Data Structures for Disjoint Sets

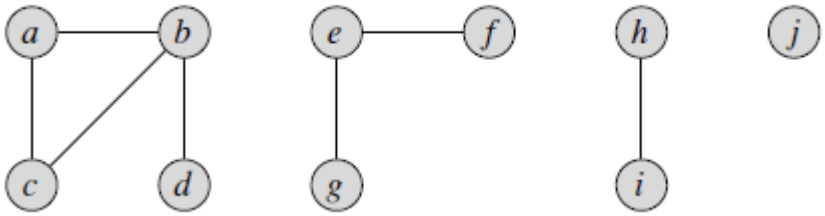
- Some applications involve grouping ***n*** distinct elements into a collection of **disjoint** sets. (Two important operations are then finding which set a given element belongs to and uniting two sets.)
- A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.
- Each set is identified by a representative, which is some member of the set.
- Each element of a set is represented by an object. Letting ***x*** denote an object, we wish to support the following **three** operations:
  1) MAKE-SET(x): creates a new set whose only member (and thus representative) is x.
     - Since the sets are disjoint, we require that x not already be in some other set.
  2) UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets.
     - The two sets are assumed to be disjoint prior to the operation.
     - The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative.
     - Since we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y, removing them from the collection S.
  3) FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

NOTE:

1. we shall analyze the running times of disjoint-set data structures in terms of two parameters:
   n: the number of MAKE-SET operations and
   m: the total number of MAKE-SET, UNION, and FIND-SET operations.
2. Since the sets are disjoint, each UNION operation reduces the number of sets by one. After $n - 1$ UNION operations, therefore, only one set remains. The number of UNION operations is thus at most $n - 1$.
3. Note also that since the MAKE-SET operations are included in the total number of operations m, we have $m \geq n$.
4. We assume that the n MAKE-SET operations are the first n operations performed.

An application of disjoint-set data structures:

- One of the many applications of disjoint-set data structures arises in determining the connected components of **an undirected graph**. Figure given below shows a graph with four connected components.d



- The following procedure uses the disjoint-set operations to compute the connected components of a graph.

