CSCI 4470 Algorithms

Data Structures for Disjoint Sets Notes

Chapter 19: Data Structures for Disjoint Sets

Introduction of Disjoint Sets

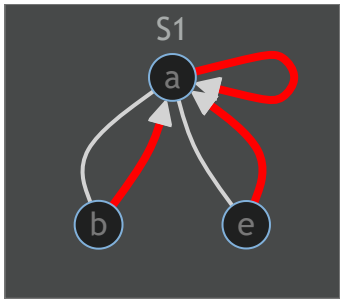
For graph algorithms, we sometimes need to manage disjoint sets of **`vertices`** or **`edges`**

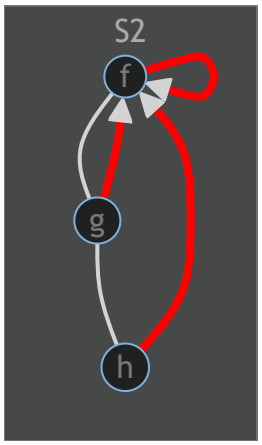
- Disjoint Set aka. `union-find`
- Used in graph algorithms. 用於圖形算法。
- Known as a disjoint data structure. 稱為不相交的數據結構。

DISJOINT SET DATA STRUCTURE

不相交集數據結構

- 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



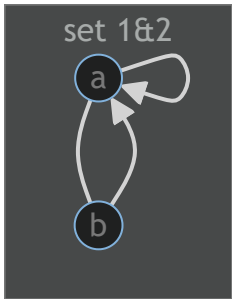


THREE OPERATIONS

1. MAKE-SET(x) Operation: Creates a disjoint set S. If starting with a single element A, A is its own representative.
創建一個不相交的集合 S。如果從單個元素 A 開始，則 A 是其自己的代表。

- `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.
合併兩個集合。如果合併 S_1 和 S_2，它們應該有一個共同的代表。



- `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. 返回集合的代表。

For example, for **Set** $1 \cup 2$, it returns its representative which is `a` .
例如，對於集合 $Set\ 1 \cup Set\ 2$，它返回其代表 `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:

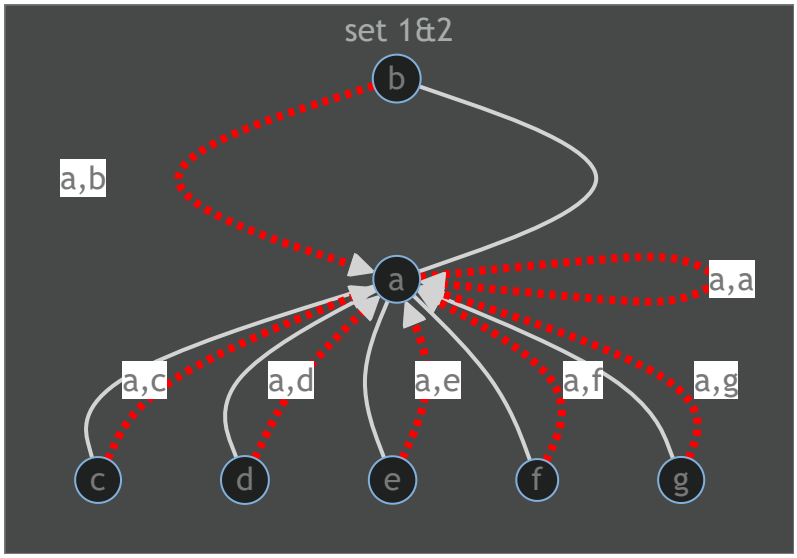
- A graph can have multiple components.
- BFS and DFS can find connected components but are less efficient, Disjoint Set can be simpler than BFS/DFS
- Disjoint sets are more efficient for this task. (merging components in near **CONSTANT TIME**)

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.
  不相交森林, 不相交森林是一組樹的集合，其中每棵樹代表一個集合。樹中的每個節點代表一個元素，樹的根代表集合的代表。

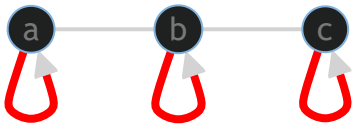
Implementations

實現方法

- **1. Linked List**: 鏈表
- **2. Disjoint Forest**: 不相交森林

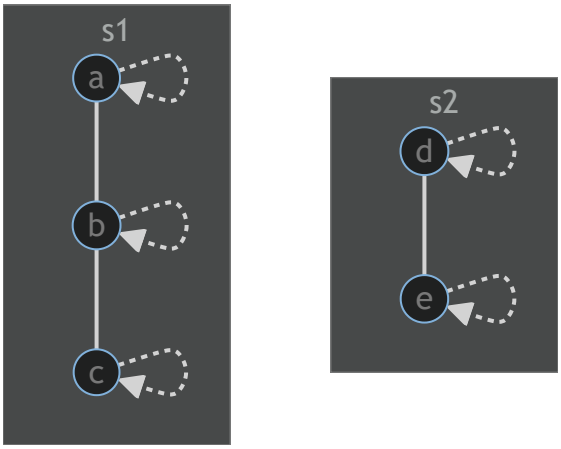
1. Linked List: 鏈表

- Each node points to its representative. 每個節點指向其代表。
- Operations can be linear. 操作可以是線性的。
- Uses many pointers. 使用許多指針。
- e.g., A set represents in Linked List.

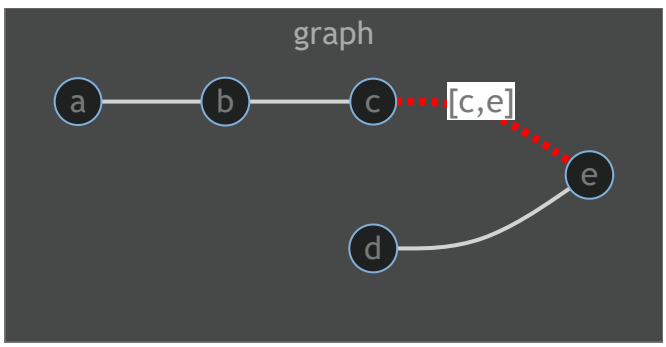


2. Disjoint Forest: 不相交森林

Initialization each element is its own set,



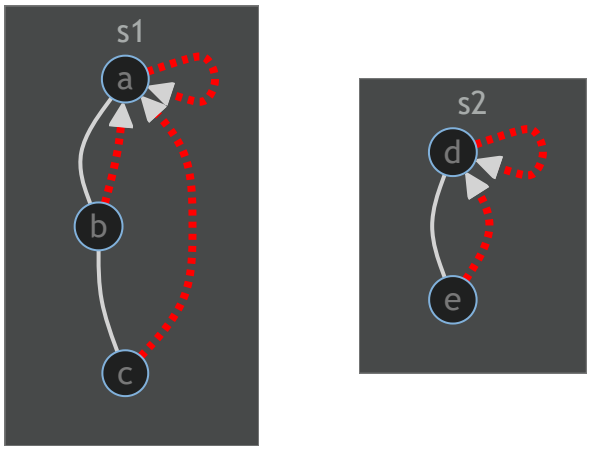
Graph Example, Assume added the `EDGE(c,e)`, the graph as shown below:



- The graph has two Disjoint Sets

- Trees where each node points to its parent.
  每個節點指向其父節點的樹。
- 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

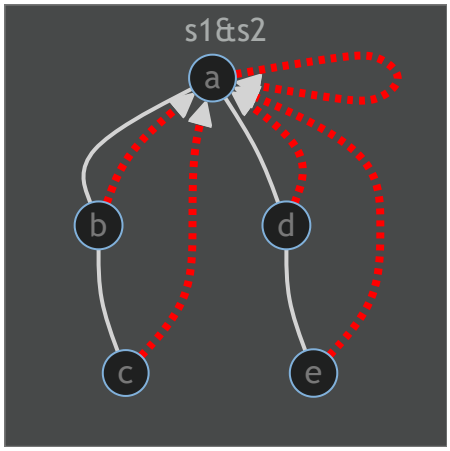


FIND-SET(e), After the execution of `UNION()`, the representatives of sets `s1` and `s2` can be determined

- 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.
  路徑壓縮：找到代表後，更新指針以直接指向根。
- **Union by Rank: Merge smaller tree into the bigger tree to minimize pointer updates.**
  按等級合併：將較小的樹合併到較大的樹中以最小化指針更新。
- `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.
  等級：樹的高度的上限。當合併兩個相同等級的集合時，結果集的等級增加一。
- 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

問題解決策略

1. Path Compression: 路徑壓縮

- After running `FIND-SET(x)`, all nodes between x and the root will now point to the root.
  執行 `FIND-SET(x)` 之後，x 和根之間的所有節點現在都將指向根。

2. Union by Rank: 按等級合併

- `UNION(x,y)` makes the smaller rank set point to the representative of the higher rank set.
  `UNION(x,y)` 使較小的等級集指向較高等級集的代表。
- If both have the same rank, the rank is incremented.
  如果兩者都具有相同的等級，則等級增加。

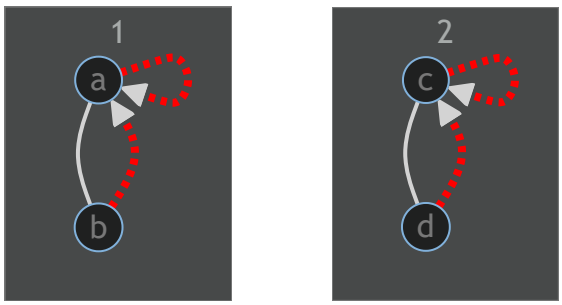
EXAMPLES of Disjoint Sets

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**

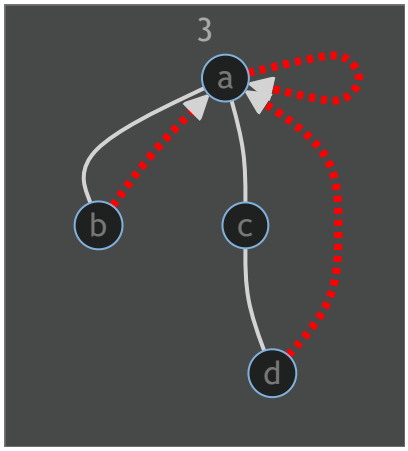
Example 5 - 6: `UNION(a,b)`, `UNION(c,d)`

- 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 圖形定義:
A graph is represented by a set of vertices and edges. 圖形由一組頂點和邊緣表示。
$G = (V, E)$

- V = set of vertices $= \{v_1, v_2, ..., v_n\}$ V = 頂點集合 $= \{v_1, v_2, ..., v_n\}$
- E = set of edges $\subseteq V \times V$ E = 邊緣集合 $\subseteq V \times V$

Edge Interpretation 邊的解釋

- $(v_i, v_j) \in E$ means there's a single step from v_i to v_j. $(v_i, v_j) \in E$ 表示從 v_i 到 v_j 有一個單步。
- For an **Undirected Graph G**, if $(v_i, v_j) \in E$, then $(v_j, v_i) \in E$ also holds true. 對於**無向圖 G**，如果 $(v_i, v_j) \in E$，那麼 $(v_j, v_i) \in E$ 也成立。

