Comparative Analysis of Graph Search Algorithms for Wikipedia Navigation

Philipp Hamara - 13524101
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: philipphamara420@gmail.com, 13524101@std.stei.itb.ac.id

***Abstract*— Wikipedia is one of the largest publicly accessible knowledge bases, where articles are interconnected through hyperlinks that naturally form a directed graph. Finding a navigation path between two articles, commonly referred to as Wikipedia navigation or Wikipedia racing, can be formulated as a graph search problem. While various online applications provide solutions for this task, limited work has focused on comparing the performance of classical graph search algorithms within this domain.**

This paper presents a comparative analysis of several graph search algorithms, namely Breadth-First Search (BFS), Bidirectional Breadth-First Search (BiBFS), and Iterative Deepening Depth-First Search (IDDFS), for navigating the Wikipedia hyperlink graph. Each algorithm is evaluated using identical navigation tasks and compared based on execution time, number of explored nodes, memory consumption, and path optimality. By conducting experiments on a real-world large-scale graph, this study highlights the strengths and limitations of each search strategy and examines how the structural characteristics of Wikipedia influence algorithmic performance.

***Keywords*— Graph Search, Breadth-First Search, Bidirectional Breadth-First Search, Iterative Deepening Depth-First Search, Wikipedia, Hyperlink Graph, Algorithm Analysis**

I. INTRODUCTION

Graph search algorithms are fundamental techniques in computer science and artificial intelligence, with applications ranging from network routing and robotics to social network analysis and knowledge graph exploration. Selecting an appropriate search strategy is essential because different algorithms exhibit varying trade-offs between execution time, memory usage, and solution optimality. The choice of algorithm can dramatically impact performance depending on the structural properties of the underlying graph.

Wikipedia is a collaboratively maintained online encyclopedia consisting of millions of interconnected articles. Each article contains hyperlinks to related topics, forming a large directed graph in which articles represent vertices and hyperlinks represent edges. This interconnected structure makes Wikipedia an excellent real-world dataset for evaluating graph search algorithms. Its hyperlink network spans virtually all domains of human knowledge and contains dense local clusters connected by relatively few bridging links.

One interesting application of graph search on Wikipedia is the problem of navigating from one article to another using only hyperlinks. This task, commonly known as Wikipedia racing or Wikipedia speedrun, has become a popular online game in which players attempt to reach a target article from a starting article in as few clicks as possible. Although several web-based tools provide shortest-path solutions, these implementations primarily focus on producing navigation results rather than analyzing the behavior and efficiency of different search strategies. Consequently, there is an opportunity to evaluate how classical graph search algorithms perform on a real-world graph with small-world characteristics.

The study of Wikipedia's network structure has been the subject of considerable research. Studies have shown that the Wikipedia link structure exhibits scale-free and small-world properties, with a short average path length between any two articles. Further analysis of information flow within Wikipedia's semantic network has confirmed the efficient navigability of the graph. These findings provide a theoretical foundation for expecting that search algorithms can efficiently find paths between Wikipedia articles.

This research compares the performance of Breadth-First Search (BFS), Bidirectional Breadth-First Search (BiBFS), and Iterative Deepening Depth-First Search (IDDFS) when solving Wikipedia navigation problems. These three algorithms represent distinct approaches to graph search: BFS explores systematically level by level, BiBFS attacks the problem from both ends simultaneously, and IDDFS trades repeated computation for minimal memory usage. Each algorithm belongs to the class of uninformed search strategies, meaning they do not use domain-specific knowledge to guide the search. The comparison is conducted using identical navigation tasks while measuring execution time, memory consumption, number of expanded nodes, and path optimality.

The contents of this paper are as follows:

1) Modeling Wikipedia's hyperlink structure as a directed graph suitable for graph search, with complete API integration and local caching to enable reproducible experimentation.
2) Implementing multiple classical graph search algorithms within the same experimental environment using a common graph interface to ensure fair comparison.

