Announcements

- Reading: Chapter 22
- Extra Help Sessions on Nov. 15, 29, and Dec. 8 in SN011 (after class)
- Final Review Dec. 12, 3-5pm, SN115

Properties of DFS

- Predecessor subgraph G_π forms a forest of trees (the structure of a depth-first tree mirrors the structure of DFS-Visit)
- The discovery and finishing time have *parenthesis structure*, i.e. the parentheses are properly nested. (See the figures and next theorem)

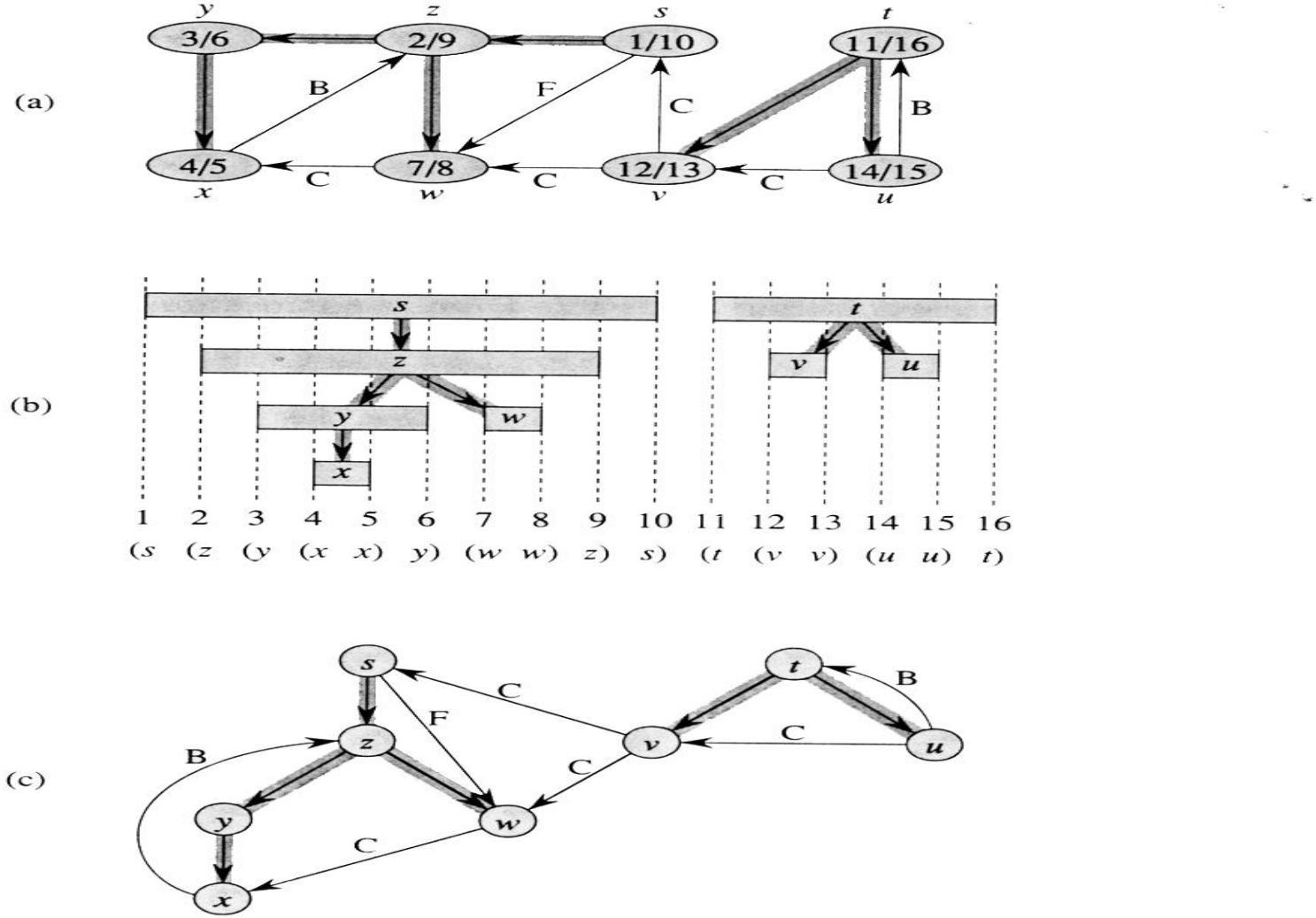
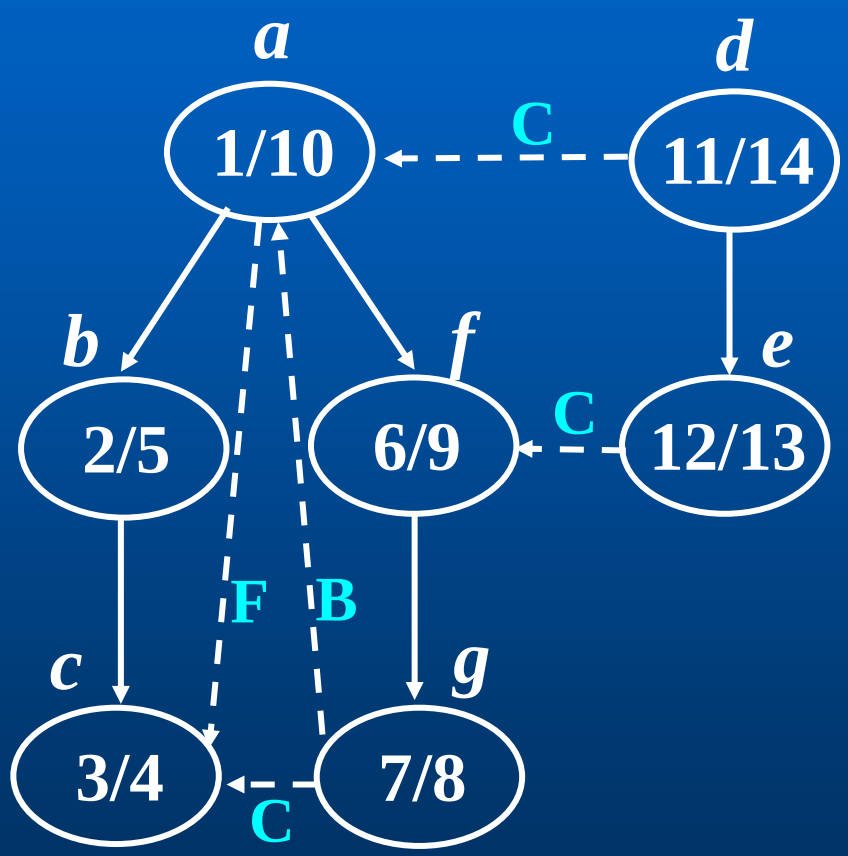
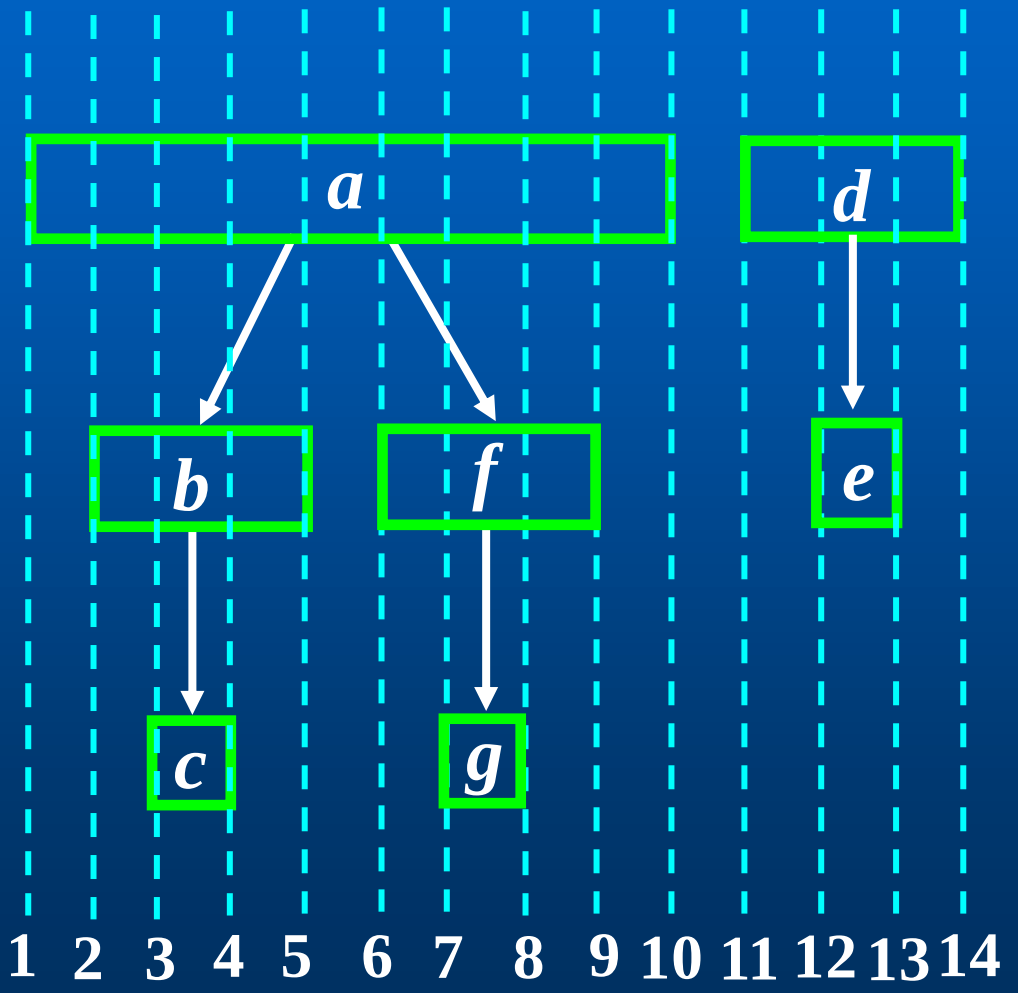


