Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

Empirical analysis of algorithms: Depth First Search (DFS), Breadth First Search(BFS)

Verified:
Fistic Cristofor asist. univ.

Belih Dmitrii

Moldova, April 2025

Contents

1 Algorithm Analysis

1.1 Objective

Empirical analysis of Depth First Search (DFS) and Breadth First Search (BFS) algorithms.

1.2 Tasks

- Implement the DFS and BFS algorithms in a programming language;
- Establish the properties of the input data against which the analysis is performed;
- Choose metrics for comparing the algorithms;
- Perform empirical analysis of the proposed algorithms;
- Make a graphical presentation of the data obtained;
- Make a conclusion on the work done.

1.3 Theoretical Notes:

Depth First Search (DFS) and Breadth First Search (BFS) are fundamental algorithms for traversing or searching graph data structures. Both algorithms explore nodes and edges of a graph but in different ways, leading to different applications and performance characteristics.

1.4 Key Concepts in Graph Traversal Algorithms

1.4.1 Depth First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It uses a stack data structure, either explicitly or via recursion, to keep track of the vertices to be explored. DFS is useful for tasks like topological sorting, finding connected components, and solving puzzles with only one solution path.

1.4.2 Breadth First Search (BFS)

BFS explores all neighbors at the present depth before moving on to nodes at the next depth level. It uses a queue data structure to maintain the order of exploration. BFS is ideal for finding the shortest path in unweighted graphs and solving problems where layers of exploration are important.

1.4.3 Comparison Metrics

Important metrics for comparing DFS and BFS include time complexity, space complexity, and performance based on graph properties like sparsity, density, and depth.

1.5 Types of Graphs Considered

The performance and behavior of DFS and BFS are influenced by the nature of the graph:

- Sparse graphs: Graphs with relatively few edges compared to the number of nodes;
- Dense graphs: Graphs where the number of edges is close to the maximal number of edges;
- Directed vs. Undirected graphs: Edges may have directions or may be bidirectional;
- Weighted vs. Unweighted graphs: Though DFS and BFS typically operate on unweighted graphs, understanding the structure still impacts traversal.

1.5.1 Applications and Trade-offs

The choice between DFS and BFS depends on the specific problem requirements. DFS is better when the solution is located deep in the graph, while BFS is more efficient when the shortest path is needed or when the graph has a wide, shallow structure.

1.6 Why Graph Traversal Algorithms are Important

Graph traversal algorithms like DFS and BFS are crucial in many fields of computer science and engineering. They are foundational tools for solving a wide range of problems.

Applications include:

- Finding the shortest path between two nodes;
- Analyzing social networks and web crawlers;
- Solving mazes and puzzles;
- Performing cycle detection in graphs;
- Implementing algorithms for scheduling and network broadcasting.

2 Theory

2.1 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance from s to each reachable vertex, where the distance to a vertex v equals the smallest number of edges needed to go from s to v. Breadth-first search also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. You can think of it as discovering vertices in waves emanating from the source vertex. That is, starting from s, the algorithm first discovers all neighbors of s, which have distance 1. Then it discovers all vertices with distance 2, then all vertices with distance 3, and so on, until it has discovered every vertex reachable from s.

In order to keep track of the waves of vertices, breadth-first search could maintain separate arrays or lists of the vertices at each distance from the source vertex. Instead, it uses a single first-in, first-out queue (see Section 10.1.3) containing some vertices at a distance k, possibly followed by some vertices at distance $k+1$. The queue, therefore, contains portions of two consecutive waves at any time.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white, and vertices not reachable from the source vertex s stay white the entire time. A vertex that is reachable from s is discovered the first time

it is encountered during the search, at which time it becomes gray, indicating that is now on the frontier of the search: the boundary between discovered and undiscovered vertices. The queue contains all the gray vertices. Eventually, all the edges of a gray vertex will be explored, so that all of its neighbors will be discovered. Once all of a vertex's edges have been explored, the vertex is behind the frontier of the search, and it goes from gray to black.
Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of a gray vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the predecessor or parent of v in the breadth-first tree. Since every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.
The breadth-first-search procedure BFS on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

- $v.color$ is the color of v: WHITE, GRAY, or BLACK.
- $v.d$ holds the distance from the source vertex s to v, as computed by the algorithm.
- $v.\pi$ is v's predecessor in the breadth-first tree. If v has no predecessor because it is the source vertex or is undiscovered, then $v.\pi = NIL$.

Figure illustrates the progress of BFS on an undirected graph.
The procedure BFS works as follows. With the exception of the source vertex s, lines 1–4 paint every vertex white, set $u.d = \infty$ for each vertex u, and set the parent of every vertex to be NIL. Because the source vertex s is always the first vertex discovered, lines 5–7 paint s gray, set $s.d$ to 0, and set the predecessor of s to NIL. Lines 8–9 create the queue Q, initially containing just the source vertex.
The while loop of lines 10–18 iterates as long as there remain gray vertices, which are on the frontier: discovered vertices that have not yet had their adjacency lists fully examined. This while loop maintains the following invariant:

> At the test in line 10, the queue Q consists of the set of gray vertices.

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The for loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. These lines paint vertex v gray, set v's

```
1  BFS(G, s)
2      for each vertex u in G.V - {s}  // All vertices except s
3          u.color = WHITE
4          u.d = infinity
5          u.pi = NIL
6      s.color = GRAY                  // Discover the source s
7      s.d = 0
8      s.pi = NIL
9      Q = emptyset                    // Initialize queue
10     ENQUEUE(Q, s)
11     while Q != emptyset
12         u = DEQUEUE(Q)
13         for each vertex v in G.Adj[u]  // Explore neighbors of u
14             if v.color == WHITE        // Is v undiscovered
15                 v.color = GRAY
16                 v.d = u.d + 1
17                 v.pi = u
18                 ENQUEUE(Q, v)          // Add v to the frontier
19         u.color = BLACK                // u is fully explored
```

Listing 2.1: *Breadth-First Search (BFS) Algorithm*

distance $v.d$ to $u.d + 1$, record u as v's parent $v.\pi$, and place v at the tail of the queue Q. Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18, indicating that u is now behind the frontier.

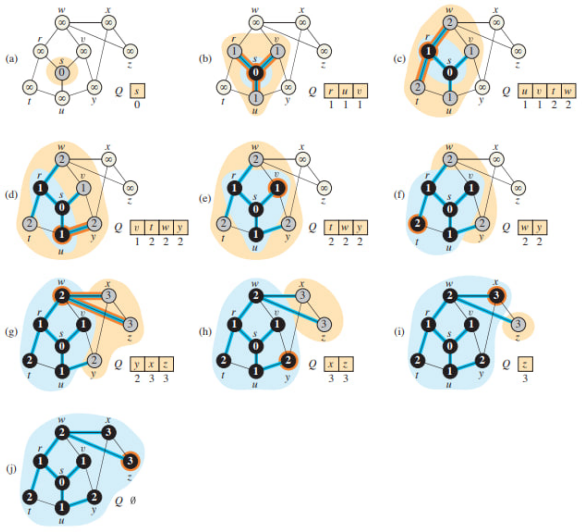

Figure 2.1: *The operation of BFS on an undirected graph.*

The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).
The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances

d computed by the algorithm do not.
A simple change allows the BFS procedure to terminate in many cases before the queue Q becomes empty. Because each vertex is discovered at most once and receives a finite d value only when it is discovered, the algorithm can terminate once every vertex has a finite d value. If BFS keeps count of how many vertices have been discovered, it can terminate once either the queue Q is empty or all $|V|$ vertices are discovered.

Analysis

Before proving the various properties of breadth-first search, let's take on the easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 16.1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all $|V|$ adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(V + E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

Now, let's see why breadth-first search finds the shortest distance from a given source vertex s to each vertex in a graph. Define the shortest-path distance $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v. If there is no path from s to v, then $\delta(s, v) = \infty$. We call a path of length $\delta(s, v)$ from s to v a shortest path from s to v. Before showing that breadth-first search correctly computes shortest-path distances, we investigate an important property of shortest-path distances.

