Elementary

Graph Algorithms

PART-4

1

Outlines: Graphs Part-4

- Applications of DFS
  - **Topological Sort of Directed Acyclic Graph**
  - **Strongly Connected Components**

2

Applications of Depth-First Search

- **Topological Sort:** Using depth-first search to perform topological sort of a directed acyclic graph.
- **Strongly Connected Components:** Decomposing a directed graph into a strongly connected components using two depth-first searches.

3

Directed Acyclic Graph (DAG)

- Arises in many applications where there are **precedence** or **ordering** constraints (e.g. scheduling problems)
  - If there are a series of tasks to be performed, and certain tasks must precede other tasks
- In general, a precedence constraint graph is a DAG, in which vertices are tasks and edge (u, v) means that task u must be completed before task v begins.

4

Generic scheduling problem

- Input:
  - Set of tasks { T_1, T_2, T_3, …, T_n}
    - Example: getting dressed in the morning: put on shoes, socks, shirt, pants, belt, …
  - Set of dependencies { $T_1 \rightarrow T_2$, $T_3 \rightarrow T_4$, $T_5 \rightarrow T_1$, …}
    - Example: must put on socks before shoes, pants before belt, …
- Want:
  - ordering of tasks which is consistent with dependencies
- Problem representation: Directed Acyclic Graph
  - Vertices = tasks; Directed Edges = dependencies
  - Acyclic: if ∃ cycle of dependencies, no solution possible

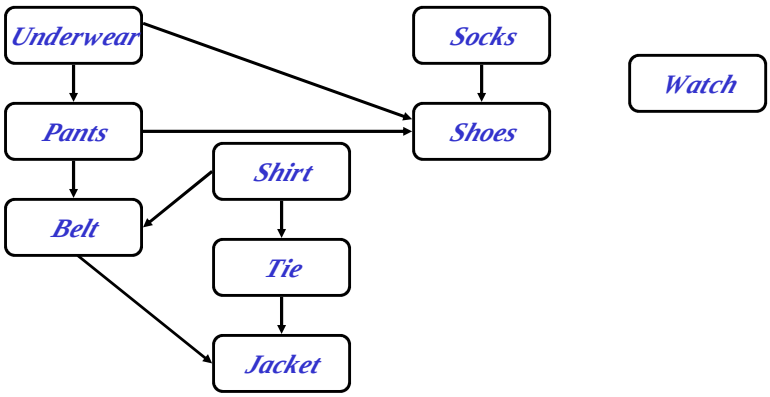
5

Topological Sort

- **Topological sort** of a DAG:
  - Linear ordering of all vertices in graph G such that if edge $(u, v) \in G$ then vertex u comes before vertex v
  - If the graph is ***not acyclic***, then no linear ordering is possible
- Topological sort is useful in scheduling jobs with precedence
- In general, there may be many legal topological orders for a given DAG.
- Real-world example: getting dressed

6

Getting Dressed



- A possible schedule for dressing can be:

  socks → shirts → pants → belt → tie → jacket → shoes → watch

7

Topological Sort Algorithm

Topological-Sort(G)
{

1. call **DFS**(*G*) to compute *finishing time* *f*[*v*] for each vertex *v*.

2. as a vertex is finished, **insert** it onto the **front** of a *linked list*.

3. **return** the linked list of vertices.

