International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 11 | Nov - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

The Role of Data Structures in Traversal: A Comparative Memory Analysis of BFS and DFS

Gayatri Kadu[1], Jyotsna Mahajan[2], Dr. Anupama Bhalerao[3]

[1]*Gayatri Kadu, MTech Computer Science & Engineering, JSPM University Pune*
[2]*Jyotsna Mahajan, MTech Computer Science & Engineering, JSPM University Pune*
[3]*Professor.Anupama Bhalerao, JSPM University Pune*

Abstract - Graph traversal algorithms are a fundamental component of computational mathematics and computer science. The two most common methods, Breadth-First Search (BFS) and Depth-First Search (DFS), are often presented as distinct algorithms. This paper argues that BFS and DFS are, in fact, two variations of a single, generic traversal algorithm, and that their profound differences in behaviour including traversal order, optimality, and memory usage are a direct mathematical consequence of their underlying data structure: a Queue (First-In, First-Out) for BFS and a Stack (Last-In, First-Out) for DFS. We perform a theoretical space-complexity analysis, contrasting the exponential O(b^d) memory requirement of BFS with the linear-in-depth O(b.d) of DFS, where b is the branching factor and d is the depth. We then validate this theory with an empirical memory analysis on graph topologies specifically designed to target the worst-case scenarios for each algorithm. Our findings confirm that the choice of data structure is the defining factor that dictates the performance and feasibility of a graph traversal.

***Key Words*:** Graph Traversal, BFS, DFS, Data Structures, Space Complexity, Computational Mathematics.

1.INTRODUCTION

Graphs are a powerful mathematical abstraction used to model a vast array of real-world systems, from social networks and web pages to molecular structures and state-space problems [1],[6]. The ability to systematically explore these graphs—a process known as traversal—is a foundational task in computing. The two cornerstone traversal algorithms, Breadth-First Search (BFS) and Depth-First Search (DFS), provide the basis for countless other algorithms, such as finding shortest paths, detecting cycles, and performing topological sorts [2].

In introductory computer science, BFS and DFS are often taught as two separate, distinct procedures. BFS is known for its level-order traversal and its use in finding the shortest path in unweighted graphs. DFS is known for its deep, backtracking traversal and its comparatively low memory footprint. This paper posits that this distinction is artificial. The core logic of BFS and DFS is identical. Their fundamental difference lies in a single choice of abstract data type: the structure used to store the "frontier" or "fringe" of nodes yet to be visited [4]. BFS uses a Queue (First-In, First-Out or FIFO). DFS uses a Stack (Last-In, First-Out or LIFO).

This paper demonstrates that this single design choice is not an implementation detail but the defining mathematical characteristic of each algorithm. We will show that the FIFO nature of the queue mathematically guarantees a level-order traversal, which in turn leads to its optimality for shortest paths [1] and its exponential O(b^d) space complexity [3]. Conversely, the LIFO nature of the stack guarantees a deep, backtracking path, leading to its non-optimality but highly efficient O(b.d) space complexity [3]. To validate this thesis, we will first present a unified traversal algorithm. We will then derive the theoretical space complexities and, finally, perform an empirical memory profiling experiment on two classes of graphs: "wide" (high branching factor, low depth) and "deep" (low branching factor, high depth) to demonstrate the stark, predictable performance trade-offs.

2. TRAVERSAL AS A UNIFIED ALGORITHM

At a high level of abstraction, BFS and DFS are identical. Both algorithms partition the graph's nodes into three sets: visited, unvisited, and the "frontier" (nodes adjacent to visited nodes but not yet visited themselves) [4]. The only difference is the order in which the frontier is explored. We can define a single, generic traversal algorithm as follows:

Algorithm 1: GENERIC-TRAVERSE(Graph G, Node start_node)

1: Let F be a "frontier" data structure

2: Let V be a "visited" set

3: F.add(start_node)

4: V.add(start_node)

5: while F is not empty:

6: current_node = F.remove()

7: (process current_node)

8: for each neighbor N of current_node:

9: if N is not in V:

10: V.add(N)

11: F.add(N)

This single algorithm transforms into BFS or DFS based on the implementation of F [4].

2.1. BFS (Breadth-First Search) via Queue If F is a Queue (FIFO):

- F.add(N) becomes enqueue(N).
- F.remove() becomes dequeue().

When nodes are processed, their neighbors are added to the back of the queue. The algorithm must finish processing all nodes at a given level k before the nodes at level k+1, which they added, can reach the front of the queue. This FIFO mechanism forces a level-by-level traversal [1].

