COMP3821 - Extended Algorithm Design and Analysis

Bananagrams Engine: Group Report

Kyler Blackburn z5421987
Julian Zincone z5425206
Flynn Lambrechts z5360922
Tim Tang z5488485

April 2024

Figure 1: A Bananagrams Board

Contents

1 Introduction

This is a report on an attempt at developing a Bananagrams engine, a creative project based on greedy algorithms and dynamic programming that analyses the game state and returns what it calculates to be the best actions.

1.1 Introduction To Bananagrams

Bananagrams is a word game similar to Scrabble where lettered tiles are used to spell words. The game-play involves two to eight players arranging tiles from a shared pouch into a grid of connected words as in Figure 1. A player can call 'dump' and return a tile to the pouch and take 3 new tiles (or the rest of the pouch if the pouch has less than 3 tiles) and when a player has no more tiles in their hand they can call 'peel' to make every player in the game draw 1 tile. When the number of tiles in the pouch is depleted below the number of players in the game then the first player to use all of their tiles to construct a board wins. The official rules are detailed at this link.

Our task is to create a Bananagrams engine that applies several different algorithms to quickly and successfully win games of bananagrams.

1.2 Motivations

Looking at the background outlined in the **Survey**, it is clear that the current body of work on the Bananagrams engine topic fails to consider the whole context of the Bananagrams game. Previous approaches have not considered the 'dump' game mechanic and the existence of multiple players. The motivation behind the creation of our Bananagrams engine is to achieve the goal of expanding on the current body of work to create a Bananagrams engine that considers the ignored game mechanics and has a greatly increased success rate.

2 Survey

2.1 Existing Work

While Bananagrams is a fairly new game (made in 2006) [4], people have been making scrabble AIs since at least 1988 [2] with some common key elements. Finding a variety of anagrams quickly is a key problem. Appel and Jacobson created a 'Directed Acyclic Word Graph' (DAWG) that was essentially a space-efficient trie for finding words. Steven Gordon [5] devised a GADDAG, another DAG, but referencing every possible starting prefix of every word, making it much more efficient for adding onto an existing board where the existing letter is in the middle of the word.

In the case of Bananagrams, there are few publicly explained solutions. AIs exist for playing it online, and there are some solutions, but many fail to consistently complete the game [1, 6], while others solve slightly different problems, such as simply making scrabble-like boards given a set of letters (rather than dealing with the specifics of peeling and dumping). These attempts are still valuable. Agrawal and Kwok [1] demonstrate another anagram data structure where letter order is alphabetically sorted beforehand and demonstrate the difficulties of dealing with one additional letter at a time, using a Markov decision process to attempt to predict whether particular searches will be fruitful. Additionally, while Grady [3] attempted a subtly different problem, Grady demonstrates how smart tiling of the words can reduce the search space for anagrams while making it easy to add words in the future.

2.1.1 Ideas That Are Unlikely To Produce An Optimal Outcome

Most engines that limit the freedom of the algorithm to alter the position of the tiles on the board run into the problem of potentially failing to find a complete solution for the given tiles. So looking at previous Bananagrams engines, the approach of sequentially adding words based on some heuristic without any way to reorder letters on the board will struggle to have a high success rate as it allows for the possibility of a dead end (where some letters are in the hand that cannot be used to make a word anywhere on the board). The approaches by Agrawal and Kwok show that even using heuristics to evaluate word quality before placement still results in failure about 20% of the time [1].

Although creating an algorithm that limits the rearrangement of tiles can run into the problem of potentially failing to find a solution, it is also possible that an algorithm could find itself stuck in a loop of rearranging the same tiles over and over again.

Additionally, several naïve approaches to anagram solving involve the brute force approach in looking up every permutation of the letters in a dictionary. This is likely to be slow and can be optimised further using a trie data structure. Furthermore, it does not take into account spatial reasoning.

2.2 Areas Of Development

After reviewing the literature discussed above these are the key areas of development targeted in this report.

2.2.1 Restructuring

The literature discussed above does not implement a method for backtracking once a point has been reached where there are not valid moves. Possible approaches to this are discussed at length in this report.

2.2.2 Multiplayer Environment

There is technically an $O(1)$ solution to single player Bananagrams. Since the dump mechanic allows a player to gain tiles, it is possible in single player to simply draw all the tiles and play them in a preconfigured layout. Having multiple players complicates the problem significantly as the player only has access to a finite number of tiles and may be given additional tiles at random. Furthermore, other players can influence the probability of receiving certain tiles if they chose to dump.

2.2.3 Making Use of Game Mechanics

All existing solutions fail to use the key game mechanic of dumping. Dumping allows a player to easily access new tiles and also remove unfavourable tiles.

2.2.4 Stranding

Grady's approach focuses on building words in strands allowing for ease of extension without considering word collisions. However, Grady's program does not appear to incorporate this into a real game environment with a correctly size pouch of tiles and peeling.

3 Solution

The problem has been broken down into the following sub-problems.

1. *Anagrams* - Given a list of letters find a way to rearrange them such that they make a valid word.
2. *Stage 1* - Given A List of Characters, arrange them into a word grid such that all letters are used.
   (a) *Heuristics* - Use heuristics to determine the best word to play given a hand and a board
   (b) *Spacing* - Ensure that words played don't create illegal configurations by overlapping or being adjacent to words that can't connect
3. *Stage 2* - Given a word grid and one character incorporate the character into the word grid such that all letters are used.
   (a) *Short Words* - Find a way to attach a single tile to an existing word grid.
   (b) *Restructuring* - When the trivial case cannot be completed, restructure the word grid to incorporate the new tile or dump a tile in exchange for new random tiles.