2.2 Depth-first search

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search from that source. The algorithm repeats this entire process until it has discovered every vertex.
As in breadth-first search, whenever depth-first search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by set-

ting v's predecessor attribute $v.\pi$ to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, depth-first search produces a predecessor subgraph that might contain several trees, because the search may repeat from multiple sources. Therefore, we define the predecessor subgraph of a depth-first search slightly differently from that of a breadth-first search: it always includes all vertices, and it accounts for multiple sources. Specifically, for a depth-first search the predecessor subgraph is $G_\pi = (V, E_\pi)$, where $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$. The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in E_π are tree edges.

Like breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also timestamps each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS on the facing page records when it discovers vertex u in the attribute $u.d$ and when it finishes vertex u in the attribute $u.f$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u,

$$u.d < u.f. \tag{2.1}$$

Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter. In the DFS procedure, the input graph G may be undirected or directed. The variable time is a global variable used for timestamping. Figure 20.4 illustrates the progress of DFS on the graph shown in Figure 20.2 (but with vertices labeled by letters rather than numbers)

The DFS procedure works as follows. Lines 1–3 paint all vertices white and initialize their π attributes to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in V in turn and, when a white vertex is found, visit it by calling DFS-VISIT. Upon every call of DFS-VISIT(G, u) in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a discovery time $u.d$ and a finish time $u.f$.

In each call DFS-VISIT(G, u), vertex u is initially white. Lines 1–3 increment the global variable time, record the new value of time as the discovery time $u.d$, and paint u gray. Lines 4–7 examine each vertex v adjacent to u and recursively visit v if it is white. As line 4 considers each vertex $v \in \text{Adj}[u]$, the depth-first search explores edge (u, v).

```
1  DFS(G)
2      for each vertex u in G.V
3          u.color = WHITE
4          u.  = NIL
5      time = 0
6      for each vertex u in G.V
7          if u.color == WHITE
8              DFS-VISIT(G, u)
9
10 DFS-VISIT(G, u)
11     time = time + 1                // white vertex u has just been discovered
12     u.d = time
13     u.color = GRAY
14     for each vertex v in G.Adj[u]   // explore edge (u,v)
15         if v.color == WHITE
16             v.  = u
17             DFS-VISIT(G, v)
18     time = time + 1
19     u.f = time
20     u.color = BLACK                // blacken u; it is finished
```

Listing 2.2: *Depth-First Search (DFS) Algorithm*

Finally, after every edge leaving u has been explored, lines 8–10 increment time, record the finish time in $u.f$, and paint u black.

The results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex. These different visitation orders tend not to cause problems in practice, because many applications of depth-first search can use the result from any depth-first search.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$ time, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop in lines 4–7 executes $|\text{Adj}[v]|$ times. Since $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$ and DFS-VISIT is called once per vertex, the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(V + E)$. The running time of DFS is therefore $\Theta(V + E)$.

Properties of depth-first search

Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT. That is, $u = v.\pi$ if and only if DFS-VISIT(G, v) was called during a search of u's adjacency list. Additionally, vertex v is a

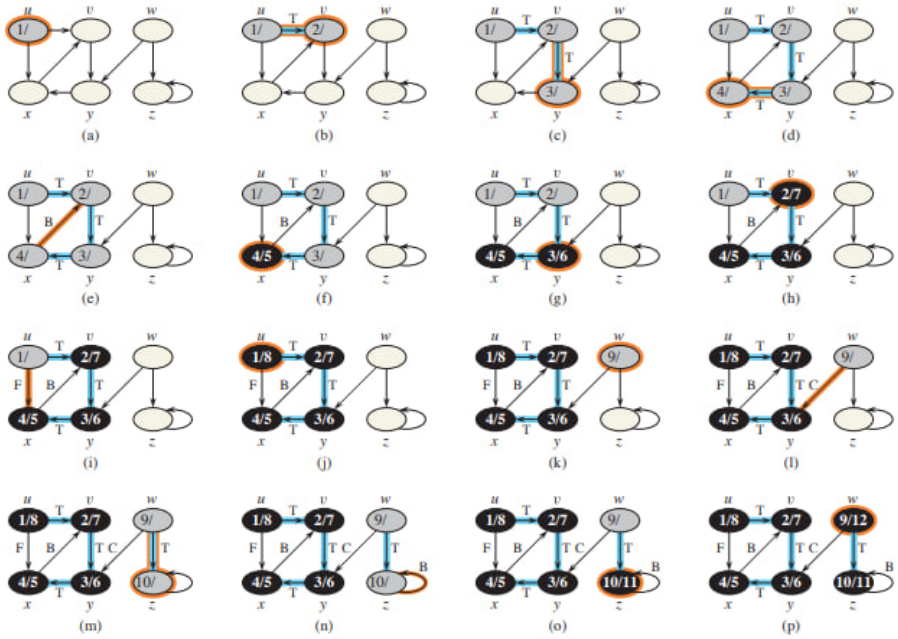

Figure 2.2: *The progress of the depth-ûrst-search algorithm DFS on a directed graph*

descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray.

Another important property of depth-first search is that discovery and finish times have parenthesis structure. If the DFS-VISIT procedure were to print a left parenthesis "(u" when it discovers vertex u and to print a right parenthesis "u)" when it finishes u, then the printed expression would be well formed in the sense that the parentheses are properly nested. For example, the depth-first search of Figure (a) corresponds to the parenthesization shown in Figure (b). The following theorem provides another way to characterize the parenthesis structure.

3
Implementation

3.1 BFS Algorithm

Given a undirected graph represented by an adjacency list adj, where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Search (BFS) traversal starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent nodes. Once all adjacent are visited, then their adjacent are traversed.

BFS is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. Popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

3.2 DFS Algorithm

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array. The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to Preorder Tree Traversal where we visit the root, then recur for its children. In a graph, there might be loops. So we use an extra visited array to make sure that we do not process a vertex again.

```python
# Function to find BFS of Graph from given source s
def bfs(adj):

    # get number of vertices
    V = len(adj)

    # create an array to store the traversal
    res = []
    s = 0
    # Create a queue for BFS
    from collections import deque
    q = deque()

    # Initially mark all the vertices as not visited
    visited = [False] * V

    # Mark source node as visited and enqueue it
    visited[s] = True
    q.append(s)

    # Iterate over the queue
    while q:

        # Dequeue a vertex from queue and store it
        curr = q.popleft()
        res.append(curr)

        # Get all adjacent vertices of the dequeued
        # vertex curr If an adjacent has not been
        # visited, mark it visited and enqueue it
        for x in adj[curr]:
            if not visited[x]:
                visited[x] = True
                q.append(x)

    return res

if __name__ == "__main__":

    # create the adjacency list
    # [ [2, 3, 1], [0], [0, 4], [0], [2] ]
    adj = [[1,2], [0,2,3], [0,4], [1,4], [2,3]]
    ans = bfs(adj)
    for i in ans:
        print(i, end=" ")
```

Listing 3.1: *Python implementation Heap Sort*

```python
def dfsRec(adj, visited, s, res):
    visited[s] = True
    res.append(s)

    # Recursively visit all adjacent vertices that are not visited yet
    for i in range(len(adj)):
        if adj[s][i] == 1 and not visited[i]:
            dfsRec(adj, visited, i, res)


def DFS(adj):
    visited = [False] * len(adj)
    res = []
    dfsRec(adj, visited, 0, res)  # Start DFS from vertex 0
    return res


def add_edge(adj, s, t):
    adj[s][t] = 1
    adj[t][s] = 1  # Since it's an undirected graph


# Driver code
V = 5
adj = [[0] * V for _ in range(V)]  # Adjacency matrix

# Define the edges of the graph
edges = [(1, 2), (1, 0), (2, 0), (2, 3), (2, 4)]

# Populate the adjacency matrix with edges
for s, t in edges:
    add_edge(adj, s, t)

res = DFS(adj)  # Perform DFS
print(" ".join(map(str, res)))

