CS350

Stack and Queue for BFS and DFS search

Project Report

Jordan Renaud, Sean Poston, Zach Hall, Teona Velkoska, Jonathan Harsy, Logan Emel
9-26-2021

ABSTRACT

The goal for the project is the traverse the given maze in the form of a BFS search, and a DFS search. More thoroughly, In this project, we will use stack and queue to solve graph search problems in the form of a maze. A maze can be viewed as being undirected or a fully connected graph.

INTRODUCTION

The goal for the project is the traverse the given maze in the form of a BFS search, and a DFS search. More thoroughly, In this project, we will use stack and queue to solve graph search problems. Before we examine our project itself, it is important that we define and describe a few terms. These terms being: stack, que, BFS, and DFS algorithms. We look at the operation and functionality of these terms , as well as the time complexity for each one. Starting off with stack. We can define Stack as a linear data structure, where the elements in the stack can only be inserted and deleted at the top. “The insertion of an element into stack is called push, deletion of an element from the stack is referred to pop operation”(GeeksForGeeks). On the other aspect of stack we have queue. A queue is a linear data structure. Elements can be inserted only from one side of the list (the rear) Additionally, elements can only be deleted from the front. Queue is a data structure that follows first in first out order. This procedure is when an element is the first one to enter the list, it will be the first one to exit the list. Insertion of an element into a queue is called an enqueue, and deletion of an element is called a dequeue (GeeksForGeeks). The two figures below show how we may visualize stacks and queues.

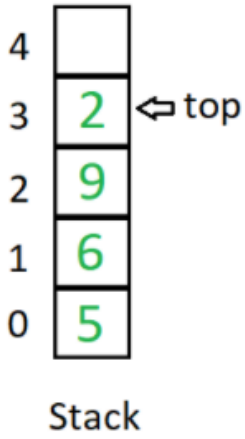
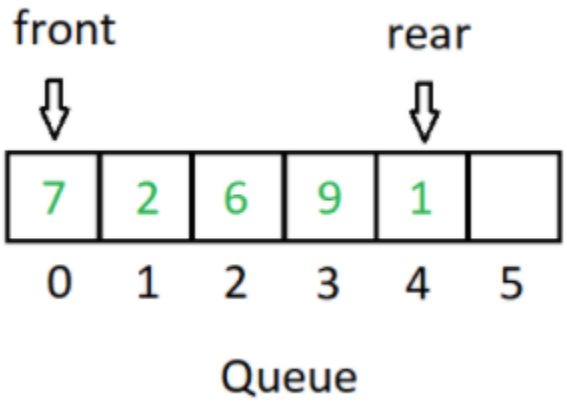


FIGURE 1.1 (Left) Shows Stack data structure.

FIGURE 1.2 (Right) Shows Que data structure.



We can note the time complexity for each data structure as O(n) for the average, and the worst time being O(1), where O(n) is linear, and O(1) is constant.

Time complexity is how long (in the amount of time) it takes an algorithm to run. It measures the amount of time taken to execute a statement (GreatLearningTeam). Time complexity is important because it can help us to make decisions on whether a certain algorithm is practical/efficient enough to apply it to certain situations in computer science.

Now that we have a basic understanding of stack and queues, we need to define and describe BFS and DFS. BFS, Breadth First Search is a certain traversing method that is used a queue for storing the visited vertices. In this method the emphasis is on the vertices of the graph, one vertex is selected at first then it is visited and marked. The adjacent vertices are then visited and stored. The vertices are then treated one by one, their adjacent vertices are visited. For this search we can say a node is fully explored before visiting any other node in the graph. So BFS traverses shallowest unexplored nodes first (GUPTA). A few notable applications for BFS are its ability to be useful in finding whether the graph has connected components or not..

