CSCI 4470 Algorithms

Part V Advanced Data Structures

- 17 Augmenting Data Structures
- 18 B-Trees
- **19 Data Structures for Disjoint Sets**

Chapter 19: Data Structures for Disjoint Sets

```
19 Data Structures for Disjoint Sets
    19.1 Disjoint-set operations
    19.2 Linked-list representation of disjoint sets
    19.3 Disjoint-set forests
    19.4 Analysis of union by rank with path compression
```

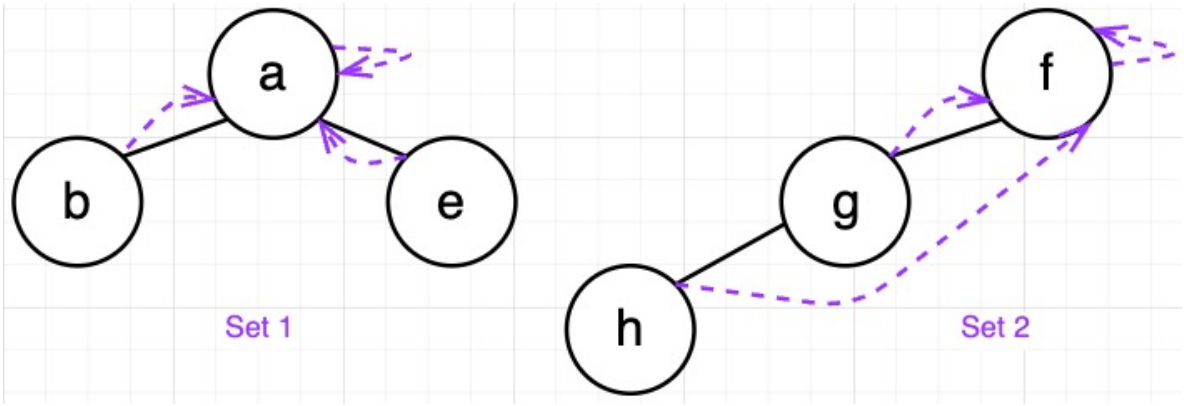
Introduction of Disjoint Sets

For graph algorithms, we sometimes need to manage disjoint sets of **`vertices`** or **`edges`**

- Disjoint Set aka. `Union-Find` Set
- Used in graph algorithms.
- Known as a disjoint data structure.

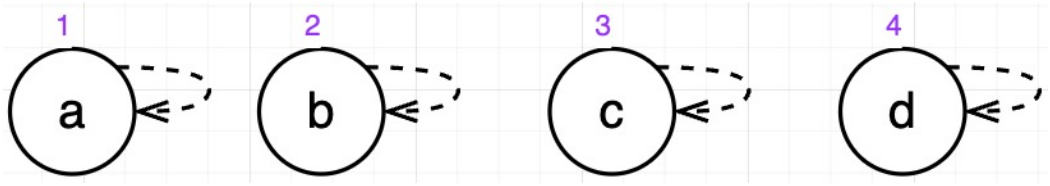
DISJOINT SET DATA STRUCTURE

- Maintains a collection of sets $S = \{S_1, ..., S_k\}$, $S_i \bigcap S_j = \emptyset$ for all $i \neq j$
- Each set (or component) is identified by a representative member, $S_1 \bigcap S_2$ and $i \neq j$, elements in both set are not the same.
    - e.g., **`Set 1 {a, b, e}`** represented by `a`
    - e.g., **`Set 2 {f, g, h}`** represented by `f`
    - all elements in a set will point to its representative (root) element

- 

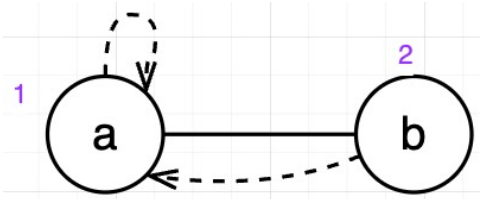
THREE OPERATIONS

1. MAKE-SET(x) Operation: Creates a disjoint set S. If starting with a single element A, A is its own representative.

- 

- `MAKE-SET(x)` : Creates a set $S = \{a, b, c, d\}$.
- When called `MAKE-SET(x)` , see the graph above, there are 4 sets.

2. UNION(x,y) Operation: Merges two sets. If merging S_1 and S_2, they should have one common representative.

- 

- `UNION(x,y)` : Merges set containing x with set containing y.
- When called `UNION(S1,S2)` , assume **x is S_1 and y is S_2**

3. FIND-SET(x) Operation: Returns the representative of a set.

For example, for **Set** $1 \cup 2$, it returns its representative which is `a` .

- `FIND-SET(S1 U S2)` : Returns a pointer to the representative for the set containing a.
- find set basically means to find the representation of the set.

Applications of Disjoint Sets

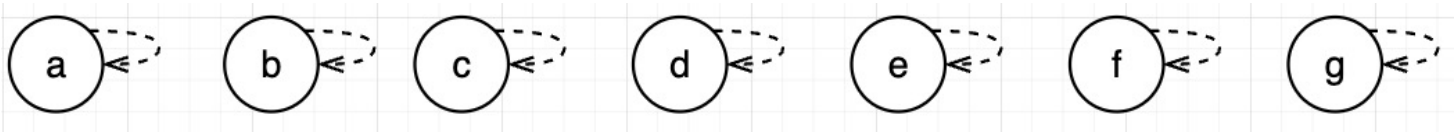
1. Finding Connected Components in a Graph:

- A graph can have multiple components.
- BFS and DFS can find connected components but are less efficient, Disjoint Set can be simpler than BFS/DFS
- Disjoint sets are more efficient for this task. (merging components in near **CONSTANT TIME**)

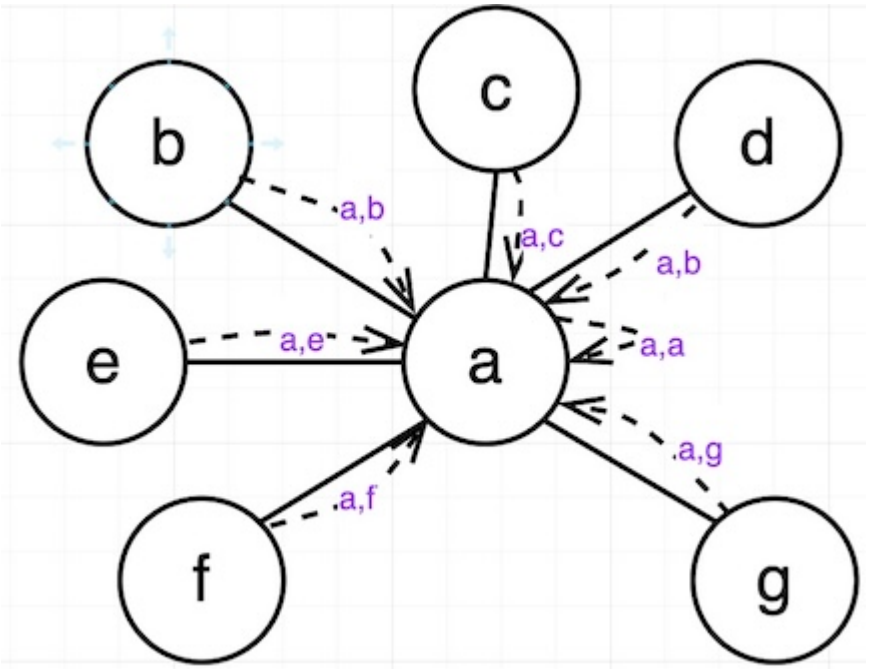
Example 1:, A graph with one connected component

Graph:
Vertices: a, b, c, d, e, f, g, h
Edges: (a, b), (a, c), (a, d), (a, e), (a, f), (a, g), (a, h)

Step 1: Initialization Each vertex is its own set



Step 2: `UNION(v,u)` sets, for each edge, union the two vertices



`UNION(a,b)` , `a` is the representative
`UNION(a,c)` , `a` is the representative

…continue this for all edges

Step 3: Using `FIND-SET(e)` , all vertices have `a` as their representative, indicating they are all in the same connected component which is **ONLY ONE COMPONENT**

2. Detecting Cycles in a Graph:

- Used in Prim's and Kruskal's algorithms for minimum spanning trees.
- If an edge forms a loop, it's discarded.
- Disjoint sets can efficiently detect cycles.

Example 2 Disjoint Set for detecting Cycles

Complexity:

- If there are M operations (Make Set, Union, Find Set) and N is the number of Union operations, the complexity is $M \times \alpha(N)$, where $\alpha(N)$ is a very slow-growing function, almost constant.

DISJOINT SET FORESTS

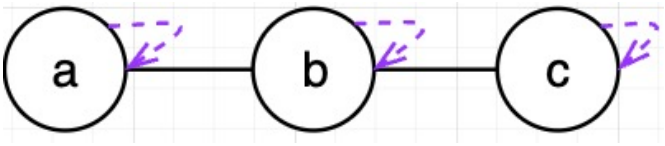
- A disjoint forest is a collection of trees where each tree represents a set. Each node in the tree represents an element, and the root of the tree represents the representative of the set.

Implementations

- **1. Linked List**:
- **2. Disjoint Forest**:

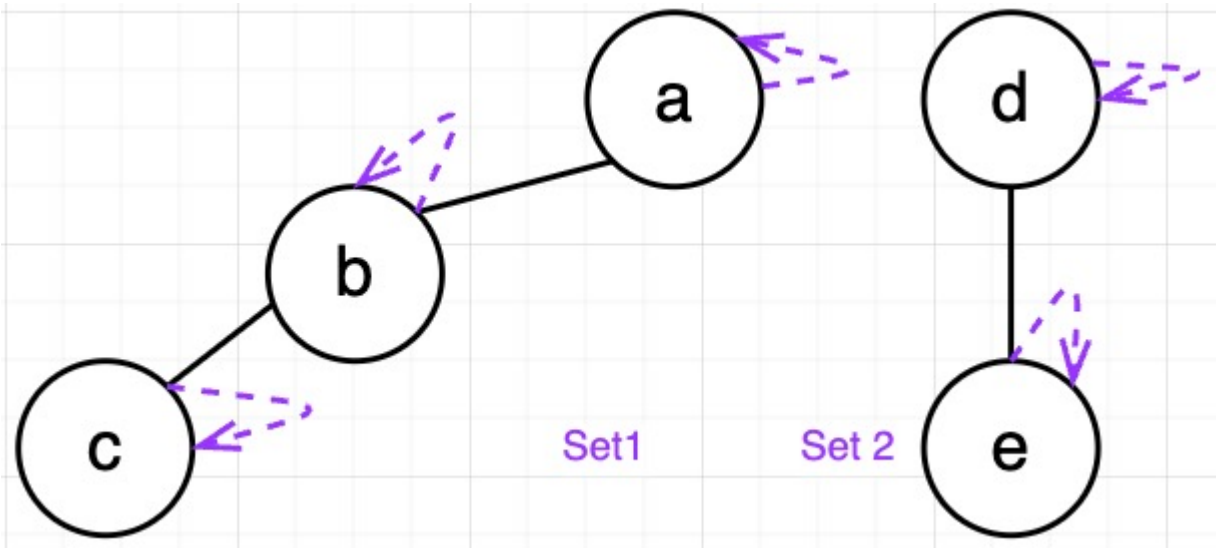
1. Linked List:

- Each node points to its representative.
- Operations can be linear.
- Uses many pointers.
- e.g., A set represents in Linked List.

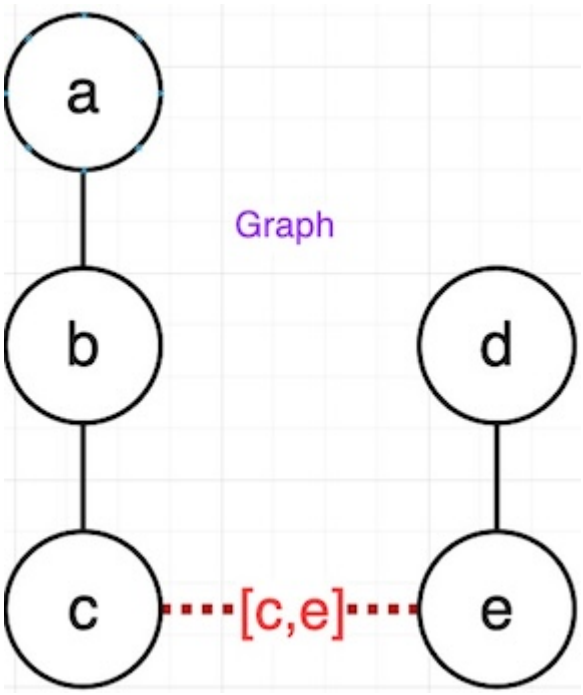


2. Disjoint Forest:

Initialization each element is its own set,

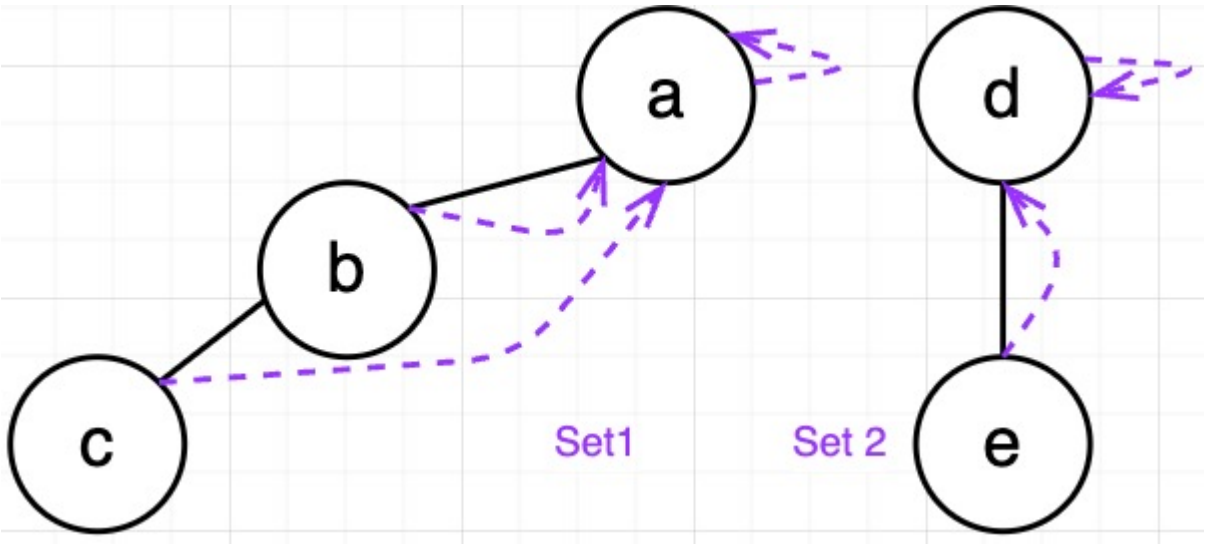


Graph Example, Assume added the `EDGE(c,e)` , the graph as shown below:



- The graph has two Disjoint Sets
- Trees where each node points to its parent.
- e.g., Two Disjoint Sets (Components)

UNION() Operation for sets s1 and s2, for elements in the same set, pick the representative for all elements

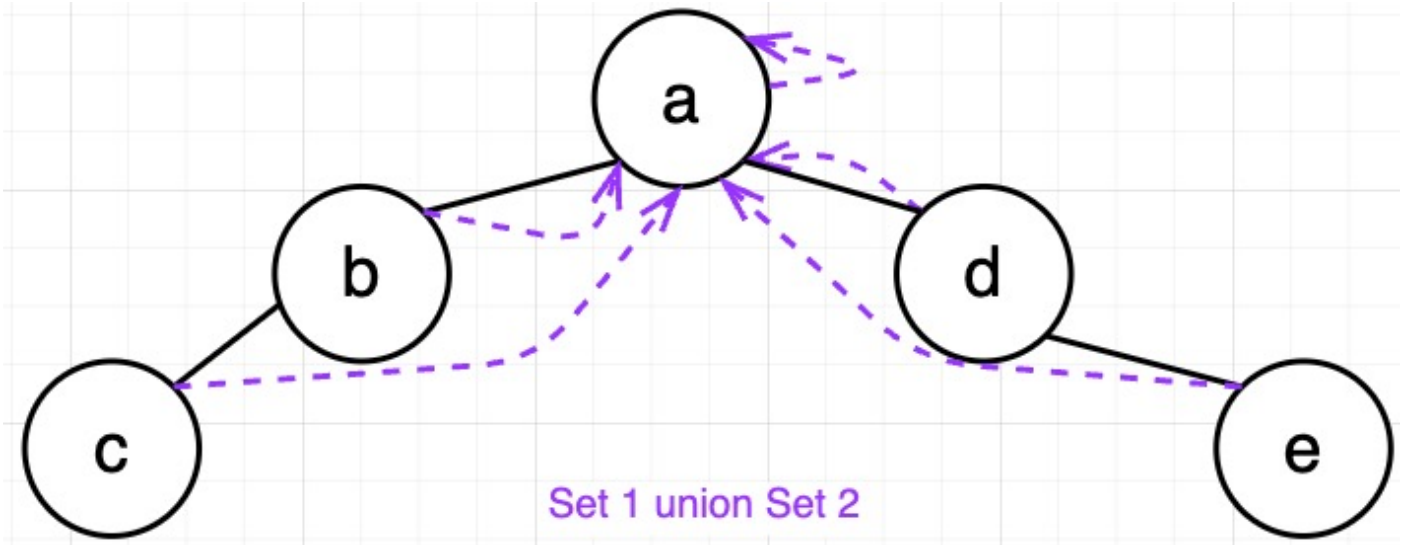


FIND-SET(e), After the execution of `UNION()` , the representatives of sets `s1` and `s2` can be determined

- e.g., Called `FIND-SET(e)` returns `d` , Called `FIND-SET(c)` returns `a`
- So, the `s1` set has `a` as the root (representative), and `s2` set has `d` as the root (representative)
- The complexity is just the **constant** when call `FIND-SET()`

UNION(s1,s2) Also, sets `s1` and `s2` can be merged

- Path compression: After finding a representative, update pointers to point directly to the root.
- **Union by Rank: Merge smaller tree into the bigger tree to minimize pointer updates.**
- `UNION(s1, s2)`



- Rank: Upper bound of the height of the trees. When merging two sets of the same rank, the resulting set's rank increases by one.
- Call `FIND-SET(c) = a` , `a` is the root, complexity of the operation is $O(1)$
- All elements in set `s1&s2` are pointed to element `a` in the example

TWO HEURISTICS

1. Path Compression:

- After running `FIND-SET(x)` , all nodes between x and the root will now point to the root.