3) Experimentally comparing the performance of each algorithm using eight real-world navigation tasks spanning diverse Wikipedia topic areas.
4) Analyzing how the structural properties of Wikipedia influence the effectiveness of different search strategies and providing practical guidance for algorithm selection in hyperlink graph navigation.

II. THEORETICAL BASIS

A. Graph Theory

Graph theory provides the mathematical foundation for representing relationships between interconnected entities. A graph is defined as $G = (V, E)$, where V denotes the set of vertices and E denotes the set of edges connecting those vertices. In a directed graph, each edge $(u, v) \in E$ has an orientation, indicating a relationship from vertex u to vertex v that is not necessarily symmetric. The out-degree of a vertex is the number of edges originating from it, while the in-degree is the number of edges terminating at it.

In this study, Wikipedia articles are modeled as vertices while hyperlinks between articles are represented as directed edges. A hyperlink from article A to article B is represented as a directed edge (A, B). The inverse relationship (which articles link to a given article) must be queried separately through the Wikipedia API using the `linkshere` endpoint, making bidirectional traversal more expensive in terms of API calls.

B. Wikipedia Hyperlink Graph

Wikipedia naturally forms a large directed graph consisting of millions of articles connected through hyperlinks. As of 2025, the English Wikipedia contains over 6.9 million articles, each containing an average of dozens of outbound hyperlinks. Due to its dense connectivity and relatively short average shortest-path length, the network exhibits characteristics commonly associated with small-world networks. The small-world property implies that most pairs of articles can be connected through a relatively short chain of intermediate articles, typically in the range of 3 to 6 clicks.

The Wikipedia hyperlink graph also exhibits a power-law degree distribution, meaning that a small number of highly connected hub articles (such as "Science," "Biology," or "Time") account for a large proportion of the total edges. This structure has important implications for graph search algorithms, as the presence of hub nodes can dramatically accelerate search when they appear on the path between source and target articles.

C. Breadth-First Search

Breadth-First Search (BFS) is one of the most fundamental graph traversal algorithms. It explores vertices level by level beginning from the source vertex, systematically examining all vertices at distance k before moving to distance $k+1$. The algorithm maintains a queue of vertices to be explored and a set of visited vertices to avoid redundant exploration.

For unweighted graphs, BFS guarantees the shortest path in terms of the number of traversed edges. The algorithm is both complete (it will always find a path if one exists) and optimal with respect to path length. The pseudocode for BFS is shown in Algorithm 1.

Algorithm 1 Breadth-First Search

```
function BFS(graph, start, goal)
    queue ← [start]
    visited ← {start}
    parent ← {}
    while queue ≠ [] do
        node ← queue.POP(0)
        if node = goal then
            return RECONSTRUCTPATH(parent, start, goal)
        end if
        for each neighbor ∈ GETNEIGHBORS(graph, node) do
            if neighbor ∉ visited then
                visited ← visited ∪ {neighbor}
                parent[neighbor] ← node
                queue.APPEND(neighbor)
            end if
        end for
    end while
    return null
end function
```

The worst-case time complexity of BFS is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges, because every vertex and every edge is potentially explored. The space complexity is also $O(|V|)$, as the queue and visited set must store all vertices in the worst case. This memory requirement can become prohibitive for very large graphs, as BFS must store the entire frontier of the search at each level.

D. Bidirectional Breadth-First Search

Bidirectional Breadth-First Search (BiBFS) improves upon BFS by simultaneously performing search from both the source and destination vertices. The forward search follows outgoing hyperlinks from the start article, while the backward search follows incoming hyperlinks (i.e., articles that link to the goal article). The search terminates once the two search frontiers intersect.

The key insight behind BiBFS is that if both searches expand to depth $\frac{d}{2}$ before meeting, the total number of explored vertices in a graph with branching factor b is approximately $2b^{d/2}$, compared to b^d for standard BFS. For large graphs with high connectivity, this can represent a massive reduction in search space. The algorithm maintains shortest-path optimality because both directions use BFS, so the meeting point is guaranteed to be on a shortest path.

