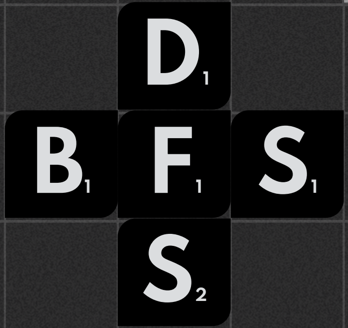
BFSDFS Project



Computer Programming III: Data structures and Algorithms

January 2024

TABLE OF CONTENTS

01 INTRODUCTION

What are BFS and DFS and what is there purpose

02 Breadth First Search

Finding the shortest word transformations and process

03 Depth First Search

Finding the longest word in 10x10 Boggle

04 Results

Summary, optimization, & further research

SEARCH ALGORITHMS

01

INTRODUCTION

What are BFS and DFS and what was this assignment

Search Algorithms

Search algorithms in computer science aim to find a specific item or goal within a dataset. These algorithms operate by systematically exploring and evaluating potential solutions until the desired item is located. Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental search algorithms employed in graph-based structures, with BFS exploring neighbor nodes first and DFS delving deeply into a branch before backtracking.

GRAPHS

Breadth First Search

Breadth-First Search (BFS) is a graph traversal algorithm that starts from a specified source vertex and explores the neighboring vertices before moving on to the next level of neighbors.

Advantages

Well-suited for finding the shortest path when the solution is closer to the starting point.

Visits all vertices at the current depth level before moving on to deeper levels

Disadvantages

Does not perform well in scenarios where the solution is located far from the starting point

It may visit a large number of unnecessary vertices in certain cases

Depth First Search

Depth-First Search (DFS) is a graph traversal algorithm that starts from a specified source vertex and explores as far as possible along each branch before backtracking.

DEPTH

Advantages

Well-suited for scenarios where solutions are distributed across various branches

Can be easily modified to explore specific paths or patterns within the graph

Disadvantages

Vulnerable to getting stuck in deep branches, especially in infinite graphs

Performance may suffer in graphs with high branching factors

2

BREADTH FIRST SEARCH

Shortest Word Transformation

The Assignment: BFS

This program takes two input words and determines the shortest chains of transformations between them, where a transformation involves deleting, adding, or modifying a character. Multiple unique shortest chains may exist, and the program should return all of them, along with the count of pop operations performed on the queue to find these chains. As an additional investigation, the program can explore transformation groups, defined as sets of words that can be pairwise transformed.

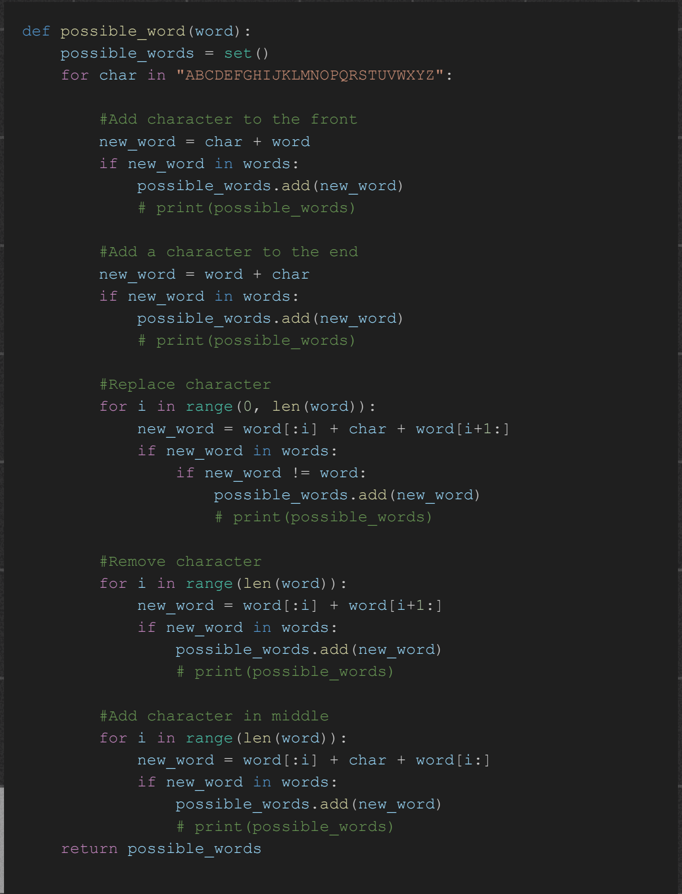
```
FOX -> BOX -> BON -> BOND -> BOUND -> HOUND
FOX -> BOX -> BOD -> BOND -> BOUND -> HOUND
FOX -> FOR -> FORD -> FOND -> FOUND -> HOUND
FOX -> FOO -> FOOD -> FOND -> FOUND -> HOUND
```