Graph Characterization:

- **Sparse Graph**: A graph where the number of edges is much less than the maximum possible number of edges. Mathematically, $|E| << |V|^2$. **稀疏圖**: 一個圖，其中邊的數量遠少於可能的最大邊數。數學上，$|E| << |V|^2$。
  - 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, $|E| \approx |V|^2$. **密集圖**: 一個圖，其中邊的數量接近於可能的最大邊數。數學上，$|E| \approx |V|^2$。

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) 圖 G=(V, E)

- **V (Vertices)**: The set of nodes or points in the graph. V (頂點): 圖中的節點或點的集合。
- For the given graph G, `V = {a, b, c, d}`. 對於給定的圖 G，V = {a, b, c, d}。
- **E (Edges)**: The set of lines connecting the vertices, representing relationships or connections. E (邊): 連接頂點的線的集合，代表關係或連接。
- For the given graph G, `E = {(a, b), (b, c), (c, d), (d, a)}`. 對於給定的圖 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}`

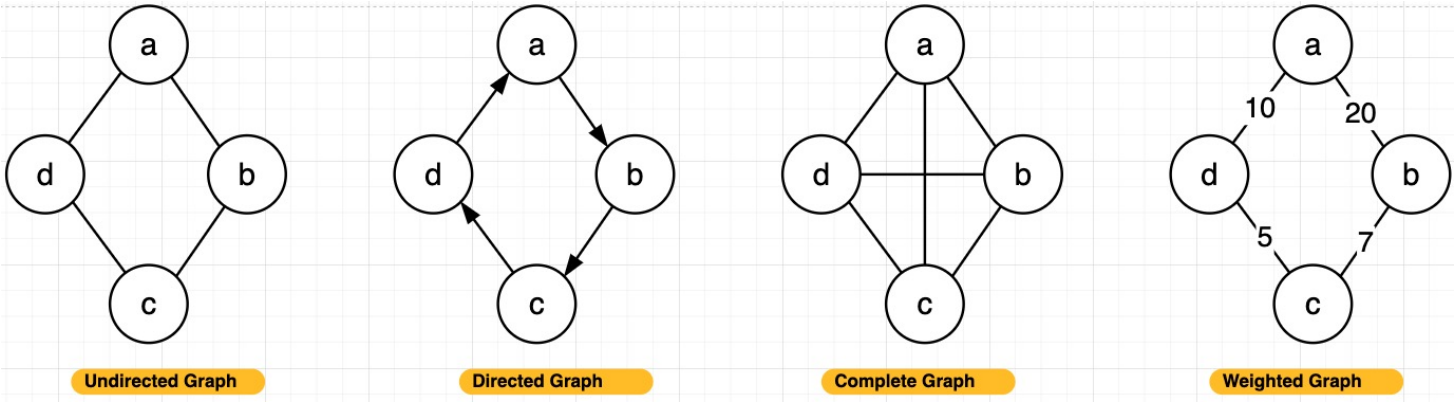
E (Edges) `E = {(a,b)(b,c)(c,d)(d,a)(a,c)(b,d)}`

- $Edge \propto |V|^2$, 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)}`
- $20, 7.5, 10$ are the weights of the edges. 這裡，$20, 7, 5, 10$ 是邊的權重。



Edges with Associated Values:

- Edges in a graph may have an associated value, indicating some measure or quantity. 圖中的邊緣可能具有相關的值，表示某種尺寸或數量。
- This value could represent distance, cost, time, or any other measurable factor. 此值可以代表距離、成本、時間或任何其他可測量的因素。
- **Edge Weight**: In cases where edges have an associated value, they are termed as 'weighted' with the value denoted as $w(u, v)$. 在邊具有相關值的情況下，它們被稱為'加權'，值表示為 $w(u, v)$。
  - For an edge from vertex u to vertex v, its weight is represented as $w(u, v)$. 對於從頂點 u 到頂點 v 的邊，其權重表示為 $w(u, v)$。

Representation in Adjacency List:

- In an adjacency list representation of a weighted graph, a weight component is added to each list element. 在加權圖的鄰接表表示中，每個列表元素都增加了一個權重組件。
- The list for a vertex can have pairs (neighbor, weight) indicating the neighbor vertex and the weight of the edge connecting them. 頂點的列表可以有對（鄰居、權重）指示鄰居頂點和連接它們的邊的權重。

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. 在加權圖的鄰接矩陣表示中，`[i][j]` 處的矩陣條目將表示頂點 i 和頂點 j 之間的邊的權重。
- 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. 如果從頂點 i 到頂點 j 有權重為 w 的邊，那麼 `[i][j]` 處的矩陣條目將是 w。如果沒有邊，該條目可能是無窮大或一個大值，具體取決於上下文。

Adjacency List Representation

鄰接表表示

- Represents a graph as an array of linked lists. 將圖表示為鏈表的數組。
- Array index denotes a vertex; linked list elements indicate vertices forming an edge with it. 數組索引表示頂點；鏈表元素指示與其形成邊的頂點。

Class Example: Adjacency List with Weighted and Undirected Graph

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

Advantages:

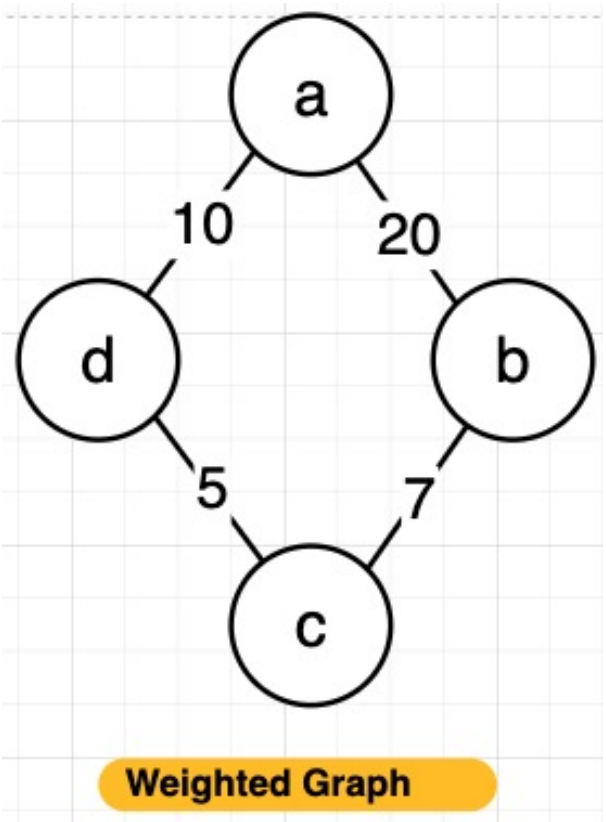
- Space-efficient for sparse graphs. 對於稀疏圖而言，空間效率高。
- Facilitates finding all neighbors of a vertex. 容易找到一個頂點的所有鄰居。

Drawbacks:

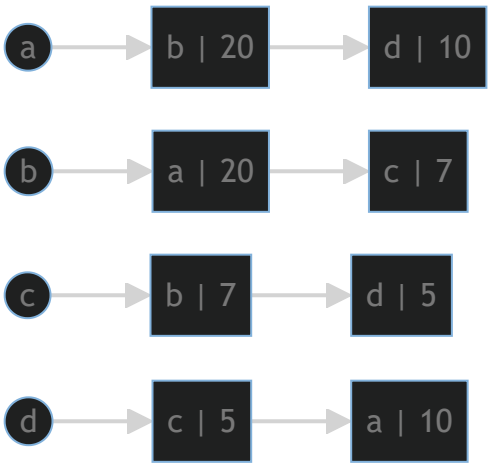
- Less efficient for dense graphs or checking the existence of an edge between two nodes. 對於密集圖效率較低，或確定兩節點之間是否存在邊。

For **sparse graphs**: 對於稀疏圖

- **Adjacency List** is preferred due to space efficiency proportional to $|V| + |E|$. 由於其空間效率與 $|V| + |E|$ 成正比，所以首選**鄰接列表**。
  - **the complexity is** $O(V + E)$



↓ **Vertices** → **Edges**



Adjacency Matrix Representation

Adjacency Matrix, A square matrix used to represent a graph. 用於表示圖的正方形矩陣。

- **Pros**: Provides a quick way to check if an edge exists between two nodes. Good for dense graphs.提供了一種快速檢查兩節點之間是否存在邊的方法。適用於密集圖。
- **Cons**: Takes up more space, especially for sparse graphs. Less space efficient than adjacency lists for such graphs. 佔用更多空間，尤其是對於稀疏圖。對於這種圖而言，空間效率不如鄰接列表。

For **dense graphs**: 對於密集圖

- **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. 鄰接矩陣受到青睞，因為它提供常數時間 $O(1)$ 的邊查找，並使用與 $|V|^2$ 成正比的空間，這接近於密集圖中的邊數。
  - 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

選擇列表與矩陣

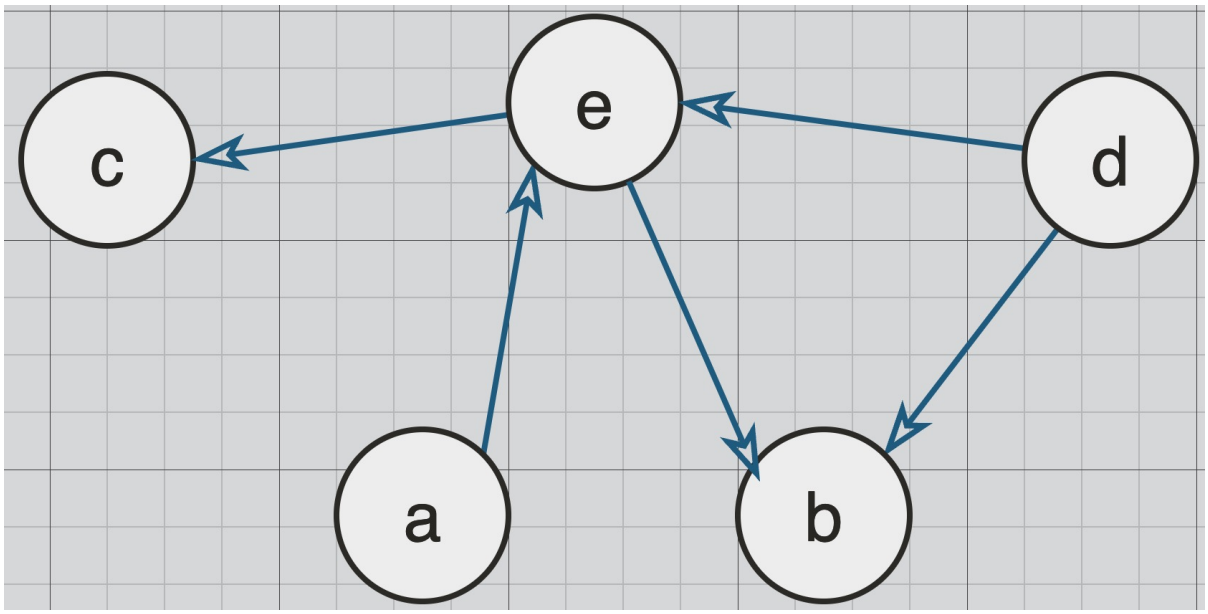
Storage Space (存儲空間):