}

8

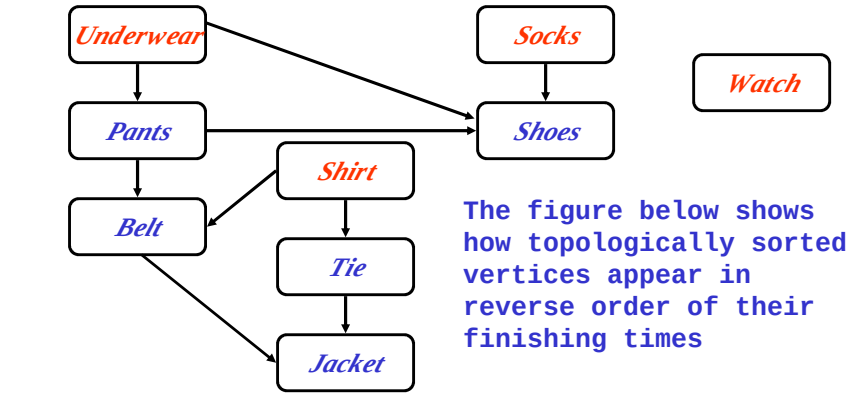
DFS Algorithm

```
DFS(G)
{
   for each vertex u ∈ V[G]
   {
      color[u] = WHITE;
      p[u] = NIL;
   }
   time = 0;
   for each vertex u ∈ V[G]
   {
      if (color[u] == WHITE)
         DFS_Visit(u);
   }
}
```

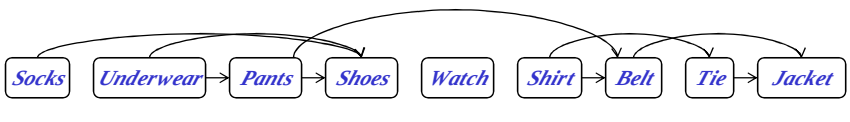
```
DFS_Visit(u)
{
   color[u] = GRAY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]

      if (color[v] == WHITE)
          p[v] = u;
          DFS_Visit(v);

   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

9

Getting Dressed



The figure below shows how topologically sorted vertices appear in reverse order of their finishing times



17/18 11/16 12/15 13/14 9/10 1/8 6/7 2/5 3/4

10

Running Time

- Time: $O(V+E)$
  - DFS algorithm is called once with a cost $O(V+E)$ plus
  - $O(1)$ time to insert each of $|V|$ vertices onto the front of the linked list

11

Strongly Connected Components (SCC)

- Digraphs are often used to model communication and transportation networks
- People want to know that the networks are complete in the sense that from any location it is possible to reach another location in the digraph
- A directed graph is ***strongly connected*** if, for every pair of vertices u and v there is a path from u to v.
  - Note that in a directed graph, the existence of a path from u to v does not imply there is a path from v to u.

12

Strongly Connected Components (SCC)

- For finding strongly connected components of a graph G uses the transpose of G.
- Given $G = (V, E)$, the transpose of G is
  - $G^T = (V, E^T)$ where
    - $E^T = \{(u, v) : (v, u) \in E\}$
- G and G^T have exactly the same SCCs:
  - u and v are reachable from each other in G iff they are reachable from each other in G^T

13

Strongly Connected Components (SCC)

Strongly-Connected-Components(G)

1. call DFS(G) to compute finish times $f[u]$ for each vertex u
2. compute G^T
3. call DFS(G^T) and consider vertices in order of decreasing $f[u]$, computed in step 1
4. output vertices of each tree in DFS(G^T) as separate strongly connected component

- After decomposition, the algorithm is run separately on each strongly connected component.
- The solutions are then combined according to the structure of connections between components.
- Total running time $\Theta(V + E)$
  - Since the time to build G^T is $O(V + E)$ plus
  - $O(V + E)$ to call DFS twice

14

Strongly Connected Components (SCC)

STRONGLY-CONNECTED-COMPONENTS (G)

1. initialize stack S to empty, and call DFS(G) with the following modification:
   - push vertices onto stack S in the order they finish their DFS-VISIT calls.
   - That is, at the end of the procedure DFS-VISIT(u) add the statement
     - PUSH(u, S) – (there is no need to compute $d[u]$ and $f[u]$ values explicitly).
2. construct the adjacency-list structure of G^T from that of G.
3. call DFS(G^T) with the following modification:
   - initiate DFS-roots in stack-S-order, i.e., in the main DFS algorithm perform the following:

```
while S ≠ ϕ do
        u = POP(S)
        if color[u] = white then DFS-VISIT(u)
end {while}
```

4. each DFS-tree of step 3 (plus all edges between its vertices) forms an SCC.

Step-1 of SCC

