Implementing graph attributes

No one best way to implement. Depends on the programming language, the algorithm, and how the rest of the program interacts with the graph.

If representing the graph with adjacency lists, can represent vertex attributes in additional arrays that parallel the *Adj* array, e.g., $d[1 \,..\, |V|]$, so that if vertices adjacent to u are in $Adj[u]$, store $u.d$ in array entry $d[u]$.

But can represent attributes in other ways. Example: represent vertex attributes as instance variables within a subclass of a `Vertex` class.

Breadth-first search

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output: $v.d$ = distance (smallest # of edges) from s to v, for all $v \in V$.
In book, also $v.\pi$ such that (u, v) is last edge on shortest path $s \leadsto v$.

- u is v's ***predecessor***.
- set of edges $\{(v.\pi, v) : v \neq s\}$ forms a tree.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

- Compute only $v.d$, not $v.\pi$. *[See book for $v.\pi$.]*
- Omitting colors of vertices. *[Used in book to reason about the algorithm. We'll skip them here.]*

Idea

Send a wave out from s.

- First hits all vertices 1 edge from s.
- From there, hits all vertices 2 edges from s.
- Etc.

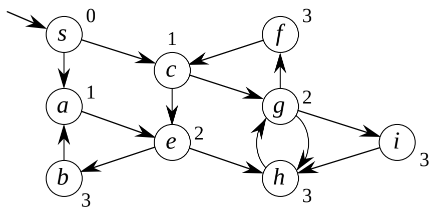
Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has hit v but has not come out of v yet.

```
BFS($V, E, s$)
  for each $u \in V - \{s\}$
      $u.d = \infty$
  $s.d = 0$
  $Q = \emptyset$
  ENQUEUE($Q, s$)
  while $Q \neq \emptyset$
      $u$ = DEQUEUE($Q$)
      for each $v \in G.Adj[u]$
          if $v.d == \infty$
              $v.d = u.d + 1$
              ENQUEUE($Q, v$)
```

Example

directed graph *[undirected example in book]* .



Can show that Q consists of vertices with d values.

$i \quad i \quad i \quad \ldots \quad i \quad i+1 \quad i+1 \quad \ldots \quad i+1$

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.

Actual proof of correctness is a bit trickier. See book.

BFS may not reach all vertices.

Time $= O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and we examine (u, v) only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

Depth-first search

Input: $G = (V, E)$, directed or undirected. No source vertex given!

Output: 2 ***timestamps*** on each vertex:

- $v.d$ = ***discovery time***
- $v.f$ = ***finishing time***

These will be useful for other algorithms later on.

Can also compute $v.\pi$. *[See book.]*

Will methodically explore *every* edge.

- Start over from different vertices as necessary.

As soon as we discover a vertex, explore from it.

- Unlike BFS, which puts a vertex on a queue so that we explore from it later.

As DFS progresses, every vertex has a ***color***:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

Discovery and finishing times:

- Unique integers from 1 to $2|V|$.
- For all v, $v.d < v.f$.

In other words, $1 \le v.d < v.f \le 2|V|$.

Pseudocode

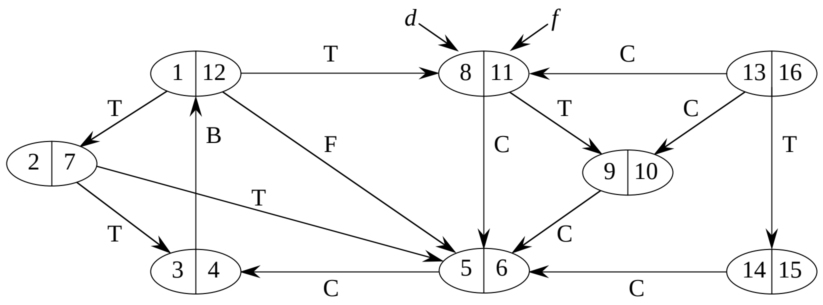
Uses a global timestamp *time*.

```
DFS(G)
  for each $u \in G.V$
      $u.color$ = WHITE
  $time = 0$
  for each $u \in G.V$
      if $u.color$ == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  $time = time + 1$
  $u.d = time$
  $u.color$ = GRAY                  // discover $u$
  for each $v \in G.Adj[u]$         // explore $(u, v)$
      if $v.color$ == WHITE
          DFS-VISIT(v)
  $u.color$ = BLACK
  $time = time + 1$
  $u.f = time$                      // finish $u$
```

Example

[Go through this example, adding in the d and f values as they're computed. Show colors as they change. Don't put in the edge types yet.]



Time $= \Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

DFS forms a ***depth-first forest*** comprised of > 1 ***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.

Theorem (Parenthesis theorem)
[Proof omitted.]

