Graph Algorithm Analysis and Visualization Report

1 Depth-First Search (DFS)

1.1 Introduction to Depth-First Search algorithm

Depth-First Search (DFS) is a fundamental graph traversal algorithm that explores as far as possible along a branch before backtracking. Introduced in the early 20th century as a theoretical tool for solving maze-like problems, DFS has since become a cornerstone of computer science and mathematics. Its systematic approach to exploring graphs underpins numerous applications, from topological sorting and cycle detection to AI search strategies and pathfinding algorithms. The iterative version of DFS, which uses a stack instead of recursion, emerged as an alternative to avoid the pitfalls of stack overflow in large or deeply nested graphs.

The algorithm's historical significance is tied to its utility in analyzing the structure of networks, solving puzzles, and optimizing paths. Its simplicity and elegance make it an essential topic in both theoretical and applied graph theory, and its visualization can provide intuitive insights into graph behavior, connectivity, and traversal dynamics.

1.2 DFS Algorithm

Below is the pseudocode for the DFS algorithm, followed by a detailed explanation of its components and their functions.

- **Initialization:** Each vertex u is initialized with a color (WHITE), representing that it has not yet been visited. The predecessor pointer $u.\pi$ is set to NIL to indicate no parent. This corresponds to setting up the necessary data structures before traversal in the Python code.
- **Time Variable:** The variable *time* is initialized to track discovery and finishing times of each vertex. While not explicitly implemented in the

Algorithm 1 DepthFirstSearch(G)

```
for each vertex u ∈ G do
    u.color ← WHITE
    u.π ← NIL
end for
time ← 0
for each vertex u ∈ G do
    if u.color = WHITE then
        DFS-Visit(G, u)
    end if
end for=0
```

Python class, this is an important feature for analyzing the traversal order.

- **Outer Loop:** The algorithm iterates through all vertices in the graph. If a vertex is still WHITE, the DFS-Visit function is called. This ensures that all connected components of the graph are traversed.

1.3 DFS-Visit Algorithm

Algorithm 2 DFS-Visit(G, u)

```
time ← time + 1
u.d ← time
u.color ← GRAY
for each vertex v ∈ Adj[u] do
    if v.color = WHITE then
        v.π ← u
        DFS-Visit(G, v)
    end if
end for
time ← time + 1
u.f ← time
u.color ← BLACK =0
```

- **Discovery:** When a vertex u is first visited, its discovery time $u.d$ is set, and its color is changed to GRAY, indicating it is currently being explored.

- **Recursion:** For each adjacent vertex v, if v is WHITE, the algorithm sets its predecessor to u and recursively calls DFS-Visit. This is analogous to the stack mechanism used in the Python implementation.
- **Finishing Time:** After all adjacent vertices are explored, $u.f$ is set to the finishing time, and the vertex is marked BLACK, signifying that it is fully processed.

1.4 Relationship Between `DFS` and `DFS-Visit`

The `DFS` function serves as the **controller** of the traversal process, managing the initialization and ensuring complete coverage of the graph. It delegates the task of exploring individual vertices and their neighbors to the `DFS-Visit` function. The `DFS-Visit` function performs the **core traversal logic**, handling discovery, exploration, and backtracking for connected components.

This hierarchical relationship ensures that:

- `DFS` manages the global traversal process across the graph.
- `DFS-Visit` handles the detailed local traversal of each connected component.

Together, they provide a complete implementation of the Depth-First Search algorithm.

The relationship between the `DFS` function and the `DFS-Visit` function can be summarized as follows:

1. `DFS` **Function:**
    - The `DFS` function acts as the entry point for the traversal process.
    - It initializes each vertex in the graph by marking them as unvisited (color = WHITE) and sets their parent pointers to `NIL`.
    - It iterates through all vertices in the graph, invoking `DFS-Visit` for any vertex still marked as unvisited (color = WHITE).
    - Its primary responsibility is to ensure all connected components of the graph are traversed, including handling disconnected graphs.
2. `DFS-Visit` **Function:**
    - The `DFS-Visit` function performs the recursive traversal of the graph starting from a given vertex.

