Analysis of Algorithms

Graph Algorithms

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.

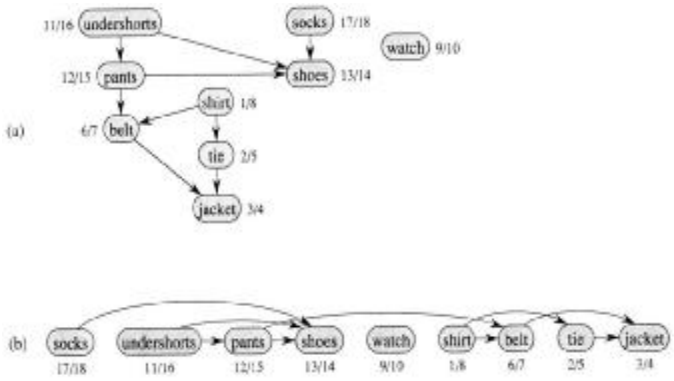
Topological Sort

- 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.
- Or, the problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraint are satisfied.
- In general, there may be many legal topological orders for a given DAG.

Topological-Sort (G)

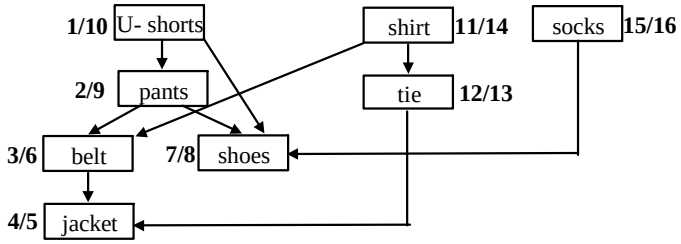
1. Call DFS(G) to compute finishing time $f[v]$ for each vertex
2. As each vertex is finished, insert it onto the front of linked list
3. Return the linked list of vertices

An Example



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

Strongly Connected Components

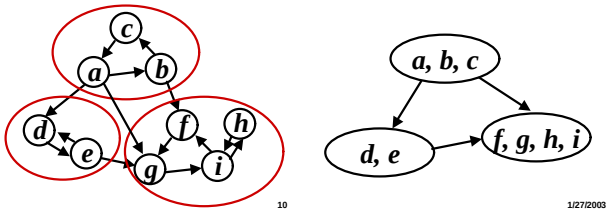
- Connectivity in undirected graphs is rather straightforward: A graph that is not connected is naturally and obviously decomposed in several connected components.
- Each restart of the algorithm marks a new connected component.

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 (reflexive, symmetric, and transitive).

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.



Component DAG

- The reason is simple: A cycle containing several strongly connected components would merge them all to a single strongly connected component.
- We can restate this observation as follows: "*Every directed graph is a dag of its strongly connected components.*"

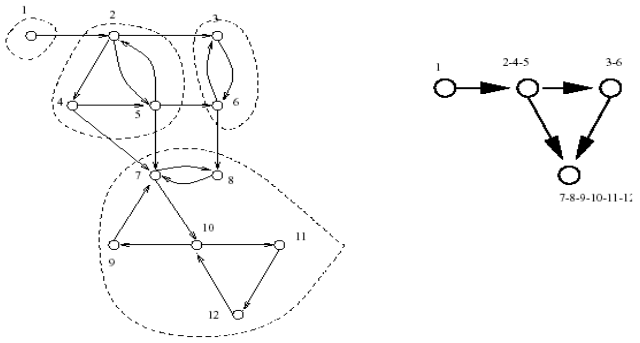
Conclusion about digraph

- At the top level we have a dag, a rather simple structure.
- A dag is guaranteed to have at least one source (a node without incoming edges) and at least one sink (a node without outgoing edges), and
- can be topologically sorted.
- We could look inside a node of the dag to see the full-fledged strongly connected component.

Properties of DFS

- Property 1: If depth-first search of a graph is started at a node u, then it will get stuck and restarted precisely when all nodes that are reachable from u have been visited.
- Therfore, if depth-first search is started at a node of a sink strongly connected component (a strongly connected component that has no edges leaving it in the dag of strongly connected components), then it will get stuck after it visits precisely the nodes of this strongly connected component.

Example



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.
- If we don't start with "reverse topological order", then the search may "leak out" into other strong components (see Fig. 2 right). However, by visiting components in reverse topological order of the component tree, each search cannot leak out into other components, since they would have already 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 $\Theta(V + E)$