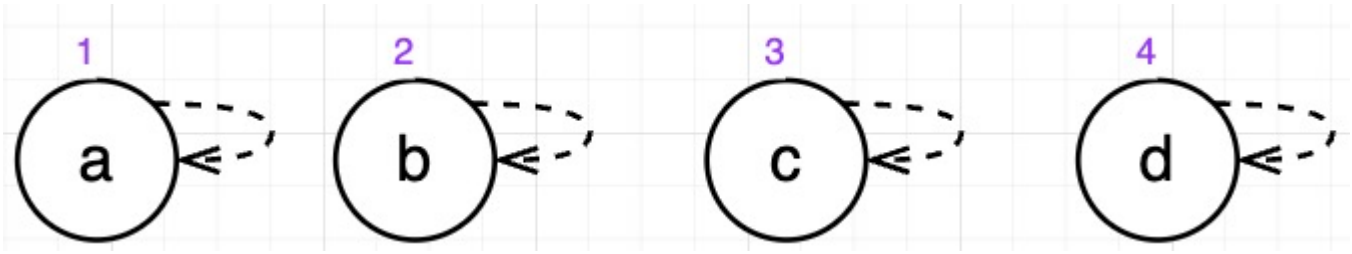
2. Union by Rank:

- `UNION(x,y)` makes the smaller rank set point to the representative of the higher rank set.
- If both have the same rank, the rank is incremented.

EXAMPLES of Disjoint Sets

Example 1 - 4: Called `MAKE-SET(a)` , `MAKE-SET(b)` , `MAKE-SET(c)` , and `MAKE-SET(d)` to generate 4 sets as below

- The **RANK = 0**

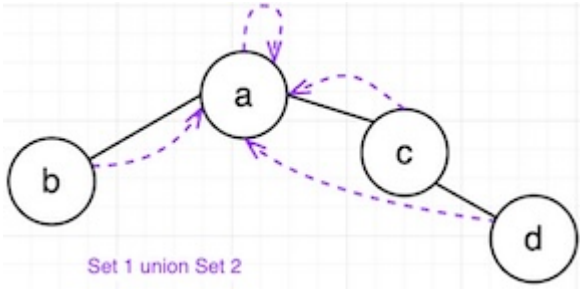


Example 5 - 6: `UNION(a,b)` , `UNION(c,d)`

- The **RANK = 1**

Example 7: `UNION(a,c)`

- The **RANK = 2**



Example 8: `FIND-SET(d)`

- The **RANK = 2** is remained.
- When Called `FIND-SET(d)` returns `a`, which is the representative.

RUNTIME

A sequence of m `MAKE-SET`, `UNION`, and `FIND-SET` operations, n of which are `MAKE-SET` operations, has worst-case runtime $O(m\,\alpha(n))$, where $\alpha(n)$ is an extremely slow-growing function

- $\alpha(n) \leq 4$ in any conceivable application
- We can effectively consider this to be $O(n)$ runtime, though technically it is not

GRAPH REPRESENTATION

Types of Graphs

- **Undirected Graph**: No direction on edges.
- **Directed Graph**: Edges have directions.
- **Weighted Graph**: Edges have weights.
- **Sparse Graph**: Number of edges is significantly less than V^2.
- **Dense Graph**: Number of edges is proportional to V^2.
- **Adjacency List**: Represents a graph using a list where each vertex points to its neighbors.
  - Storage Complexity: O(V+E)
  - Used when the graph is sparse.
- **Adjacency Matrix**: Represents a graph using a matrix where rows and columns represent vertices and the presence of an edge is marked.
  - Storage Complexity: $O(V^2)$
  - Used when the graph is dense.

Graph Definition:
A graph is represented by a set of vertices and edges.
$G = (V, E)$

- V = set of vertices $) = \{v_1, v_2, ..., v_n\}$.
- E = set of edges $\subseteq V \times V$.

Edge Interpretation

- $(v_i, v_j) \in E$ means there's a single step from v_i to v_j.
- For an **Undirected Graph G**, if $(v_i, v_j) \in E$, then $(v_j, v_i) \in E$ also holds true.

Graph Characterization:

- **Sparse Graph**: A graph where the number of edges is much less than the maximum possible number of edges. Mathematically, $|E| << |V|^2$.
  - the maximum number of edges is $\frac{|V|(|V|-1)}{2}$
- **Dense Graph**: A graph where the number of edges is close to the maximum possible number of edges. Mathematically, $|E| \approx |V|^2$.

Undirected Graph

Undirected Graph Example G=(V, E): No direction on edges `V = {a, b, c, d}` , `E = {(a,b), (b,c), (c,d), (d,a)}`

- in the **Undirected Graph**, Edge `(d, a) = (a, d)`

Directed Graph

Directed Graph Example G=(V, E)

- **V (Vertices)**: The set of nodes or points in the graph.
- For the given graph G, `V = {a, b, c, d}` .
- **E (Edges)**: The set of lines connecting the vertices, representing relationships or connections.
- For the given graph G, `E = {(a, b), (b, c), (c, d), (d, a)}` .
  - $E(d, a) \neq E(a, d)$

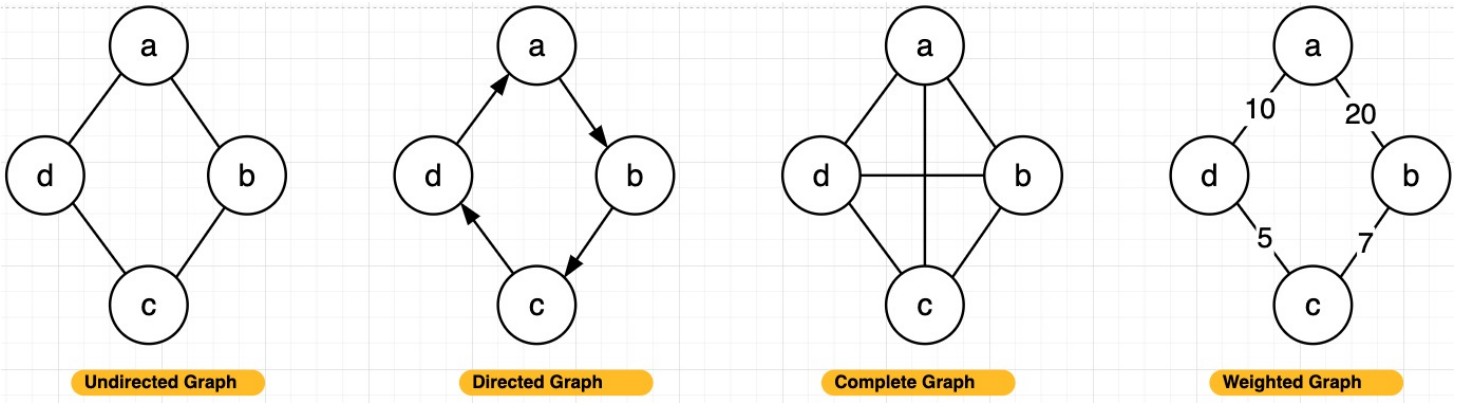
Complete Graph

V (Vertices): `V = {a, b, c, d}` , **E (Edges)** `E = {(a,b)(b,c)(c,d)(d,a)(a,c)(b,d)}`

- $Edge \propto |V|^2$, A Dense Graph

Weighted Graph

Weighted Graph Example G = (V, E): **V (Vertices):** `V = {a, b, c, d}` , **E (Edges with weights):** `E = {(a, b, 20), (b, c, 7), (c, d, 5), (d, a, 10)}`



Edges with Associated Values:

- Edges in a graph may have an associated value, indicating some measure or quantity.

- This value could represent distance, cost, time, or any other measurable factor.
- **Edge Weight**: In cases where edges have an associated value, they are termed as 'weighted' with the value denoted as $w(u, v)$.
  - For an edge from vertex u to vertex v, its weight is represented as $w(u, v)$.

Representation in Adjacency List:

- In an adjacency list representation of a weighted graph, a weight component is added to each list element.
- The list for a vertex can have pairs (neighbor, weight) indicating the neighbor vertex and the weight of the edge connecting them.

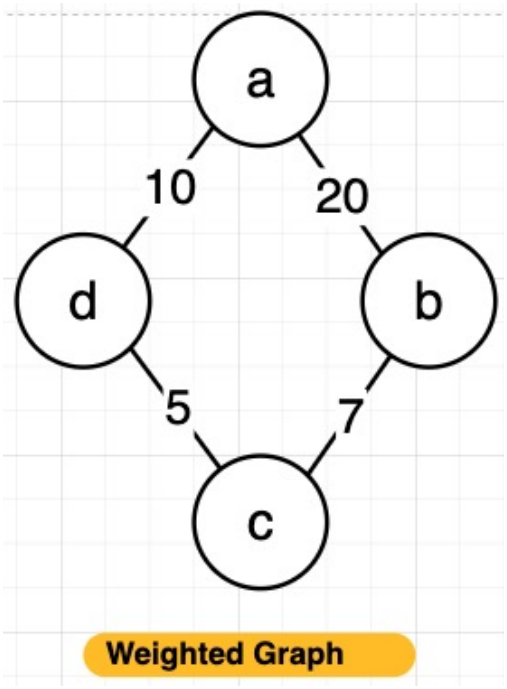
Representation in Adjacency Matrix:

- In an adjacency matrix representation of a weighted graph, the matrix entry at `[i][j]` will represent the weight of the edge between vertex i and vertex j.
- If there's an edge from vertex i to vertex j with weight w, then the matrix entry at `[i][j]` will be w. If there's no edge, the entry could be infinity or a large value, depending on the context.

Adjacency List Representation

- Represents a graph as an array of linked lists.
- Array index denotes a vertex; linked list elements indicate vertices forming an edge with it.

Class Example: Adjacency List with Weighted and Undirected Graph



- **Vertices (V)**: {a, b, c, d}
- **Edges (E)**: {(a, b, 20), (b, c, 7), (c, d, 5), (d, a, 10)}

Advantages:

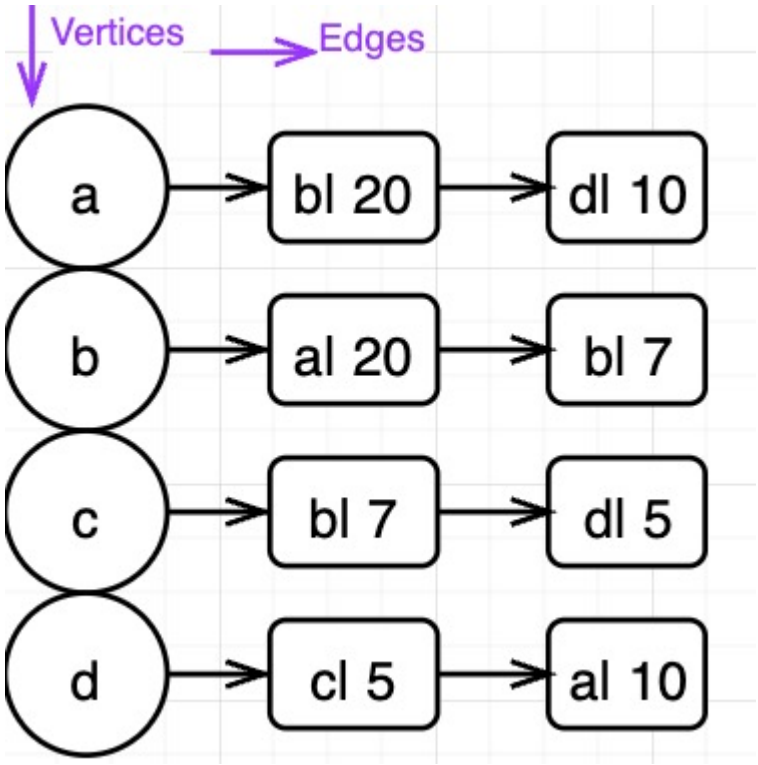
- Space-efficient for sparse graphs.
- Facilitates finding all neighbors of a vertex.

Drawbacks:

- Less efficient for dense graphs or checking the existence of an edge between two nodes.

For **sparse graphs**:

- **Adjacency List** is preferred due to space efficiency proportional to $|V| + |E|$.
  - **the complexity is $O(V + E)$**



Adjacency Matrix Representation

Adjacency Matrix, A square matrix used to represent a graph.

- **Pros**: Provides a quick way to check if an edge exists between two nodes. Good for dense graphs.
- **Cons**: Takes up more space, especially for sparse graphs. Less space efficient than adjacency lists for such graphs.

For **dense graphs**:

- **Adjacency Matrix** is favored because it offers constant-time $O(1)$ edge look-up and uses space proportional to $|V|^2$, which is close to the number of edges in dense graphs.
  - The complexity of adjacency matrix is $O(V^2)$

	a	**b**	**c**	**d**
a	0	20	0	10
b	20	0	7	0
c	0	7	0	5
d	10	0	5	0

CHOOSING LIST VS. MATRIX

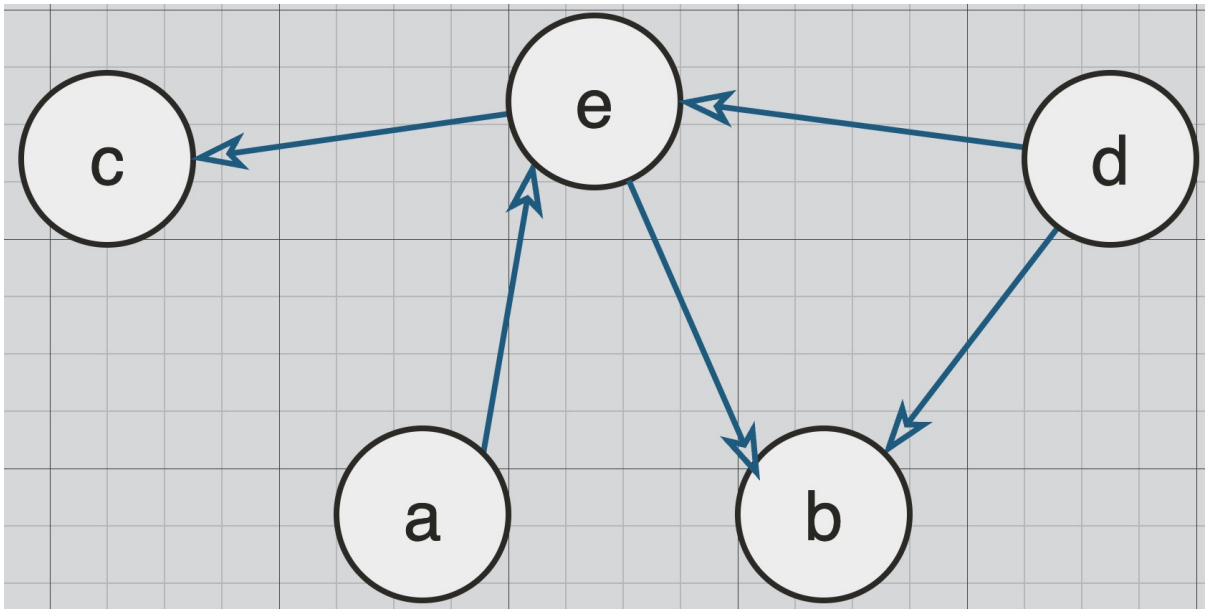
Storage Space:

- **Adjacency List**: Proportional to the number of vertices and edges, $O(V + E)$.
- **Adjacency Matrix**: Always $O(V^2)$, regardless of the number of edges.

