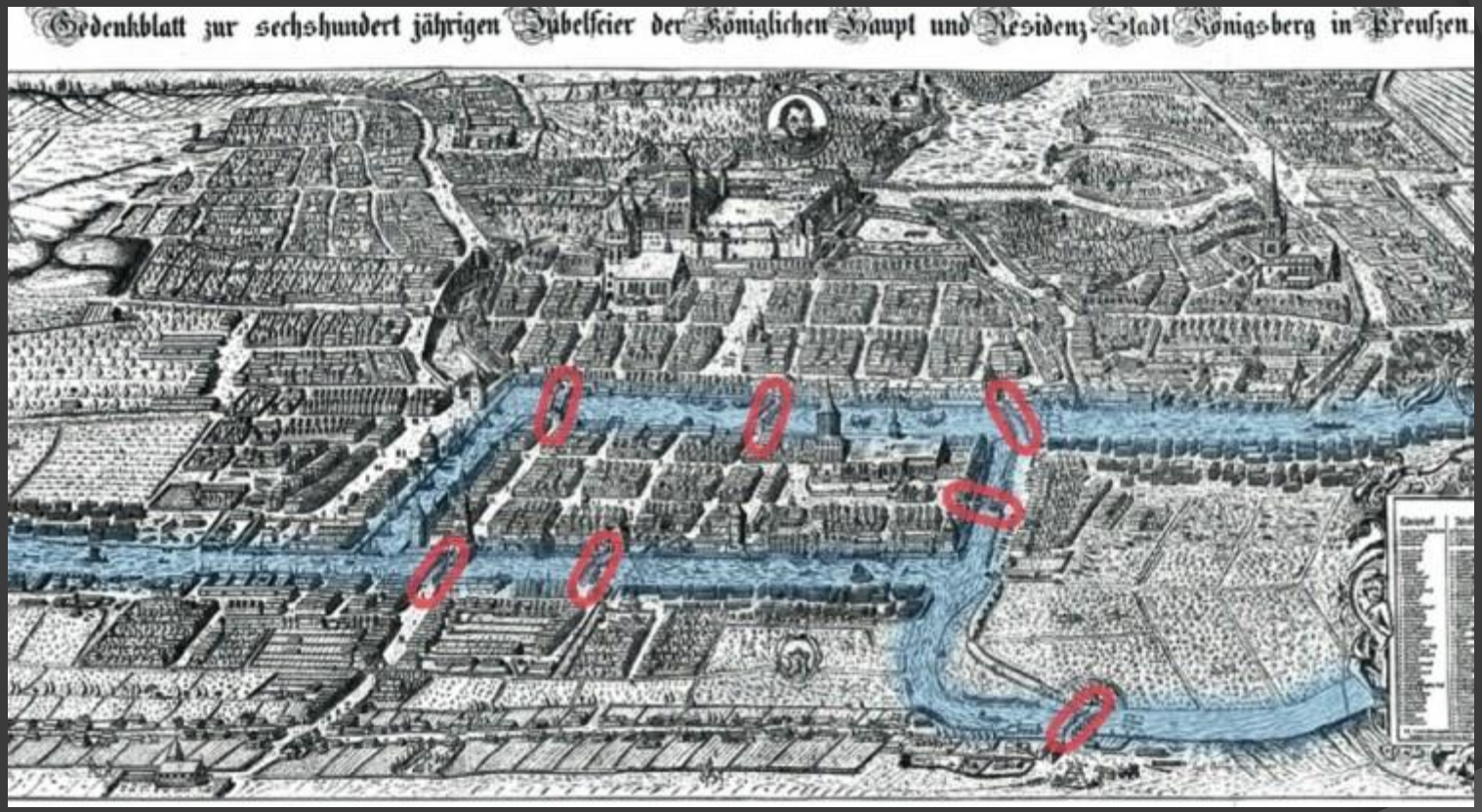
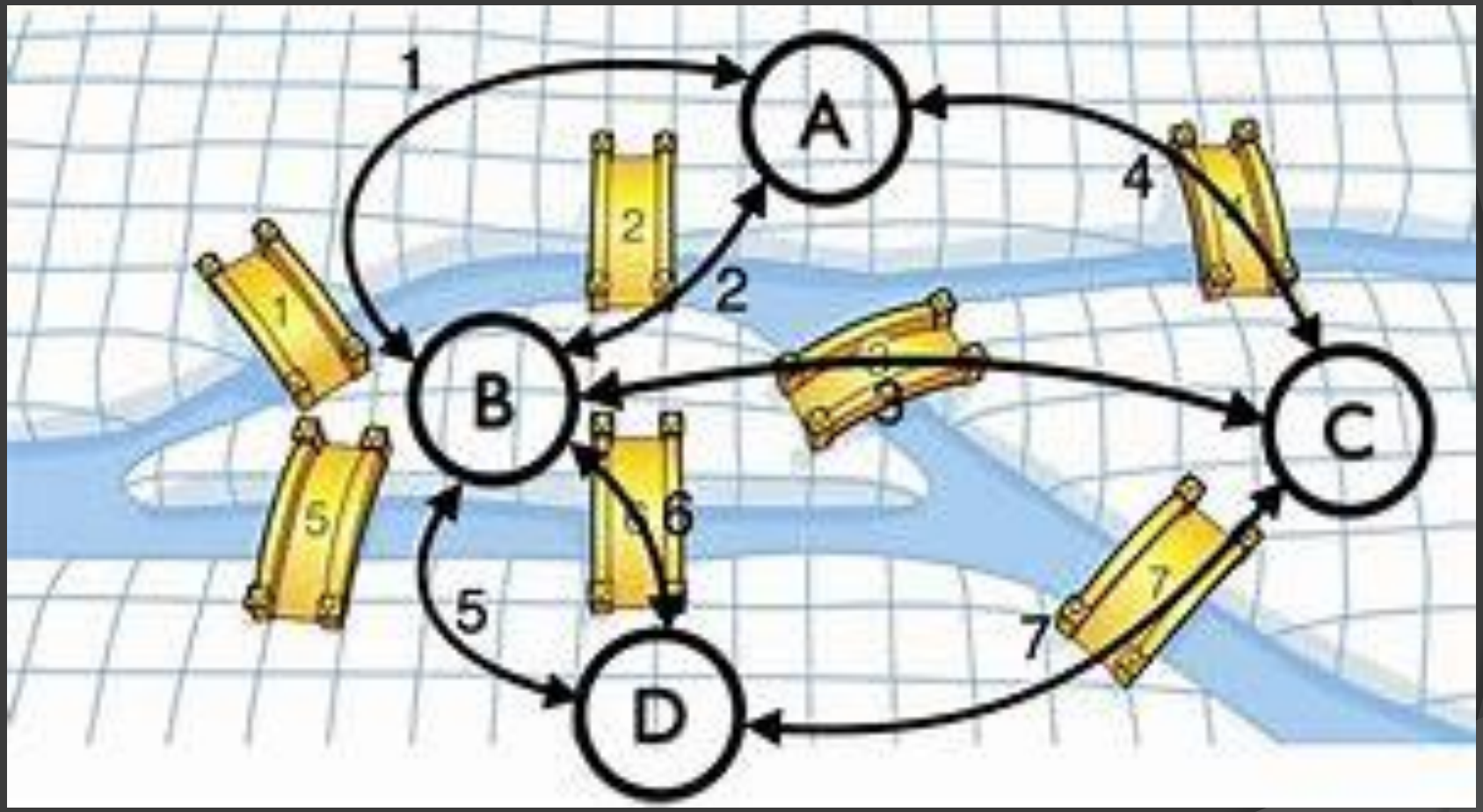
ICSI 403: DESIGN AND ANALYSIS OF ALGORITHMS

Chapter 22: Elementary Graph Algorithms – Part 2

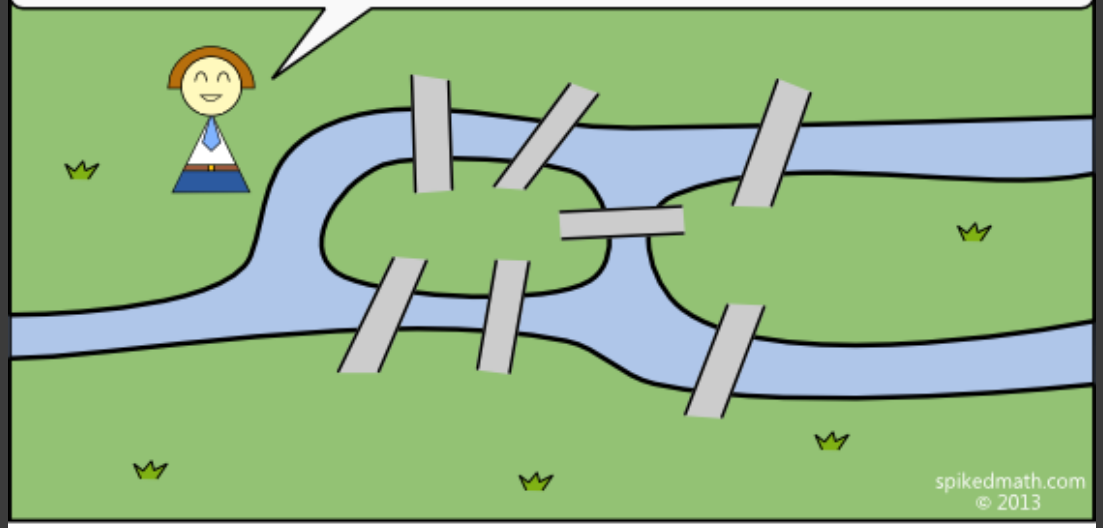
J Marques de Carvalho

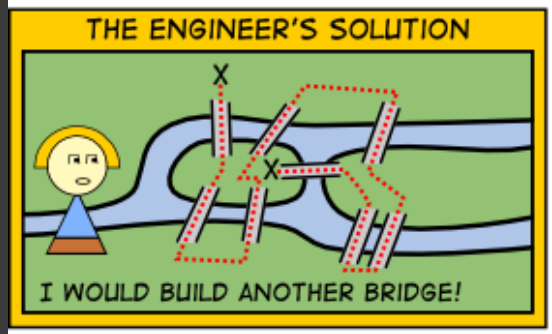

Gedenkblatt zur sechshundert jährigen Jubelfeier der Königlichen Haupt und Residenz-Stadt Königsberg in Preußen.