3.1 Anagrams

3.1.1 Tries

Finding valid words requires being able to quickly find anagrams from a given selection of letters. Finding all anagrams of a word or collection of letters can be done in $O(n)$ time with a cleverly designed hash - assigning each letter its own prime number and multiplying them results in the same key for all anagrams of the same word. Even if the practical implementation of this wouldn't be quite as good, this could easily have an average case time complexity very close to $O(n)$. However, the main issue with this approach is that in most instances, what is wanted is all anagrams of all combinations of letters given. From here onwards, 'anagrams' will refer to the anagrams that use every letter of an input, while 'subwords' will refer to the anagrams of every combination of the input. For example, given the letters 'HENW', the hash table would return ['HEWN', 'WHEN'], but the desired list of all subwords is ['EH', 'EN', 'EW', 'HE', 'NE', 'WE', 'HEN', 'HEW', 'NEW', WEN', 'HEWN']. Simply adapting the hash table to consider every combination of letters transforms it from an $O(n)$ algorithm to an exponential time algorithm. As a result, a better algorithm is needed.

Instead, we created a trie, where each word's key is it's sorted letters. This structure can be recursively searched much faster. While there are still many combinations of letters to consider, any combination that is not a word or part of a word will not be a node on the trie, and so won't be checked. The increasing sparsity of the trie with increasing word length means added more letters does not significantly increase search time - at the extreme end, once the letters considered contain every word in the dictionary, the time complexity becomes constant as every node of the trie is checked.

We created some additional functionality to our trie recursive lookup (**trie.all_subwords**) to suit particular use cases. First, most words are created with the intention of being connected to an existing tile (anchor). As a result, the anchor must always be in the provided output. Second, sometimes you want the anchor to explicitly be either the first or last letter of the word. As a result, we also created **forward_trie** and **reverse_trie** (with the default being called **sorted_trie**), where, in the construction of the trie, the key is either entered unsorted or in reverse. In these cases, specifying an anchor means that the anchor will be the first or last letter of the word, for the forward and reverse tries respectively.

3.1.2 Heuristics

As part of finding valid words, a word prioritization mechanism was also created to select a single word to place on the board from the anagrams that our trie's **all_subwords** method found. The purpose of this word selection process is not only to filter out words that cannot legally be placed on the board, but also to ensure that the word placed on the board maximizes the opportunity for further words to be placed on the board (in a Bananagrams legal configuration).

So, words are selected first by ensuring that all words provided by the **all_subwords** method are able to be placed on the board, then the word with the highest score according to the players' word scorer is returned. A player's word scorer which is passed in for each player in the game can be one of the following:

1. **Dead Letter Avoidance Word Scorer (ScoreWordTwoLetter)**:
   This word scorer ranks words higher the more characters they have and ranks words slightly higher the rarer their characters are in the set of two letter words (the less a character can be used to make a two letter word the higher its rank). Additionally, for the each instance of the letter 'V' or 'Q' the scorer ranks the word significantly higher and for each instance of the letter 'J', 'X', 'C', or 'Z' the scorer ranks the word significantly higher as well (but less than 'V' or 'Q'). The purpose of this word scorer is to prioritize words that include letters that are harder to play, so letters that caused lots of errors in testing and/or are in very few two letter words will be heavily prioritized when selecting which word to play.

2. **Simple Stranding Word Scorer (ScoreWordSimpleStrandingLongest)** This word scorer designed for the **Stranding approach** to the Bananagrams engine ranks words higher based on greater lengths in initially while adjusting the score based on the following criteria:

   (a) Increase the score based on the number of two letter words that can be made from the start and end characters of a word.

   (b) Decrease the score based on the number of two letter words that can be made from the rest of the characters in the word.

   This ensures that letters who can make more two letter words are prioritized to be placed at the start and end of the word, while letters who can't make as many two letter words are prioritized to be placed somewhere in the rest of the word (not start or end).

3. **Hand Balance Word Scorer (ScoreWordHandBalanceLongest)** This word scorer ranks words higher the more balanced (accurately representing the distribution of letters in the pouch) the hand of the player would be after playing the word (on top of ranking them higher the longer the word is). So if our hand after playing a word contains more rarer letters than common letters, then our hand will be unbalanced and the word that would cause such unbalance will be ranked lower than a word that would cause our hand to include a distribution of letters closer to the distribution of letters in the pouch.

4. **Letter Count Word Scorer (ScoreLetterCountLong)** This word scorer scores words higher the fewer times that the letters it contains appear in the dictionary (on top of word length). So for each letter in the word the score is increased by a set amount (10^7), and decreased by the number of times that the given letter appears in the whole dictionary (always a number much smaller than 10^7).

Additional word scorers could be created that combine features from the above word scorers such as ranking based on the number of two letter words that a letter makes combined with its letter count in the dictionary.

3.2 Stage 1

3.2.1 Finding a valid word

Limits Words can only be placed on the board if there is space to do so. Depending on how the recognition of spacial limits is done, it could take up a significant part of the time of each turn. Most previous implementations we encountered simply ran a check for how much space there was in each direction as each word was considered[1, 6]. To reduce the time burden of this step and avoid having to recalculate how far words can extend from different anchors every time, we created the attribute **lims** that stores how much space there is to play a word in each direction relative to the given tile.

Update_lims runs as follows. Probes are sent out in all 4 directions from the target tile until there is a collision or the probe leaves the extent of the board. The **lims** of the target tile and those that were collided with are updated accordingly. There is slightly more nuance required to cover every case (like a tile that is in position (row + 1, col + 1)), but the general structure and time complexity remain the same.

Update_lims is run every time a tile is placed, and when a tile is removed, probes are sent out from the location of the removed tile. Every tile hit by the probes then runs **update_lims**.

TABLE I: Time complexities for lims operations

Operation	Time Complexity
Insert Tile	$O(w+h)$
Remove Tile	$O(w+h)$
Check Limits	$O(1)$

Table I shows the time complexities of the key operations, noting that w, h are the width and height of the board, respectively. Since checking limits is performed at least as often as insert tile, and realistically usually many times more than that, having it as an $O(1)$ operation is very valuable.

3.2.2 Searching for and placing words

Now that both the anagram and spacial components of finding valid words has been covered, as well as scoring heuristics, a general strategy for finding and playing words must be formed. We cover two general strategies: the regular approach and the stranding approach.

```
best_words_list = empty list of best words
for each anchor:
    words = all_subwords(hand, anchor)
    words = filter(words: there is space for word)
    add max(words, key = scoring heuristic) to best_words_list
play max(best_words_list, key = scoring heuristic)
```

Figure 2: Pseudo-code for Regular Approach

Regular Approach The most simple approach, shown in Figure 2 is to consider all the words that can be created for every anchor, and then play the word that is rated highest by the scoring function. Most historical algorithms we found did this or, after playing a long first word, greedily went through each anchor and played the longest word possible each time. The main downsides of this algorithm are that it can be slow and doesn't consider the overall structure of the board.