Algorithm 2 Bidirectional Breadth-First Search

```
function BIDIRECTIONALBFS(graph, start, goal)
    f_queue ← [start], b_queue ← [goal]
    f_visited ← {start}, b_visited ← {goal}
    f_parent ← {}, b_parent ← {}
    while f_queue ≠ [] and b_queue ≠ [] do
        if |f_queue| ≤ |b_queue| then
            node ← f_queue.POP(0)
            for each neighbor ∈ GETNEIGHBORS(graph,node) do
                if neighbor ∉ f_visited then
                    f_visited ← f_visited ∪ {neighbor}
                    f_parent[neighbor] ← node
                    f_queue.APPEND(neighbor)
                    if neighbor ∈ b_visited then
                        return MERGEPATHS(f_parent, b_parent, neighbor)
                    end if
                end if
            end for
        else
            node ← b_queue.POP(0)
            for each pred ∈ GETPREDECESSORS(graph,node) do
                if pred ∉ b_visited then
                    b_visited ← b_visited ∪ {pred}
                    b_parent[pred] ← node
                    b_queue.APPEND(pred)
                    if pred ∈ f_visited then
                        return MERGEPATHS(f_parent, b_parent, pred)
                    end if
                end if
            end for
        end if
    end while
    return null
end function
```

The worst-case time complexity of BiBFS remains $O(|V|+|E|)$, but in practice it explores far fewer vertices than BFS on graphs where the two search frontiers meet well before exhausting the graph. The space complexity is $O(b^{d/2})$ for both frontiers combined, which can be significantly lower than BFS in some cases but may also be higher if the backward search explores a large region.

E. Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search (IDDFS) combines the low memory consumption of Depth-First Search (DFS) with the completeness and optimality of BFS. The algorithm repeatedly executes depth-limited DFS with increasing depth limits until the destination is reached. At each iteration, the algorithm is restricted to exploring paths no longer than the current depth limit.

Although IDDFS may appear wasteful because it re-explores nodes at each iteration, the overhead is relatively small for graphs with a moderate branching factor. For a graph with branching factor b and solution depth d, IDDFS explores approximately $\frac{b^d(b-1)}{b-1}$ nodes, which is only about $\frac{b}{b-1}$ times the number explored by BFS. A depth cap must be set to prevent infinite exploration in graphs with cycles; in this implementation, the cap is set to 20.

Algorithm 3 Iterative Deepening Depth-First Search

```
function IDDFS(graph, start, goal, max_depth)
    for depth ← 0 to max_depth do
        ancestors ← {start}
        result ← DLS(graph, start, goal, depth, ancestors)
        if result.found then
            return result.path
        end if
    end for
    return null
end function

function DLS(graph, node, goal, depth, ancestors)
    if node = goal then
        return FOUND([node])
    end if
    if depth = 0 then
        return NOTFOUND
    end if
    for each neighbor ∈ GETNEIGHBORS(graph,node) do
        if neighbor ∉ ancestors then
            ancestors ← ancestors ∪ {neighbor}
            result ← DLS(graph, neighbor, goal, depth − 1, ancestors)
            if result.found then
                result.path.PREPEND(node)
                return result
            end if
        end if
    end for
    return NOTFOUND
end function
```

The time complexity of IDDFS is $O(b^d)$ in the worst case, where b is the branching factor and d is the solution depth. The space complexity is $O(bd)$, which is significantly better than BFS. However, IDDFS can be dramatically slower than BFS for deep solutions on graphs with a high branching factor, as is the case with Wikipedia.

III. IMPLEMENTATION

This section describes the system architecture, the graph modeling approach, the algorithm implementations, and the benchmark design used in this study.

A. System Architecture

The system is implemented in Python and consists of four main components: the Wikipedia API wrapper, the graph abstraction layer with disk caching, the algorithm

implementations, and the benchmark runner. The architecture follows a layered design where each component has a well-defined responsibility.

