⋐ Boggle Solver Algorithm ⋑

September 12, 2024

- Written By -
Nile Roth

- Professor -
Tayyaba Shaheen

Table of Contents

1 Algorithm Overview

1.1 runBoard()

This Boggle Solver Algorithm runs with the call of our main runBoard() function. The main function takes in two text files, the board and a dictionary. In order to work with the text contained in the board file, we must load it into a data structure that we can easily access/analyze (see 1.1.1). The same idea needs to be implemented with the dictionary as well. We need to bring the words in the text file dictionary into memory for future comparison (see 1.1.2). Now that we have both the board and dictionary loaded into memory we can begin our recursive search. In order to recurse without repeatedly loading in the board and dictionary, we must create a recursive subfunction. This function will handle the entire search process which starts by calling the recursive runBoardHelper() function on each element of the board. Because runBoardHelper() recurses with each neighbor of the current position, we ensure that all paths on the board are searched. All information about the recursive algorithm and how it works is presented in section 1.1.3.

1.1.1 | loadBoard()

loadBoard is an essential function that allows us to work with the board text file. The most useful data structure to store our board is a 2-dimensional array, so that we can access elements (letters) through a [row][col] format. This process simply involves opening the file, reading in each line into an array of characters using .strip().split(). More specifically, each line is stripped of its whitespace and split up into a list of individual letters, and because there are multiple lines (rows) in a file, the result is a list of lists (2D array).

1.11.2 | readDictionary()

readDictionary is the most simple function in the algorithm, for it only requires creating a 1D list. Similar to loadBoard(), after opening the file, we use Python's built-in readLines function to convert each line of the file into a list element. All that is left is to strip away any trailing whitespace from all words read in from the file. We accomplish all of this with a loop through the output of readlines along with a call to strip() for each element.

1.11.3 | runBoardHelper()

I decided to implement an embedded function so that it could share memory with the outer main function. This eliminates the need to pass in the dictionary, board, wordList, nor the wordCount into the recursive function. Instead, we only need to pass it a current position, along with a path of visited positions. We start off each recursion by examining our current state to see

if our path forms a word with our current position (see 1.1.4). If examineState() identifies a word, we add it to our wordList (a list of all words found on the board). Lastly we loop our recursive call with all possible neighbors. Possible moves consist of all neighbors of the current position that have not yet been visited (see 1.15 & 1.16).

1.11.4 | examineState()

This function uses the parameters of the recursive function (current position & visited path) to check if we have formed a word. To check for a word, we start by concatenating our path of letters into a string and appending our current position letter to that string. The string formed is our current state. Next, we simply check if this string is in our dictionary list. Depending on whether it is in the dictionary or not we return either ‘yes’ or ‘no’ along with the current string formed.

1.11.5 | possibleMoves()

This function takes in the 2D array board along with a current position on that board. It simply returns the neighbor positions of the parameter keeping in mind to not go off the grid. This is done by simply checking if the row/col value plus/minus one is greater than -1 or less than the grid size. In simpler terms, this function returns all neighbor positions of the parameter position that are on the grid.

1.11.6 | legalMoves()

legalMoves is basically an add-on to the possibleMoves function. We know that the grid is not our only boundary, we also cannot visit positions that have already been visited. For this reason the parameters of this function include both a call to possibleMoves as well as a list of visited moves. The idea is to remove all elements from the possibleMoves list that are also in the visited list. The result is all legal moves that a path can continue recursing through.

2 Testing

2.1 Results From 2x2 Board

2.1.1 | Board

```
T A
X P
```

2.1.2 | Output

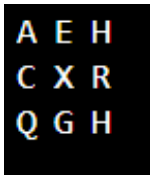
```
T A
X P

And we're off!
Running with cleverness ON
Searched total of 50 moves in 0.4803013801574707 seconds

Words found:
2-letter words: ['ta', 'ax', 'at', 'pa']
3-letter words: ['tap', 'tax', 'apt', 'pax', 'pat']

found 9 words total
Alpha-Sorted List:
['apt', 'at', 'ax', 'pa', 'pat', 'pax', 'ta', 'tap', 'tax']
```

2.1.3 | Key Results

50 paths
9 words
0.48 seconds

2.2 Results From 3x3 Board

2.2.1 | Board



2.2.2 | Output

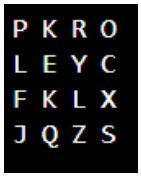
```
A E H
C X R
Q G H

And we're off!
Running with cleverness ON
Searched total of 356 moves in 4.602518081665039 seconds

Words found:
2-letter words: ['ax', 'ae', 'ex', 'er', 'eh', 'he', 're']
3-letter words: ['ace', 'axe', 'erg', 'hex', 'her', 'rex', 'rec']
4-letter words: ['rhea']

found 15 words total
Alpha-Sorted List:
['ace', 'ae', 'ax', 'axe', 'eh', 'er', 'erg', 'ex', 'he', 'her', 'hex', 're', 'rec', 'rex', 'rhea']
```