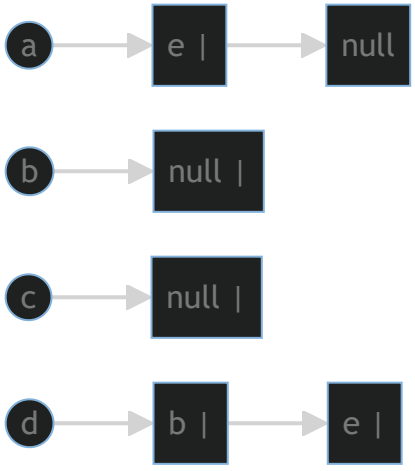
Time to Verify Edge Existence:

- **Adjacency List**: $O(1)$ for sparse graphs, but can be up to $O(E)$ for dense graphs.
- **Adjacency Matrix**: Always $O(1)$ since you're directly accessing a cell in the matrix.
- For **dense graphs**, the adjacency matrix offers constant time edge verification but uses more space.

Example: Adjacency List and Adjacency Matrix



↓ Vertices → Edges



Adjacency Matrix

	a	b	c	d	e
a	0	0	0	0	1
b	0	0	0	0	0
c	0	0	0	0	0
d	0	1	0	0	1
e	0	1	1	0	0

Search Algorithms BFS and DFS

Introducing two primary search techniques:

- Their distinction lies in the sequence of node visits.
- Both methods initiate from a source vertex s.

1. Applications:

- **BFS**: Shortest path in unweighted graphs, network broadcasting.
- **DFS**: Topological sorting, cycle detection, pathfinding in weighted graphs.

2. Data Structures Used:

- **BFS**: Queue
- **DFS**: Stack or Recursion

3. Time Complexity:

- For both BFS and DFS on adjacency list representation: $O(V + E)$, where V is ˜/the number of vertices and E is the number of edges.

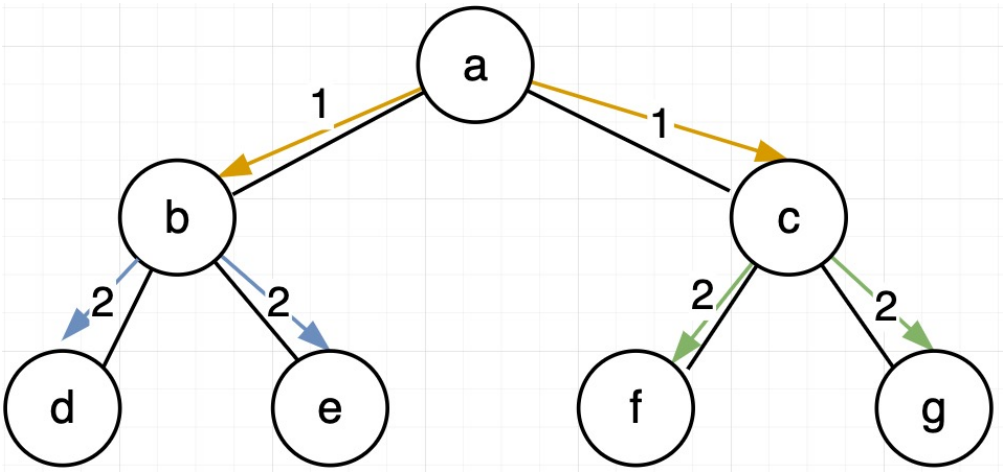
Breadth First Search (BFS)

- Aims to discover all nodes at distance d from s before any nodes at distance $d + 1$.

Steps (BFS):

1. **Start** at vertex `a`.
2. ***Visit** all neighbors of `a` : `b` and `c`.*
3. ***Move** to `b`, visit its unvisited neighbors: `d` and `e`.*
4. ***Move** to `c`, visit its unvisited neighbors: `f` and `g`.*
5. **Order** of visited vertices: `{a, b, c, d, e, f, g}`.

BFS Example: (Undirected and Unweighted Graph)



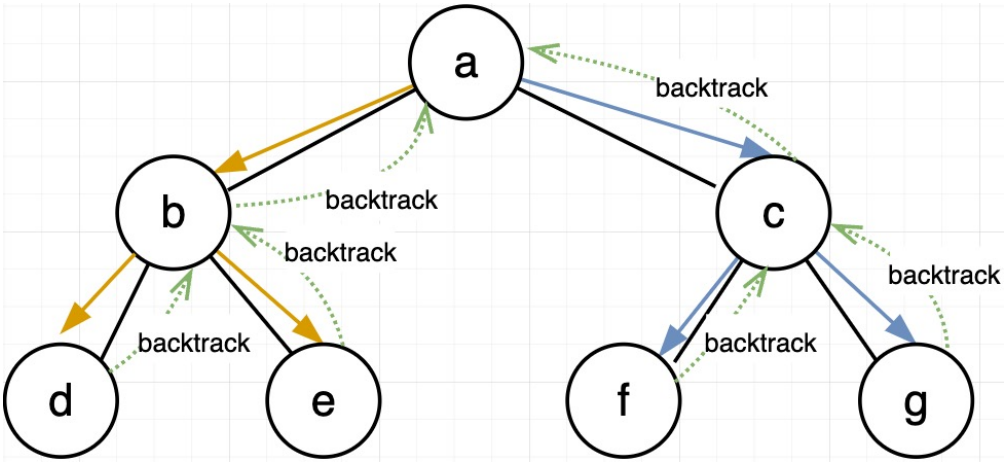
Depth First Search (DFS)

- Pursues each path to its fullest extent and backtracks, opting for any unexplored paths during the return journey.
- Follow each path as far as possible and backtrack, taking any untraveled paths while returning

Steps (DFS):

1. **Start** at vertex `a`.
2. **Explore** as far as possible along each branch before backtracking.
3. ***Visit** `b` from `a`, then move to its unvisited neighbor `d`.*
4. ***Backtrack** to `b`, then visit `e`.*
5. ***Return** to `a`, then visit `c`. From `c`, visit `f` and then `g`.*
6. **Order** of visited vertices: `{a, b, d, e, c, f, g}`.

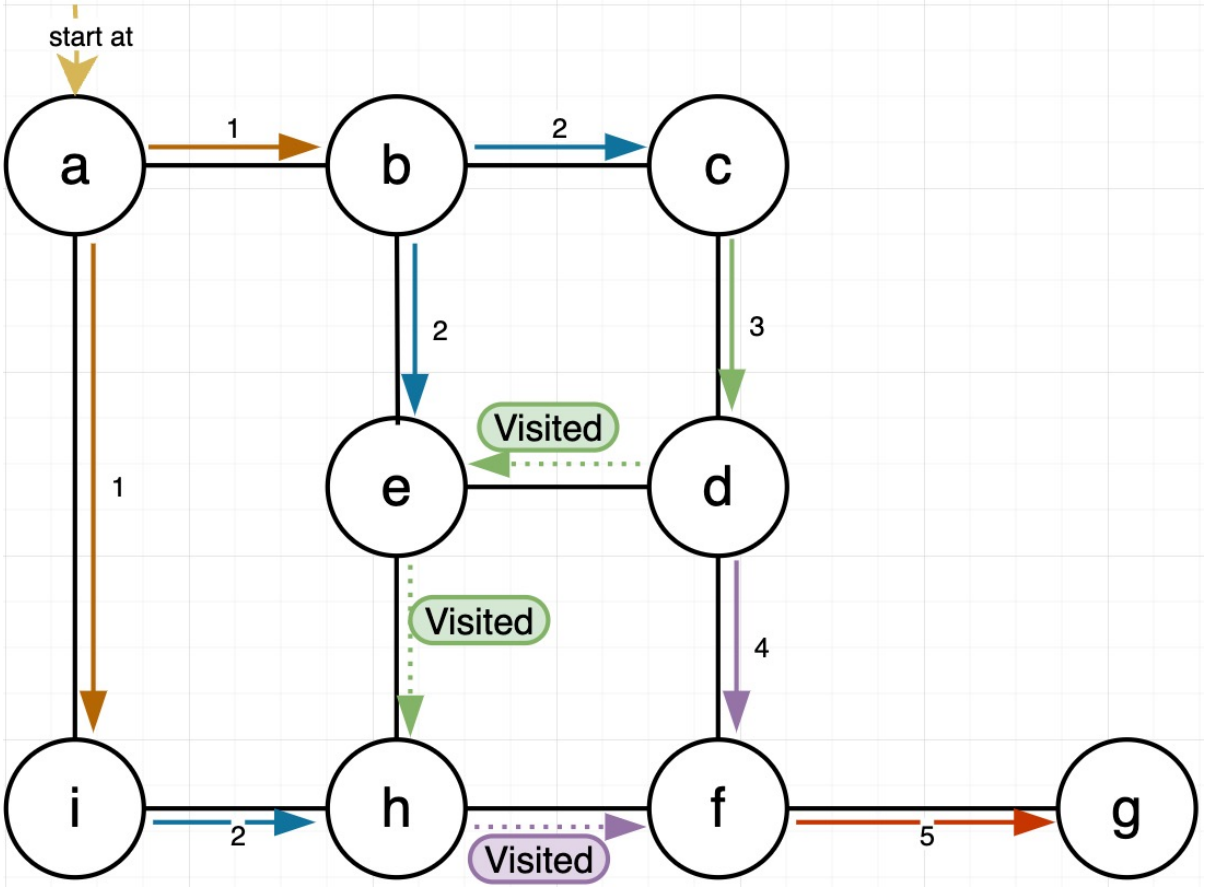
DFS Example: (Undirected and Unweighted Graph)



Note: The order in DFS can vary based on the starting point and the order of neighbors in the adjacency list.

Undirected Graph Example of BFS and DFS, The graph shown below:

V Vertices = {a, b, c, d, e, f, g, h , i},
E Edges = {(a,b), (b,c), (c,d), (d, f), (f,g), (a,i), (i,h), (h,f), (b,e), (e,h)}



QUIZ and EXAM,

BFS Solution (QUEUE), Start from `Node a` ,

1. Visit the starting vertex `a` and enqueue it.

- Queue: [a] and Visited: `{a,}`

2. Dequeue `a` and enqueue its unvisited neighbors `b` and `i`.
   - Queue: [b, i] and Visited: `{a, b, i}`
3. Dequeue `b` and enqueue its unvisited neighbor `c` and `e`.
   - Queue: [i, c, e] and Visited: `{a, b, i, c, e}`
4. Dequeue `i` and enqueue its unvisited neighbor `h`.
   - Queue: [c, e, h] and Visited: `{a, b, i, c, e, h}`
5. Dequeue `c` and enqueue its unvisited neighbor `d`.
   - Queue: [e, h, d] and Visited: `{a, b, i, c, e, h, d}`
6. Continue this process until the queue is empty.

BFS (Breadth First Search) table Starting from vertex `a`:

Step	Current Vertex	Queue	Visited Vertices
1	a	[b, i]	{a}
2	b	[i, c, e]	{a, b}
3	i	[c, e, h]	{a, b, i}
4	c	[e, h, d]	{a, b, i, c}
5	e	[h, d]	{a, b, i, c, e}
6	h	[d, f]	{a, b, i, c, e, h}
7	d	[f]	{a, b, i, c, e, h, d}
8	f	[g]	{a, b, i, c, e, h, d, f}
9	g	[]	{a, b, i, c, e, h, d, f, g}

- `{a, b, i, c, e, h, d, f, g}`, The Complexity is $O(V + E)$, which linear
- Enqueue and Dequeue of all Vertices and Edges ONLY ONE TIME which is a constant

BFS(G, s), This function performs Breadth First Search on a `Graph G` starting from `Vertex s`.

```
BFS(G, s)
1. for each vertex u ∈ V-{s}
2.    u.color = white
3.    u.π = NIL; u.d = ∞ // u.π is parent vertices, u.d is the distance of vertex u from source s
4. s.color = gray
5. s.d = 0
6. Q = ∅
7. enqueue (Q, s)   // all vertices will be enqueued ONE TIME
8. while Q != ∅   // while is true, each vertex will be enqueued ONE TIME, While Loop runs O(V) times.
9.    u = dequeue(Q)
10.   for each v ∈ G.Adj[u]   // This for loop is nested in the while loop, but it ONLY goes next neighbors
O(E)
11.     if v.color == white
12.        v.color = gray
13.        v.π = u; v.d = u.d + 1
14.        enqueue (Q,v)
15.   u.color = black   // the vertex is marked as black, which means never enqueue again
```

1. Initialization:

```
1. for each vertex u ∈ V-{s}
2.    u.color = white
3.    u.π = NIL; u.d = ∞
```

For every vertex `u` in the graph except the starting vertex `s`, we:

- Set its color to `white` indicating it's unvisited.
- Set its predecessor `u.π` to `NIL` (no predecessor yet).
- Set its distance `u.d` from the source to infinity ∞ initially.

4. Starting Vertex Initialization:

```
4. s.color = gray
5. s.d = 0
```

For the starting vertex `s`:

- Set its color to `gray` indicating it's discovered but not fully explored.
- Set its distance `s.d` from itself to 0.

6. Queue Initialization:

```
6. Q = ∅
7. enqueue (Q,s) // all vertices will be enqueue ONE TIME
```

Initialize an empty queue `Q` and enqueue the starting vertex `s`.

8. BFS Loop:

```
8. while Q != ∅
9.     u = dequeue(Q)
```

While the queue is not empty, dequeue a vertex `u` from the front of the queue.

10. Neighbor Exploration:

```
10.   for each v ∈ G.Adj[u] // This for loop is nested in while loop, but it ONLY goes next Neighbors
11.     if v.color == white
12.        v.color = gray
13.        v.π = u; v.d = u.d + 1
14.        enqueue (Q,v)
```

For each neighbor `v` of `u`:

- If `v` is unvisited (`white`), mark it as discovered (`gray`).
- Set `u` as the predecessor of `v` and update the distance of `v` from the source.
- Enqueue `v` to the queue. 將 `v` 入隊到隊列。

15. Vertex Fully Explored:

```
15.   u.color = black
```

After exploring all neighbors of `u`, mark `u` as fully explored (`black`).

- The Complexity is $O(V + E)$

DFS Solution (STACK), Start from `Node a`,

V Vertices = {a, b, c, d, e, f, g, h, i}
E Edges = {(a,b), (b,c), (c,d), (d, f), (f,g), (a, i), (i,h), (h,f), (b,e), (e,h), (e,d)}

DFS Solution, Starting from `Node a`:

1. `a`: Start at node `a`.

- Stack: [a] and Visited nodes: `{a,}`

2. `b`: Move from `a` to `b`.

- Stack: [a, b] and Visited nodes: `{a, b}`

3. `c` : Move from `b` to `c` .

- Stack: [a, b, c] and Visited nodes: `{a, b, c}`

4. `d` : Move from `c` to `d` .

- Stack: [a, b, c, d] and Visited nodes: `{a, b, c, d}`

5. `e` : Move from `d` to `e` .

