Solving Boggle in Haskell

And related algorithms, data structures

Charles Capps

March 14, 2012

Contents

1 Introduction

The goal of this project is to analyse different methods of solving Boggle using Haskell. The space and time complexity of the different methods is explored both experimentally and mathematically. The runtime as a function of the board size, N, the maximum word size, M, and the size of the dictionary, D, is discussed. The code to construct the underlying data structures from an input dictionary is explained. A program is written to automatically measure the runtime of the algorithms using the `CPUTime` module. Some `QuickCheck` tests are also written to verify desired properties of the algorithms.

Two different data structures were used: a hash table and a trie. A Boggle board is represented as a 2D array of `Char`'s. In Sections 2.1-2.3, the construction of these data structures is discussed. Some measurements of how efficient the hash table is in terms of space (percentage of buckets that are empty, maximum bucket size) are also discussed. The memory usage of these data structures was determined experimentally and is discussed in Section 2.4.

Section 3 covers how the algorithms work. Some code is presented, but all the code for the project can be found at `https://github.com/charlescapps/Boggle-Solver-Haskell` and important snippets are in the Appendix. Graphs of the actual runtime are also presented in sections 3.1-3.2.

In Section 4 some facts about the runtime are derived mathematically. Unfortunately, some facts we'd like to prove were too difficult to show. However, we do derive several non-trivial, interesting results, and we make some educated guesses for how the runtime varies as the size of the dictionary changes. The effect of changing the contents of the dictionary (distribution of word sizes, etc.) is discussed qualitatively.

2 Data structures

I use the convention that `Boggle*.hs` is for names of modules/files, and `Bog[Structure]` is for names of data types.

2.1 BogGame: Representing a boggle game

The data type for a Boggle Game has one constructor that takes the size of the board and an array with `Ix` type `(Int,Int)` that holds Chars. We also have a function to create an arbitrary NxN game from an Int and a String (the String has the boggle board with spaces separating each character and newlines separating rows). Using an Array here is key so that we get O(1) access times. We never need to use mutable arrays; the `Data.Array.accumArray` function makes it convenient to build immutable arrays when we can't get all the data in order.

```haskell
--size of board, then a 2d array with chars
data BogGame = BogGame Int (Array (Int,Int) Char)

--read NxN game from a string
readGameN :: String -> Int -> BogGame --construct arbitrary size game
```

Listing 1: BogGame data type

2.2 BogHash data structure

A hash table was used to quickly check if words are in the dictionary. The hash function turns a word into a 26-bit integer where bit n is present IFF the n^{th} letter of the alphabet is present. Here is the code for this function. It uses the `Data.Bits` module.

```haskell
--Computes the hash *before* taking the mod.
wordHash' :: String -> Int
wordHash' [] = 0
wordHash' (c:cs) = charToBit c .|. wordHash' cs

-- 'a'-> 1, 'b'-> 2, 'c' -> 4, 'd' -> 8 ...
charToBit::Char -> Int
charToBit c = shift 1 (fromEnum c - fromEnum 'A')
```

Listing 2: BogGame data type

To reduce the size of the hash table from $\approx$ 256MB to $\approx$ 4MB we then take the hash value module 2^{20}. I chose this number after graphing the maximum bucket size in the hash table vs. the size of the table. We want to keep the maximum bucket size small, because that indicates the max time it could take to find a word in the table (the time to traverse a bucket). However, we also want to reduce the size of the hash table in order to reduce the overhead of the program and to potentially speed up the program due to better spacial locality.

There's only a $\approx$ 70% increase in maximum bucket size when we reduce the table size from 2^{26} down to 2^{20}. There's another increase of $\approx$ 70% from 2^{20} down to 2^{19}. So 2^{20} is a good compromise.

As we will see below in the runtime section, a hash table isn't very well suited to Boggle. A trie is much faster since we can prune branches based on whether a string is a prefix of any word in the dictionary.

See Figure 1 below for a graph of bucket size vs. table size. See Figure 2 for some exact data points.

The maximum bucket sizes reflect the structure of the dictionary so it would be difficult (if not impossible) to figure this out without experimenting.

The fraction of non-empty buckets was also determined for different table sizes. This is a measure of how efficient the data structure is in space. Since the hash function doesn't take into account the structure of the dictionary, there will inevitably be empty hash buckets. See Figure 3.

3.86% of the buckets are used with a table size of 2^{19}. This is very sparse, but it is worth the tradeoff of space for $O(1)$ performance. We could investigate other hash functions that reduce the space used and have similar maximum bucket sizes. The performance of the hash function is highly dependent on the data used, but it may be worth the trouble since for games such as Boggle and Scrabble the dictionary is fixed.