- **Adjacency List**: Proportional to the number of vertices and edges, $O(V + E)$. (鄰接列表：與頂點和邊的數量成正比，$O(V + E)$。)
- **Adjacency Matrix**: Always $O(V^2)$, regardless of the number of edges. (鄰接矩陣：始終為 $O(V^2)$，與邊的數量無關。)

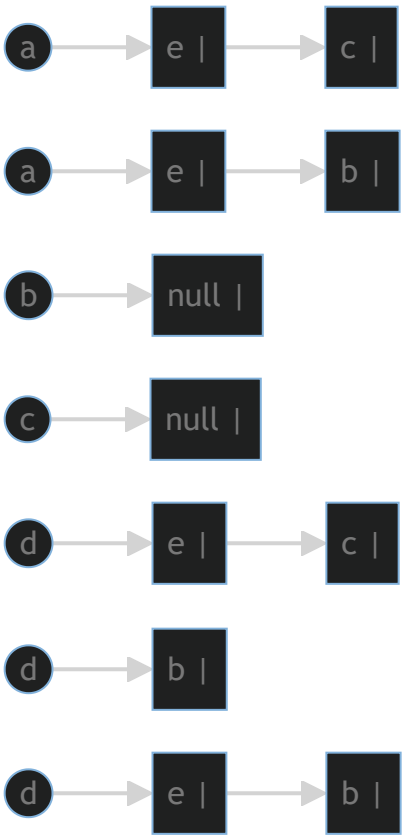
Time to Verify Edge Existence (驗證邊存在的時間):

- **Adjacency List**: $O(1)$ for sparse graphs, but can be up to $O(E)$ for dense graphs. (鄰接列表：對於稀疏圖為 $O(1)$，但對於密集圖可以高達 $O(E)$。)
- **Adjacency Matrix**: Always $O(1)$ since you're directly accessing a cell in the matrix. (鄰接矩陣：始終為 $O(1)$，因為您直接訪問矩陣中的一個單元。)
- For **dense graphs**, the adjacency matrix offers constant time edge verification but uses more space. (對於**密集圖**，鄰接矩陣提供常數時間的邊驗證，但使用更多的空間。)

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

	a	b	c	d	e
e	0	1	1	0	0

Search Algorithms BFS and DFS

搜尋算法

Introducing two primary search techniques:

- Their distinction lies in the sequence of node visits. 它們的區別在於節點訪問的順序。
- Both methods initiate from a source vertex s. 兩種方法都從源頂點 s 開始。

1. Applications: 應用

- **BFS**: Shortest path in unweighted graphs, network broadcasting. 在非加權圖中的最短路徑，網絡廣播。
- **DFS**: Topological sorting, cycle detection, pathfinding in weighted graphs. 拓撲排序，循環檢測，加權圖中的路徑查找。

2. Data Structures Used: 使用的數據結構

- **BFS**: Queue 佇列
- **DFS**: Stack or Recursion 堆疊或遞歸

3. Time Complexity: 時間複雜度

- 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. 對於鄰接列表表示的 BFS 和 DFS：$O(V + E)$，其中V是頂點的數量，E是邊的數量。

Breadth First Search (BFS)

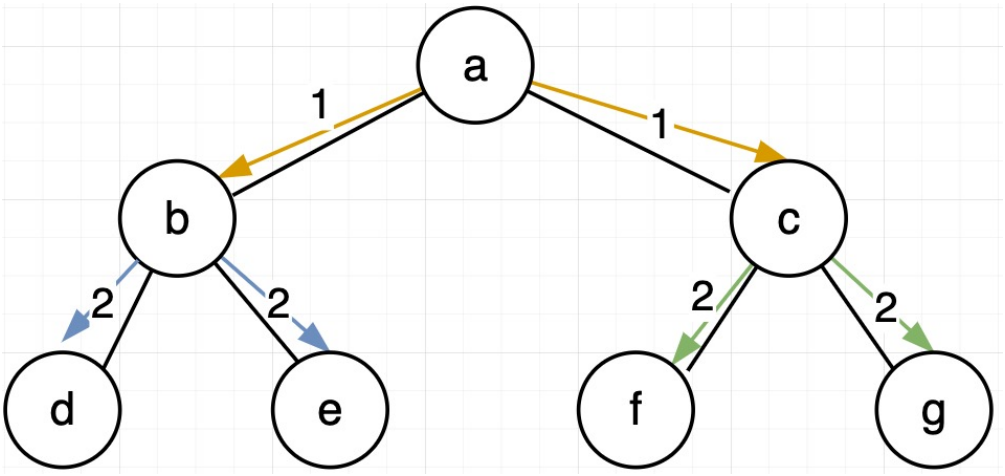
廣度優先搜索

- Aims to discover all nodes at distance d from s before any nodes at distance $d + 1$. 旨在在距離 $d + 1$ 的任何節點之前，發現距離 s 為 d 的所有節點。

Steps (BFS): 廣度優先搜索

1. **Start** at vertex `a`. 從頂點 `a` 開始。
2. ***Visit all neighbors of `a` : `b` and `c`.*** 訪問 `a` 的所有鄰居：`b` 和 `c`。
3. ***Move to `b`, visit its unvisited neighbors: `d` and `e`.*** 移動到 `b`，訪問其未訪問的鄰居：`d` 和 `e`。
4. ***Move to `c`, visit its unvisited neighbors: `f` and `g`.*** 移動到 `c`，訪問其未訪問的鄰居：`f` 和 `g`。
5. **Order** of visited vertices: `{a, b, c, d, e, f, g}`. 訪問頂點的順序：`{a, b, c, d, e, f, g}`。

BFS Example: (Undirected and Unweighted Graph)



Depth First Search (DFS)

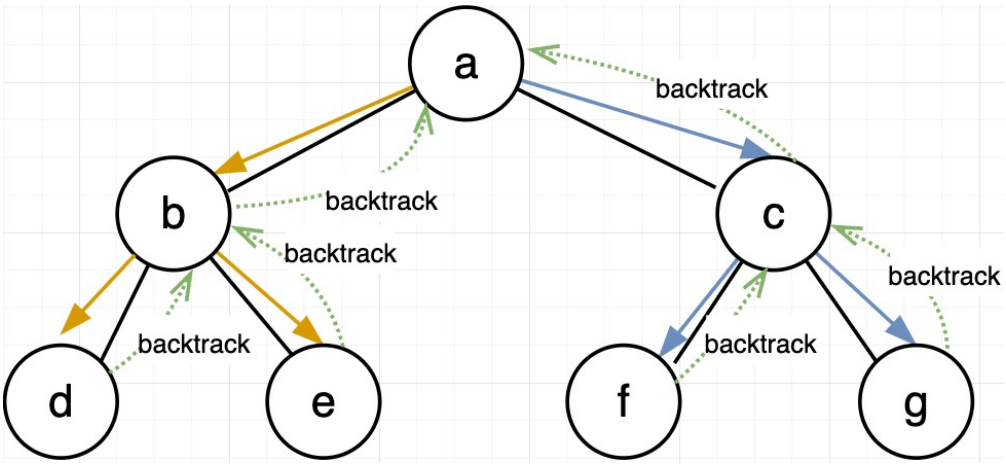
深度優先搜索

- Pursues each path to its fullest extent and backtracks, opting for any unexplored paths during the return journey. 追求每條路徑到其最大範圍，然後回溯，在返回過程中選擇任何未探索的路徑。
- Follow each path as far as possible and backtrack, taking any untraveled paths while returning

Steps (DFS): 深度優先搜索

1. **Start** at vertex `a` . 從頂點 `a` 開始。
2. **Explore** as far as possible along each branch before backtracking. 在回溯之前，沿每個分支盡可能地探索。
3. ***Visit*** `b` ***from*** `a` ***, then move to its unvisited neighbor*** `d` **.** 從 `a` 訪問 `b` ，然後移動到其未訪問的鄰居 `d` 。
4. ***Backtrack to*** `b` ***, then visit*** `e` **.** 回溯到 `b` ，然後訪問 `e` 。
5. ***Return to*** `a` ***, then visit*** `c` **.** ***From*** `c` ***, visit*** `f` ***and then*** `g` **.** 返回到 `a` ，然後訪問 `c` 。從 `c` 訪問 `f` ，然後訪問 `g` 。
6. **Order** of visited vertices: `{a, b, d, e, c, f, g}` . 訪問頂點的順序： `{a, b, d, e, c, f, g}` 。

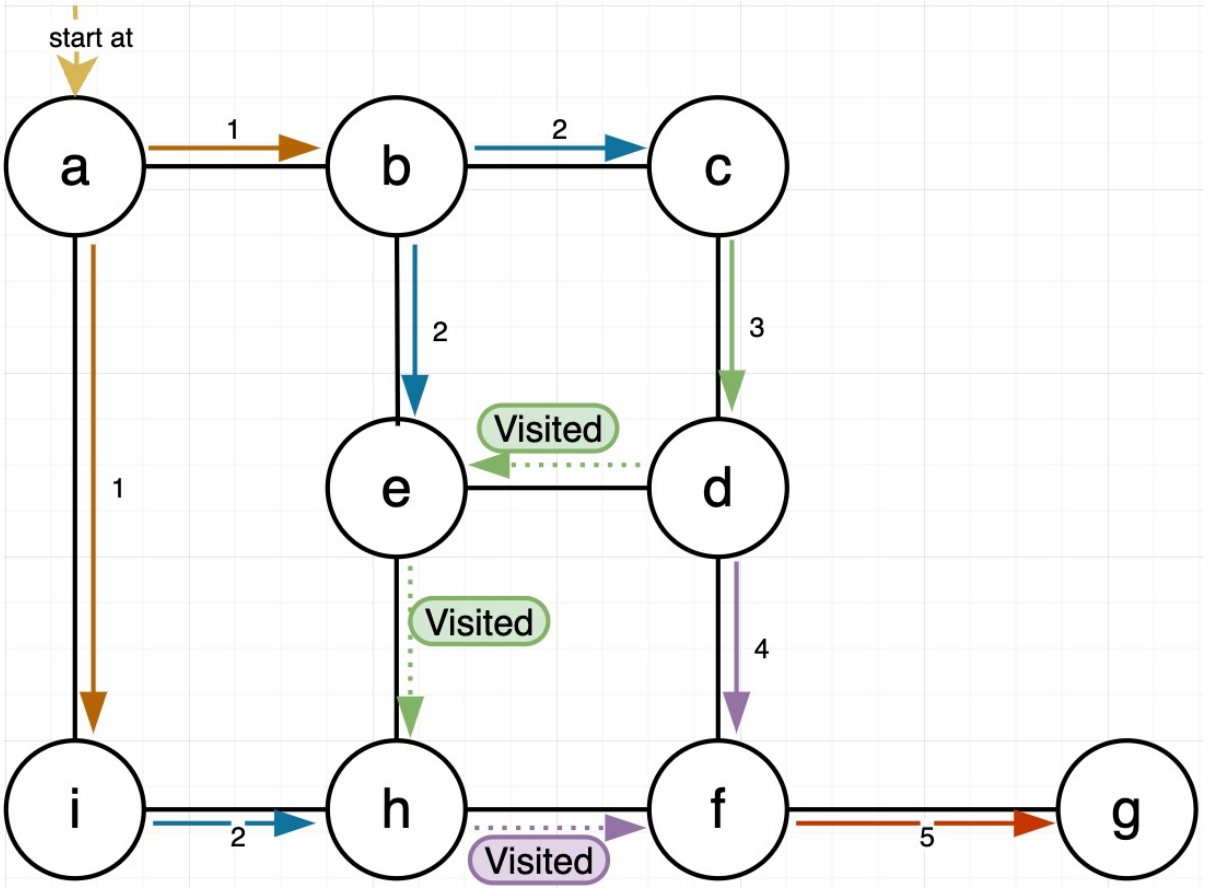
DFS Example: (Undirected and Unweighted Graph)



