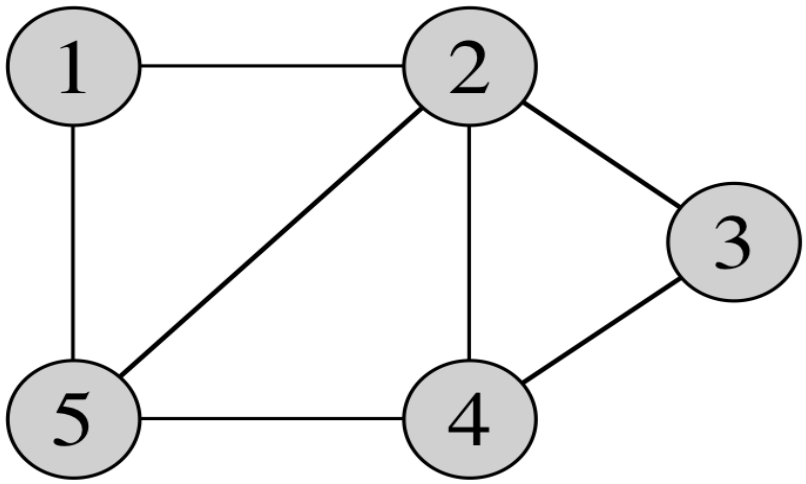
Chapter 22 Elementary Graph Algorithms

- A graph is a set of vertices and a set of edges connecting pairs of vertices. Formally, a graph G = (V, E), where V is a set of vertices and E is a set of pairs, (u, v), where both u and v are elements of V.

- If there is an edge connecting two vertices, u and v, then u and v are adjacent to each other.

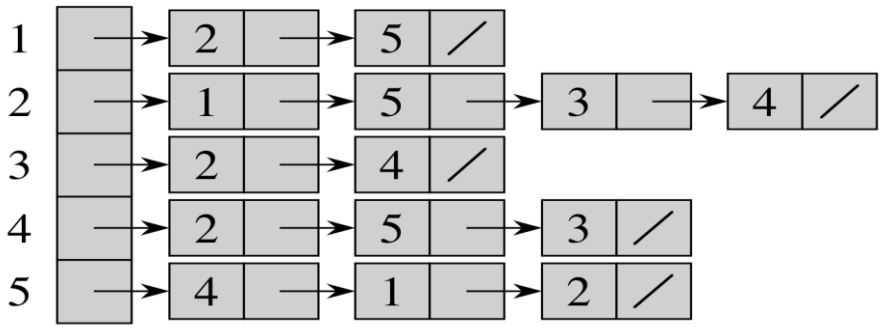
- The number of vertices in G = (V, E) will be denoted by |V| and the number of edges by |E|. The running time of a graph algorithm may depend on both |V| and |E|.

- An example graph with 5 vertices and 7 edges is shown below:



- How can this graph be represented in a computer? One way uses adjacency lists. The **adjacency-list representation** of a graph G = (V, E) consists of an array Adj of |V| lists, one for each vertex in V. For each $u \in V$, the adjacency list Adj[u] contains all the vertices v such that there is an edge $(u, v) \in E$. That is, Adj[u] consists of all the vertices adjacent to u in G. The example graph above has an array, Adj[1..5], of pointers (one for each

vertex). Adj[u] points to the head of a linked-list of vertices adjacent to vertex u. The vertices in each list are in arbitrary order. Each edge in the graph will appear twice in the set of lists; an edge connecting vertices u and v will appear in both the u-list and the v-list. The adjacency-list representation of the example graph is:



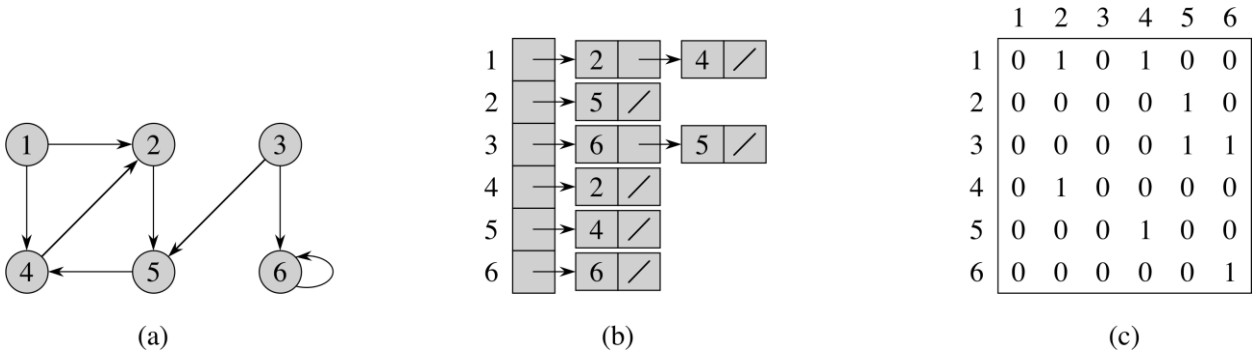
- Another way of representing the example graph in a computer is with an adjacency matrix. The example graph has a 5x5 matrix, A, with a row and a column for each vertex. $a_{ij} = \begin{cases} 1 & if\ (i,j)\ \in E, \\ 0 & otherewise \end{cases}$

  For the example, matrix A is:

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- A **directed graph** is a graph with directed edges; each edge is an arrow instead of a line. If there is an arrow running from vertex u to vertex v then v is adjacent to u. Is u adjacent to v? Only if the graph has another arrow running from v to u. Below is an example of a directed graph with 6

vertices and 8 edges, its adjacency-list representation, and its adjacency-matrix representation:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

- Which is the better representation of a graph? Adjacency lists take $\Theta(|V| + |E|)$ space while an adjacency matrix takes $\Theta(|V|^2)$ space; most graphs are sparse (only a few edges incident on each vertex) so adjacency-list take less space unless the graph is dense. Most graph algorithms traverse a graph from vertex to adjacent vertices; this is faster with the list representation for sparse graphs.

- A **weighted graph** has weights attached to the edges. Usually the weight attached to an edge represents the cost for traversing the edge. How is a weighted graph represented in a computer? With adjacency lists each node in the linked lists is augmented with a weight field. Also, with an adjacency matrix, A, entry A[i, j] equals the weight attached to the edge from vertex i to vertex j. If there is no edge from i to j then A[i, j] = ∞ to inhibit any attempt to traverse the non-existent edge.

Breadth-First Search (BFS)