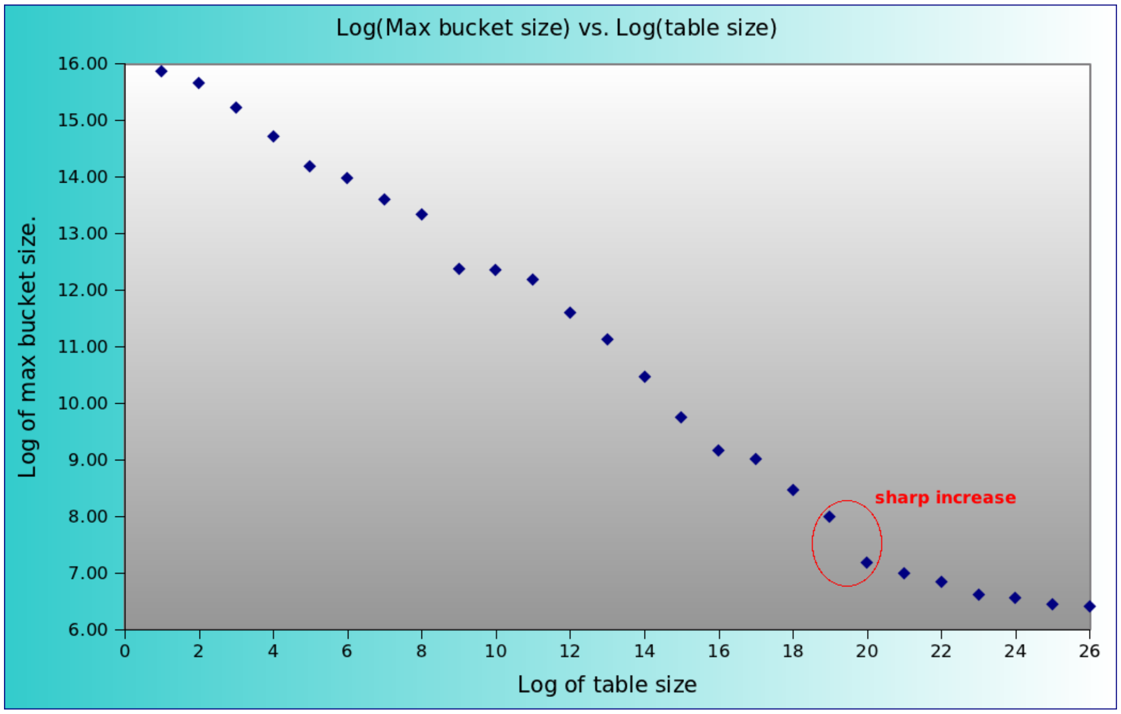

Figure 1: Log of max bucket size vs. Log of table size

Log(table size)	Max Bucket Size
19	254
20	144
21	127
22	114
23	98
24	94
25	86
26	84