Note: The order in DFS can vary based on the starting point and the order of neighbors in the adjacency list. (注意：根據起始點和鄰接列表中鄰居的順序，DFS 中的順序可能會有所不同。)

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)n, Start from `Node a` ,

1. Visit the starting vertex `a` and enqueue it. 訪問起始頂點 `a` 並將其入隊。
   - Queue: [a] and Visited: `{a,}`
2. Dequeue `a` and enqueue its unvisited neighbors `b` and `i` . 出隊 `a` 並將其未訪問的鄰居 `b` 和 `i` 入隊。
   - Queue: [b, i] and Visited: `{a, b, i}`
3. Dequeue `b` and enqueue its unvisited neighbor `c` and `e` . 出隊 `b` 並將其未訪問的鄰居 `c` 和 `e` 入隊。
   - Queue: [i, c, e] and Visited: `{a, b, i, c, e}`
4. Dequeue `i` and enqueue its unvisited neighbor `h` . 出隊 `i` 並將其未訪問的鄰居 `h` 入隊。
   - Queue: [c, e, h] and Visited: `{a, b, i, c, e, h}`
5. Dequeue `c` and enqueue its unvisited neighbor `d` . 出隊 `c` 並將其未訪問的鄰居 `d` 入隊。
   - 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}

Step	Current Vertex	Queue	Visited Vertices
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: 初始化

```
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: 對於圖中除了起始頂點 `s` 的每一個頂點 `u` ，我們:

- Set its color to `white` indicating it's unvisited. 將其顏色設為 `white` ，表示它尚未被訪問。
- Set its predecessor `u.π` to `NIL` (no predecessor yet). 將其前驅 `u.π` 設為 `NIL` (還沒有前驅)。
- Set its distance `u.d` from the source to infinity (∞) initially. 最初將其距離 `u.d` 從源頭設為無窮大(∞)。

4. Starting Vertex Initialization: 起始頂點初始化

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

For the starting vertex `s` : 對於起始頂點 `s`

- Set its color to `gray` indicating it's discovered but not fully explored. 將其顏色設為 `gray` ，表示它已被發現但尚未完全探索。
- Set its distance `s.d` from itself to 0. 將其距離 `s.d` 從自己設為 0。

6. Queue Initialization: 隊列初始化

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

Initialize an empty queue `Q` and enqueue the starting vertex `s` . 初始化一個空隊列 `Q` ，並將起始頂點 `s` 入隊。

8. BFS Loop: BFS 循環

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

While the queue is not empty, dequeue a vertex `u` from the front of the queue. 當隊列不為空時，從隊列前端出隊一個頂點 `u` 。

10. Neighbor Exploration: 鄰居探索

```
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` : 對於 `u` 的每一個鄰居 `v`

- If `v` is unvisited (`white`), mark it as discovered (`gray`). 如果 `v` 未被訪問(`white`)，將其標記為已發現(`gray`)。
- Set `u` as the predecessor of `v` and update the distance of `v` from the source. 將 `u` 設為 `v` 的前驅並更新 `v` 從源頭的距離。
- Enqueue `v` to the queue. 將 `v` 入隊到隊列。

15. Vertex Fully Explored: 頂點完全探索

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

After exploring all neighbors of `u` , mark `u` as fully explored (`black`). 在探索了 `u` 的所有鄰居後，將 `u` 標記為已完全探索(`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` . 從節點 `a` 開始。

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

2. `b` : Move from `a` to `b` . 從節點 `a` 移動到 `b` 。

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

3. `c` : Move from `b` to `c` . 從節點 `b` 移動到 `c` 。

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

4. `d` : Move from `c` to `d` . 從節點 `c` 移動到 `d` 。

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

5. `e` : Move from `d` to `e` . 從節點 `d` 移動到 `e` 。

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

6. `h` : Move from `e` to `h` . 從節點 `e` 移動到 `h` 。

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

7. `i` : Move from `h` to `i` . 從節點 `h` 移動到 `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` . 由於 `i` 沒有未訪問的鄰居，所以回溯到 `h` 。

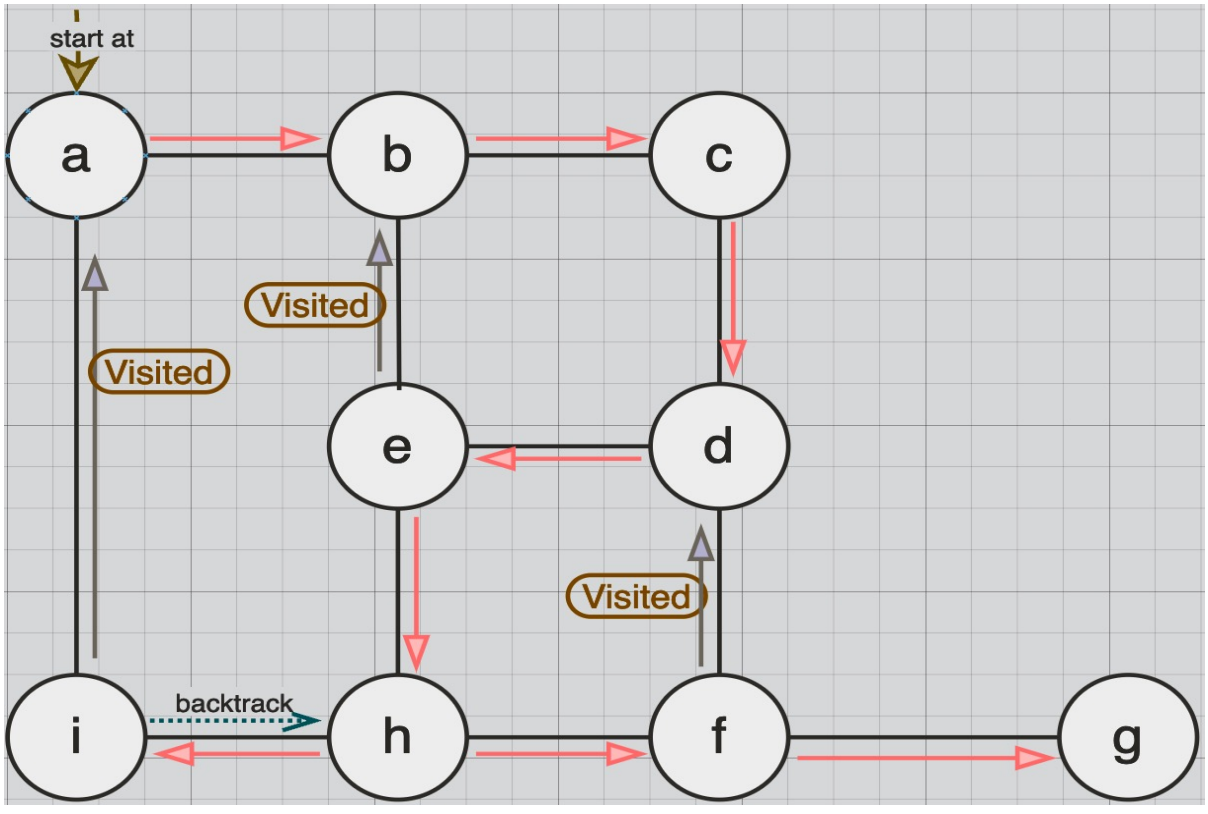
9. `f` : Move from `h` to `f` . 從節點 `h` 移動到 `f` 。

- 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` . 從節點 `f` 移動到 `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 遍歷結束。序列中的所有節點都已被訪問。



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. 想像一個帶有源頂點 s 的圖。從 s 到頂點 v 有多條路徑。在這些路徑中，有一條路徑通過頂點 u，這是從 s 到 v 的最短路徑。

Visualization: 視覺化

Vertices and Paths

頂點和路徑

Paths Visualization:

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

- Let's have three vertices: s, u, and v. 讓我們有三個頂點：s、u 和 v。
- There's a path from s to v that goes through u. This is the shortest path. 有一條從 s 到 v 的路徑通過 u。這是最短的路徑。
- There might be other paths from s to v that don't go through u. 可能還有其他從 s 到 v 的路徑，不經過 u。

2. $u.d$: The BFS distance from s to u is represented as $u.d$. 從 s 到 u 的 BFS 距離表示為 $u.d$。

- The shortest path distance from s to u is $\delta(s, u)$. 從 s 到 u 的最短路徑距離是 $\delta(s, u)$。

3. $v.d$: The BFS distance from s to v is represented as $v.d$. 從 s 到 v 的 BFS 距離表示為 $v.d$。

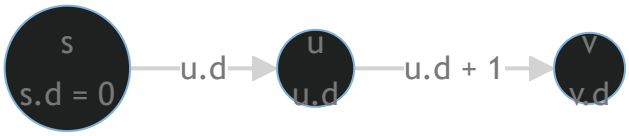
- The shortest path distance from s to v is $\delta(s, v)$. 從 s 到 v 的最短路徑距離是 $\delta(s, v)$。

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

距離

- $\delta(s, u)$ is the shortest distance from s to u. $\delta(s, u)$ 是從 s 到 u 的最短距離。
- $\delta(s, v)$ is the shortest distance from s to v. $\delta(s, v)$ 是從 s 到 v 的最短距離。



In this graph:

- The vertex `s` has a BFS distance `d` of 0 since it's the source. 頂點 `s` 的 BFS 距離 `d` 為 0，因為它是源頭。
- The edge from `s` to `u` is labeled with `u.d`, representing the BFS distance from `s` to `u`. 從 `s` 到 `u` 的邊被標記為 `u.d`，表示從 `s` 到 `u` 的 BFS 距離。
- 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`. 從 `u` 到 `v` 的邊被標記為 `u.d + 1`，表示從 `s` 到 `v` 的 BFS 距離 因為它比到 `u` 的距離多一。

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. 引理聲稱 $v.d$（從 s 到 v 的 BFS 距離）總是小於或等於 $u.d + 1$。這在視覺上表示為 v 可以直接從 u 到達，而 u 可以直接從 s 到達。