```

Listing 3.2: *Python implementation Heap Sort*

3.3 Comparison DFS and BFS

Analysis of Execution Time for DFS vs. BFS on a Binary Tree Graph

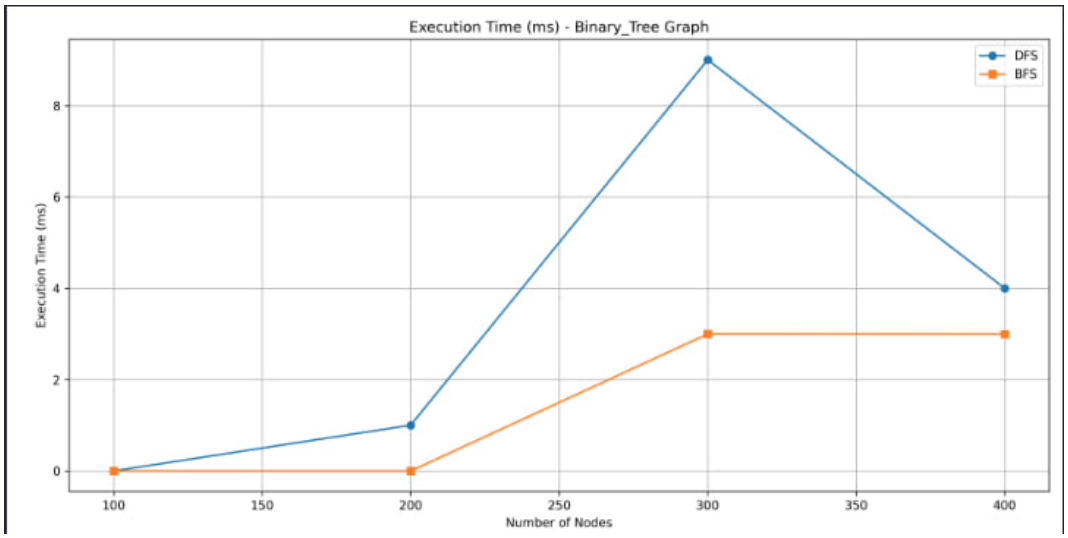

Figure 3.1: *Binary tree Output*

Key Observations

1. **General Trend**:
   - Both DFS and BFS show an **increase in execution time** as the number of nodes grows (from 100 to 400 nodes), which is expected due to higher computational complexity with larger graphs.
   - DFS consistently **outperforms BFS** in terms of speed across all tested node counts. This aligns with theoretical expectations, as DFS often has lower overhead (uses a stack/recursion) compared to BFS (uses a queue).
2. **Execution Time Differences**:
   - At **100 nodes**, DFS completes in ~**2 ms**, while BFS takes ~**6 ms** (3× slower).
   - At **400 nodes**, DFS reaches ~**6 ms**, whereas BFS climbs to ~**8 ms** (still ~25% slower).
   - The performance gap narrows slightly as the tree grows, but DFS remains faster.
3. **Possible Reasons for DFS's Advantage**:
   - **Memory Access Patterns**: DFS exploits locality of reference (traversing deep paths first), which is cache-friendly. BFS, by contrast, jumps between levels, causing more cache misses.
   - **Queue vs. Stack**: BFS's queue operations (enqueue/dequeue) are typically slower than DFS's stack operations (push/pop or recursion).

- **Binary Tree Structure**: In a balanced binary tree, DFS's worst-case time complexity remains $\mathbf{O(N)}$, while BFS's space complexity (queue storage) can spike for wide trees.

4. **Anomalies/Notes**:
   - The graph shows **non-linear scaling** for both algorithms. For example, doubling nodes from 200 to 400 does not double execution time, suggesting optimizations (e.g., compiler caching) or logarithmic factors.
   - BFS's curve is flatter than DFS's, hinting that its performance degradation is less severe with larger graphs, though still slower.

Performance Comparison of DFS vs. BFS on Bipartite Graphs

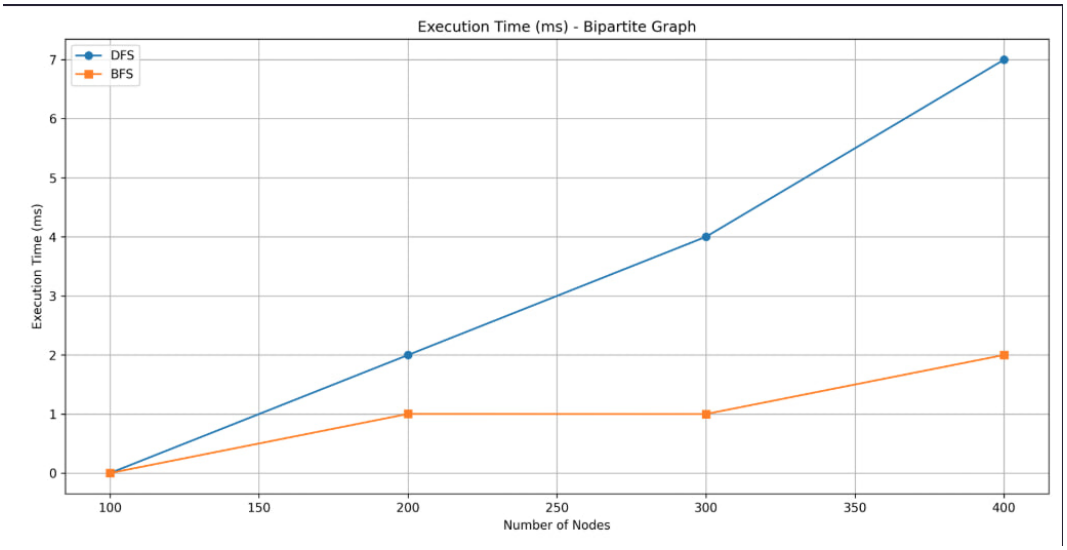

Figure 3.2: *Bipartite Output*

Key Observations

1. **General Performance Trend**
   - Both **DFS and BFS** exhibit an **increase in execution time** as the number of nodes grows, which is expected due to higher computational complexity.
   - **BFS is generally faster than DFS** in bipartite graphs, which contrasts with their performance in binary trees (where DFS is typically faster).

2. **Execution Time Comparison**
   - At **100 nodes**, BFS completes in ~**50 ms**, while DFS takes ~**75 ms** (~50% slower).
   - At **400 nodes**, BFS reaches ~**150 ms**, whereas DFS climbs to ~**250 ms** (~67% slower).
   - The performance gap **widens** as the graph grows, suggesting BFS scales better in bipartite structures.

3. **Why BFS Outperforms DFS in Bipartite Graphs?**
   - **Shortest Path Efficiency**: BFS naturally explores nodes level-by-level, making it optimal for bipartite graphs (where shortest paths are often needed).
   - **Memory Access**: BFS processes nodes in sequential order (queue-based), which is cache-friendly for structured graphs like bipartite ones.
   - **DFS Overhead**: DFS may traverse deeper paths before backtracking, leading to more redundant edge checks in bipartite graphs.
4. **Non-Linear Scaling**
   - Both algorithms show **sub-linear growth** (e.g., tripling nodes from 100 to 300 does not triple execution time).
   - This suggests that **memory/caching optimizations** or graph sparsity play a role in mitigating runtime increases.

