (C00170460)

Liam Strevens (C00110605)

Date: 3 March 2016

Supervisor: Dr. Greg Doyle

Table of Contents

[2 Project Overview 1](#_Toc444779606)

[3 Core Algorithms 2](#_Toc444779607)

[3.1 Trie - Load Words 2](#_Toc444779608)

[3.2 Trie - Single Word Search 2](#_Toc444779609)

[3.3 Trie - Search by String (Many Results) 2](#_Toc444779610)

[3.4 AI Perform Turn 4](#_Toc444779611)

[3.4.1 Overview 4](#_Toc444779612)

[3.4.2 Design 4](#_Toc444779613)

[3.4.3 Code 5](#_Toc444779614)

[4 GUI Algorithms 10](#_Toc444779615)

[4.1 Submit Word 10](#_Toc444779616)

[4.2 Place Word 10](#_Toc444779617)

[4.3 Place Tile 10](#_Toc444779618)

[4.4 Select Tile 10](#_Toc444779619)

[4.5 Show Possible Placements 10](#_Toc444779620)

[5 Other Algorithms (Notable Mentions) 10](#_Toc444779621)

[6 Performance Analysis 11](#_Toc444779622)

[6.1 Initial Selection 11](#_Toc444779623)

[6.2 Candidate Selection 12](#_Toc444779624)

[6.3 Algorithmic Time (AI) 13](#_Toc444779625)

[7 Change of Approach 14](#_Toc444779626)

[8 Learning Outcomes 15](#_Toc444779627)

[8.1 Algorithmic Selection & Data Structures 15](#_Toc444779628)

[8.2 Scoping 15](#_Toc444779629)

[8.3 Research Techniques 15](#_Toc444779630)

[9 Conclusion 16](#_Toc444779631)

[10 References 17](#_Toc444779632)

[11 Work Division Declaration 19](#_Toc444779633)

# Project Overview

Scrabble is a word based game that is played by two or more players, typically with a maximum of 4 to 6, depending on the allocation of character tiles. The basic constraints involved in Scrabble are that entries must be played on the 15x15 grid play area and flow left to right as rows or downwards as columns and entries must be valid Scrabble words, not including nouns.

To construct a system to analyse and play Scrabble in the same fashion as a human player. This will require the use an efficient data structure and subsequently efficient algorithms for on demand loading and structuring of data, sorting and searching and potentially other uses [1] [2] [3].

# Core Algorithms

## Trie - Load Words

This algorithms uses the node class to occupy the Trie with the game word values. Each node has a value and a list of pointers to successive child nodes. When a word is taken into the algorithm it checks the first letter to see if the root node contains a matching child. If so, the child becomes the new root. The same process is then repeated for the next letter. If, however, the root does not contain a matching child, it creates a new node with the letter as its value and a pointer in the child array in the current root. This process is repeated until there are no letters left to map.

When the last letter is placed in the trie, it marks that node as a terminator node, this essentially means it is the end of a word. Also if the node has no children it is marked as a leaf in the Trie.

## Trie - Single Word Search

The algorithm starts by sending in the root node and the word into a method. It takes the first letter and checks if a child in the current root contains that value. If it does not contain the value it returns that it is not a valid word. If the root does have a child with the value, the algorithm makes the child the root and moves on to the next letter.

It continues this process until there are no letter lefts in the word. If the last node is a terminator node then it returns true. If it is not a terminator then it returns false from the algorithm.

## Trie - Search by String (Many Results)

This algorithm contains two method, the AISearch method and the AI Algorithm. The first method takes a string as a parameter and creates an empty list for the found words. It sends the root node, found words list, an empty string and the input string into the AI Algorithm method. The list of possible words is returned from that method and that list is return to the calling code.

**Method 1 declaration**

**public** List**<**string**>** AIsearch**(**string key**)**

**Method 2 declaration**

**private** List**<**string**>** AIalgorithm**(**TrieNode**<**V**>** node**,** List**<**string**>** foundwords**,** string currentWord**,** string inputString**)**

The second method uses a pre-order recursive algorithm to search through the Trie data structure. When it first starts it adds the value of the current node to the current word. It then checks if the input string is empty or the node is a leaf. If either of these conditions are satisfied, then it adds the current word to the list of found words and returns the found word list.