Stranding Approach The only approach we saw that differed from the naïve approach is one that we will refer to as the *stranding with lanes* approach. It was interesting because it dramatically reduced the search space by joining most words through two letter words. This appeared to have three key benefits.

First, if an algorithm considers every option each time, reducing the search space can dramatically speed the search up. And, if the difference between the 99th and 90th percentile evaluated words is not substantial, the time-quality trade-off may be worthwhile.

Second, the likelihood of potential words colliding with existing words goes down as the board becomes more spread out. This means that as the board gets larger, rather than searching through a larger number of anchors with a decreasing proportion of anchors that actually have space for new words, the places to expand grow, while the proportion of anchors that are useful remains high.

Third, because words are being attached via two letter words, rather than running **all_subwords** once for each anchor, it can instead be run just once, as the new word will have no tiles in common with what is on the board. Then, the first word found in the list (sorted by the scoring algorithm) that can attach to an anchor will be played.

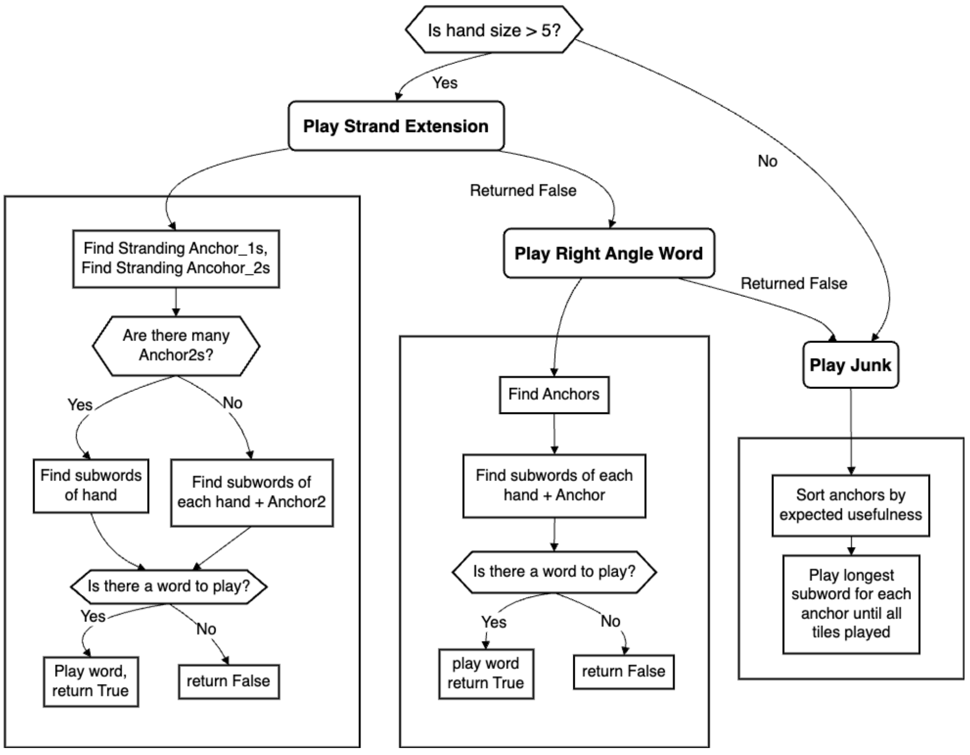


Figure 3: Flow Chart for Stranding Approach to Stage 1

Because there is only a small number of anchors suitable for stranding at the start of the game, we first wrote the algorithm by considering each stranding anchor and running **forward/reverse_trie.all_subwords** on it, but we later also added the ability to run a **sorted_trie.all_subwords** once if it seemed like there were enough anchors for that to be worthwhile. In Figure 3, Stranding Anchor1s are the existing tiles on the board, while Anchor2s are the tiles in hand that will link to the Anchor1s and either be the first or last letter of the new word.

Three different algorithms using Stranding as the main strategy have been implemented. The first and most simple is referred to as Stranding in the results section. It only uses the method of running **forward/reverse_trie.all_subwords** on each anchor. The second, **TwoLetterStranding** is identical to Stranding in all aspects that are discussed here

```
    0 1 2 3 4 5 6 7 8 9 0 1
  -5               L
  -4               I
  -3               Q
  -2               U
  -1               I
   0U N G R O U N D E D L Y
   1    I          Y
   2    V
   3    E
```

(a) Naïve

```
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
  -1        D         G     B R O G H
   0M U L T I G R A V I D A E
   1  P
```

(b) Stranding

Figure 4: Completed Stage 1 Boards with Naïve and Stranding Approaches

3.2.3 Potential Improvements

Limits Small improvements to the **Lims** attribute could be implemented. First, the probe could terminate earlier when there are no collisions - instead terminating at min(max word length, dist to edge of board), improving the asymptotic properties, even if in practicality, there aren't enough tiles in bananagrams for this to be a major concern.

Another improvement would be to the behaviour when a tile is removed. Rather than running **update_lims** on each tile that the probe hits, these tiles only need to update their limit in one direction, not all 4, on average making the limits aspect of removing a tile 4 times faster.

A more radical improvement to calculating limits would be to design word placement such that limits essentially don't need to be calculated. This is done here, assigning 'lanes' for growth of strands[3]. While we took inspiration from this to develop our stranding algorithm, our looser placement rules meant that the limits information is still needed. In Figure 5, there are lanes in rows -1,0 and 4,5 extending to the left. Unfortunately, we did not have time to implement it and assess if the reduction in search space is compensated for by not needing to calculate limits.

```
  2 1 0 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6
-1  P A N C Y T O P E N I A
 0                        B O W L I K E
 1                            U
 2                            T
 3                            H
 4      A X L I K E           E
 5F A C T         T W I R L E R
 6                            I
 7                            N
 8                            G
```

Figure 5: Stranding With Lanes

Play strategy Currently, the stranding strategy recalculates the valid stranding anchors each time. It would be fairly easy to instead store the list of stranding anchors, since no more than 2 anchors are added and no more than 1 anchor is removed with each play.