The Wikipedia API wrapper handles all communication with the Wikimedia REST API. It implements rate limiting with a configurable delay between requests and exponential backoff for HTTP 429 (rate limit exceeded) responses, with a maximum cooldown of 300 seconds. The wrapper supports both forward link retrieval (`prop=links`) and backward link retrieval (`prop=linkshere`), as well as batch fetching of up to 50 articles in a single API call.

The graph layer abstracts the underlying API and cache, presenting a uniform interface to the search algorithms. Each algorithm receives a graph object through its constructor and calls `get_neighbors(title)` or `get_predecessors(title)` without needing to be aware of whether the data comes from the API or the local cache.

B. Graph Modeling and Caching

Modeling Wikipedia as a graph presents unique challenges due to the scale of the data and API rate limits. The English Wikipedia contains millions of articles, and retrieving the complete link structure of even a single article can require multiple API calls for highly connected pages. To enable reproducible experimentation without repeated network requests, the system implements a per-article disk cache using JSON files stored in a local `cache/` directory.

Each cached article is stored as a separate file (e.g., `Dog.json`) containing its list of forward links. Backlinks are cached separately with a distinct file prefix. The cache layer supports two modes of operation: online mode, in which uncached articles are fetched from the API on demand; and `cache_only` mode, in which only cached data is used. Benchmark experiments are conducted entirely in `cache_only` mode after pre-populating the cache using a dedicated `crawl.py` script.

The crawling script operates in two phases. Phase 1 fetches all seed articles from the benchmark input list individually with a 3-second delay between requests to respect API rate limits. Phase 2 fetches all uncached neighbor articles (up to one degree of separation) in batches of 50 using the batch API endpoint. This two-phase approach ensures that the entire relevant subgraph is available locally before benchmarking begins.

C. Algorithm Implementation

All three algorithms are implemented using a common abstract base class `SearchAlgorithm` that defines the `search(start, goal)` $\rightarrow$ `SearchResult` interface. The `SearchResult` dataclass standardizes the output format across algorithms, containing the following fields:

- `path`: the sequence of articles from start to goal
- `path_length`: the number of edges in the path
- `runtime`: wall-clock execution time in seconds
- `expanded_nodes`: the number of nodes whose neighbors were retrieved
- `visited_nodes`: the total number of nodes added to the visited set
- `memory_usage`: peak memory consumption in bytes, measured using `sys.getsizeof()` on internal data structures
- `found`: a boolean indicating whether a path was discovered

The BFS implementation uses Python's `collections.deque` for efficient queue operations and a dictionary for parent tracking. It includes an optimization that checks whether the goal has been reached both when a node is dequeued and during neighbor iteration, which can terminate the search one iteration earlier in some cases. Progress is printed every 25 expansions for long-running searches.

The bidirectional BFS implementation maintains two frontiers and alternates between them based on which queue is smaller. This strategy keeps the frontiers balanced and reduces the total search space. The backward search uses the Wikipedia API's `linkshere` endpoint, which retrieves articles that link to the current page, enabling true backward traversal on the directed graph. When a common node is found in both visited sets, the `merge_paths()` function reconstructs the full path by joining the forward path from the start to the meeting node with the backward path from the meeting node to the goal.

The IDDFS implementation uses a depth-limited recursive DFS with an `ancestors` set to prevent cycles. The depth limit starts at 0 and increases incrementally up to a maximum of 20. To reduce API calls, IDDFS uses a batch-fetch mechanism that retrieves the neighbors of all unvisited nodes at the current depth before proceeding to the next depth.

D. Benchmark Design

The benchmark comprises eight article pairs selected to represent varying degrees of semantic distance and connectivity within Wikipedia. The pairs are shown in Table I.

TABLE I
BENCHMARK ARTICLE PAIRS

Start	Goal	Relationship
Dog	Cat	Closely related (both domestic animals)
Cat	Brown bear	Moderately related (mammals)
Dog	Volcano	Distantly related
Cat	Triassic	Distantly related
Wolf	Biology	Related via scientific context
Dog	Time	Distantly related
Wolf	Volcano	Distantly related
Cat	Science	Moderately related (knowledge domains)

These pairs include direct connections (e.g., Dog $\rightarrow$ Cat, which are directly linked), close connections (e.g., Cat $\rightarrow$ Brown bear), and more distant connections requiring multiple hops through hub articles. This diversity ensures that the benchmark captures algorithm behavior across a range of search difficulties.