Next it checks if the current node is a terminator. This means that a word ends here so it add the current word to the list of found words. The next part of the method is to loop through the input string and check if the current node contains the letter in the loop. If it does contain a letter it send the node that contains the letter, the found word list, the current word and the input string without the current letter.

When the Algorithm is finished it will return the list of found words back to the AISearch method.

This method will recursively search all possible word from a set of letters. On average it will return about 600 words from 8 letter in about 4 milliseconds.

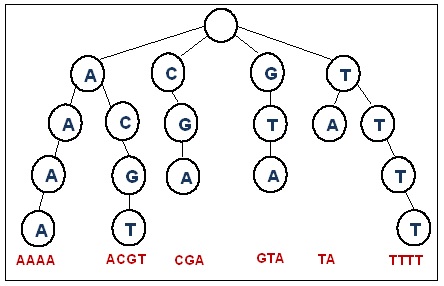


Figure 1 Example Trie Structure

## AI Perform Turn

### Overview

AI Perform Turn is the principal algorithm for handling the CPU turn control. This algorithm provides the core to playability in conjunction with search operations available from the Trie algorithm. The construct of the algorithm is similar to the steps in how a player would typically perform their own turn. However, during initial development, following this methodology created more issues than formally designing the algorithm. Mainly, validation and score calculation efforts before placements proved to be the most efficient strategy.

### Design

*Declare candidate\_entry\_points as List of Board Tiles*

*For each Board Tile x currently on Game Board*

*If x is registered as placed and it’s north tile and south tile is not occupied*

*Add to candidate\_entry\_points Board Tile : x*

*Add to AI’s current list Board Tile : x*

*Formula = Game Mode Chosen [‘Average’, ‘Learning’, ‘…’]*

*For each Board Tile x currently in candidate\_entry\_points*

*For each Board Tile j in AI Tray*

*Add to AI’s current list : j*

*Add to AI’s current list: x*

*AI retrieveSuperSet { Trie Search Algorithm }*

*For each String s in Words { retrieveSuperSet: return Words }*

*Selection = map String to Board Tile*

*If s is higher scoring than current best*

*If placement possible*

*Current best = s*

*For each Fittest scoring word*

*Take best*

*Place word (Includes revalidation and re-tiling contents)*

*Increment score (UI & Game State Logic)*

*Add to played word list (UI)*

### Code

**private** void AIPerformTurn**()**

**{**

List**<**BoardTile**>** joint\_structure **=** **new** List**<**BoardTile**>();**

List**<**BoardTile**>** candidate\_entry\_points **=** **new** List**<**BoardTile**>();**

long begin **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

**foreach** **(**BoardTile bt **in** GameBoard**.**Children**)**

**{**

**try**

**{**

**if** **(**bt**.**accepted\_placement **==** **true** **&&** bt**.**up**.**accepted\_placement **==** **false** **&&** bt**.**down**.**accepted\_placement **==** **false)**

**{**

candidate\_entry\_points**.**Add**(**bt**);**

AI**.**current\_list**.**Add**(**bt**);**

**}**

**else** **if** **(**bt**.**accepted\_placement **==** **true** **&&** bt**.**left**.**accepted\_placement **==** **false** **&&** bt**.**right**.**accepted\_placement **==** **false)**

**{**

candidate\_entry\_points**.**Add**(**bt**);**

AI**.**current\_list**.**Add**(**bt**);**

**}**

**}**

**catch(**Exception e**){}**

**}**

long end **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

long time **=** end **-** begin**;**

List**<**int**>** placed\_char\_index **=** **new** List**<**int**>();**

List**<**String**>** ai\_placement\_buffer **=** **new** List**<**String**>();**

Dictionary**<**KeyValuePair**<**int**,** int**>,** List**<**BoardTile**>>** play\_candidates **=** **new** Dictionary**<**KeyValuePair**<**int**,** int**>,** List**<**BoardTile**>>();**

int index **=** 0**;**

int root **=** 0**;** //Will need later for loc recall

List**<**ComparableSelection**>** list **=** **new** List**<**ComparableSelection**>();**

List**<**BoardTile**>** reserve **=** **new** List**<**BoardTile**>();**

int measure **=** 0**;**

**if(**game\_type**.**name**.**Equals**(**"Learning"**)){**

int average **=** player\_tracker**.**score **/** player\_tracker**.**round**;**

measure **=** average**;**

**}**

**else{**

measure **=** game\_type**.**formula**;**

**}**

begin **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

int marker **=** 0**;**

**foreach(**BoardTile bt **in** candidate\_entry\_points**){**

AI**.**current\_list**.**Clear**();**

**foreach** **(**BoardTile tray **in** AITray**.**Children**){**

AI**.**current\_list**.**Add**(**tray**);**

**}**

//AI.current\_list = reserve;

AI**.**current\_list**.**Add**(**bt**);**

AI**.**retrieveSuperSet**();**

list**.**Add**(new** ComparableSelection**(**0**));**

//Consider word placement from here

//Calculate score, determine strongest

//Option to place, verify and place.