3.3 Stage 2

3.3.1 Adding Short Words

The peel mechanic (each player drawing a single time when a player has completed their board) means that a large part of the game is played with players having a very small number of tiles in their hand. As a result, it can be useful to have a specific approach to this stage of the game. While the Naïve playing approach can play short words as easily as long words, the stranding approach constricts the search space enough for playing short words to be difficult.

3.3.2 Restructuring

Despite getting stuck being a common issue[1], no existing algorithms were found in literature that described a process of removing or rearranging tiles besides removing everything. Two general strategies that went beyond this were developed - *Dangling* and *Playing Junk*.

Naïve Approach In the first working version of the player, often the player would reach a point where it cannot play anymore letters from it's hand as shown in Figure 6.

```
StandardPlayer 0: [FINDING WORD] Available Letters V
StandardPlayer 0: [FOUND] 0 word candidates
StandardPlayer 0: [PROBLEM] Could not find next word from hand V
StandardPlayer 0: [BOARD]
    4 3 2 1 0 1 2 3 4 5 6 7 8 9 0 1 2 3
  -3              R
  -2              A
  -1              D           Q
   0        S U B I N F E U D A T O R Y
   1              O           T   R
   2              A               Y
   3              C               X
   4              T
   5              I
   6              V
   7              A
   8              T
   9              I
  10              N
  11S W O T T I N G S
```

Figure 6: Board After Player Got Stuck

The naïve solution to this was to take all the letters off the board and start again choosing a slightly less optimal starting word. This proved to be very costly since rebuilding the board takes an extremely long time, slowing down the player significantly and in some cases even after 20 attempts not resulting in a final solution at all.

Dangling Approach During the trivial case the player ends up playing a significant number of two letter words as this is the only option when one tile is received. This means the board becomes quite cluttered making it difficult to tack on new words. Figure 7 illustrates a cluttered board with only one available position to add a letter (the C in Juice). A more advanced approach to the naïve approach is to purge all tiles that are dangling and return them to the hand.

```
StandardPlayerDangling 0: [BOARD]
    0 1 2 3 4 5 6 7 8
  -5            S
  -4            O R
  -3              E
  -2            A L E
  -1    A     J   I
   0F O R E J U D G E
   1  O       I   I
   2          C   E
   3          E   U T
   4              X
```

Figure 7: Board With Numerous Dangling Tiles (Shown in Blue)

A dangling word is a word with 1 or fewer junctions in it. For example in Figure 7 'JUICE' and 'ALE' would be considered dangling.

By removing all dangling tiles the player can access a significant amount of anchors and also allows the player to create even more anchors by using the dangling tiles to play a longer word. Figure 8 illustrates the board after the dangling tiles are returned to the hand, the board now has 13 anchors.

```
StandardPlayerDangling 0: [BOARD]
    0 1 2 3 4 5 6 7 8
  -4            O R
  -3              E
  -2              L
  -1              I
   0F O R E J U D G E
   1              I
   2              E
   3              U
   4              X
StandardPlayerDangling 0: [DANGLING] Found 10 dangling tiles
StandardPlayerDangling 0: [FINDING WORD] Available Letters
QJICEAEAOTS
StandardPlayerDangling 0: [FOUND] 3 word candidates
```

Figure 8: Board With One Layer Of Dangling Tiles Removed

To determine whether a word is dangling, the method is to do a linear search through the list of tiles to determine if there is at most 1 junction. This runs in $O(n)$ where n is the number of tiles, as it looks through all words and hence all tiles.

Table II illustrates a significant speed up in the performance of the player when using the dangling approach as opposed to the Naïve approach

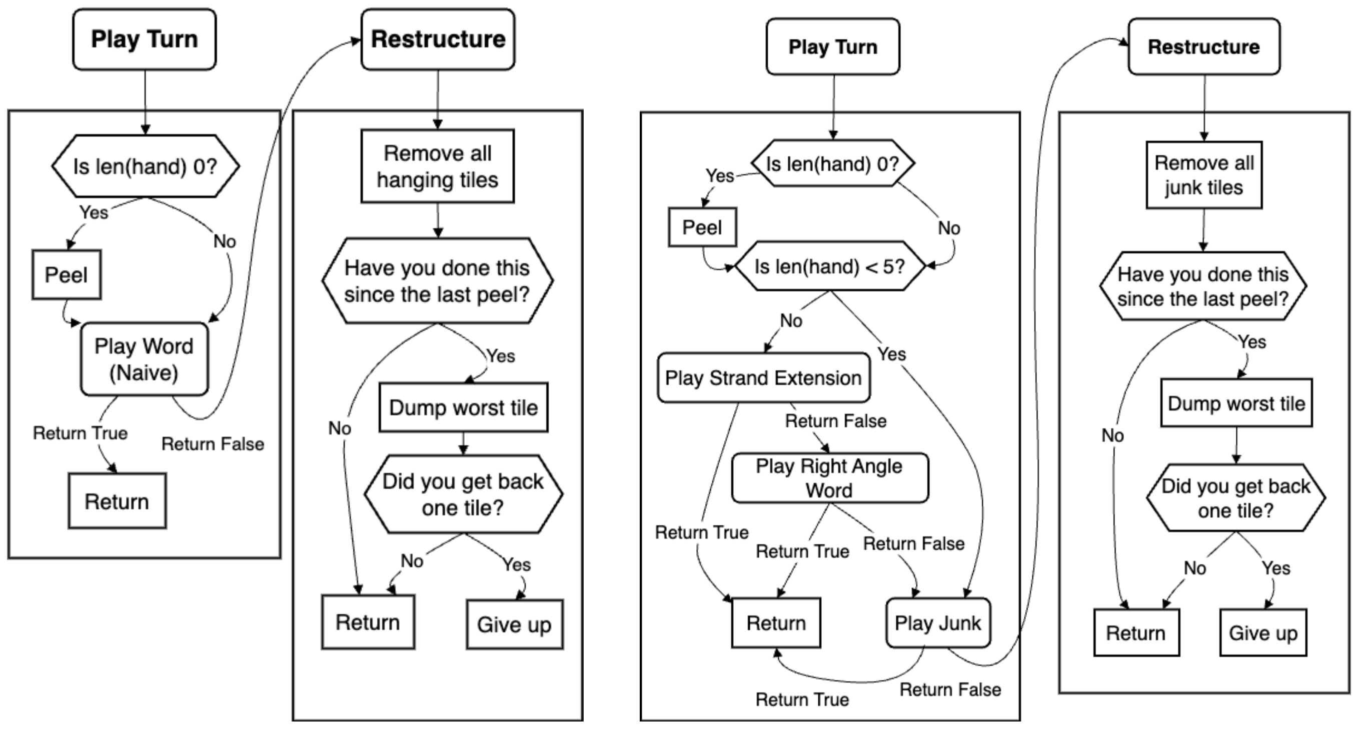
TABLE II: Naïve Approach Compared to Dangling Approach Across 25 Iterations

