Week -7-8
Topic - Graph Algorithms
CSE – 5311

Prepared by:-
Sushruth Puttaswamy
Lekhendro Lisham

Contents

- Different Parts of Vertices used in Graph Algorithms
- Analysis of DFS
- Analysis of BFS
- Minimum Spanning Tree
- Kruskal's Algorithm
- Shortest Path's
- Dijkstra's Algorithm

Representing Graphs

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- G may be either directed or undirected.
- Two common ways to represent graphs for algorithms:
  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$.

Adjacency Lists

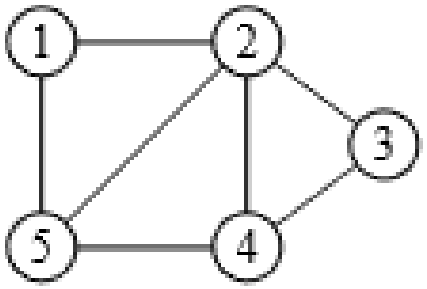
Array Adj of $|V|$ lists, one per vertex.

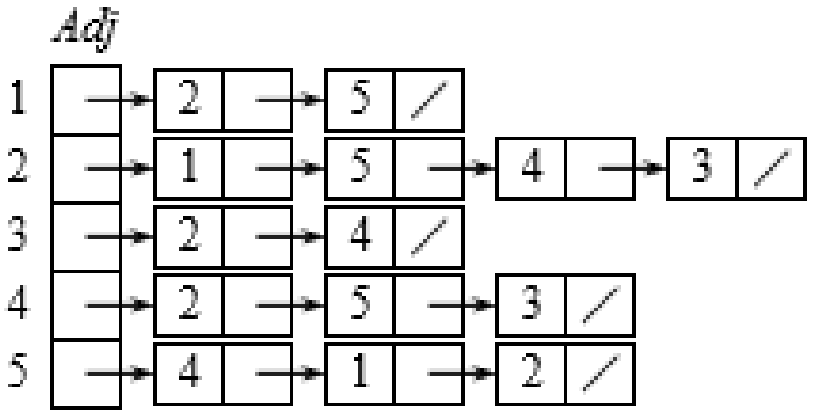
Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

Example

For an undirected graph:





Adjacency Lists

If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

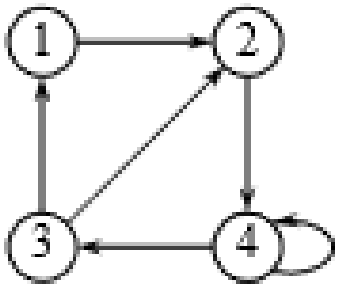
Space: $\Theta(V + E)$.

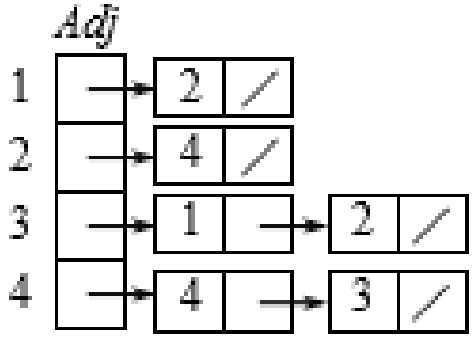
Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

Example

For a directed graph:





Adjacency Matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise}. \end{cases}$$

	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

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

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

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 $s \rightsquigarrow v$.

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

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.

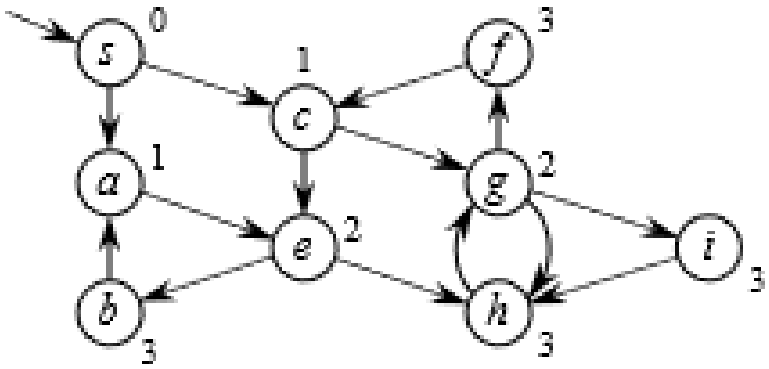
Use FIFO queue Q to maintain wavefront.