**foreach** **(**String s **in** AI**.**possible\_words**){**

**if** **(**s**.**Contains**(**bt**.**tag**.**letter\_alpha**)){**

//root = s.IndexOf(bt.tag.letter\_alpha);

//Mark buffer as possible word placement

//game\_logic.selection = game\_logic.getTilesFromString(s, bt, root - 1, candidate\_entry\_points, bt.tray\_location);

game\_logic**.**selection **=** game\_logic**.**mapTiles**(**s**,** bt**);**

game\_logic**.**calculateScore**();**

//game\_logic.root\_location = root;

**if** **(**list**[**marker**].**score **==** 0**){**

**if** **(**validatePlacement**(**game\_logic**.**selection**).**Key **==** **true){**

//Before adding to play candidates, check placement validity

list**[**marker**].**current **=** game\_logic**.**selection**;**

list**[**marker**].**score **=** game\_logic**.**score**;**

**}**

**else{**

list**[**marker**].**score **=** 0**;**

**}**

**}**

**else** **if** **(**list**[**marker**].**score **<** game\_logic**.**score **&&** game\_logic**.**score **<=** measure**){**

**if** **(**validatePlacement**(**game\_logic**.**selection**).**Key **==** **true){**

list**[**marker**].**current **=** game\_logic**.**selection**;**

list**[**marker**].**score **=** game\_logic**.**score**;**

**}**

**}**

**}**

**}**

marker**++;**

**}**

end **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

end **=** end **-** begin**;**

int loc **=** 0**;**

int tracker **=** 0**;**

int max **=** 0**;**

//Then get the largest of each root largest

List**<**ComparableSelection**>** seq **=** **new** List**<**ComparableSelection**>();**

begin **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

**foreach(**ComparableSelection cs **in** list**.**Where**(**o **=>** o**.**score **!=** 0**)){**

**if(**cs**.**score **>=** max**){**

max **=** cs**.**score**;**

tracker **=** list**.**IndexOf**(**cs**);**

**}**

loc**++;**

**}**

end **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

end **=** end **-** begin**;**

**this.**cpu\_tracker**.**updateScore**(**list**[**tracker**].**score**);**

int n **=** 0**;**

String output\_word **=** ""**;**

marker **=** 0**;**

List**<**ComparableSelection**>** new\_items **=** **new** List**<**ComparableSelection**>();**

new\_items**.**Add**(**list**[**tracker**]);**

**if** **(**new\_items**.**Count **==** 0 **||** new\_items**[**0**].**score **==** 0**){**

newAITray**();**

**}**

**else{**

begin **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

**foreach** **(**ComparableSelection item **in** new\_items**){**

//int marker = 0;

//Attempt to play word selection

//List<BoardTile> seq = item.current;

**foreach** **(**BoardTile t **in** item**.**current**){**

**if** **(**t**.**accepted\_placement **==** **false){**

int delete\_sequence **=** 0**;**

loc **=** 0**;**

**foreach** **(**BoardTile x **in** AITray**.**Children**){**

**if** **(**x**.**tag**.**letter\_alpha **==** t**.**tag**.**letter\_alpha**){**

delete\_sequence **=** loc**;**

**}**

loc**++;**

**}**

AITray**.**Children**.**RemoveAt**(**delete\_sequence**);**

**}**

output\_word **+=** t**.**tag**.**letter\_alpha**;**

List**<**BoardTile**>** outer\_storage **=** **new** List**<**BoardTile**>();**

outer\_storage**.**AddRange**(**item**.**current**);**

**if** **(**t**.**accepted\_placement **==** **true){**

**if** **(**t**.**up**.**accepted\_placement **==** **false** **&&** t**.**down**.**accepted\_placement **==** **false){**