DFS (Depth First Search) is a search method uses a stack for vertices visited. DFS is an edge based method and works with recursion. Node exploration is paused as soon as another unexplored node is found. DFS visit each vertex exactly once and each edge is inspected exactly twice. From the descriptions above, we can arrive at these conclusions BFS and DFS. The major difference between BFS and DFS is that BFS executes level by level while DFS follows paths from the start to end . BFS uses the queue for storing the nodes and DFS uses the stack for traversal of the nodes. The scope of the project was written in the python programming language. Our group used GitHub to organize our program. A video on the data structures discussed in the intro can be found in the video link section.

MAIN CONTEXT

Programs begin input with maze text files as given by Dr. Dai in the project 1 description. Example of the graph maze/matrix below.

```
maze = [[' ', '*', '-', '-', '-', '-', '-', '-', '-', '*', ' '],
        ['>', '|', 'A', ' ', 'B', ' ', 'C', ' ', 'D', '|', ' '],
        [' ', '|', '-', '-', '-', '-', ' ', '-', '-', '|', ' '],
        [' ', '|', 'E', ' ', 'F', ' ', 'G', ' ', 'H', '|', ' '],
        [' ', '|', '-', '-', ' ', '-', '-', '-', '-', '|', ' '],
        [' ', '|', 'I', ' ', 'J', ' ', 'K', ' ', 'L', '|', ' '],
        [' ', '|', '-', '-', '-', '-', ' ', '-', '-', '|', ' '],
        [' ', '|', 'M', ' ', 'N', ' ', 'O', ' ', 'P', '|', '>'],
        [' ', '*', '-', '-', '-', '-', '-', '-', '-', '*', ' ']]
```

Figure 1.3 Matrix for the rooms.

This matrix will be what is printed for output between each step as it is modified with the path taken. The output for each step is saved in additional text files (deep copying methods).

Starting off our group needed to differentiate the two > arrows by knowing for a fact that the entry point is in the first column and the exit is in the last column of the matrix. Traversal of the the matrix until we find the coordinates of the starting point, then skip two items to the right to find the actual starting node for the graph.

The next operation for the program is to access the "maze" looking for rooms and connections. We being at the starting at the coordinates for the first room, in our case: maze[1][2]. We want to check each direction and act accordingly. If it is a wall, ignore it. If it is a space then check the next character in that direction and create the connection for both directions in the adjacency matrix for the graph. Afterwards the search goes to next node. The traversal moves to the right until you find a letter or until you hit a wall. This operation will go to the next row in the graph and search again. The process repeats until there is nothing left to traverse in the maze matrix. With the newly formed graphs, BFS and DFS were implemented and running.

Example pseudocode for the BFS algorithm (as followed from the definition listed in the introduction) :

```
BFS_andQueue (vertex a){
        queue q;
        a then enqueued;
        enqueue(q,a);
        while(!isEmpty(q)){  //check if the que is empty, if not traverse again
                a1 then dequeue(q);
                a1 is visited;
                for(all a2∈(adjacent vertices)){ //for all members in adjacent verticies
```

```
                    if(a2!=enqueued){
                        w then enqueued;
                        enqueue(q,w);
                    }}}}
```

Example pseudocode for the DFS algorithm (as followed from the definition listed in the introduction) :