For all u, v, exactly one of the following holds:

1. $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ (i.e., the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint) and neither of u and v is a descendant of the other.
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u.
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v.

So $u.d < v.d < u.f < v.f$ *cannot* happen.

Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

Corollary
v is a proper descendant of u if and only if $u.d < v.d < v.f < u.f$.

Theorem (White-path theorem)
[Proof omitted.]

v is a descendant of u if and only if at time $u.d$, there is a path $u \leadsto v$ consisting of only white vertices. (Except for u, which was *just* colored gray.)

Classification of edges

- ***Tree edge:*** in the depth-first forest. Found by exploring (u, v).
- ***Back edge:*** (u, v), where u is a descendant of v.
- ***Forward edge:*** (u, v), where v is a descendant of u, but not a tree edge.
- ***Cross edge:*** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

[Now label the example from above with edge types.]

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

Theorem
[Proof omitted.]

In DFS of an *un*directed graph, we get only tree and back edges. No forward or cross edges.

Topological sort

Directed acyclic graph (dag)

A directed graph with no cycles.

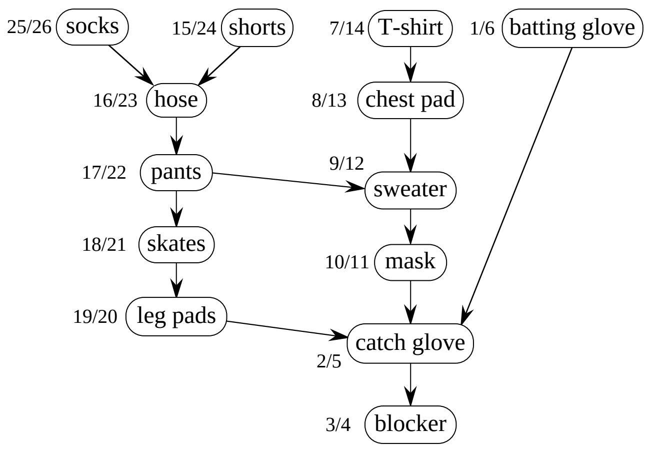
Good for modeling processes and structures that have a ***partial order:***

- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$.

Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order. In fact, that's what a topological sort will do.

Example

Dag of dependencies for putting on goalie equipment: *[Leave on board, but show without discovery and finishing times. Will put them in later.]*

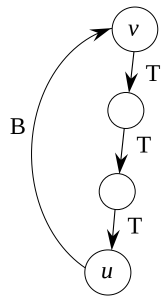


Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof $\Rightarrow$: Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest.



Therefore, there is a path $v \leadsto u$, so $v \leadsto u \rightarrow v$ is a cycle.

$\Leftarrow$: Show that cycle $\Rightarrow$ back edge.

Suppose G contains cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, vertices of c form a white path $v \leadsto u$ (since v is the first vertex discovered in c). By white-path theorem, u is descendant of v in depth-first forest. Therefore, (u, v) is a back edge. ■ (lemma)

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

```
TOPOLOGICAL-SORT(G)
  call DFS(G) to compute finishing times v.f for all v ∈ G.V
  output vertices in order of decreasing finishing times
```

Don't need to sort by finishing times.

- Can just output vertices as they're finished and understand that we want the *reverse* of this list.
- Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topologically sorted order.

Time

$\Theta(V + E)$.

Do example. *[Now write discovery and finishing times in goalie equipment example.]*

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

Correctness

Just need to show if $(u, v) \in E$, then $v.f < u.f$.
When we explore (u, v), what are the colors of u and v?

- u is gray.

- Is v gray, too?
  - *No*, because then v would be ancestor of u.
    $\Rightarrow (u, v)$ is a back edge.
    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).
- Is v white?
  - Then becomes descendant of u.
    By parenthesis theorem, $u.d < v.d < \underline{v.f < u.f}$.
- Is v black?
  - Then v is already finished.
    Since we're exploring (u, v), we have not yet finished u.
    Therefore, $v.f < u.f$. ■

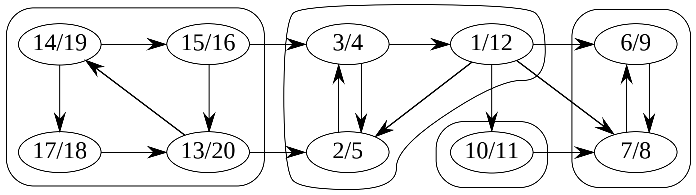
Strongly connected components

Given directed graph $G = (V, E)$.

A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example

[Just show SCC's at first. Do DFS a little later.]



Algorithm uses G^{T} = ***transpose*** of G.

- $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
- G^{T} is G with all edges reversed.

Can create G^{T} in $\Theta(V + E)$ time if using adjacency lists.

Observation

G and G^{T} have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^{T}.)

Component graph

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