Solution Process

- Create functions that are able to transform words
- Create a queue (so we can pop off the end) that will continue to search for solutions so long as we're not at the target word
    - Trim the branch of the queue if its grows longer than an already known solution
- Return all possible shortest chains

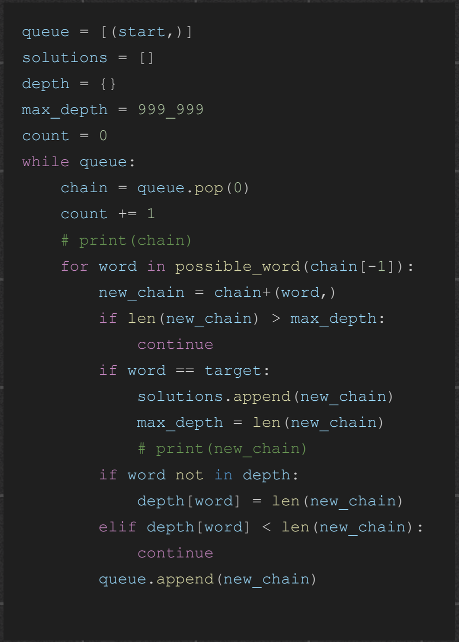
Solution Process

Transformations



```
def possible_word(word):
    possible_words = set()
    for char in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":

        #Add character to the front
        new_word = char + word
        if new_word in words:
            possible_words.add(new_word)
            # print(possible_words)

        #Add a character to the end
        new_word = word + char
        if new_word in words:
            possible_words.add(new_word)
            # print(possible_words)

        #Replace character
        for i in range(0, len(word)):
            new_word = word[:i] + char + word[i+1:]
            if new_word in words:
                if new_word != word:
                    possible_words.add(new_word)
                    # print(possible_words)

        #Remove character
        for i in range(len(word)):
            new_word = word[:i] + word[i+1:]
            if new_word in words:
                possible_words.add(new_word)
                # print(possible_words)

        #Add character in middle
        for i in range(len(word)):
            new_word = word[:i] + char + word[i:]
            if new_word in words:
                possible_words.add(new_word)
                # print(possible_words)
    return possible_words
```

Finding Chain



```
queue = [(start,)]
solutions = []
depth = {}
max_depth = 999_999
count = 0
while queue:
    chain = queue.pop(0)
    count += 1
    # print(chain)
    for word in possible_word(chain[-1]):
        new_chain = chain+(word,)
        if len(new_chain) > max_depth:
            continue
        if word == target:
            solutions.append(new_chain)
            max_depth = len(new_chain)
            # print(new_chain)
        if word not in depth:
            depth[word] = len(new_chain)
        elif depth[word] < len(new_chain):
            continue
        queue.append(new_chain)
```

CODE

3

DEPTH FIRST SEARCH

DEPTH

Boggle Word Finding

The Assignment: DFS

The program should find and return the longest possible word in a [seeded] 10 x 10 Boggle board.