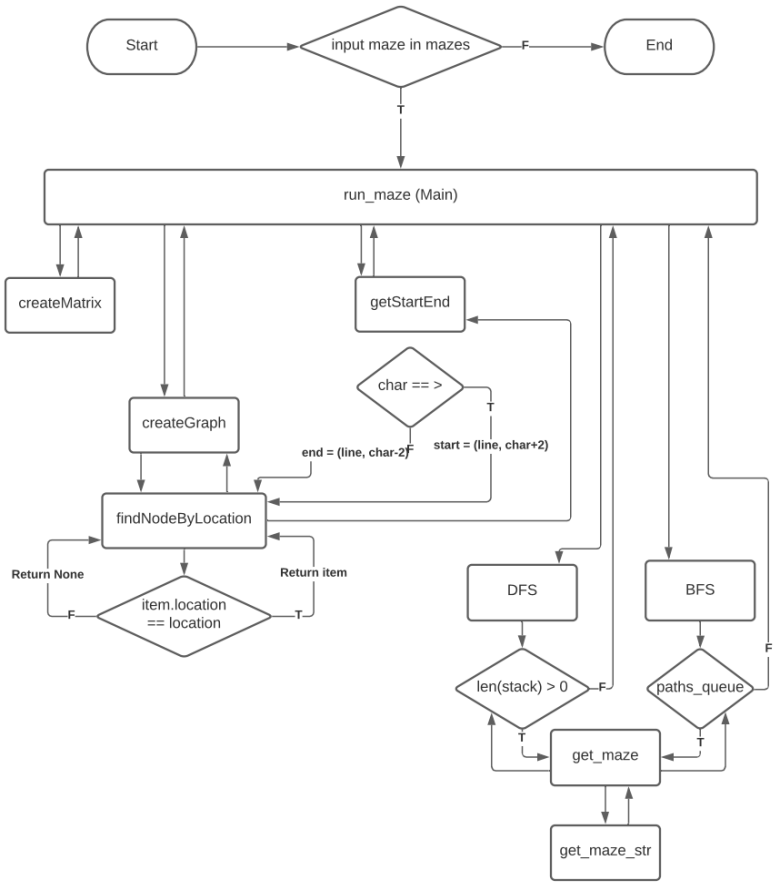
```
DFS_andStack (vertex a) {    //starting vertex
    stack s;                 // stack
    a is pushed;             //starting vertex pushed to stack
    push(s,a);
    while(!isEmpty(s)){    //while the stack is not empty, it will be popped.
        a1 is popped(s);
        a1 is visited;
        for(all a2 in the set of (adjacent vertices of u)){ // adjacent vertices visited
            if(a2!=pushed){
                a2 is pushed;
                push(s,a2);
            }}}}
```

The implementation is below for the reader to follow. Screenshots for the program and output to begin on the next page. Screenshots will be listed in the report as listed in the order of the GitHub repository . Written in python. Programs begin input with maze text files as

given by Dr. Dai in the project 1 description. NOTE: Readers can visit the video demo link listed on the reference page for the verbal explanation of the context.

The flowchart for the outline of our project can be referenced in figure 1.4 below.

Figure 1.4 Flowchart



CONLUSION

In conclusion, we visited and defined the meaning of the stack and queue data structures. We defined and looked at how we can use BFS and DFS search to solve a given problem , and roughly analyze how long and how efficient it is with solving the problem's is

slower than DFS, where we can note the time complexity of the algorithms; O(V+E) V is vertices and E is edges. Our group used Python programming language with a GitHub repository to conduct the project. Through the given report, we hope the readers gained a better understanding to new computer science concepts, or refresh on previously learned principles.

TEAM MEMBERS AND CONTRIBUTION

Jordan Renaud: Wrote search algorithms, function to print a maze with its path, allowed nodes with the same ID to exist and be differentiated from, output file generation, step by step output

Zach Hall: Worked to allow multiple nodes with the same letter, created the function to get the start and end nodes, Created the function to create the graph from the matrices, helped convert search algorithms to work with new object Nodes

Sean Poston: Converted text files to matrices, Implemented get_maze

Jonathan Harsy: data structure and sort references, BFS and DFS pseudo, written report.

Teona Velkoska: Flowchart, Stack, queue, BFS, DFS video.

Logan Emel:

REFERENCES

Great Learning Team, et al. “Time Complexity: What Is Time Complexity & Algorithms of It?” *GreatLearning Blog: Free Resources What Matters to Shape Your Career!*, GreatLearning, 17 Aug. 2021, www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

GeeksForGeeks. “Difference between Stack and Queue Data Structures.” *GeeksforGeeks*, 7 July 2020, www.geeksforgeeks.org/difference-between-stack-and-queue-data-structures/.