Figure 23.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 23.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

DFS & Parenthesis Lemma





Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices u and v, exactly one of the followings holds:

- the interval $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint
- the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in the depth-first tree, or
- the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in the depth-first tree

Nesting of Descendent' Intervals

- Vertex v is a proper descendant of vertex u in the depth-first forest for a (direct or undirected) graph G if and only if $d[u] < d[v] < f[v] < f[u]$

Proof follows directly from Parenthesis Theorem (see the complete proof in the textbook CLR)

White-Path Theorem

- In a depth-first forest of a graph G, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Proof:

$\Rightarrow$ Assume v is a descendant of u and use Corollary on Nesting of Descendant's Interval

$\Leftarrow$ By contradiction: suppose that v is reachable from u but v doesn't become a descendant of u. Use Parenthesis Theorem and NDI Corollary.

Classification of Edges

DFS can be used to classify edges of G:

1. ***Tree edges*: edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).**
2. ***Back edges*: edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered to be back edges.**
3. ***Forward edges*: nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.**
4. ***Cross edges*: all other edges.**

Algorithm for Edge-Classification

- Modify DFS so that each edge (u, v) can be classified by the color of the vertex v that is reachable when the edge is first explored:

1. WHITE indicates a tree edge
2. GRAY indicates a back edge
3. BLACK indicates a forward or cross edges

Theorem

- In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

This lays the foundation for topological sort and many other applications (to be discussed in the next lecture).

Directed Acyclic Graph

- A directed acyclic graph (DAG) arises in many applications where there are precedence or ordering constraints (e.g. scheduling problems). For instance, if there are a series of tasks to be performed, and certain tasks must precede other tasks (e.g. in construction you must build the first floor before the second, but you can do the electrical wiring while you install the windows).
- 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.

Topological Sort

- Given a directed acyclic graph (DAG) $G = (V, E)$, topological sort is a linear ordering of all vertices of the DAG such that if G contains an edge (u, v), u appears before v in the ordering.
- In general, there may be many legal topological orders for a given DAG.