Player	Mean (ms)	Median (ms)	Standard Deviation (ms)
Naïve Player	2159.20556	2072.53200	526.97131
Dangling Player	129.14928	111.51800	51.86977

Table III shows the results when these two players were played against each other with one other passive player 25 times. Clearly the Dangling player has a greater win rate.

TABLE III: Win Portions of Dangling and Naïve Players

Player	Win Rate
Naïve Player	24%
Dangling Player	76%



(a) Using Dangling to Restructure (b) Using Stranding and Play Junk to Restructure

Figure 9: Flow charts of Dangling and Stranding Algorithms with their Restructuring

3.3.3 Removing Junk and Dumping

When designing the stranding algorithm, consideration was given to an alternative approach to restructure the board. Since the search space is narrowed already in playing longer words, an algorithm optimised for short words was necessary. The simple implementation simply goes through each anchor and tried to play as much as possible (which is typically only 2-3 letter words given right angle and stranding words can't be played) until either all letters are played, or it has gone through every anchor. However, because all of these added words are short and aren't adding to improving placement opportunities for future words, tiles placed during this phase are labeled as "Junk".

```
StrandingPlayer 0: [BOARD]
    4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
  -6
  -5
  -4            O
  -3C A P S U L E D
  -2  G     S     O N O       T                 L       I X I A
  -1                  M A T Z O H       Q     F A I K E D   N
   0                    X       E N G R I E V E   T   S
   1                                      R
```

Figure 10: Board with Numerous Junk Tiles (Shown in Blue)

Then, when a restructure must occur, all junk tiles are removed from the board and played back in the player's hand. To avoid infinite loops, the game keeps track of if it has pickup up new tiles since the last restructuring. If it has not, then after taking all of the junk tiles off, it dumps the least common tile in its hand (returns the tile to the pouch in exchange for 3 new random ones). It gives up if it tries to dump when there is only 1 tile left in the game.

3.3.4 Two Letter Stranding

The difference between the Stranding player and the Two Letter Stranding player (a.k.a. Two Letter Junk Stranding player) is that the latter only plays junk words of length two. This restriction in search enables the player to precompute a greater portion of the work required to find the best junk words. The precomputed work includes a map from each letter of the alphabet to a set of letters which form two-letter word with the key. For example, ”A” maps to a set containing ”AA”, ”AU”, and ”AB”. The player also creates a initialises a map from each letter to the number of non-unique two-letter words the key can be a part of. The former is used to see the candidate letters that can be paired with a certain letter, and the latter is used to rank letters based on how many two-letter words, with duplicates allowed, use that letter. The latter is especially useful for when it comes to selecting a candidate letter to pair with another letter, as it allows the player to choose the candidate letter which is used the least, so as to preserve the popular candidates for the future.

The final precomputation uses the existing maps to create a map from a junk letter with an anchor letter which returns the proportion of two-letter words formed with the anchor letter that are also formed using the junk letter. The higher the value, the more desirable it is to match the junk letter to the anchor. This map is given then name **junk_anchor_synergy** and using it, the anchor candidates are considered in descending order of synergy until a valid anchor is found. Because of the search space reduction, anchor and anagram selection is fully handled by **junk_anchor_synergy**, eliminating the need to query the tries when it comes to playing junk, which should significantly reduce runtime. Besides this, the Stranding and Two Letter Stranding player are identical. When it comes to benchmarking, the results are somewhat inconsistent, but the Two Letter Stranding player performs better in most cases depending on the number of (pseudo) players, the presence of any opposing real players, and the choice of scorer.

3.3.5 Improvements

A possible restructuring approach that could run in faster than $O(n)$ time (where n refers to tiles on board) is to keep track of the chronological order of playing words. This might work particularly well for the stranding player after removing junk, as removing the one or two most recent non-junk words completely changes what anchors are available to use for strand extensions.

3.4 Summary of Algorithms

Table IV provides a summary of the different algorithms we created during this project. The building strategy refers to the details discussed in Stage 1, while the restructuring strategy refers to the details discussed in Stage 2.

TABLE IV: Description of Each Algorithm

Algorithm Name	Building Strategy	Restructuring Strategy
Naïve	Standard	Reset Board
Dangling	Standard	Remove Dangling Words
Stranding	Stranding	Play Junk
TwoLetterStranding	Stranding	Two Letter Junk
NewStranding	2 Method Stranding	Two Letter Junk

4 Implementation

See the appendices A and B for a visual reference on how these different strategies behave.

4.1 Code Repository

The repository is on GitHub here, and is structured as follows.

```
.
|-- analysis/
|-- assets/
|-- src/
|   |-- MiscWIP/
|   |-- Not in Use/
|   |-- board/
|   |-- pickles/
|   `-- players/
`-- tests/
```

The main bodies of code are kept in **src**, including the algorithms, models, abstractions, and controllers. Both **MiscWIP** and **Not in Use** contain source code that is not currently used but may be useful in the future, including a script which attempted to find the best initial word at the start of the game, a dictionary that mapped a string to a list of words which could be formed by any subset of the string, a serialisation of the hash dictionary, an AVL tree implementation, and the original Player implementation before it was made abstract.