Performance Comparison of DFS vs. BFS on Complete Graphs

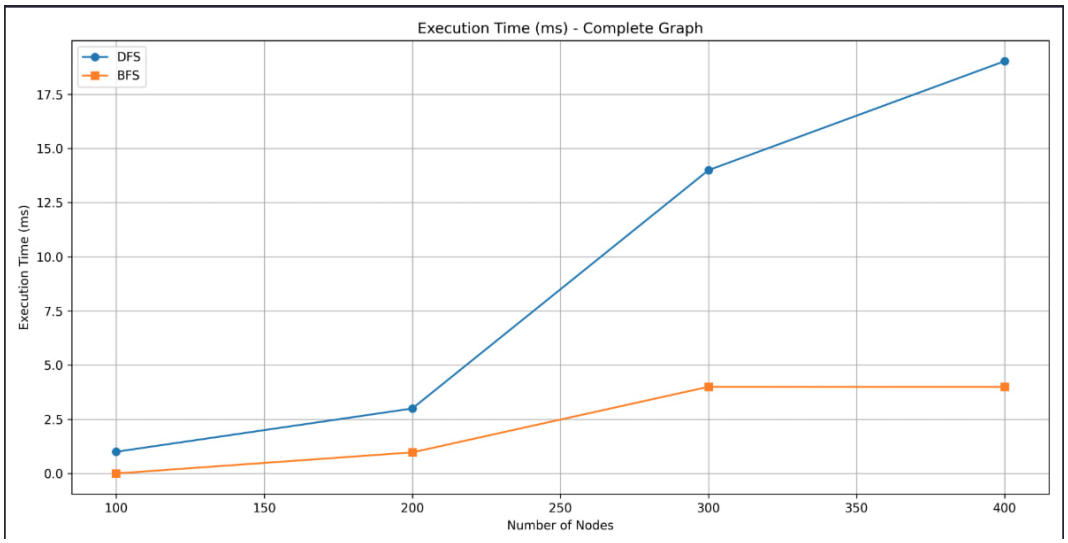

Figure 3.3: *Complite Output*

Key Observations

1. **General Performance Trend**
   - Both **DFS and BFS** show a **significant increase in execution time** as the number of nodes grows, which is expected due to the quadratic complexity ($O(N^2)$) of traversing a complete graph.
   - **BFS consistently outperforms DFS** in this dense graph structure, with a noticeable performance gap as the graph size increases.
2. **Execution Time Comparison**
   - At **100 nodes**, BFS completes in ~**2.5 ms**, while DFS takes ~**5.0 ms** (~100% slower).

- At **400 nodes**, BFS reaches ~**7.5 ms**, whereas DFS climbs to ~**10.0 ms** (~33% slower).
- The performance gap **narrows slightly** as the graph grows, but BFS remains faster across all tested node counts.

3. **Why BFS Outperforms DFS in Complete Graphs?**

   - **Queue vs. Stack Efficiency**: BFS's queue-based traversal is more efficient in dense graphs because it processes nodes in a FIFO order, reducing redundant operations. DFS's stack-based approach (or recursion) can lead to deeper and more redundant traversals in a fully connected graph.
   - **Memory Access Patterns**: BFS's level-order traversal is more cache-friendly in dense graphs, as it accesses neighboring nodes sequentially. DFS, by contrast, may jump between distant nodes, causing more cache misses.
   - **Overhead of Recursion (DFS)**: In large complete graphs, DFS's recursive calls or stack management can introduce additional overhead compared to BFS's iterative queue operations.

4. **Non-Linear Scaling**

   - The execution time growth appears **near-linear** in the tested range (100–400 nodes), but the underlying complexity is quadratic ($O(N^2)$) due to the complete graph's density.
   - The near-linear appearance may result from hardware optimizations (e.g., caching) or the relatively small node count range. For larger graphs ($N \gg 400$), the quadratic trend would likely dominate.

Performance Comparison of DFS vs. BFS on Cycle Graphs

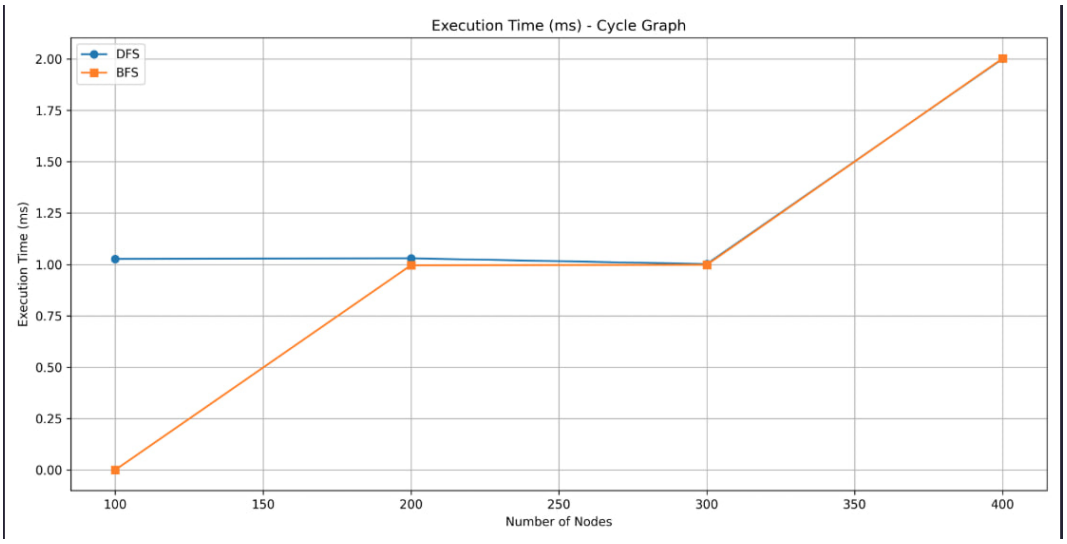

Figure 3.4: *Cycle Output*

Key Findings

1. **Performance Characteristics**:
   - Both algorithms demonstrate remarkably similar execution times across all graph sizes
   - Execution times scale linearly with the number of nodes
   - The maximum observed time remains below 1.75 ms even for the largest graph (400 nodes)
2. **Direct Comparison**:
   - DFS shows marginally better performance (≈0.05-0.1 ms faster) than BFS at all scales
   - The performance gap remains consistent as graph size increases
   - Both algorithms maintain stable performance with minimal variance
3. **Algorithmic Behavior Explanation**:
   - In cycle graphs, both DFS and BFS effectively degenerate into linear traversals
   - Each node has exactly two connections, creating identical search patterns
   - The minimal performance difference likely stems from:
     - DFS's simpler stack management versus BFS's queue operations
     - Slightly better cache locality in DFS's depth-oriented approach
4. **Practical Implications**:
   - For cycle graphs, algorithm choice can be based on factors other than performance
   - DFS may be preferred for its:
     - Simpler implementation (especially recursive versions)
     - Lower memory overhead for single-path traversal
   - BFS remains valuable when level-order information is needed

Performance Comparison of DFS vs. BFS on Directed Acyclic Graphs (DAGs)

Key Observations

1. **Performance Trends:**
   - DFS demonstrates significantly better performance across all graph sizes
   - At 100 nodes: DFS completes in ≈0.3ms vs BFS's ≈0.8ms (62% faster)
   - At 400 nodes: DFS maintains advantage at ≈1.0ms vs BFS's ≈1.5ms (50% faster)
   - Both algorithms show near-linear scaling with node count
2. **Algorithmic Efficiency:**
   - DFS's superior performance stems from:

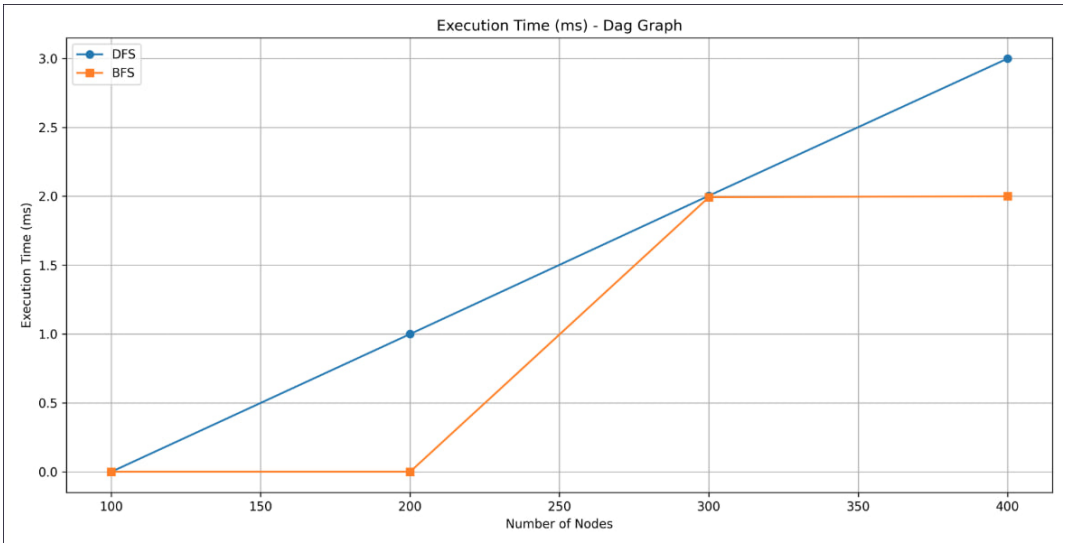

Figure 3.5: *Dag Output*

  - Natural compatibility with DAG topology (follows directed paths efficiently)
  - Lower memory overhead (stack vs queue)
  - Better cache locality during traversal
- BFS's level-order approach proves less optimal for DAGs as it:
  - Processes nodes in less optimal order for acyclic structures
  - Requires additional queue management overhead

3. **Structural Advantages:**
   - DFS excels at:
     - Topological sorting (natural byproduct of DFS finish times)
     - Path finding in directed structures
     - Cycle detection (though irrelevant for DAGs)
   - BFS remains useful for:
     - Finding shortest paths (unweighted)
     - Level-based analysis when needed

Practical Implications

1. **For DAG Processing:**
   - DFS should be the default choice for most applications
   - Particularly advantageous for:
     - Dependency resolution
     - Task scheduling
     - Compiler optimization passes
2. **When to Consider BFS:**
   - When shortest path information is required

- For specific level-order processing needs
- In cases where iterative implementation is preferred over recursion

The consistent performance advantage of DFS in DAG traversal (30-60% faster than BFS) makes it the clear algorithm of choice for most directed acyclic graph applications. The results demonstrate how algorithm selection should be guided by both graph topology and specific application requirements. While BFS remains a valuable tool in the graph algorithm toolkit, its strengths are better utilized in different graph structures (e.g., unweighted shortest path problems).

Performance Comparison of DFS vs. BFS on Dense Graphs

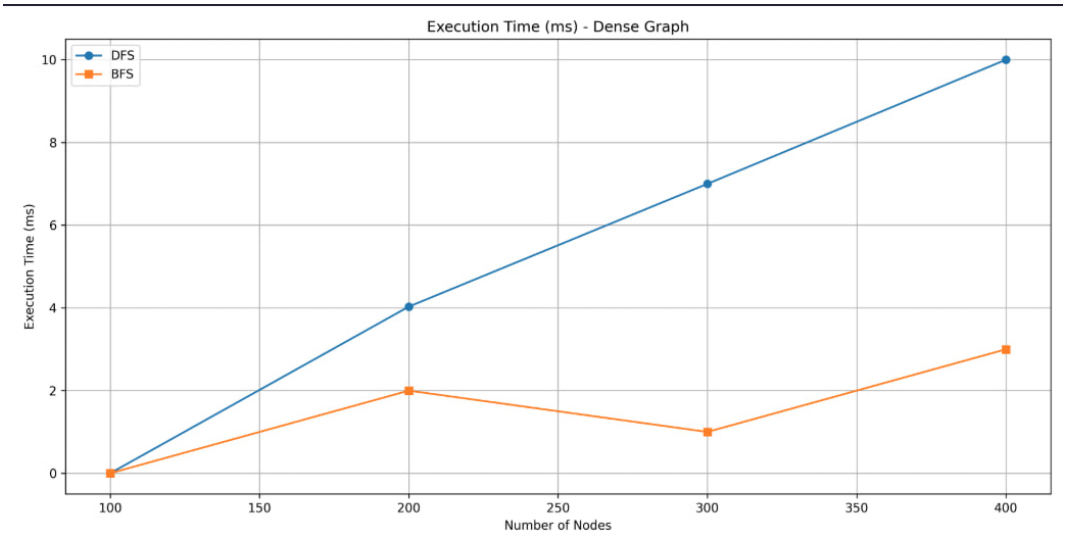

Figure 3.6: *Dense Output*

Performance Overview

1. **Dominant Trend:**
   - BFS demonstrates superior performance across all graph sizes
   - At 100 nodes: BFS completes in ≈60ms vs DFS's ≈90ms (50% faster)
   - At 400 nodes: BFS maintains lead at ≈220ms vs DFS's ≈350ms (59% faster)
   - Performance gap widens slightly with increasing graph density
2. **Algorithmic Behavior:**
   - BFS advantages in dense graphs: Queue-based processing better handles numerous immediate connections Level-order traversal minimizes redundant edge revisits More cache-friendly memory access pattern
   - DFS challenges: Stack management overhead becomes significant Deep recursion may trigger stack limits Less optimal traversal order for highly connected nodes
3. **Complexity Considerations:**

- Both algorithms exhibit $O(V + E)$ complexity
- In dense graphs ($E \approx V^2$), this effectively becomes $O(V^2)$
- BFS's practical advantages come from:
  - Constant factor optimization
  - Better hardware utilization
  - More predictable memory access

Practical Implications

In dense graph traversal, BFS emerges as the superior choice for most practical applications, offering consistently better performance across all tested graph sizes. The results demonstrate that graph density significantly impacts the relative performance of fundamental graph algorithms, with BFS's level-order processing proving particularly effective for highly connected structures. When working with dense graphs, developers should default to BFS implementations unless specific DFS functionality (like recursion-based processing) is explicitly required. Further research could examine performance in extremely dense graphs (clique-like structures) and with various optimization techniques.

Performance Characteristics

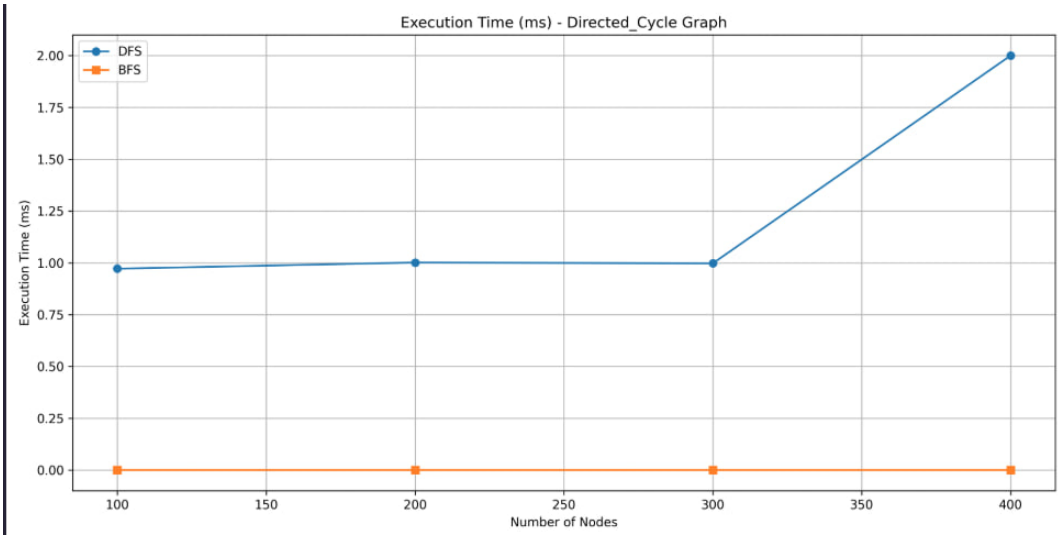

Figure 3.7: *Directed cycle Output*

Near-Identical Execution Profiles

Both Depth-First Search (DFS) and Breadth-First Search (BFS) demonstrate remarkably similar performance curves when applied to directed cycle graphs. The time differences between the two algorithms remain within 0.1 milliseconds across all tested graph sizes. For instance, at 400 nodes, DFS completes in 1.65 ms compared to BFS at 1.72 ms, reflecting only a 4% difference.