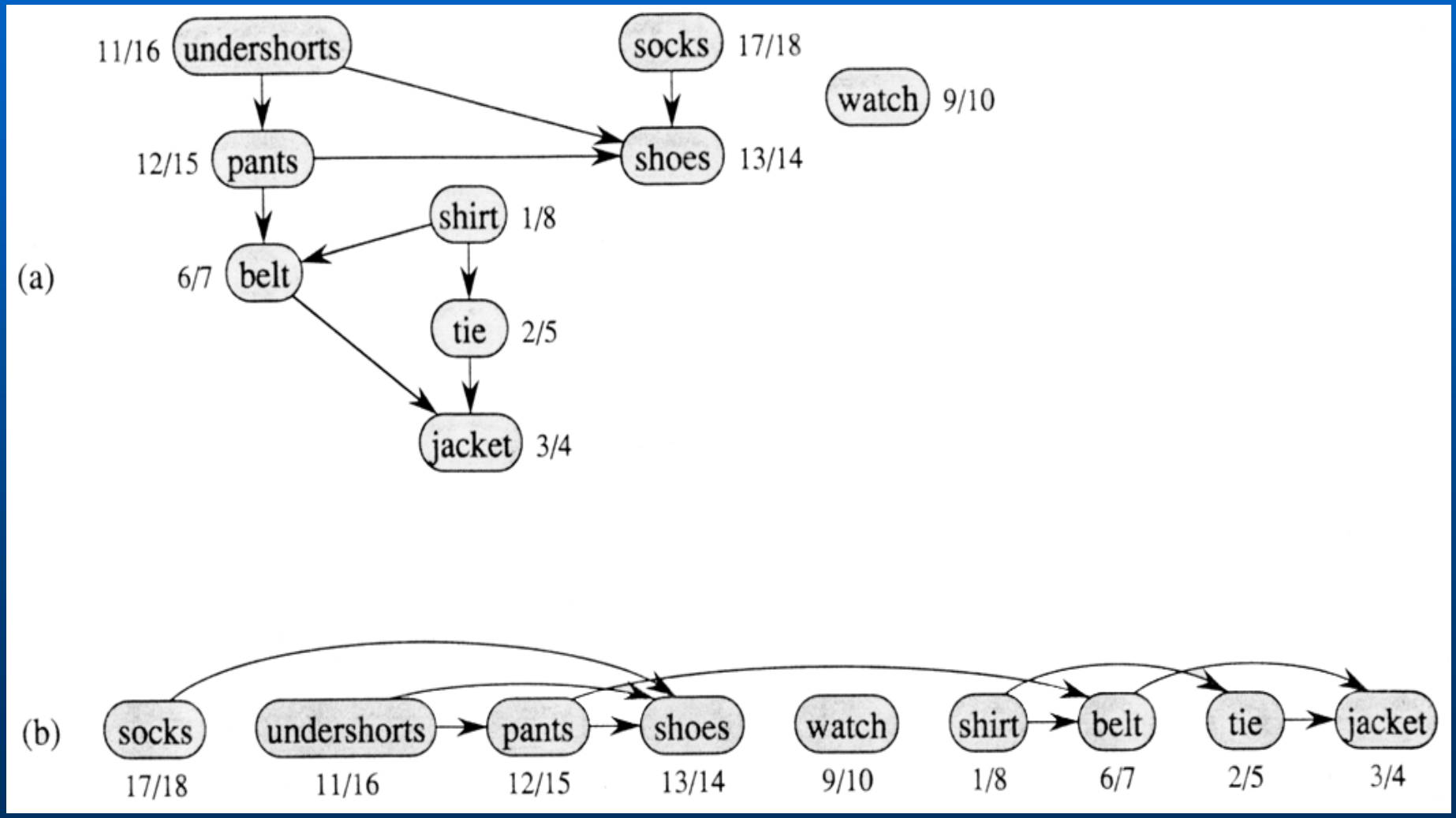
Topological-Sort (G)