int new\_loc **=** t**.**id**;**

**for** **(**int i **=** marker**;** i **>=** 0**;** i**--){**

**try{**

mapAITileToBoard**(**outer\_storage**[**i**],** new\_loc**);**

**}**

**catch** **(**Exception e**){}**

new\_loc **=** new\_loc **-** 15**;**

**}**

new\_loc **=** t**.**id**;**

**for** **(**int i **=** marker**;** i **<** item**.**current**.**Count**;** i**++){**

**try{**

mapAITileToBoard**(**outer\_storage**[**i**],** new\_loc**);**

**}**

**catch** **(**Exception e**){}**

new\_loc **=** new\_loc **+** 15**;**

**}**

**}**

**else** **if** **(**t**.**right**.**accepted\_placement **==** **false** **&&** t**.**left**.**accepted\_placement **==** **false)**

**{**

int new\_loc **=** t**.**id**;**

**for** **(**int i **=** marker**;** i **>=** 0**;** i**--){**

**try{**

mapAITileToBoard**(**outer\_storage**[**i**],** new\_loc**);**

**}**

**catch** **(**Exception e**){}**

new\_loc **=** new\_loc **-** 1**;**

**}**

new\_loc **=** t**.**id**;**

**for** **(**int i **=** marker**;** i **<** item**.**current**.**Count**;** i**++){**

**try{**

mapAITileToBoard**(**outer\_storage**[**i**],** new\_loc**);**

**}**

**catch** **(**Exception e**){}**

new\_loc **=** new\_loc **+** 1**;**

**}**

**}**

**}**

marker**++;**

**}**

**this.**AIPlayedWords**.**Items**.**Add**(**new\_items**[**0**].**ToString**());**

end **=** DateTime**.**Now**.**Ticks **/** TimeSpan**.**TicksPerMillisecond**;**

end **=** end **-** begin**;**

**}**

**foreach** **(**BoardTile t **in** GameBoard**.**Children**){**

**if** **(**t**.**accepted\_placement **==** **false){}**

**else{**

t**.**placement\_possible **=** **true;**

**}**

**}**

**this.**AIScore**.**Content **=** **this.**cpu\_tracker**.**score**;**

game\_logic**.**clearBuffer**();**

beginAISequence**();**

**}**

**}**

# GUI Algorithms

## Submit Word

When the user clicks the submit word button. It sort the tiles into a word based on the board ID of the tile. The word starts with the smallest ID and ends at the largest. It then uses the search for a single word algorithm to make sure it is a valid word. If it is not a valid word it returns the tiles to the tray. If it is a valid word it then tries to place the word on the board.

## Place Word

First it checks the board and marks all the board tiles with pieces on them as ‘placed’. Then it calculates the score, updates the score, update the move counter, update the score history with the score and the word, clears the buffer and update the rounds counter, clears the selection placement border and then tell the AI to take a turn.

## Place Tile

Iterate through all the board tiles. If the current board tile is equal to the placed tile. If it is occupied it returns an error. If the placement conditions are satisfied then you de-queue from the selection queue. Then is will assign the properties of the de-queue to a new board tile and check the neighbour mappings to see if the tile was an intersection placement, if so, any adjacent tiles are then added to the game logic selection list (This will consider valid word selections when submitted).

## Select Tile

When a player clicks on a tile. It adds the tile to the selection queue and displays the letter in the letter feed. It also remove the tile from the user tray in the GUI.

## Show Possible Placements

This algorithm takes the number in the selection queue and draws a green box around in all direction from the placed tiles already on the board.

# Other Algorithms (Notable Mentions)

* Bonus Scoring Mapping
* Game Tree
* Score Calculation
* String – Tile Mapping
* Tray Fill (AI & Player)
* Refresh Tray
* Sort Tile Ordering
* Reset Tiles
* Draw Tray
* Grid Adjacency Algorithm

# Performance Analysis

## Initial Selection

First and foremost, it is important to note that performance cannot be analysed at the beginning stages of the play state as it incredibly difficult to report timings as they are typically below 0 milliseconds and lower resolutions are somewhat inaccurate using C#. Once the player (human) has placed a valid word and the system confirms this, the AI sequence will begin. The first step of this sequence is to compile a list of valid pre placed tiles that have been confirmed as accepted placement board tiles. Given the sample set:

