Experimental Performance Benchmarking of Popular Search Algorithms in Java and Python

Redha Zidan
Department of Computer Science & Engineering
Shadan College of Engineering & Technology, JNTUH
Hyderabad, India

Khaja Nabeel Ur Rehman
Department of IT
Shadan College of Engineering & Technology, JNTUH
Hyderabad, India

Dr. Imtiyaz Khan
Associate Professor & Head, Department of IT
Shadan College of Engineering & Technology, JNTUH, Hyderabad, India

ABSTRACT
Search algorithms form the backbone of computer science applications ranging from information retrieval and artificial intelligence to database management and network optimization. Although their theoretical complexities are well studied, practical performance can vary significantly depending on the choice of programming language and runtime environment. This study presents a comparative performance analysis of widely used search algorithms Linear Search, Binary Search, Depth-First Search (DFS), Breadth-First Search (BFS), and A Search* are implemented in two popular programming languages: Java and Python.

The analysis focuses on measuring execution time across varying dataset sizes and graph structures to highlight differences in efficiency, scalability, and runtime behavior. Empirical results demonstrate that while Java generally outperforms Python in computation-intensive tasks due to its compiled nature and Just-In-Time (JIT) optimizations, exceptions arise in certain cases.

The findings of this research emphasize that performance cannot be judged solely on algorithmic theory; instead, language characteristics, data structures, memory models, and runtime environments play crucial roles in determining practical efficiency. The study concludes with insights into the suitability of Java versus Python for algorithm-intensive applications, offering guidance for researchers, educators, and software developers in selecting the right combination of algorithm and language for performance-critical systems.

General Terms
Algorithms, Performance Evaluation, Computational Complexity, Experimental Analysis, Data Structures, Programming Languages.

Keywords
Search Algorithms, Linear Search, Binary Search, Depth-First Search (DFS), Breadth-First Search (BFS), Java, Python, Time Complexity, Space Complexity, Algorithm Optimization. Comparative Study, Runtime Analysis.

1. INTRODUCTION
In today's fast-growing technological landscape, data plays a central role in almost every system. With the rise of artificial intelligence, machine learning, and data science, it has become more important than ever to handle data efficiently in a retrievable format to optimize applications and reduce computational expenses. The volume of data continues to grow with the expansion of digital knowledge bases, user-driven content, and real-time analytics. Systems where data forms the backbone of operations are referred to as data-intensive systems.

In such systems, information retrieval is critical to ensuring responsiveness and efficiency. The effectiveness of a system often depends on the ability to quickly locate, traverse, and process data. To achieve this, a variety of search algorithms are employed. Each algorithm comes with its own set of advantages and limitations, making it impractical to assume a "one-size-fits-all" solution for all scenarios. For example, approaches range from simple methods like Linear Search, which sequentially checks each element, to highly sophisticated heuristic-based methods such as A*, which aim to find optimal paths but come with greater computational costs [1].

While theoretical complexity analysis using Big-O notation provides a foundation for understanding these algorithms, real-world performance often deviates from theory. Factors such as dataset size, memory management, cache efficiency, and the underlying programming language can significantly impact runtime [1]. As a result, conducting empirical performance studies is essential to complement theoretical insights and guide practical implementations.

2. SEARCH ALGORITHMS
This study focuses on five widely used search algorithms:

2.1 LINEAR SEARCH
Linear search, also called sequential search, is one of the most fundamental searching techniques in computer science. The algorithm works by traversing a list or array element by element, starting from the first index and moving sequentially until the desired target value is either found or the list ends. At each step, the current element is compared with the target value: if a match is found, the algorithm immediately returns the index (or position) of the element; if no match is found after checking all elements, the algorithm concludes that the target does not exist in the collection [1]. This process makes linear search intuitive and easy to implement, since it requires no complex logic or additional data structures. Its time complexity is **O(n)** in both the worst and average cases, because in the worst case it may need to examine every element. However, in the best case, when the target is found at the first position, the time complexity is O(1). The space complexity is **O(1)**, since the search only requires a constant amount of extra memory regardless of input size.

Advantages: It can be applied to both sorted and unsorted datasets without modification, making it highly versatile. It is also useful for small datasets where performance differences are negligible, or in scenarios where the dataset is so small that implementing a more complex algorithm would not provide practical benefits. Additionally, linear search works well with data structures that do not allow random access, such as linked lists, where binary search would be inefficient or impossible [1].

Limitations: linear search becomes highly inefficient as the dataset grows. Since it does not leverage any structure or ordering within the data, each comparison is performed blindly, leading to a linear increase in time as input size increases. For large datasets, this can be computationally expensive and slow compared to other searching algorithms like binary search (O(log n)) or hash-based search (O(1) on average). Another limitation is that linear search does not exploit sorted data in any way whether the list is ordered or not, the algorithm performs the same [1].