- Stack: [a, b, c, d, e] and Visited nodes: `{a, b, c, d, e}`

6. `h` : Move from `e` to `h` .

- Stack: [a, b, c, d, e, h] and Visited nodes: `{a, b, c, d, e, h}`

7. `i` : Move from `h` to `i` .

- Stack: [a, b, c, d, e, h, i] and Visited nodes: `{a, b, c, d, e, h, i}`

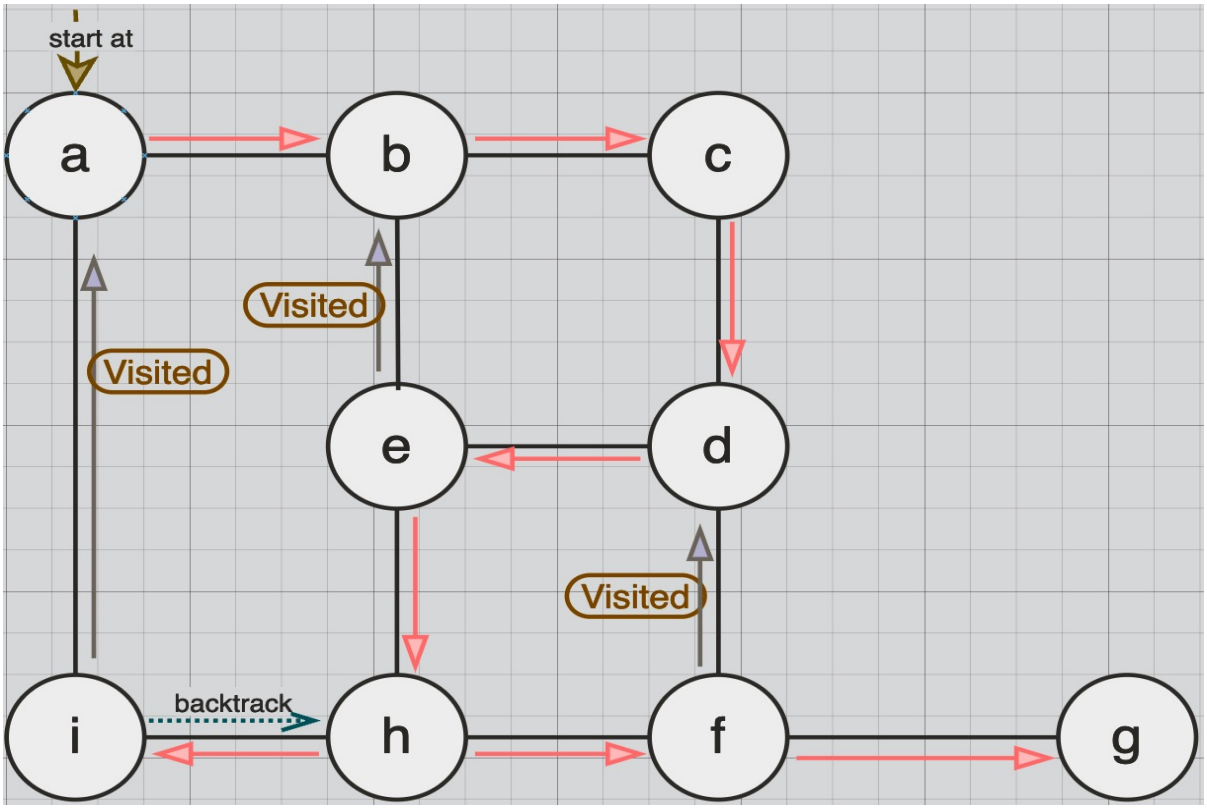
8. Backtrack to h: Since `i` has no unvisited neighbors, backtrack to `h` .

9. `f` : Move from `h` to `f` .

- Stack: [a, b, c, d, e, h, h, f] and Visited nodes: `{a, b, c, d, e, h, i, f}`

10. `g` : Move from `f` to `g` .

- Stack: [a, b, c, d, e, h, i, f, g] and Visited nodes: `{a, b, c, d, e, h, i, f, g}`
  End of DFS traversal. All nodes in the sequence have been visited.



DFS (Depth First Search) Table Starting from vertex `a` :

Step	Current Node	Action	Path So Far	Reason
1	a	Visit	a	Start node
2	b	Visit	a -> b	Unvisited neighbor of a
3	c	Visit	a -> b -> c	Unvisited neighbor of b
4	d	Visit	a -> b -> c -> d	Unvisited neighbor of c
5	e	Visit	a -> b -> c -> d -> e	Unvisited neighbor of d
6	h	Visit	a -> b -> c -> d -> e -> h	Unvisited neighbor of e
7	i	Visit	a -> b -> c -> d -> e -> h -> i	Unvisited neighbor of h
8	i	Backtrack	a -> b -> c -> d -> e -> h	No unvisited neighbors for i
9	f	Visit	a -> b -> c -> d -> e -> h -> f	Unvisited neighbor of h
10	g	Visit	a -> b -> c -> d -> e -> h -> i -> f -> g	Unvisited neighbor of f

After visiting `g` , the DFS traversal is complete for the connected component starting from `a` .

Before The BFS LEMMAS

Imagine a graph with a source vertex s. There are multiple paths from s to a vertex v. Among these paths, there's a path that goes through vertex u which is the shortest path from s to v.

Visualization:

Vertices and Paths

Paths Visualization:

1. **Direct**: `[s] ----> [v]`
2. **Indirect**: `[s] ----> [u] ----> [v]`

- Let's have three vertices: s, u, and v.
- There's a path from s to v that goes through u. This is the shortest path.
- There might be other paths from s to v that don't go through u.

2. $u.d$: The BFS distance from s to u is represented as $u.d$.

- The shortest path distance from s to u is $\delta(s, u)$.

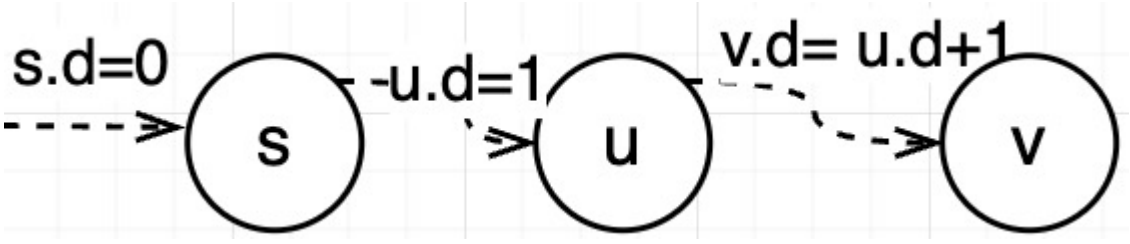
3. $v.d$: The BFS distance from s to v is represented as $v.d$.

- The shortest path distance from s to v is $\delta(s, v)$.

4. $u.d + 1$: This represents the BFS distance from the source vertex s to a vertex adjacent to u. Since BFS explores layer by layer, the distance to any neighbor of u would be $u.d + 1$.

Distances

- $\delta(s, u)$ is the shortest distance from s to u. $\delta(s, u)$
- $\delta(s, v)$ is the shortest distance from s to v. $\delta(s, v)$



In this graph:

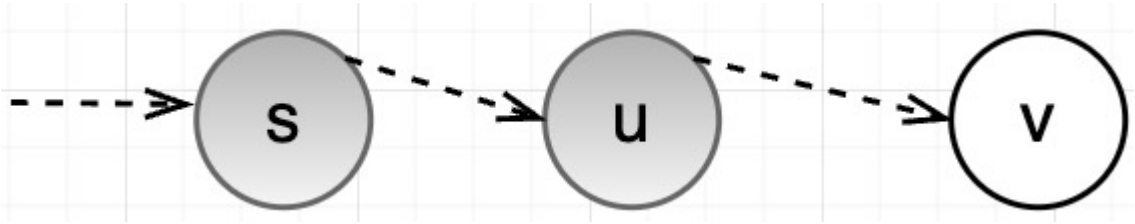
- The vertex `s` has a BFS distance `d` of 0 since it's the source.
- The edge from `s` to `u` is labeled with `u.d`, representing the BFS distance from `s` to `u`.

- The edge from `u` to `v` is labeled with `u.d + 1`, representing the BFS distance from `s` to `v` since it's one more than the distance to `u`.

The lemma states that $v.d$ (the BFS distance from s to v) is always less than or equal to $u.d + 1$. This is visually represented by the fact that v is directly reachable from u, which is directly reachable from s.

Lemma 1 of the BFS

Lemma 1: Given a vertex u that precedes vertex v on the shortest path from source s to v, we want to prove that $v.d = \delta(s, v)$, where $\delta(s, v)$ represents the shortest distance from source s to vertex v. For an edge (u, v), we assume $\delta(s, v) \leq \delta(s, u) + 1$ and $v.d > u.d + 1$.



Case 1:
1. Unreachable Vertex: If a vertex (either u or v) can't be reached from s, then $\delta(s, u)$, $\delta(s, v)$, $u.d$, and $v.d$ are all ∞.

Case 2:
1. Direct Path: If the edge from u to v is part of the shortest path from s to v, then $\delta(s, v) = \delta(s, u) + 1$.

2. Through u: If the shortest path from s to v passes through u, then $\delta(s, v) = \delta(s, u) + 1$.

- Graph: `s -> ... -> u -> v`

Case 3:
3. Indirect Path: If adding edge u to v isn't shortest, then $\delta(s, v) < \delta(s, u) + 1$.

- Graph: `s -> ... -> u -> v` and `s -> ... -> v`

3. Not Through u: Otherwise, $\delta(s, v) < \delta(s, u) + 1$.

3. If the shortest path from s to v does not pass through u, then $\delta(s, v) < \delta(s, u) + 1$.

- Graph: `s -> ... -> u` and `s -> ... -> v`

Lemma 2 of the BFS

Lemma 2: Upon termination of BFS, for all vertices v reachable from source s, $v.d \geq \delta(s, v)$, where $v.d$ is the distance assigned by BFS and $\delta(s, v)$ is the shortest path distance from s to v.

Proof:

1. Base Case: The source vertex s is the first to be enqueued and dequeued. When s is dequeued, $s.d = 0$, which is equal to $\delta(s, s)$. Thus, the base case holds.

2. Inductive Step: Assume that for some vertex u that is dequeued before v, the property $u.d \geq \delta(s, u)$ holds. Now, consider an edge (u, v). When v is first dequeued, its distance $v.d$ is set to $u.d + 1$. Given our assumption, $u.d \geq \delta(s, u)$, so $v.d \geq \delta(s, u) + 1$. Since $\delta(s, v)$ is the shortest path from s to v, $v.d \geq \delta(s, v)$.

Thus, by induction, the lemma holds for all vertices reachable from s.

Lemma 3 of The BFS

Lemma 3: Given a BFS on a graph, if vertex u is at distance d from source vertex s, and the BFS queue at some point is $< v_1, v_2, v_3, v_4, ...v_r >$ with distances $< d, d, d + 1, d + 1 >$ respectively, then:

1. v_1 and v_2 are at distance $d + 1$.
2. v_4 is a neighbor of v_1 and is at distance $d + 2$.
3. v_r is the last neighbor of v_3 and is at distance $d + 2$.

Proof:

- Vertex v was enqueued because it is adjacent to the last vertex that was dequeued (let's call this vertex u).
- Therefore, $v.d = u.d + 1$.
- By the induction hypothesis, $v.d \geq \delta(s, u) + 1$.
- By the previous lemma, $v.d \geq \delta(s, v)$.

From the above, we can conclude:

1. $v_r.d \leq v_1.d + 1$.
2. $v_i.d \leq v_{i+1}.d$.

To show: Upon termination of BFS, $v.d = \delta(s, v)$.

Proof by contradiction:

Assume a vertex v does not receive the shortest distance during BFS, i.e., $v.d > \delta(s, v)$.

1. If v is unreachable, then $v.d = \infty$ and $\delta(s, v) = \infty$. Thus, $v.d = \delta(s, v)$.
2. If v is the source vertex s, then $v.d = 0$ and $\delta(s, s) = 0$.

3. If v is reachable and not s, and it doesn't receive the shortest distance during BFS, then let u be the vertex before v in the shortest path from s. We have $u.d = \delta(s,u)$ and $\delta(s,v) = \delta(s,u) + 1$. But $v.d > \delta(s,v)$, which implies $v.d > u.d + 1$, contradicting our BFS property.

Lemma 3:
Vertex u is at the distance d from s, This is the queue for vertex $u < v_1, v_2, v_3, v_4, ...v_r >$, and at any moment in given the queue distances $< d, d, d+1, d+1 > v_1$ is $d+1$, $v2$ is $d+1$, v_4 is $d+2$, and v_r is $d+2$, assume that v_r is the last neighbor of v_3 and distance of v_r is $d+2$, and v_4 is the neighbor of v_1, the distance of v_4 is $d+2$

$v.d \geq \delta(s,v)$, assume v is the $(k+1)^{th}$ enqueued node

Why was v enqueued?
What do we know about $v.d$?

V was enqueued because it is adjacent to the last vertex that was dequeued (say u)
So v.d = u.d + 1
≥ δ(s,u) + 1 (by induction hypothesis)
≥ δ(s,v) (by previous lemma)

2 conclusions will be true in Lemma 3: $v_r \leq v_1.d + 1$, and $v_i.d \leq v_{i+1}.d$

To Show: Upon termination $v.d = \delta(s,v)$
proof by contradiction, let a vertex v does not receive a shortest distance during BFS, i.e., $v.d > \delta(s,v)$

1. If v is unreachable $v.d = \infty$, $\delta(s,v) = \infty$, $v.d = \delta(s,v)$,
2. Assume vertex v is reachable if v is s, $v.d = 0$, $\delta(s,s) = 0$
3. v is reachable, it is not s and distance receive shortest distance during BFS

Let u is vertex before v and path form s to u is shortest, $u.d = \delta(s,u)$, $\delta(s,v) = \delta(s,u) + 1$, $v.d > \delta(s,v)$, $v.d > \delta(s,u) + 1$, Assumption that $v.d > u.d + 1$

Two Possibilities

Assume dequeue v and there are vertices u, v, and w, and the vertex w is between u and v, $v.d = u.d + 1$

The vertices are being colored, `white` means the vertex is not visited, and `grey` means the vertex is visited. the `black` means the vertex and its neighbors are all explored. $v.d < u.d$, $v.d = w.d + 1$, $1 + w.d \leq u.d + 1$, and $v.d \leq u.d + 1$

White: Then v will be added to queue while processing u's adjacency list – i.e., v.d. = u.d. + 1 →←
Gray: Then v is in queue when u is dequeued – i.e., u and v were in queue simultaneously. By previous lemma v.d ≤ u.d + 1 →←
Black: v.d ≤ u.d (because v was dequeued before u – corollary to previous lemma) →←

Before The DFS LEMMAS

DFS (Depth-First Search)

- DFS explores as deep as possible along a branch before backtracking.
- When a dead-end is reached, it backtracks to find new paths.
- The process continues until all nodes are visited.
- If the graph G is connected, DFS describes a tree G_π.
  - a tree $G_\pi = (V, E_\pi)$
- If G is not connected, G_π is a **forest** (a collection of trees).
- The `Stack` for DFS, The `Queue` for BFS

DFS TIMESTAMPS & VERTEX COLORS

DFS Colors: (White, Grey, Black)