- It marks the vertex as discovered (color = GRAY) and assigns discovery time.
- It explores all adjacent vertices, recursively calling itself for any unvisited neighbors.
- Once all adjacent vertices have been explored, it marks the vertex as fully processed (color = BLACK) and assigns a finishing time.

1.5 Cycle Handling

The Depth-First Search (DFS) algorithm is capable of efficiently handling graphs with cycles. Below, we explain how the algorithm processes cycles and avoids infinite loops:

1. **Cycle Detection:** When a graph contains cycles, the DFS algorithm may encounter a vertex that has already been visited. The algorithm detects this by maintaining a `visited` set, which tracks all vertices that have already been processed. If a vertex is found in the `visited` set during traversal, it indicates a cycle, and the vertex is skipped.

2. **Handling Repeated Nodes:** The `visited` set ensures that each vertex is processed only once during the traversal. When a previously visited neighbor is encountered, the algorithm skips further exploration of that vertex. This mechanism prevents redundant processing and ensures the traversal does not revisit nodes unnecessarily.

3. **Traversal Order:** The presence of cycles does not affect the traversal order of DFS. The algorithm continues to explore other branches of the graph after encountering a cycle. It ensures that all reachable vertices are processed exactly once by skipping vertices already marked as visited.

By leveraging the `visited` set, DFS can safely and efficiently traverse graphs with cycles, avoiding infinite loops and redundant work.

1.6 Comparison to Python Implementation

While the pseudocode uses recursion, the Python implementation uses an iterative stack-based approach.

1.6.1 Traversal Logic

- A stack is used to track the current node and its neighbors.
- Each node is marked as discovered (gray) when first visited and fully explored (black) once all its neighbors are processed.
- Discovery and finish times are recorded for each node, allowing for detailed analysis of the traversal.

1.6.2 Performance

The algorithm has a time complexity of $O(V + E)$, where V is the number of vertices and E the number of edges. The use of an adjacency list ensures efficient neighbor lookup and edge traversal.

1.7 Visualization Design and Implementation

The visualization aims to illustrate the dynamic processes of DFS, including discovery, exploration, and backtracking. This was achieved through the following steps:

- Colors represent the state of each node (white, gray, or black), updated dynamically during traversal.
- The progression of the algorithm is captured frame-by-frame. Matplotlib is used to generate and save each frame of the traversal process, showcasing the algorithm's internal states.
- Each frame reflects the current stack state, node colors, and edges being traversed.
- Frames are compiled into a video using FFmpeg, creating a smooth animation that visually conveys the algorithm's execution.

2 The Knight's Shortest Path Problem

The second problem analyzed in this report involves finding the shortest path for a knight on a chessboard. The knight moves in an L-shape: two squares horizontally or vertically and one square in the perpendicular direction. Given the starting square (x, y) and the destination square (i, j) on an 8×8 chessboard, the task is to calculate the minimum number of moves required for the knight to reach its destination.

Problem Representation

The problem can be represented as a graph where:

- Each square of the chessboard is a vertex.
- An edge exists between two vertices if the knight can move directly from one square to the other.

The graph is undirected and sparse, with at most 8 neighbors for each vertex due to the knight's movement constraints.

Algorithmic Approach

The horizontal search algorithm (breadth-first search, BFS) is used to solve this problem efficiently. BFS is well-suited for unweighted graphs and guarantees the shortest path in terms of the number of edges. The steps are as follows:

1. Initialize all vertices as unvisited.
2. Start from the source vertex (x, y), mark it as visited, and enqueue it.
3. While the queue is not empty:
    - Dequeue a vertex and examine its neighbors.
    - If a neighbor has not been visited, mark it as visited, enqueue it, and record its distance from the source.
    - If the destination vertex (i, j) is reached, terminate the search and return the distance.

Implementation and Complexity

The BFS algorithm was implemented using a queue to manage vertices. The adjacency list representation of the graph ensures efficient traversal and neighbor lookup. The time complexity is $O(V + E)$, where V is the number of vertices (64 for an 8×8 board) and E the number of edges.

This visualization provides an intuitive understanding of how BFS explores the chessboard and finds the shortest path.

Conclusion

This report examined two graph problems: the depth-first search traversal and the knight's shortest path problem. Through iterative implementation and visualization, we demonstrated the practical application of DFS in graph traversal and BFS in shortest path calculation.