2.2.3 | Key Results

356 paths
15 words
4.60 seconds

2.3 Results From 4x4 Board

2.3.1 | Board



2.3.2 | Output

```
And we're off!
Running with cleverness ON
Searched total of 2256 moves in 26.623677730560303 seconds

Words found:
2-letter words: ['pe', 're', 'oy', 'or', 'er', 'ef', 'el', 'yo', 'ye', 'fe']
3-letter words: ['per', 'kep', 'kef', 'key', 'kye', 'rye', 'rep', 'ref', 'roc', 'ore', 'orc', 'lek', 'ley', 'els', 'elk', 'elf', 'yep', 'coy'
, 'cor', 'cry', 'fer', 'fey', 'lye', 'sly']
4-letter words: ['perk', 'pelf', 'kelp', 'ryke', 'rely', 'oyer', 'oryx', 'yore', 'yerk', 'yelk', 'yelp', 'cory', 'core', 'cork', 'clef', 'fle
y', 'lyre']
5-letter words: ['perky', 'coyer', 'corky', 'clerk', 'slyer']

found 56 words total
Alpha-Sorted List:
['clef', 'clerk', 'cor', 'core', 'cork', 'corky', 'cory', 'coy', 'coyer', 'cry', 'ef', 'el', 'elf', 'elk', 'els', 'er', 'fe', 'fer', 'fey', '
fley', 'kef', 'kelp', 'kep', 'key', 'kye', 'lek', 'ley', 'lye', 'lyre', 'or', 'orc', 'ore', 'oryx', 'oy', 'oyer', 'pe', 'pelf', 'per', 'perk'
, 'perky', 're', 'ref', 'rely', 'rep', 'roc', 'rye', 'ryke', 'sly', 'slyer', 'ye', 'yelk', 'yelp', 'yep', 'yerk', 'yo', 'yore']
```

2.3.3 | Key Results

2256 paths
56 words
26.62 seconds

2.4 Projections

2.4.1 | Runtime

With the results from sections 2.1-2.3, I used ordered pairs with x-values representing N (board dimension) and y-values representing time in seconds to create a system of equations. Each equation is created using the quadratic formula.

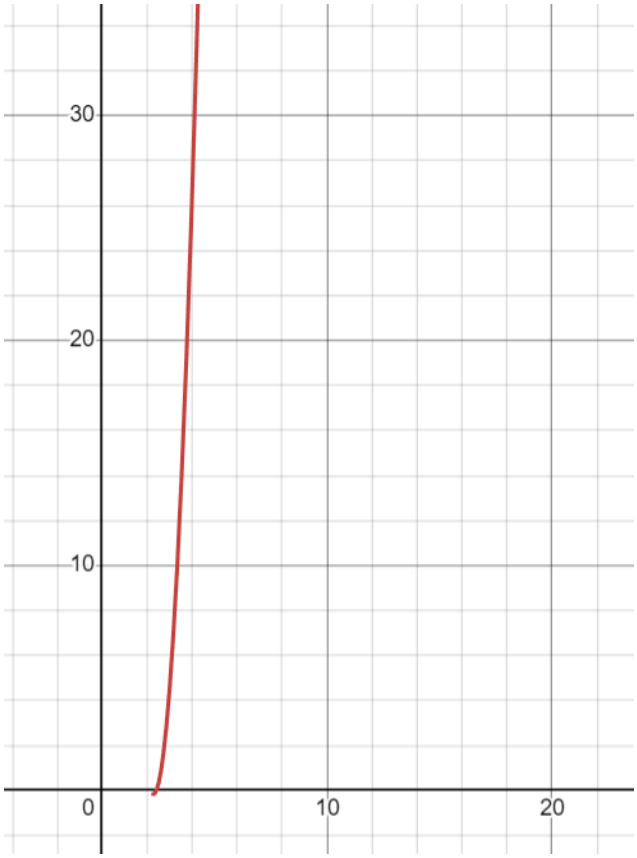
$$Y = ax^2 + bx + c$$

Ordered pairs:
(2 , 0.48)
(3 , 4.6)
(4 , 26.62)

Give us the following systems of equations:
4a+2b+c=0.48
9a+3b+c=4.69
16a+4b+c=26.62

Which when solved provides us with:

$$\boldsymbol{Y = 8.95x^2 - 40.63x + 45.94 \text{ for } x > 0}$$



Plugging in the value 5 for x will provide an estimated runtime for completing the search on a 5x5 boggle board.