2.2 BINARY SEARCH

Binary search is a highly efficient searching algorithm that works on the principle of divide and conquer, but it can only be applied to sorted datasets. The algorithm begins by comparing the target value with the middle element of the sorted list or array. If the target matches the middle element, the search ends successfully. If the target is smaller than the middle element, the search continues recursively or iteratively in the left half of the dataset; if the target is larger, the search continues in the right half. This process of halving the search interval continues until the target is found or the subarray size reduces to zero, meaning the element is not present in the collection [1].

The time complexity of binary search is **O(log n)** in the worst and average cases, because the dataset is reduced to half with each iteration, drastically minimizing the number of comparisons compared to linear search. In the best case, when the target is found at the middle element in the first step, the time complexity is O(1). The space complexity is **O(1)** for iterative implementations, since only a few pointers or indices are needed, though recursive implementations may use O(log n) space due to function call stack usage [1].

Advantages: It is exponentially faster than linear search for large datasets, as the logarithmic growth ensures that even in very large collections, only a small number of comparisons are needed. For instance, in a dataset of one million elements, binary search would take at most about 20 comparisons, whereas linear search could take up to one million. It is also conceptually simple once sorting is ensured, and it is widely used in real-world systems such as dictionary lookups, searching in databases, or checking membership in sorted data structures [1].

Limitations: The primary drawback is that the dataset must be sorted before applying the algorithm, and maintaining sorted data may require extra preprocessing time and cost. For unsorted data, sorting first would require O(n log n) time, which can outweigh the benefits if only a single search is performed. Additionally, binary search works best with random-access data structures like arrays, but is inefficient or impractical with linked lists, where accessing the middle element requires traversing multiple nodes. Another limitation is implementation complexity compared to linear search, as handling edge cases like overflow in index calculations can be error-prone.

2.3 DEPTH FIRST SEARCH

Depth First Search (DFS) is a fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking, making it particularly useful for searching or traversing tree and graph structures. Starting from a source node, DFS moves to an adjacent unvisited node, then continues exploring deeper into the graph until it reaches a node with no unvisited neighbors. At this point, it backtracks to the most recent node with unexplored edges and continues the process until all reachable nodes have been visited. DFS can be implemented using recursion (via the call stack) or an explicit stack data structure. It is commonly used in applications such as pathfinding, topological sorting, cycle detection in graphs, solving puzzles (like mazes), and analyzing connectivity in networks. The time complexity of DFS is **O(V + E)**, where V is the number of vertices and E is the number of edges, since in the worst case every vertex and edge is explored once. The space complexity is **O(V)** in the case of recursion due to the depth of the call stack or the explicit stack used, which can be significant in deep or dense graphs [1].

Advantages: It is relatively simple to implement and works well for problems where solutions lie deep in the graph, such as searching paths in a maze. Since it follows one path deeply before considering alternatives, it can be memory-efficient in sparse graphs where branching is limited. It is also useful for detecting cycles in directed and undirected graphs [1].

Limitations: It does not always find the shortest path in weighted or unweighted graphs, unlike Breadth First Search (BFS). It can also get stuck exploring long or infinite paths if not carefully controlled, especially in graphs with cycles, which is why maintaining a "visited" set is critical. Additionally, its performance can degrade in very deep graphs due to stack overflow in recursive implementations. DFS is also less optimal when the solution is likely to be found closer to the root, as it may unnecessarily explore deep branches before reaching the correct node [1].

2.4 BREADTH FIRST SEARCH

Breadth First Search (BFS) is a fundamental graph traversal algorithm that explores nodes level by level, making it particularly effective for finding the shortest path in unweighted graphs. Starting from a source node, BFS visits all its immediate neighbors before moving on to the neighbors' neighbors, and so on, until all reachable nodes have been visited. This level-order exploration is achieved using a queue data structure, where nodes are enqueued when discovered and dequeued when explored. BFS is commonly used in applications such as shortest path finding in unweighted graphs, peer-to-peer networks, crawling web pages, broadcasting in networks, and solving puzzles like finding the minimum number of moves in a game. The time complexity of BFS is **O(V + E)**, where V is the number of vertices and E is the number of edges, since every vertex and edge is processed at most once. The space complexity is also **O(V)**, due to the storage requirements of the queue and the visited list. In dense graphs, this can result in significant memory usage compared to DFS [1].

Advantages: The most notable is that it guarantees the discovery of the shortest path in an unweighted graph, which makes it more suitable than DFS in pathfinding problems where efficiency and minimal distance matter. Its level-order traversal makes it well-suited for applications that require visiting nodes in layers, such as spreading information in a network or analyzing social connections. BFS is also non-recursive by

nature (iterative), so it avoids the stack overflow risks that recursive DFS implementations may face in very deep graphs.