Figure 2: Some sample datapoints of table size vs. max bucket size

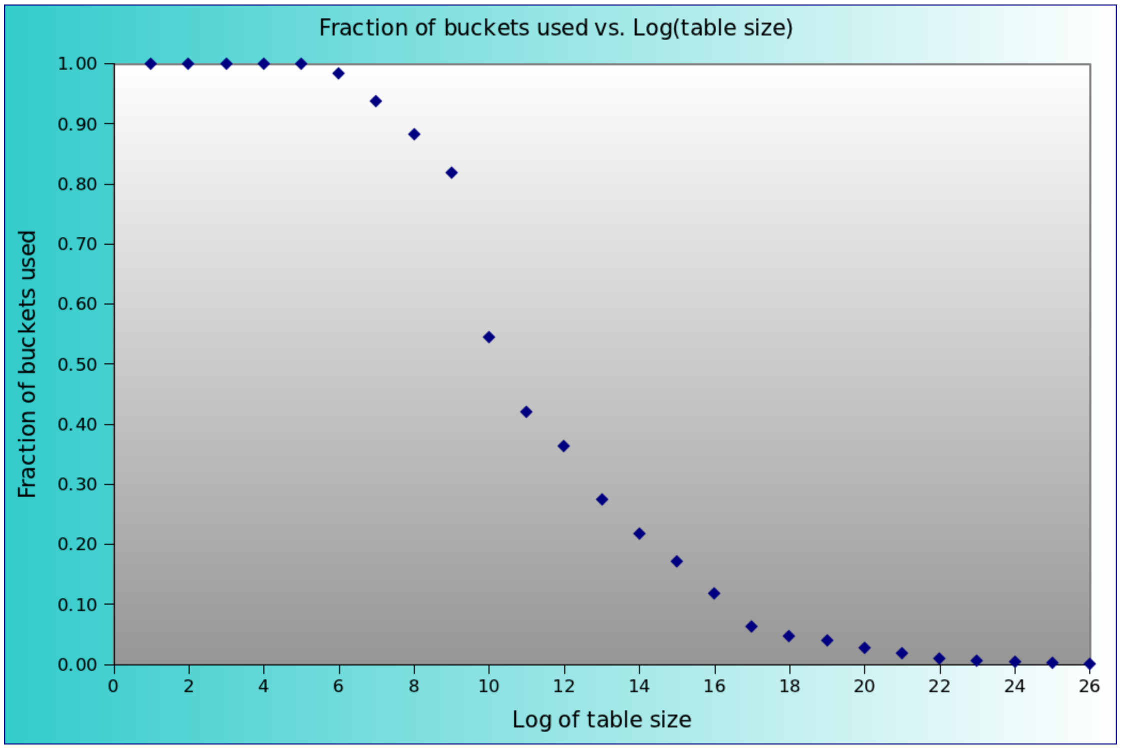

Figure 3: Fraction of non-empty buckets vs. table size

2.3 BogTrie data structure

A fellow student [1] mentioned that the Trie data structure would be perfect for Boggle. Wikipedia has a good explanation of tries. I decided to not store the String that a node represents explicitly in each node, because we know what the String is implicitly based on the path from root to a node. Instead each node holds a `Bool` indicating if the String is in the dictionary.

```
--realWord == True iff the word defined by the path so far is in the dictionary.
data BogTrie = BogTrie { realWord :: Bool, branches :: [(Char,BogTrie)] }
--Labeled version. Is an instance of LabeledTree so we can visualize small tries for testing
data LabelTrie = LabelTrie {lRealWord::Bool, hash::String, lBranches::[(Char,LabelTrie)] }
```

Listing 3: BogTrie data type

The bulk of the work, however, was to parse a dictionary into the Trie structure efficiently and to use the trie to find all moves on a board (see section 2.2).

To increase my confidence that the implementation actually works, I also created a LabelTrie structure that is an instance of the LabeledTree class from Treedot. The labeled Trie *does* store a String in each node for display purposes. The labeled trie also includes T or F on each node's label indicating if the String is in the dictionary. I generally used the Official Scrabble Player's 2nd Edition dictionary because it was freely available from `infochimps.com` and has far more words (117,969). However, I also found a basic version of the boggle dictionary and my command line programs can dynamically select the dictionary used.

The `LabelTrie` structure was used to get a visualization on a small set of data. I used the first 30 words in the Official Scrabble Player's Dictionary to build a small trie. (Yes, "AA" is a lavaflow and "AAL" is an East Indian shrubbery, these aren't bugs in my algorithm.) Note the 1 letter word "A" isn't included in this

[1] Greg Haynes

dictionary (since 1-letter words aren't allowed in Scrabble), so that node is set to False. See Figure 4 for the GraphViz output.

The algorithm to parse the dictionary into a trie works as follows.

1. Create a function `buildTrie` that takes a list of Strings, `words`, as input. Initially pass in the dictionary.
2. If the first String is the empty string ("") then set `realWord` to True and remove the empty String from the list (we exhausted a String so the path represents a word in the dictionary.)
3. Otherwise, set `realWord` to False.
4. Split `words` into `groupByFirstLetter::[(Char, [String])]` where each `Char` represents the first letter of a word, and each `[String]` contains the tails of Strings that start with that letter.

   For example, `["A","ABACUS","BAT","BARN"] --> [('A', ["","BACUS"]), ('B',["AT","ARN"])]`

   Then `["","BACUS"] --> [('B', ["ACUS"])]` etc.
5. Recursively call `buildTrie` by adding a branch `(c,buildTrie ords)` for each `(c, ords)` in `groupByFirstLetter`. (We humorously call the variable `ords` since we removed the first letter of each word in `words`.)

2.4 Memory Usage

A rough measurement of the memory usage for the two data structures was taken. The memory usage was found by creating a simple program `get_memory_usage` that just reads in the dictionary, builds the chosen data structure (or no structure), then pauses for the user to input a character. A challenge was to ensure the data structures were actually created — if they're not used then lazy evaluation will ensure they aren't even built. To get around this, I tried using the " $! " operator. Even so, I found that, depending on the code, a data structure might not be built (no space taken up in memory.)

To force evaluation, the `get_memory_usage` program writes the String representation of a data structure to disk. This is proof that the data structure was built in memory. Then the memory usage is measured after this operation is complete. The following assumptions are made: 1) The memory used by the data structure + dictionary is just the total memory used minus the memory used for a basic program that does nothing, 2) The dictionary takes up the same amount of space in memory as on disk, and 3) It's meaningful to subtract the memory used for the dictionary from the total memory to get an estimate of a data structure's "overhead" beyond the memory needed for the dictionary.

Data structure	Total memory usage	Data structure + dictionary	Data structure overhead
None	404 kB	0 kB	0 kB
TRIE	4.4 MB	4.0 MB	2.9 MB
HASH	33.8 MB	33.4 MB	32.3 MB

Here the dictionary file takes up 1.1 MB on disk. It was surprising that the trie used so much less memory than the hash table, because it stores so many extra nodes along the path to form a word. However, a trie only implicitly stores the words in the dictionary based on the path to a node. Also, a trie only stores exactly the paths necessary to define the words, and there's "sharing" between words that have the same prefix. The hash table, however, has 96.14% empty buckets, so this is a believable result.

