IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, May 2026
Impact Factor: 8.2

A Hybrid Approach to Recursive and Iterative Problem Solving in Graph Traversal: A Quantitative Performance Analysis

Palve P. B. Kiran Sonawane, Vivek Bhosale, Mayuresh Waghmode, Sushant Londhe
Adsul Technical Campus, Chas

Abstract: *Graph traversal algorithms—Depth-First Search (DFS) and Breadth-First Search (BFS)—are fundamental to problem solving in computer science. The choice between recursive and iterative implementations profoundly influences runtime efficiency, memory consumption, and stack safety, especially for large or deep graphs. This paper presents a controlled empirical study comparing recursive and iterative variants of DFS and BFS on directed acyclic graphs (DAGs) of varying sizes (10^2 to 10^6 nodes). We measure execution time, peak memory usage, and maximum recursion depth encountered. Results show that iterative implementations outperform recursive ones by an average factor of 2.3× in time and reduce memory overhead by up to 47% for graphs deeper than 5,000 nodes. Based on these findings, we propose a hybrid strategy that selects the appropriate paradigm at runtime using a lightweight, conservative depth estimator. The hybrid method achieves near-optimal performance across all tested graph sizes, demonstrating that algorithmic problem solving benefits from adaptive paradigm selection.*

Keywords: problem solving, recursion, iteration, graph traversal, performance analysis, hybrid algorithms.

I. INTRODUCTION

Problem solving in programming often requires choosing between recursive and iterative implementations of the same logical algorithm. Recursion enables concise, mathematically elegant code that mirrors the problem's decomposition, while iteration provides explicit control over state and avoids function call overhead. However, no universal guideline exists for when to favour one over the other. This gap becomes critical in domains such as compiler design, network routing, and artificial intelligence, where graph traversal forms the backbone of larger systems.

The primary contributions of this work are:

1. A rigorous, reproducible benchmarking framework for recursive vs. iterative DFS and BFS.
2. Quantitative thresholds at which recursion becomes practically infeasible due to stack limits.
3. A hybrid problem-solving strategy that dynamically selects the optimal implementation based on estimated graph depth, with conservative fallback.
4. Open-source reference implementations to support replication (provided as supplementary material).

The remainder of this paper is organised as follows. Section 2 reviews related work on recursion limits and algorithm performance. Section 3 describes the experimental methodology, including graph generation and measurement protocols. Section 4 presents the results, and Section 5 discusses their implications for engineering problem solving. Section 6 concludes with recommendations and future directions.

II. RELATED WORK

The trade-offs between recursion and iteration have been studied since the early days of structured programming. Abelson and Sussman (1996) contrasted recursive processes with iterative processes, noting that tail-recursive

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 11, May 2026 Impact Factor: 8.2

implementations can be as efficient as loops in languages that guarantee tail-call optimisation [1]. However, most mainstream languages (Python, Java, C#) do not provide this guarantee [2].

For graph algorithms, Sedgewick and Wayne (2011) note that recursive DFS is natural but risks stack overflow on large graphs [3]. Several researchers have proposed explicit stack simulation as a safer alternative. Jones (2015) measured Python's recursion limit (default 1000) and showed that iterative rewrites are mandatory for deep traversals [4].

Iterative BFS using a queue is well understood, but recursive BFS is rarely used because it fundamentally requires an explicit queue or leads to recursion depth equal to the graph's diameter. To date, no study has provided a unified performance model across a wide range of graph depths and densities. This paper fills that gap by offering actionable thresholds and a hybrid solution.

III. METHODOLOGY

3.1 Algorithms Implemented

We implemented four variants in Python 3.11 (chosen for its widespread use and default recursion limit of 1000; we increased the limit to 5000 for this experiment):

R DFS: Recursive DFS using the system call stack.

I DFS: Iterative DFS using an explicit list as a stack.

R BFS: Recursive BFS (implemented via a recursive helper that processes levels – non-standard but included for completeness).

I BFS: Classic iterative BFS using collections.deque.

All algorithms operate on directed acyclic graphs (DAGs) generated as random Erdős–Rényi graphs with edge probability p = 0.1. For each graph size $n \in \{10^2, 5\times10^2, 10^3, 5\times10^3, 10^4, 5\times10^4, 10^5, 5\times10^5, 10^6\}$, we generated 30 distinct graphs. Traversal always started from the node with the highest out-degree to maximise depth.

3.2 Measurement Setup

Experiments ran on a dedicated machine (Intel Xeon E5-2680 v4 @ 2.40 GHz, 64 GB RAM, Ubuntu 22.04). For each run we recorded:

Wall clock execution time (excluding graph generation).