Limitations: It requires more memory than DFS, since the queue can grow large, particularly when exploring wide or dense graphs where many nodes exist at the same level. Unlike DFS, BFS does not perform well in situations where the target node is located deep in the graph, as it must explore all nodes at shallower levels first. Additionally, BFS can be more complex to implement compared to DFS due to the explicit use of a queue and management of node states.

2.5 A* SEARCH

A* (pronounced "A-star") is one of the most widely used and powerful graph search algorithms, particularly in the field of artificial intelligence and pathfinding. It is essentially an informed search algorithm that combines the strengths of Dijkstra's Algorithm (which finds the shortest path using actual costs) and Greedy Best-First Search (which uses heuristic estimates to guide the search). A* works by maintaining a priority queue (often implemented as a min-heap) of nodes to be explored [2].

Each node is assigned a cost function:

$$f(n) = g(n) + h(n)$$

where,

f(n) = the total estimated cost of the cheapest path passing through node n.

g(n) = the actual cost from the start node to the current node.

h(n) = a heuristic estimate of the cost from the current node to the goal.

At every step, A* selects the node with the lowest f(n) value for expansion. If the heuristic h(n) is admissible (never overestimates the true cost) and consistent, A* is guaranteed to find the shortest path [3].

The time complexity of A* varies depending on the quality of the heuristic. In the worst case, it can approach **O(b^d)**, where b is the branching factor and d is the depth of the solution, because it may explore many nodes. However, with a good heuristic, A* prunes large portions of the search space, making it much faster in practice. The space complexity is typically **O(b^d)** as well, since A* must keep track of all visited nodes in memory, which can be a drawback in large search spaces [2].

Advantages: It is optimal and complete when using an admissible heuristic, meaning it is guaranteed to find the shortest path if one exists. Its heuristic-driven approach makes it far more efficient than uninformed searches like BFS or DFS, especially in large or complex search spaces. A* is widely used in real-world applications such as robotics, video games (pathfinding for characters), GPS navigation systems, and AI planning problems, where both speed and accuracy are critical.

Limitations: Its biggest drawback is memory consumption: because it stores all generated nodes in memory, it can quickly run out of space when solving very large problems. The efficiency of A* is heavily dependent on the quality of the heuristic: a poor heuristic (e.g., always returning 0) essentially reduces A* to Dijkstra's Algorithm, which can be slow. On the other hand, an overestimating heuristic breaks optimality, leading to incorrect results. Implementing a well-designed heuristic requires domain knowledge, which may not always be available.

3. OBJECTIVES

To evaluate these algorithms, two programming languages, **Java** and **Python** are selected owing to their widespread usage, distinct execution paradigms, and contrasting performance characteristics. **Java**, being a *statically typed and compiled language*, leverages **Just-In-Time (JIT) compilation** and **automatic garbage collection**. This enables it to transform bytecode into optimized machine code at runtime, reducing overhead and ensuring consistent performance for large-scale and multithreaded applications [4]. This approach is often referred to as **ahead-of-time runtime optimization**, which explains why Java is widely used in enterprise-grade systems and backend processing.

Python, conversely, is an *interpreted and dynamically typed language*, emphasizing **developer productivity** and **ease of experimentation**. It operates through an interpreter rather than direct compilation, meaning the code is executed line by line. This characteristic, while making Python slower in raw execution speed, enhances flexibility in rapid prototyping and iterative development [5]. Over time, however, Python's performance has been substantially improved by **highly optimized libraries** such as **NumPy**, **bisect**, and **pandas**, which internally use efficient C-based operations. This layered architecture is known as **hybrid execution**, allowing Python to balance readability with computational efficiency.
The primary objectives of this study are detailed as follows:

3.1 PERFORMANCE EVALUATION

To empirically measure and analyze the **execution times** of widely used **search algorithms** including Linear Search, Binary Search, and Graph Traversal methods (DFS and BFS) in both Java and Python. This process is known as **empirical benchmarking**, which involves executing algorithms on datasets of varying sizes and complexities to determine their real-world efficiency. Such evaluation helps verify whether theoretical complexities (like *O(n)* or *O(log n)*) align with actual implementation outcomes, a process often referred to as **practical complexity validation**.

3.2 COMPARATIVE ANALYSIS

To investigate how **language-specific characteristics** such as memory management techniques, runtime optimizations, and compiler/interpreter behavior affect algorithmic performance. For instance, Java's use of the **Java Virtual Machine (JVM)** provides consistent optimization through JIT, while Python's interpreter relies on **dynamic type inference**, which can introduce overhead but allows more adaptability. This objective is also known as **cross-language performance profiling**, a method that helps identify how different execution models influence the same algorithmic logic.