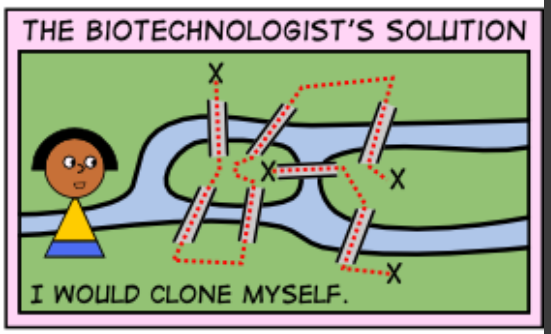

A
B
C
D
1
2
3
4
5
6
7

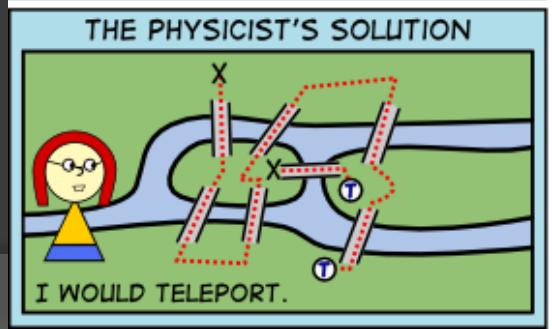
The Seven Bridges of Königsberg

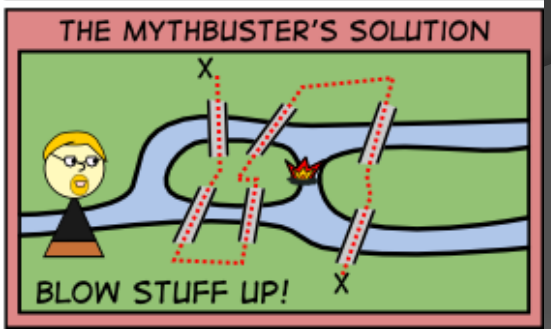
Below is the city of Königsberg with four land masses and seven bridges connecting the various land masses. Can you find a walk through the city of Königsberg that crosses each bridge exactly once? You may start at any land mass you wish but may only travel between land masses by using a bridge.

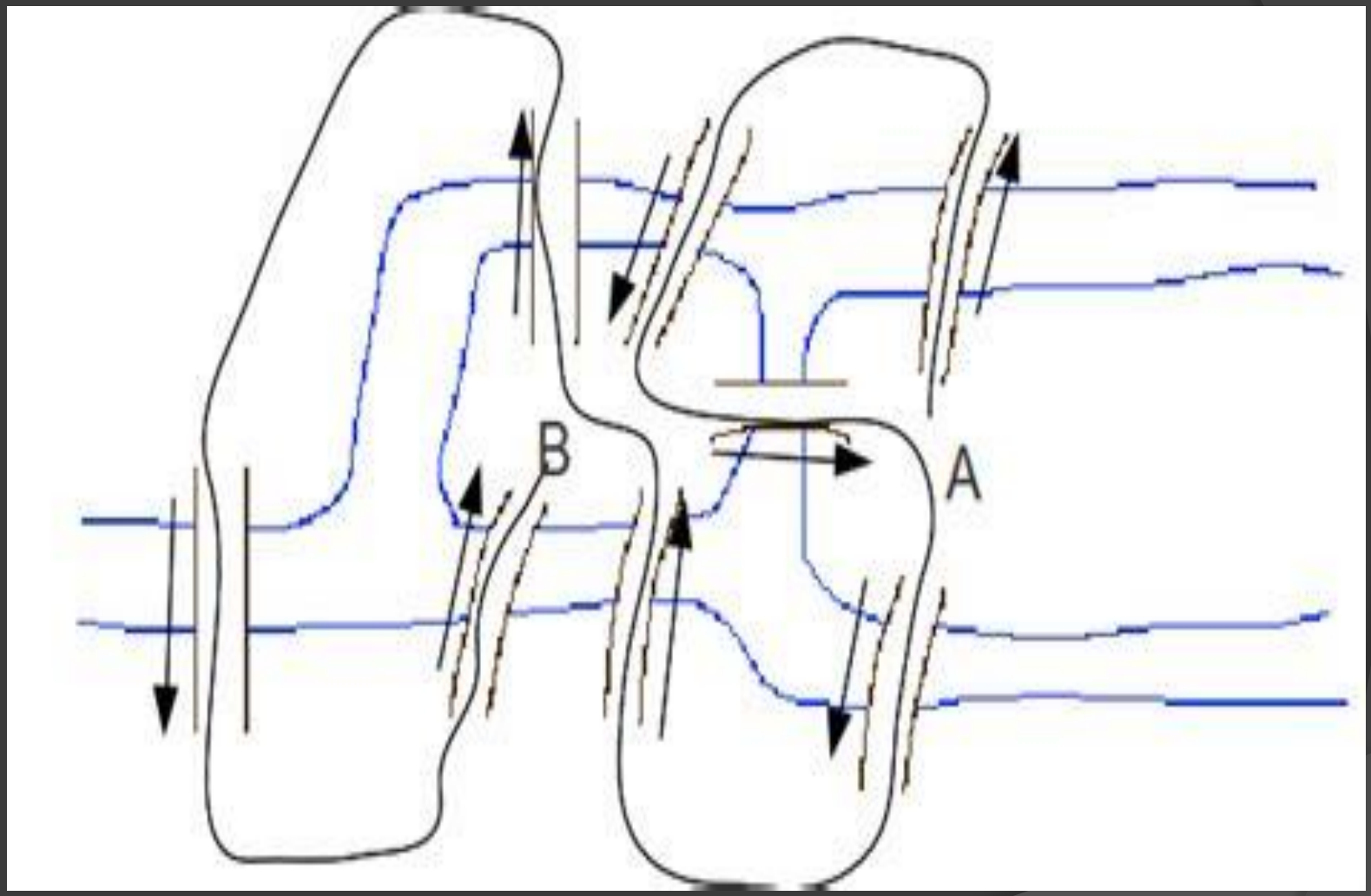








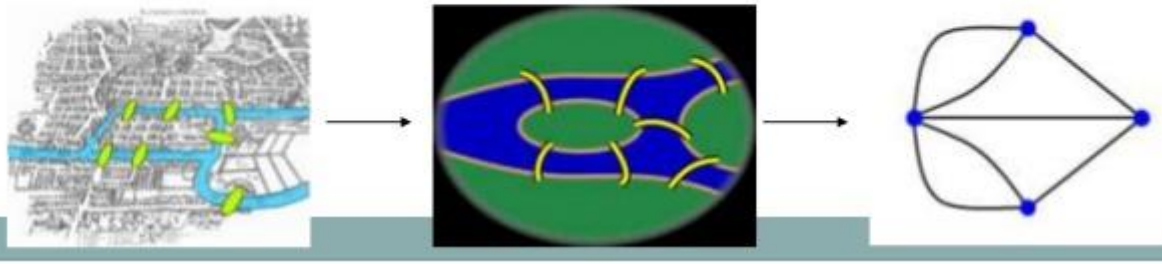



B
A

Graph Theory - History

- Begun in 1735
- Mentioned in Leonhard Euler's paper on "*Seven Bridges of Konigsberg* ".

Problem : Walk all 7 bridges without crossing a bridge twice



Review – Last Time

- The seven bridges of Königsberg problem
- Graphs:
  - Sets of vertices (or nodes), $G.V$, and Edges, $G.E$
  - Directed vs non-directed
  - Degree of a Vertex
    - In- and Out-Degree in directed graphs
  - Graph representations:
    - Adjacency lists (better for sparse graphs)
    - Adjacency matrix (better for dense graphs)
      - Symmetrical about its diagonal in non-directed graphs

Review – Last Time (2)

- Graph-related terms:
  - Fully-connected (complete) graphs
    - Number of edges is always $\frac{1}{2}(|V^2| - |V|)$
  - Paths from one node to another
    - Simple / non-simple
    - Path length
  - Reachable nodes
  - Cyclic paths
  - Acyclic graphs
- Trees as a special case of a graph

Review – Last Time (3)

- Breadth-First Search (BFS)
  - From a given vertex s, systematically expand the "circle" around s, discovering all vertices reachable from s, and the path from s to all reachable vertices
  - Uses a queue of nodes to handle the expansion
- For more basic information on Graphs in general, see Appendix B.3 (p. 1168 – 1172)

Depth-First Search (DFS)

- BFS (above) works by locating all nodes at distance 1 before looking for nodes of distance 2, etc. It's an ever-widening circle
- Depth-First Search (DFS) goes farther and farther from where it starts before it backs up.
- In tree terms, BFS looks at all children, then at all grandchildren, then at all great-grandchildren, …
- DFS looks at a child, then a grandchild, a great-grandchild, … before looking at the next child

Depth-First Search (DFS)

- Each vertex v has two timestamps – one, $v.d$, is when we *discover* the vertex; the other, $v.f$, is when we are *finished* with the vertex.
  - These are used by some other algorithms
- To accomplish this, each vertex is one of three colors:
  - *White* – we haven't discovered that vertex yet
  - *Gray* – vertex has been discovered, but we're not finished exploring beyond it
  - *Black* – we're finished with that vertex (and everything beyond it)

Depth-First Search (DFS)

- The algorithm keeps track of "time" in terms of the number of steps it has taken. The time variable is global.
  - A vertex v is *white* while $time < v.d$
  - A vertex v is *gray* while $v.d < time < v.f$
  - A vertex v is *black* while $time > v.f$
- DFS doesn't start from any particular vertex
  - This is different from BFS, in which we're looking to see what is reachable from a single (given) vertex.

DFS Algorithm

```
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)
```

Initialization:

1 – 3: Color all nodes white, set the predecessor of all nodes to NIL

4: Set the "time" to 0.

Processing:

5 – 7: For each vertex in the graph, if the vertex is white (a new discovery), then "visit" it.

Note: "visiting" a vertex may change the color of other vertices, and will give vertex u its discovery and finalize time stamps, $u.d$ and $u.f$, resp.

It also creates a new DFS tree, rooted at u

DFS-VISIT Algorithm

```
DFS-VISIT(G,u)
 1  u.color = GRAY      //vertex u has just been discovered
 2  time = time + 1     // Let the clock tick
 3  u.d = time          //mark the discovery time for vertex u
 4  for each v ∈ G.Adj[u]   // for each vertex adjacent to u
 5       if v.color == WHITE // If we haven't been here yet
 6            v.π = u                    // we came to v via u
 7            DFS-VISIT(G, v)            // Process v and beyond
 8  u.color = BLACK             // finished with u.  Mark it as
 9  time = time + 1             // black and log its finish time
10  u.f = time
```

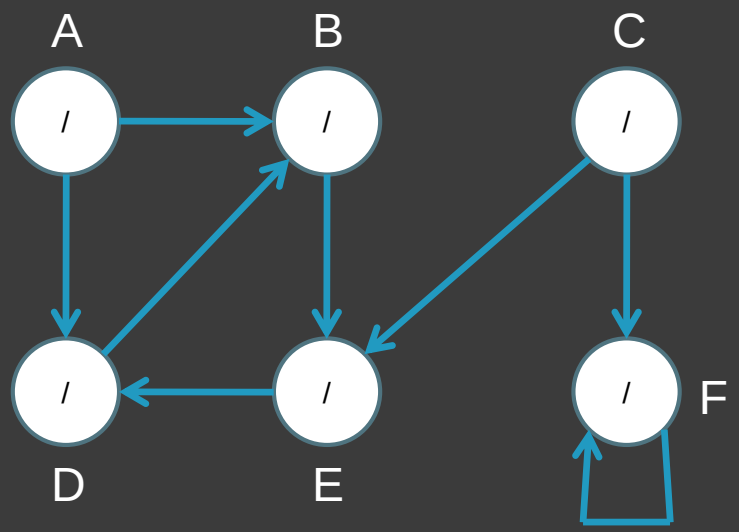
DFS Runtime

- ***Time*** = $\Theta(V + E)$
- Similar to BFS analysis
- Θ, not just O, since it is guaranteed to examine *every* vertex and *every* edge

