Mind Boggling Speed

CP-3

Robert Boerwinkle

Spring 2023

Introduction

Boggle is a game in which the player attempts to make words out of a grid of letters. The player can string together letters which touch orthogonally or diagonally. Tiles cannot be reused, but paths can be self-intersecting as long as it is done diagonally. Typically it is played on a 4x4 board. How would one go about finding all the words in a given board? The obvious solution is to simply try every possible string of letters to see if it is a word. This, like many problems, can be simplified into a tree-searching algorithm. Among such algorithms, Breadth First Search (BFS) and Depth First Search (DFS) stick out for their simplicity.

BFS consists of searching nodes in order of depth. All nodes 1 layer deep are searched, then all nodes 2 layers deep, and so on. This will find the shortest words first. It requires memory proportional to the size of the widest layer of nodes. DFS always searches the first child node first. This finds sollutions in no particular order, but uses much less memory. This project explores both (in Python) with a couple optimizations along the way.

Naïve Approach

The implementations of DFS and BFS are very similar. Both searching functions keep track of which nodes have yet to be explored. When a node is explored, all of its children are added to the queue. DFS always takes the newest element, while BFS always takes the oldest. In the following pseudocode, `X` is the end of the list for DFS and the beginning of the list for BFS.

```
function search(board)
        queue = list of all starting positions
        while queue has elements
                current = pop X from queue
                run some function on current
                for child in current
                        add child to queue
```

For boggle specifically, the function to be run on the current node is to check if the current path is a word (see `dictionary.txt`). There are also some precautions against going off the side of the board or reusing tiles. This is implemented in `search_1.py` as `search`, taking an extra argument corresponding to `X`. This iteration surpassed 15 mins of runtime at a board size of 4 / 5.

DFS

board	words	largest queue	time (s)
1	0	1	1.8835067749023438e-05
2	9	7	0.0002923011779785156
3	41	23	0.03968381881713867
4	175	44	57.82228708267212
5			>900

BFS

board	words	largest queue	time (s)
1	0	1	1.1444091796875e-05
2	9	24	0.00020956993103027344
3	41	2995	0.04120635986328125
4			>900

Prefix Pruning

In order to trim down the search space, an additional condition for adding to the queue was implemented: the path has to be a valid prefix. A prefix in this case is just any substring that starts at the beginning of the original string. A dictionary for each length of prefix was constructed. If a node was not in the corresponding set, its children would not be added to the queue. This resulted in much more reasonable execution times, and significantly more interesting data. This approach (`search_2.py`) reused a lot of dictionary data, but the dictionaries by themselves took up less than 150 Mebibytes.

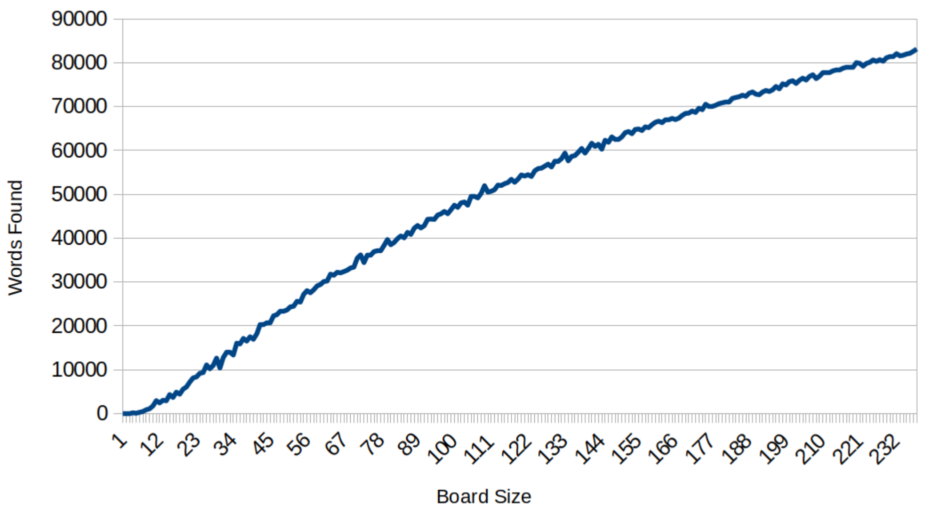

Figure 1: Total number of words found by board size (BFS and DFS returned the same result)

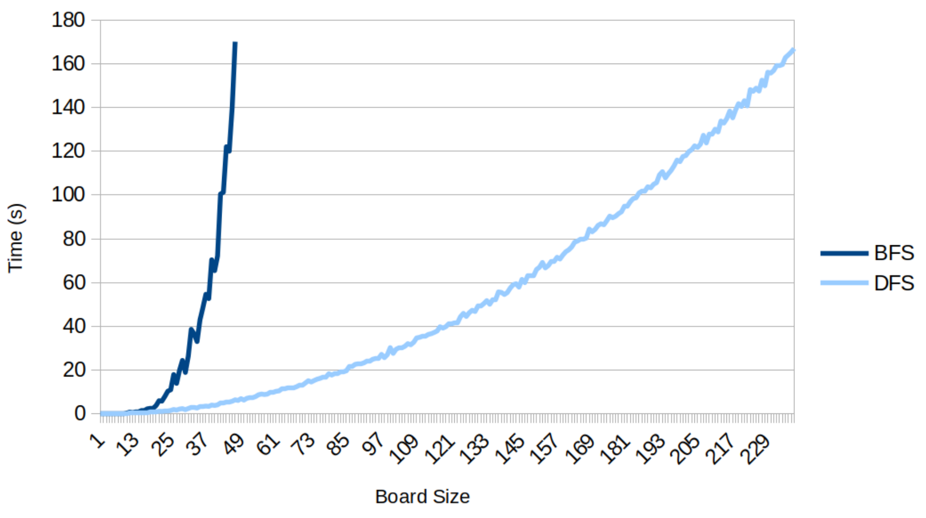

Figure 2: Execution time

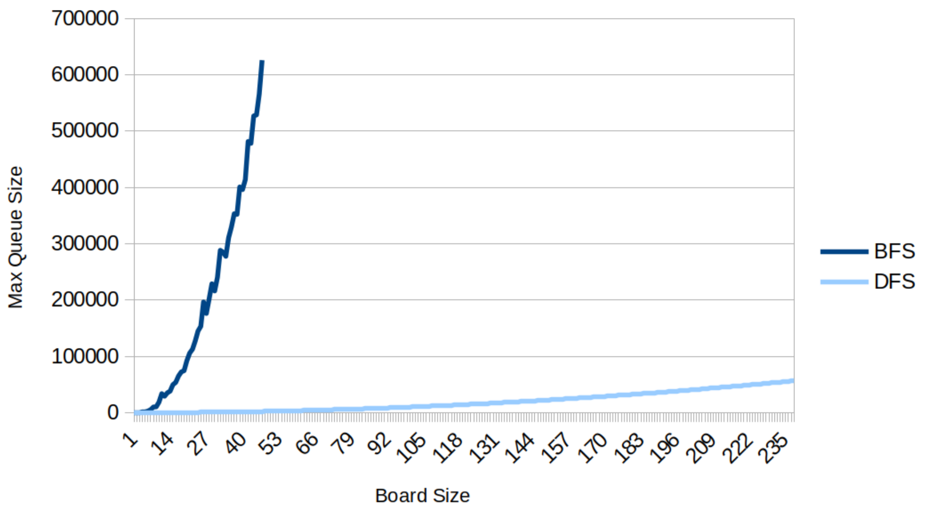

Figure 3: Peak size of the queue during execution