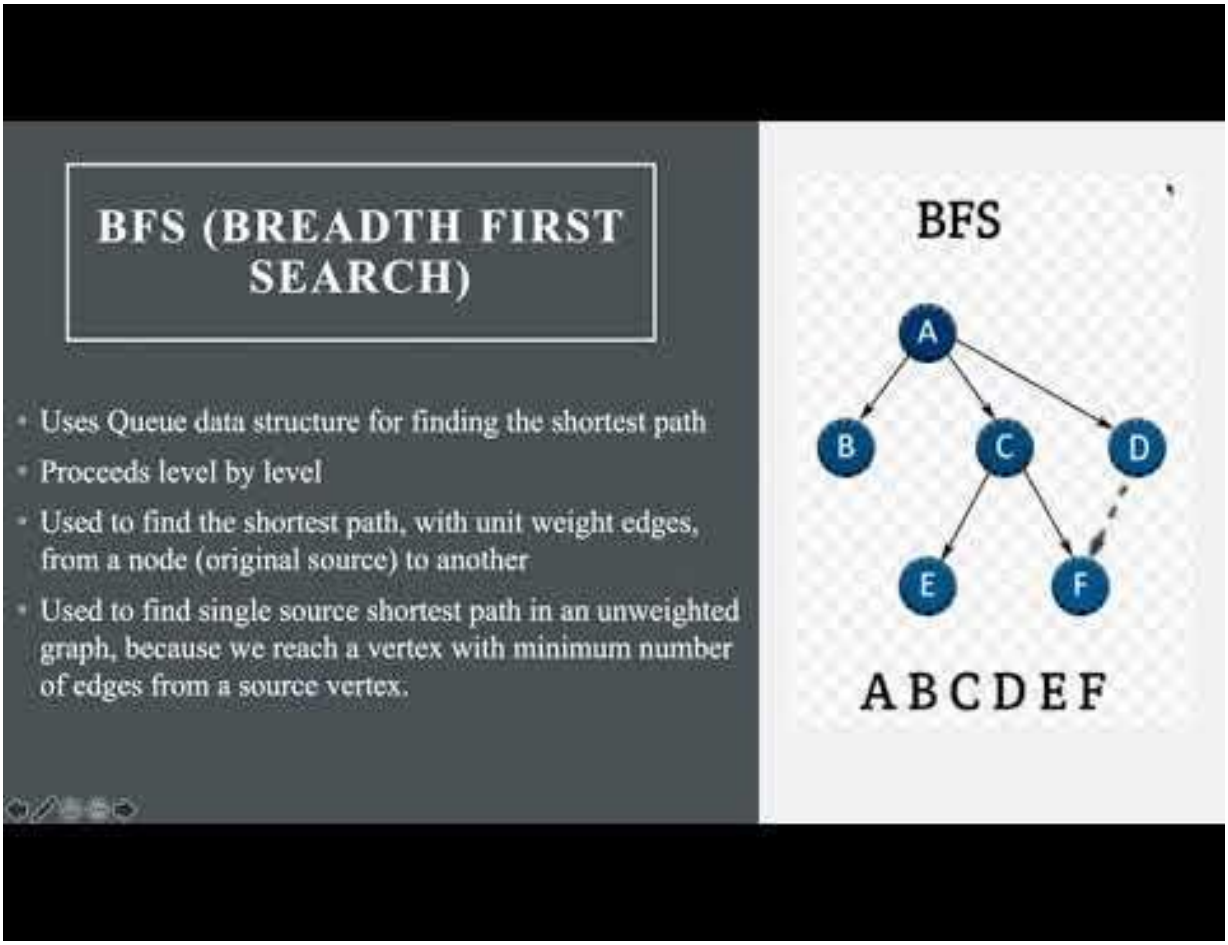
SUMIT KUMAR GUPTA, et al. “Difference between BFS and DFS (with COMPARISON CHART).” *Tech Differences*, Tech Differences, 28 Dec. 2019, techdifferences.com/difference-between-bfs-and-dfs.html.

VIDEO LINKS:

Demo Video: https://www.youtube.com/watch?v=Le1k33VcxHM

Fixes Video: https://youtu.be/sI9wkBaZNaI

Data Structure Video: Project1



SOURCE CODE

Code Link: jtrenaud1s/350Project1 (github.com)

Replit Link: https://replit.com/@JordanRenaud/350Project1#main.py

Written Code in entirety of files (as listed on the GitHub), are posted below for the reader to reference.