DFS Trees

- DFS forms a Depth-First *Forest*, composed of (potentially) multiple Depth-First *Trees*
- Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored

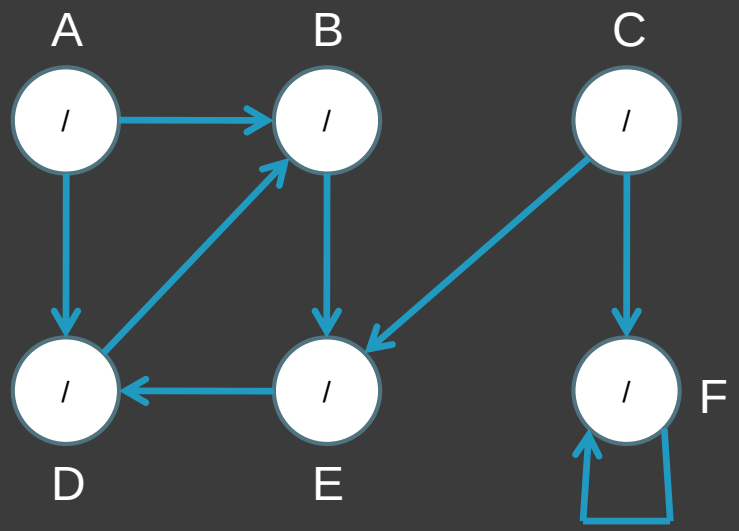
DFS Walkthrough (1)



1. Start with all vertices colored white.
2. No vertex has a discovery time nor a finalization time yet.
3. The time is zero.
4. For each vertex, if it's white, visit it.

So, we visit A.

DFS Walkthrough (2)

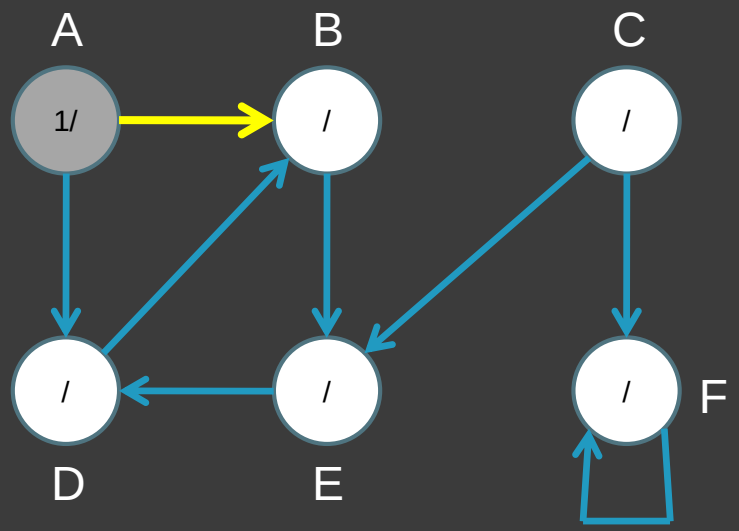


1. Color A gray
2. Increment the time (from 0 to 1), and set A's discovery time to 1
3. For each vertex adjacent to A, if it's white, then visit it.

4. So, we visit B next (we'll come back later to try to visit D)

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

DFS Walkthrough (3)

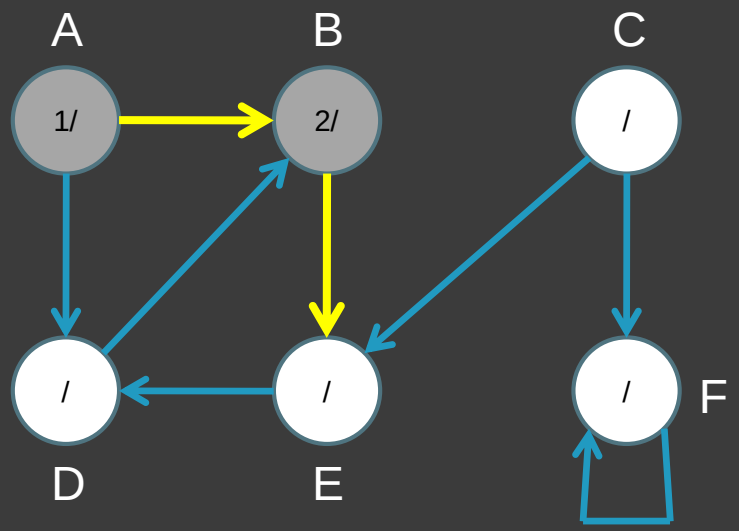


1. Color B gray
2. Increment the time (from 1 to 2), and set B's discovery time to 2
3. For each vertex adjacent to B, if it's white, then visit it.

4. So, we visit E (the only vertex adjacent to B) next

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

DFS Walkthrough (4)

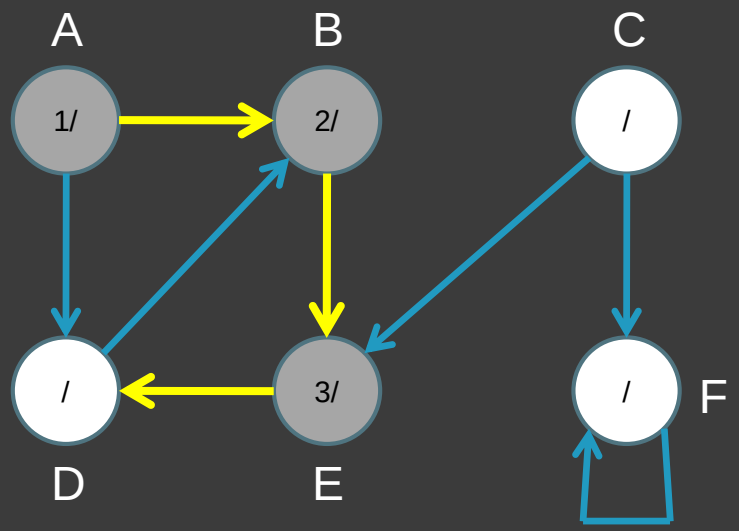


1. Color E gray
2. Increment the time (from 2 to 3), and set E's discovery time to 3
3. For each vertex adjacent to E, if it's white, then visit it.

4. So, we visit D (the only vertex adjacent to E) next

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

DFS Walkthrough (5)

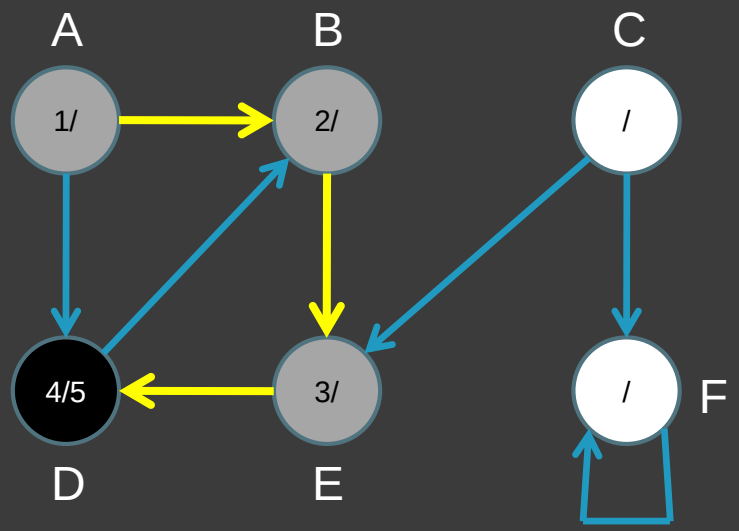



```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

1. Color D gray
2. Increment the time (from 3 to 4), and set D's discovery time to 4
3. For each vertex adjacent to D, if it's white, then visit it.

4. There aren't any, so we go on to line 8 of the algorithm, and color D black, increment the time again (from 4 to 5), and set D's finalization time to 5.

DFS Walkthrough (6)

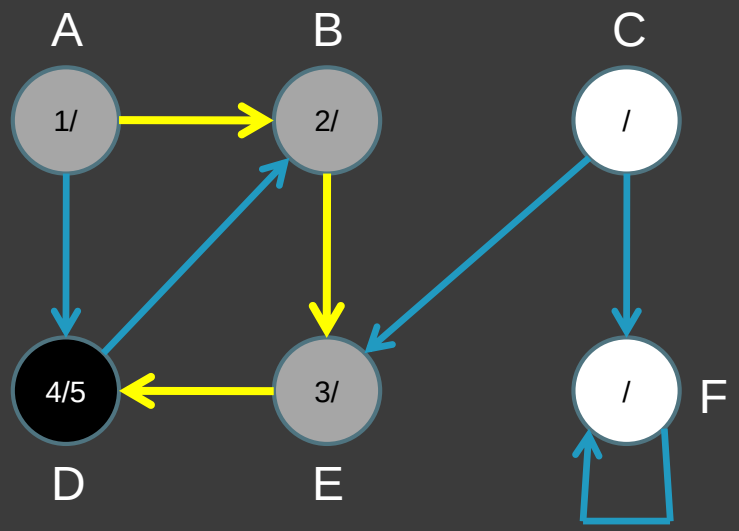



```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

1. Color D gray
2. Increment the time (from 3 to 4), and set D's discovery time to 4
3. For each vertex adjacent to D, if it's white, then visit it.

4. There aren't any, so we go on to line 8 of the algorithm, and color D black, increment the time again (from 4 to 5), and set D's finalization time to 5.
5. We then pop back up the recursion stack, and find ourselves back at E.

