Algorithms
Assignment 2 Report

Warren Roberts
Student ID: 100450116
Student Email: wlr006@uark.edu
University of Arkansas

Purpose

The purpose of this assignment was to implement and analyze three graph search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), and Recursive Depth-First Search (RDFS). The goal was not only to make them function correctly, but to fully understand how traversal order changes depending on whether a queue, stack, or recursion is used.

In addition to implementation, this assignment required performing operation counting on several algorithms to better understand time complexity and worst-case behavior. The final program also visualized the discovered path on the campus map using OpenCV, which confirmed both correctness and practical application of the algorithms.

Approach and Implementation

For this assignment, the Queue, Stack, and Graph classes were already provided. My responsibility was to correctly use those structures to implement the search algorithms while maintaining proper visitation tracking and trace reconstruction.

Breadth-First Search (BFS) For BFS, I used a queue to enforce FIFO behavior. The algorithm begins by resetting the graph, pushing the starting vertex into the queue, and marking it as visited using the graph's built-in visitation functions.

While the queue is not empty, a vertex is dequeued and examined. For each unvisited neighbor, I marked it as visited immediately when discovered, recorded the trace using `g.set_trace(v, u)`, and then pushed it into the queue. Marking vertices as visited at discovery time prevents duplicate insertions and ensures the shortest path property of BFS in an unweighted graph.

This implementation guarantees that vertices are explored level-by-level, and the trace array correctly reconstructs the shortest path from start to destination.


```python
# Breadth-First Search (BFS)
# Uses a queue (FIFO) to explore the graph level by level.
# Ensures the shortest path (in number of edges) is found in an unweighted graph.
# visited prevents revisiting nodes and infinite cycles.
# g.set_trace(v, u) records the parent of each node for path reconstruction.

def bfs(g: Graph, start: int, destination: int) -> None:
    queue: Queue[int] = Queue()
    g.reset()

    # add start to the queue so it gets explored later
    queue.push(start)
    # create and add start to the visited state
    g.set_visited(start)

    # while queue is still available
    while not queue.empty():
        u = queue.pop()
        # dequeue to move onto the next node to explore

        # if this node is the goal lets call it quits
        if u == destination:
            return

        # gets the number of neighbors to node u
        number_of_adjacency_nodes = g.e[u].size()
        p = g.e[u].get_root()
        # set p to be the head of the adjacency list for node u

        # iterate through neighbors of u
        for _ in range(number_of_adjacency_nodes):
            v = p.value

            # now if this neighbor has not been visited add it
            if not g.is_visited(v):
                g.set_visited(v)
                # add tracr from node to node on graph by traversing the edges between v and u
                g.set_trace(v, u)
                # add this to the list of nodes to explore
                queue.push(v)

            p = p.next
```

Figure 1: Breadth-First Search Implementation

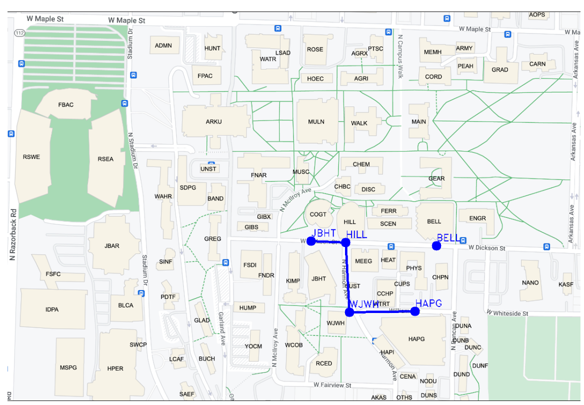

Figure 2: BFS Path Visualization

Depth-First Search (DFS) DFS was implemented using a stack to enforce LIFO behavior. The structure closely mirrors BFS, except that vertices are pushed onto and popped from a stack instead of a queue.

Like BFS, I reset the graph and marked the starting vertex as visited. During traversal, when encountering an unvisited neighbor, I recorded its parent using `g.set_trace` before pushing it onto the stack.