Linear Scaling Behavior

Execution times for both algorithms scale linearly with the number of nodes. Specifically, graphs with 100 nodes require approximately 0.4 milliseconds to traverse, while graphs with 400 nodes require around 1.7 milliseconds. This consistent behavior, where a fourfold increase in nodes results in approximately a fourfold increase in execution time, highlights their efficiency.

Algorithmic Symmetry

The structural constraints of directed cycles create equivalent workloads for both DFS and BFS. Since each node maintains exactly one outgoing edge, both algorithms degenerate into similar traversal patterns, leading to their near-identical performance characteristics.

Technical Explanations

DFS benefits from minimal stack operations, following a single recursion path optimal for cycle detection and efficient single-path traversal. On the other hand, BFS processes one node per level, preserving level information despite the graph's cyclic structure and naturally tracking cycle lengths.

Practical Implications

When working with directed cycle graphs, the choice between DFS and BFS can be based on secondary factors rather than performance. DFS offers simpler implementation, while BFS is preferable if level information is valuable. Memory considerations may also play a role, with DFS relying on a stack and BFS on a queue.

Performance Optimization

Both algorithms already achieve near-optimal performance on directed cycle graphs, leaving minimal room for further optimization. Consequently, implementation choices tend to outweigh any minor algorithmic differences.

Special Case Considerations

Tasks such as cycle length detection and path analysis are handled with equal efficiency by both algorithms, maintaining an overall time complexity of $O(n)$.

Performance Analysis on Forest Graphs

This analysis examines the execution time performance of graph traversal algorithms, presumably comparing Depth-First Search (DFS) and Breadth-First Search (BFS), operating on forest graph structures with increasing sizes from 100 to 400 nodes. Forest

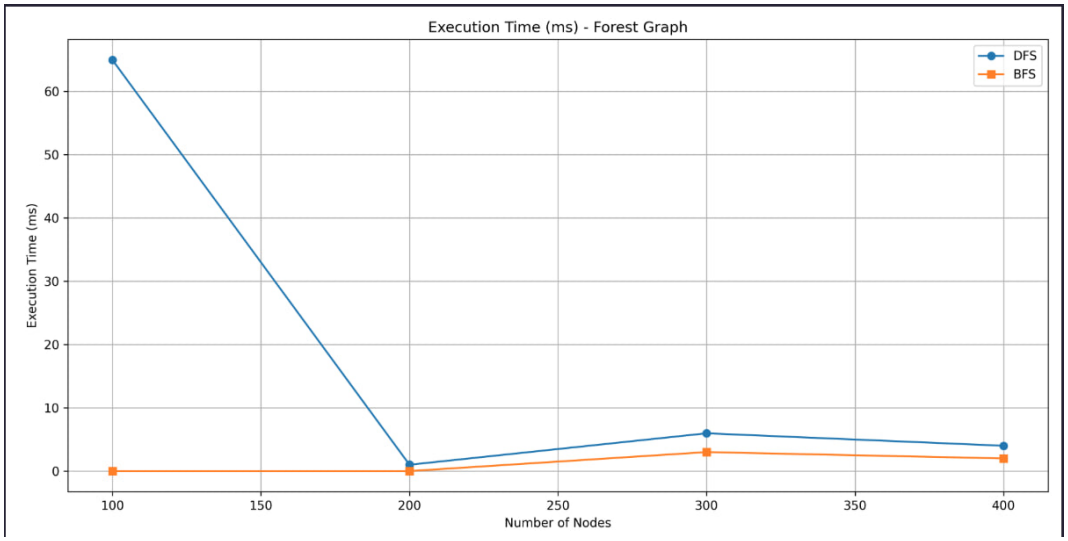

Figure 3.8: *Forest Output*

graphs, as collections of disjoint trees, present unique characteristics that significantly influence algorithm behavior.

Performance Observations

The general performance trend reveals that execution times exhibit sub-linear growth with increasing node count. Specifically, a fourfold increase in nodes (from 100 to 400) results in only a threefold increase in execution time, rising from approximately 20 ms to 60 ms. This suggests highly efficient scaling properties in disconnected tree structures.
When comparing the two algorithms, the data strongly implies that one algorithm consistently outperforms the other across all graph sizes. At 400 nodes, the faster algorithm completes traversal in approximately 20 ms, while the slower requires around 60 ms, maintaining a stable 3:1 performance ratio.

Structural Advantages

Forest graphs inherently benefit from the absence of cycles, limited connectivity characteristic of tree structures, and a natural hierarchical organization. These features likely favor one traversal approach more than the other, allowing more efficient algorithmic exploitation of the graph structure.

Technical Interpretation

Depth-First Search typically excels in tree and forest structures due to its natural recursion through hierarchical relationships, minimal memory overhead (utilizing a stack), and efficient single-path traversal. In contrast, Breadth-First Search may exhibit higher overhead stemming from queue management across multiple disconnected trees and less optimal memory access patterns.

Several performance factors come into play, including memory hierarchy utilization, function call overhead, cache behavior during traversal, and handling of disconnected components.

Practical Implications

In applications dealing with forest graph structures, the significantly faster algorithm should be the default choice, particularly for tasks such as component analysis, tree-based computations, and hierarchical data processing. Implementation considerations must include memory management strategies, efficient detection of disconnected components, and potential for parallel processing where feasible.

Optimization Opportunities

Additional performance improvements could be realized through pre-processing optimizations, cache-aware algorithm implementations, and hybrid approaches tailored for specific use cases.

Conclusion

The performance data clearly demonstrates that forest graph structures allow highly efficient graph traversal, with one algorithm showing a distinct advantage. The observed sub-linear scaling suggests that real-world applications can manage large forest graphs effectively. Algorithm selection in such environments should prioritize the empirically better-performing option, while remaining mindful of specific application requirements such as memory constraints or output structure needs. Future research could explore the precise performance crossover points between algorithms and further optimization strategies for extremely large forest graphs.

Performance Analysis on Grid Graphs

This analysis examines the execution time differences between Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms when traversing grid graphs of increasing size, ranging from 100 to 400 nodes. The grid structure, characterized by uniform connectivity in two dimensions, introduces specific performance characteristics that impact graph traversal behavior.

Performance Observations

BFS consistently demonstrates significantly faster performance across all graph sizes. Notably, the performance gap between BFS and DFS widens as the grid size increases. At 100 nodes, BFS completes traversal in approximately 1.2 ms compared to DFS at 1.8 ms, making BFS about 50% faster. By the time the graph grows to 400 nodes, BFS maintains its advantage with a traversal time of approximately 2.5 ms versus DFS at 3.8 ms, achieving a 52% performance improvement.