Peak resident memory (via tracemalloc).

Maximum recursion depth for recursive variants (recorded just before a RecursionError would occur, using a custom depth counter).

We enforced a time limit of 300 seconds and a recursion limit of 5000. Any run exceeding these limits was flagged as "failure".

3.3 Hybrid Strategy Design

Based on preliminary experiments, we designed a hybrid selector for DFS that:

1. Performs a bounded BFS from the start node, tracking the current depth (number of edges from the start). The BFS stops if the depth exceeds 5000 or after visiting 10,000 nodes.
2. After the BFS, if all reachable nodes were visited (i.e., the BFS queue became empty) and the maximum depth found is < 5000, the algorithm uses recursive DFS. Otherwise, it conservatively falls back to iterative DFS.

This conservative rule guarantees that recursion is only used when the full depth is known to be safe. For BFS, we always use the iterative implementation, because recursive BFS offers no advantage and fails on deep graphs.

The estimator adds less than 0.2% overhead for graphs larger than 10^4 nodes. The Python implementation is provided in the Appendix.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, May 2026
Impact Factor: 8.2

IV. RESULTS

4.1 Execution Time

Table I reports median execution times (in milliseconds) for successful runs. Iterative variants consistently outperform recursive ones for graphs with more than 5×10^3 nodes. For shallow graphs ($n \leqslant 1000$), recursion is competitive (within 15% of iterative).

TABLE I: MEDIAN EXECUTION TIME (MS) FOR DFS AND BFS VARIANTS

Graph size (nodes)	R DFS (ms)	I DFS (ms)	R BFS (ms)	I BFS (ms)
100	1.2	1.1	2.4	1.3
500	5.7	5.2	14.1	6.0
1,000	12.4	11.8	32.6	12.9
5,000	89.3	73.1	failure	74.5
10,000	412.0	157.2	failure	161.8
50,000	failure	892.4	failure	910.1
100,000	failure	1842.0	failure	1901.3

failure indicates recursion depth exceeded 5000 or time >300 seconds.

4.2 Memory Consumption

At $n = 10^4$ nodes, recursive DFS used 178 MB of peak resident memory, whereas iterative DFS used only 94 MB (a reduction of 47%). Iterative BFS used slightly more memory than I-DFS because the queue maintained up to 62% of the nodes on average. For graphs with depth > 5000, recursive variants caused stack memory exhaustion, leading to process termination.

4.3 Hybrid Strategy Performance

The hybrid selector never failed on any graph up to 10^6 nodes. Its median execution time was within 5% of the better pure implementation for each graph size. For example, at $n = 10^4$ the hybrid method took 165 ms, compared to 157 ms (I-DFS) and 412 ms (R-DFS). Overhead from depth estimation was below 0.5 ms for all tested graphs.

V. DISCUSSION

Three key insights emerge for engineering problem solving.

First, recursion is not universally safer or more elegant. For deep graphs (>5000 edges from the start), recursion introduces real risk of stack overflow. Many production systems (e.g., web crawlers, dependency resolvers) operate on graphs that exceed this threshold without warning. The hybrid strategy mitigates this risk while preserving recursion's clarity for shallow structures.

Second, performance penalties of recursion stem from function call overhead and stack memory management. Our measurements show that for medium-sized graphs (5,000–50,000 nodes), recursion is 2–3× slower than iteration. This gap widens with depth because each recursive call allocates a new stack frame, triggering more cache misses.

Third, BFS should never be implemented recursively. The recursive BFS variant we tested required an auxiliary parameter to track levels, leading to a call depth equal to the graph's diameter. On dense DAGs, this becomes catastrophic. Practitioners should enforce iterative BFS as a coding standard.

The proposed hybrid strategy addresses the core problem: how to choose an implementation paradigm without a priori knowledge of input characteristics. Lightweight depth estimation with conservative fallback is a practical, low-overhead solution that can be embedded in library functions. Future work could extend this to other recursive algorithms (quicksort, tree traversals) and to languages with tail-call optimisation.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 11, May 2026
Impact Factor: 8.2

VI. CONCLUSION AND RECOMMENDATIONS

This paper quantitatively demonstrated that iterative graph traversal outperforms recursive traversal for graphs deeper than 5,000 nodes, both in time and memory. Recursive DFS remains viable for shallow graphs ($\leqslant$1,000 nodes). Based on these results, we recommend the following engineering guidelines:

1. Never use recursion for BFS – always implement BFS iteratively with a queue.
2. For DFS, use recursion only when the maximum depth is known a priori to be safely below the language's recursion limit (e.g., <1000 for default Pyth