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 $v.d > u.d + 1$. 給定頂點 u 在從源 s 到 v 的最短路徑上先於頂點 v，我們想要證明 $v.d = \delta(s, v)$，其中 $\delta(s, v)$ 代表從源 s 到頂點 v 的最短距離。對於邊 (u, v)，我們假設 $\delta(s, v) \leq \delta(s, u) + 1$ 且 $v.d > u.d + 1$。

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 ∞. 如果頂點（無論是 u 還是 v）無法從 s 到達，那麼 $\delta(s, u)$、$\delta(s, v)$、$u.d$ 和 $v.d$ 都是 ∞。

Case 2:
1. Direct Path: If the edge from u to v is part of the shortest path from s to v, then $\delta(s, v) = \delta(s, u) + 1$.
如果從 u 到 v 的邊是從 s 到 v 的最短路徑的一部分，那麼 $\delta(s, v) = \delta(s, u) + 1$。

2. Through u: If the shortest path from s to v passes through u, then $\delta(s, v) = \delta(s, u) + 1$.
如果從 s 到 v 的最短路徑通過 u，那麼 $\delta(s, v) = \delta(s, u) + 1$。

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

Case 3:
3. Indirect Path: If adding edge u to v isn't shortest, then $\delta(s, v) < \delta(s, u) + 1$. 如果添加邊 u 到 v 不是最短的，那麼 $\delta(s, v) < \delta(s, u) + 1$。

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

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

3. If the shortest path from s to v does not pass through u, then $\delta(s, v) < \delta(s, u) + 1$. 如果從 s 到 v 的最短路徑不經過 u，那麼 $\delta(s, v) < \delta(s, u) + 1$。

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

In summary, BFS correctly computes shortest path distances in unweighted graphs by comparing BFS distances to actual shortest path distances. 總之，BFS 通過將 BFS 距離與實際的最短路徑距離進行比較，正確地計算了無權重圖中的最短路徑距離。

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. BFS 終止時，對於所有從源點 s 可達的頂點 v，都有 $v.d \geq \delta(s, v)$，其中 $v.d$ 是 BFS 分配的距離，而 $\delta(s, v)$ 是從 s 到 v 的最短路徑距離。

Proof: 證明

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. 基礎情況: 源頂點 s 是首先被加入和取出的。當 s 被取出時，$s.d = 0$，這等於 $\delta(s, s)$。因此，基礎情況成立。

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, $v.d \geq \delta(s, v)$. 假設對於某個在 v 之前被取出的頂點 u，屬性 $u.d \geq \delta(s, u)$ 成立。現在，考慮一條邊 (u, v)。當 v 首次被取出時，其距離 $v.d$ 被設定為 $u.d + 1$。根據我們的假設，$u.d \geq \delta(s, u)$，所以 $v.d \geq \delta(s, u) + 1$。由於 $\delta(s, v)$ 是從 s 到 v 的最短路徑，$v.d \geq \delta(s, v)$。

Thus, by induction, the lemma holds for all vertices reachable from s. 因此，通過歸納法，引理對於所有從 s 可達的頂點都成立。

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: 給定一個圖的 BFS，如果頂點u與源頂點s的距離為d，且 BFS 隊列在某一時刻為$< v_1, v_2, v_3, v_4, ...v_r >$，其距離分別為$< d, d, d + 1, d + 1 >$，那麼：

1. v_1 and v_2 are at distance $d + 1$. v_1和v_2的距離為$d + 1$。
2. v_4 is a neighbor of v_1 and is at distance $d + 2$. v_4是v_1的鄰居，距離為$d + 2$。
3. v_r is the last neighbor of v_3 and is at distance $d + 2$. v_r是v_3的最後一個鄰居，距離為$d + 2$。

Proof:

- Vertex v was enqueued because it is adjacent to the last vertex that was dequeued (let's call this vertex u). 頂點v被加入隊列，因為它與最後被出隊的頂點相鄰（我們稱此頂點為u）。
- Therefore, $v.d = u.d + 1$. 因此，$v.d = u.d + 1$。
- By the induction hypothesis, $v.d \geq \delta(s, u) + 1$. 根據歸納假設，$v.d \geq \delta(s, u) + 1$。
- By the previous lemma, $v.d \geq \delta(s, v)$. 根據前一個引理，$v.d \geq \delta(s, v)$。

From the above, we can conclude: 從上面我們可以得出結論

1. $v_r.d \leq v_1.d + 1$.

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

To show: Upon termination of BFS, $v.d = \delta(s, v)$. 要證明：BFS 終止時，$v.d = \delta(s, v)$。

Proof by contradiction: 反證法

Assume a vertex v does not receive the shortest distance during BFS, i.e., $v.d > \delta(s, v)$. 假設頂點v在 BFS 期間沒有收到最短距離，即$v.d > \delta(s, v)$。

1. If v is unreachable, then $v.d = \infty$ and $\delta(s, v) = \infty$. Thus, $v.d = \delta(s, v)$. 如果v是不可達的，那麼$v.d = \infty$且$\delta(s, v) = \infty$。因此，$v.d = \delta(s, v)$。
2. If v is the source vertex s, then $v.d = 0$ and $\delta(s, s) = 0$. 如果v是源頂點s，那麼$v.d = 0$且$\delta(s, s) = 0$。
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. 如果v是可達的且不是s，且在 BFS 期間沒有收到最短距離，那麼讓u成為從s到v的最短路徑中的前一個頂點。我們有$u.d = \delta(s, u)$和$\delta(s, v) = \delta(s, u) + 1$。但$v.d > \delta(s, v)$，這意味著$v.d > u.d + 1$，與我們的 BFS 性質相矛盾。

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)

深度優先搜索

- DFS explores as deep as possible along a branch before backtracking. 深度優先搜索沿著一個分支盡可能深入地探索，然後再回溯。
- When a dead-end is reached, it backtracks to find new paths. 當達到死胡同時，它將回溯以尋找新的路徑。
- The process continues until all nodes are visited. 這個過程將持續到訪問所有節點。
- If the graph G is connected, DFS describes a tree G_π. 如果圖 G 是連接的，DFS 描述了一棵樹 G_π。
  - a tree $G_\pi = (V, E_\pi)$
- If G is not connected, G_π is a **forest** (a collection of trees). 如果 G 沒有連接，那麼 G_π 是一個**森林**（樹的集合）。
- The `Stack` for DFS, The `Queue` for BFS

DFS TIMESTAMPS & VERTEX COLORS

DFS Colors: (White, Grey, Black)

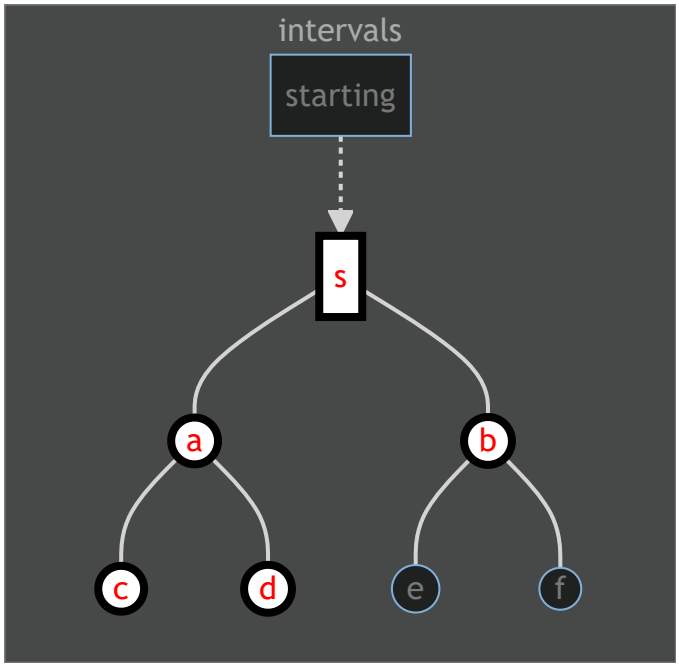
- Initially white, not visit yet
- Set to gray upon discover, visited
- Set to black when that node is finished, explored vertex and its neighbors

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 $1, 2, \ldots, 2|V|$

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`.

解答：$O(|V|)$ 因為它初始化圖 `G` 中的所有頂點。

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.

解答：最多$O(|V|)$次，對於圖 `G` 中尚未訪問的每個頂點一次。

How many times is line 3 of DFS-Visit executed?

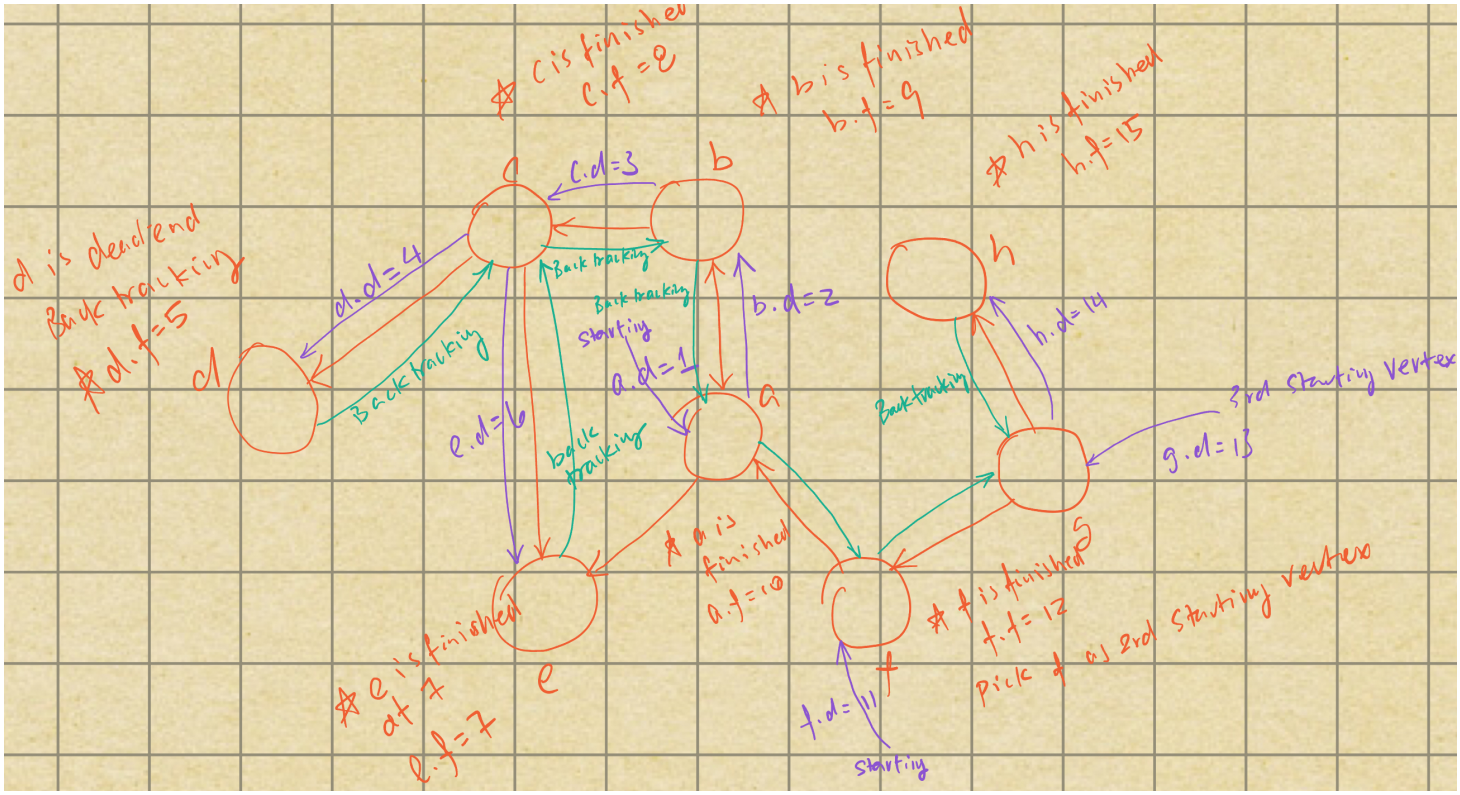
Answer: For each vertex and its adjacent vertices, so at most $O(|V| + |E|)$ times, where $|E|$ is the number of edges.