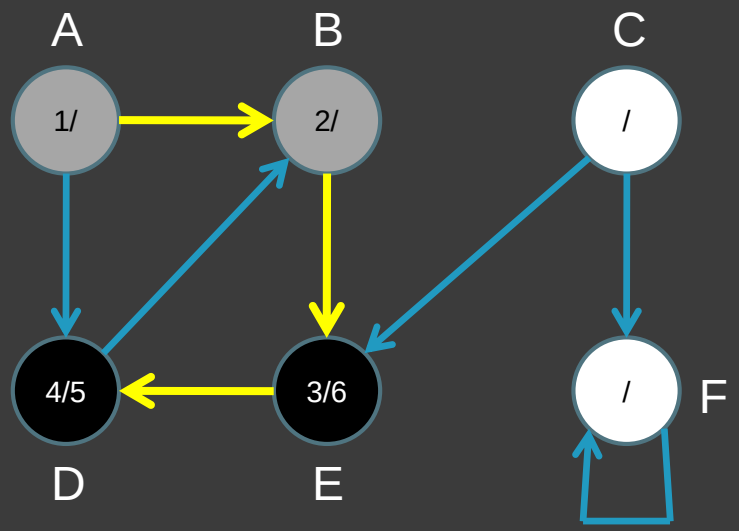
DFS Walkthrough (7)




```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

1. E had no other vertices adjacent to it, so we find ourselves at line 8 of the algorithm.
2. Color E black, increment the time from 5 to 6, and mark E's finalization time at 6.
3. We pop back up the recursion stack and find ourselves back at B.

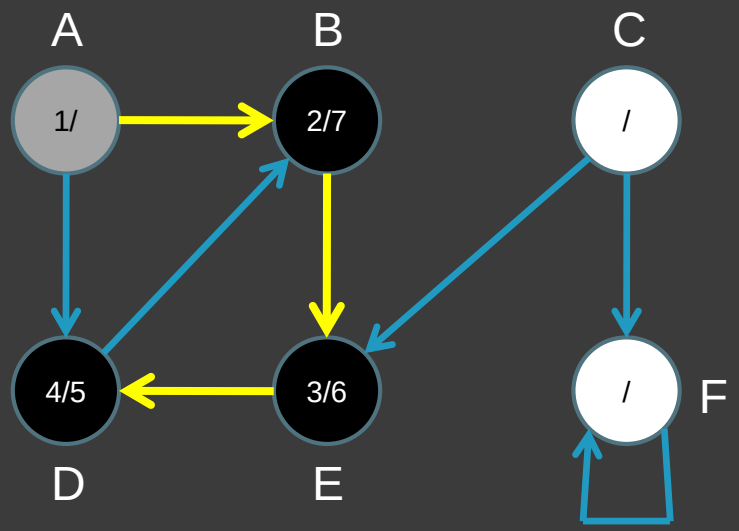
DFS Walkthrough (8)



1. B had no other adjacent vertices, so color B black, increment the time (from 6 to 7), and set B's finalization time.
2. We pop back up the recursion stack and find ourselves back at A

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

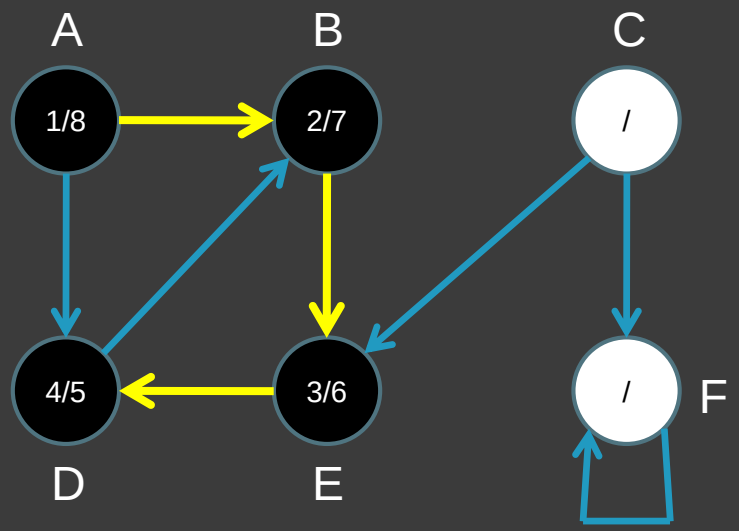
DFS Walkthrough (9)




```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

1. B had no other adjacent vertices, so color B black, increment the time (from 6 to 7), and set B's finalization time.
2. We pop back up the recursion stack and find ourselves back at A
3. When we left A, it was to pursue B, and we put D "on hold".
4. D has since been colored black, so we don't consider D.
5. That was A's last adjacent vertex, so increment the time (from 7 to 8), and set A's finalization time to 8

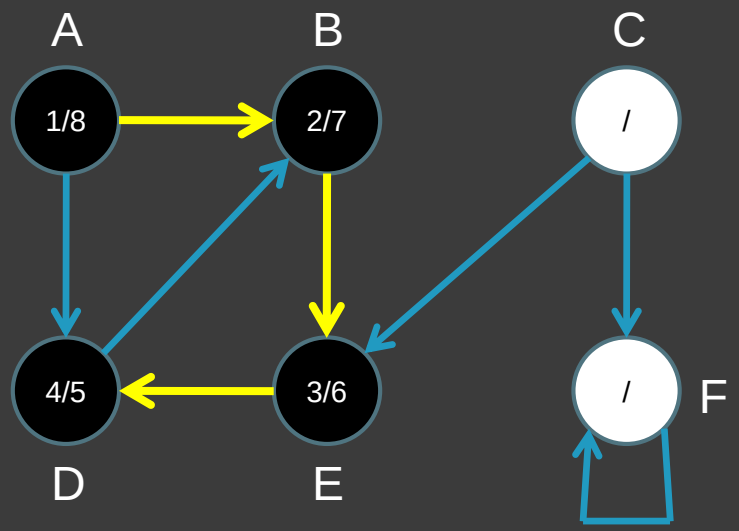
DFS Walkthrough (10)




```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

1. B had no other adjacent vertices, so color B black, increment the time (from 6 to 7), and set B's finalization time.
2. We pop back up the recursion stack and find ourselves back at A
3. When we left A, it was to pursue B, and we put D "on hold".
4. D has since been colored black, so we don't consider D.
5. That was A's last adjacent vertex, so increment the time (from 7 to 8), and set A's finalization time to 8.
6. That's the end of DFS-Visit, so we return to DFS, which has us visit the next white node, C

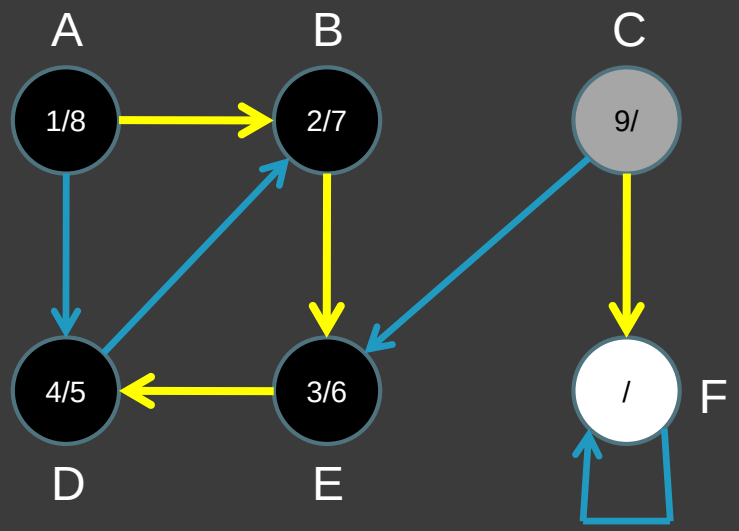
DFS Walkthrough (11)



1. Color C gray.
2. Advance the time from 8 to 9, and set C's discovery time (to 9)
3. C's first adjacent vertex is E, but it's already black, so we try the next one, F, which is still white, so we visit F.

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

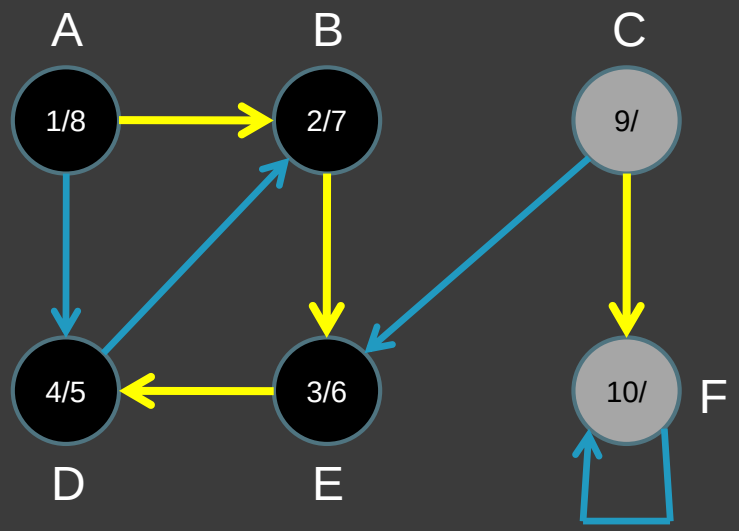
DFS Walkthrough (12)



1. Color F gray.
2. Advance the time from 9 to 10, and set F's discovery time (to 10)
3. F's only adjacent vertex ***is*** F, but we just colored it gray.

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

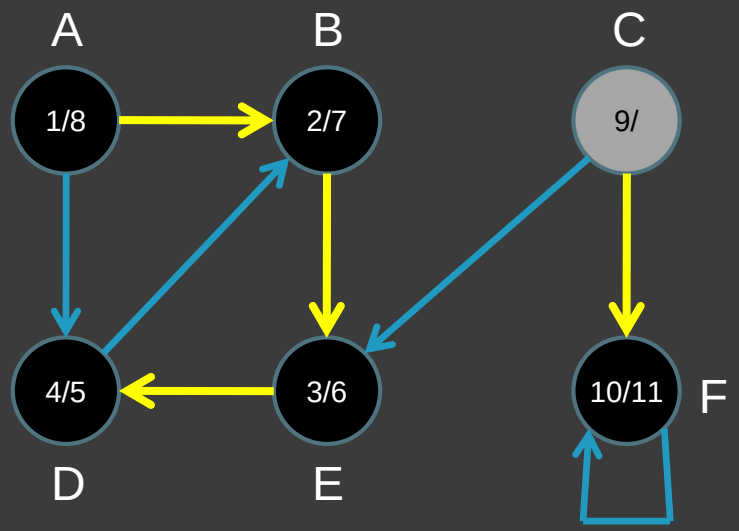
DFS Walkthrough (13)



1. F doesn't have another adjacent vertex, so color it black, advance the time from 10 to 11, and set F's finalization time to 11.
2. We back up the recursion stack and find ourselves at C.

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

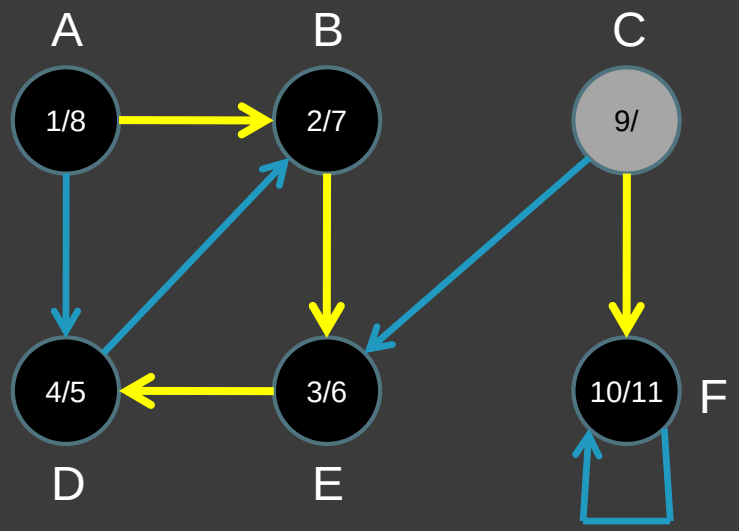
DFS Walkthrough (14)



1. F doesn't have another adjacent vertex, so color it black, advance the time from 10 to 11, and set F's finalization time to 11.
2. We back up the recursion stack and find ourselves at C.

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

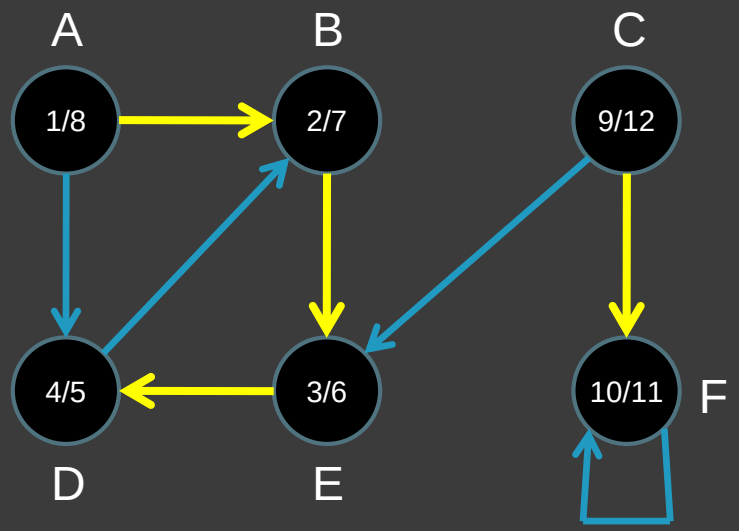
DFS Walkthrough (15)



1. C doesn't have any other adjacent vertices, so we color C black, advance the time from 11 to 12, and stamp C's finalization time.
2. That's the end of DFS-VISIT, so we return to DFS.
3. There are no more white vertices, so we're finished.