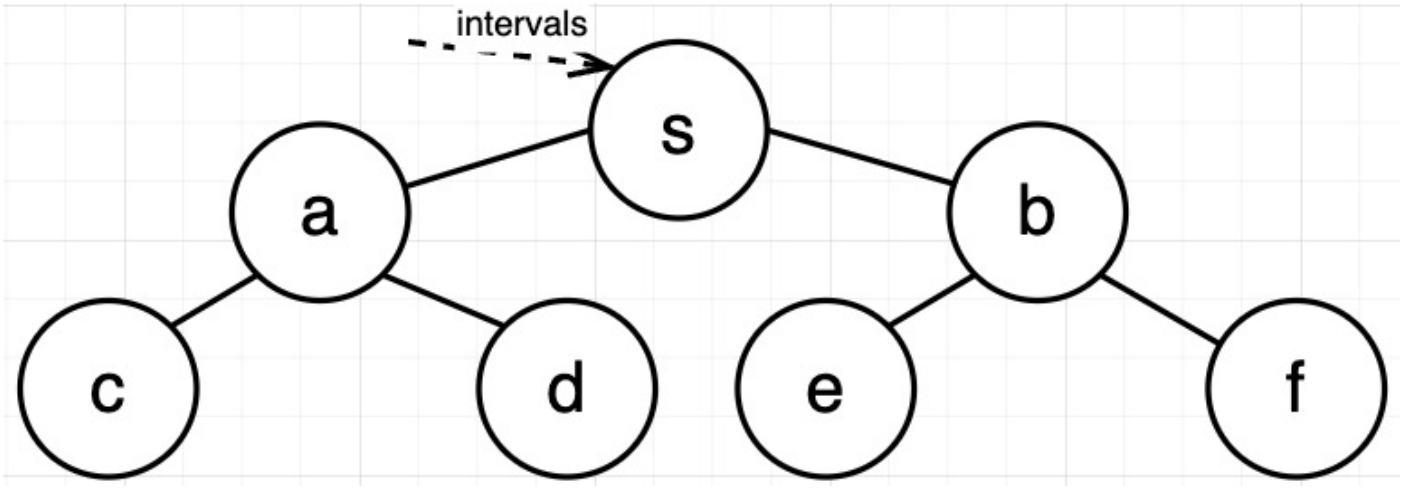
- Initially white, not visit yet
- Set to gray upon discover, visited
- Set to black when that node is finished, explored vertex and its neighbors

Timestamps: Instead of `Distance` when using the BFS

- $u.d$, $v.d =$ Time when u, v are discovered, color vertex grey
- $u.f$, $v.f =$ Time when u, v are finished, color vertex black
- Timestamps are distinct integers $1, 2, ..., 2|V|$

Relationship:

- Keeping trace the discover time, and finish time in the **DFS** Algorithm
- Time intervals when v is colored:
  - White, Gray, and Black, White: $v.d > t_{curr}$, Gray: $v.d \leq t_{curr} < v.f$ Black: $v.f \leq t_{curr}$
- What is the relationship of finish times of vertices `s`, `a`, and `c`?
  - $s.f > a.f > c.f$



```
DFS(G)
1. for each vertex u ∈ G.V
2.    u.color = white; u.π = NIL // // Initialization: Set all vertices as unvisited and without
predecessors
3. time = 0 // initialize the global timestamp
4. for each vertex u ∈ G.V  // start DFS for each unvisited vertex.
5.    it u.color == white   // the unvisited vertex becomes source vertex s,
6.        DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u)
1. time = time + 1; u.d = time    // increment the time stamp and mark the discovery time for vertex u
2. u.color = gray   // mark the vertex u as being visited
3.    for each v in G.Adj[u]    // explore each adjacent vertex of u
4.        if v.color == white
5.            v.π = u
6.            DFS-VISIT(G,v)
7.    u.color = black     // mark the vertex u as completely visited (neighbors too)
8.    time = time + 1; u.f = time   // increment the timestamp and mark the finish time for vertex u
```

- `DFS-VISIT(G,u)` is a Recursive Function.

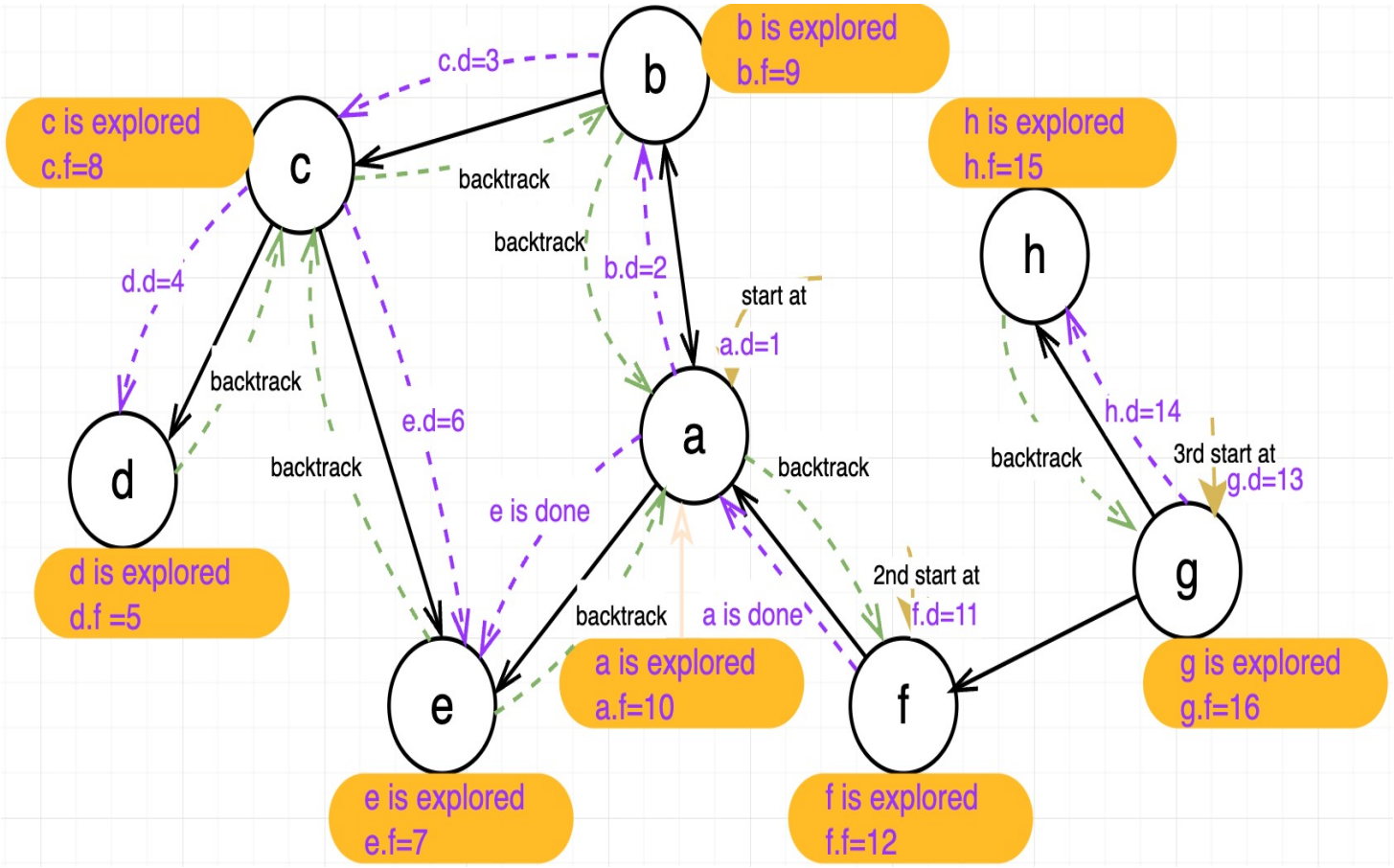
What is the run time of DFS prior to first DFS-Visit call?
Answer: $O(|V|)$ because it initializes all vertices in the graph `G`.

How many times is DFS-Visit called?
Answer: At most $O(|V|)$ times, once for each vertex in the graph `G` that hasn't been visited.

How many times is line 3 of DFS-Visit executed?
Answer: For each vertex and its adjacent vertices, so at most $O(|V| + |E|)$ times, where $|E|$ is the number of edges.

What is the total runtime of DFS?
Answer: $O(|V| + |E|)$ because every vertex and every edge is explored once.

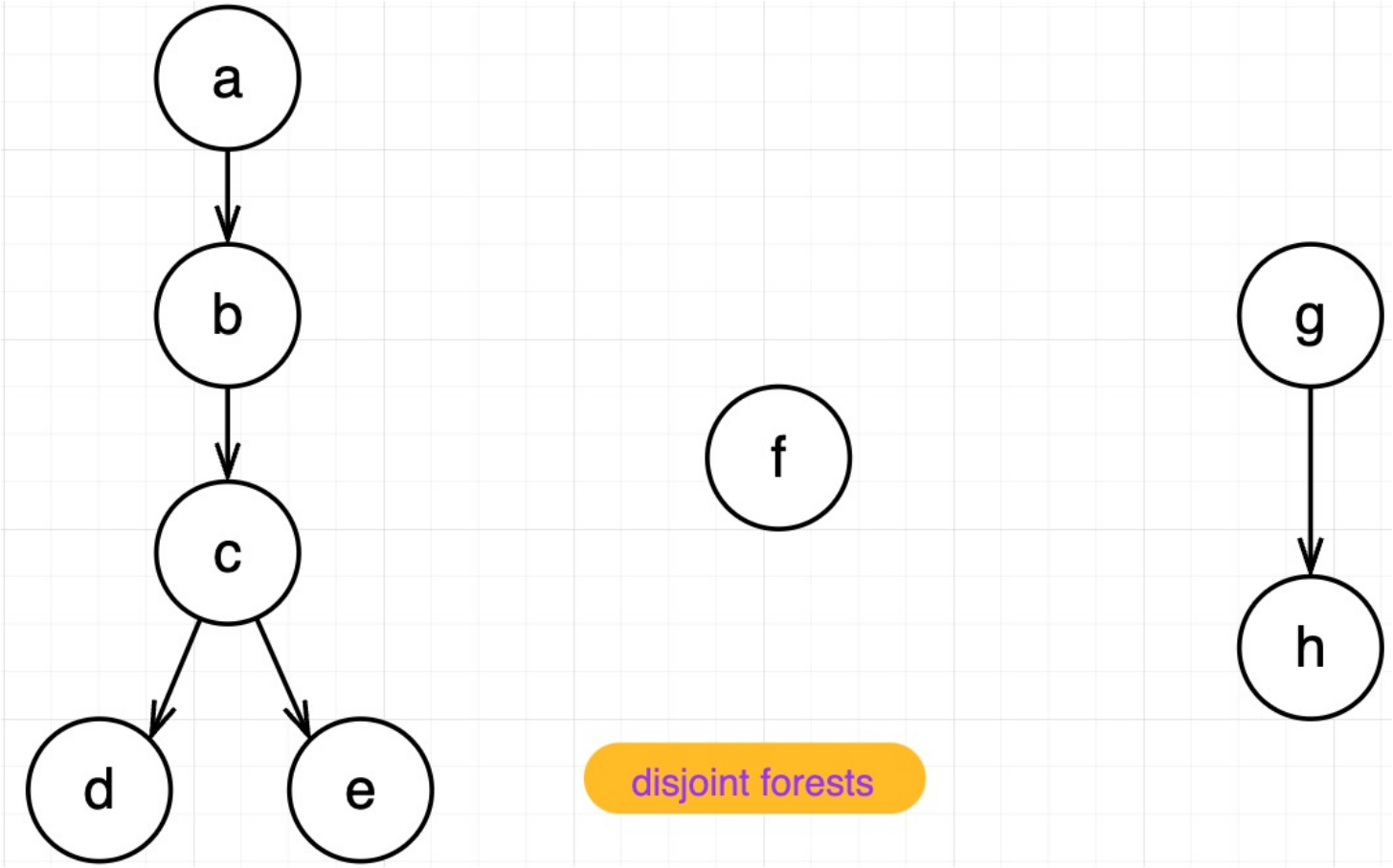
Directed Graph DFS Example:



v	v.d	v.f
a	1	10
b	2	9
c	3	8
d	4	5
e	6	7
f	11	12
g	13	16
h	14	15

What will be the $v.\pi$ looked lie?

- From the example, it has 3 starting vertices, the $v.\pi$ are **disjoint forests** as the result
- any vertex can be the starting vertex, and if pick vertex `g` , the result will be totally different

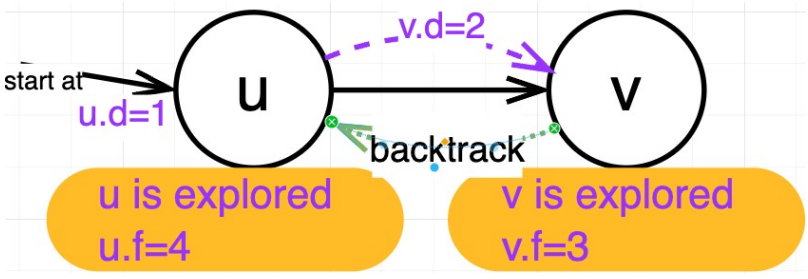


PARENTHESIS THEOREM

For any DFS on a graph $G = (V, E)$, given any vertices $u, v \in V$, consider the (TIME)intervals $[u.d, u.f]$ and $[v.d, v.f]$. One of these three conditions will always be true:

1. $[u.d, u.f]$ is entirely within $[v.d, v.f]$, and u is a descendant of v in the DFS tree.
2. $[v.d, v.f]$ is entirely within $[u.d, u.f]$, and v is a descendant of u in the DFS tree.
3. The intervals $[u.d, u.f]$ and $[v.d, v.f]$ do not overlap, meaning neither u nor v is a descendant of the other in the DFS forest.

PROOF OF PARENTHESIS THEOREM There are two main cases based on the discovery times of vertices u and v:



Case 1 ($u.d < v.d$) 情況 1 ($u.d < v.d$)

- During u's exploration, v is discovered and becomes a descendant of u in the DFS tree. v completes its exploration before u: $v.f < u.f$.
- The active interval of v, $[v.d, v.f]$, is entirely within u's active interval, $[u.d, u.f]$.

Case 1 ($u.d < v.d$):

- v is discovered during the exploration of u, making v a descendant of u in the DFS tree. v finishes before u, so $v.f < u.f$.
- u, being an ancestor of v in the DFS tree, finishes last.
- The interval $[v.d, v.f]$ during which v is active is entirely contained within the interval $[u.d, u.f]$ during which u is active.

Subcase 1: $u.d < v.d < u.f$

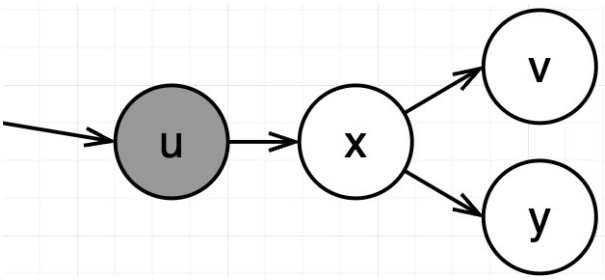
- **Vertex Color** When v was discovered, u was gray, indicating u was being visited.
- **Ancestry Conclusion:** v is a descendant of u in the DFS tree.
- **Return Order:** `DFS-Visit(v)` returns before `DFS-Visit(u)` due to the recursive nature of DFS.
- **Interval Relationship:** $u.d < v.d < v.f < u.f$

Subcase 2: $u.d < u.f < v.d$

- **Interval Relationship:** $u.d < u.f < v.d < v.f$. 區間 : $u.d < u.f < v.d < v.f$
- **Interval Conclusion:** The intervals $[u.d, u.f]$ and $[v.d, v.f]$ don't overlap.
- **Ancestry Conclusion:** Neither u nor v is a descendant of the other in the DFS tree.

WHITE PATH THEOREM

In a DFS of graph `G = (V,E)`, vertex `v` is a descendant of `u` if, at `u`'s discovery time (`u.d`), there's a white-path from `u` to `v`.