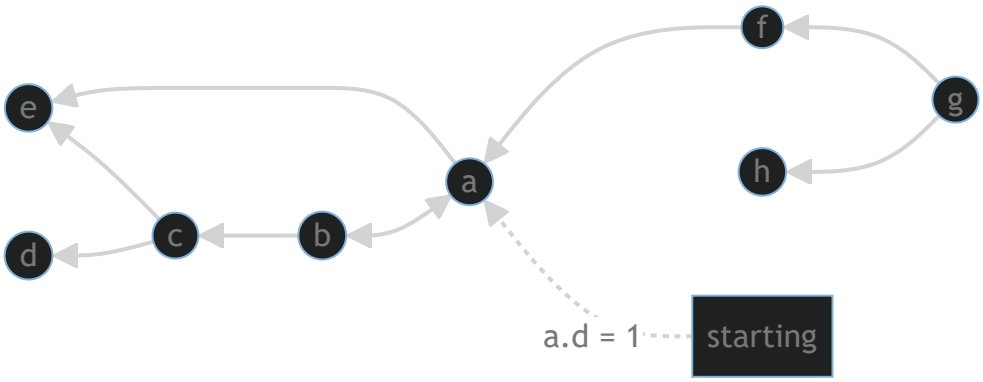
解答：對於每個頂點及其相鄰的頂點，所以最多$O(|V| + |E|)$次，其中$|E|$是邊的數量。

What is the total runtime of DFS?

Answer: $O(|V| + |E|)$ because every vertex and every edge is explored once.
解答：$O(|V| + |E|)$ 因為每個頂點和每條邊都被探索一次。

NOT YET DONE **Directed Graph DFS Example:**



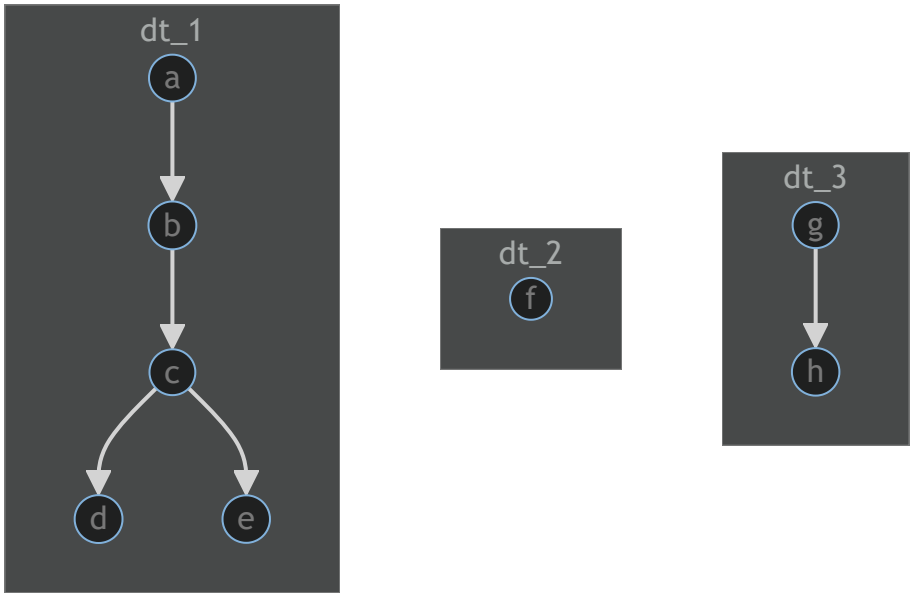


v	v.d	v.f
a	1	10
b	2	9
c	3	8
d	4	5
e	6	7
f	11	12
g	13	16

v	v.d	v.f
h	14	15

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

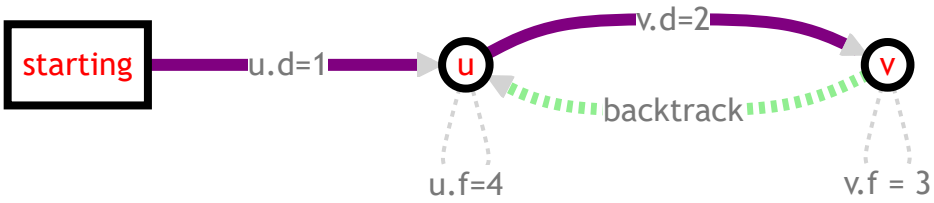
- From the example, it has 3 starting vertices, the $v.\pi$ are **disjoint forests** as the result
- any vertex can be the starting vertex, and if pick vertex `g`, the result will be totally different



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: 對於圖 $G = (V, E)$ 上的任何 DFS，給定任何頂點 $u, v \in V$，考慮區間 $[u.d, u.f]$ 和 $[v.d, v.f]$。以下三個條件之一始終成立：

1. $[u.d, u.f]$ is entirely within $[v.d, v.f]$, and u is a descendant of v in the DFS tree. $[u.d, u.f]$ 完全位於 $[v.d, v.f]$ 內，且 u 是 DFS 樹中 v 的後代。
2. $[v.d, v.f]$ is entirely within $[u.d, u.f]$, and v is a descendant of u in the DFS tree. $[v.d, v.f]$ 完全位於 $[u.d, u.f]$ 內，且 v 是 DFS 樹中 u 的後代。
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. 區間 $[u.d, u.f]$ 和 $[v.d, v.f]$ 不重疊，這意味著在 DFS 森林中，u 和 v 都不是另一個的後代。



PROOF OF PARENTHESIS THEOREM There are two main cases based on the discovery times of vertices u and v: 根據頂點 u 和 v 的發現時間，有兩種主要情況：

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: $v.f < u.f$. 在 u 的探索期間，v 被發現並在 DFS 樹中成為 u 的後代。v 在 u 之前完成：$v.f < u.f$.
- The active interval of v, $[v.d, v.f]$, is entirely within u's active interval, $[u.d, u.f]$. v 的活動區間 $[v.d, v.f]$ 完全位於 u 的活動區間 $[u.d, u.f]$ 內。

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 $v.f < u.f$. 在探索 u 期間發現了 v，使得 v 在 DFS 樹中成為 u 的後代。 v 在 u 之前完成，所以 $v.f < u.f$。
- u, being an ancestor of v in the DFS tree, finishes last. u 作為 DFS 樹中 v 的祖先，最後完成。
- 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. v 處於活動狀態的區間 $[v.d, v.f]$ 完全包含在 u 處於活動狀態的區間 $[u.d, u.f]$ 內。

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

- **Vertex Color** When v was discovered, u was gray, indicating u was being visited 當發現 v 時，u 是灰色的。
- **Ancestry Conclusion:** v is a descendant of u in the DFS tree. v 是 DFS 樹中 u 的後代。
- **Return Order:** `DFS-Visit(v)` returns before `DFS-Visit(u)` due to the recursive nature of DFS. 由於 DFS 的遞歸性質， `DFS-Visit(v)` 在 `DFS-Visit(u)` 之前返回。
- **Interval Relationship:** $u.d < v.d < v.f < u.f$ 區間：$u.d < v.d < v.f < u.f$

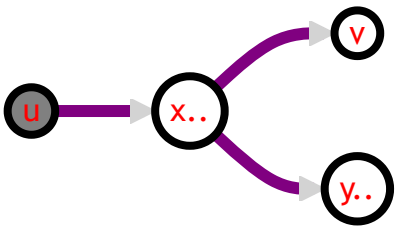
Subcase 2: $u.d < u.f < v.d$ 子情況 2: $u.d < u.f < v.d$

- **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. 在 DFS 樹中，u 和 v 都不是另一個的後代。

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`. 在圖 `G = (V,E)` 的 DFS 中，若在 `u` 的發現時間（ `u.d` ）有一條從 `u` 到 `v` 的白色路徑，則 `v` 是 `u` 的後代。

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



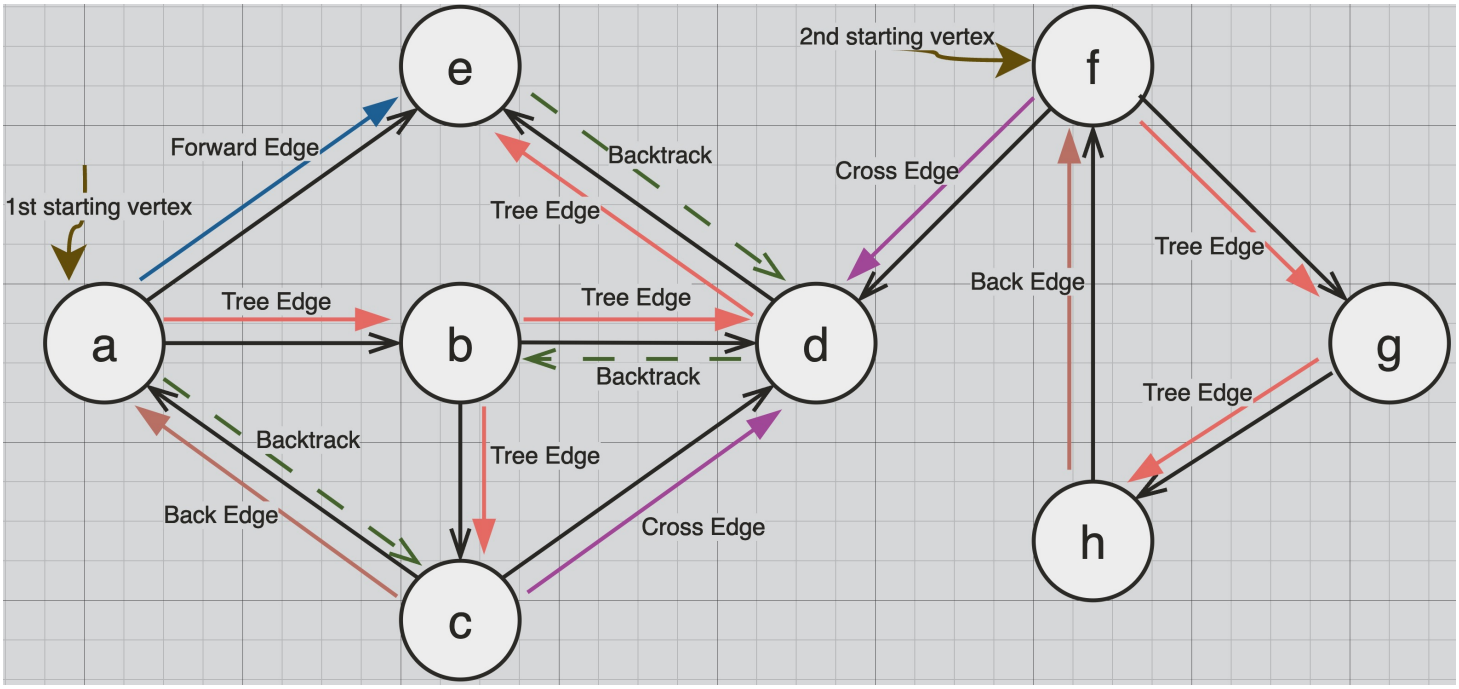
Proof White Path Theorem in 2 Parts: 白色路徑定理的證明：