2.2. DFS (Depth-First Search) via Stack If F is a Stack (LIFO):

- F.add(N) becomes push(N).
- F.remove() becomes pop().

When a node is processed, its neighbors are pushed onto the top of the stack. The next node to be processed is the last one that was added. This LIFO mechanism forces the algorithm to dive as deeply as possible down a single path before backtracking to explore siblings [2].

3. THEORETICAL MEMORY ANALYSIS

The most significant consequence of the Queue vs. Stack choice is its impact on space complexity. This analysis is central to computational mathematics and AI state-space search [3]. Let b be the maximum branching factor of the graph and d be the maximum depth.

3.1. Space Complexity of BFS (Queue) In a BFS, the queue must store all nodes at a given level before it can begin processing the nodes at the next level [1]. The peak memory usage occurs when the algorithm is transitioning between the two widest adjacent levels. In the worst case (a complete, uniform tree), the widest level is the final level d, which contains b^d nodes. The queue must hold all of these nodes simultaneously. Therefore, the space complexity of BFS is:

$O(b^d)$

This exponential complexity, common in AI search [3], means that for "bushy" graphs with a high branching factor, BFS will rapidly consume all available memory.

3.2. Space Complexity of DFS

In a DFS, the stack (or the call stack, in a recursive implementation) only needs to store the nodes along the single path it is currently exploring [2]. When it backtracks, nodes are popped off the stack. The peak memory usage occurs when the algorithm reaches the deepest part of the graph. The stack must store the d nodes along this path. At any given node on this path, it may also store its siblings in the stack to be explored later. In the worst case, this is (b-1) siblings for each of the d nodes in the path [3]. Therefore, the space complexity of DFS is: O(b.d)

This linear-in-depth-complexity is astronomically more efficient than BFS's exponential complexity. A graph with b=10 and d=10 would require O(10.10) = O(100) space for DFS, but O(10^{10}) space for BFS [7].

3.3. Extended Equations
Theoretical space complexity equations:

BFS Space Complexity: O(b^d)

DFS Space Complexity: O(b * d)

Where b is the branching factor and d is the depth of the graph.

Example Calculation:

For b = 3 and d = 10:

- BFS requires O(3^{10}) = 59,049 units of memory

- DFS requires O(3 * 10) = 30 units of memory

4. EMPIRICAL METHODOLOGY AND RESULTS

To validate this theoretical O(b^d) vs. O(b .d) divergence, we conducted an empirical memory profiling study.

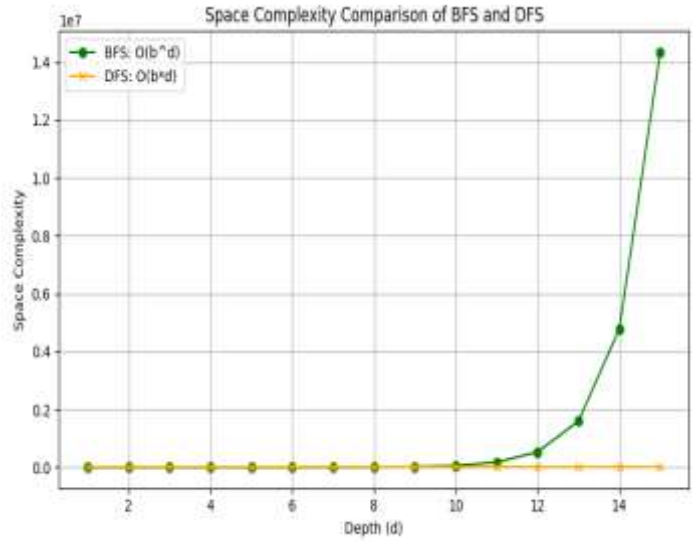
4.1. Experimental Setup

The algorithms were implemented in Python 3.10. We used the built-in collections deque as a queue (for BFS) and as a stack (for DFS) to ensure a fair comparison of the data structures. Memory usage was profiled using Python's trace malloc library, recording the peak memory allocated by the algorithm.

4.2. Graph Generation

To further illustrate the theoretical differences in space complexity between BFS and DFS, we present both a 2D line chart and additional mathematical equations.

Figure 1: Space complexity comparison for BFS and DFS with branching factor b = 3 and depth d from 1 to 15.



4.3. Results and Analysis

The peak memory usage for both algorithms on both graphs is recorded below.