So the trie structure has two advantages: it allows our algorithm to trim branches of computation that couldn't form valid words, and it takes up less space in memory. Taking up less space could also improve performance due to better spatial locality.

Interestingly, the hash table takes up significantly more memory than measured for a similar hash table implemented in Java. This may be due to the implementation of arrays in Haskell.

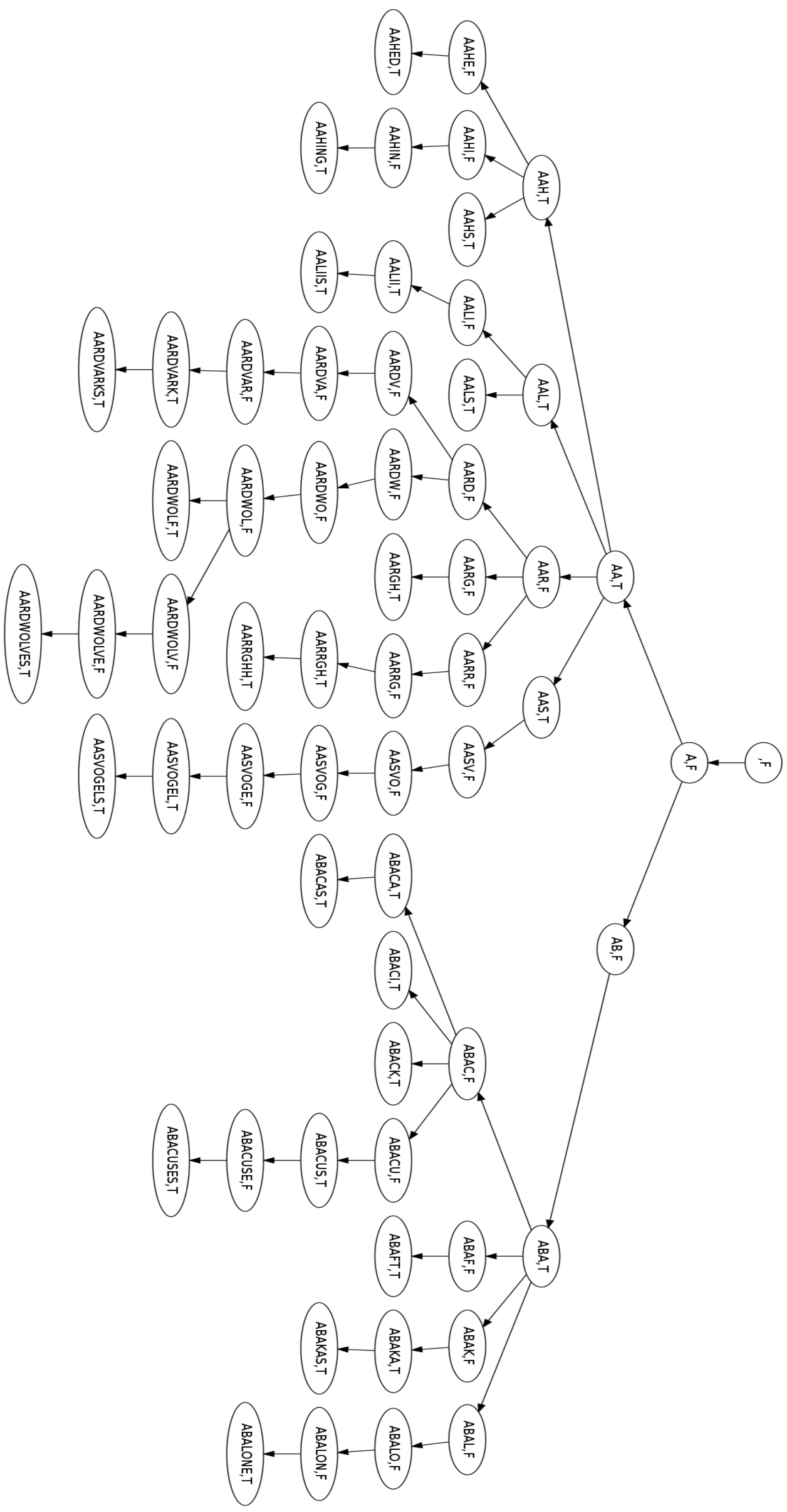

Figure 4: LabelTrie using the first 30 words in the dictionary. T/F indicate if the node is an actual word.

3 Boggle Solver Algorithms

Having defined our data structures and successfully parsed the dictionary into these structures, we can now describe some algorithms for solving Boggle. We represent a Play as a pair with 1) the path taken and 2) the word formed. Here is the Haskell data type:

```
type Play = ([(Int,Int)],String) --path in the matrix and the word formed
```

3.1 Boggle Solver Algorithm using Hash Table

First an algorithm using the `BogHashTable` data structure was used to find valid Boggle moves. To do so we use a function to find all valid moves starting at an arbitrary cube (i, j), then run this function on all of the $\mathbf{N^2}$ cubes and concatenate these solutions. Finally, we have to filter for duplicate words (arbitrarily—your score isn't affected by where the word is on the board!)