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

Breadth First Search

```
BFS(V, E, s)
  for each u ∈ V − {s}
        u.d = ∞
  s.d = 0
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ G.Adj[u]
              if v.d == ∞
                    v.d = u.d + 1
                    ENQUEUE(Q, v)
```

Directed Graph Example

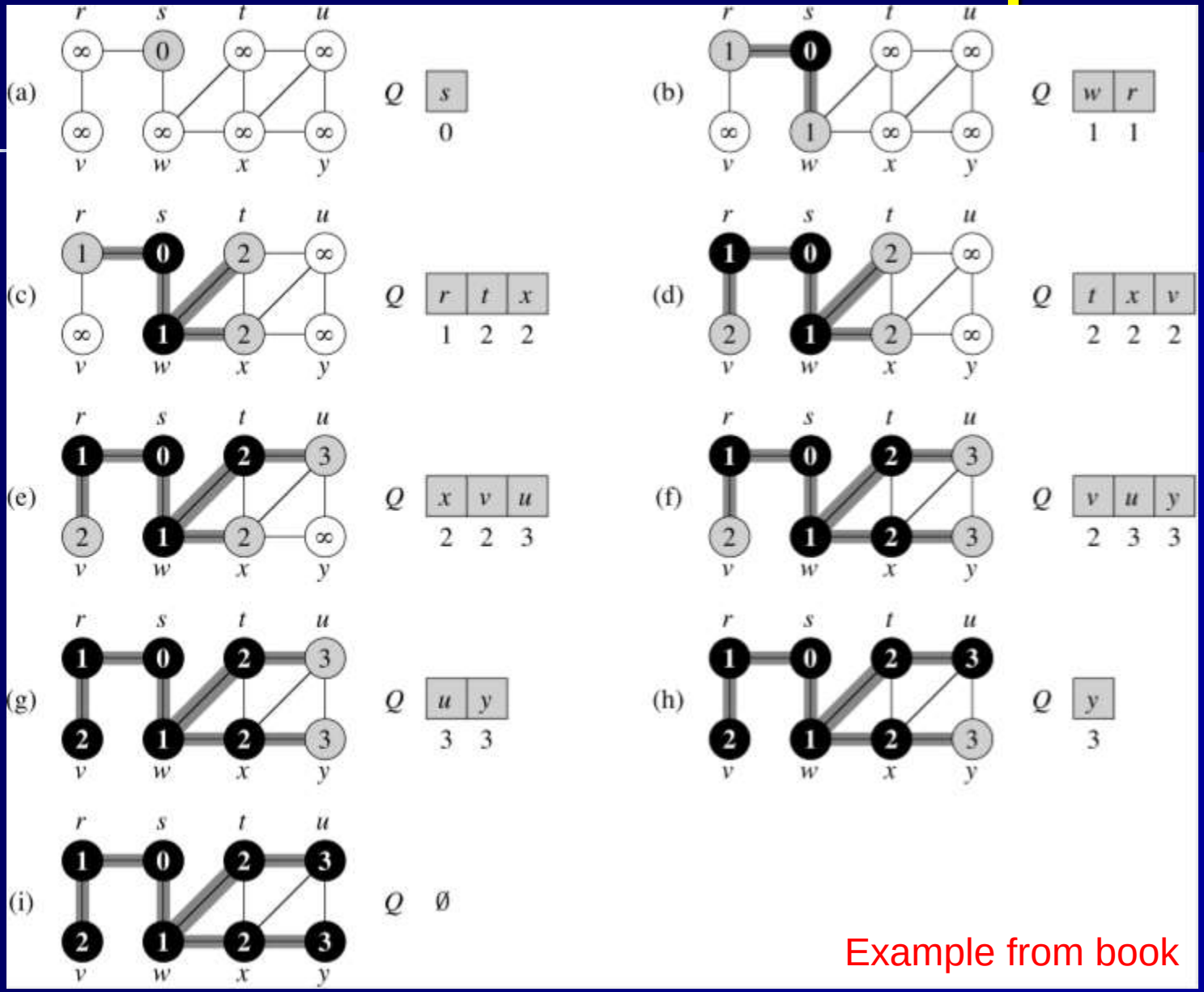


Can show that Q consists of vertices with d values.