Each benchmark run is performed in `cache_only` mode to eliminate network latency from timing measurements. All three algorithms are executed on the same article pair before moving to the next pair, and each pair is run once per algorithm. The entire benchmark is automated through a single CLI command that reads the input CSV, runs all 24 combinations (3 algorithms $\times$ 8 pairs), and writes the results to a CSV file.

E. Evaluation Metrics

Four metrics are collected for each algorithm run:

- **Execution time**: measured using Python's `time.perf_counter()`, providing high-resolution wall-clock timing. All measurements exclude API call time since the cache is pre-populated.
- **Expanded nodes**: the number of nodes whose outgoing neighbors (or incoming predecessors for backward search) were retrieved. This measures the actual work performed by the algorithm.
- **Memory consumption**: measured using `sys.getsizeof()` on all active data structures, including visited sets, parent dictionaries, and queues. While this does not capture the full Python object overhead, it provides a consistent basis for comparison across algorithms.
- **Path length**: the number of edges in the discovered path, reflecting the optimality of each algorithm.

IV. RESULTS AND DISCUSSION

All three algorithms successfully found a path for every benchmark pair. Table II summarizes the average performance metrics across all eight article pairs.

TABLE II
AVERAGE PERFORMANCE METRICS

Algorithm	Time (s)	Exp. Nodes	Mem. (KB)	Length
BFS	0.0079	2,024	1.13	3.125
BiBFS	0.0019	3	61.9	3.125
IDDFS	0.0336	64,260	165.2	3.125

A. Runtime Analysis

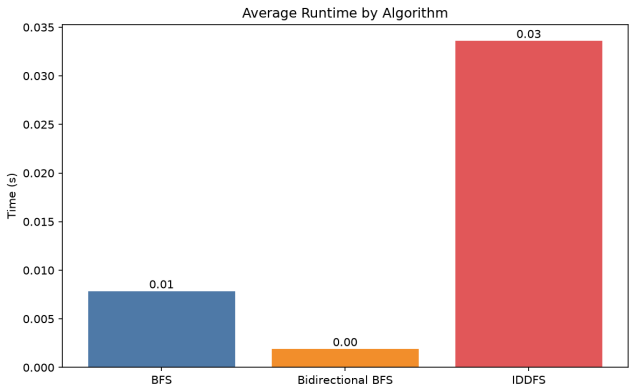

Fig. 1. Average execution time comparison across algorithms

Figure 1 shows the average execution time for each algorithm. Bidirectional BFS achieved the lowest average runtime at 0.001915 seconds, approximately four times faster than BFS (0.007861 seconds) and seventeen times faster than IDDFS (0.033615 seconds). This performance advantage is most pronounced for pairs requiring longer paths. For example, on the Wolf $\rightarrow$ Biology pair, BiBFS completed in 0.000225 seconds while BFS took 0.011850 seconds and IDDFS took 0.073387 seconds.

The superior runtime of BiBFS stems directly from the reduced search space. By simultaneously expanding from both ends, the algorithm meets in the middle after each direction has explored only half the depth of the full path. In a graph with the branching factor of Wikipedia, this represents an exponential reduction in the number of nodes that must be processed.

BFS demonstrated consistent intermediate performance. Its linear exploration pattern ensures predictable behavior, with runtime scaling proportionally to the number of articles that must be examined to find the target. For directly connected pairs (e.g., Dog $\rightarrow$ Cat), BFS terminated almost instantly. For pairs requiring multiple hops through densely connected hub articles, BFS explored thousands of nodes before finding the target.