The idea of the algorithm is to recursively build a path by trying adjacent cubes we haven't already visited. As we go along letters are added to this path. When the recursive calls complete all of these paths that represent words in the dictionary are added to a list. This method is somewhat brute force, because it calculates every possible path that doesn't visit the same cube twice and doesn't go out of bounds.

As a simple example, suppose we have this board:

```
  0 1 2 3
0 A C F G
1 D W Z X
2 E Q R T
3 F S E E
```

Then if we are finding all moves starting at (1, 1) with length 3 or less then one execution path would look like this (in informal notation) :

```
Call getAllPlaysAt (1,1) ([],"") --empty move, immediately add [(1,1)], 'W' to move
Call getAllPlaysAt (2,0) ([(1,1)],"W") --since we haven't visited (2,0) yet
Call getAllPlaysAt (1,0) ([(1,1),(2,0)],"WE")

    Add ([(1,1),(2,0),(1,0)],"WED") to move list since "WED" is in dictionary.
    Remaining length is 0 so terminate
```

A program was written to solve 1 random board of a chosen size with a chosen algorithm. The usage is:
`Usage: solve_random "dictfile" ("HASH" | "TRIE") "BOARDSIZE" [MAX_WORD_LENGTH]`

Here's an example solution for a small board. Note for the hash table algorithm we have to specify the maximum word size to search for. Otherwise the algorithm has no halting condition except for searching every possible path on the board! We could automatically use the maximum word size in the dictionary (21 in this case), but it takes a ridiculously long time to search for all words of length 21 or less! This call to `solve_random` generates a random 5x5 board and uses the Hash algorithm to find all plays of length 6 or less.

This demonstrates a huge benefit of the trie structure. It only searches paths that can be prefixes of words in the dictionary, so the computation performed is greatly reduced. For most boards the hash algorithm can find all solutions, but for boards with words over 15 characters in length, it's prohibitively slow.

```
./solve_random dict/OSPDv2.txt HASH 5 6
Dictionary Input from: 'dict/OSPDv2.txt'
Max bucket size in hash table: 144
Max word size in dict: 21

Y Q L U N
F A A W X
V U Y C O
D A F H D
Y T W V L


1 POINT WORDS (Length 3)
LAW:  [(0,2),(1,2),(1,3)]
LAC:  [(0,2),(1,2),(2,3)]
LUX:  [(0,2),(0,3),(1,4)]
LAY:  [(0,2),(1,1),(2,2)]
LAV:  [(0,2),(1,1),(2,0)]
ALA:  [(1,2),(0,2),(1,1)]
AAL:  [(1,2),(1,1),(0,2)]
AWL:  [(1,2),(1,3),(0,2)]
AWN:  [(1,2),(1,3),(0,4)]
VAT:  [(2,0),(3,1),(4,1)]
VAW:  [(2,0),(3,1),(4,2)]
VAU:  [(2,0),(1,1),(2,1)]
CAY:  [(2,3),(1,2),(2,2)]
CAW:  [(2,3),(1,2),(1,3)]
COW:  [(2,3),(2,4),(1,3)]
COX:  [(2,3),(2,4),(1,4)]
COD:  [(2,3),(2,4),(3,4)]
OWL:  [(2,4),(1,3),(0,2)]
OWN:  [(2,4),(1,3),(0,4)]
OCA:  [(2,4),(2,3),(1,2)]
DAW:  [(3,0),(3,1),(4,2)]
DAY:  [(3,0),(3,1),(4,0)]
YAW:  [(4,0),(3,1),(4,2)]
YAY:  [(4,0),(3,1),(2,2)]
AVA:  [(3,1),(2,0),(1,1)]
AFT:  [(3,1),(3,2),(4,1)]
TAV:  [(4,1),(3,1),(2,0)]
TAU:  [(4,1),(3,1),(2,1)]
TAD:  [(4,1),(3,1),(3,0)]
TAW:  [(4,1),(3,1),(4,2)]
TWA:  [(4,1),(4,2),(3,1)]
FUD:  [(3,2),(2,1),(3,0)]
FAD:  [(3,2),(3,1),(3,0)]
FAT:  [(3,2),(3,1),(4,1)]
FAY:  [(3,2),(3,1),(4,0)]
WAD:  [(4,2),(3,1),(3,0)]
WAT:  [(4,2),(3,1),(4,1)]
WAY:  [(4,2),(3,1),(4,0)]
WHY:  [(4,2),(3,3),(2,2)]
WHO:  [(4,2),(3,3),(2,4)]
HOW:  [(3,3),(2,4),(1,3)]
HOD:  [(3,3),(2,4),(3,4)]
DOW:  [(3,4),(2,4),(1,3)]
DOC:  [(3,4),(2,4),(2,3)]

1 POINT WORDS (Length 4)
LAWN:  [(0,2),(1,2),(1,3),(0,4)]
LACY:  [(0,2),(1,2),(2,3),(2,2)]
LUAU:  [(0,2),(0,3),(1,2),(2,1)]
LAUD:  [(0,2),(1,1),(2,1),(3,0)]
LAVA:  [(0,2),(1,1),(2,0),(3,1)]
ACHY:  [(1,2),(2,3),(3,3),(2,2)]
WAUL:  [(1,3),(1,2),(0,3),(0,2)]
WYCH:  [(1,3),(2,2),(2,3),(3,3)]
YAWL:  [(2,2),(1,2),(1,3),(0,2)]
YAWN:  [(2,2),(1,2),(1,3),(0,4)]
CAUL:  [(2,3),(1,2),(0,3),(0,2)]
CHOW:  [(2,3),(3,3),(2,4),(1,3)]
COWL:  [(2,3),(2,4),(1,3),(0,2)]
COWY:  [(2,3),(2,4),(1,3),(2,2)]
DAFT:  [(3,0),(3,1),(3,2),(4,1)]
DAWT:  [(3,0),(3,1),(4,2),(4,1)]
DUAL:  [(3,0),(2,1),(1,2),(0,2)]
YAUD:  [(4,0),(3,1),(2,1),(3,0)]
WADY:  [(4,2),(3,1),(3,0),(4,0)]
WAFT:  [(4,2),(3,1),(3,2),(4,1)]
HOWL:  [(3,3),(2,4),(1,3),(0,2)]
DOWN:  [(3,4),(2,4),(1,3),(0,4)]
DHOW:  [(3,4),(3,3),(2,4),(1,3)]

2 POINT WORDS
ALWAY:  [(1,1),(0,2),(1,3),(1,2),(2,2)]

TOTAL SCORE: 69
```