3.3 PRACTICAL INSIGHTS

To provide **actionable guidelines and recommendations** for developers and researchers in selecting the most appropriate language and search approach for **data-intensive and performance-critical applications**. This goal is sometimes referred to as **applied performance translation**, where theoretical analysis is converted into practical decision-making frameworks. It seeks to answer questions like *"Which language should be chosen when speed, scalability, or ease of implementation is the priority?"* thereby assisting practitioners in making informed design choices in real-world systems.

By bridging the divide between theoretical algorithmic complexity and practical implementation results, this study emphasizes the intricate relationship among algorithm design, data structure behavior, and language-specific execution models. This holistic perspective underscores how programming paradigms and optimization strategies shape the true computational efficiency of algorithms in modern computing environments.

4. PERFORMANCE MEASUREMENT

4.1 EXPERIMENT SETUP

To evaluate and compare the performance of search algorithms implemented in java and python, a structured experimental setup was carried out under identical hardware and software environments to ensure fairness and reproducibility.

4.1.1 Environment Specification

All experiments were conducted on a machine equipped with an intel core i7 processor, 8 GB ram, and running windows 11 (64-bit). For the java implementations, the code was compiled and executed using the OpenJDK 17 environment, while the python programs were executed with CPython version 3.10. Integrated development environments (ides) such as eclipse (for java) and visual studio code (for python) were used to manage code execution and consistency.

4.1.2 Data Levels

To study the effect of input size on algorithm performance, eight progressively increasing dataset levels were prepared. This approach allowed a fine-grained observation of performance scaling across small to very large inputs.
Datasets were generated using random number generators for numerical search tasks and adjacency list/matrix structures for graph-based algorithms such as DFS, BFS, and A*. Both Java and Python implementations operated on identically structured datasets to eliminate bias.

4.1.3 Implementation Procedure

The selected algorithms, linear search, binary search, depth first search (DFS), breadth first search (BFS), and a* search, were implemented independently in both java and python. Each implementation was designed to follow the same logical flow and input-output structure, ensuring that observed performance differences were attributable to language-level characteristics rather than implementation discrepancies.

4.1.4 Performance Measurement Approach

Metric considered: Execution time: measured using built-in high-resolution timers. In java, the system.nanotime() method was used, while in python, the time.perf_counter() function was employed. Execution time was recorded in milliseconds for consistency.
Each experiment was repeated ten times per dataset size, and the average values of execution time were considered to minimize the impact of transient fluctuations.

4.1.5 Execution Workflow

The experimental workflow proceeded as follows:
- Load dataset of defined size.

- Run algorithm implementation in java and record runtime.

- Run algorithm implementation in python on the same dataset and record runtime.

- Repeat steps 1–3 for each algorithm and each dataset size.

- Compute average values across all repetitions.

- Store results in tabular form (csv/excel) for further analysis.

4.1.6 Result Analysis

The collected results were plotted to visualize the comparative performance of java and python across different dataset sizes. Separate graphs were generated for runtime performance and memory consumption. These visualizations enabled identification of scalability patterns, trade-offs, and the overall efficiency of each language in handling search algorithms.

4.2 EXPERIMENT OBSERVATIONS & ANALYSIS

4.2.1 LINEAR SEARCH

Linear search is a fundamental, straightforward search algorithm characterized by its,

O(n) time complexity, meaning it sequentially examines each element of a dataset until the target is located. This report evaluated the performance of linear search in Java (JIT compiled) and Python (interpreted) using unsorted integer arrays of increasing sizes, ranging from 5,000 to 40,000 elements, with the target consistently set to the last element to represent a worst-case scenario.

The experimental results confirmed the expected theoretical behavior: execution time increased linearly with dataset size in both languages. Java consistently exhibited significantly better performance, benefiting from Just-In-Time (JIT) compilation and low-level optimizations. Python was approximately 40 times slower than Java, primarily due to its interpreted nature and slower loop execution.

While both languages demonstrated a manageable difference on smaller datasets (5k–15k), the performance gap became substantial for larger datasets (35k–40k). The sustained linear growth in both languages validated the O(n) theoretical complexity but underscored Java's practical advantage, making it the preferred choice for repeated linear search operations on large datasets. Conversely, Python remains suitable for prototyping but is slower for raw linear search on large arrays. The high sensitivity of linear search to dataset size ultimately highlights the inefficiency of the algorithm for large inputs, suggesting the importance of more optimized alternatives.

