CS-204: Design and Analysis of Algorithms

220001022, 220001023, 220001024

February 7, 2024

1 Graph Traversal Algorithms

Graph traversal algorithms are methods used to systematically visit and explore all vertices and edges of a graph, enabling the examination of its structure and properties. Depending on the structure of the graph, we decide which algorithm is optimal to traverse the graph.

1.1 Depth-First Search (DFS)

1.1.1 Pseudocode

Algorithm 1 DFS

Input: Graph G with vertices V and edges E, starting vertex s
Output: Depth-first traversal of G starting from s
Procedure: DFS(G, s)
Mark s as visited
for each vertex v **in** $G.adj[s]$ **do**
    if v is not visited **then**
        DFS(G, v)
    end if
end for

1.1.2 Time Complexity Analysis

The time complexity of DFS is different depending on the graph representation used:

- **Adjacency Matrix**: In the case of an adjacency matrix, the time complexity of DFS is $O(V^2)$, where V is the number of vertices. This is because we need to check all possible edges for each vertex.

- **Adjacency List**: When using an adjacency list, the time complexity of DFS is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because, for each vertex, we need to traverse its adjacency list, which takes $O(\text{degree}(v))$ time, where $\text{degree}(v)$ is the number of edges incident to vertex v. The sum of all degrees in the graph is $2E$, so the total time complexity is $O(V + 2E) = O(V + E)$.

In the worst-case scenario where every vertex is connected to every other vertex, $E = V^2$, and therefore $O(V+E)$ becomes $O(V^2)$.

1.1.3 Space Complexity Analysis

The space complexity of DFS is $O(V + V)$ in case of explicit graph, where V is for visited array used and another V is the worst case length of recursion stack in case of skewed graph. In case of implicit graph, space complexity is $O(bd)$ where b is the branching factor and d is the depth of the graph.

1.2 Breadth-First Search (BFS)

1.2.1 Pseudocode

Algorithm 2 BFS

```
Input: Graph G with vertices V and edges E, starting vertex s
Output: Breadth-first traversal of G starting from s
Procedure: BFS(G, s)
Initialize an empty queue Q
Mark s as visited and enqueue s into Q
while Q is not empty do
    Dequeue a vertex v from Q
    for each vertex w in G.adj[v] do
        if w is not visited then
            Mark w as visited and enqueue w into Q
        end if
    end for
end while
```

1.2.2 Time Complexity Analysis

The time complexity of BFS is different depending on the graph representation used:

- **Adjacency Matrix**: In the case of an adjacency matrix, the time complexity of BFS is $O(V^2)$, where V is the number of vertices. This is because we need to check all possible edges for each vertex.
- **Adjacency List**: When using an adjacency list, the time complexity of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because, for each vertex, we need to traverse its adjacency list, which takes $O(\text{degree}(v))$ time, where $\text{degree}(v)$ is the number of edges incident to vertex v. The sum of all degrees in the graph is $2E$, so the total time complexity is $O(V + 2E) = O(V + E)$.

In the worst-case scenario where every vertex is connected to every other vertex, $E = V^2$, and therefore $O(V+E)$ becomes $O(V^2)$.

1.2.3 Space Complexity Analysis

The space complexity of BFS is $O(V)$, where V is the number of vertices in the graph. This space is required for maintaining the visited array to keep track of visited vertices and for the queue used in BFS traversal. In case of implicit graph, spaace complexity is $O(b^d)$ where b is the branching factor and d is the depth of the graph.

2 Completeness of Algorithms

Completeness in algorithms means that the algorithm will always find a solution within a reasonable amount of time if at least one solution exists.For example, DFS and BFS are two fundamental graph traversal algorithms used to explore and search for nodes in a graph. While both algorithms are widely used and effective in various scenarios, they differ in their completeness. BFS is considered a complete algorithm, on the other hand, DFS is not considered a complete algorithm.