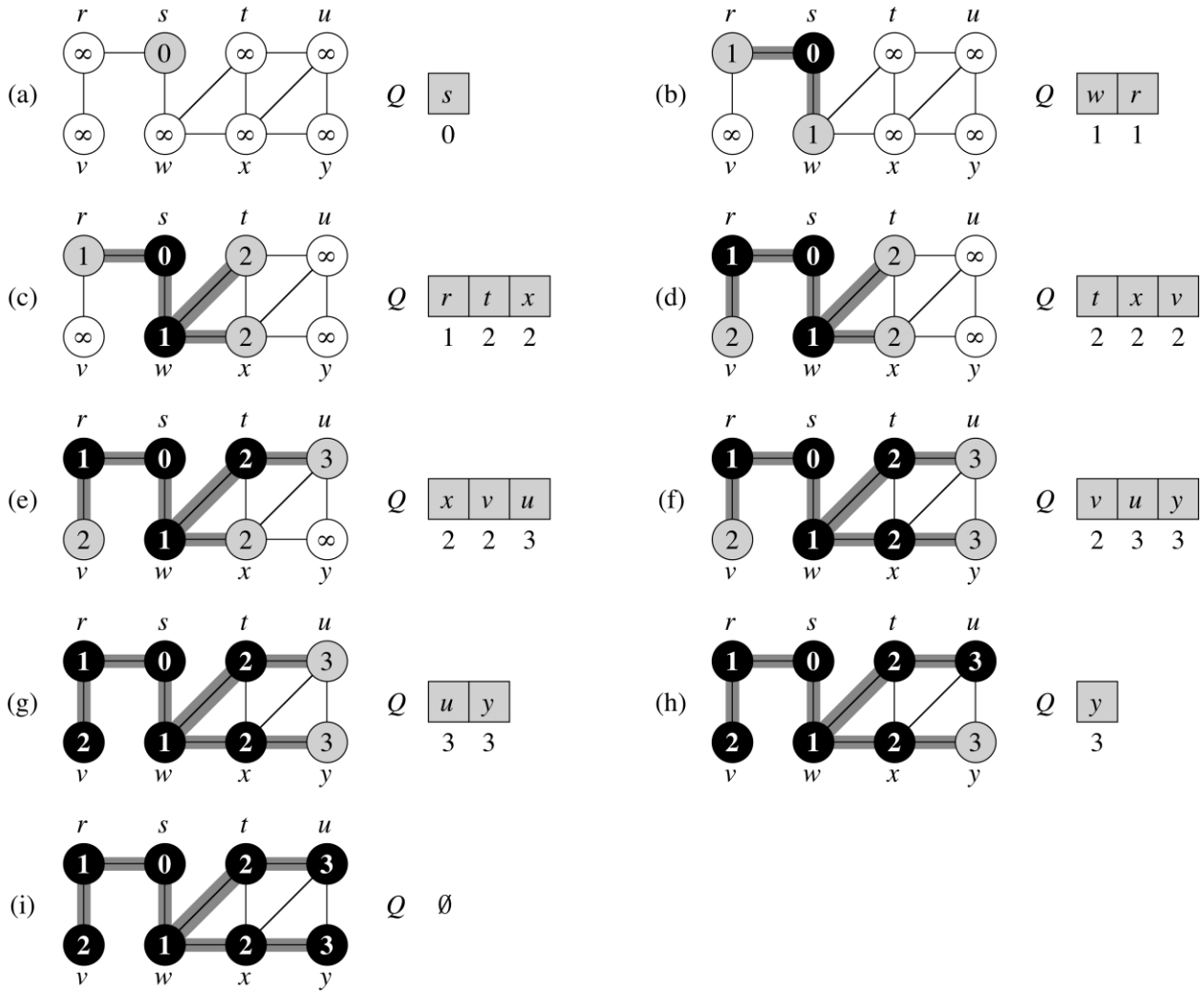
- Breadth-first search is one way to find all the vertices reachable from a given source vertex, s. Each vertex is in one of three states:
  1. Undiscovered
  2. Discovered but not fully explored; and
  3. Fully explored.

- A vertex is fully explored once all of its adjacent vertices have been examined. The state of a vertex, u, is stored in a color variable; color[u] = White, Gray, or Black for the three states, respectively.

- The algorithm, **BFS**, develops a breadth-first-search tree with the source vertex, s, as its root. The parent of any other vertex in the tree is the vertex from which it was first discovered. For each vertex, v, the parent of v is placed in $v.\pi$. Another variable, $v.d$, computed by **BFS** contains the number of tree edges on the path from s to v. **BFS** uses a **FIFO** queue, **Q**, to store gray vertices.