IDDFS exhibited the highest runtime, particularly for pairs requiring longer paths. The worst case was Dog $\rightarrow$ Time, where IDDFS required 0.087203 seconds compared to 0.020306 seconds for BFS and 0.000370 seconds for BiBFS. This performance penalty is a direct consequence of IDDFS's iterative deepening mechanism: nodes at shallow depths are repeatedly re-explored at each iteration as the depth limit increases.

B. Node Expansion Analysis

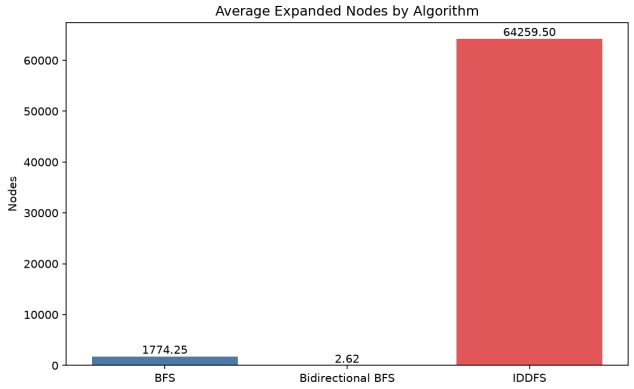

Fig. 2. Average number of expanded nodes

Figure 2 reveals the most dramatic difference among the three algorithms. Bidirectional BFS expanded an average of only 2.6 nodes per search. This remarkably low number reflects the efficiency of meeting in the middle: on most benchmark pairs, the two frontiers intersected almost immediately. For directly connected pairs, BiBFS expanded only

1 node before the forward and backward frontiers met. Even for the most distant pair (Dog → Time), BiBFS expanded only 5 nodes.

BFS expanded an average of 2,024 nodes. The number of expansions varied significantly by pair: directly connected articles required only 1 expansion, while distant pairs such as Dog → Time required 5,776 expansions and Wolf → Biology required 2,554. This variation reflects the density of Wikipedia's link structure around different topic areas.

IDDFS expanded an average of 64,260 nodes, over 30 times more than BFS and nearly 25,000 times more than BiBFS. The worst cases were Dog → Time with 161,740 expansions and Wolf → Biology with 147,414 expansions. These figures represent the cumulative expansion across all depth-limited iterations. Notably, IDDFS expanded zero nodes for directly connected pairs (Dog → Cat, Cat → Brown bear, Cat → Triassic), because the goal was found within the first iteration before any expansion occurred.

The extreme difference in node expansions between IDDFS and the other algorithms highlights the primary weakness of iterative deepening on high-branching-factor graphs: the cost of repeated re-exploration at each depth level.

C. Memory Usage Analysis

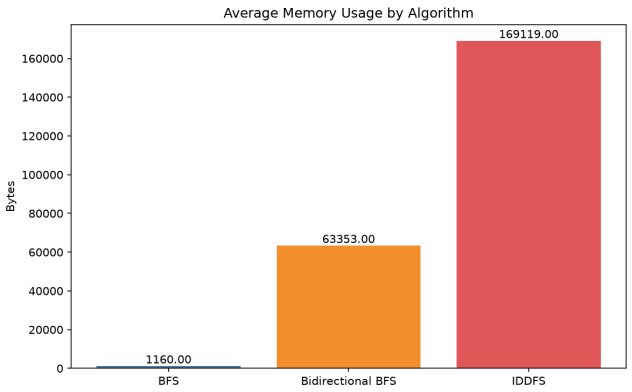

Fig. 3. Average memory consumption in bytes

Figure 3 shows the average memory consumption. BFS exhibited the lowest and most consistent memory usage at exactly 1,160 bytes across all benchmark pairs. This consistency stems from the fact that BFS's memory overhead is dominated by the data structures for the visited set and parent dictionary, which grow proportionally with the search frontier.

Bidirectional BFS showed higher and more variable memory usage, averaging 63,353 bytes. The variance is substantial: directly connected pairs used only 2,320 bytes, while distant pairs such as Dog → Time required 170,592 bytes and Wolf → Biology required 44,912 bytes. The higher memory usage reflects the need to maintain two separate visited sets, two parent dictionaries, and two frontier queues. For some pairs, the backward search explored a large region before finding a connection, driving up memory consumption.