```
DFS(G)
{
   for each vertex u ∈ V[G]
   {
      color[u] = WHITE;
      p[u] = NIL;
   }
   time = 0;
   for each vertex u ∈ V[G]
   {
      if (color[u] == WHITE)
         DFS_Visit(u);
   }
}
```

```
DFS_Visit(u)
{
   color[u] = GRAY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]

      if (color[v] == WHITE)
           p[v] = u;
           DFS_Visit(v);

   color[u] = BLACK;
   time = time+1;
   f[u] = time;
   PUSH(u, S);
}
```

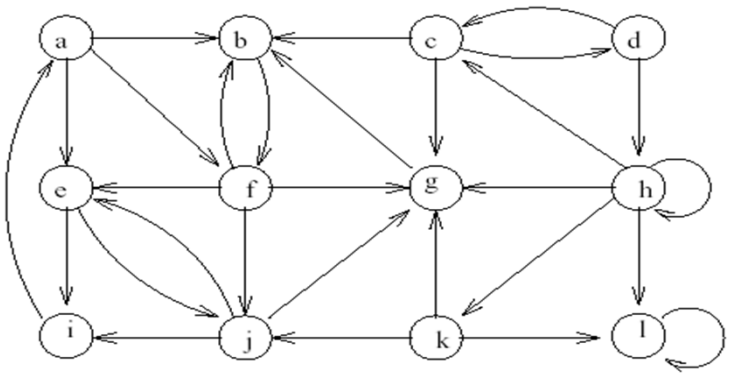
Step-3 of SCC

```
DFS(G^T)
{
   for each vertex u ∈ V[G]
   {
      color[u] = WHITE;
      p[u] = NIL;
   }
   time = 0;

   while S ≠ ϕ do
         u = POP(S)

    {
         if color[u] = white
         then DFS-VISIT(u)
     }
}
```

```
DFS_Visit(u)
{
   color[u] = GRAY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]

      if (color[v] == WHITE)
          p[v] = u;
          DFS_Visit(v);