Graph.py file.

```
class
Node:
              def __init__(self, letter, location,):
                self.letter = letter
                self.location = location

              def __str__(self):
                return (str(self.letter) +": " +  str(self.location[0]) + "," +
            str(self.location[1]))

              __repr__ = __str__

              def __eq__(self, other):
                return self.location == other.location

              def __hash__(self):
                return hash("foo" + str(self))

            def findNode(graph, Node):
              for item in graph.keys():
                if item == Node:
                  return item
              return None

            def findNodeByLocation(graph, location):
              for item in graph.keys():
                if item.location == location:
                  return item
              return None
```

```
def createGraph(mat):
  graph = {}
  for line in range(len(mat)):
    for char in range(len(mat[line])):
      if (str.isalpha(mat[line][char])):
        letter = mat[line][char]
        node = Node(letter, (line, char))
        search = findNodeByLocation(graph, (line, char))

        if not search:
          graph[node] = []
        else:
          node = search

        directions = {
          "L": (0, -2, 0, -1),
          "R": (0, 2, 0, 1),
          "U": (-2, 0, -1, 0),
          "D": (2, 0, 1, 0),
          # y change, x change, y for dash, x for dash
        }

        for item in directions:
          dir = item
          data = directions[item]

          try:
            if ( str.isalpha(mat[line + data[0]][char + data[1]] ) ):
              if (dir == "L" or dir == "R"):
                node_location = (line + data[0], char + data[1])
                connection_location = (line + data[2], char + data[3])
                letter = mat[node_location[0]][node_location[1]]

                temp = Node(letter, node_location)

                # if child not already in graph keys
                found = findNodeByLocation(graph, node_location)
                if found:
                  graph[node].append( (found, connection_location, '-'))
                else:
                  graph[node].append( (temp, connection_location, '-'))
                  graph[temp] = []
```

```
                elif (dir == "U"):
                  if (mat[line-1][char] != '-'):
                    node_location = (line + data[0], char + data[1])
                    connection_location = (line + data[2], char + data[3])
                    letter = mat[node_location[0]][node_location[1]]
                    temp = Node(letter, (line+data[0], char+data[1]))
                    found = findNodeByLocation(graph, node_location)
                    if found:
                      graph[node].append( (found, connection_location, '|'))
                    else:
                      graph[node].append( (temp, connection_location, '|'))
                      graph[temp] = []
                elif (dir == "D"):
                  if (mat[line+1][char] != '-'):
                    node_location = (line + data[0], char + data[1])
                    connection_location = (line + data[2], char + data[3])
                    letter = mat[node_location[0]][node_location[1]]
                    temp = Node(letter, node_location)
                    found = findNodeByLocation(graph, node_location)
                    if found:
                      graph[node].append( (found, connection_location, '|'))
                    else:
                      graph[node].append( (temp, connection_location, '|'))
                      graph[temp] = []
            except:
              pass


  return graph

def get_connection(graph, first, second):
  candidate = [g for g in graph[first] if g[0] is second]

  if candidate:
    return candidate[0]

  return None
```

Main.py

```
Fromsearch
import
bfs, dfs
            from graph import createGraph
            from maze import createMatrix, get_maze, getStartEnd

            def run_maze(filename):
                matrix = createMatrix('input/' + filename)
                graph = createGraph(matrix)
                start, end = getStartEnd(matrix, graph)

                print("Start: " + str(start))
                print("DFS", filename)
                result_dfs = dfs(graph, start, end, get_maze, matrix, "output/dfs-" +
            filename)
                print("BFS", filename)
                result_bfs = bfs(graph, start, end, get_maze, matrix, "output/bfs-" +
            filename)

            mazes = ["maze_1.txt", "maze_2.txt", "maze_3.txt"]

            [run_maze(m) for m in mazes]
```