Table 1. Comparison of Java and Python in Linear Search

Level	Sample Size	Execution Time in Java (ms)	Execution Time in Python (ms)
1	5,000	0.007387	0.270491
2	10,000	0.014836	0.565209
3	15,000	0.022180	0.818377
4	20,000	0.029654	1.075892
5	25,000	0.039595	1.375556
6	30,000	0.046089	1.597479
7	35,000	0.054585	2.061335
8	40,000	0.062778	2.526917

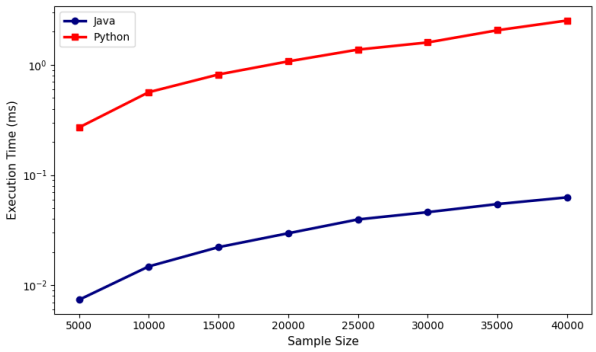

Fig 1: Graph Plot for Time Taken to Perform Linear Search

4.2.2 Binary Search

Binary search is an efficient search algorithm with a time complexity of **O(log n)**, which operates exclusively on sorted arrays. It works by repeatedly dividing the search interval in half until the target element is located [1]. This report evaluated the binary search performance in both Java (JIT compiled) and Python (interpreted) using sorted integer arrays ranging in size from 5,000 to 40,000 elements.

The experimental results confirmed the expected logarithmic scaling of the algorithm in both languages. Both Java and Python exhibited extremely fast per-search times, with the overall time trend being almost flat as the dataset size increased. This minimal growth highlights the exceptional efficiency of binary search, even on large sorted datasets, with both languages handling the 40,000-element array in under 0.01 ms per search. The tiny increase in time as the dataset size grew slightly validated the O(log n) complexity. Interestingly, the Python times (~0.0015-0.0019 ms) appeared to be consistently lower than the Java times (~0.004-0.009 ms) for several datasets. These differences are considered minor (microseconds scale) and are largely attributed to measurement artifacts rather than actual algorithmic inefficiency. The primary reasons for the observed higher-per-search times in Java measurements include:

1. JIT Compilation Overhead / Warm-up Effects: Java uses a Just-In-Time (JIT) compiler to dynamically optimize code. While warm-up runs were included to mitigate this, the first few runs of a method can still be interpreted or partially optimized, leading to slightly higher initial timings or minor fluctuations in the data.

2. Timer and Measurement Noise: The operation itself is extremely fast (~ nanoseconds per comparison). When measuring such short durations, the overhead and resolution of the high-resolution timer (System.nanoTime() in Java) can make Java's measured times appear higher than those recorded by Python's time.perf_counter(), due to differences in how the respective runtime environments handle timer calls.

3. Cache and Memory Effects: For extremely small operations, factors like CPU cache states, branch prediction, and memory access patterns can introduce variability. JIT-compiled code might experience minor cache misses in some iterations, slightly increasing a few measurements.

4. Operating System Scheduling: Background processes, thread scheduling, and context switches from the operating system can inject small, unpredictable delays into the Java runtime.

While the raw microsecond measurements suggested Python was faster, these differences are noise. In practice, Java would typically outperform Python for larger datasets or realistic workloads due to its compilation benefits. The tiny variations in Java do not affect the overwhelming algorithmic advantage of binary search over linear search.

Table 2. Comparison of Java and Python in Binary Search

Level	Sample Size	Execution Time in Java (ms)	Execution Time in Python (ms)
1	5,000	0.009004	0.001452
2	10,000	0.007440	0.001535
3	15,000	0.005769	0.001612
4	20,000	0.004233	0.001655
5	25,000	0.004096	0.001684
6	30,000	0.004058	0.001693
7	35,000	0.007083	0.001815
8	40,000	0.005517	0.001875

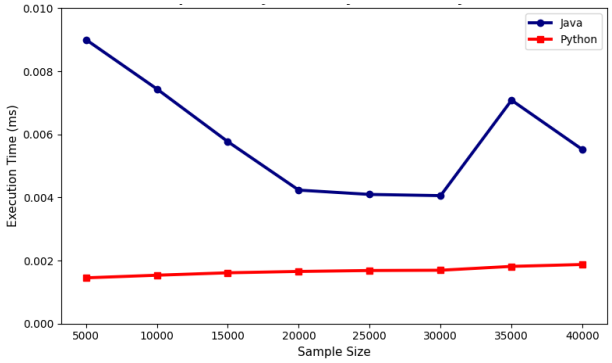

Fig 2: Graph Plot for Time Taken to Perform Binary Search

4.2.3 Depth First Search

Depth-First Search (DFS) is a graph traversal algorithm with a time complexity of O(V+E), where V is the number of nodes (vertices) and E is the number of edges [1]. This report evaluates the performance of an iterative DFS implementation using a stack and adjacency list in both Java (JIT compiled) and Python (interpreted) across large undirected, connected graphs. Eight datasets were used, with graph sizes increasing from 10,000 nodes and 50,000 edges up to 500,000 nodes and 2,500,000 edges. Only the DFS traversal was timed, excluding graph construction.