IDDFS exhibited the highest average memory usage at 169,119 bytes, with significant variance. The Dog → Time pair used 415,624 bytes, and Cat → Science also used 415,624 bytes. This high memory usage appears counterintuitive given IDDFS's theoretical space advantage. However, the implementation stores cumulative visited sets and ancestor tracking across depth iterations, and the parent dictionaries for path reconstruction grow with each iteration. Additionally, the Python recursion stack and data structure overhead contribute to the measured memory footprint.

D. Path Length Analysis

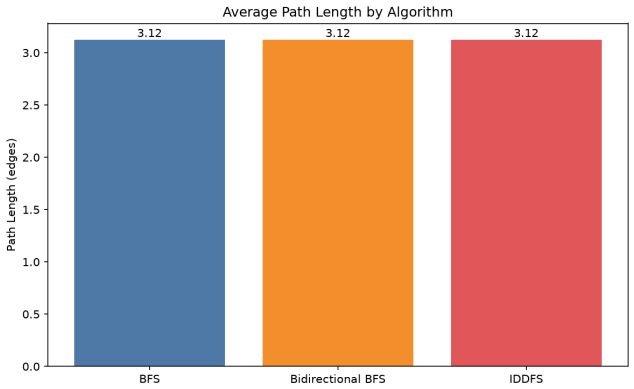

Fig. 4. Average path length in edges

Figure 4 demonstrates that all three algorithms found paths of identical average length (3.125 edges) for each benchmark pair. This result confirms the theoretical expectation that all three algorithms are optimal for unweighted graphs. BFS and BiBFS are inherently optimal due to their level-by-level expansion. IDDFS, by incrementally increasing the depth limit, also guarantees finding a shortest path.

The path lengths ranged from 2 edges (for directly connected pairs such as Dog → Cat) to 4 edges (for pairs requiring multiple hops, such as Dog → Time and Wolf → Biology). The relatively short path lengths across all pairs confirm the small-world property of the Wikipedia hyperlink graph.

E. Discussion

The experimental results reveal clear trade-offs among the three algorithms. Bidirectional BFS emerges as the best overall performer for Wikipedia navigation, offering the fastest runtime and fewest node expansions by a substantial margin. Its main drawback is elevated and unpredictable memory usage, which could become problematic for extremely large or deep searches.

BFS offers a useful middle ground: predictable memory usage, reasonable runtime, and guaranteed optimality. For applications where memory constraints are the primary concern and the graph is not excessively deep, BFS remains a solid choice.

IDDFS, despite its theoretical space advantage, performed poorly on the Wikipedia hyperlink graph due to the high branching factor. The cumulative cost of repeated exploration at each depth iteration resulted in the longest runtimes and the most node expansions by a wide margin. However, IDDFS may still be valuable in environments with severe memory constraints, where storing the BFS frontier would be infeasible.

The small-world structure of Wikipedia had a visible impact on algorithm performance. The presence of hub articles with high connectivity meant that most search paths could be found by navigating through a small number of intermediate nodes. This property particularly benefited BiBFS, whose bidirectional approach efficiently converged on these hub nodes from both ends.

V. CONCLUSION

This paper presents a comparative analysis of three classical graph search algorithms for navigating Wikipedia's hyperlink network. By modeling Wikipedia as a directed graph and implementing BFS, Bidirectional BFS, and IDDFS within a consistent experimental framework, the study evaluates each algorithm's performance across eight real-world navigation tasks.

The experimental results demonstrate that Bidirectional BFS is the most efficient algorithm for this domain, achieving the fastest average runtime (0.001915 seconds) and the fewest node expansions (average 2.6) while maintaining path optimality. BFS offers predictable memory consumption and reasonable performance, making it a reliable alternative. IDDFS, while theoretically appealing for its space efficiency, suffers from excessive node re-exploration on Wikipedia's high-branching-factor graph, resulting in the longest runtimes and the most expansions.