1. **If v is a descendant of u in DFS:** All descendants of `u`, including `v`, are white when `u` is discovered. 當 `u` 被發現時， `u` 的所有後代（包括 `v` ）都是白色。
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`. 如果從 `u` 到 `v` 存在白色路徑，則由於在 `u` 之前尚未發現 `v` ，因此將其作為 `u` 的後代進行探索。

The presence of a white path during `u`'s discovery signifies `v`'s descendant relationship with `u` in DFS. 在發現 `u` 時，白色路徑的存在表示 `v` 與 DFS 中的 `u` 的後代關係。

CLASSIFICATION OF EDGES

邊的分類



1. Tree Edge: (GRAY to WHITE) 樹邊

- Directly connects a vertex to its descendant in the DFS tree. 直接連接頂點到其在 DFS 樹中的後代。
- i.e., **Tree Edge** (a, b), (b, d) in the graph.

2. Back Edge: (GRAY to GRAY) 後向邊

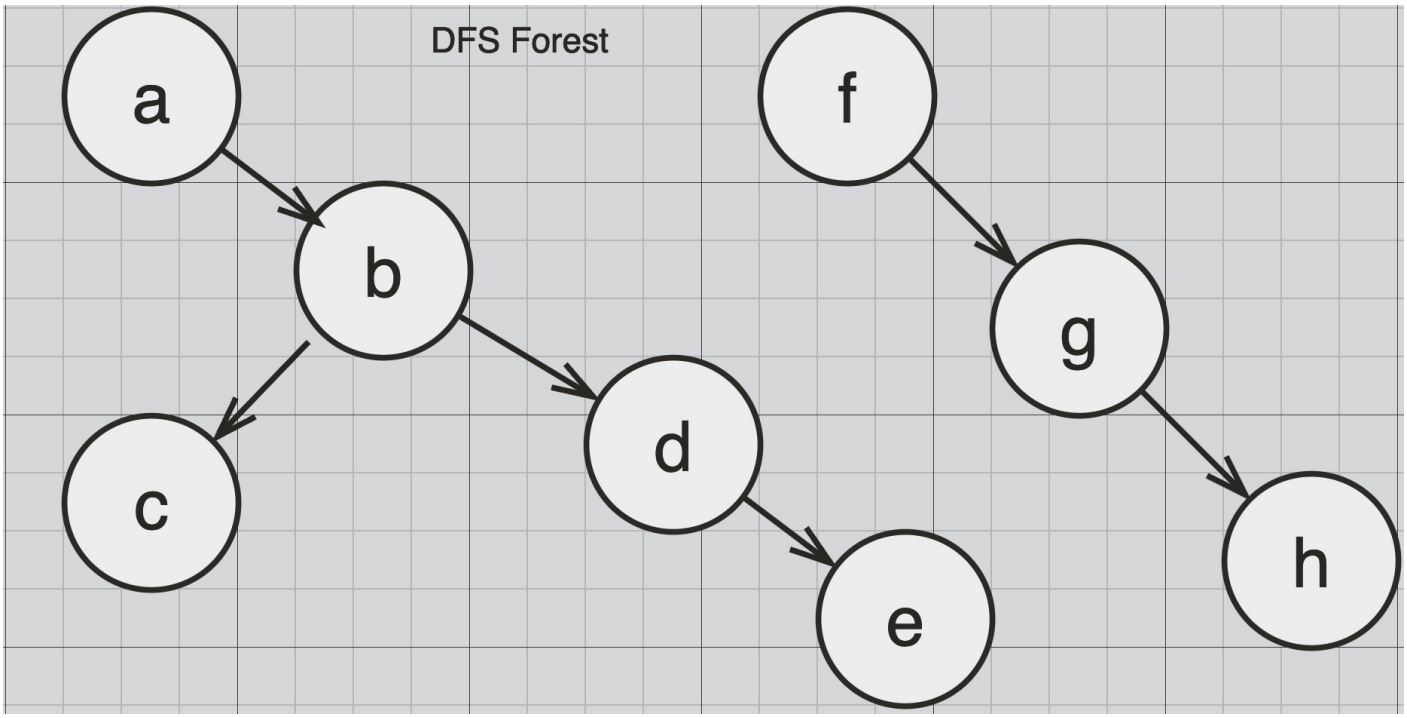
- Edges that connect a vertex to one of its ancestors in the DFS tree. This includes self-loops. 連接頂點到其 DFS 樹中的某個祖先的邊。這包括自循環。
- i.e., **Back Edge** (c, b), and (h, f) in the graph.

3. Forward Edge: (GRAY to BLACK) 前向邊

- Non-tree edges that connect a vertex to one of its descendants in the DFS tree. 連接頂點到其 DFS 樹中的某個後代的非樹邊。
- i.e., Forward Edge (a, e) in the graph.

4. Cross Edge: (GRAY to BLACK) 交叉邊

- 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. 其中沒有一個頂點是另一個的直接祖先或後代的邊。它們可以在同一 DFS 樹內或在不同的 DFS 樹之間。
- Example: Cross Edge (c, d), and (f, d)in the graph.



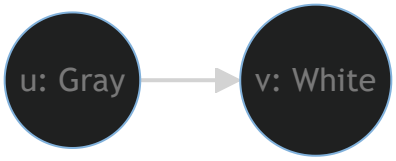
EDGE COLORING

In DFS, the color of vertex v upon encountering edge (u, v) determines the edge type. 當 DFS 遇到邊 (u, v) 時，頂點 v 的顏色決定了邊的類型。

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

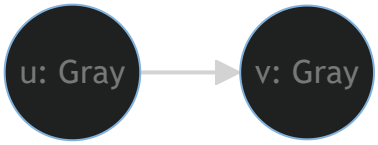
1. v **is White**: Tree Edge, Directly connects a vertex to its descendant. 直接連接頂點到其後代。

- $u.d < v.d < v.f < u.f$
- v is discovered and finished after u. v is a descendant in the DFS tree. v 是 DFS 樹中的後代。
- After vertex u is discovered, vertex v is discovered and finished before u finishes. v is a descendant of u in the DFS tree. 在頂點 u 被發現之後，頂點 v 被發現並在 u 完成之前完成。在 DFS 樹中，v 是 u 的後代。



2. v **is Gray**: Back Edge, Connects a vertex to its ancestor or even itself (self-loop). 連接頂點到其祖先或其本身（自循環）。

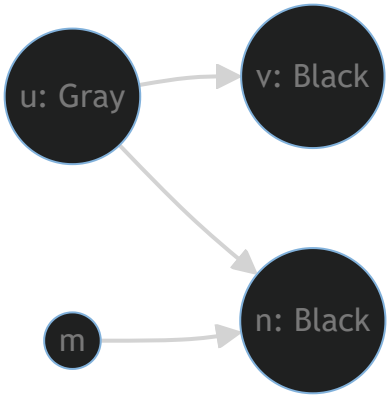
- $v.d < u.d < u.f < v.f$
- Creates a cycle as v is an ancestor in the DFS tree. 創建循環。



3. v **is Black**: Forward or Cross Edge (前向邊或交叉邊)

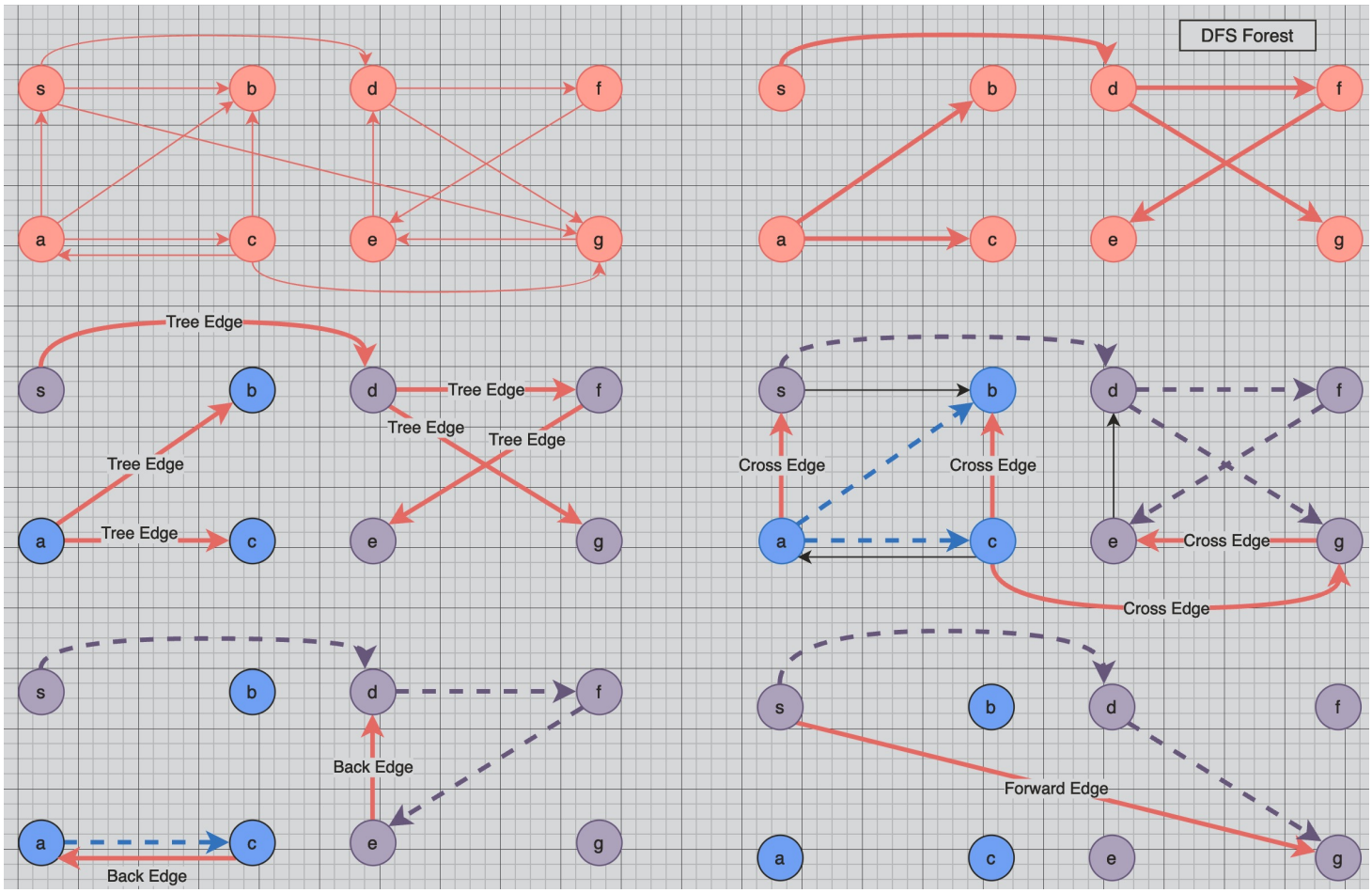
- Forward Edge: A non-tree edge that connects a vertex to a non-child descendant. 一個非樹邊，連接頂點到非子後代。
  - $u.d < v.d < u.f < v.f$