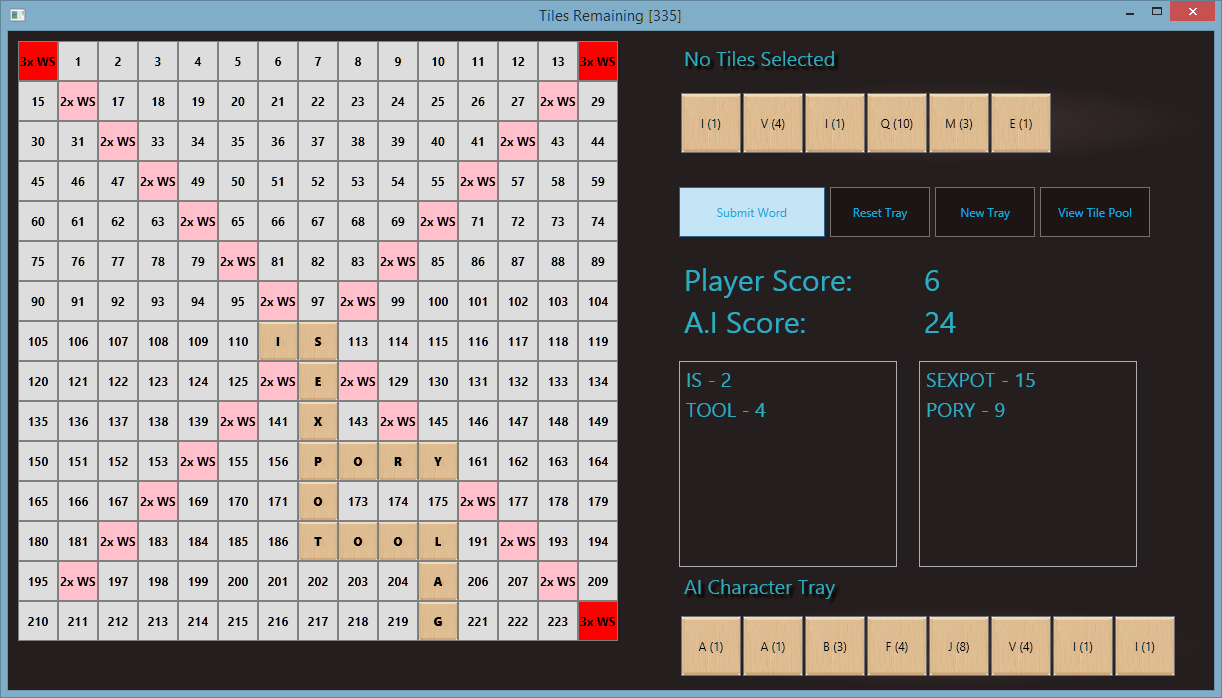


Figure 2 Scrabble Game State 1

The following list of words have been confirmed from rounds played by both player and CPU: IS, SEXPOT, PORY, TOOL and LAG. Once the AI turn begins, from ten sample data sets the following results were given for timings with a candidate list of between 10 and 15. Results were typically within the range of 300 and 600 milliseconds. This shows the expected increase of computational demand the more tiles on the board.

Figure 3 Algorithm Performance Figures

## Candidate Selection

Once the AI has collected the list of valid possible start locations, the algorithm iterates through the list and subsequently finds all possible word candidates using the AI Search functions available using the Trie. The following game environment is responsible for this metric set:

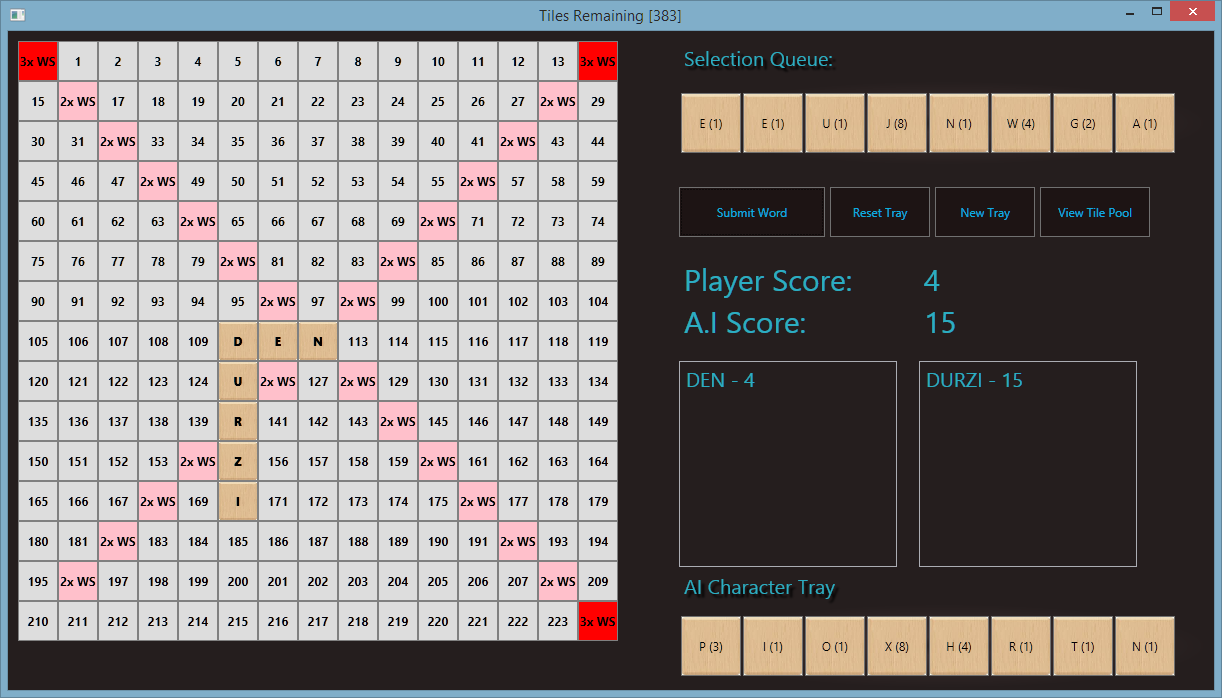


Figure 4 Scrabble Game State 2

From analysis of the roots: D, E and N. The algorithm performed with measurable results.

1. ***D*** *produced a possible set of 51 words from the tray and calculated these combinations in 11 milliseconds.*
2. ***E*** *produced a possible set of 57 words from the tray and calculated these combinations in 25 milliseconds*
3. ***N*** *produced a possible set of 55 words from the tray and calculated these combinations in 9 milliseconds.*

From these results it can be concluded that the number of letters in the search does not directly affect the time it takes to compute possible results. Other variables have to be taken into consideration such as system processes and background tasks also. Therefore, this result set is not a conclusive set, rather an indication of relative times to produce search results.

The Trie algorithm performs its own searches in unmeasurable producing times below 0 milliseconds beneath this measure level, thus, the majority of time it takes for this algorithm to complete is due to the validation of placements from the search candidates.

## Algorithmic Time (AI)

Below is the results set for the staggered timeline that begins when the AI initiates its game sequence to the point where the completed word selection has been consider, counted and placed on the board. The results have been calculated from the following game state where the last player is awaiting submission of [S, I, T], prepending the T at the end of [A,W,R,I,E,S,T].