The findings confirm that the small-world properties of Wikipedia's hyperlink graph significantly influence search algorithm performance, with highly connected hub articles serving as natural convergence points that accelerate bidirectional search. The study also validates that all three algorithms are optimal with respect to path length on the unweighted Wikipedia graph.

VI. SUGGESTION

Future work may extend this study in several directions. First, incorporating informed search algorithms such as Greedy Best-First Search or A* using domain-specific heuristics could potentially improve search efficiency further. Possible heuristics include the semantic similarity of article titles, the overlap of link sets between pages, or the presence of shared categories.

Second, the effects of graph preprocessing techniques such as link filtering (e.g., removing navigation and administrative links), graph compression, or the construction of a weighted graph based on link relevance could improve both search speed and path quality.

Third, a larger-scale evaluation using hundreds or thousands of random article pairs would provide more statistically robust conclusions. Such an evaluation would also enable analysis of how factors such as Wikipedia topic categories, article popularity, and graph density affect algorithm performance.

Finally, implementing parallel or distributed versions of these algorithms could leverage modern multi-core processors to further reduce search times on Wikipedia-scale graphs.

APPENDIX

The GitHub repository for this paper and all source code can be accessed at: https://github.com/philipphqiwu/Makalah-IF2211-Wikipedia-Navigation

ACKNOWLEDGMENT

The author expresses his heartfelt gratitude to Prof. Dr. Ir. Rinaldi, M.T., his lecturer for Algorithm Strategy at Bandung Institute of Technology, for his comprehensive teaching which laid down the groundwork for understanding the concepts applied in this paper.

The author would also like to thank his friends and family for all the support they have shown throughout his journey as a student at ITB.

Lastly, the author would like to show his appreciation to Jalen Brunson, OG Anunoby, Karl-Anthony Towns, Mikal Bridges, Josh Hart and the 2025-2026 New York Knicks for inspiring the author with their triumph.

REFERENCES

[1] Munir, Rinaldi. 2026. "Penentuan rute (Route/Path Planning) - Bagian 1". *Department of Informatics, Institut Teknologi Bandung, 2025-2026*, [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/21-Route-Planning-(2026)-Bagian1.pdf. [Accessed: Jun. 19, 2026].

[2] A. Spoerri, "What is popular on Wikipedia and why?," *First Monday*, vol. 12, no. 4, 2007. [Online]. Available: https://doi.org/10.5210/fm.v12i4.1766. [Accessed: Jun. 19, 2026].

[3] V. Zlatić, M. Božičević, H. Štefančić, and M. Domazet, "Wikipedias: Collaborative web-based encyclopedias as complex networks," *Physical Review E*, vol. 74, no. 1, 016115, 2006. [Online]. Available: https://doi.org/10.1103/PhysRevE.74.016115. [Accessed: Jun. 19, 2026].

[4] A. P. Masucci, A. Kalampokis, V. M. Eguíluz, and E. Hernández-García, "Wikipedia information flow analysis reveals the scale-free architecture of the semantic network," *Physical Review E*, vol. 82, no. 1, 016108, 2010. [Online]. Available: https://doi.org/10.1103/PhysRevE.82.016108. [Accessed: Jun. 19, 2026].

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: MIT Press, 2022. [Online]. Available: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/. [Accessed: Jun. 19, 2026].

[6] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985. [Online]. Available: https://doi.org/10.1016/0004-3702(85)90084-0. [Accessed: Jun. 19, 2026].

[7] S. Milgram, "The small world problem," *Psychology Today*, vol. 1, no. 1, pp. 61–67, 1967.

[8] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ: Pearson, 2021. [Online]. Available: https://www.pearson.com/en-us/subject-catalog/p/artificial-intelligence-a-modern-approach/P200000003500. [Accessed: Jun. 19, 2026].

STATEMENT

Hereby, I declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not a product of plagiarism.

Bandung, 19 Juni 2026

Philipp Hamara
13524101