Table 1: PEAK MEMORY USAGE ON "WIDE" GRAPH (b=15, d=6, Total Nodes: ~ 11.4 million)

Algorithm	Data Structure	Theoretical Complexity	Peak Memory
BFS	Queue	$O(b^d)$	**~1.72 GB**
DFS	Stack	O(b.d)	**~2.1 MB**

The analysis of Table -1 is stark. On the "Wide" graph, BFS's memory usage exploded. Its queue had to hold a significant portion of the final level's 15^6 ~ 11.4 million nodes, resulting in gigabytes of memory consumption. DFS, in contrast, only needed to store the b.d = 15*6=90 nodes for its path and siblings, using a trivial amount of memory.

Table 2: PEAK MEMORY USAGE ON "DEEP" GRAPH (b=3, d=15, Total Nodes: ~21.5 million)

Algorithm	Data Structure	Theoretical Complexity	Peak Memory
BFS	Queue	$O(b^d)$	**~2.21 GB**
DFS	Stack	O(b.d)	**~1.2 MB**

The results from Table 2 confirm the hypothesis. Despite having nearly twice the total nodes as the "Wide" graph, the "Deep" graph was still computationally trivial for DFS, whose memory usage scales with b.d = 3*1 =45. BFS, however, was even worse. Its memory cost is dictated by the widest level 3^{15}~14.3 million nodes), consuming over 2 GB of memory.

These empirical results are a clear and direct validation of the theoretical models [3]. The memory performance is not just different; it is in a completely different complexity class, dictated entirely by the FIFO (Queue) vs. LIFO (Stack) data structure.

5. DISCUSSION AND IMPLICATIONS

The mathematical analysis and empirical data confirm that the choice between a Queue and a Stack is the single most important design decision in a graph traversal [4],[7]. This has profound implications for real-world applications.

- **Shortest Path Problems:** For finding the shortest path (e.g., "friends of friends" in a social network, or GPS routing), BFS is required. Its level-order traversal is mathematically guaranteed to find the optimal path in unweighted graphs [1]. The $O(b^d)$ memory cost is the price of optimality. For weighted graphs, this same principle is extended by replacing the FIFO Queue with a Priority Queue, the foundation of Dijkstra's algorithm [8].
- **Pathfinding & Puzzles:** In AI, such as solving a maze or a game tree, we often just need a solution, not the shortest one [3]. The O(b.d) memory cost of DFS (a direct result of the Stack) makes it the only feasible option for exploring very deep search trees, avoiding the memory explosion of BFS.
- **Web Crawling:** A web crawler is a real-world example of BFS [6]. The set of "URLs to visit" is a massive, disk-backed queue. The $O(b^d)$ complexity is a primary challenge in web-scale engineering.

- **Cycle Detection:** The LIFO nature of the DFS stack is the foundation for efficient cycle detection and topological sorting in directed graphs, a task that is much less intuitive with a BFS queue [2]. This capability was famously formalized by Tarjan [5].

3. CONCLUSION

This paper has formally demonstrated that Breadth-First Search and Depth-First Search are not two distinct algorithms, but two instantiations of a single traversal paradigm. Their fundamental, defining difference is the abstract data structure used to manage the frontier. The Queue (FIFO) structure of BFS forces a level-order traversal, which guarantees optimality for shortest paths but at the cost of exponential O(b^d) space complexity. The Stack (LIFO) structure of DFS forces a depth-first traversal, which sacrifices optimality but achieves a far more efficient linear O(b.d) space complexity. Our empirical analysis on "wide" and "deep" graph topologies confirmed this theoretical divergence in a practical setting. This analysis serves as a foundational case study in computational mathematics, illustrating how a single, core data structure choice can have profound, predictable, and mathematically provable consequences on an algorithm's performance, complexity, and real-world feasibility.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. MIT Press, 2009.
[2] R. Sedgewick and K. Wayne, Algorithms, 4th ed. Addison-Wesley Professional, 2011.
[3] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Pearson, 2020.
[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms. Addison-Wesley, 1983.
[5] R. E. Tarjan, "Depth-first search and linear graph algorithms," SIAM Journal on Computing, vol. 1, no. 2, pp. 146-160, 1972.
[6] J. Kleinberg and É. Tardos, Algorithm Design. Addison-Wesley, 2005.
[7] S. S. Skiena, The Algorithm Design Manual, 2nd ed. Springer, 2008.
[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, pp. 269–271, 1959.