```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

DFS Walkthrough (16)




```
DFS-VISIT(G,u)
 1  u.color = GRAY
 2  time = time + 1
 3  u.d = time
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f  = time
```

The vertices have all been visited.
They are all stamped with their discovery / finalization times
The tree paths are indicated with yellow arrows (there are two trees in this graph's forest: A→B→E→D and C→F
What about the other edges?

Classifying DFS Tree Edges (1)

- All edges fall into one of four types:
  1. ***Tree edge:*** Edges in the depth-first forest. Found by exploring (u, v). These are the edges we've already highlighted with yellow arrows in the walk-through.
  2. ***Back edge:*** (u, v), where u is a descendant of v The edge traces back to a previously discovered vertex. These include self loops, where an edge traces back to its start vertex (children pointing back to an ancestor)

Classifying DFS Tree Edges (2)

- All edges fall into one of four types:

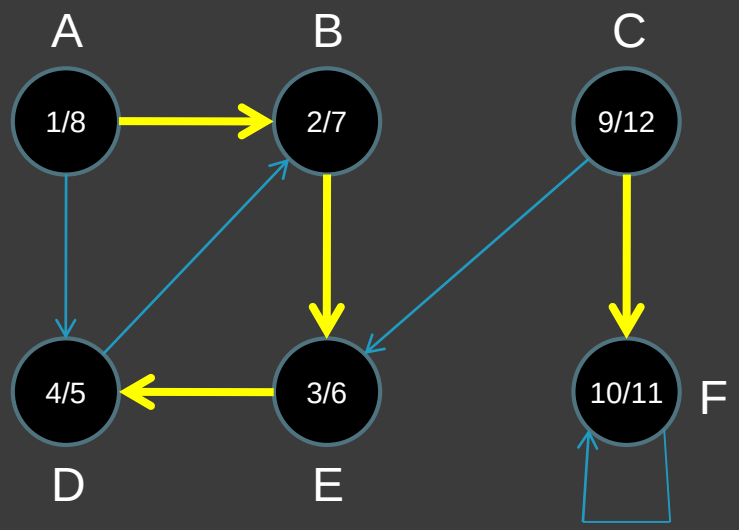
  3. ***Forward edge:*** (u, v), where v is a descendant of u, but not a tree edge. Forward edges connect u to descendant v, but v has already been discovered, i.e. there are tree edges that connect u to v already

  4. ***Cross edge:*** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

Classifying DFS Tree Edges (3)

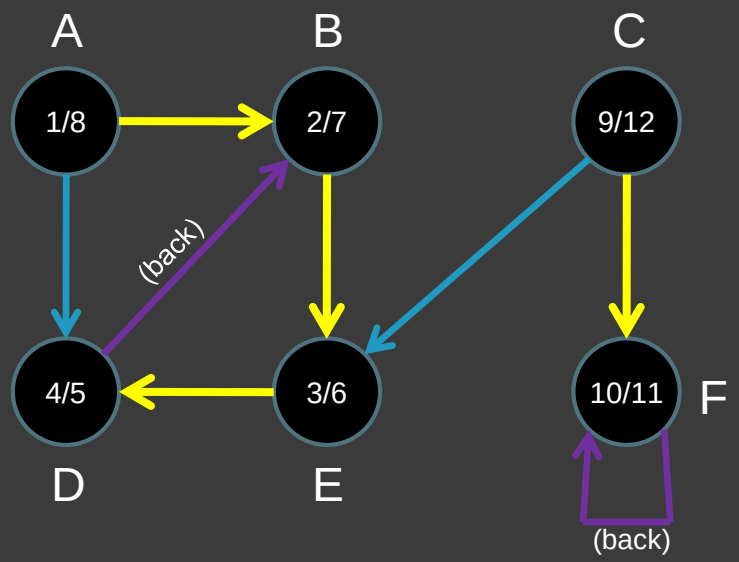
- Edge classification can run into a snag in an undirected graph, given that edge (u, v) is the same as edge (v, u). In this case, we classify the edge by the first one of the above cases that matches

DFS Edge Types



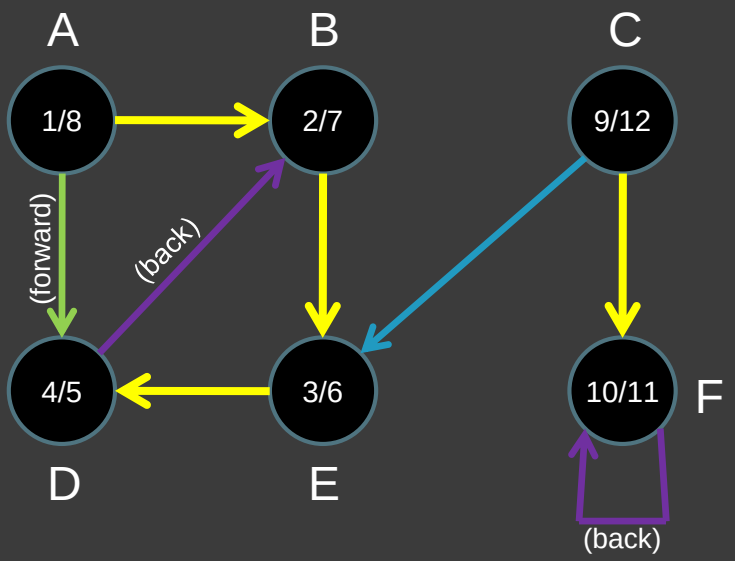
Tree edge: Edges in the depth-first forest. Found by exploring (u, v). These are the edges we've already highlighted with yellow arrows in the walk-through.

DFS Edge Types



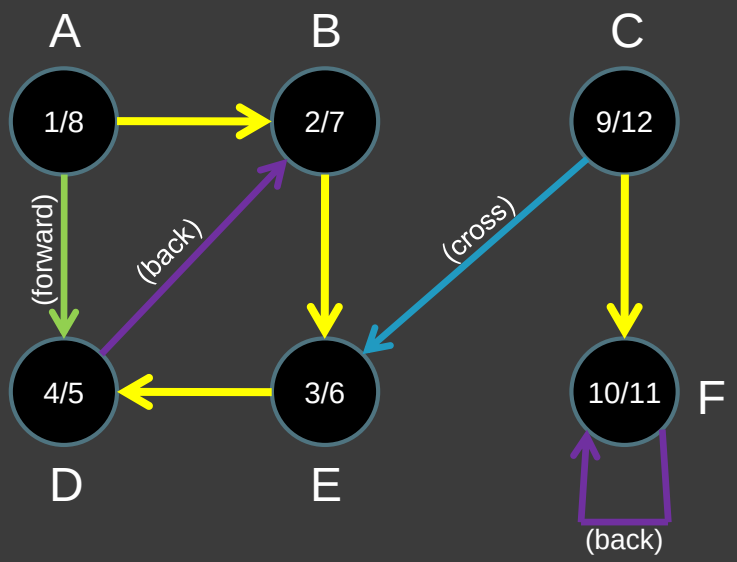
***Back edge*:** (u, v), where u is a descendant of v The edge traces *back* to a previously discovered vertex. These include self loops, where an edge traces back to its start vertex

DFS Edge Types



Forward edge: (u, v), where v is a descendant of u, but not a tree edge. Forward edges connect u to descendant v, but v has already been discovered, i.e. there are tree edges that connect u to v already

DFS Edge Types

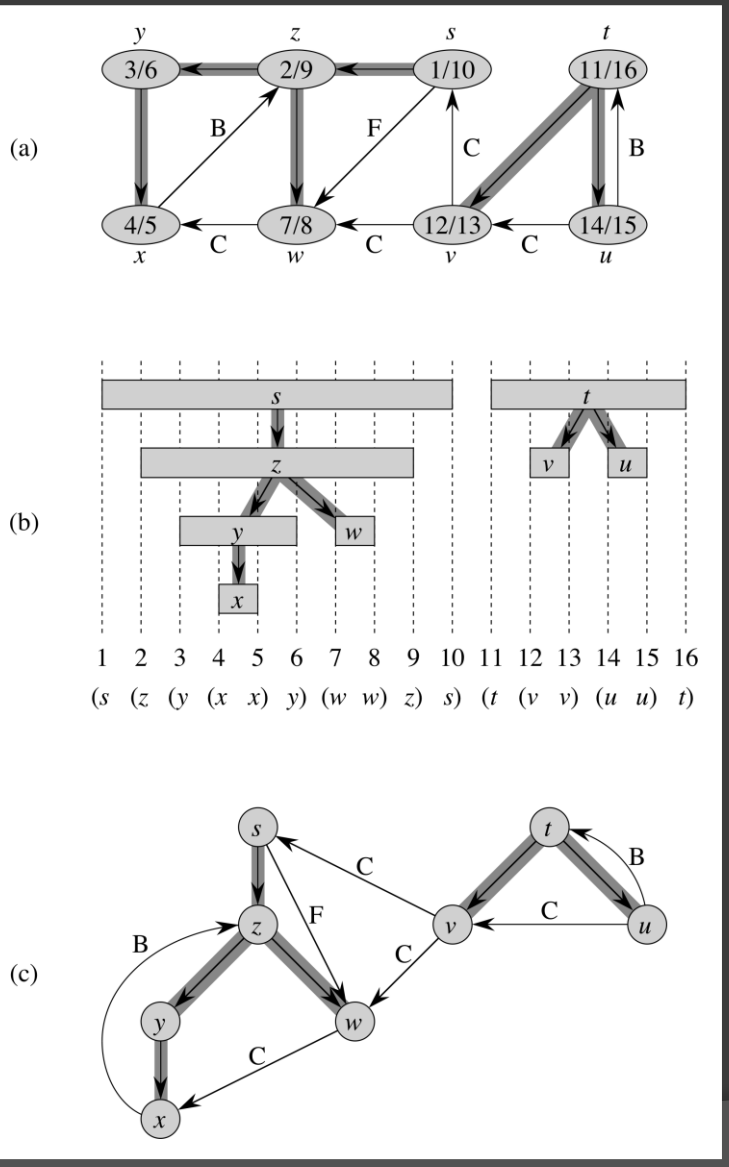


Cross edge: Any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

DFS Edge Types

- Edge classification can be done on-the-fly as we encounter the edges.
- Each edge (u, v) can be classified by the color of v when the edge is first explored:
  - If v is white: (u, v) is a ***tree*** edge
  - If v is gray: (u, v) is a ***back*** edge
  - If v is black: (u, v) is either a forward or a back edge
    - If $u.d < v.d$ then (u, v) is a ***forward*** edge
    - If $u.d > v.d$ then (u, v) is a ***cross*** edge