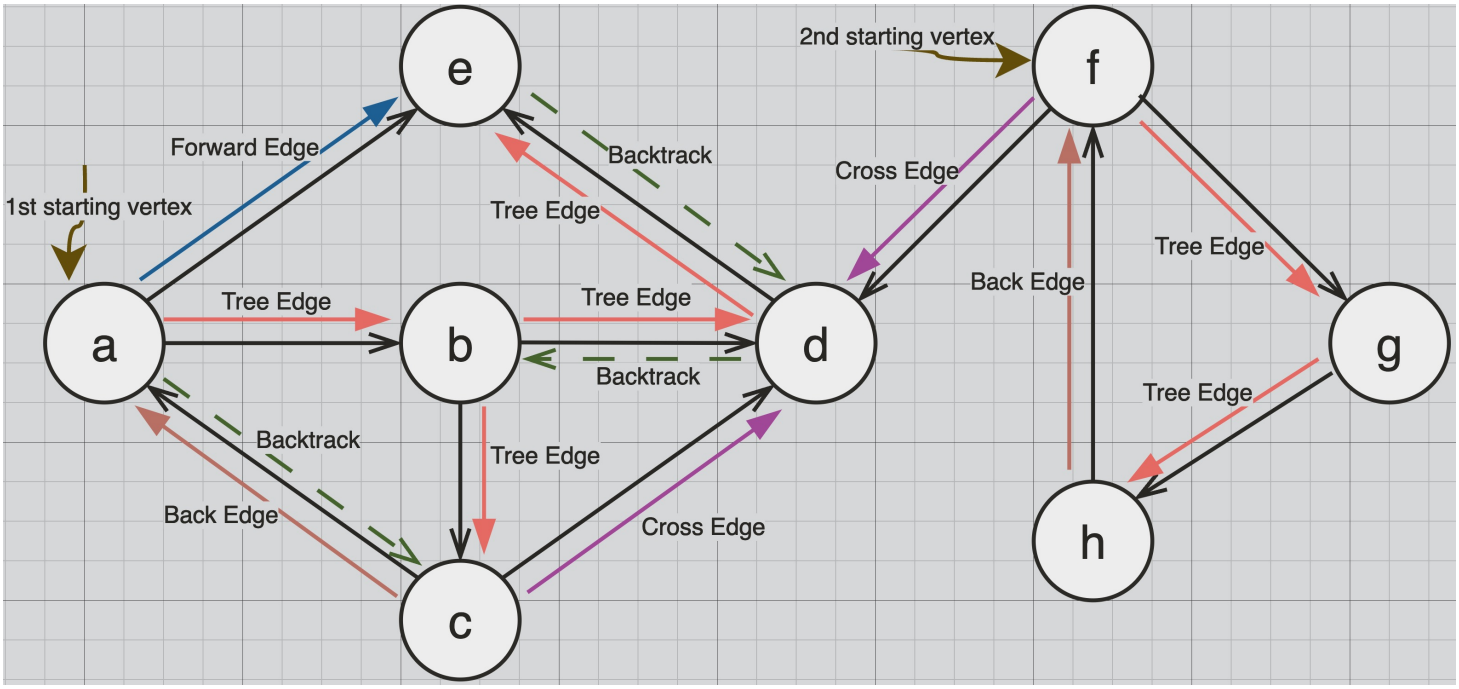
- From the graph, `u` is discovered (visited), but not fully explored, from `u` to `v` all descendant vertices should be white.

Proof White Path Theorem in 2 Parts:

1. **If v is a descendant of u in DFS:** All descendants of `u`, including `v`, are white when `u` is discovered.
2. **If a white path exists from u to v:** `v` will be explored as a descendant of `u` since it hasn't been discovered before `u`.

The presence of a white path during `u`'s discovery signifies `v`'s descendant relationship with `u` in DFS.

CLASSIFICATION OF EDGES



1. Tree Edge: (GRAY to WHITE)

- Directly connects a vertex to its descendant in the DFS tree.
- i.e., **Tree Edge** (a, b), (b, d) in the graph.

2. Back Edge: (GRAY to GRAY)

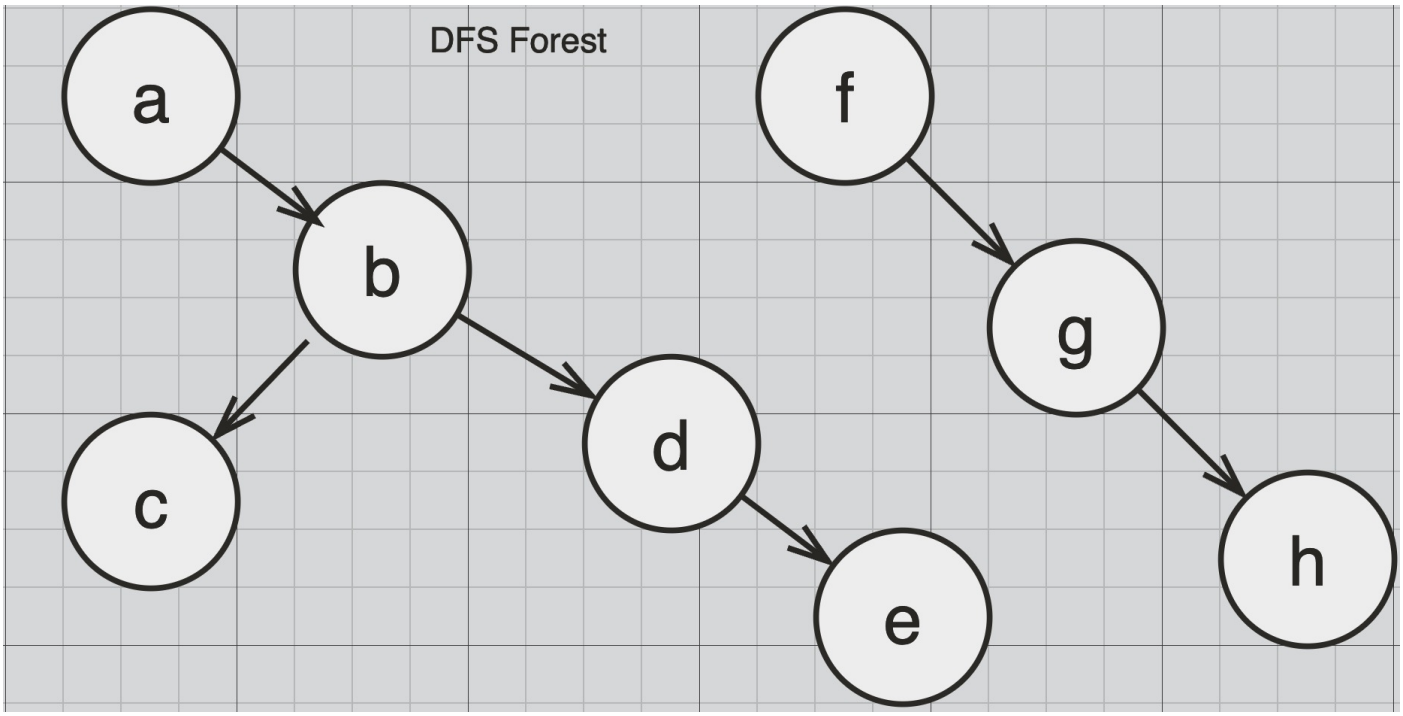
- Edges that connect a vertex to one of its ancestors in the DFS tree. This includes self-loops.
- i.e., **Back Edge** (c, b), and (h, f) in the graph.

3. Forward Edge: (GRAY to BLACK)

- Non-tree edges that connect a vertex to one of its descendants in the DFS tree.
- i.e., Forward Edge (a, e) in the graph.

4. Cross Edge: (GRAY to BLACK)

- Edges where neither vertex is a direct ancestor or descendant of the other. They can be within the same DFS tree or between different DFS trees.
- Example: Cross Edge (c, d), and (f, d)in the graph.



EDGE COLORING

In DFS, the color of vertex v upon encountering edge (u, v) determines the edge type.

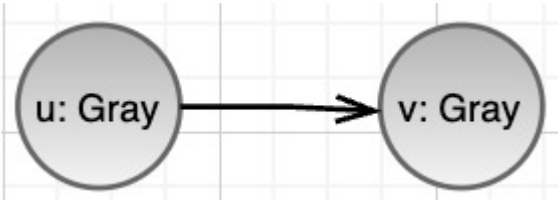
Assume u ***is in Gray*** in the DFS

1. v **is White**: Tree Edge, Directly connects a vertex to its descendant.

- $u.d < v.d < v.f < u.f$
- v is discovered and finished after u. v is a descendant in the DFS tree.
- After vertex u is discovered, vertex v is discovered and finished before u finishes. v is a descendant of u in the DFS tree.

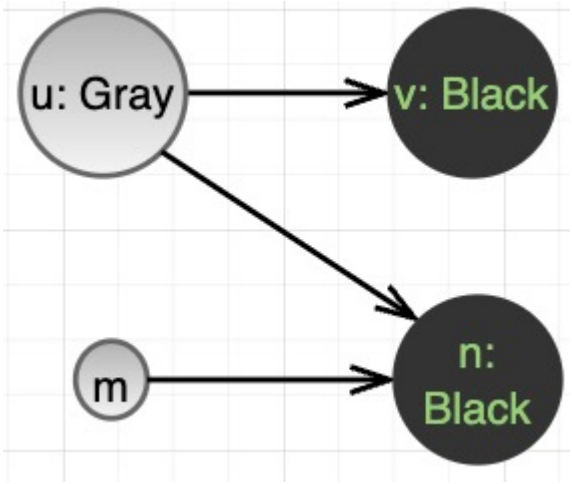
2. v **is Gray**: Back Edge, Connects a vertex to its ancestor or even itself (self-loop).

- $v.d < u.d < u.f < v.f$
- Creates a cycle as v is an ancestor in the DFS tree.



3. v **is Black**: Forward or Cross Edge

- Forward Edge: A non-tree edge that connects a vertex to a non-child descendant.
  - $u.d < v.d < v.f < u.f$ OR $u.d < v.d$ **and** $v.f < u.f$
- Cross Edge: Connects unrelated vertices, either within the same DFS tree or between different DFS trees.
  - $v.d < v.f < u.d < u.f$



Show that edge (u, v) is
a. a tree edge or forward edge if and only if u.d < v.d < v.f < u.f,

> a. u is an ancestor of v.

b. a back edge if and only if v.d ≤ u.d < u.f ≤ v.f, and

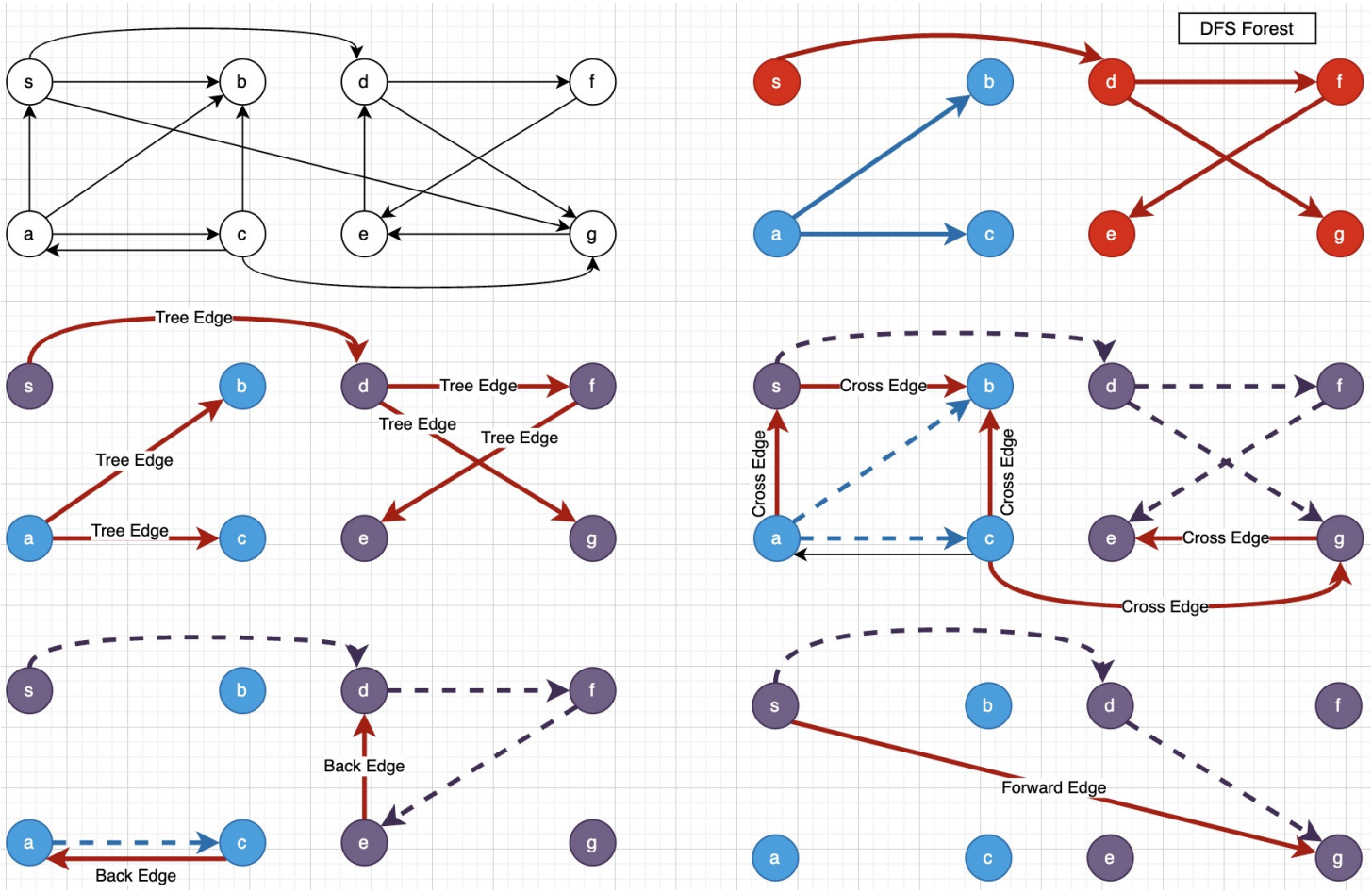
> b. u is a descendant of v.

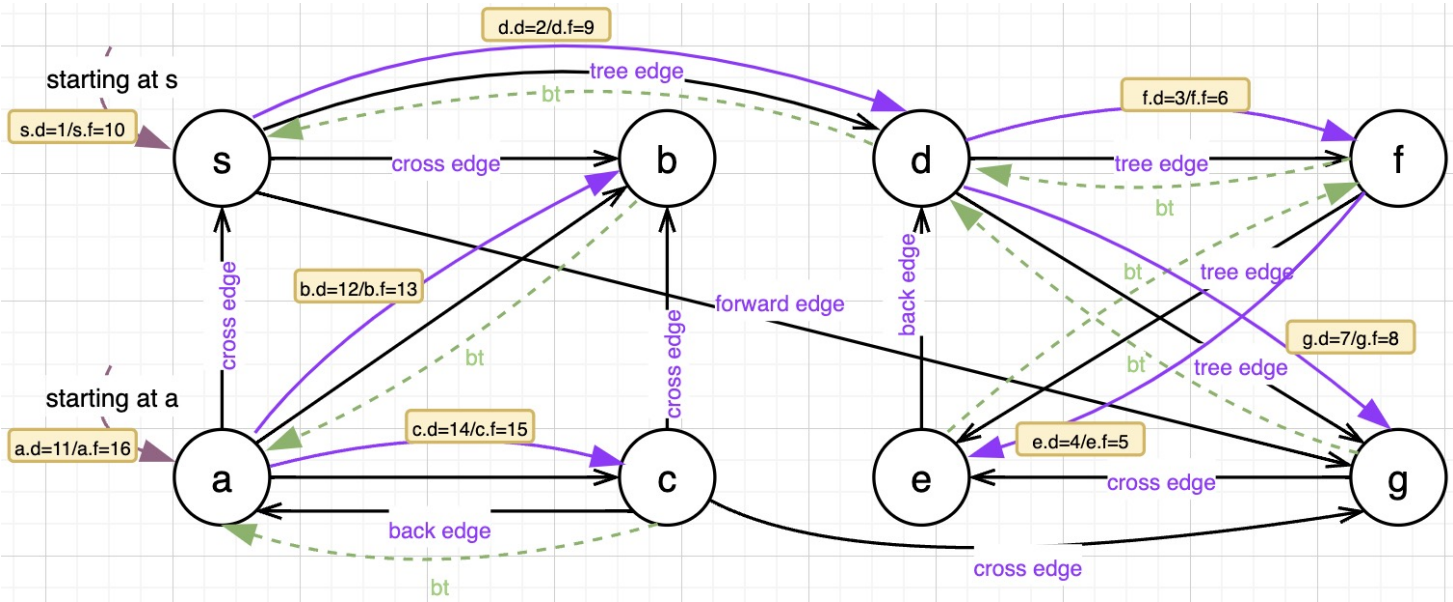
c. a cross edge if and only if v.d < v.f < u.d < u.f.

> c. v is visited before u.