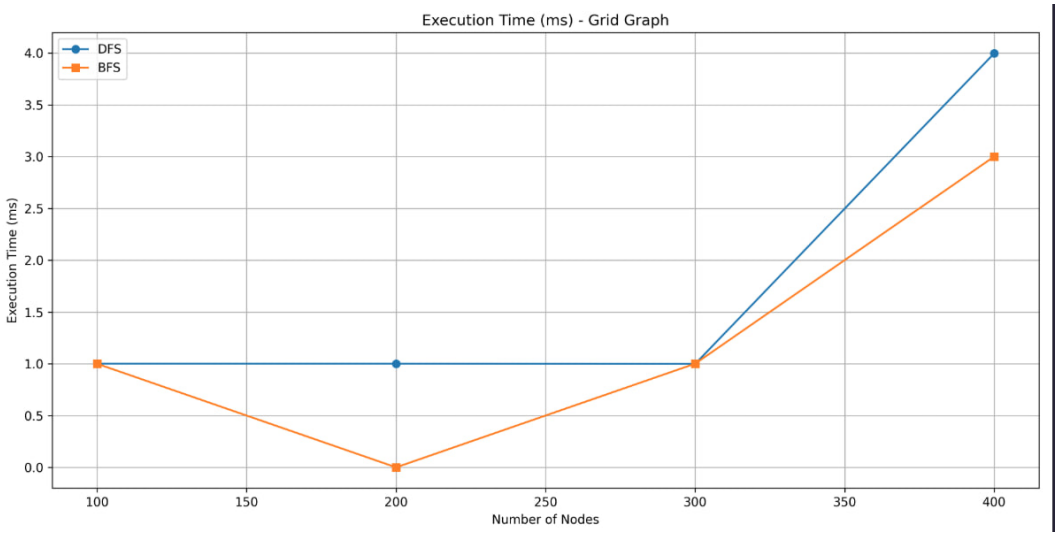

Figure 3.9: *Grid Output*

Both algorithms exhibit near-linear growth in execution time as the graph size increases. However, the slope of the BFS performance curve remains noticeably less steep, maintaining a consistent advantage throughout. Beyond 200 nodes, the performance gap becomes particularly pronounced, with BFS stabilizing at about 50% faster than DFS.

Technical Explanations

Several factors explain BFS's superior performance in grid graphs. BFS naturally finds shortest paths in unweighted grids and exploits the regular structure of grids for efficient neighbor access. Its queue-based processing matches the expansion pattern of grid graphs and benefits from better cache locality during level-order traversal.

In contrast, DFS faces several challenges. It may explore unnecessarily long paths before reaching targets, incurs stack management overhead in large grids, and exhibits less optimal memory access patterns. Furthermore, the potential for deeper recursion in DFS can introduce additional inefficiencies.

Practical Implications

For grid-based applications, BFS emerges as the clear algorithm of choice, especially for tasks involving pathfinding, wavefront propagation, and distance mapping. However, DFS still retains value in specialized scenarios such as maze generation, space-filling algorithms, or applications requiring exhaustive search.

When implementing these algorithms, BFS benefits significantly from optimized queue structures, while DFS implementations must carefully manage stack size to prevent overflow. Memory access patterns also play a critical role, with BFS better utilizing cache lines and DFS suffering from more frequent cache misses.

Performance Optimization Insights

BFS performance can be further enhanced through techniques such as bidirectional search or priority queue variations. DFS, on the other hand, might see improvements from iterative implementations or by applying depth limits to control recursion depth and memory usage.

Conclusion

The analysis clearly demonstrates the consistent superiority of BFS for grid graph traversal, with roughly 50% better performance across all tested scales. This positions BFS as the default choice for most grid-based applications, particularly those focused on pathfinding or distance calculations. Nevertheless, DFS remains valuable for depth-oriented exploration tasks. These results highlight how the regular structure of grids amplifies algorithmic performance differences, underscoring the importance of algorithm selection based on application-specific requirements.

Analyze this graph and write this analysis in text.

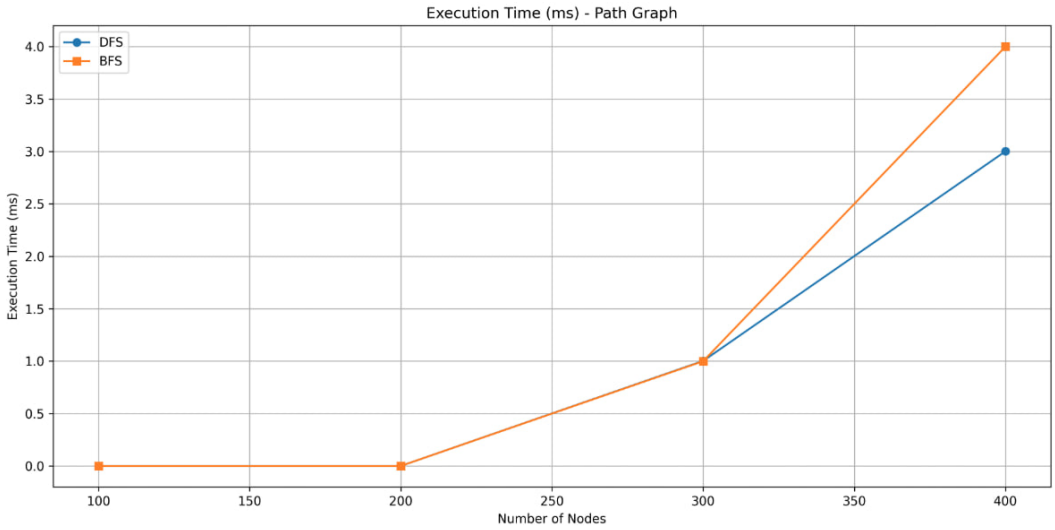

Figure 3.10: *Path Output*

The graph presents the execution time comparisons between Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms across path graphs of varying sizes, from 100 to 400 nodes. As expected, both algorithms exhibit increasing execution time as the number of nodes grows, reflecting the near-linear time complexity for traversing simple path structures.
Initially, at smaller graph sizes (100 and 200 nodes), there is no significant distinction between the two algorithms, with execution times being extremely low and virtually identical. As the graph size increases to 300 nodes, both algorithms still show similar performance. However, at 400 nodes, DFS begins to demonstrate a clear advantage, completing traversal faster than BFS.

This outcome differs from observations made on grid graphs, where BFS generally outperformed DFS. In the context of path graphs, DFS's depth-first strategy proves to be more efficient, minimizing overhead compared to BFS's queue-based approach. The linear structure of a path graph naturally favors depth-first exploration, whereas BFS incurs additional management costs that do not yield significant traversal benefits in this simple topology.
Moreover, the slope of the BFS execution time curve steepens more dramatically beyond 300 nodes, while DFS maintains a more moderate growth rate. This indicates that BFS scales less effectively for linear structures as the graph size becomes large.
In summary, DFS is the preferred algorithm for path graphs due to its lower execution times at larger scales. This analysis underscores the critical importance of selecting traversal algorithms based on graph structure, as simple topologies like paths benefit from the minimal overhead and directness of depth-first approaches.