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

2. As each vertex is finished, insert it onto the front of linked list

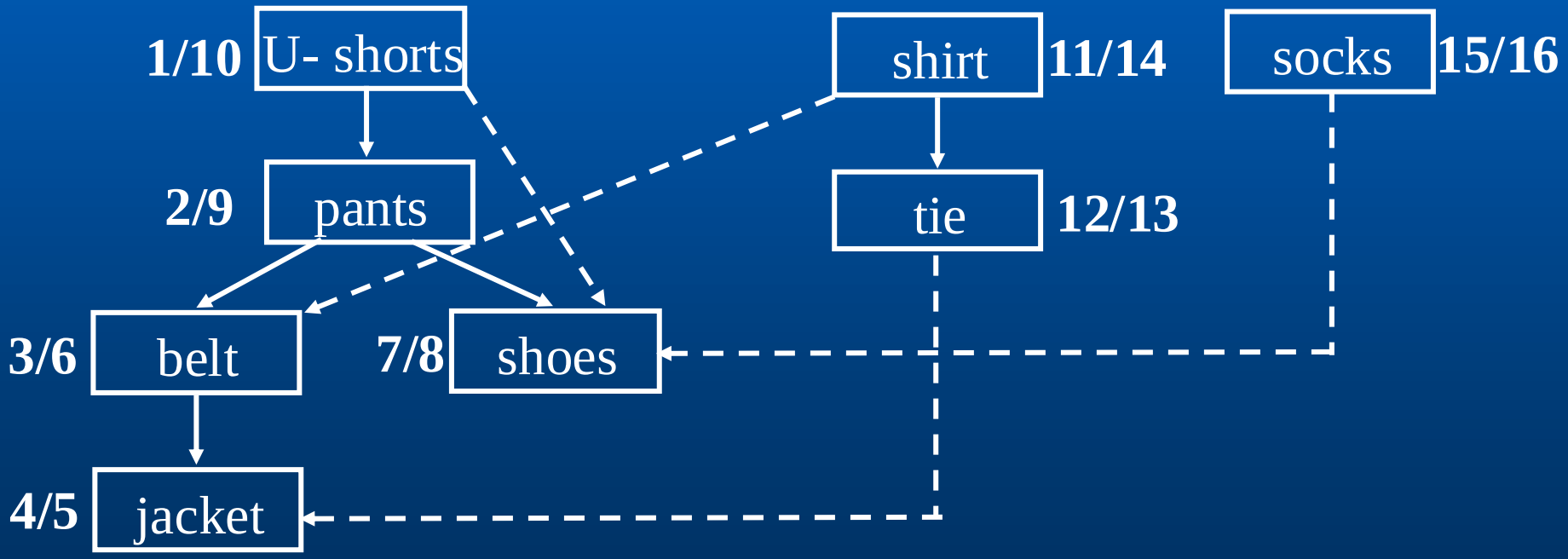
3. Return the linked list of vertices

An Example


11/16 undershorts
socks 17/18
watch 9/10
12/15 pants
shoes 13/14
shirt 1/8
(a)
6/7 belt
tie 2/5
jacket 3/4
(b)
socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