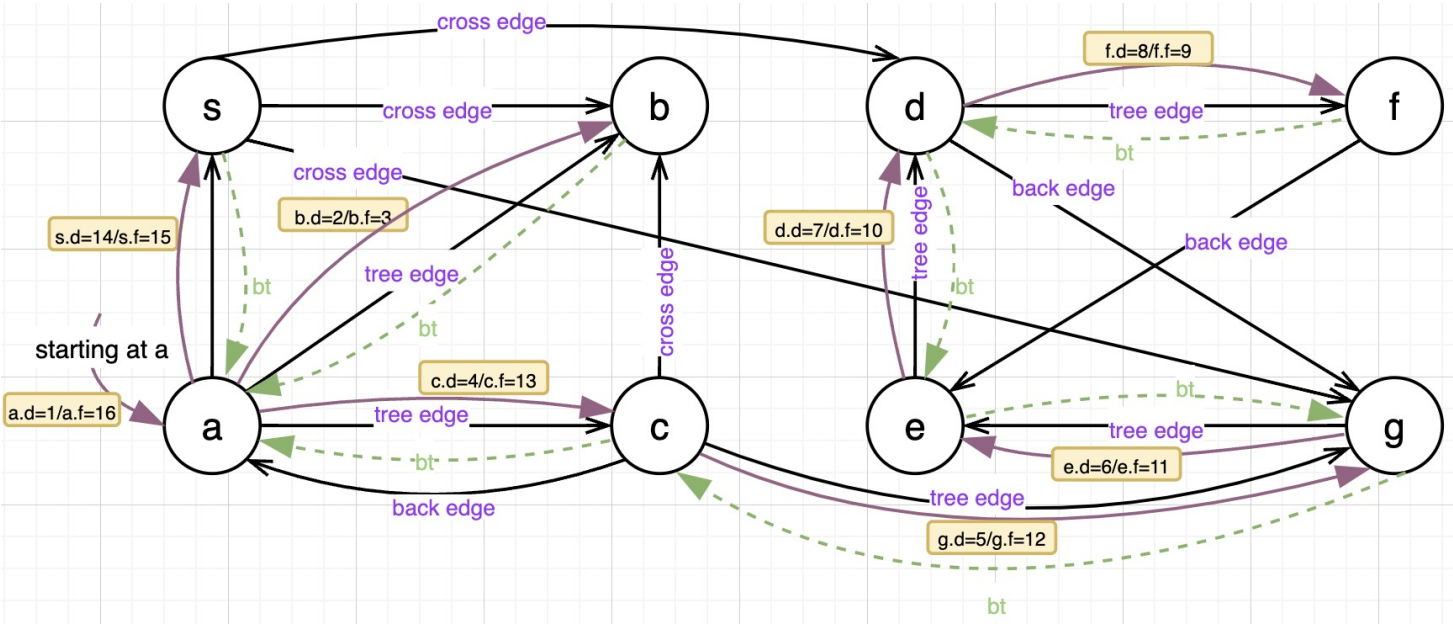
```
DFS-VISIT(G,u)
1. time = time + 1; u.d = time    // increment the time stamp and mark the discovery time for vertex u
2. u.color = gray   // mark the vertex u as being visited
3.    for each v in G.Adj[u]    // explore each adjacent vertex of u
4.        if v.color == white
5.            v.π = u
6.            DFS-VISIT(G,v)
7.    u.color = black     // mark the vertex u as completely visited (neighbors too)
8.    time = time + 1; u.f = time   // increment the timestamp and mark the finish time for vertex u
```

Example DFS Edges:


DFS Forest
Tree Edge
Cross Edge
Back Edge
Forward Edge

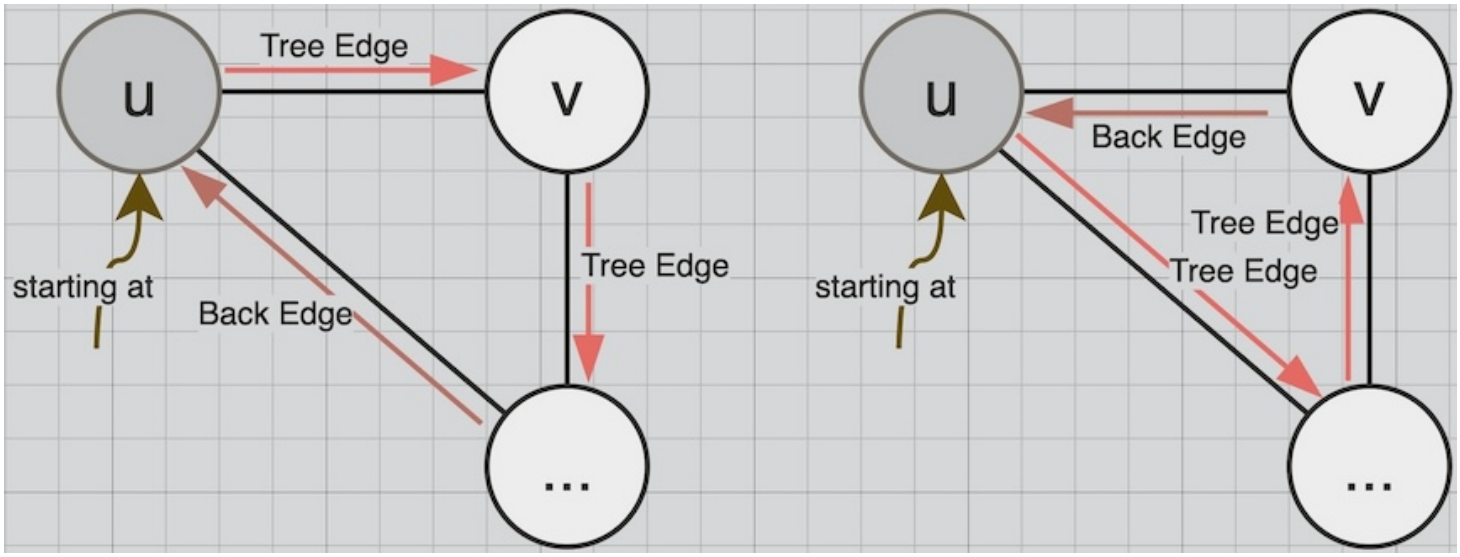

d.d=2/d.f=9
starting at s
s.d=1/s.f=10
tree edge
bt
cross edge
f.d=3/f.f=6
b.d=12/b.f=13
forward edge
back edge
g.d=7/g.f=8
starting at a
a.d=11/a.f=16
c.d=14/c.f=15
e.d=4/e.f=5
back edge
cross edge

Solution 2:



(a, b) - Tree Edge, (a, c) - Tree Edge, (c, g) - Tree Edge, (g, e) - Tree Edge
(e, d) - Tree Edge, (d, f) - Tree Edge, (a, s) - Tree Edge, (f, e) - Back Edge, (d, g) - Back Edge, (c, a) - Back Edge, (c, b) - Cross Edge, (s, b) - Cross Edge, (s, d) - Cross Edge, (s, g) - Cross Edge

UNDIRECTED GRAPHS



Theorem (Specific to DFS): In a DFS traversal of an undirected graph, every edge is either a **TREE EDGE** or a **BACK EDGE**.

What does this mean with respect to edge colors in DFS?

1. **White**: Vertex not discovered.
2. **Gray**: Vertex discovered but not finished.

3. **Black**: Vertex finished.

Proof:

Let's consider an edge (u, v) in E. Assume $u.d$ and $v.d$ represent the discovery times of vertices u and v respectively, and $u.d < v.d$.

- **Claim**: The discovery and finish times of the vertices satisfy $u.d < v.d < v.f < u.f$.

There are two possibilities:

1. $v.\pi = u$**:** This means vertex v is a direct descendant of u in the DFS tree. The edge (u, v) is a tree edge.
2. $v.\pi \neq u$**:** This means vertex v is not a direct descendant of u. The edge (u, v) is a back edge.

DFS APPLICATIONS

Two Applications of DFS: **Topological Sort**, and **Strongly Connected Components**

Topological Sort

It's a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v.

In simpler terms: Imagine you have tasks and some tasks must be done before others. A topological sort provides an order to complete tasks without violating any prerequisites.

Topological Sorting in DAGs

1. **Starting Vertex for DFS**: In topological sorting of a DAG, initiate DFS at vertices with no incoming edges, as they precede others in the ordering.
2. **General DFS**: Starting vertex in a general DFS can be any vertex, not confined to DAGs.
3. **Purpose of Topological Sorting**: To linearly order vertices so that for each directed edge (u, v), u precedes v.
4. **Specificity to DAGs**: Starting from vertices with no incoming edges is a strategy tailored for topological sorting in DAGs, not a general DFS rule.

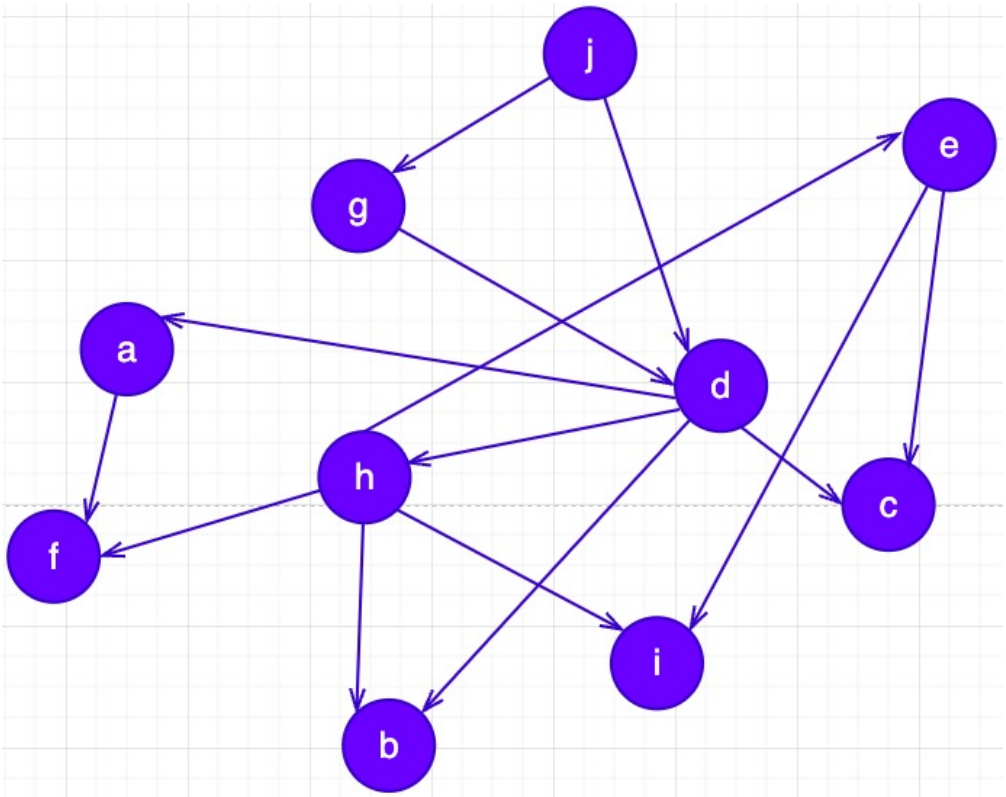
By adding these notes to your collection, you should have a clear and concise reference for understanding the relationship between DFS and topological sorting in the context of Directed Acyclic Graphs.

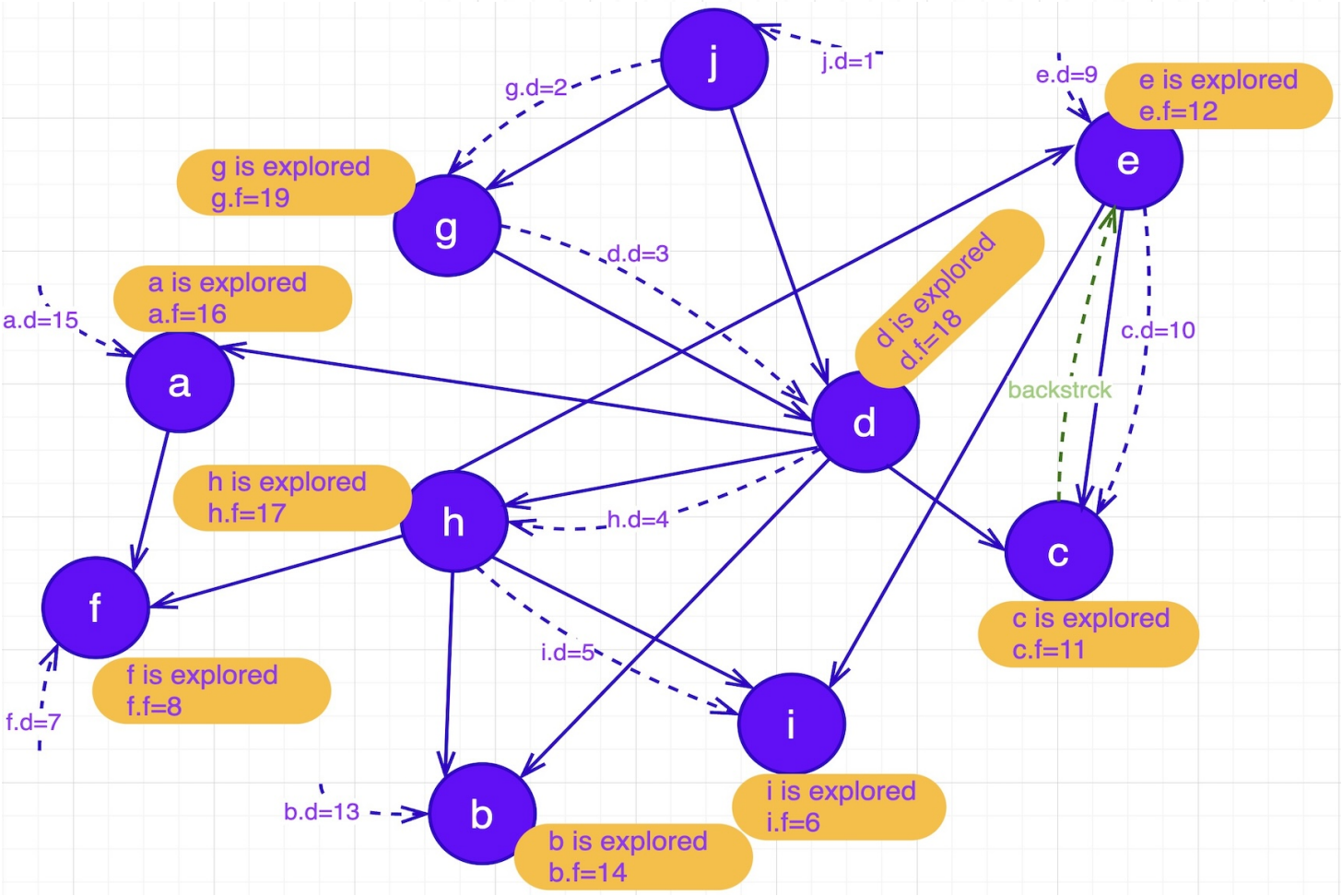
Topological Sort Example:

Here is a list of things I usually do when I get home (in alphabetical order)

- `list = {a. Close door, b. Feed dogs, c. Get a glass of water, d. Get inside, e. Give dogs treats, f. Lock door, g. Open door, h. Put down bags, i. Take off coat, j. Unlock door}`
- and vertices `v = {a, b, c, d, e, f, g, h, i, j}`

Draw a graph indicating which things must be done before others





TOPOLOGICAL SORT(G)

1. Call `DFS(G)`
2. As each vertex is finished, insert it into the front of a linked list

3. Return the linked list

	v	v.d	v.f
a. Close Door	a	15	16
b. Feed Dogs	b	13	14
c. Get a Glass of Water	c	10	11
d. Get Inside	d	3	18
e. Give Dogs Treats	e	9	12
f. Lock Door	f	7	8
g. Open Door	g	2	19
h. Put Down Bags	h	4	17
i. Take off Coat	i	5	6
j. Unlock Door	j	1	20

- `j - g - d - h - a - b - e - c - f – i`
- `Unlock door, Open door, Get inside, Put down bags, Close door, Feed dogs, Give dogs treats, Get glass of water, Lock door, Take off coat`

DAG Topological Sort

- A `Directed Acyclic Graph` **DAG**, meaning it has **NO CYCLES** and **NO BACK EDGES**.
- **Topological Order**: A linear ordering of vertices such that for every directed edge (u, v), vertex u precedes v.

Procedure:

1. **DFS**: Perform a depth-first search to compute finish times for each vertex.
2. **Sort**: Sort the vertices in decreasing order of finish time.