- Cross Edge: Connects unrelated vertices, either within the same DFS tree or between different DFS trees. 連接不相關的頂點，可以在同一 DFS 樹中或在不同的 DFS 樹之間。
  - $v.d < v.f < u.d < u.f$ or $u.d < u.f < v.d < v.f$




```
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:

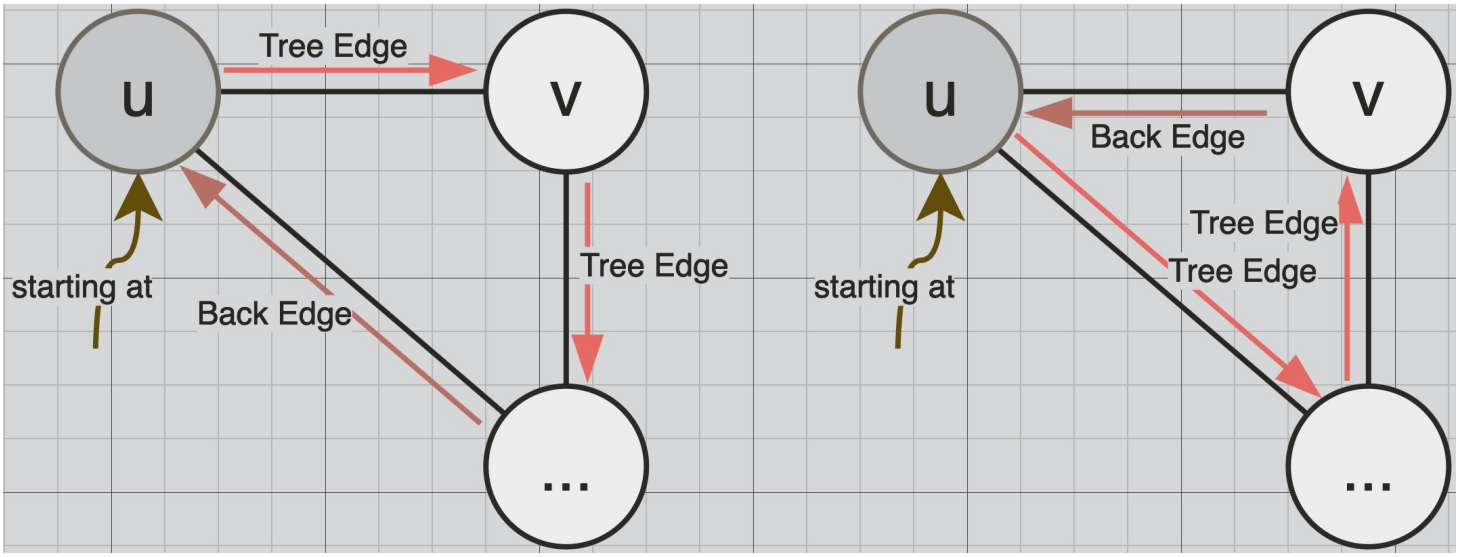


Solution 2:

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

UNDIRECTED GRAPHS

無向圖



Theorem (Specific to DFS): In a DFS traversal of an undirected graph, every edge is either a **TREE EDGE** or a **BACK EDGE**. 在進行無向圖的深度優先搜索(DFS)時，每條邊要麼是**樹邊**，要麼是**後向邊**。

What does this mean with respect to edge colors in DFS? 這與 DFS 中的邊的顏色有什麼關係？

1. **White**: Vertex not discovered. 白色：頂點未被發現。
2. **Gray**: Vertex discovered but not finished. 灰色：頂點已被發現但未完成。
3. **Black**: Vertex finished. 黑色：頂點已完成。

Proof: 證明

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 $u.d < v.d$. 考慮 E 中的一條邊 (u, v)。假設 $u.d$ 和 $v.d$ 分別代表頂點 u 和 v 的發現時間，且 $u.d < v.d$。

- **Claim**: The discovery and finish times of the vertices satisfy $u.d < v.d < v.f < u.f$. **主張**：頂點的發現和完成時間滿足 $u.d < v.d < v.f < u.f$。

There are two possibilities: 有兩種可能性

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. $v.\pi = u$：這意味著在 DFS 樹中，頂點 v 是 u 的直接後代。邊 (u, v) 是一條樹邊。
2. $v.\pi \neq u$: This means vertex v is not a direct descendant of u. The edge (u, v) is a back edge. $v.\pi \neq u$：這意味著頂點 v 不是 u 的直接後代。邊 (u, v) 是一條後向邊。

DFS APPLICATIONS

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

Topological Sort

拓撲排序

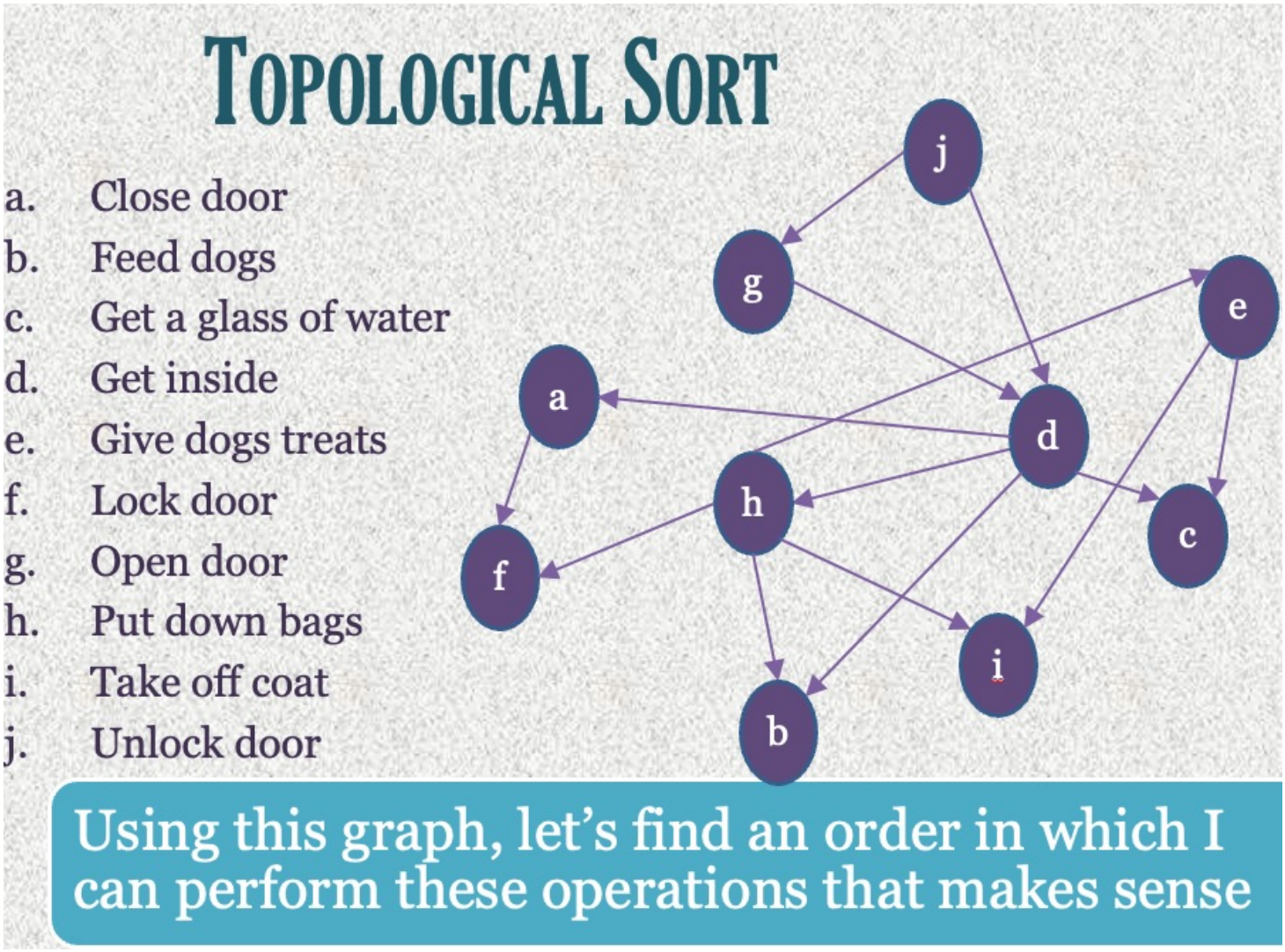
It's a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v. 它是有向無環圖 (DAG) 中頂點的線性排序，使得對於每個有向邊 (u, v)，頂點 u 出現在頂點 v 之前。

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. 簡單來說：想像您有多個任務，且某些任務必須在其他任務之前完成。拓撲排序提供了一種完成任務的順序，而不違反任何先決條件。

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. ball 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 15 16
b 13 14
c 10 11
d 3 18
e 9 12
f 7 8
g 2 19
h 4 17
i 5 6
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

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

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)
• vis white
• 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

STRONGLY CONNECTED COMPONENTS

Find all subgraphs in G such that u and v are in the same subgraph iff there is a path from u to v in G and there is a path from v to u in G

- u and v are in the same SCC if there exist paths u ~~ v and v ~~ u

STRONGLY CONNECTED COMPONENTS
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 ET = {(v,u) | (u,V) € ES
For any graph G= (V,E), consider the
graph GT = (V,ET)
• Note G and in GT will have the same strongly connected
components

SCC(G)

1. Call DFS(G) creating finishing times uf for each vertex u
2. Compute G'
3. Call DFS(G), 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 ALGORITHM

First DFS

• Arranges vertices in a sort of
"topological" order
• Not a true topological sort because G
contains cycles

Second DFS

• Finds nodes in GT connected to each root node
• Because of sort order, v in tree
rooted at u = u ~ v and v ~ u

EXAMPLE

• Decreasing u.f order of DFS(G)

- `a, f, g, e, b, c, d`
  • Compute GT

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

First SCC: a, b, c, e
Second SCC: f, g, h
Third SCC: d

V^{SCC} = {a, d, h}, E^{SCC} = {(a,d), (a,h)}