The benchmark results confirmed that DFS times increase approximately linearly with V+E, consistent with the O(V+E) complexity in both Java and Python. Both languages showed similar scaling trends, which is ideal for comparative studies. However, Java was consistently faster, particularly on larger datasets, benefiting from its JIT compilation and optimizations. For the largest graph (Level 8: 500,000 nodes, 2,500,000 edges), Java completed the DFS traversal in 193.537 ms, while Python took 920.437 ms. This means Java was approximately 4.8 times faster than Python for the most demanding benchmark. The performance disparity is attributed to the inherent differences between the language execution models.

Java's Advantage: Java's JVM and JIT compilation allow for significant low-level code optimization and highly efficient loop execution [4] and data structure (stack/adjacency list) manipulation after warm-up. This provides superior absolute performance and more stable scaling as the graphs become massive.

Python's Limitation: Python's interpreted nature introduces considerable interpreter overhead during the frequent loop iterations required for graph traversal. This overhead, compounded by dynamic typing, means Python's DFS time grows significantly and disproportionately with graph size compared to Java.

In conclusion, Java provides better performance for large-scale graph algorithms like DFS, making it suitable for production environments or performance-critical research. Python is better suited for algorithm testing, prototyping, or smaller datasets.

Table 3. Comparison of Java and Python in Depth First Search

Level	No. of Nodes	No. of Edges	Execution Time in Java (ms)	Execution Time in Python (ms)
1	10,000	50,000	7.552	3.751
2	20,000	100,000	7.421	13.713
3	40,000	200,000	13.067	48.637
4	60,000	300,000	26.438	78.174
5	80,000	400,000	27.037	106.080
6	100,000	500,000	38.354	138.614
7	200,000	1,000,000	63.495	307.667
8	500,000	2,500,000	193.537	920.437

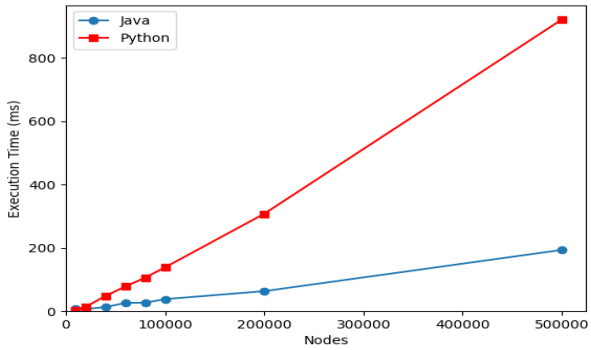

Fig 3: Graph Plot for Time Taken to Perform Depth First Search

4.2.4 BREADTH FIRST SEARCH

Breadth-First Search (BFS) is a graph traversal algorithm with a time complexity of O(V+E), identical to DFS, where V is the number of nodes (vertices) and E is the number of edges [1]. This report evaluates the performance of an iterative BFS implementation using a queue and an adjacency list representation in both Java (JIT compiled) and Python (interpreted) across a set of large undirected, connected graphs. The datasets ranged in size from 10,000 nodes and 50,000 edges up to 500,000 nodes and 2,500,000 edges. Only the time taken for the BFS traversal was measured, excluding the time required for graph construction.

The benchmark results consistently showed that BFS runtime grows approximately linearly with V+E, which aligns with the expected O(V+E) complexity.

Java was significantly more efficient for large-scale BFS due to its compiled performance. For the largest graph (Level 8: 500,000 nodes, 2,500,000 edges), Java completed the BFS in 224.460 ms, while Python took 969.520 ms. This means Java was approximately 4.3 times faster than Python for the most complex graph traversal. The performance difference is primarily due to the execution model of the languages:

Java's Advantage: Java benefits from its Just-In-Time (JIT) compiler, which optimizes the bytecode into machine code, leading to highly efficient execution of the core traversal loops and queue operations [4]. This compiled performance minimizes overhead, resulting in faster and more stable execution times, particularly as graph size increases.

Python's Limitations: Python, being an interpreted language, suffers from interpreter overhead and the overhead of dynamic typing [5] during the massive number of operations required for large graph traversal. This overhead amplifies as the graph size grows, causing the runtime to increase more rapidly compared to Java.

In conclusion, Java is recommended for large-scale, performance-critical BFS experiments, while Python is more suitable for algorithm prototyping or smaller datasets.