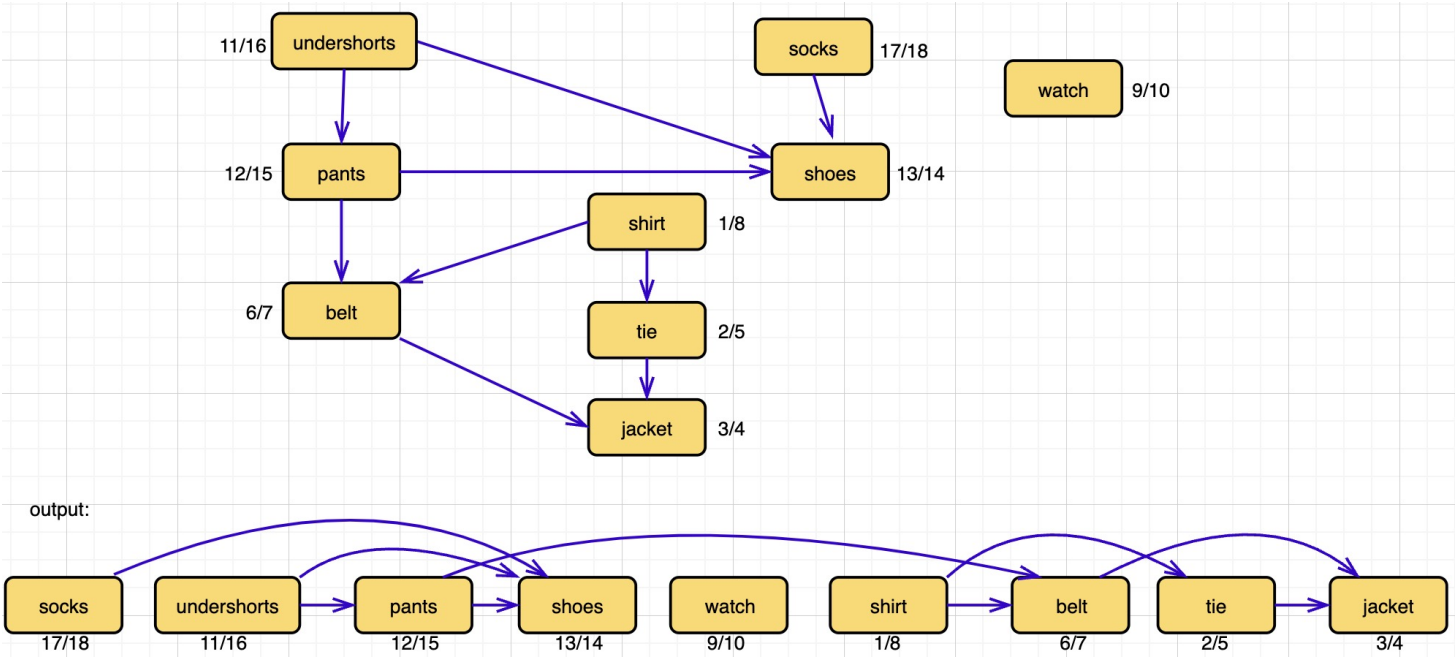
Topological Sort of Vertices Example:

- **Professor Bumstead's Clothing**: Garments are vertices, and edges represent dressing constraints.
- **Result**: A valid dressing order that respects all constraints.

(a) Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finish times from a depth-first search are shown next to each vertex.

(b) The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finish time. All directed edges go from left to right.

Figure 20.7:



Advantages of Topological Sort:

- **Efficiency**: Quickly find a valid order in $O(V + E)$ time.
- **Applicability**: Useful in scheduling, dependency resolution, and more.

LEMMA

A directed graph G is acyclic iff a DFS of G yields no back edges
G acyclic = no back edges
No back edges = G acyclic

⇒ (proof by contradiction)
Assume DFS yields a back edge (u,v)
Then v is an ancestor of u in the DFF (v.d < u.d < v.f)
So G contains a cycle v ⇝ u → v
⇐ (proof by contradiction)
Assume G contains a cycle c
Let v be the first vertex in c discovered and u be v's predecessor in c v ⇝ u → v
At time v.d all vertices in c are white, so all vertices in c are descendants of v
In particular, v.d < u.d < u.f < v.f
At time u.d, v will be gray, so (u,v) is a back edge, **DAG can't have a back edge**

CORRECTNESS OF TOPOLOGICAL SORT

TOPOLOGICAL-SORT produces a topological sort of any directed acyclic graph.
Suffices to show that v.f < u.f for any (u,v) € E Let (u.v) be any edge in E
• Can there be a path from v to u? v is white or black at time v.d (by prev. lemma)
• v is white, v is descendant of u, hence u.d < v.d < v.f < u.f$$

• v is black

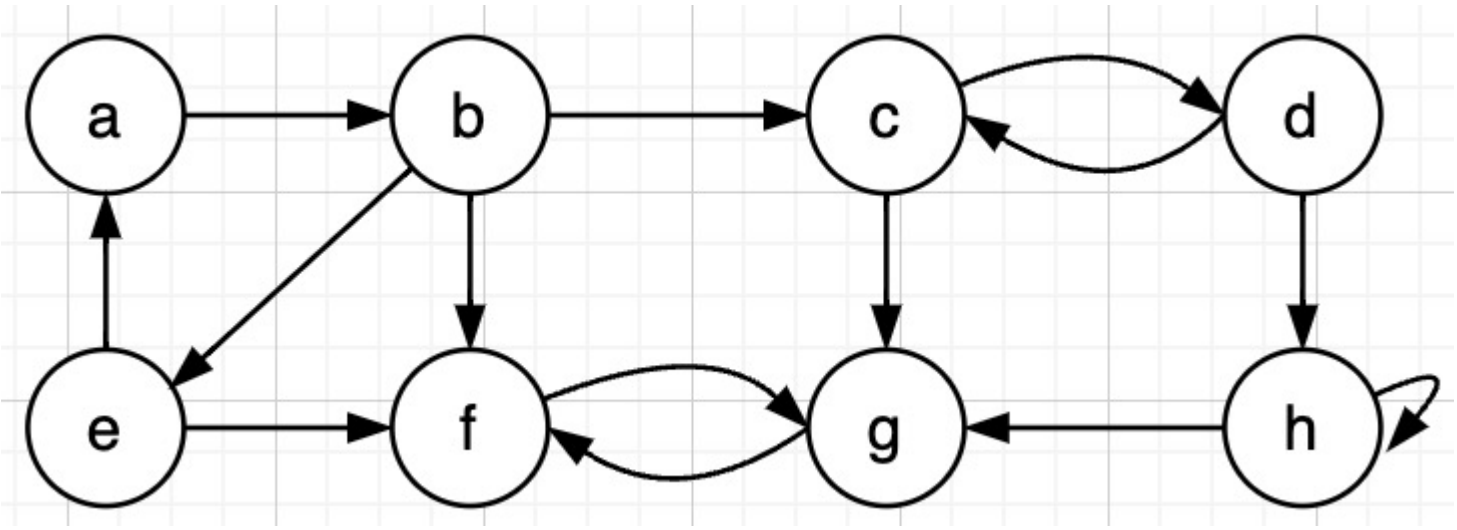
(u,v) ∈ E
No path v ⇝ v exists because acyclic
v can be Black or white (by previous lemma)
v is white: v is a descendant of u, so u.d < v.d < v.f < u.f
v is black: v.f < u.d < u.f, v is already finished $v.f < u.f$

STRONGLY CONNECTED COMPONENTS

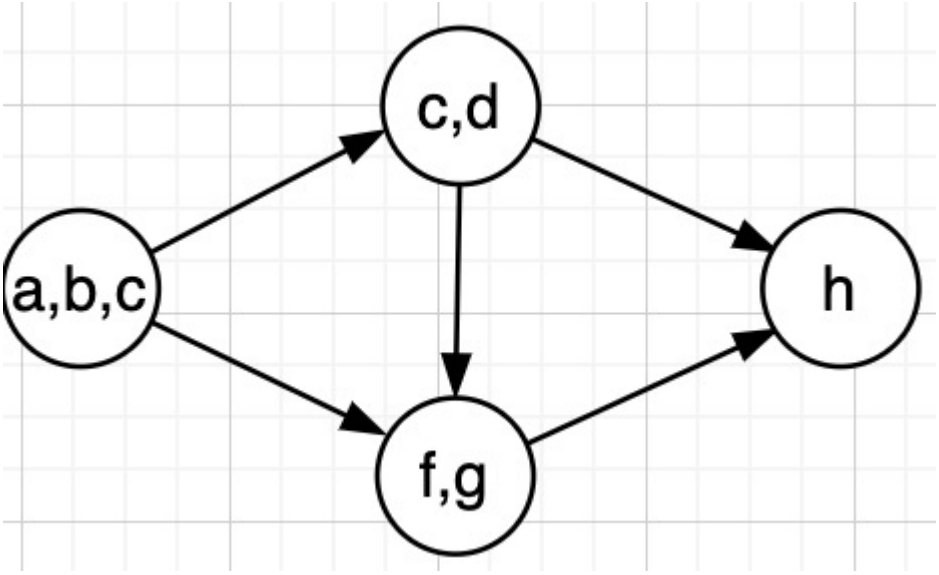
- A **Strongly Connected Component (SCC)** in a **Directed Graph** $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, there is a directed path from u to v and from v to u, ensuring internal connectivity within the subgraph formed by C.
- Find all **Subgraphs** in G, Vertices u and v are in the same SCC if and only if there's a path from u to v and a path from v to u in G.
- **Transpose Graph:** For any graph $G = (V, E)$, its transpose $G^T = (V, E^T)$ is formed by reversing the direction of all its edges, where $E^T = (v, u)|(u, v) \in E$. Both G and G^T share the same SCCs.

Vertices u and v are in a strongly connected component iff there is a path from u to v and from v to u
• Let $E^T = (v, u)|(u, v) \in E$

For any graph $G = (V, E)$, consider the graph $G^T = (V, E^T)$
• Note G and in G^T will have the same strongly connected components



- Strongly Connected Components



STRONGLY CONNECTED COMPONENTS aka. SCC

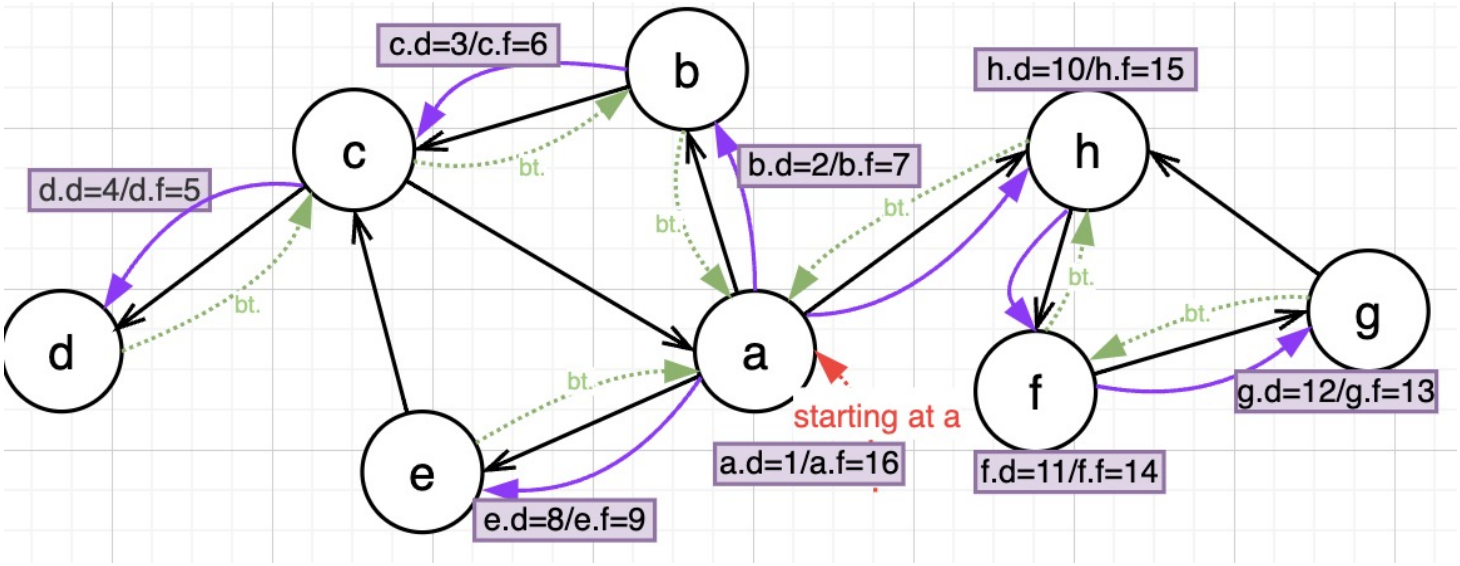
```
SCC(G)
```

1. Call First $DFS(G)$ creating finishing times $u.f$ for each vertex u
2. Compute G^T
3. Call Second $DFS(G^T)$, processing vertices in main loop in order of decreasing $u.f$
4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Idea of SCC Algorithm with Example

First DFS on G

- Arranges vertices in a sort of “topological” order
- Not a true topological sort because G contains cycles



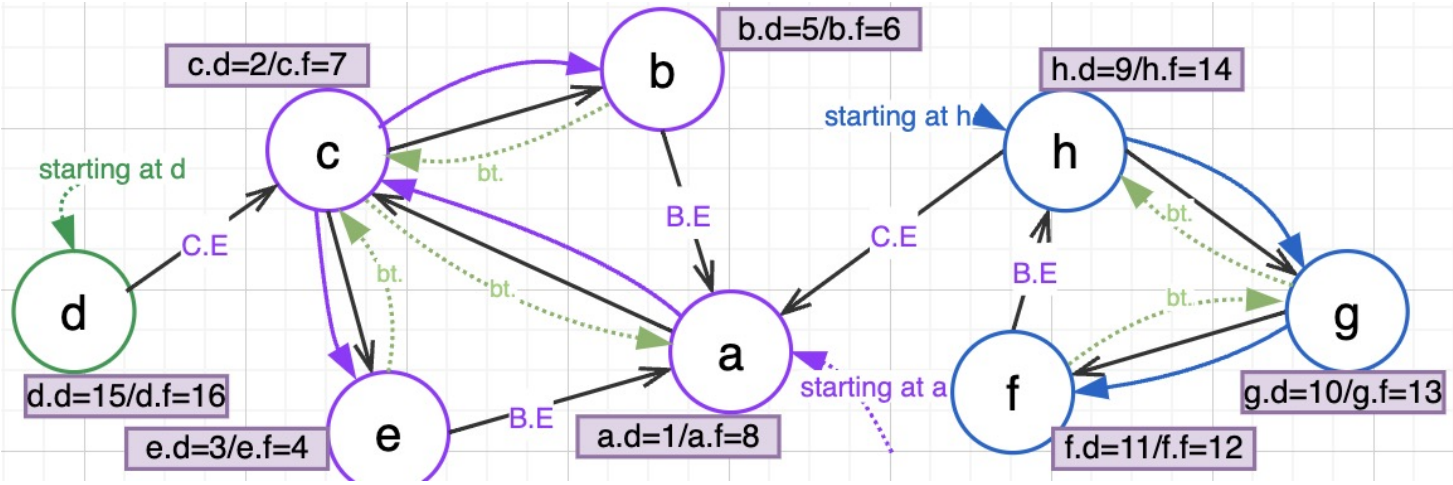
- After running the $DFS(G)$, Decreasing $u.f$ order of $DFS(G)$

- `a, h, f, g, e, b, c, d` , check with the graph

v	$v.d$	$v.f$
a	1	16
b	2	7
c	3	6
d	4	5
e	8	9
f	11	14
g	12	13
h	10	15

Second DFS on G^T

- Finds nodes in G^T connected to each root node
  - `a` , `d` , `h` are the root nodes
- Because of sort order, v in tree rooted at $u = u \rightarrow v$ and $v \rightarrow u$



- Compute G^T

v	$v.d$	$v.f$
a	1	8
b	5	6
c	2	7
d	15	16

v	$v.d$	$v.f$
e	3	4
f	11	12
g	10	13
h	9	14

- The G^T contains 3 Strongly Connected Components
  - First SCC: `a, b, c, e` ? `a, c, b, e`
  - Second SCC: `f, g, h` ? `h, g, f`
  - Third SCC: `d`
- $V^{SCC} = a, d, h, E^{SCC} = (a, d), (a, h)$