Why BFS is Complete:

- **Systematic Exploration:** BFS explores the graph systematically, visiting nodes in a level-by-level manner. It ensures that all nodes at each level are visited before moving deeper into the graph.
- **Shortest Path Property:** BFS discovers the shortest path from the starting node to any reachable node. Since it explores level by level, the first occurrence of a node guarantees the shortest path to that node.

Why DFS is Not Complete:

- **Unbounded Exploration:** In the case of implicit graphs, DFS may continue indefinitely along an unexplored branch where the destination could be located.

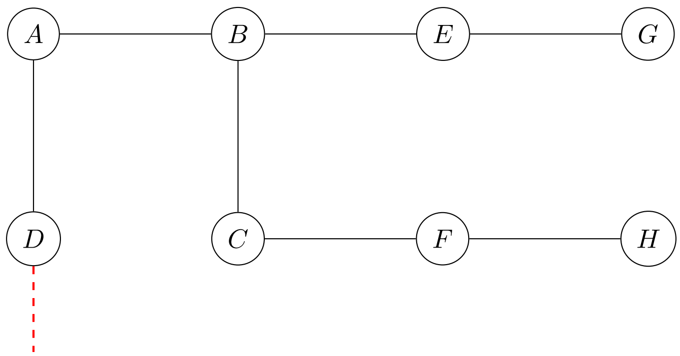


Figure 1: DFS continuing indefinitely

  Here, suppose node A is the start node and destination is node H, but the DFS might explore the branch D indefinitely in case of implict graph.

- **Lack of Shortest Path Guarantee:** DFS does not guarantee the discovery of the shortest path between the starting node and any other node. Due to its nature of exploring one branch deeply, it may find a longer path before discovering a shorter one.

3 Connectivity of Graphs

A graph can be classified based on its connectivity.

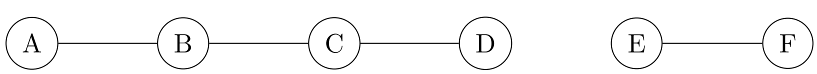
3.1 Connected Graphs

A connected graph is a graph in which there exists a path between every pair of vertices. In other words, there are no isolated vertices, and all vertices are reachable from every other vertex.

The graph above is an example of a connected graph. There is a path between every pair of vertices (A, B, C, and D), making it a connected graph.

3.2 Disconnected Graphs

A disconnected graph is a graph in which there are two or more disjoint sets of vertices, with no path between them.



The graph above is an example of a disconnected graph. There are two disjoint sets of vertices (A, B, C, and D) and (E, F), with no path between them.

3.2.1 Algorithm for Finding Connected Components

To find the number of connected components in a disconnected graph, we can use a DFS or BFS algorithm to traverse the graph and count the number of separate connected regions.

Algorithm 3 Count Connected Components

```
function CountComponents(G)
    visited ← Empty set
    count ← 0
    for all v in G do
        if v is not visited then
            DFS_Visit(G, v) {or BFS_Visit}
            count ← count + 1
        end if
    end for
    return count
end function

function DFS_Visit(G, v)
    Mark v as visited
    for all u in G.adj[v] do
        if u is not visited then
            DFS_Visit(G, u)
        end if
    end for
end function
```

This algorithm performs a depth-first search traversal of the graph, marking visited vertices and incrementing the count each time a new connected component is encountered. The function CountComponents returns the total number of connected components in the graph.

4 Cycle Detection in Graphs

4.1 Undirected Graphs

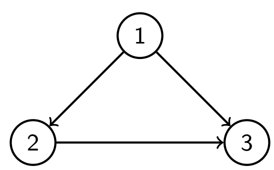
To detect a cyle in undirected graph, we will perform BFS and mark each edge we visit as visited. After that, if there is any unmarked edge left, it shall mean there is a cycle in the graph.