The **board** directory contains our model of the board and its sub-models **tile** and **lims**. The **pickles** directory contains pickles, a Python feature, which are saved-to-file memory snapshots of the three tries, as loading the tries from their snapshots versus rebuilding them from scratch improved runtime. This is possible because the tries are never modified once they are fully built. The **players** directory contains an abstract Player which is not a fully operational player, but contains common methods and properties used by every concrete player. The directory also contains the concrete players which inherit from the abstract Player and implement new and unique methods that make them different from each other.

The two controllers are **main.py**, designed to run a single game of Bananagrams with a collection of players, and **benchmarker.py**, designed to run multiple games of Banangrams with multiple collections of players to collect statistics over a large sample.

The file **algorithms.py** is a miscellaneous collection and archive of player algorithms, including **where_to_play_word** and **long_with_best_rank**. Common and unchanging but relatively small pieces of data are kept in **constants.py**. Algorithms tied to lims are found in **lims_algos.py** and algorithms related to the two letter junk strategy are found in **two_letter_junk.py**. The files **game.py**, **pouch.py** and **word.py** along with **board** form our implementation and model of the Bananagrams game. The Game object manages the pouch and players, and each player manages their individual board. The file **trie.py** contains all three trie implementations, and each trie is built from scratch when they are directly initialised. The alternative method of trie creation is through pickling and de-pickling, which is handled by **pickle_manager.py**. To improve runtime and memory usage, the tries are made global by being instantiated only once and made available to every player through **trie_service.py**. The file **race.py** is a simple script to compare the wall clock times of two games, and **utils.py** contain simple utility functions that are not related to algorithms.

The **assets** directory contains non-code files used for running the game, such as the Scrabble dictionary and a list of possible starting hands. The **tests** directory contains tests on individual components, but has been seldom maintained over testing and bench-marking entire games altogether. The **analysis** directory contains a Jupyter Notebook which analysed the starting hands which could not pass Stage One.

4.2 Threading

Bananagrams is not a turn-based game like chess. Instead, each player attempts to complete their board in real time, so we modelled multiplayer by placing each player on their own thread. This allows every player to compete approximately side-by-side in real time while leaving most of the work to the operating system. This, however, also raises concerns over concurrency, such as whether each thread has the same amount of runtime, and whether the operating system thread scheduler favours one type of player over another by giving them more runtime and priority. While we believe these biases are unlikely, we have not proved that they do not exist.

The implementation of this was to have a Game class as the main thread which maintains player threads. The player threads build their board and can request new tiles from the board via dumps or peels.

Once one player has completed their board they notify the game and the game subsequently tells all other players to stop playing.

4.2.1 Avoiding Data Races

Since each player works on their own board the only shared data between players is the 'pouch' containing the unseen letters. A lock is placed on the peel and dump commands to prevent two players peeling in parallel resulting in race conditions.

5 Experimental Technique (Benchmarking)

Benchmarking was performed using **benchmarker.py**, a controller which runs multiple games with multiple permutations of players and word scoring strategies. We opted to not use multiprocessing to run multiple games in parallel as it, for unknown reasons, changed results.

5.1 Aim

To determine the runtime statistics and winner frequencies of permutations of players and word scoring strategies.

5.2 Prerequisites

- The set of inputs to test. A list of permutations of players and scoring strategies to benchmark
- The number of repetitions or iterations to test each input
- Fully initialised trie service

5.3 Method

1. Make a space to record the runtime and winner of each repetition and the number of game failures for each input.
2. Create a game with players and scoring strategies.
3. Start a timeout and a process timer.
4. Run the game.
5. If the game fails to end before the timeout, increase the number of failures.
6. Otherwise, stop the process timer and record the CPU time and winner.
7. Repeat from the second step for each iteration and for each input.
8. Calculate the mean, median, and standard deviation for each input.
9. Calculate the frequency of each player type winning for each input.

5.4 Controls

We begin by noting our limitations when it comes to controlling other variables. Drawing tiles from the pouch by initial hand creation, peeling, and dumping is by definition random, and it is especially challenging to place a deterministic order for peeling and dumping as every player accesses the same pouch, and so each player is able to influence every other player. Our choice to model concurrent real-time game-play using multi-threading means that the the runtime and order of each player is at the mercy of the operating system's scheduler. While it should not affect overall results, it is still very challenging to control real-time game-play to have identically on a case-by-case basis. The multi-threading factor also bleeds into the former issue, as if two players attempt to peel and dump, respectively, at approximately the same time, the scheduler may allow the first player to go first sometimes, and the second will go first some other times, making it near-impossible to make the results of peeling and dumping replicable. To compensate and make sure our results are accurate, we have maximised the number of repetitions of benchmarking that our machines were capable of.

We note the use of the Pseudo Player, which is a dummy entity that does nothing except receive tiles when another player peels. In a real game of Bananagrams, a player will not receive every tile, and the Pseudo Player brings our benchmarking closer to realism. The Pseudo Player is not without flaws, as real players may choose to dump but the dummy does not.

5.5 Real-Time Performance

We have not chosen to measure the runtime of each individual player under the assumption that the operating system's scheduler is, at least on average over a large sample, equally generous.

Benchmarkers can choose to measure either wall clock time or CPU time. Wall clock time is the difference between the time one sees on the wall clock when starting the measurement and the time one sees on the wall clock when terminating the measurement. It is, in other words, real-time measurement. This sort of measure includes the time the machine spends on other threads, other processes, and the time the target process spends waiting on actions like I/O operations. On the other hand, CPU time, also known as the sum of user and system time, is only the time the machine's CPU spends executing the target process's instructions. CPU time is very favourable for us as this it does not count the time occupied by other processes, other threads, and the game and each player printing information to STDOUT. Time spent printing to STDOUT is especially detrimental to measurement as each player prints large quantities of text at very high frequency, and if a player needs to print, they must for every other player to finish printing since editing STDIN and STDOUT must be done synchronously.

5.6 Other Threats to Validity

One downside of the way we have integrated all of our algorithms to make them easy to test is that most of the infrastructure required for any given player or scoring strategy is then used for each player, even if it isn't utilised. So stranding approaches still maintain a more complex than necessary word lookup (more examples)... While none of the algorithms try to be fast through being much lighter weight than the others, this may still impact the performance of some algorithms more than others.