Table 4. Comparison of Java and Python in Breadth First Search

Level	No. of Nodes	No. of Edges	Execution Time in Java (ms)	Execution Time in Python (ms)
1	10,000	50,000	5.217	3.671
2	20,000	100,000	8.818	13.396
3	40,000	200,000	14.203	43.467
4	60,000	300,000	26.559	80.617
5	80,000	400,000	32.636	114.399
6	100,000	500,000	46.969	141.803
7	200,000	1,000,000	76.682	321.228
8	500,000	2,500,000	224.460	969.520

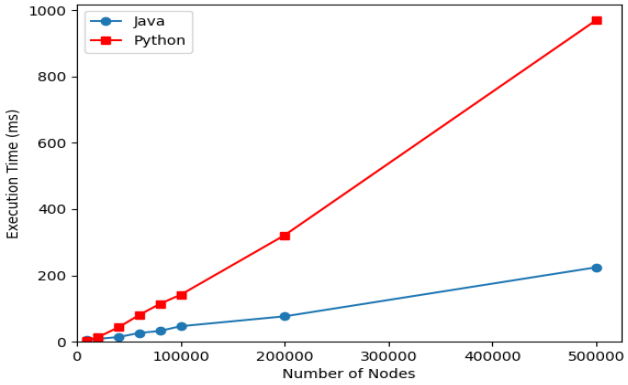

Fig 4: Graph Plot for Time Taken to Perform Breadth First Search

4.2.5 A SEARCH*

A* search is an informed search algorithm used widely in graph applications and AI. It is distinct from uninformed strategies like DFS or BFS because it uses a heuristic to guide the search, combining the cost to reach a node (g) with an estimate of the cost to the goal (h) [3].

Here, the performance of A* in Java (JIT compiled) and Python (interpreted) is compared on eight levels of weighted graphs ranging from 10,000 nodes and 50,000 edges up to 500,000 nodes and 2,500,000 edges. Both implementations utilized a Priority Queue (Java's PriorityQueue and Python's heapq) for management of the open set. Java consistently outperformed Python across all graph sizes, with runtimes generally being 3× to 10× faster on average. Java's JIT compilation and optimized data structures provided a significant advantage. For the largest graph tested (Level 8: 500,000 nodes), Java completed the

search in 106.257 ms, while Python took 797.308 ms. For large graphs, Python's runtime grew disproportionately, while Java demonstrated more stable scaling.

The factors contributing to the disproportions are,

A* Runtime Complexity: A* search time is heavily influenced by factors beyond just the number of nodes and edges, including heuristic accuracy, graph density, and the start-to-goal distance [2]. An observed decrease in Java's runtime for Level 8 compared to Level 7 (106 ms < 180 ms) is likely due to the graph's structure or effective heuristic guidance, resulting in fewer nodes being explored.

Language Overhead: Java benefits from JIT compilation, which ensures faster execution and stable scaling after a warm-up period. Conversely, Python's performance is hampered by interpreter overhead, dynamic typing, and comparative inefficiency in large-scale heap operations (using heapq), which severely amplify runtime as graph size increases [6].

Conclusion: Java is the recommended language for large-scale, performance-critical A* experiments. Python remains suitable for prototyping and educational purposes but will not scale efficiently for complex graph problems.

Table 5. Comparison of Java and Python in A* Search

Level	No. of Nodes	No. of Edges	Execution Time in Java (ms)	Execution Time in Python (ms)
1	10,000	50,000	4.225	17.542
2	20,000	100,000	3.987	8.987
3	40,000	200,000	13.028	103.371
4	60,000	300,000	5.425	41.598
5	80,000	400,000	2.852	31.038
6	100,000	500,000	2.136	20.197
7	200,000	1,000,000	180.055	1252.108
8	500,000	2,500,000	106.257	797.308

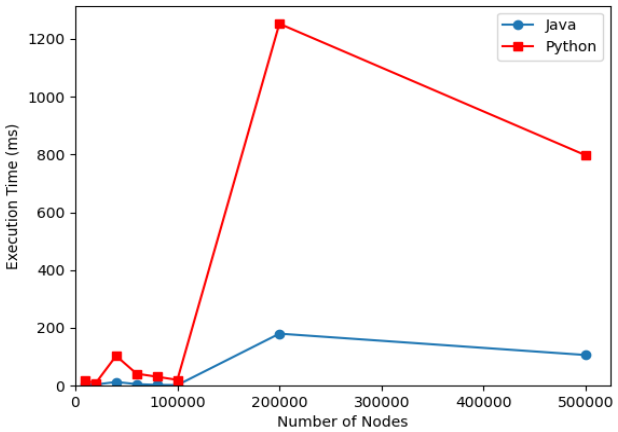

Fig 5: Graph Plot for Time Taken to Perform A* Search

5. CONCLUSION

This research systematically benchmarked the practical runtime performance of five fundamental algorithms, Linear Search, Binary Search, Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search, across two dominant execution environments: Java (Just-In-Time compiled) and Python (Interpreted). The primary objective was to quantify the performance differential attributable to the language execution model when scaling dataset sizes up to 40,000 elements for searching and 500,000 nodes for graph traversal.