Algorithm 4 Detect Cycle in Undirected Graph using BFS

```
function DetectCycle(G)
visitedEdges ← empty hashmap {Initialize empty hashmap to track visited edges}
queue ← empty queue
for each vertex v in G do
   if v is not visited then
      enqueue v into queue
      mark v as visited
      while queue is not empty do
         u ← dequeue from queue
         for each neighbor w of u do
            if (u, w) is not in visitedEdges then
               add (u, w) to visitedEdges
               enqueue w into queue
            end if
         end for
      end while
   end if
end for
for each edge (u, v) in E(G) do
   if (u, v) is not in visitedEdges then
      return false {Cycle detected}
   end if
end for
return true {No cycle detected} =0
```

4.2 Directed Graphs

The above algorithm would not work in case of directed graphs. Consider the example given below



Suppose we start BFS from node 1, then after the completion, edges $1 \longrightarrow 2$ and $1 \longrightarrow 3$ will be marked visited and the BFS will end. The edge $2 \longrightarrow 3$ will be unvisited and the algorithm will say cycle is present, but as we can see, there is no cycle.

5 DFS Numbering

DFS numbering assigns a unique number to each vertex of a graph during a depth-first search traversal. The numbering reflects the order in which vertices are discovered and processed.

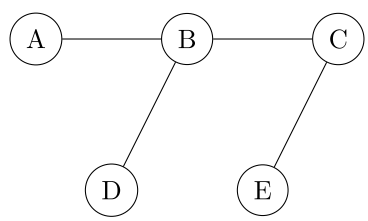
5.1 Pseudocode:

Algorithm 5 DFS with Numbering

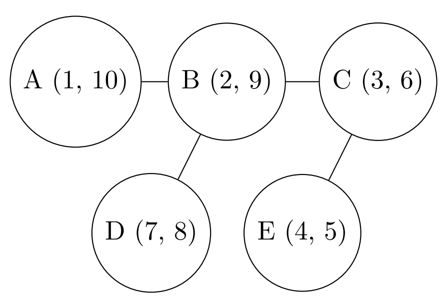
Input: Graph G with vertices V and edges E, starting vertex s
Output: Depth-first traversal of G starting from s with vertex numbering
Procedure: DFS_Numbering($G, s, count$)
Initialize a tuple t (∞, ∞)
Mark s as visited
$t \leftarrow (count, \infty)$
$count \leftarrow count + 1$
for each vertex v **in** $G.adj[s]$ **do**
  if v is not visited **then**
    DFS_Numbering($G, v, count$)
  end if
end for
$t \leftarrow (t.first, count)$
$count \leftarrow count + 1$ =0

5.2 Example:

Consider the following graph:



Starting from vertex A, let's perform DFS numbering:



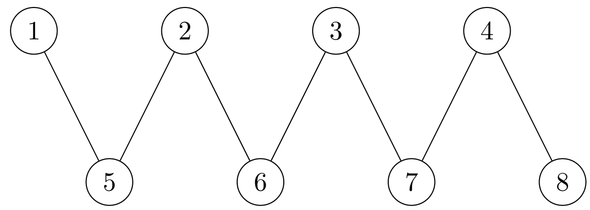
In this example, each vertex is annotated with two numbers: the first number indicates the order in which the vertex was first visited during DFS traversal, and the second number indicates the order in which the vertex was last visited.

6 Articulation Points

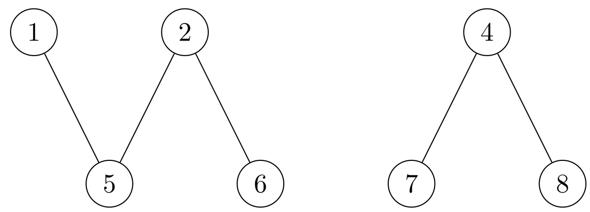
In graph theory, an articulation point (or cut vertex) is a vertex in a graph whose removal would disconnect the graph. Formally, a vertex v is an articulation point if and only if its removal increases the number of connected components in the graph.

6.1 Example

Consider the following graph:



Suppose we remove vertex 3:



After removing vertex 3, the graph becomes disconnected into two components: one containing vertices 1, 2, 5, and 6, and the other containing vertices 4, 7, and 8. Therefore, vertex 3 is an articulation point in the graph.