Maze.py

```
        from graph import findNodeByLocation


        def createMatrix(file):
            file = open(file, 'r')
            lines = file.readlines()


            mat = []


            for line in lines:
                mat.append([c for c in line])

            return mat
```

```
def getStartEnd(mat, graph):
    for line in range(len(mat)):
        for char in range(len(mat[line])):
            character = mat[line][char]

            if(character == '>'):
                if (char == 0):
                    start = (line, char+2)
                else:
                    end = (line, char-2)

    start = findNodeByLocation(graph, start)
    end = findNodeByLocation(graph, end)


    return start, end



def get_maze_str(maze):
    lines = [''.join(line) for line in maze]
    return ''.join(lines)


def get_maze(path, graph, maze):
    # maze = copy.deepcopy(maze)
    #output = ' -> '.join([p.letter for p in path]) + "\n"
    output = ""
    while len(path) > 1:
        second = path.pop()
        first = path[-1]


        connection = [g for g in graph[first] if g[0] == second][0]


        (y, x) = connection[1]
        character = connection[2]


        maze[y][x] = character

    return output + get_maze_str(maze) + "\n\n"
```

```
def write(filename, out):
  with open(filename, "w") as fp:
    fp.write(out)
```

Search.py

```
import copy
from maze import write


def dfs(graph, start_node, end_node, step, args, filename):


  """ Run a DFS search on the maze and run the step function at the end of each
iteration. Returns the successful path stack
  :type graph: dict()
  :param graph: The dictionary representation of a graph


  :type start_node: str()
  :param start_node: The ID of the node to start the search from


  :type end_node: str()
  :param end_node: The ID of the node to search for


  :type step: function(list())
  :param step: The function that runs at the end of each step, which passes in the
current path stack


  :rtype: list()
  """
  # All the nodes we've already looked at
  seen = []
  # All the nodes that comprise the path we're currently on
  stack = [(start_node, [start_node])]


  #full_path = []


  output = ""
```

```
  # While there are still nodes to look at
  while len(stack):
    (current, path) = stack.pop()
    #full_path.append(current)
    output += step(copy.copy(path), graph, args)
    if current not in seen:

      # if current == end_node:
      #   write(filename, output)
      #   return path

      seen.append(current)
      # Get all available connections and sort them alphabetically by node ID
      available = sorted(graph[current], key=lambda x: x[0].letter, reverse=True)
      # Get a list of all available candidate nodes that haven't been seen yet
      candidates = [candidate[0] for candidate in available if candidate[0] not in
seen]
      for candidate in candidates:
        stack.append((candidate, path + [candidate]))

  write(filename, output)
  return path



def bfs(graph, start_node, end_node, step, args, filename):


  """ Run a DFS search on the maze and run the step function at the end of each
iteration. Returns the successful path stack
  :type graph: dict()
  :param graph: The dictionary representation of a graph


  :type start_node: str()
  :param start_node: The ID of the node to start the search from


  :type end_node: str()
  :param end_node: The ID of the node to search for


  :type step: function(list())
```

```
  :param step: The function that runs at the end of each step, which passes in the
current path stack


  :rtype: list()
  """
  # This queue holds multiple lists. each list is a path that can be taken
  paths_queue = []
  seen = []


  # Push the first path into the queue
  paths_queue.append([start_node])


  # full_path = [start_node]


  output = ""


  # While there are paths to check
  while paths_queue:
    # Dequeue the next available path into a variable
    current_path = paths_queue.pop(0)
    # Get the last node in the current path
    current_node = current_path[-1]
    seen.append(current_node)
    #full_path.append(current_node)
    output += step(copy.deepcopy(current_path), graph, args)


    # path found
    # if current_node == end_node:
    #   write(filename, output)
    #   return current_path

    available = sorted(graph[current_node], key=lambda x: x[0].letter)


    # Get a list of all available candidate nodes
    candidates = [candidate[0] for candidate in available if candidate[0] not in
seen]
```

```
    # Create a new path for each of the candidate nodes and append those paths to
the queue
    for candidate in candidates:
      paths_queue.append(current_path + [candidate])


  write(filename, output)
  return current_path
```