The key difference in behavior comes from the data structure: DFS explores one branch as deeply as possible before backtracking. Unlike BFS, it does not guarantee the shortest path, but it fully explores each branch before moving to the next.

```
# Depth-First Search (DFS)
# Uses a stack (LIFO) to explore as deep as possible before backtracking.
# visited prevents cycles.
# g.set_trace(v, u) records the parent of v for path reconstruction.


def dfs(g: Graph, start: int, destination: int) -> None:
    stack: Stack[int] = Stack()
    g.reset()
    # push the first node and add that to the visited set()
    stack.push(start)
    g.set_visited(start)

    # while stack is valid (not empty)
    while not stack.empty():
        # get the node of the graph being investigated and store its information as u
        u = stack.pop()

        # now if u happens to be where we need to be lets return!
        if u == destination:
            return

        # adjacent nodes is = to graph edge[u]'s size
        number_of_adjacency_nodes = g.e[u].size()
        # p will be the root of this graph and serve as a point of traversal
        p = g.e[u].get_root()
        # traverse through each adjacent node
        for _ in range(number_of_adjacency_nodes):
            v = p.value
            # if this node had not been visited
            if not g.is_visited(v):
                g.set_visited(v)
                # make graph trace between v and u (current node and the last node)
                g.set_trace(v, u)
                # add v to the stack to be explored later
                stack.push(v)

            p = p.next
```

Figure 3: Depth-First Search Implementation

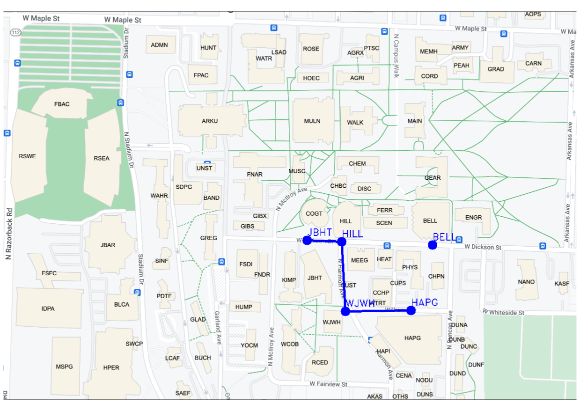

Figure 4: DFS Path Visualization

Recursive Depth-First Search (RDFS) Recursive DFS was implemented using the implicit call stack instead of an explicit stack structure. The helper function checks whether a vertex has already been visited using `g.is_visited`. If not, it marks the vertex as visited and checks whether it is the destination.

For each neighbor, I recorded the trace and then recursively called the helper function. If the destination was found, the function returned `True`, allowing the success condition to propagate upward and stop further exploration early. This early termination was important to avoid unnecessary recursion after the target was found.

Using the graph's internal visited tracking ensured consistency with the assignment's required style.

```
# Recursive Depth-First Search (RDFS)
# Explores the graph by going as deep as possible along each branch before backtracking.
# Uses recursion (implicit call stack) instead of an explicit stack.
# A shared visited set prevents cycles and repeated work.
# g.set_trace(v, u) records parent relationships so the final path can be reconstructed.
# The helper function returns True when the destination is found,
# allowing the success to propagate upward and stop further exploration early.


def rdfs(g: Graph, start: int, destination: int) -> None:
    g.reset()
    # establish visited set and start the recursive call of rdfs
    _rdfs(g, start, destination)
                              (parameter) destination: int

def _rdfs(g: Graph, u: int, destination: int) -> bool:
    # stop if this node has already been visited (prevents cycles)
    if g.is_visited(u):
        return False
    # add to visited set
    g.set_visited(u)

    # if destination is reached, return True to stop further exploration
    if u == destination:
        return True

    # gets the number of neighbors to u
    number_of_adjacency_nodes = g.e[u].size()
    p = g.e[u].get_root()
    # gets the head of u's adjacency list

    for _ in range(number_of_adjacency_nodes):
        v = p.value
        # v is one neighbor of u
        if not g.is_visited(v):
            # record u as the parent of v for path reconstruction
            g.set_trace(v, u)
            # if recursive call finds destination, propagate True upward
            if _rdfs(g, v, destination):
                return True
        p = p.next

    return False
```