6 Results

6.1 Comparison within Stranding

TABLE V: Time taken for different algorithms/word scorers in a 2-player game over 1000 games (any combinations with ≥ 15 fails is excluded)

Players	Scorers	Avg. (ms)	Median (ms)	Standard Deviation (ms)	Fails
Pseudo Stranding	Dead Letter Avoidance	100.28565	93.2914	55.7881	1
Pseudo Stranding	Simple Stranding	131.60462	122.69095	50.59741	4
Pseudo Stranding	Letter Count	116.03044	107.2627	63.72227	1
Pseudo TwoLetterStranding	Dead Letter Avoidance	99.64881	92.3163	59.46733	0
Pseudo NewStranding	Dead Letter Avoidance	116.62931	108.7404	59.15012	7

TABLE VI: Time taken for different algorithms/word scorers in an 8-player game over 1000 iterations (any combinations with ≥ 15 fails is excluded)

Players	Scorers	Avg. (ms)	Median (ms)	Standard Deviation (ms)	Fails
7x Pseudo 1x Stranding	Dead Letter Avoidance	150.82733	81.2215	167.31206	6
7x Pseudo 1x Stranding	Letter Count	51.32842	48.22273	25.9376	7
7x Pseudo 1x TwoLetterStranding	Dead Letter Avoidance	151.99204	104.9494	99.02723	6

From the above tables we can see that on their own in a two-player context the two fastest algorithms are the Stranding and TwoLetterStranding algorithms with the Dead Letter Avoidance Word Scorer (although the regular Stranding Algo failed once), however, in an eight-player context, the stranding player using the Letter Count Word Scorer is three times as fast as any other algorithm (although it fails 1 more time).

First, it is interesting that NewStranding, theoretically the best algorithm we created performed badly. It is identical to TwoLetterStranding, except that if there are enough anchors, it will run **sorted_trie.all_subwords** on its hand once and find the first word that fits rather than running **forward/reverse_trie.all_subwords** on each anchor. This suggests some combination of a badly chosen threshold for when to use which algorithm and that there aren't enough tiles for the additional word search technique to be of much value. It would be interesting if, despite this worse performance, the second word search would lead to better asymptotic properties. It is possible if there were far more tiles in the game, this algorithm would benefit the most from the additional anchors.

It is also notable that while no scorer is significantly stronger than any other in the 2 player instance, the Letter Count scorer appears to be far better than any other in the 8 player instance. This again is an illustration of the trade-off between good asymptotic properties and a greedier approach. While Dead Letter Avoidance is strongly focused on avoiding accumulating a hand with bad letters in the long run, when there are 4 times less tiles per player, simply optimising for playing long words quickly with a heuristic for tile value less heavily weighted towards playing a small number of difficult tiles seems to work better.

The TwoLetterStranding player and the default Stranding player were additionally tested for 250 iterations and the TwoLetterStranding player won 52% of the games where the Stranding player won 48%. Running the same 250 iterations except with the order swapped so that the TwoLetterStranding player would receive the hands of the

Stranding player and vice versa, the TwoLetterStranding player won 55% of the games where the Stranding player won 45%. These results on top of the solo runs indicate that the TwoLetterStranding player is slightly faster than the default Stranding player.

6.2 Non-Stranding vs Stranding

In 250 runs, the Standard Player (with Dangling tile handling) had 12 failed attempts and took an average of 225 ms to complete its games whereas all stranding players finished significantly faster and with significantly fewer fails (with the same word scorers). Additionally, when the stranding player competed against the standard player, the stranding player had a 73% win rate against the standard player.

7 Variations

This section is dedicated to ways to vary the game's rules and parameters which may be worth investigating in the future.

7.1 An Aggressive Approach

We have considered an alternative approach where the player immediately dumps repeatedly until there are less than three tiles in the pouch. This kind player would take extra time to implement and benchmark due to its aggressive and malicious play style. The aim of the player is to prevent other players from ever having a complete board by minimising the number of tiles in their hand as a smaller hand is more likely to be impossible to fully decompose into words. Conversely, the player themselves has the biggest hand and so is, theoretically, most likely the only player that could win. It follows that the aggressive player may not even have to rush to complete its board.

7.2 Realistic Words

The Scrabble dictionary is very generous with regard to what counts as a word. While this is logical for official Scrabble tournaments, most of the words in the dictionary aren't used in modern communication, which alienates our computer players from a human player. Exploring how computer players play with a realistic set of words by, for example, using only the ten thousand most common words, may shed insight on the best ways a human could play and the nature of the words used by humans. By using words that humans are likely to know, it may also be easier to isolate and illuminate ideas about the difference between computers and humans when it comes to recalling words in various ways. It is likely that the Stranding approach would fare far worse once most two letter words ceased to be valid.

7.3 Playing In French

Using a French dictionary could shed insights into the similarities and differences between human languages with similar language systems. It may be interesting to see how assumptions that were true for the English language fall apart when tested with the French language, and consequently how performance changes across all the players as each player has made different types and levels of assumptions about their language.

7.4 Cost of Moving Tiles

To make the game engine's decision more realistic future work could add a associated cost to picking/moving tiles. This would mean that the engine should prioritise making good decisions early on rather than backtracking. Furthermore, a consideration of taking longer to decide which word to play against the cost of reversing that decision should be considered.

8 Conclusion

8.1 Remarks

This report set out to improve on this existing progress in Bananagrams engine by incorporating real game play mechanics with multiple players and increasing rate of success through methods such as backtracking. The work of Agrawal and Kwok had a 80% success [1] rate while the best run of the Two Letter Stranding approach discussed in this report did not fail once in a thousand iterations (see Table V), showing a drastic improvement.

Multiplayer with other active players was also incorporated and lastly the engine also makes use of the game mechanics such as dump which both were not evident in previous work.

8.2 Potential Future Work

We have discussed ways to improve individual aspects of our algorithms throughout the text. They were generally fairly well-defined improvements, where our only constraint preventing us was time. The below extensions are far more broad options with many possible approaches to how they could be implemented.