5.1 Synthesis of Core Findings

Validation of Theoretical Complexity: The experimental data conclusively validated the theoretical time complexities of all algorithms: Linear Search adhered strictly to O(n) scaling, while Binary Search demonstrated near-constant time performance consistent with O(log n). Similarly, the graph algorithms (DFS, BFS, A*) exhibited scaling directly related to O(V+E), where traversal time increased linearly with the total volume of nodes and edges.

5.2 Impact of Language Execution Model

A clear and consistent performance hierarchy was established, dictated by the frequency of low-level operational overhead:

5.2.1 High Frequency Operations

(O(n) and O(V+E)), For algorithms requiring vast numbers of loop iterations and low-level memory/data structure accesses (Linear Search, DFS, BFS), Java demonstrated overwhelming superiority. In the worst-case Linear Search scenario, Java was up to 40 times faster than Python on large arrays. For graph traversals (DFS/BFS), Java maintained a speed advantage of 4 to 5 times on the largest graphs, underscoring the performance cost incurred by Python's interpreter and dynamic typing during iterative operations.

5.2.2 Low Frequency Operations

(O(log n)), For the highly efficient Binary Search, the search time was extremely minimal (microseconds), and the observed performance difference between the two languages was marginal and largely attributable to timer and measurement noise rather than significant algorithmic inefficiency. This suggests that for operations with minimal computational steps, the language overhead is negligible.

5.2.3 Algorithmic Nuance (A)*

The A* search results highlighted that algorithmic factors can sometimes overshadow language overhead. While Java was consistently faster (up to 10 times), the raw runtime data showed non-linear scaling dependent on the effectiveness of the heuristic and the specific sparsity of the graph, confirming that A* complexity is path-dependent and not solely a function of graph size.

5.3 Implications and Recommendations

The findings carry significant implications for software development and computational research. Java is unequivocally the recommended choice for production environments and large-scale computational experiments where absolute performance, stable scaling, and minimal overhead are critical. Its JIT compilation minimizes the performance penalties associated with high-iteration algorithms. Conversely, Python, while indispensable for its ease of use and rapid prototyping, is best reserved for small-to-medium dataset analysis, educational purposes, or algorithm development where the integration of optimized C-extension libraries (such as NumPy or SciPy) can mitigate the performance bottleneck of the native interpreter. Future work should focus on integrating these highly optimized Python libraries into the benchmark to determine if the performance gap observed in this study can be substantially closed by leveraging native code execution within the Python ecosystem.

6. ACKNOWLEDGEMENTS

The authors would like to express their sincere gratitude to the experts and contributors involved in the development and continuous refinement of the IJCA. Their dedicated efforts in creating clear formatting standards and comprehensive guidelines have greatly supported the preparation and presentation of this paper. The authors truly appreciate the dedication and expertise that have gone into creating a template that supports researchers and enhances the overall presentation of scholarly work. The authors would also like to extend their heartfelt appreciation to Dr. Imtiyaz Khan, Associate Professor and Head of the Department of Information Technology, for his valuable guidance, encouragement, and insightful feedback throughout the course of this work. His consistent support and academic leadership have been instrumental in shaping the direction and quality of this research.

7. REFERENCES

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. 2009. Introduction to algorithms (3rd ed.). The MIT Press.

[2] Rios, L. H. O., & Chaimowicz, L. 2010. A survey and classification of A*-based best-first heuristic search algorithms. In Advances in artificial intelligence – SBIA 2010 (pp. 253–262). Springer Berlin Heidelberg.

[3] Chowdhary, K. R. 2020. Heuristic search. In Fundamentals of artificial intelligence (pp. 239–272). Springer India.

[4] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., & Bierman, G. 2025. The Java® Language Specification: Java SE 25 Edition.

[5] Hetland, M. L. 2014. Python algorithms: Mastering basic algorithms in the Python language. Apress.

[6] Wang, Y. 2023. Runtime comparative analysis of Java and Python programs with algorithms of different time complexities. Bradley University.

[7] Durrani, O. K., & Abdulhayan, S. 2022. Performance measurement of popular sorting algorithms implemented using Java and Python.

[8] Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. 2013. Data Structures and Algorithms in Python. Wisley.

[9] Khoirom, S., Sonia, M., Laikhuram, B., Laishram, J., & Singh, T. D. 2020. Comparative analysis of Python and Java for beginners. International Research Journal of Engineering and Technology (IRJET), 7(8).

[10] Cutting, V., & Stephen, N. 2022. Comparative review of Java and Python. International Journal of Research and Development in Applied Science and Engineering (IJRDASE).

[11] Naveed, M. S. 2024. Pedagogical suitability: A software metrics-based analysis of Java and Python. ResearchGate.