BFS(G, s)

```
1     for each vertex u ϵ G, V – {s}
2         u.color = WHITE
3         u.d = ∞
4         u.π = NIL
5     s.color = GRAY
6     s.d = 0
7     s.π = NIL
8     Q = Φ
9     ENQUEUE (Q, s)
10    while Q ≠ Φ
11        u = DEQUEUE (Q)
12        for each v ϵ G.Adj[u]
13            if v.color = = WHITE
14                v.color = GRAY
15                v.d = u.d + 1
16                v.π = u
17                ENQUEUE (Q, v)
18        u.color = BLACK
```



- The while-loop in BFS is executed at most |V| times. The for-loop inside the while-loop is executed at most |E| times if G is a directed graph or 2|E| time if G is undirected. The running time for BFS is *O(|V| + |E|).*

- **Lemma 22.3**: Suppose that during the execution of BFS on a graph G (V, E), the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \geq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for i = 1, 2, …, r -1.

- Let v be any vertex in V[G]. If v is reachable from s then let δ(s, v) be the minimum number of edges in E[G] that must be traversed to go from vertex s to vertex v. If v is not reachable from s then let δ(s, v) = ∞.

- **Theorem 22.5:** Let G = (V, E) be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge (v, π, v)

- BFS builds a tree called a breadth-first-tree containing all vertices reachable form s. The set of edges in the tree (called tree edges) contain $(\pi.v, v)$ for all v where $\pi.v \neq NIL$. If v is reachable from s then there is a unique path of tree edges from s to v. PRINT-PATH(G, s, v) prints the unique path.

PRINT-PATH(G, s, v)

```
1    if v = = s
2        print s
3    elseif v.π = = NIL
4        print "no path from " s "to" v "exists"
5    else PRINT-PATH(G, s, v. π)
6        print v
```

Depth-first Search

Depth-first search (DFS) is another way to find all the vertices reachable from a source vertex, s. Like BFS, each vertex is in one of the three states:

a) White = undiscovered
b) Gray = discovered but not fully explored
c) Black = fully explored

DFS selects a source vertex and takes a path deeper and deeper into the graph to discover new vertices until the path ends when no new vertices are found. Then it backtracks along the path and explores new edges to find more new vertices. When it has backtracked all the way back to the original source vertex it has built a DFS tree of all vertices reachable from that source. If there are still undiscovered

vertices then it selects one of them as the source for another DFS tree. The result is a forest of DFS-trees.

Like BFS, DFS uses $\pi.v$ to record the parent of v. $\pi.v = NIL$ if and only if v is the root of a DFS-tree.

