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=0

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 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 $O(b^d + bd)$ where b^d is for visited array and bd is for recursion stack. Here is 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=0

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 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.

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:

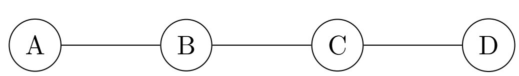
- **Unbounded Exploration:** DFS can get trapped in infinite loops or cycles if the graph contains cycles. Without proper termination conditions, DFS may continue indefinitely without finding a solution.
- **Depth-First Nature:** DFS may explore one branch of the graph extensively before exploring other branches. If the solution lies in an unexplored branch, DFS may fail to find it until it exhausts all other possibilities.

3 Connectivity of Graphs

A graph can be classified based on its connectivity.

3.1 Connected Graphs

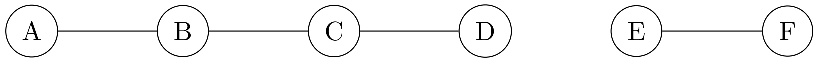
A connected graph is a graph in which there exists a path between every pair of vertices. In other words, there are no isolated vertices, and all vertices are reachable from every other vertex.



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=0
```

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

For undirected graphs, we can detect cycles using a depth-first search (DFS) algorithm. The idea is to perform a DFS traversal of the graph and check for back edges. If we encounter an already visited vertex (other than the parent), it indicates the presence of a cycle.

Algorithm 4 Detect Cycle using DFS

```
procedure DFS(G, v, visited, parent)
  visited[v] ← True
  for all u in G.adjacent(v) do
    if visited[u] is False then
      DFS(G, u, visited, v)
    else if u ≠ parent then
      return True {Cycle Detected}
    end if
  end for
  return False
end procedure

procedure DetectCycle(G)
  visited ← [False] * |G.vertices()|
  for all v in G.vertices() do
    if visited[v] is False then
      if DFS(G, v, visited, −1) then
        return True {Cycle Detected}
      end if
    end if
  end for
  return False
end procedure=0
```

4.2 Directed Graphs

We can find a cycle using DFS algorithm such that when if our vertex edge encounters an vertex which is already in our path, then we conclude it is a cycle.

Algorithm 5 Detect Cycle in Undirected Graph

```
Function HasCycle(graph)
visited[vertex] ← array of size V (number of vertices), initialized to False
path[vertex] ← array of size V, initialized to False

Function DFS(vertex, parent)
visited[vertex] ← True
path[vertex] ← True
for each neighbor of vertex do
  if neighbor is not visited then
    if DFS(neighbor, vertex) is True then
      return True {Cycle detected}
    end if
  else if path[neighbor] is True and neighbor ≠ parent then
    return True {Cycle detected}
  end if
end for
path[vertex] ← False
return False

for each vertex in graph do
  if vertex is not visited then
    if DFS(vertex, None) is True then
      return True {Cycle detected}
    end if
  end if
end for
return False {No cycle detected}
End Function =0
```

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.

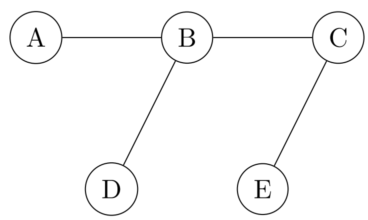
5.1 Pseudocode:

Algorithm 6 DFS with Numbering

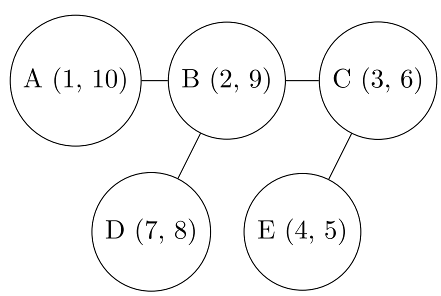
```
Input: Graph G with vertices V and edges E, starting vertex s
Output: Depth-first traversal of G starting from s with vertex numbering
Procedure: DFS_Numbering(G, s, count)
Initialize a tuple t (∞, ∞)
Mark s as visited
t ←(count, ∞)
count ← count + 1
for each vertex v in G.adj[s] do
    if v is not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

5.2 Example:

Consider the following graph:



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



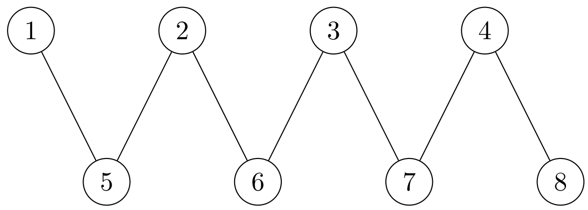
In this example, each vertex is annotated with two numbers: the first number indicates the order in which the vertex was first visited during DFS traversal, and the second number indicates the order in which the vertex was last visited.

6 Articulation Points

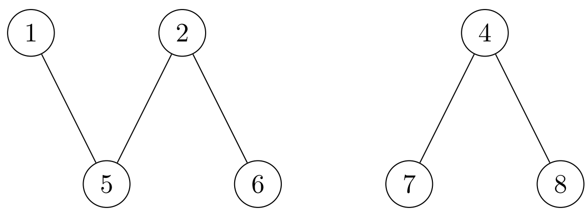
In graph theory, an articulation point (or cut vertex) is a vertex in a graph whose removal would disconnect the graph. Formally, a vertex v is an articulation point if and only if its removal increases the number of connected components in the graph.

6.1 Example: Butterfly Graph

Consider the following butterfly 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.