The DFS Tree



(a) The result of a DFS of a directed graph. Vertices are time-stamped and edge types are indicated

(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 edges. Thee 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 downward with a depth-first tree and all back edges going up from a descendant to an ancestor

DFS Summary

- Depth-First Search (DFS)
  - Explores all vertices and edges
  - Goes “deeper” before it goes “wider”
  - Doesn’t start from a particular vertex
  - Builds a “forest” of one or more trees, whose structures are based on the graph’s structure
  - Colors vertices as it processes them (W→G→B)
  - Marks vertices with discovery ($v.d$) and finalization ($v.f$) “time” stamps
  - Classifies edges as it encounters them:
    - Tree, Back, Forward, Cross

Topological Sort (22.4)

- Given a Directed, Acyclic Graph (a “dag”)
- Topological Sort of a dag is a linear ordering (a listing) of its vertices V such that for every edge (u, v), u appears before v in the ordered list.
  - The graph ***must*** be acyclic to do a linear ordering
  - This ordering is different from solving the sorting problem we have examined earlier

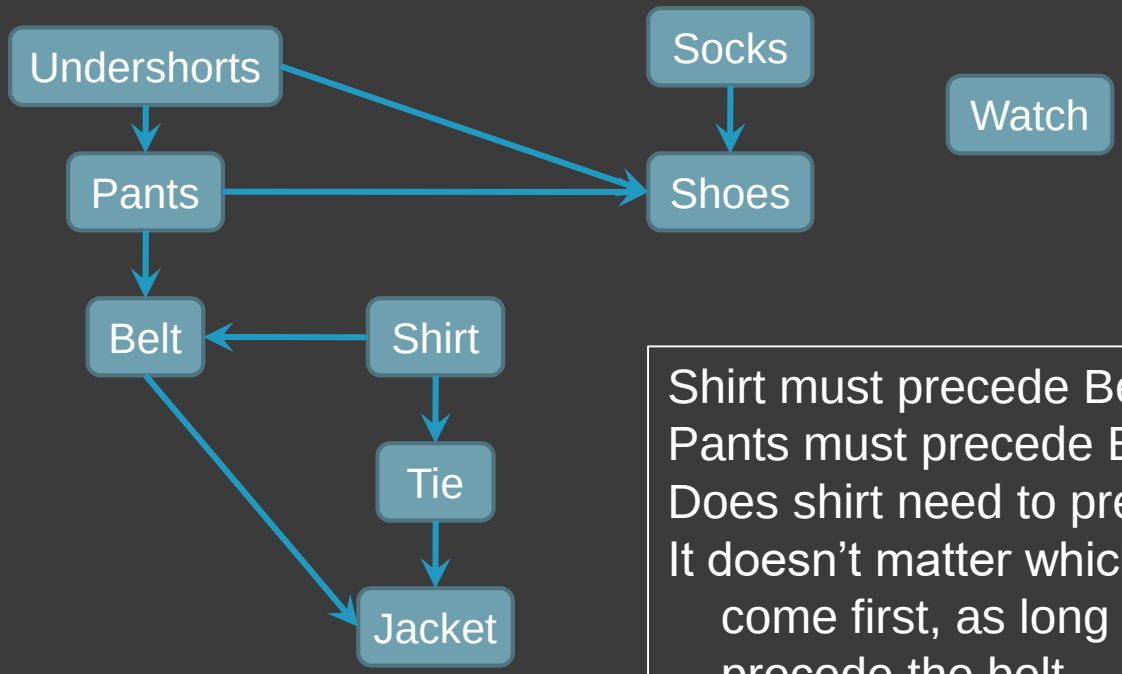
Topological Sort (22.4)

- DAGs are often used to model sequences of activities where there are *partial orderings* of the events
  - *Partial ordering* – there's not one single sequence in which all of the events have to occur, but ***some*** events have to precede ***some*** others
  - This is one of the applications of A.I.
    - Planning the assembly sequence of components and sub-assemblies

Topological Sort (22.4)

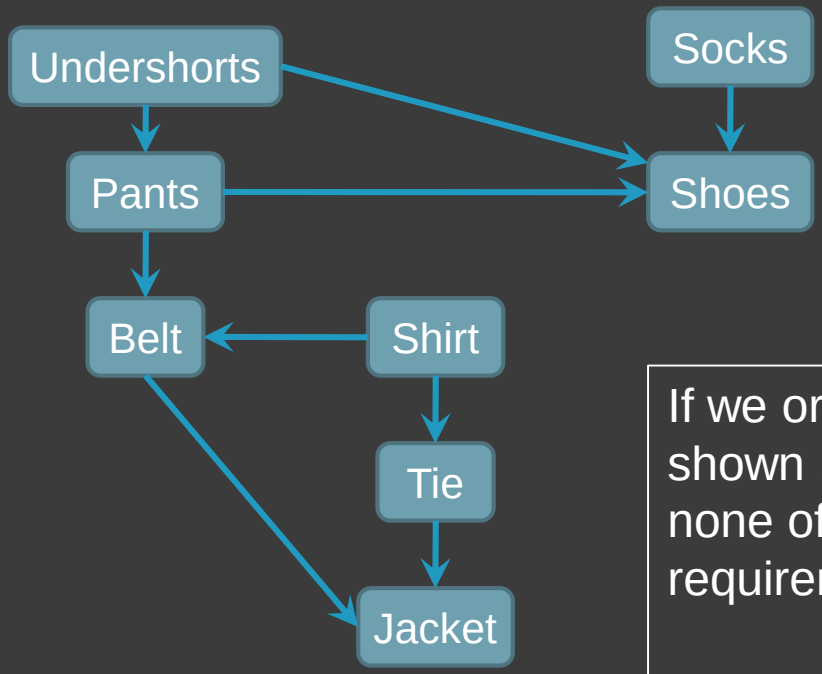
- If event (or activity) a must precede some other event b, and event b must precede event c, we write that as $a < b$ and $b < c$, in which case $a < b < c$, and there is only one proper complete ordering.
- Sometimes, there's no *particular* ordering between two events – it doesn't matter whether a or b precedes the other.
- Topological sort will take a partial order and give us a full ordering (perhaps one of many)

Topological Sort (22.4)



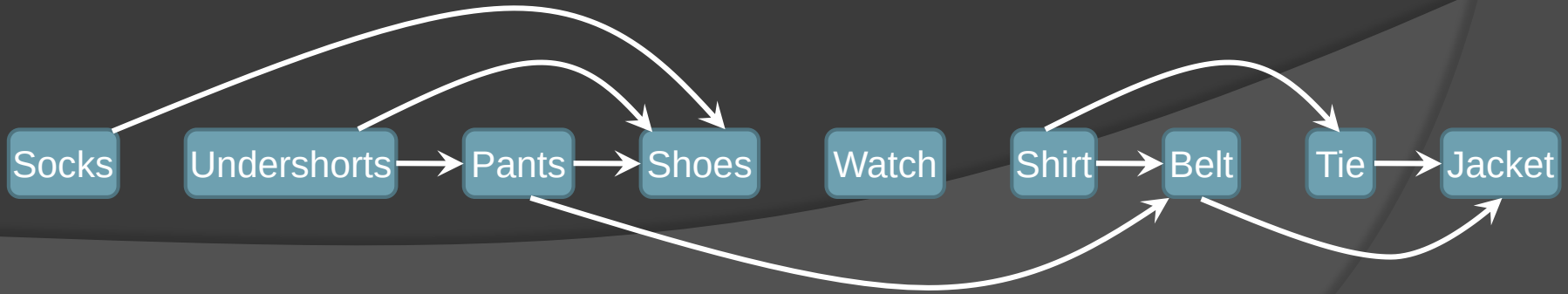
Shirt must precede Belt
Pants must precede Belt
Does shirt need to precede pants?
It doesn't matter which of those two come first, as long as they both precede the belt.

Topological Sort (22.4)



If we order the operations as shown below (left-to-right), none of the precedence requirements are violated

Note that all of the edges point *somewhere* to the right



TOPOLOGICAL-SORT Algorithm

TOPOLOGICAL-SORT(G)

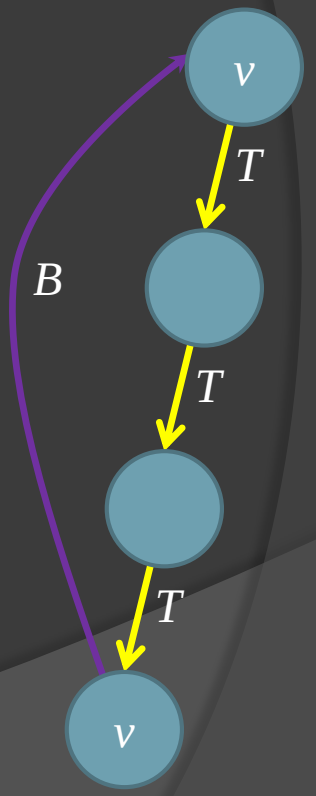
1 Call DFS(G) to compute finish times $v.f$ for all vertices $v \in G.V$