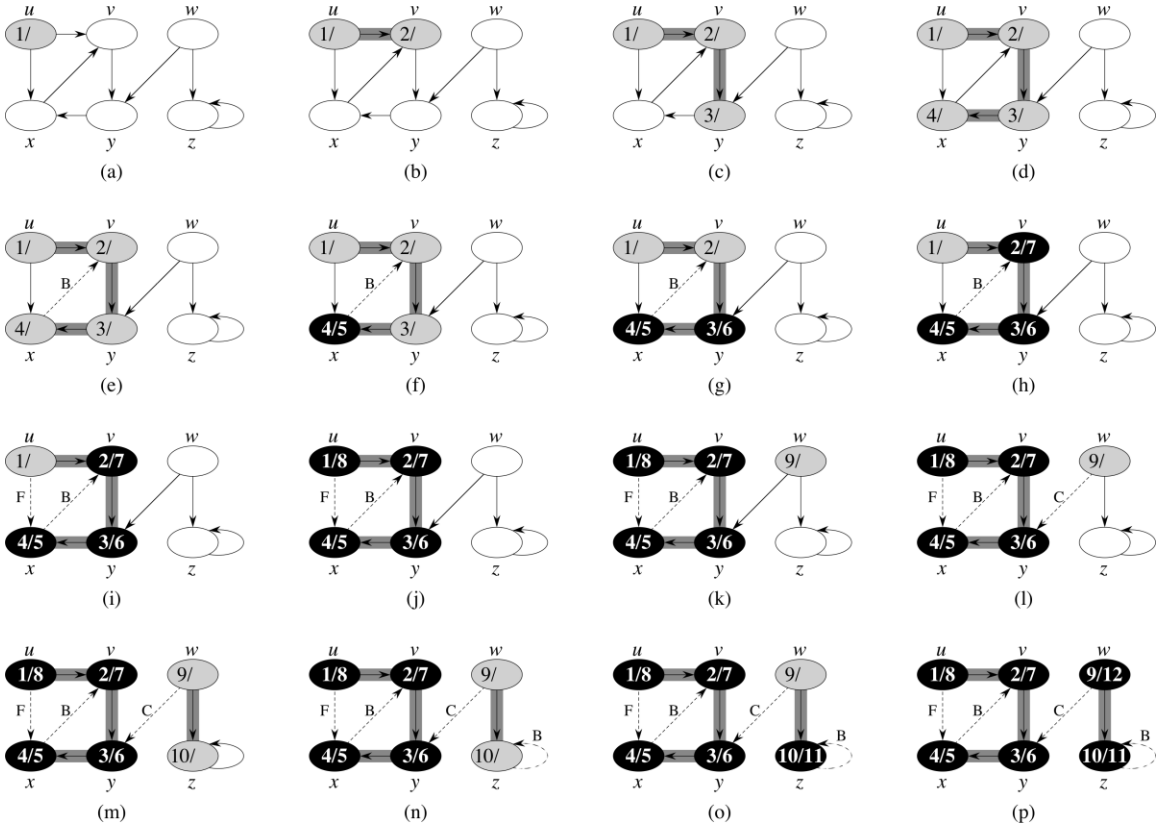
DFS time-stamps each vertex when its color is changed. When v is changed from white to gray the time is recorded in $v.d$ and when v is changed from gray to black the time is recorded in $v.f$. Each time-stamp is an integer in the range of 1 to $2|V|$.

DFS(G)

```
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1   time = time + 1                 // while vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u]           // explore edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK                 // blacken u; it is finished
9   time = time + 1
10  u.f = time
```



What is the running time of DFS? DFS-VISIT is called (from DFS or from itself) once for each vertex in $G.V$ since each vertex is changed from white to gray once. The for-loop in DFS-VISIT is executed a total of $|E|$ times for a directed graph or $2|E|$ times for and undirected graph since each edge is explored one time. Initialization is $O(|V|)$ so the running time for DFS is $O(|V| + |E|)$.

Consider two vertices, u and v, in $G.V$ after a depth-first search. Suppose v is a descendant to u in some DFS-tree. Then $u.d < v.d < v.f < u.f$ because

1) vertex u was discovered before
2) vertex v was fully explored before u was fully explored.

The converse also holds: if $u.d < v.d < v.f < u.f$ then vertex v is in the same DFS-tree and is a descendant of u.

Suppose u and v are in different DFS-trees or suppose they are in the same DFS-tree but neither vertex is the descendant of the other. Then one vertex was discovered and fully explored before the other was discovered;

$$u.f < v.d \text{ or } v.f < u.d.$$

Let G be a directed graph. After a depth-first search of G we can put each edge into one of four classes:

1) A **tree edge** is an edge in a DFS-tree.
2) A **back edge** connects a vertex to an ancestor in a DFS-tree. A self-loop is also a back edge.
3) A **forward edge** is a nontree edge that connects a vertex to a descendant in a DFS-tree.
4) A **cross edge** is any other edge of G. It connects vertices in two different DFS-trees or two vertices in the same DFS-tree neither of which is the ancestor of the other.

Let G be undirected graph. Each edge is a "two-way street" so how do we classify the edges? If an edge connects a vertex with its parent then it's a tree edge. If a nontree edge connects a vertex with an ancestor then it's a back edge. We don't allow forward edges (they become back edges when considered in the opposite direction). Can there be any cross edges? No (every edge of G must connect an ancestor with a descendant).

