Ethan Baranowski

CS-470 Artifical Intelligence

Assignment 1 – Fred Flinstone Problem Solving

09/17/21

2021

Assignment One – Fred Flintstone Problem Solving -- Boggle

**Critical Thinking**

Q: Observe your own results: Run your solver on several 4x4, 3x3, and 2x2 boards. How many different words did your solver explore on each? How much time was taken on each. Now analyze: Come up with a curve showing your results. Then use this to predict the time/moves it would take to explore a 5x5 board.

1. When I tested the above functions in an IDE, it corresponded to what was predicted given the provided test outputs. When operating on 4x4 board, I was able to find the remaining possible moves that could be taken. Given that these functions have a straightforward task, I see the result of say a 5x5 or 6x6 board operating the same. The fundamentals are consistent in terms of formatting the proper Boggle board.

['D U I T', 'N Q K Y', 'U A P G', 'N C H Y']

D U I T

N Q K Y

U A P G

N C H Y

Possible Moves:

[(4, 3), (4, 2), (2, 3), (2, 2), (3, 2)]

Total moves found:

5

When testing with a larger board, I obtained this:

['D U I T L P', 'N Q K Y G X', 'U A P G H E', 'N C H Y B V', 'A B G L Z A']

D U I T L P

N Q K Y G X

U A P G H E

N C H Y B V

A B G L Z A

Possible Moves:

[(4, 3), (4, 4), (4, 2), (2, 3), (2, 4), (2, 2), (3, 4), (3, 2)]

Total moves found:

8

Possible Moves:

[(3, 1), (3, 2), (3, 0), (1, 1), (1, 2), (1, 0), (2, 2), (2, 0)]

Total moves found:

8

Q: Analyze the problem generally:  How many possible combinations of letters (i.e. actual words or not) can be constructed from an NxN board? Walk through your reasoning carefully, showing how your value comes together. Let's keep it simple just to get a decent upper bound without needing a PhD in combinatorics: Ignore detailed paths possible on the board and just assume that every letter on the board could be chained with every other letter on the board...how many words could be made that way? How does your analysis match up with your empirical findings in the last question?

1. Finding a combination of letters comes down to where the current position within the game is located. In correlation with a lookup for legal words, possibilities increase the more tiles are openly available.

Q: Use your solver to solve at least 10-20 different boards, then ponder the solution stats you got. Based clearly on your observations, consider the following:  Suppose there is a Boggle competition where human players are given a sequence of boards to solve, and the time they have to do so decreases with each board.  Now examine the outcomes from the boards that you've run your solver on. What strategy for finding words would a "smart" (or as we'll call it in this course, "rational") player employ to maximize points in a time-limited time? Don't just speculate, support your answer clearly with your empirical results!

1. Upon trying the Boggle class with numerous boards, the number of possible moves increases drastically. In correlation this would allow the valid legal moves to also increase. The larger the board, the larger the possibilities. A strategy a “smart” player might utilize would be looking at each possible direction for their next combination. Avoid looking at illegal moves that will waste time in the game.

Q: In the sample output provided above, there are two runs on the same board shown, with/without "cleverness" turned on...with drastic differences in time/resources used to find identical results.  What the heck could that devious Dr. W be doing here to achieve this? Magic? Hint: put in some print statements to watch your program work...and then reflect on the implications of where effort is wasted.  The difference between the two outputs in my source code is exactly one line of code...plus another easily-created resource.

1. The difference in the two runs is time efficiency. When searching and validating words that don’t exist, the process becomes extended. If the same board is tested, but one has a reference and one does not, then the combinations without the references should be longer than the one with it.

Output:

2X2

![Graphical user interface, text, application

Description automatically generated]()

3X3

![Graphical user interface, text, application, email

Description automatically generated]()

4X4

![Graphical user interface, text, application, email

Description automatically generated]()

**Full Code**

**![Text

Description automatically generated]()**

**![Graphical user interface, text, application

Description automatically generated]()**

**![Graphical user interface, text, application, email

Description automatically generated]()**

**Algorithm, Approach and Strategy**

The primary goal of solving a Boggle puzzle is to pinpoint each possible letter combination to successfully form a word. The longer the word the more points generated, and the total points calculated after a particular time limit. There are a multitude of paths one may take to generate a word.

**Approach**

Taking a step back, we need to access the problem and prerequisites to our end goal. Here are a few things to consider before jumping into the code itself.

1. How is a Boggle board organized? ( N x N )
2. How will we display the board to the user?
3. Which words are correct or incorrect?
4. In what way will we verify a found word is in fact legal?
5. What will we reference to confirm that a letter combination is valid?

These questions are crucial to grasp the understanding of our primary goal. Thus, to generate as many possible letters to form a legal word. The greater number of rows and columns said board has, the longer the program will have to process these possibilities.

**Strategy**

With the questions mentioned above, we can then plan out a strategy to achieve this goal. First, we consider, where are we? Meaning, what is our location on the board and what direction can we take without visiting a previously selected tile. Additionally, what are the bounds of the board? When do we know when N number of rows and columns are met in reference to our current position?

The strategy I attempted to follow include conditional statements. Repeating a set number of tasks until a certain condition was met. Although this provided some struggles when referencing the location, I had already traveled to.

**Algorithm**

The algorithm I ended up going with will be shown in the following images.

*loadBoard:* For loadBoard I utilized the “with open” statement to open a filename that was given by the user. For testing, I used the fourboard3.txt file provided to ensure this was working properly. Within this function I also loop through the lines of the file and use split() and strip() to get rid of extra ‘\n’ characters, while formatting the list correctly. These results are appended to an empty list for the correct output of the board. This function returns a list of lists of the characters in the board.

![Graphical user interface, text, application, email

Description automatically generated]()

*printBoard:* This function is rather straightforward. It is a simple loop through the ‘myBoard’ variable that was previously saved in loadBoard. It then does the main formatting for the matrix. With the use of .split() and .join() I was able to separate each row once the last character was met in the subset of the main list found in loadBoard. This properly displays the N x N board the way it was intended.

![Graphical user interface, text, application, email

Description automatically generated]()

*possibleMoves:* This function is where recursion would take place, had I implemented it. For simplicity and problem solving, I needed to find each possible direction that could be made after making a previous move. With the use of conditionals, I was able to do this. Additionally, an empty list and a counter was created to keep track of the path taken and the total number of pairs found. This was to clarify that the list it produced was accurate. After a condition is met, I append that result to the empty list and return the empty list. To modify and further optimize this, I would utilize loops alongside these conditionals.

![Graphical user interface, application

Description automatically generated with medium confidence]()

*legalMoves:* When testing this function, I was unable to accurately produce results, but this is how I approached the method itself. I had already found the possible moves from the previous function, so therefore I simply needed to subtract the possible moves from the locations I had already visited. I used a pointer to the “possibleMoves” function to obtain the list that had already been produced. After subtracting, it would then output a list of legal moves that could be made from the location in space.

![Graphical user interface, text, application, email

Description automatically generated]()

*examineState:* For this function I wanted to use a simple lookup within the dictionary file I was given. This function would have taken in the existing board, a current position, a list of possible moves and the dictionary itself. I did not implement this function. In theory, it would have referenced the dictionary and responded with a ‘yes’ or ‘no’ for whether it was a proper entry or not.

![Text

Description automatically generated]()