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

3 **return** the linked list of vertices

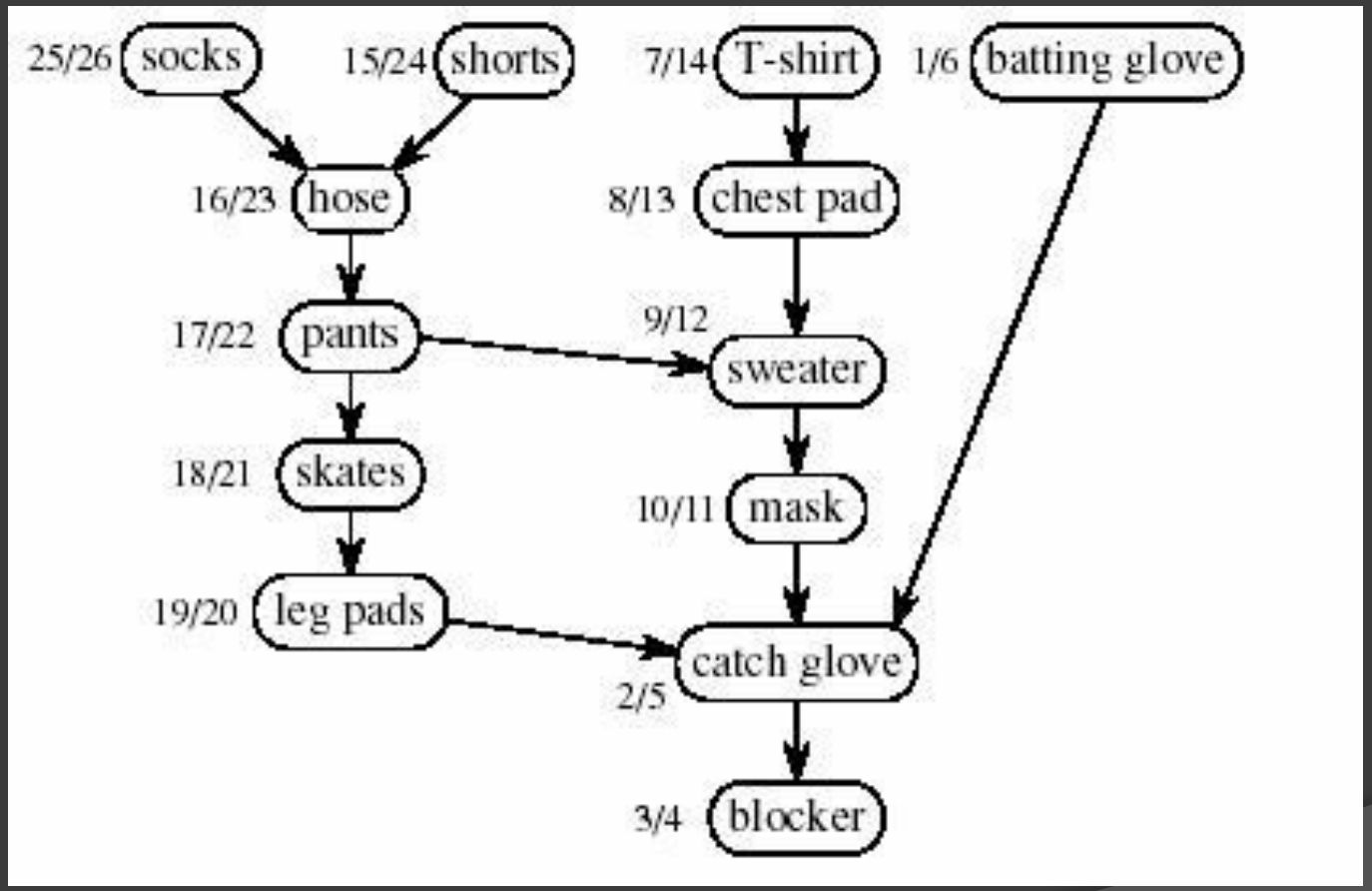
TOPOLOGICAL-SORT Algorithm

- ***Lemma:*** A directed graph G is acyclic if and only if a DFS of G yields no back edges.
- In example to the right, if the back edge B exists, then the graph is cyclic (i.e., not acyclic)
- This property is needed to prove the correctness of the TOPOLOGICAL-SORT algorithm



Another Example

EXAMPLE: DAG of dependencies for putting on goalie equipment



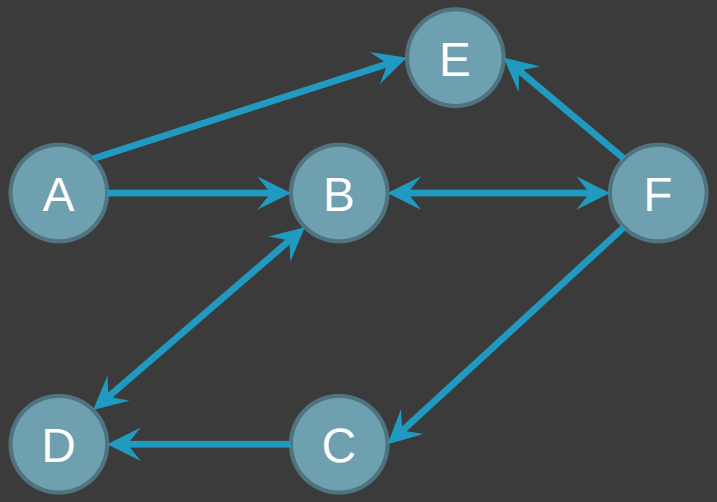
Order:
26 socks
24 shorts
23 hose
22 pants
21 skates
20 leg pads
14 t-shirt
13 chest pad
12 sweater
11 mask
6 batting glove
5 catch glove
4 blocker

Strongly Connected Components (22.5)

- A graph is ***strongly connected*** if every two vertices are reachable from each other
- *Strongly connected components* are portions of a graph that are strongly connected
- More formally:
  - For a given directed graph $G = (V, E)$, a ***strongly connected component*** (**SCC**) of G is a maximal set of vertices C, *a subset of* V, such that $\forall\ u,v \in C$, there is a path from u to v and from v to u

Strongly Connected Components (22.5)

- Is this graph strongly connected? Why?



- Does it have any strongly-connected components? Why?

Software Module

- A well-defined component of a software system
- A part of a system that provides a set of services to other modules
  - Services are computational elements that other modules may use

Questions

- How to define the structure of a modular system?
- What are the desirable properties of that structure?

Modules and relations

- Let S be a set of modules

$$S = \{M_1, M_2, \ldots, M_n\}$$

- A binary relation r on S is a subset of S x S
- If M_i and M_j are in S, $\langle M_i, M_j \rangle \in r$ can be written as $M_i \, r \, M_j$

Relations

- Transitive closure r^+ of r

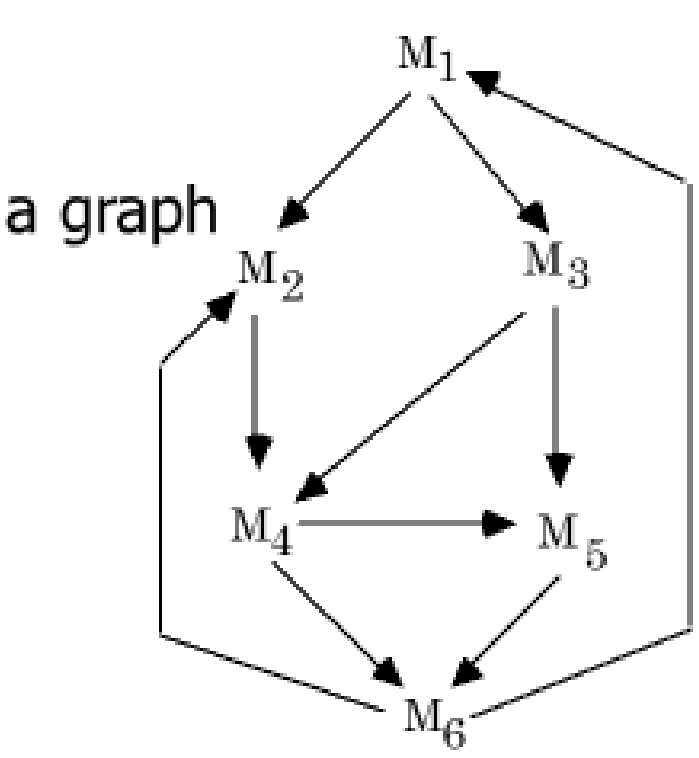
  $M_i\ r^+\ M_j$ iff

  $M_i\ r\ M_j$ or $\exists\ M_k$ in S s.t. $M_i\ r\ M_k$ and $M_k\ r^+\ M_j$

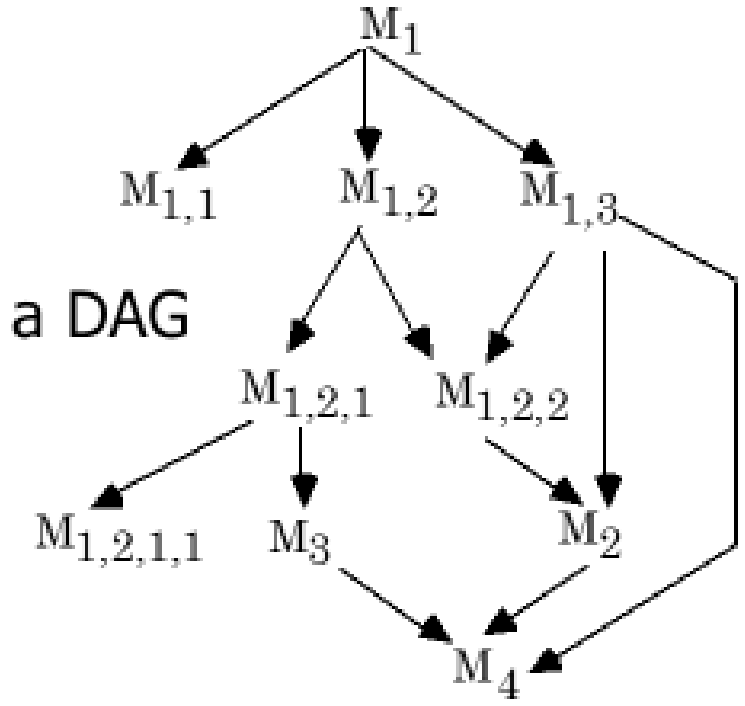
- (We assume our relations to be irreflexive)
- r is a hierarchy iff there are no two elements M_i, M_j s.t. $M_i\ r^+\ M_j \wedge M_j\ r^+\ M_i$

Relations

- Relations can be represented as graphs
- A hierarchy is a DAG (directed acyclic graph)



a)



b)

The USES relation

- A uses B
  - A requires the correct operation of B
  - A can access the services exported by B through its interface
  - it is “statically” defined
  - A depends on B to provide its services
    - example: A calls a routine exported by B
- A is a client of B; B is a server

Desirable property

- USES should be a hierarchy
- Hierarchy makes software easier to understand
  - we can proceed from leaf nodes (who do not use others) upwards
- They make software easier to build
- They make software easier to test

Hierarchy

- Organizes the modular structure through *levels of abstraction*
- Each level defines an *abstract (virtual) machine* for the next level
  - *level* can be defined precisely
    - M_i has level 0 if no M_j exists s.t. M_i r M_j
    - For each module M_i, let k be the maximum level of all nodes M_j s.t. M_i r M_j. Then M_i has level k+1

Hierarchy: USES example

- Let M_R be a module that provides input-output of record values.
- Let M_R use another module M_B that provides I/O of a single byte at a time.
- When used to output record values, the job of M_R consists of transforming the record into a sequence of bytes and isolating a single byte at a time to be output by means of M_B.
- M_B provides a service that is used by M_R.

Module Level Concepts

- Ideally we decompose up to have a minimum of interaction between modules and, conversely, a high degree of interaction within a module.
- Coupling: measure of independence
- Cohesion: logical relationship
- Cohesion and coupling help determine “quality” of the architecture.

Module Level Concepts

- The USES relation provides a way to reason about the coupling in a precise manner.
- With reference to a USES graph, we can distinguish the number of incoming edges (fan-in) and the number of outgoing edges (fan-out).