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

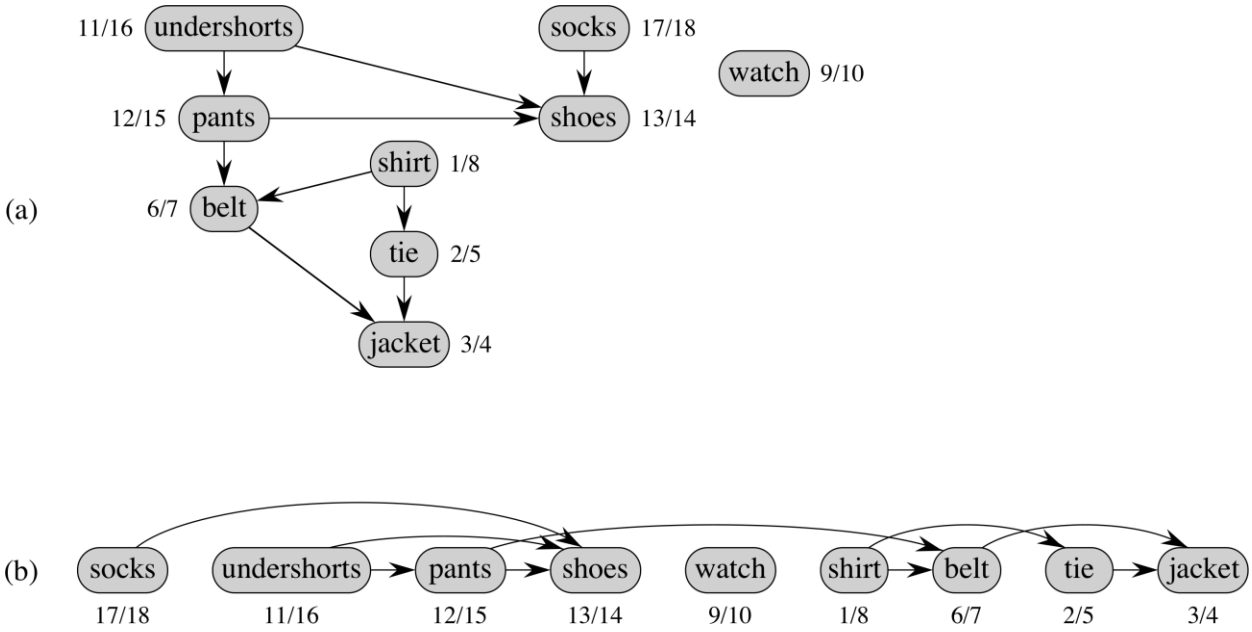
If G is an undirected graph then each DFS-tree will completely span all vertices of a connected component of G. The number of connected components equals the number of DFS-trees.

Topological Sort

A cycle in a directed graph G is a set of edges:
$\{(V_1, V_2), (V_2, V_3), \ldots, (V_{r-1}, V_r)\}$ where $V_1 = V_r$. A directed graph is acyclic if it has no cycles. A directed acyclic graph is sometimes called a DAG.

Lemma: A directed graph G is acyclic iff a depth first search of G yields no back edges.

The vertices of a DAG can be ordered in such a way that every edge goes from an earlier vertex to a later vertex. This is called a **topological sort**. A DAG may have several different topological sorts. As an example consider Prof. Bumstead getting dressed in the morning:



Two topological sorts of this DAG are: {socks, undershorts, watch, shirt, tie, pants, shoes, belt, jacket} and {watch, undershorts, socks, pants, shoes, shirt, belt, tie, jacket}.

One way to get a topological sort of a DAG is to run a depth first search and then order the vertices so their *f* time-stamps are in descending order. This requires $\Theta(|V|\lg|V|)$ time if a comparison sort algorithm is used.