This output was produced after removing duplicate words and computing the points of each play.

The pseudo-code for this algorithm is below. (See appendix for full Haskell code.) This code gets all moves starting from an initial row and column (i, j). The parameter `maxLen` is the maximum length of words to look for. As mentioned before, we have to give a maximum length since otherwise this algorithm has no halting condition. `currentPlay` is the play we are building up from the path we have taken so far. Initially we pass in the "empty play", `([], "")`

```
call getAllPlaysAt (game, (n,m), maxLen, ([],""), new list)

function getAllPlaysAt (game, (i,j), maxLen, currentPlay, goodPlays):
  if (maxLen == 0) //Base case
    return empty

  if (currentPlay.word is in the dictionary)
    Add currentPlay to goodPlays

  Let adjPlays = (plays formed by adding an adjacent cube (or the current cube) not in
      currentPlay.path to currentPlay)

  for each play in adjPlays
    Add getAllPlaysAt (game, play.path.last, maxLen-1, play, goodPlays) to goodPlays

  return goodPlays
```

Listing 4: Pseudo code for naive boggle solver

In this code, we add the `currentPlay` to the list `goodPlays` if it's in the dictionary. Then we recursively call `getPlaysAt` on adjacent cubes we haven't visited yet. The Haskell code looks different but this is the fundamental idea.

Then a binary `get_runtime` was created that runs the algorithm on random boards of different sizes. The usage looks like this:

```
Usage: get_runtime "dictfile" ("HASH" | "TRIE") "MIN N" "MAX N" "NUM_REPEAT"
"OUTPUT_FILE" "AVG_FILE" [MAX_WORD_LENGTH]"
```

For each boardsize between `MIN N` and `MAX N`, `NUM_REPEAT` random boards are generated and the runtime is measured. Raw data is written to `OUTPUT_FILE` and aggregate data such as the average and standard deviation of the runtime over all boards solved at each board size is written to `AVG_FILE`. The `MAX_WORD_LENGTH` is necessary for the HASH data structure, because we need to specify a maximum word length for the algorithm to terminate.

See Figure 5 for a graph of runtime vs. N. See Figure 6 for a graph of runtime vs. N^2. Each data point is the average runtime over 5 boggle boards. The error bars are the standard deviation. The `MAX_WORD_LENGTH` was arbitrarily chosen as 8 so that the algorithm could be run in a reasonable amount of time.

A quick inspection would suggest that the runtime is linear in N^2. This is correct and will be explored further in Section 3. The reason is that for a cube in the center of the board (more than `MAX_WORD_LENGTH` distance to the edge of the board) the runtime is bounded by a constant. The runtime is faster at the edge of a board, so overall the runtime is bounded by CN^2 for some C. C is only a function of `MAX_WORD_LENGTH` so it's constant for a fixed word length.