Module Level Concepts(cont)

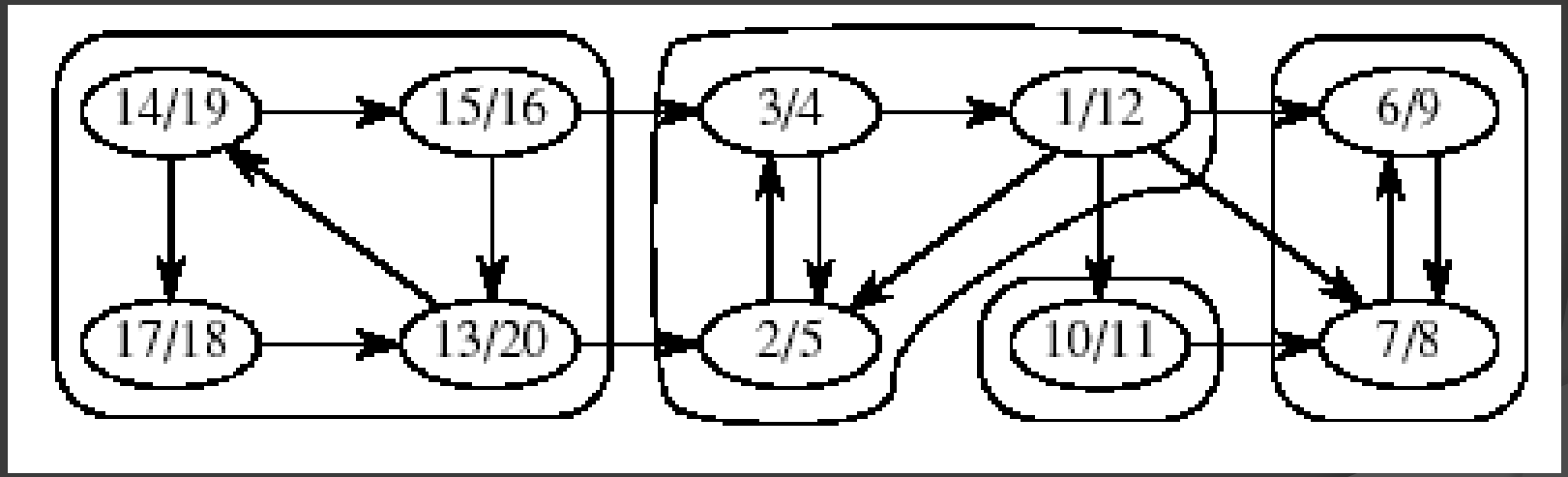
- A good design structure should keep the fan-out low and the fan-in high.

Module Level Concepts

- A high fan-in is an indication of good design because a module with high fan-in represents a meaningful i.e. general abstraction that is used heavily by other modules.
- A high fan-out is an indication that a module is doing too much which in turn may imply that a module has poor cohesion.
- The evaluation of the quality of design should not merely depend on the USES relation.

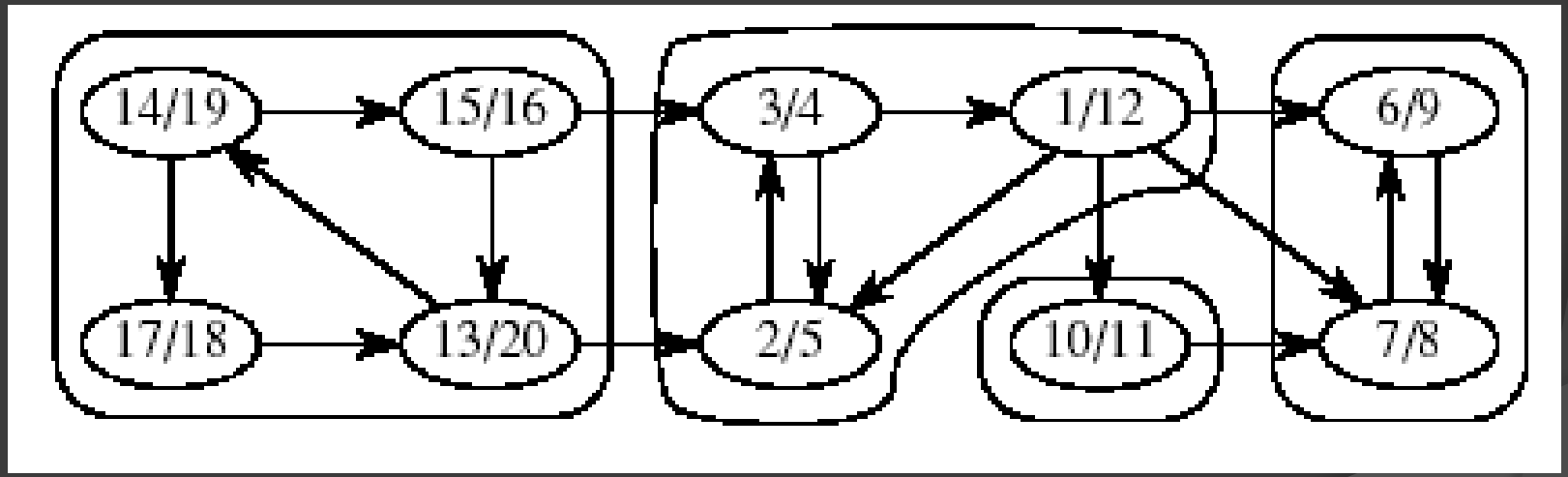
Strongly Connected Components (22.5)

- Below is an example of strongly connected components, including the $v.d$ & $v.f$ times after the depth-first search is run



Strongly Connected Components (22.5)

- Below is an example of strongly connected components, including the $v.d$ & $v.f$ times after the depth-first search is run

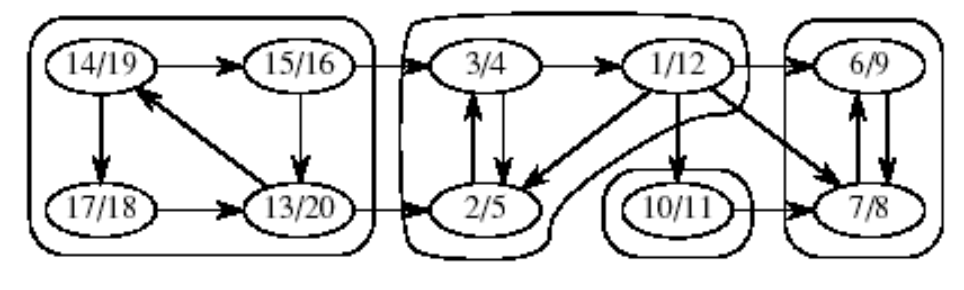


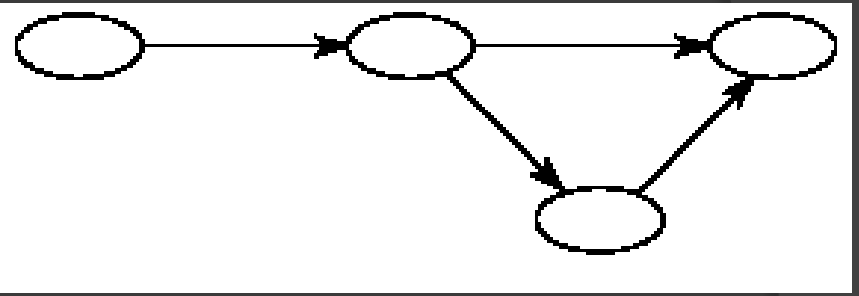
Strongly Connected Components (22.5)

- The algorithm uses G^T, made from $G(V, E^T)$
  - $E^T = \{(u, v) : (v, u) \in E\}$
  - G^T is G with all the edges reverved
- We can create G^T in $\Theta(V + E)$ time if using adjacency lists
- G and G^T have the same strongly connected components
- u and v are reachable from each other in G iff they are reachable from each other in G^T

Strongly Connected Components (22.5)

- The SCC's form their own graph:





- $G^{SCC} = (V^{SCC}, E^{SCC})$
- V^{SCC} has one vertex for each SCC in G
- E^{SCC} has an edge if there's an edge between the corresponding SCCs in G

Finding SCC's

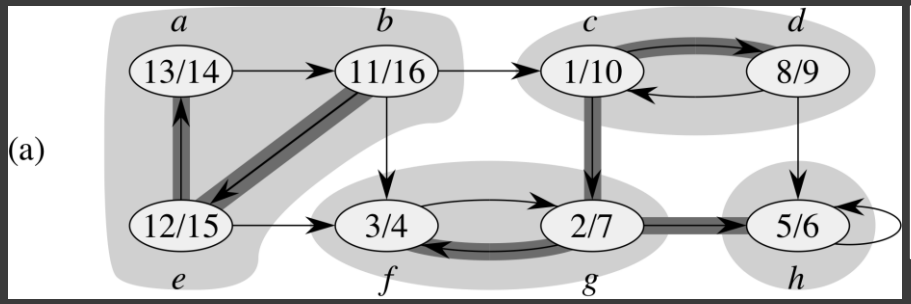
- Algorithm uses two searches, one for the graph G, and one for its transpose
- The algorithm runs in time $\Theta\,(V+E)$

STRONGLY-CONNECTED-COMPONENTS Algorithm

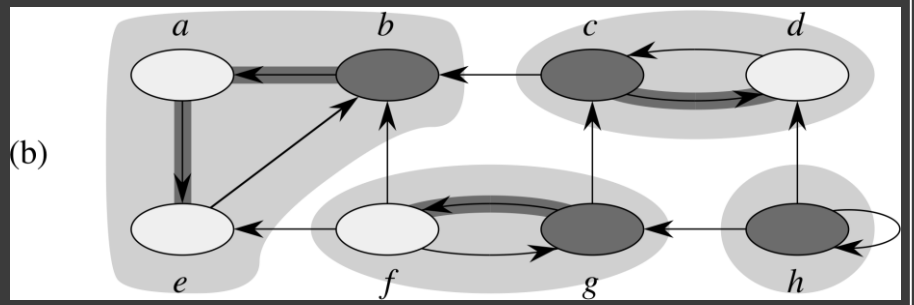
STRONGLY-CONNECTED-COMPONENTS(G)

1 Call DFS(G) to compute the finish 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 SCC

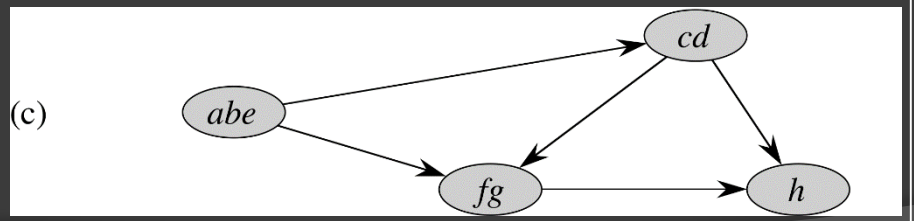
STRONGLY-CONNECTED-COMPONENTS Algorithm



(a) A Directed Graph G. The SCCs of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded.



(b) The graph G^T, the transpose of G. The DFS Forest computed in line 3 of Strongly-Connected-Components is shown, with tree edges shaded. Each SCC corresponds to one depth-first tree. Vertices b, c, g, & h, shown heavily shaded, are the roots of the depth-first trees produced by the DFS of G^T



(c) The acyclic component graph G^{SCC} obtained by contracting all edges within each SCC of G so that only a single vertex remains in each component.

Strongly Connected Component Notes

- The idea behind the concept of SCC graphs is that by considering vertices in the second DFS in decreasing order of finishing times computed from the first DFS, we are visiting vertices of the component graph in topological sort order.

End of Chapter 22

? Questions ?