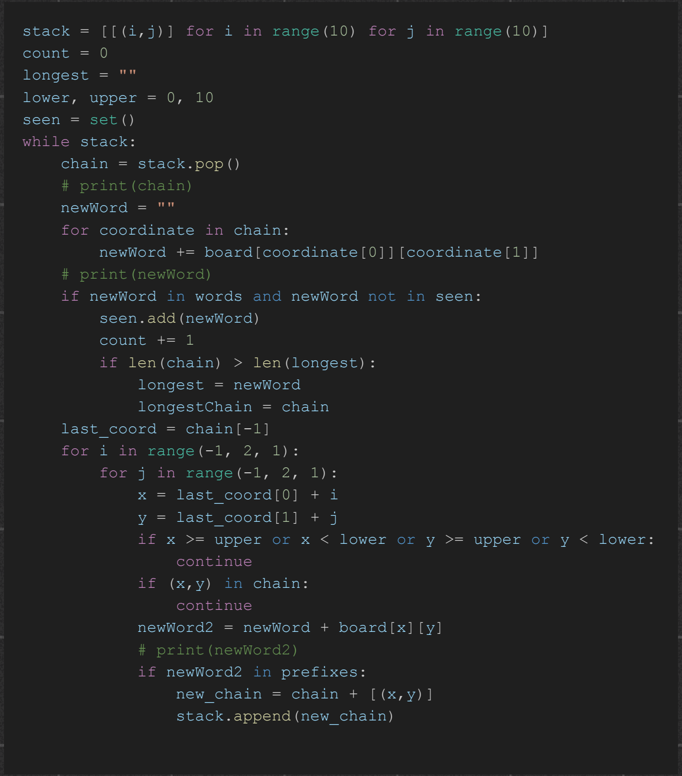
```
egecrrhanh
fnupprsbss
ernoyeipot
sncarsfhrs
sreekciaas
eascscaooe
morpughnna
niocedeenc
ghytitoedw
oboougshss
```

Seed: https://cp3-jgrhx6e4yq-uc.a.run.app/BOGGLE/10

Solution Process

- Create the Boggle board as a 2D Array and a corresponding stack pointing to each coordinate
- Using the stack, pop off the last element and see it's the beginning of a word [prefix]
    - Add adjacent elements until "new chain" is no longer a prefix
- Check for all possible starting locations, keeping track of the longest possible word

Solution Process



```
stack = [[(i,j)] for i in range(10) for j in range(10)]
count = 0
longest = ""
lower, upper = 0, 10
seen = set()
while stack:
    chain = stack.pop()
    # print(chain)
    newWord = ""
    for coordinate in chain:
        newWord += board[coordinate[0]][coordinate[1]]
    # print(newWord)
    if newWord in words and newWord not in seen:
        seen.add(newWord)
        count += 1
        if len(chain) > len(longest):
            longest = newWord
            longestChain = chain
    last_coord = chain[-1]
    for i in range(-1, 2, 1):
        for j in range(-1, 2, 1):
            x = last_coord[0] + i
            y = last_coord[1] + j
            if x >= upper or x < lower or y >= upper or y < lower:
                continue
            if (x,y) in chain:
                continue
            newWord2 = newWord + board[x][y]
            # print(newWord2)
            if newWord2 in prefixes:
                new_chain = chain + [(x,y)]
                stack.append(new_chain)
```

GENERAL RESULTS

4

SEEN

RESULTS

SAVE

NOTE

General Results

Solutions

Breadth First Search:

- Shortest Chain: 4
- Total Pops: 67355
- Time Elapsed: 12.90

Depth First Search:

- Total Words: 1155
- Longest Chain: 7
- Longest Word: DETECTORS
- Time Elapsed: 0.05

Notes

- A greater understanding of stacks/chains/tuples would go a long way
    - I was getting stuck with a lot of syntax and trying to avoid using these topics
- Optimization
    - A better understanding of what is slowing my code and how to fix it would’ve not only saved time, but solved problems sooner rather than later

THANKS!

DO YOU HAVE ANY QUESTIONS?

daniel.s.nkunga@gmail.com
+1 501 237 6337

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik**

Please, keep this slide for attribution

THANKS

REACH OUT