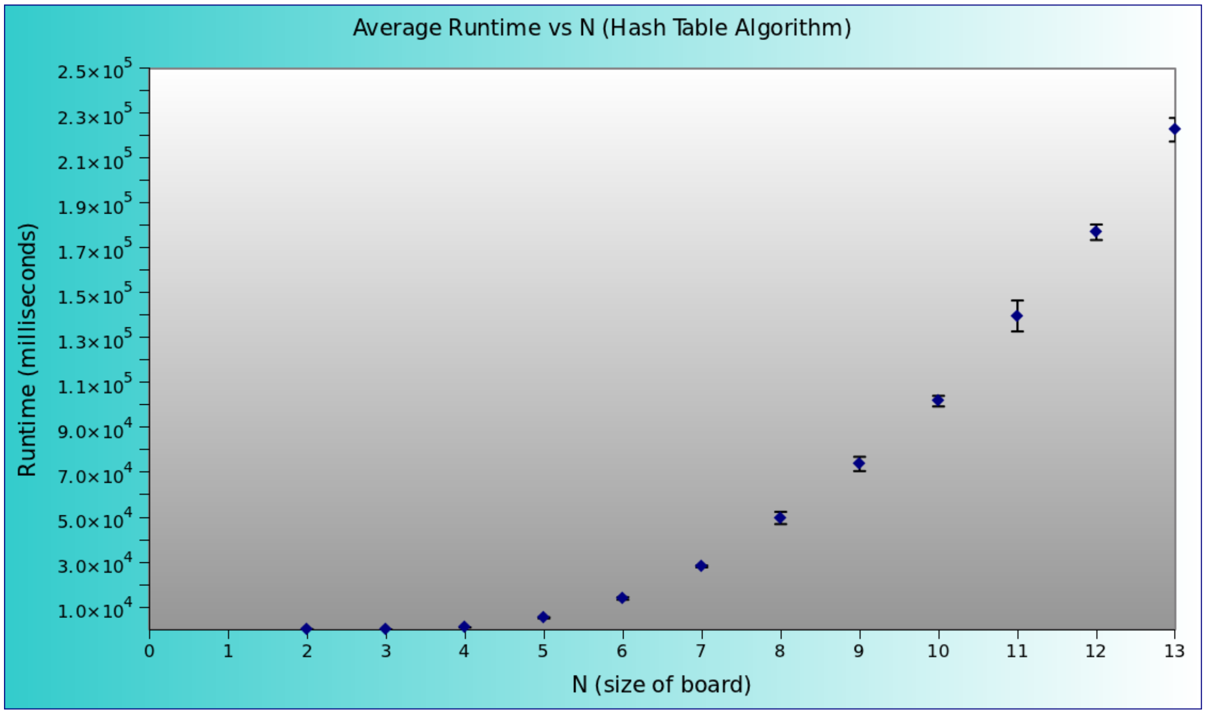

Figure 5: Average runtime (over 5 boards) vs. N

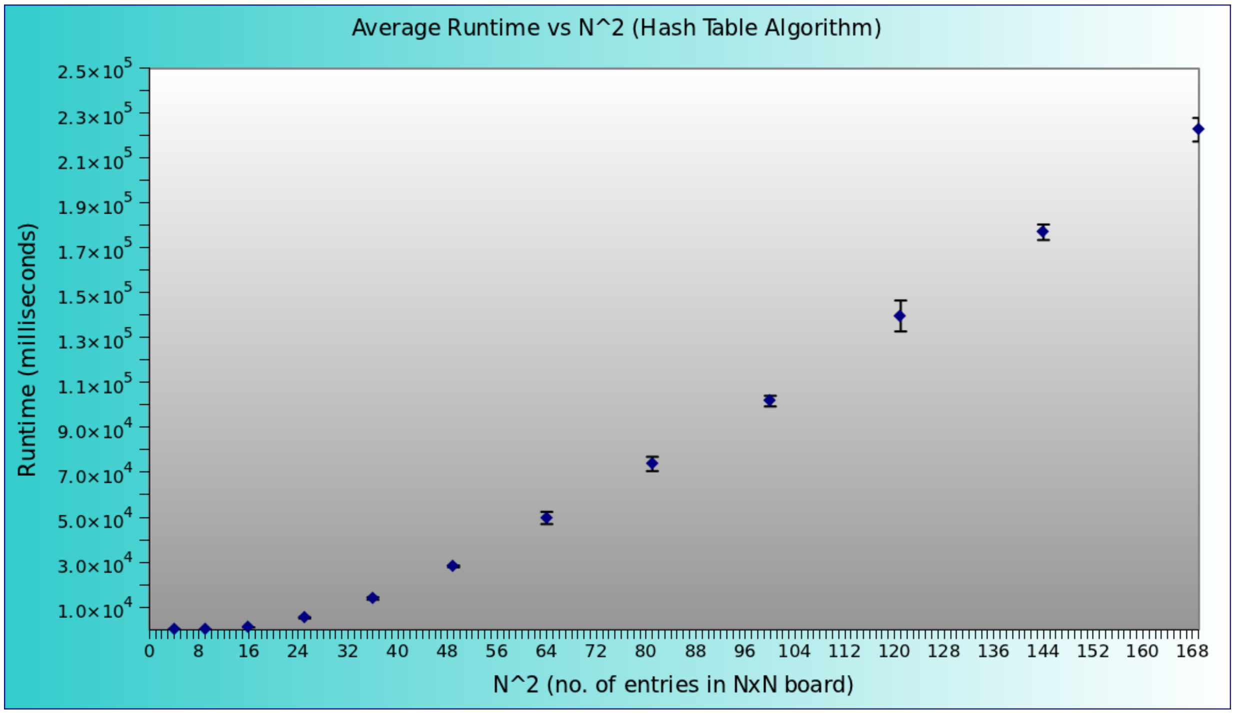

Figure 6: Average runtime (over 5 boards) vs. N^2, the number of entries in a board