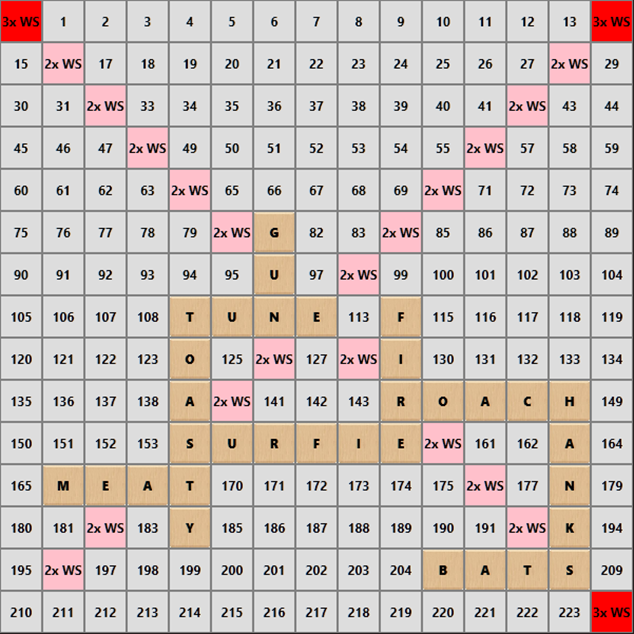


Figure 5 Extended Scrabble Game State

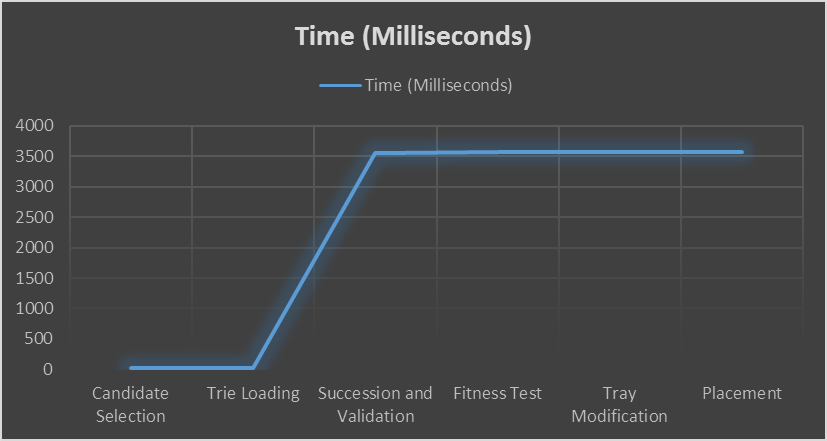


Figure 6 Algorithmic Timeline

The above info graph concludes that the majority of time taken is due to the validation of selections. This falls under the rules and restrictions of inputs and data.

# Change of Approach

Overall, from a begin-end development process, not a huge amount could be altered in terms of making the system more efficient. Every effort was made to adhere to OOP principles throughout the development process and encapsulation of necessary classes was a key asset in this process.

One of the largest and most significantly appreciable areas of redevelopment and development concerns was the GUI components relating to the overall structure of the Game Board. This ranged from both learning about Windows Presentation Foundation XMLSX and the content deployment options for programs using this design concept.

When designing the board a simple array structure was considering and any algorithms would utilise this array as a two dimensional grid structure. Nevertheless, this was redesigned at a later point and substituted with a linear game board system ranging from tiles at index 0 to tiles at index 254. Using this system is was easier to implement the bonus system and also to dynamically create, modify and delete tiles on the board.

Below is an ideal representation of a grid system:

The implemented design of the UI board system:

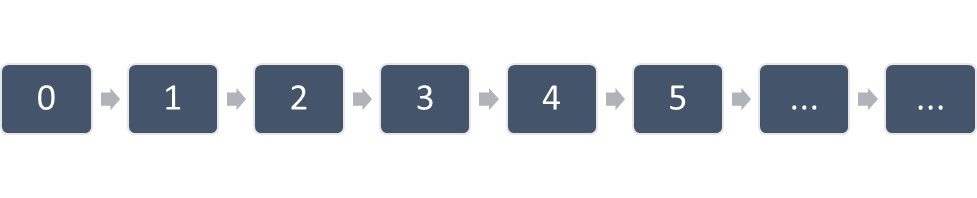


Figure 7 Implemented UI Design

However, issues began to emerge when implementing the AI components as developing a constraint system proved to be challenging when working in a linear tile fashion. Subsequently, many efficiency points were lost due to reiteration of an entire list of board tiles rather than direct grid referencing using indices that could be easily retrieved using the algorithms.

The design forced modulo index calculations and the workaround resulted in encapsulating the Button class and adding neighbour tile attributes which worked exceptionally well when operating with human interaction. Although the AI components were not as easily interoperable due to the inability to change items during an iteration of an ENUM collection. Thus, many object copies had to be stored and re-referenced for replacement and deletion.

Efficiency gains were only returned after revising the system and removing unnecessary iterations through the data.

# Learning Outcomes

## Algorithmic Selection & Data Structures

It was understood from the beginning that one of the largest considerations of how well the system will perform depends on choosing the right algorithms to operate on the right data structures. Initially, options such as SQL procedures to search for words, list/arrays, dictionaries etc. were considered. In the end, Trie structures were considered the most efficient and stable. This was concluded using in line analysis of speeds of operations. List/Arrays proving to be the slowest.

This providing us with an experience of in depth research and analysis using primary research results from analysis to provide sufficient conclusion from which a design will be based.