Runtime for 5x5 board = $\boldsymbol{8.95}(5)^2 - \boldsymbol{40.63}(5) + \boldsymbol{45.94}$

$8.95(25) - 203.15 + 45.94$

$223.75 - 157.21$ = ***66.54 seconds***

2.4.2 | Search Count

With the results from sections 2.1-2.3, I used ordered pairs with x-values representing N (board dimension) and y-values representing number of searches to create a system of equations. Each equation is created using the quadratic formula.

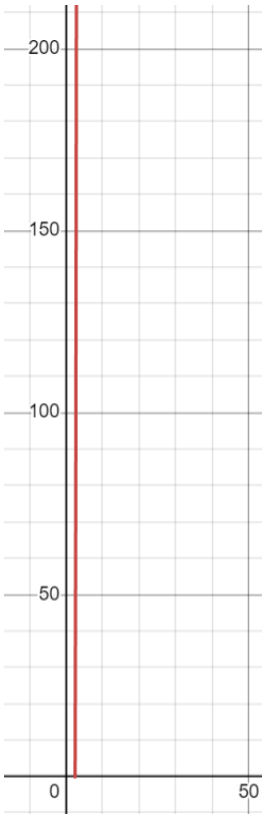
$$Y = ax^2 + bx + c$$

Ordered pairs:
(2 , 50)
(3 , 356)
(4 , 2256)

Give us the following systems of equations:
4a+2b+c=50
9a+3b+c=356
16a+4b+c=2256

Which when solved provides us with:

$$\boldsymbol{Y = 797x^2 - 3679x + 4220 \text{ for } x > 0}$$



Plugging in the value 5 for x will provide an estimated amount of searches for completing the search on a 5x5 boggle board.

Search Count for 5x5 board = $797(5)^2 - 3679(5) + 4220$

$797(25) - 18,395 + 4220$

$19,925 - 14,175$ = ***5,750 searches***

3 Problem Analyzation

Computing the number of possible combinations of letters from a boggle board is a very complex problem, but can be computed easily using our written recursive algorithm. To compute this value manually, it is more feasible to come up with a decent upper bound than to find an exact value.

To start lets disregard the borders and assume each element has 8 neighbors to traverse to. The longest path an element can take is NxN or N^2. With these two facts in mind lets first determine the number of 2-letter paths a single element can search. Because an element hypothetically has 8 neighbors, then each element has 8 ways of creating a 2-letter path. For 3-letter combinations, we now have 8 more neighbors to choose from for each of those 2-letter combinations. This gives us $8 * 8$ ways of creating a 3-letter path. We can now derive a pattern

2-letter paths: 8 ways per element

3-letter paths: 8^2 ways per element

4-letter paths: 8^3 ways per element

Because our paths can have a length up to N^2 we can write a simple summation equation with our findings above

$\sum_{L=2}^{N^2} 8^{L-1}$ Where L represents the length of a word, and N is dimension of the board

The equation above provides us with an upper bound for the amount of paths from a single element. In order to calculate the total amount of paths from all elements, we simply multiply this by the amount of elements on the board which is consistently N^2.

$$N^2 * \sum_{L=2}^{N^2} 8^{L-1}$$

This is a decent upper bound, but it does not take into account that not all elements have 8 neighbors. For this reason, this equation will be progressively more accurate as the dimension of the board increase (when more elements have 8 neighbors). When calculating number of paths for any board with a dimension $N \leq 5$ it is more accurate to change the number of neighbors from 8 to 6 (more accurate average of neighbors).

4 Problem Optimization

A rational player would discontinue searching a path if at any time it is not a prefix of another word. For example if a player is searching paths starting from element ‘j’, they would not waste their time traversing into the neighboring element ‘t’ because of how little words start with ‘jt’. In order to implement this, I simply added a conditional into the recursive function to check if our current sequence of letters is not a substring of any word in the dictionary. If it isn’t then we should stop searching that path. After adding this to my recursive function, I experienced a significant increase in speed. For a 4x4 board, the runtime diminished from 40 minutes to 30 seconds.

Another way a player could maximize their points is by prioritizing words containing more letters. Generally, a search through the board follows more of a Breadth-First algorithm (from a certain element searching all 2-letters words, then 3-letter words, and so on). When trying to get the longest words we can instead follow a Depth-First search. We can traverse a full N^2 long path and search for words embedded inside of this string. This gives us the opportunity to find longer words right from the start.

My last optimization recommendation is to search for post-fix letters and combinations to add to any found word. When finding any word, search for an ‘s’ neighboring the last element to get its plural (noun) or past tense (verb) version. We can also look for an ‘ed’ to add to the end of our word to make it past tense. When finding these letters to add onto a word, we increase the score for the given word by more than double.