3.2 Trie Search

Next I used the Trie datastructure to find all good moves more efficiently. This datastructure allows the algorithm to prune the search where a string fails to be the prefix of a word in the dictionary.

4 Runtime Analysis

There are several variables that are interesting to measure runtime against. We could do a multi-variable analysis of the runtime but that would be incredibly challenging! Define the following relevant variables:

- **N** : the number of rows in a boggle game (i.e. an NxN game.)
- **DICT** : the list of words in the dictionary, i.e. the actual content of the dictionary
- **D** = |**DICT**| : the number of words in the dictionary
- **M** : the maximum length of a word in the dictionary

Of course DICT determines D and M, but the converse is not true so it's useful to have these as separate variables.

4.1 Runtime as a function of the board size N, with a fixed dictionary

After a few weeks of working on this project, I realized that the runtime (for a fixed dictionary) has to be bounded by $\mathbf{O(N^2)}$. The basic idea is that the cost to find all moves centered at a position (i, j) doesn't depend on the board size, N. The Hash Table and Trie algorithms operate by concatenating the results from all N^2 positions, so we can then conclude that the runtime is $\mathbf{O(N^2)}$ by linearity of big-oh.

This almost seems trivial, but we really need to prove it formally, because the runtime could be affected by the borders of the board or the way the algorithms operate to find paths.

Proof: For a fixed dictionary, the runtime of the Hash Table and Trie algorithms is $\mathbf{O(N^2)}$

Let B be an arbitrary NxN Boggle board. Let (i, j) be an arbitrary position on the board. We'll show the runtime to find all moves starting at (i, j) doesn't depend on N.

It's enough to find an upper bound that doesn't depend on N, so we may as well make the board bigger. This should just increase the runtime of the Hash Algorithm since the runtime is proportional to the number of possible paths (it literally enumerates them). It should also increase the runtime of the Trie algorithm since it will need to check more paths that are prefixes of words in the dictionary (if it's near the border it would just do a constant time check to not go out of bounds.)

Expand B to a new $\mathbf{N'xN'}$ board so that there are M cubes in either direction from (i, j). In other words, (i, j) is now at the center of a $\mathbf{(2M+1)x(2M+1)}$ sub-board, let's call it G. Since the Hash and Trie algorithms don't even read cubes that are more than M positions away, they should perform the exact same computations on B' with N' as the input length as on G with 2M+1 as the input length.

In other words, in pseudo-code we've shown

$$\mathbf{TIME(getAllPlaysAt(B, (i, j), N)) \leq TIME(getAllPlaysAt(B', (i, j), N')}$$

$$\mathbf{= TIME(getAllPlaysAt(G, (i, j), 2M+1)}$$

But the runtime on the board G can't possibly depend on N since we gave 2M+1 as the input board size. There's one more thing we have to show. This bound could be different for different positions (i', j'). We need a uniform bound that works for any starting position on the board.

Let $\mathbf{R_{MAX}} = \max\{\mathbf{TIME(getAllPlaysAt(G, (i, j), 2M+1)}\}$ over every possible $\mathbf{(2M+1)x(2M+1)}$ board using letters from the fixed dictionary. This still can't possibly depend on N so it's uniform bound for runtime at any position (i', j').

Therefore, $\mathbf{TIME(solveBoggle(B))} \leq \sum_{\mathbf{i,j=0}}^{\mathbf{N}} \mathbf{R_{MAX}} = \mathbf{N^2 R_{MAX}}$
B was an arbitrary NxN board so we may conclude:

$\mathbf{TIME(solveBoggle(NxN))} = \mathbf{O(N^2)}$ for the Hash and Trie algorithms.

In fact, the algorithms must then be $\mathbf{\Theta(N^2)}$, because any algorithm has to at least read all $\mathbf{N^2}$ positions.

This proof should hold for any algorithm that doesn't perform unnecessary computation at the borders of a boggle board and operates by summing up the solutions from all N^2 positions. The algorithm must also not consider paths of length greater than M, the max word length.

References

[1] Wikipedia article, *Trie*
`http://en.wikipedia.org/wiki/Trie`

[2] Word List - Official Scrabble (TM) Player's Dictionary (OSPD) 2nd ed
`http://www.infochimps.com/datasets/word-list-official-scrabble-tm-players-dictionary-ospd-2nd-ed`

[3] Github Repository
`https://github.com/charlescapps/Boggle-Solver-Haskell`