## Scoping

Initially, the project was over scoped with somewhat overly optimistic goals in mind. Due to this, strategic de-scoping was necessary. Deciding which critical functions should remain versus functions which would be in the “nice to have” category was somewhat frustrating at times. This involved learning to prioritise functions that were critical to developing a core prototype. Essentially, putting main goal completion ahead of features. This would have been a more pleasant process if research was done more accurately at the beginning and system design was formalised.

## Research Techniques

Extended research techniques were used during the life cycle of the project. These involved researching the projects own results, compared secondary results from data available and used live results to base revisions off.

# Conclusion

Scrabble as a computational concept using an intelligent system is very much possible using the correct mix of data structures, algorithms and design techniques, even given a specific set of constraints.

During each successive play-through using the engine it was incredibly difficult to match the playability of AI. Even with necessary cutbacks made to functioning and possible play points of the AI, it was impossible for either group team members to beat the system. Perhaps an expert level player would hold the challenge better, however this is not conclusive.

Overall, this shows that the initial proposal and main deliverable was met with resounding success and ambitions for a completely tailored system still remain. Further adaptation of intersection options, improved root searches and a generally more sensible AI would make it highly improbably of any human being able to match the system’s ability.

With that in mind, it can be concluded that machines are exceptionally better at calculating Scrabble possibilities than typical humans. Although this was not disputed prior to research and development, it was at the time merely a hypothesis that it can be done efficiently.

With the ability to compute a list of several hundred words for all tiles on the board at one time in less than three seconds, the greatest options are easily chosen faster than most humans can even consider a single, basic and low scoring word. This type of fast computation has many extended uses beyond Scrabble game techniques.

In terms of expectations versus realities through the project, the greater challenge lies in the implementation adhering to specific data sets and governing the AI to respect the rules and reduce delays in operations to satisfy a human player, even as a UI consideration, fast calculations.

# References

|  |  |  |
| --- | --- | --- |
| [1] |  | SQLITY, “B-Trees How SQL Server Indexes are Stored,” SQLITY, 2 November 2015. [Online]. Available: http://sqlity.net/en/2445/b-plus-tree/. |
| [2] |  | Hasbro, “Scrabble Hasbro,” 04 10 2015. [Online]. Available: http://scrabble.hasbro.com/en-us. |
| [3] |  | “The English Open Word List,” Dreamsteep, 04 10 2015. [Online]. Available: http://dreamsteep.com/projects/the-english-open-word-list.htm. [Accessed 04 10 2015]. |
| [4] |  | “Scrabble - The Pixie Pit,” The Pixie Pit, 01 10 2015. [Online]. Available: http://thepixiepit.co.uk/scrabble/dictions.htm. [Accessed 01 10 2015]. |
| [5] |  | “Merriam - Webster Scrabble,” Merriam - Webster, 04 10 2015. [Online]. Available: http://www.merriam-webster.com/game/scrabble.htm. [Accessed 04 10 2015]. |
| [6] |  | R. Bell, “A Beginner's Guide to Big O Notation,” Rob Bell, 28 October 2015. [Online]. Available: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/. |
| [7] |  | A. W. A. &. G. J. JACOBSON, “The World's Fastest Scrabble Program,” *Communications of the ACM,* vol. 31, no. 5, 1988. |
| [8] |  | Microsoft, “Microsoft .NET Homepage,” Microsoft, [Online]. Available: https://www.microsoft.com/net. [Accessed 11 December 2015]. |

[Figure 1 Example Trie Structure 3](#_Toc444768648)

[Figure 2 Scrabble Game State 1 11](#_Toc444768649)

[Figure 3 Algorithm Performance Figures 11](#_Toc444768650)

[Figure 4 Scrabble Game State 2 12](#_Toc444768651)

[Figure 5 Extended Scrabble Game State 13](#_Toc444768652)

[Figure 6 Algorithmic Timeline 13](#_Toc444768653)

[Figure 7 Implemented UI Design 14](#_Toc444768654)

# Work Division Declaration

All works contained herein this document, preceding documentation and deliverable materials submitted as part of the project submission were completed on an even group basis between Keith Byrne and Liam Strevens. No disputes of workload share have emerged throughout the project and both individuals agree joint recognition of all material.

All associated development work can be found at the following public GitHub Repository:

<https://github.com/kmjbyrne/ScrabbleEngine>

I \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ completed an even share of work and confirm that work was carried out fairly.

I \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ completed an even share of work and confirm that work was carried out fairly.

Date Signed: 03/03/2016