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

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

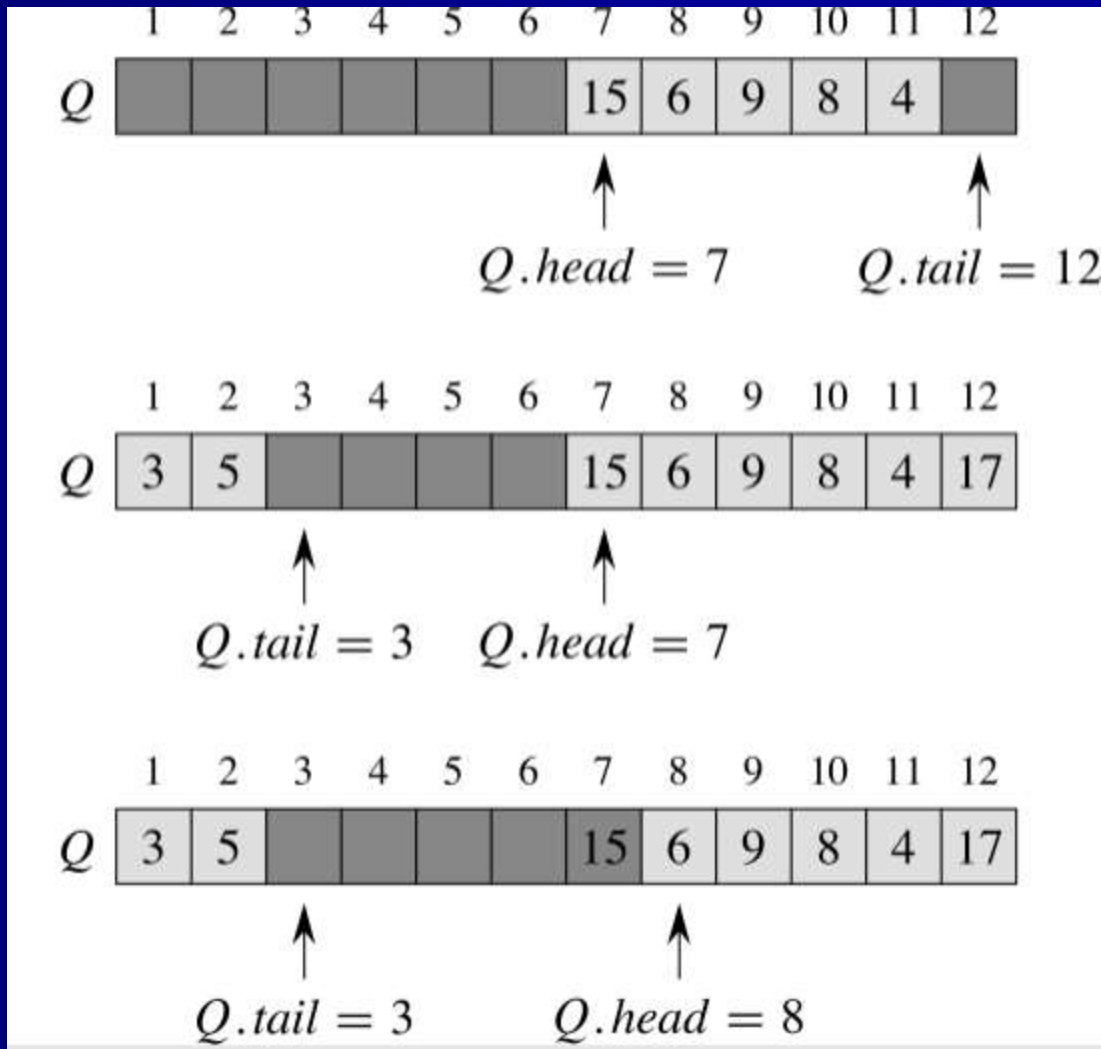
Breadth First Search Example 2



Example from book

Queues

- A queue implemented using an array $Q[1 \ldots 12]$. Queue elements are light grey cells
- Enqueue(Q, 17), Enqueue(Q, 3), and, Enqueue(Q, 5) are shown
- Dequeue(Q) returns the key value 15, formerly at the head of Q



Analysis of BFS

For a graph $G = (V, E)$

- When **Adjacency List** is used
  - ✓ Time complexity is $O(V+E)$
  - ✓$O(V)$ because every vertex is enqueued at most once
  - ✓$O(E)$ because every vertex is dequeued at most once and we examine (u, v) only when u is dequeued. Therefore every edge is examined at most once if directed, at most twice if undirected
- When **Adjacency Matrix** is used