Language Choice Python is an excellent language for rapid development and allowed rapid progress on this project. However, this has an associated cost to runtime. Possible future work could employ a different language such as Rust to improve runtime and also safety.

Extending Words The available ways to play tiles can be expanded by considering cases such as extending words through prefixing, suffixing or incorporating letters into the middle of a word. For example, a player may have an 'L' in their hand with no way to build a new word from it but if the word "SANG" is on the board this could be turned into "SLANG". If doing so the structure of the board should be considered carefully since this may mean numerous tiles attached to the word may have to be shifted resulting in a potentially significant time cost.

Awareness of Other Players A use of Game Theory could be considered to adapt based on how other players are acting. While each player builds their own board, if another player has dumped many 'Q's it may be better to take more time using the tiles the player already has rather than dumping with high probability of receiving a 'Q'.

Dynamism A dynamic player class could also be considered for different game environments. Since Two Letter Stranding was the fastest approach in a two player environment while Stranding was better in an eight player environment so perhaps a dynamic player could be used to adapt to the different environments. Furthermore, strategies may excel in different stages of the game so switching between these strategies based on the number of tiles played or remaining may increase overall performance.

Stranding More Right Angles The use of right angle plays in stranding allows access to more anchors for stranding. Since if no right angle word was played, there are only two anchor from which the strand can be extended. Future work may prioritise playing right angle words to increase the likelihood of being able to strand without the need to restructure.

Machine Learning The aim of this project was to development an algorithmic approach to solving Bananagrams. However, machine learning may offer insights into more optimal strategies that improve speed, especially since there are no elegant solutions and heuristics play a strong role.

References

[1] Saahil Agrawal and David Kwok. *CS221 Final Project: Bananas for Bananagrams*. 2018. URL: `https://github.com/pyrobluestar/bananagrams/blob/master/final.pdf`.

[2] Andrew W Appel and Guy J Jacobson. "The World's Fastest Scrabble Program". In: *Communications of the ACM* (1988).

[3] *Bananagrams*. 2016. URL: `https://grady.dev/project/bananagrams/`.

[4] Bananagrams. *Our Mission*. 2023. URL: `https://bananagrams.com/pages/our-mission`.

[5] Steven A Gordon. *Software Practice and Experience*. Vol. 24. Department of Mathematics, East Carolina University, 1994.

[6] Chris Piech. *CS106B*. 2016. URL: `https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1164//handouts/bananagrams.html`.

Appendices

A Completed Board of Dangling Player in Single Player

```
   3 2 1 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
 -7                        N
 -6                      K Y
 -5                    K I     A
 -4                  N U     E R
 -3                    V R O W
 -2                  P A
 -1        N   O   A   S   E
  0      P U L V E R I Z E R
  1    G       E     F   M
  2  Q U A V E R E D     B A
  3X U         M   O   J A
  4  O     F L A W       R
  5  M   P E   N         C O G O N
  6  O   I   T Y G     J A   I
  7A D           I       D E         S T
  8  O             S     E   R     S   E
  9                T H I R T E E N T H L Y
 10            X         O     R       A
 11          B U L L D U S T           N
 12        D I     I   N               G I
 13                                    I   I F
 14                                  W E     E
 15                                    C E A Z E D
 16                                C   T       T
 17                            B O H E A
 18                            O       S T
 19                                  Q I
```

B Completed Board of Stranding Player in Single Player

```
  8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
-3                                                    S A V
-2J A X I E                                       O W T
-1        F I B R O                 E U K A R Y O N
 0                P U L V E R I Z E R     W     O
 1                  M                     D     F
 2                H M                   A L
 3            P O O                     T
 4                N                     Y
 5                K                     P
 6                E                     I
 7                R                     C
 8              E S                     A
 9              M                       L
10              B                       I
11              A                       T
12              R                       Y U
13              G                         N
14            T O                         D
15            E                           E
16            G                           R
17            U H                         I
18              O                         N
19              L                         F
20              I                         L
21              B                         A
22              U                         T
23            E T                         E
24            S                           D
25            C
26            H
27            E
28            A
29            T
30            A
31            G
32          D E
33          E
34        Z E
35      T O
36        U
37        A
38        V
39      D E
40      E
41    N E
42    O
43    N
44    T
45    O                       E A R W I G
46    X           Q I N D A R S
47    I       A J I
48    C I N Q S
49    S
```

C Larger version of repository structure

```
.
|-- analysis/
|-- assets/
|   |-- starting_lettersx10.txt
|   |-- starting_lettersx100.txt
|   |-- starting_lettersx1000.txt
|   `-- word_dictionary.txt
|-- src/
|   |-- MiscWIP/
|   |   `-- play_stage_one.py
|   |-- Not in Use/
|   |   |-- Classes/
|   |   |   |-- hash_dict.json
|   |   |   `-- hash_dict.py
|   |   |-- Game_Structure/
|   |   |   `-- AVLTree/
|   |   |       |-- avl_tree.py
|   |   |       `-- test_avl_tree.py
|   |   `-- player.py
|   |-- board/
|   |   |-- board.py
|   |   |-- lims.py
|   |   `-- tile.py
|   |-- pickles/
|   |   |-- all_words.pkl
|   |   |-- board.pkl
|   |   |-- forward_words.pkl
|   |   `-- reverse_words.pkl
|   |-- players/
|   |   |-- PseudoPlayer.py
|   |   |-- StandardPlayer.py
|   |   |-- StandardDanglingPlayer.py
|   |   |-- StrandingPlayer.py
|   |   |-- TwoLetterJunkStrandingPlayer.py
|   |   `-- player.py
|   |-- algorithms.py
|   |-- benchmarker.py
|   |-- constants.py
|   |-- dangling_experimentation.py
|   |-- game.py
|   |-- lims_algos.py
|   |-- main.py
|   |-- parent_word.py
|   |-- pickle_manager.py
|   |-- pouch.py
|   |-- race.py
|   |-- test.py
|   |-- trie.py
|   |-- trie_service.py
|   |-- two_letter_junk.py
|   |-- utils.py
|   `-- word.py
|-- tests/
`-- run_test.py
```