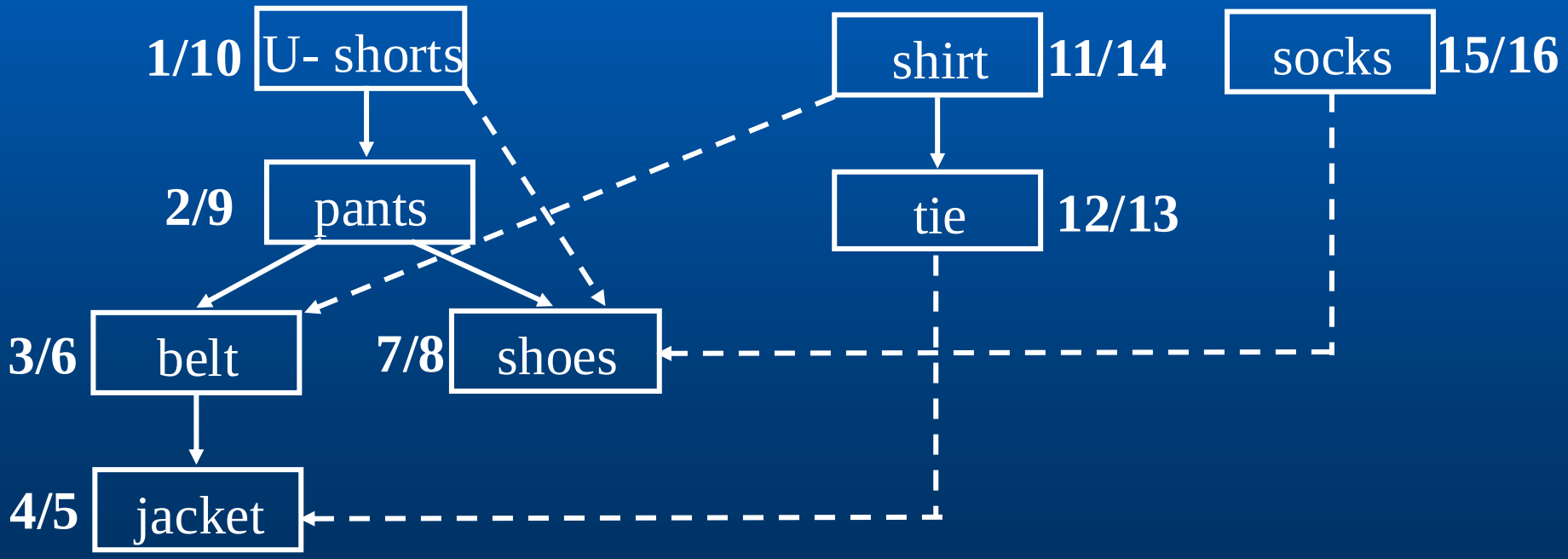
Another Example

- Consider the example given in the book, but we do our DFS in a different order, so we get a different final ordering. But, both are legitimate, given the precedence constraints.



Another Example

- Consider the example given in the book, but we do our DFS in a different order, so we get a different final ordering. But, both are legitimate, given the precedence constraints.



Final Ordering: socks, shirt, tie, u-shorts, pants, shoes, belt, jacket

Analysis of Topological-Sort

- Line1: DFS takes $\Theta(V+E)$
- Line 2: Each insertion takes $O(1)$ for $|V|$ vertices
- Total: $\Theta(V+E)$ time

A Compact Version: TopSort(G)

```
1.  for each u∈V color[u] = white;        // initialization
2.  L = new linked_list;                  // L: empty linked list
3.  for each u∈V
4.        if (color[u] == white) TopVisit(u);
5.  return L;                             // L gives final order

TopVisit(u) {                             // start search at u
1.  color[u] = gray;                      // mark u visited
2.  for each v∈Adj(u)
3.        if (color[v] == white) TopVisit(v);
4.  append u to the front of L            // finish & add u to L
}
```

Lemma

- Consider a directed graph G and a forest of DFS for G. G has a cycle if and only if the DFS forest has a back edge.

Proof:

(⇐) If there is a back edge (u, v) , then v is an ancestor of u, and by following tree edges from v to u, we get a cycle.

(⇒) By contrapositive: suppose there are no back edges. By White-Path Theorem, each of the remaining types of edges, tree, forward and cross all have the property that they go from vertices with higher finishing time to vertices with lower finishing time. Thus, along any path, finish time monotonically decrease. This implies there can be no cycle.

Applications of DFS

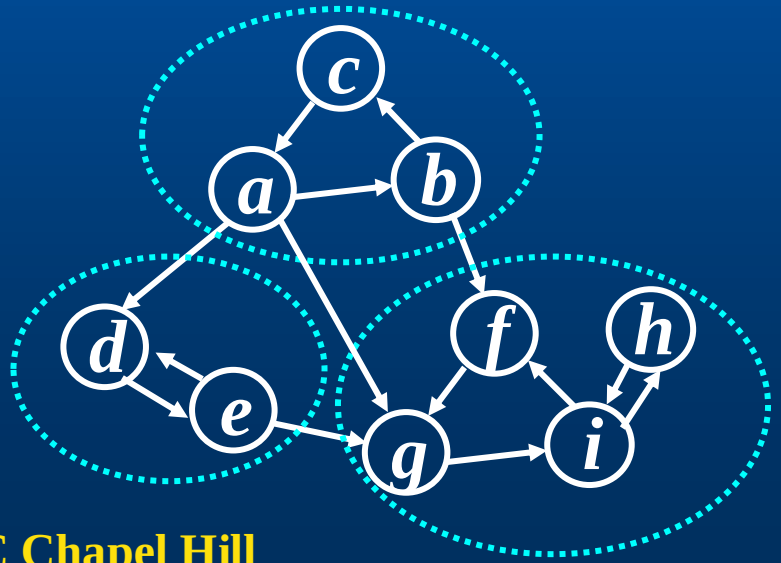
- Decomposing a directed graph into its strongly connected components
- Determining articulation points, bridges & biconnected components of an undirected graph