  - ✓This requires an array for maintaining the state of a node.
  - ✓Scanning each row for checking the connectivity of a vertex is in order $O(V)$
  - ✓So, Complexity is $O(V^2)$

Brief Description of DFS

- Explores a graph using greedy method.
- Vertices are divided into 3 parts as it proceeds i.e. into Finished, Discovered & Undiscovered
- Fringe part is maintained in a Stack.
- The path is traced from last visited node.
- Element pointed by *TOS* is the next vertex to be considered by DFS algorithm to select for Visited part.

Different Parts of Vertices
(used in Graph Algorithms)

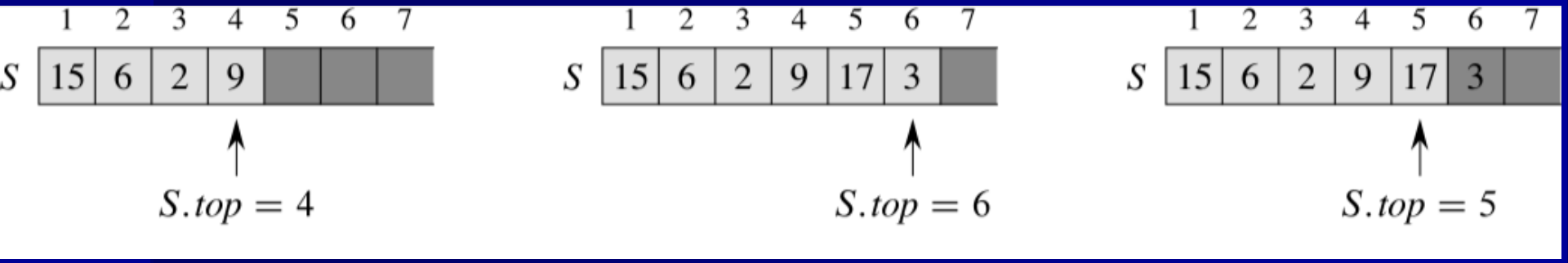
Vertices of the Graph are kept in 3 parts as below:

- **Finished**: Vertices already selected.
- **Discovered**: Vertices to be considered for next selection.
- **Undiscovered**: Vertices yet to consider as a possible candidate.