   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

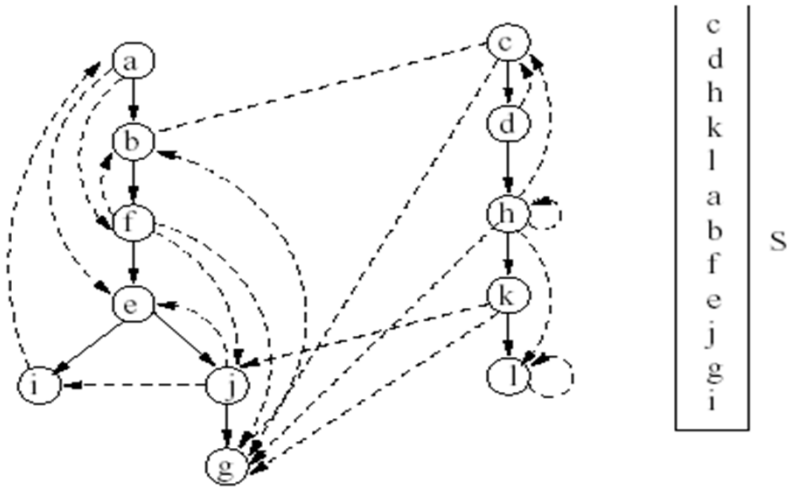
17

Example: Graph G



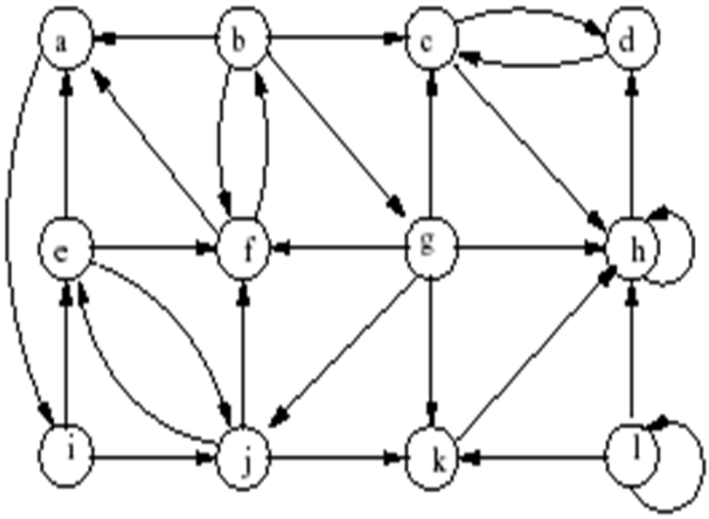
18

Step 1 : DFS(G)



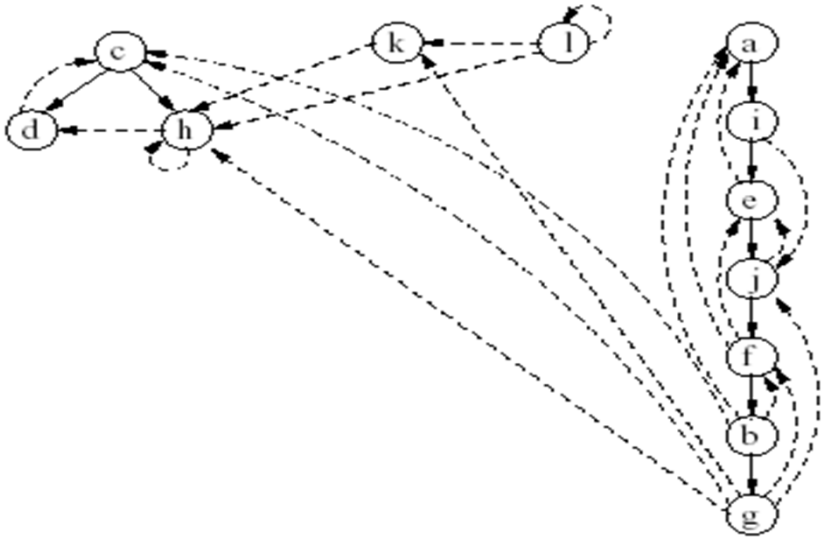
19

Step 2 : G^T



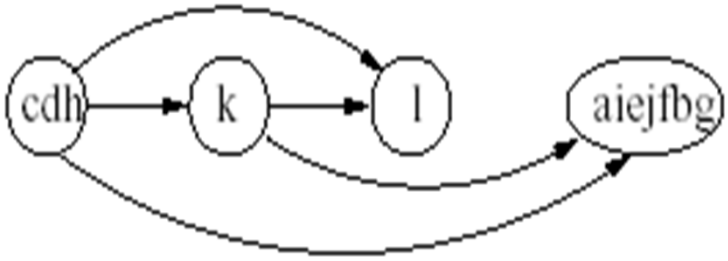
20

Step 3 : DFS(G^T)