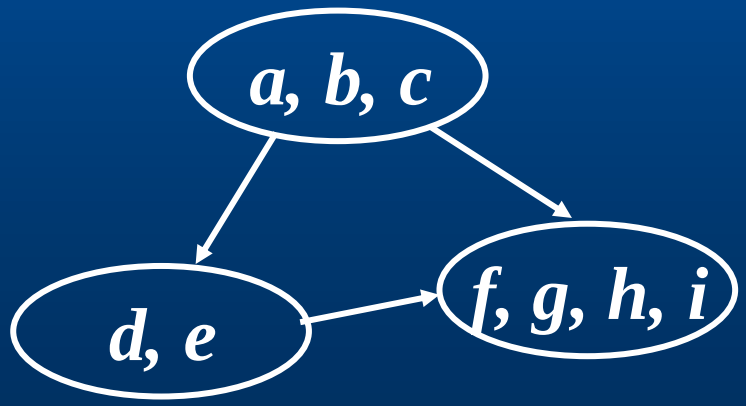
Strongly Connected Components

- Digraphs are often used in communication and transportation networks, where 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 digraph is "*strongly connected*", if for every pair of vertices $u, v \in V$, u can reach v and vice versa. We say that two vertices u and v are "*mutually reachable*". Mutual reachability is an equivalence relation.

Component DAG

- If we merge the vertices in each strong component into a single *super vertex*, and joint two super vertices (A, B) if and only if there are vertices $u \in A$ and $v \in B$ such that $(u, v) \in E$, then the resulting digraph, called the *component digraph*, is necessarily acyclic.





Finding Strongly Connected Components

- Use depth-first search twice
- Trick: Reverse directions of edges, search in reverse order of finishing times (topological order for reversed graph)
- But why do we need this trick…?

Strong Components & DFS

- Each strong component is just a subtree of DFS Forest. Is it always true for any DFS? (See Figure 2 on the board or the handout)
- Does there always exist a way to order the DFS such that it is true?

Ordering DFS

- Once the DFS starts within a given strong component, it must visit every vertex within the component (and possibly some others) before finishing.
- The search may “leak out” into other strong components, so doesn’t give us a list of strong components.
- But, if we visit nodes in reverse topological order, each search cannot leak out into other components, since those components would have been visited earlier in the search.

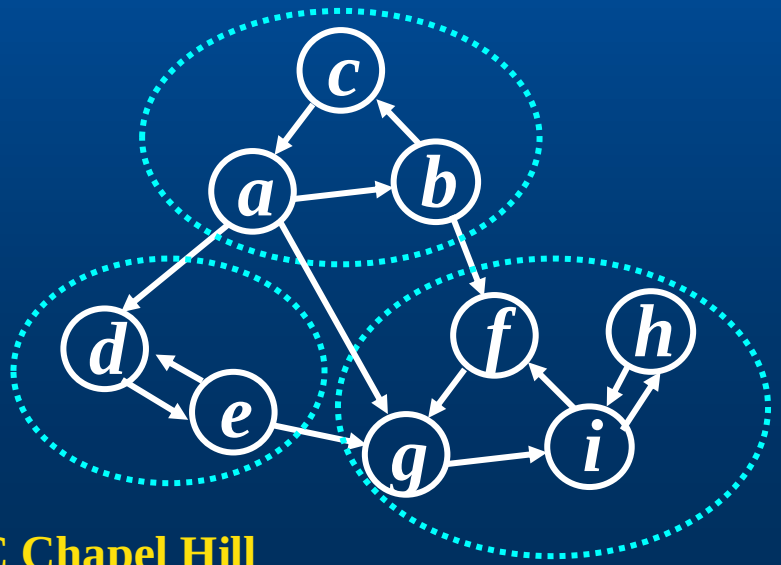
StrongComp(*G*)

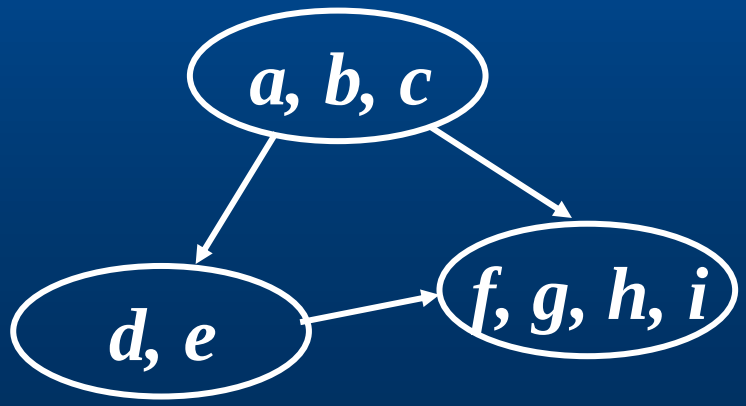
1. Run DFS(G), computng finish time *f*[*u*] for each vertex *u*
2. Compute *R* = Reverse(*G*), reversing all edges of *G*
3. Sort the vertices of *R* (by CountingSort) in decreasing order of *f*[*u*]
4. Run DFS(*R*) using this order
5. Each DFS tree is a strong component; output vertices of each tree in the DFS forest

Total running time is Θ(n + e)

Correctness (Intuition)

- At least one component must have all edges incoming. Want to visit that one first, to avoid leakage.
- If you reversed the edges, that component would be the first one in a topological sort.