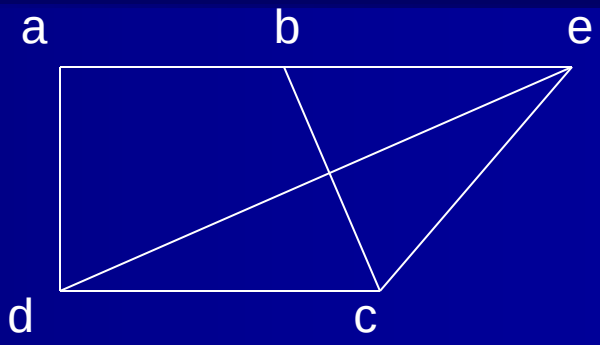
❖ *For DFS, the **Discovered part** can be kept in a **STACK** while for **BFS, QUEUE** is used.*

Stacks

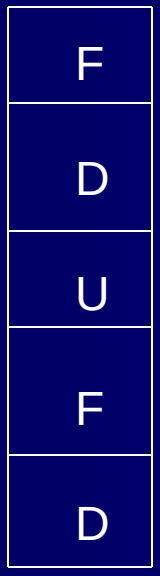
- An array implementation of a stack S, is shown below. Stack elements appear in the lightly shaded slots.
- The top element $S.top$ (aka TOS) is 9 at location 4
- PUSH(S,17) and PUSH(S, 3) are shown in the second figure.
- POP(S) returns 3 and sets $S.top$ to location 5



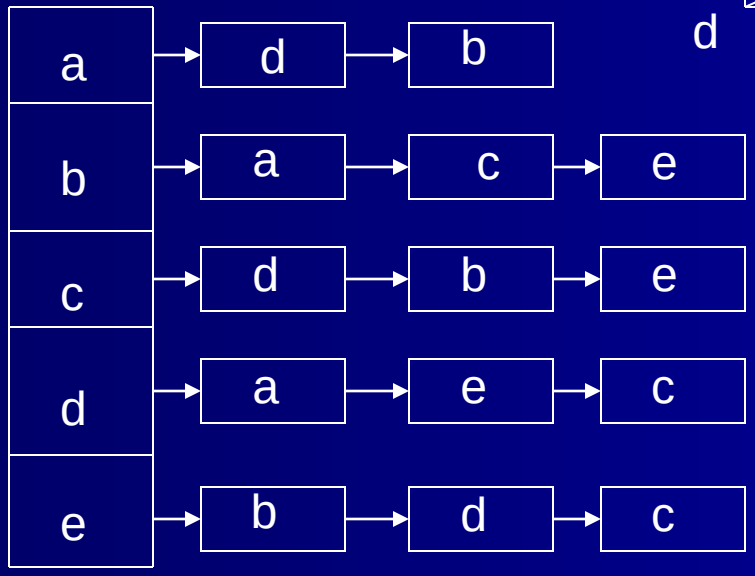
Data Structures used for DFS



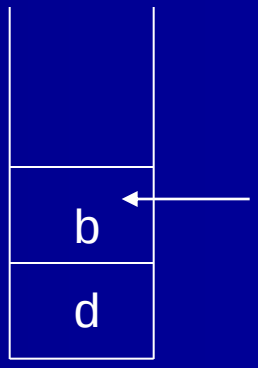
Graph with 5 Vertices



(F/D/U)
1-D Array



Adjacency list



Fringe
(Stack)

Depth First Search, Discussion

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 $v.\pi$.

Will methodically explore *every* edge.

- Start over from different vertices as necessary.

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

DFS Pseudocode

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ 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 ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

DFS Description

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:

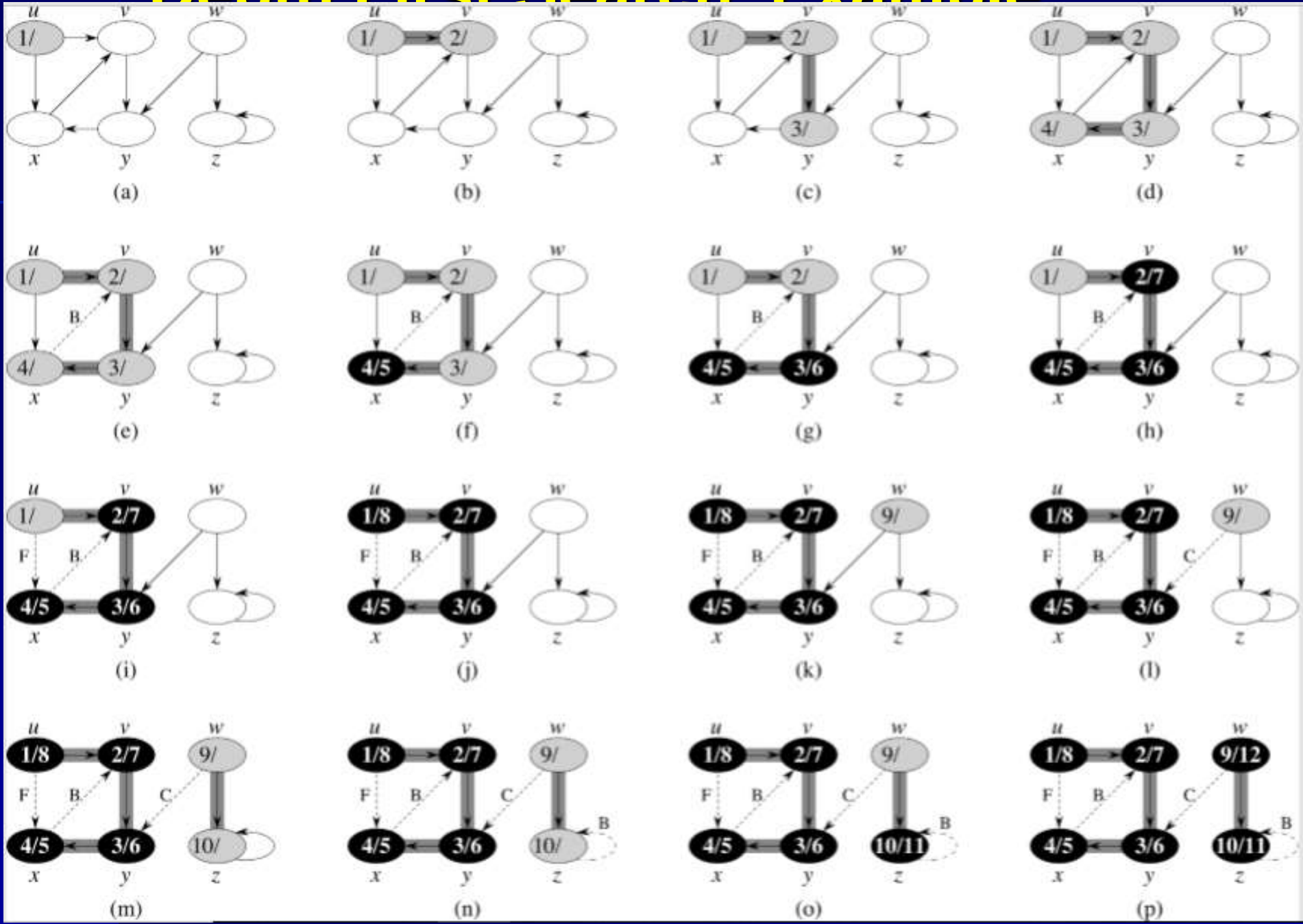
- Unique integers from 1 to $2|V|$.
- For all v, $v.d < v.f$.

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

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ 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 ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

Depth First Search. Example



Analysis of DFS

For a Graph $G = (V, E)$

- When **Adjacency List** is used
  - ✓ Complexity is $\Theta(V+E)$
  - ✓ Θ not just O, since guaranteed to examine every vertex and edge
- When **Adjacency Matrix** is used
  - ✓ This again requires a stack to maintain the fringe & an array for maintaining the state of a node.
  - ✓ Scanning each row for checking the connectivity of a Vertex is in order $O(V)$.
  - ✓ So, Complexity is $O(V^2)$
- DFS forms a *depth-first forest* (>1 *depth-first tree*). Each tree is made of edges (u, v) such that u is grey and v is white when (u, v) is explored