Figure 5: Recursive Depth-First Search Implementation

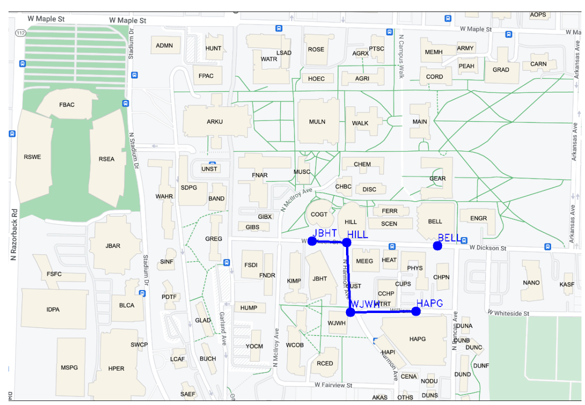

Figure 6: Recursive DFS Path Visualization

Operation Counting

Algorithm 1 The nested loops execute $10 \times 100 = 1000$ times. Inside the loop body, there is one multiplication and two additions. Therefore:

- 1000 multiplication operations $(10 \times 100 = 1000)$
- 2000 addition operations $(10 \times 100 \times 2 = 2000)$

Algorithm 2 In the bubble sort implementation, each swap consists of three assignments. The inner loop executes $n(n-1)$ comparisons in the worst case, and each swap requires 3 assignments, resulting in $3n(n-1)$ assignments in the worst case.

Minimum number of assignments (already sorted): 0

Maximum number of assignments: $3n(n-1)$

Algorithm 3 Binary search reduces the search space by half during each iteration. For an array of size 10, the worst-case number of iterations is $\lceil \log_2(10) \rceil = 4$.

Each iteration will perform (worst-case):

- One while-condition comparison
- One equality comparison
- One less-than comparison

Then, after the 4th iteration, one more call to the while condition (closing call) is made, adding one final comparison before terminating. The worst-case number of comparisons is 13; $(4\ \textbf{iterations} \times 3\ \textbf{comparisons}) + 1\ \textbf{final while check} = 13$. The best-case scenario (target found immediately) requires only 2 comparisons (while loop comparison then the first comparison within while loop).

Challenges and Bug Fixes

One issue I initially encountered was incorrectly handling visited tracking when transitioning from a Python set to the Graph's built-in visitation methods. This caused logical inconsistencies until I ensured that `g.is_visited` and `g.set_visited` were used consistently across BFS, DFS, and RDFS.

Another debugging moment occurred in the recursive DFS implementation. At first, the recursion did not terminate early because I was not properly propagating the return value upward. Once I modified the helper function to return a boolean and propagate `True` when the destination was found, the recursion behaved correctly and avoided unnecessary exploration.

These fixes reinforced how small logical mistakes in visitation tracking or return conditions can significantly affect traversal correctness.

Results

All three search algorithms executed successfully and produced correct paths between the selected start and destination vertices. The terminal output confirmed that BFS, DFS, and RDFS were functioning as expected.

The OpenCV visualization further verified correctness by clearly highlighting the path on the campus map. Seeing the path rendered visually helped confirm that the trace reconstruction logic was implemented correctly.

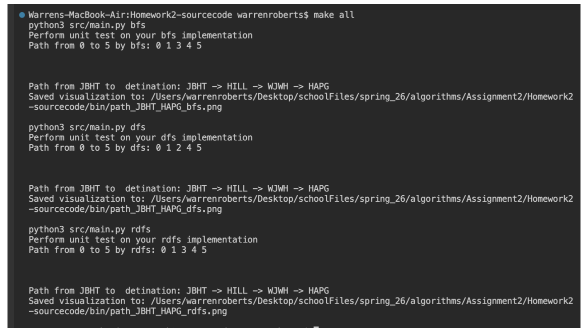

Figure 7: Terminal Output Showing Successful Execution