Correctness (Intuition)

- Fortunately, reversing the edges doesn't change the strongly connected components. That is, mutual reachability between two vertices is the same for G and G^T.

- So: Do a topological sort, then reverse the edges, then do DFS in sorted order.

Correctness

- Refer to the textbook for a formal proof. Here is some intuition why it works:
  - Strong components are not affected by reversing all edges in the digraph. i.e. mutual reachability between two vertices are the same in G and G^T
  - The main idea is to visit the strong components in a reverse topological order so to avoid "leak".
  - If we simply visit the components in reverse topologoical order, we would visit the vertices in increasing order of finish time, starting with the lowest finishing time. (This won't work)
  - Consider max finish time in each component. Visit the components related to the topological order.

Lemma

- **Consider a digraph G on which DFS has been run. Label each component with the maximum finish time of all the vertices in the component, and sort these in decreasing order. Then this order is a topological order for the component digraph.**

Example (Figure 3): The maximum finish time for each component are 18 (for $\{a, b, c\}$), 17 (for $\{d,e\}$), and 12 (for $\{f, g, h, I\}$). The order $<18, 17, 12>$ is a valid topological order for the component digraph.

Definitions

- *Articulation Point* (or *cut vertex*): any vertex whose removal (together with any incident edges) results in a disconnected graph
- *Bridge*: an edge whose removal results in a disconnected graph
- *Biconnected*: containing no articulation points. (In general a graph is k -connected, if k vertices must be removed to disconnect the graph.)
- *Biconnected Components*: a maximal set of edges s.t. any 2 edges in the set lie on a common simple cycle

Articulation Points and DFS

- *Basic idea:* use the structures in the DFS tree to find articulation points.

- Assume G is connected, so there is only one tree in the DFS forest. Since G is undirected, we don't distinguish between forward & back edges -- we call them both back edges. There are no cross edges.

Observations

1. An internal vertex u of the DFS tree (other than the root) is an articulation point if and only if there exists a subtree rooted at a child of u such that there is no back edge from any vertex in this subtree to a proper ancestor of u.
2. A leaf of the DFS tree is never an articulation point.
3. The root of the DFS tree is an articulation point if and only if it has two or more children.

Articulation Points by DFS

- Observation 2 & 3 are easy to check.
- Observation 1: is there some back edge from some subtree to an ancestor of a given vertex? Keep track of back edge that goes highest in the tree (cloest to the root). We do this by keeping track of "discovery time": i.e. find the back edge (v, w) with the smallest $d[w]$.

Basic Algorithm

- Given G, compute the DFS tree, $T(G)$.
- Traverse $T(G)$ in postorder. For each node v visited, calculate $low[v]$ defined as

$$low[v] = \min(d[v], d[w])$$

(u, w): a back edge for some descendant u of v

- Articulation points are determined as follows:
  - Root of $T(G)$, if and only if it has more than 1 child
  - A node v other than the root, if and only if has a child x such that $low[x] \geq d[v]$

ArtPt(u)

```
1.  Color[u] = gray
2.  Low[u] = d[u] = ++time
3.  for each (v ∈ Adj(u)) {
4.        if (color[v] == white)                  // (u, v) is a tree edge
5.           pred[v] = u
6.           ArtPt(v)
7.           Low[u] = min(Low[u], Low[v])  // update Low[u]
8.           if (pred[v] == NULL) {              // root:  apply Observation 3
9.              if (v is u's second++ child)
10.                Add u to set of articulation points
11.          }
12.          else if (Low[v] ≥ d[u]) {           // internal node: apply Obs. 1
13.                   Add u to set of articulation points
14.          }
15.       else if (v ≠ pred[u])                          // (u, v) is a back edge
16.                Low[u] = min(Low[u], d[v])            // update Low[u]
17.  }
```