21

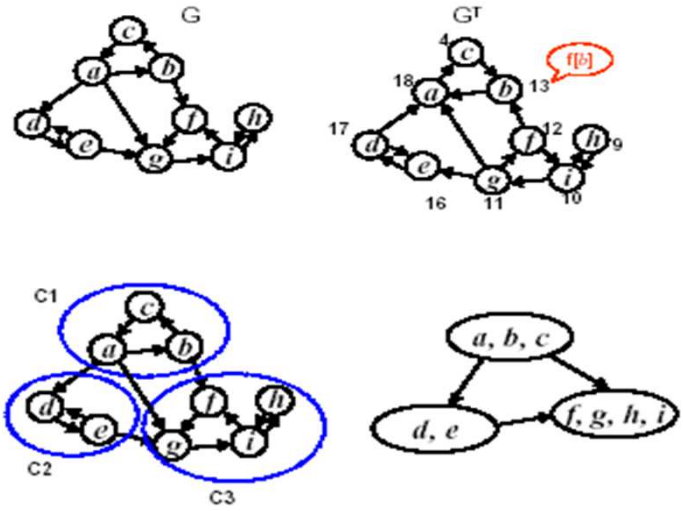
Step 4 : SCC of G



22

Strongly Connected Components

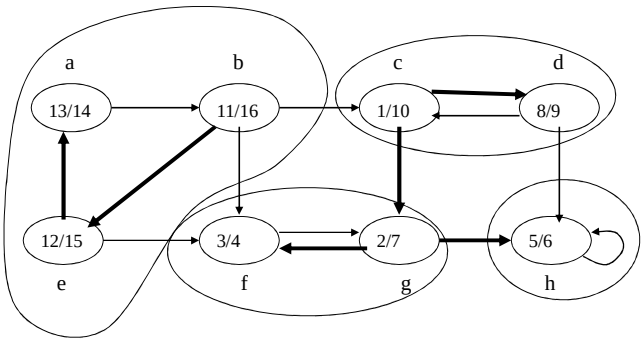
a	1	18
d	14	17
e	15	16
b	2	13
f	5	12
g	6	11
i	7	10
h	8	9
c	3	4



23

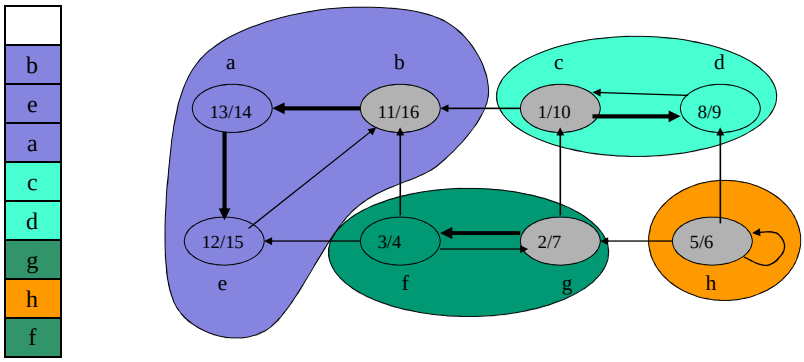
Book Example: G

b
e
a
c
d
g
h
f



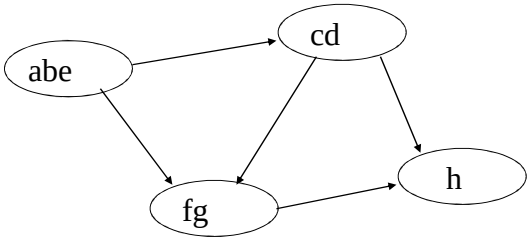
24

Book Example: G^T



25

Book Example: SCCs of G



26

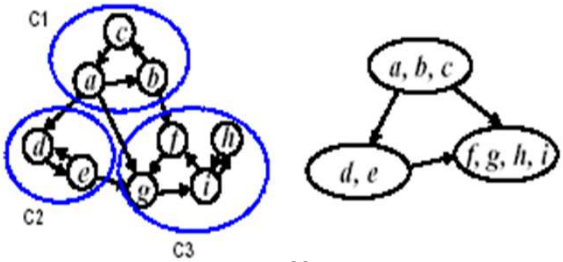
Component DAG

- The main idea comes from the component graph $G^{SCC} = (V^{SCC}, E^{SCC})$
- Suppose that G has k SCC $C_1, C_2, \ldots, C_k$
  - $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$: vertex v_i for each C_i
  - $(v_i, v_j) \in E^{SCC}$ if G contains a **directed edge** (x, y) for some $x \in C_i$ and $y \in C_j$

27

Component DAG

- If we merge the vertices in each SCC into a single super vertex,
- and join two **super vertices** C_i and C_j iff there are vertices $v_i \in C_i$ and $v_j \in C_j$ such that $(v_i, v_j) \in E$,
- then the **resulting digraph**, called the **component digraph**, is necessarily **acyclic**.



28