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ V[G]
2       do MAKE-SET(v)
3  for each edge (u, v) ∈ E[G]
4       do if FIND-SET(u) ≠ FIND-SET(v)
5            then UNION(u, v)
```

- **Note**: In an actual implementation of this connected-components algorithm, the representations of the graph and the disjoint-set data structure would need to reference each other. That is, an object representing a vertex would contain a pointer to the corresponding disjoint-set object, and vice-versa. These programming details depend on the implementation language, and we do not address them further here.

Example:

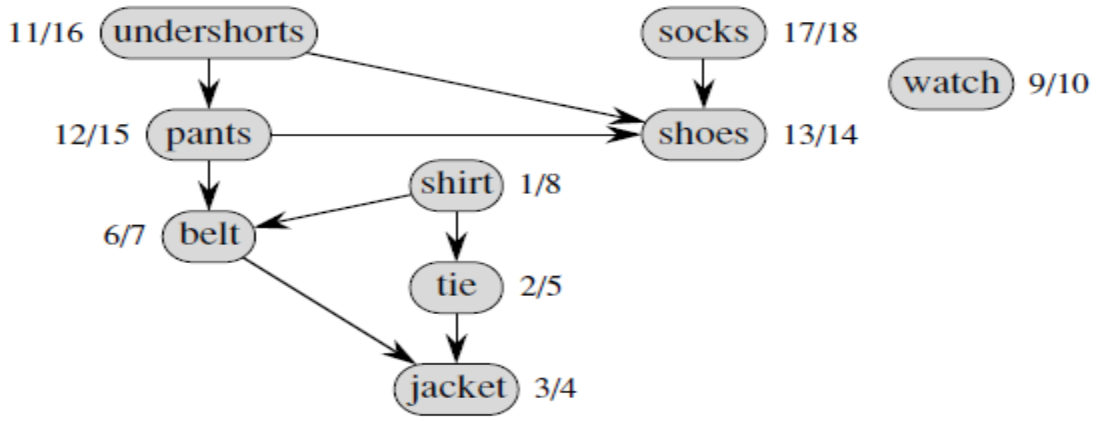
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

- This initially places each vertex v in its own set. Then, for each edge (u, v), it unites the sets containing u and v.
- It should be noted that after all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set.

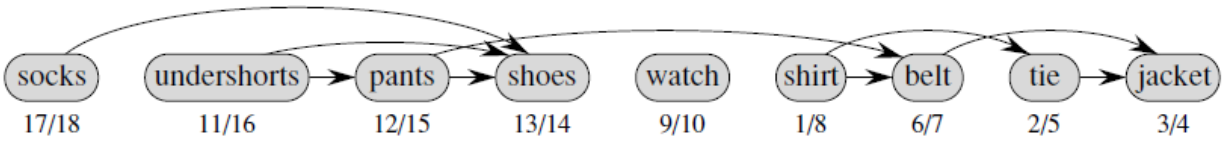
TOPOLOGICAL-SORT*(G)*

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Example: Figure below shows that how a person 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 finishing times from a depth-first search are shown next to each vertex.



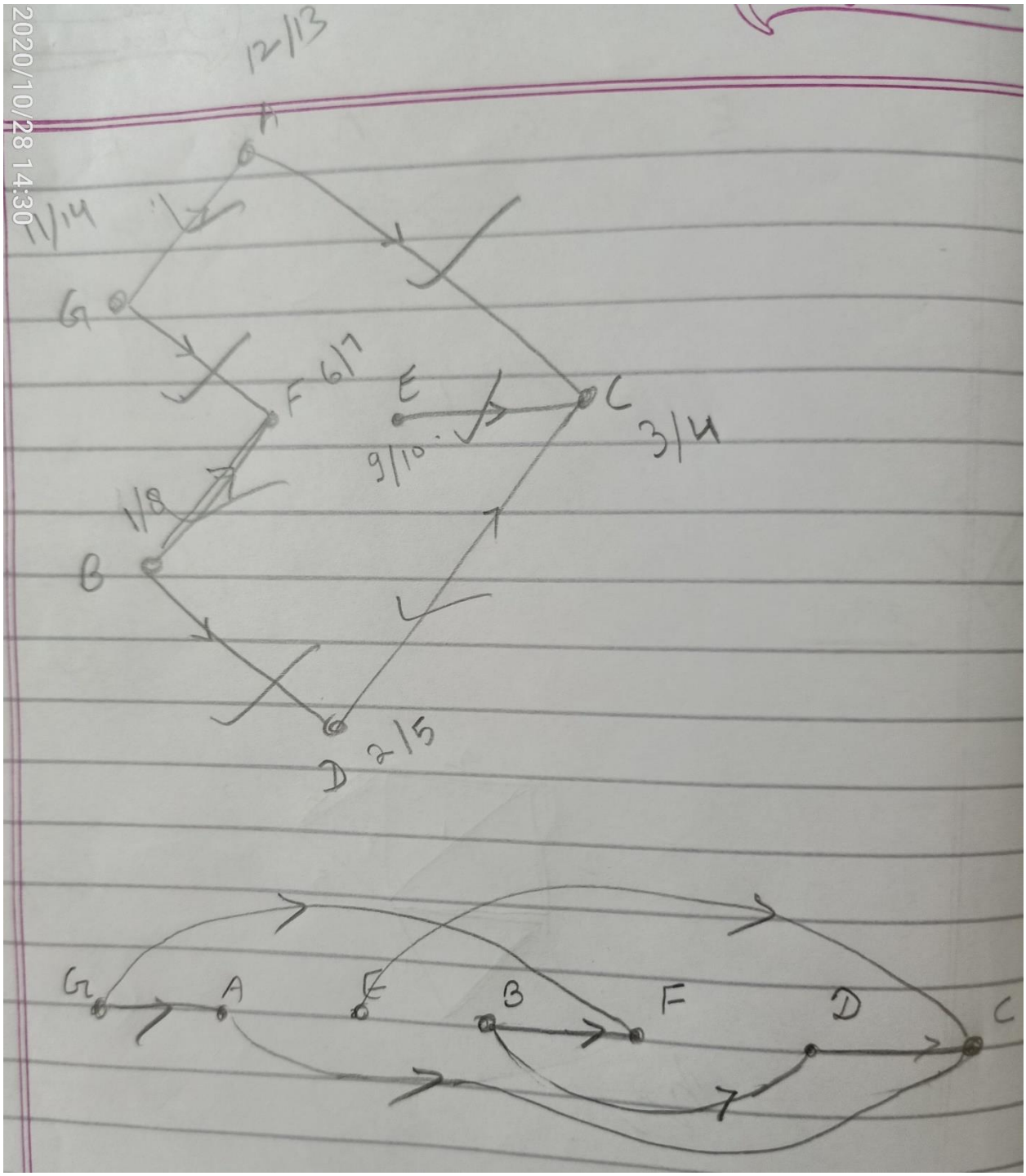
Further, the above graph is redrawn for demonstrating a topological ordering. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right.



Running Time:

We can perform a topological sort in time Θ(V+E), since depth-first search takes Θ(V + E) time and it takes O(1) time to insert each of the |V| vertices onto the front of the linked list.

Example:



Theorem: A directed graph *G* is acyclic if and only if a depth-first search of *G* yields no back edges.

Proof: Suppose that there is a back edge (u, v). Then, vertex v is an ancestor of vertex u in the depth-first forest. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.

Now, suppose that *G* contains a cycle *c*. We show that a depth-first search of *G* yields a back edge….

Theorem: TOPOLOGICAL-SORT(G) produces a topological sort of a directed acyclic graph G.

Proof: It suffices to show that for any pair of distinct vertices u, v $\in$ V, if there is an edge in dag G from u to v, then f [v] < f [u].