One can modify the depth first search of a DAG to produce a topological sort. When a vertex changes to black push it on a stack or put it on the front of a linked list. The running time is $\Theta(|V| + |E|)$

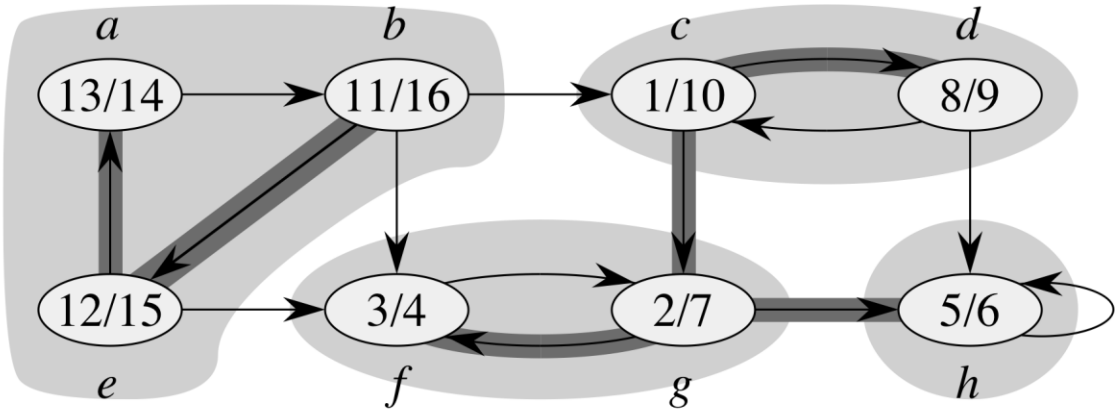
TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times *v.f* 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

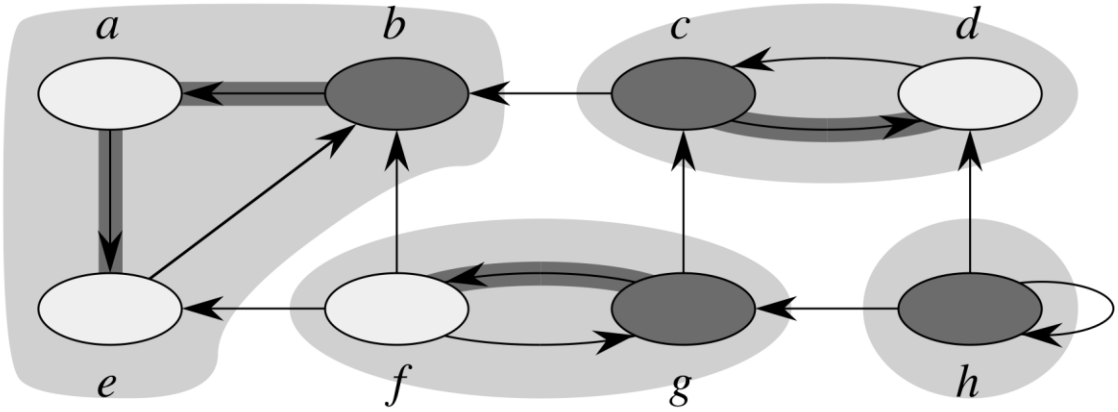
Strongly Connected Components

Any directed graph can be decomposed into its **strongly connected components**. Two vertices are in the same component if and only if they are reachable from each other. For example, consider the following directed graph:

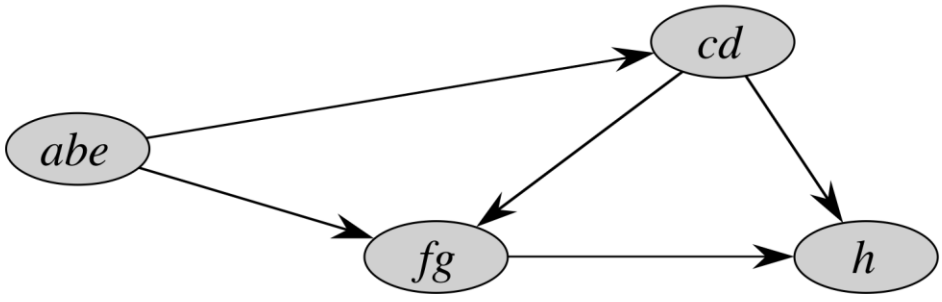
(a)



(b)



(c)



The above directed graph has four strongly connected components: $\{a, b, e\}, \{c, d\}, \{f, g\}, and\ \{h\}$. From any vertex, *v*, one can visit any other vertex in the same component as *v* and then return back to *v*; if one visits a vertex in a different component the return to *v* is impossible.