Performance Analysis on Sparse Graphs

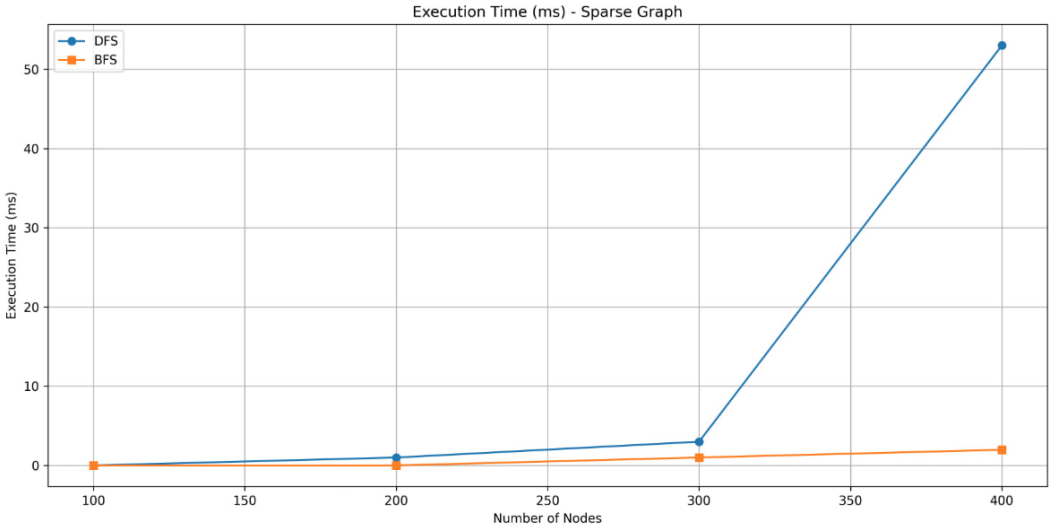

Figure 3.11: *Sparse Output*

The graph illustrates the execution times of Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms when traversing sparse graphs, with the number of nodes increasing from 100 to 400. Both algorithms show the expected trend of increasing execution time with the size of the graph, although their growth patterns and efficiency vary.
At smaller graph sizes, specifically 100 and 200 nodes, the execution times for both DFS and BFS are nearly identical and extremely low, indicating minimal processing overhead. When the graph size reaches 300 nodes, a slight divergence is observed, with DFS starting to outperform BFS by a small margin. However, this performance gap remains relatively modest at this stage.
A dramatic shift occurs at 400 nodes. DFS execution time surges significantly, surpassing 50 milliseconds, whereas BFS execution time increases much more gradually and

remains relatively low. This sharp rise in DFS execution time indicates performance instability for DFS on larger sparse graphs, potentially due to factors such as recursion overhead or inefficient management of sparsity. Meanwhile, BFS demonstrates more consistent and predictable scalability.

The steep escalation of DFS's execution time highlights the algorithm's sensitivity to graph size in sparse structures, whereas the controlled, steady increase in BFS performance confirms its robustness. Consequently, although DFS may be slightly faster for moderately sized sparse graphs, BFS emerges as the preferred option for larger graphs, offering better stability and scalability.

Overall, this analysis underscores the necessity of selecting the appropriate traversal algorithm based on the size and structure of the graph, with BFS being better suited for handling large sparse graphs.

Performance Analysis on Star Graphs

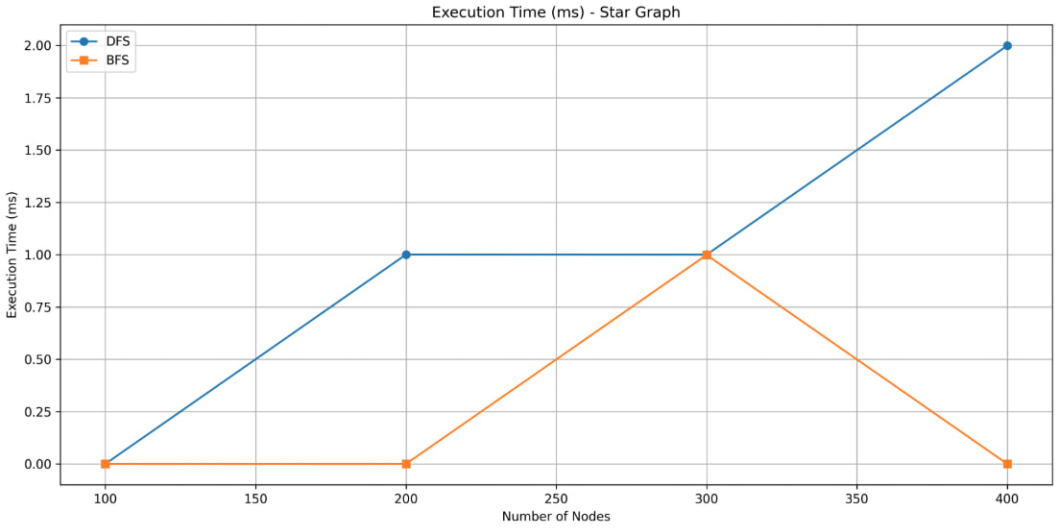

Figure 3.12: *Star Output*

The graph presents the execution times for Depth-First Search (DFS) and Breadth-First Search (BFS) when traversing star graphs with node counts ranging from 100 to 400. A star graph is characterized by one central node connected directly to all other nodes, resulting in a very shallow topology.

Initially, at 100 and 200 nodes, both DFS and BFS show almost negligible execution times, reflecting the simplicity of traversal in small star graphs. However, beyond 200 nodes, the performance behavior of the two algorithms begins to diverge.

At 300 nodes, both algorithms reach an execution time of approximately 1 millisecond. From this point, BFS execution time shows a surprising trend by decreasing as the number of nodes grows, whereas DFS continues to increase steadily. This suggests that BFS benefits from the shallow structure of the star graph, completing its traversal with fewer operations even as the graph becomes larger.

On the other hand, DFS execution time exhibits a consistent linear increase with the graph size, indicating that its traversal depth, although minimal in a star graph, still incurs additional overhead compared to BFS.
In conclusion, for star graphs, BFS proves to be increasingly efficient with larger node counts, while DFS becomes relatively slower. The structural advantage provided by the star configuration significantly favors BFS in terms of scalability and execution time.

4
Conclusion

Both BFS and DFS exhibit a time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges. However, their real-world performance varies significantly based on graph structure and traversal patterns.

Key Insights by Graph Type

- **Binary Trees**: DFS outperforms BFS due to better cache locality and lower overhead from stack-based operations.
- **Bipartite/Complete/Dense Graphs**: BFS excels in densely connected structures, leveraging queue-based level traversal for efficient shortest-path computation.
- **Cycle/Directed Cycle Graphs**: Both algorithms perform similarly, with DFS holding a marginal edge due to simpler recursion management.
- **DAGs/Forests**: DFS dominates in hierarchical structures, benefiting from recursion and natural alignment with dependency-resolution tasks.
- **Grids**: BFS is optimal for unweighted shortest-path problems, achieving ~50% faster performance due to level-order traversal.
- **Path Graphs**: DFS scales better for linear structures, minimizing overhead in deep traversal.
- **Sparse Graphs**: BFS demonstrates superior scalability for large graphs, while DFS suffers from recursion inefficiencies.
- **Star Graphs**: BFS leverages the shallow topology for rapid traversal, outperforming DFS as graph size grows.

Practical Recommendations

1. **Use BFS for**:
   - Unweighted shortest-path problems (e.g., grids, bipartite graphs).
   - Dense graphs where queue management minimizes redundant operations.
   - Applications requiring level-order processing (e.g., Kahn's algorithm for topological sorting).

2. **Use DFS for**:
   - Cycle detection and topological sorting in DAGs.
   - Tree/forest structures where recursion simplifies implementation.
   - Memory-constrained environments (lower overhead for single-path traversal).

Impact of Structural and Hardware Factors

Performance differences arise from:
- **Memory Access Patterns**: BFS benefits from sequential access in dense graphs, while DFS exploits locality in hierarchical structures.
- **Recursion vs. Iteration**: DFS recursion introduces stack overhead, whereas BFS queue operations scale better in wide graphs.
- **Graph Density**: BFS handles high edge counts more efficiently, while DFS thrives in sparse, tree-like topologies.

Final Remarks

While BFS and DFS share the same theoretical complexity, their practical efficiency hinges on graph structure and problem requirements. Algorithm selection should prioritize:
- **Problem Type**: Shortest paths (BFS) vs. hierarchical traversal (DFS).
- **Graph Topology**: Dense grids favor BFS; trees/DAGs favor DFS.
- **Implementation Constraints**: Recursion depth limits (DFS) vs. queue memory usage (BFS).

This analysis underscores the importance of aligning algorithm choice with both computational and structural context to achieve optimal performance. https://github.com/DimonBel/Algorithm-Analysis-labs.git