Let G be a directed graph with **r** strongly connected components, $C_1, C_2, \ldots, C_r$. If G has an edge from some vertex in C_i to some vertex in C_j where $i \neq j$, then one can reach any vertex in C_j form any vertex in C_i but not return.

The component graph, G^{SCC} , is a DAG showing these relations. G^{SCC} has a vertex, c_i, for each strongly connected component, C_i , of G. G^{SCC} has an dege from vertex c_i to vertex c_j if and only if G has an edge from some vertex in C_i to some vertex in C_j . The components graph for the above example are shown in part (c) of the above graph.

If G = (V, E) is a directed graph, its transpose, $G^T = (V, E^T)$ is the same as G with all arrow reversed. Edge set E^T contains edge (u, v) if and only if E contains edge (v, u). Note that G^T has the same strongly connected components as G and its component graph ($G^T)^{SCC}$ will be the transpose of G^{SCC}.

How can we find the strongly connected components of a directed graph, G? A depth first search of G produces a forest of DFS-trees. Let C be any strongly connected component of G, let *v* be the first vertex in C discovered by the search, and let T be the DFS-tree containing *v*.

When DFS is called, all vertices in C are white and reachable from *v* along paths containing white vertices; DFS will visit every vertex in C, add it to T as a descendant of *v*, and color it black.

DFS-tree T contains all vertices of C but it may also contain other components of G (T may have started in another component from which *v* is reachable and also T may cover other components reachable from C). Somehow we have to decompose T to find the strongly connected component of G.

Consider the counterpart, T^{SCC}, of DFS-tree T in the component graph G^{SCC}; T^{SCC} will be a tree connecting those vertices of G^{SCC} corresponding to components in T. If T has more than one component then T^{SCC} is a tree with multiple vertices and one or more edges. Consider any edge (x, y) in T^{SCC} running from vertex x to y and let X and Y be the components of G corresponding to x and y, respectively. Let v_x be the first vertex of X discovered by the search and let v_y be the first

vertex of Y discovered by the search. Components X and Y are in the same DFS-tree so $f[v_x] > f[v_y]$ (else Y would be in a different DFS-tree).

Why are components X and Y in the same DFS-tree? Because Y is reachable from X and the search happened to discover X first. Can we re-order the search to make sure that Y is discovered before X? Unfortunately, we don't know which vertices are v_x and v_y (component X may have another vertex z where $[z] < f[v_y]$).

Suppose we run a depth first search on the transpose, G^T? Every strongly connected component is the same but all the arrows in the component graph, G^{SCC}, are reversed. If component Y is reachable from component X in G it won't be reachable in G^T so when the search discovers X it won't discover Y. But if this search discovers Y before X then it will put them in the same DFS-tree. We have to make sure that when G^T is searched the components are discovered in the same order they were discovered by the search of G.

For the search of G^T we modify the main loop of DFS. We order the vertices in V[G] so vertex *u* is before vertex *v* iff $f[u] > f[v]$ in the search of G. How can we do this? When G is searched, DFS pushes each vertex on a stack when its color is changed to black and when G^T is searched the main loop of DFS pops each vertex off the stack and calls DFS if its color is white.

STRONGLY-CONNECTED-COMPONENTS (G)

1 call DFS(G) to compute finishing times *u.f* for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing *u.f* (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.