Parenthesis Theorem

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

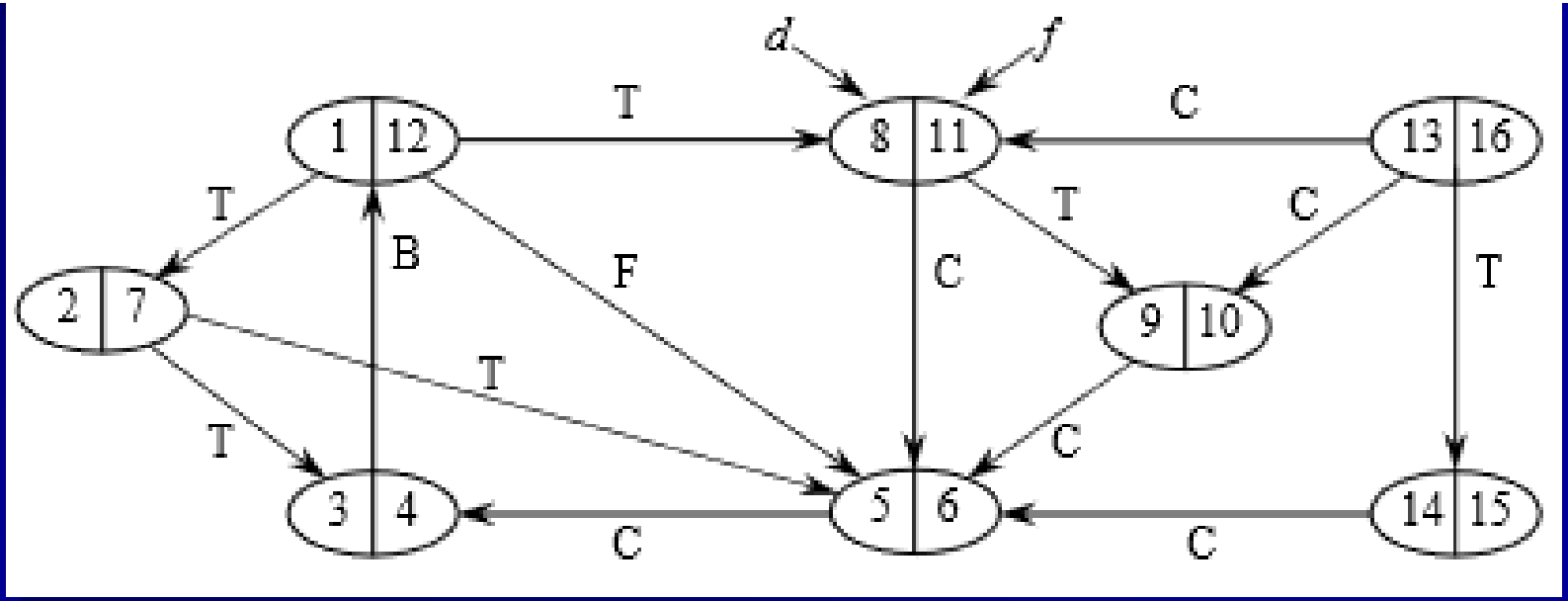
v is a proper descendant of u if and only if $u.d < v.d < v.f < u.f$.

White Theorem (proof in book)

v is a descendant of u if and only if at time $u.d$, there is a path $u \rightsquigarrow v$ 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.



In an *undirected* graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Further, there are no forward or cross edges in undirected graphs

Applications of DFS

- ✓ To check if a graph is connected. **The algorithm ends when the Stack becomes empty.**
  - **Graph is CONNECTED if all the vertices are visited.**
  - **Graph is DISCONNECTED if one or more vertices remained unvisited.**
- ✓ Finding the connected components of a disconnected graph. **Repeat DFS until all the vertices become visited. Next vertex can also be taken randomly.**
- ✓ Check if a graph is bi-connected **(i.e. if 2 distinct paths exist between any 2 nodes.).**
- ✓ Detecting if a given directed graph is strongly connected **(path exists between a & b and b & a).**

DFS is the champion algorithm for all connectivity problems

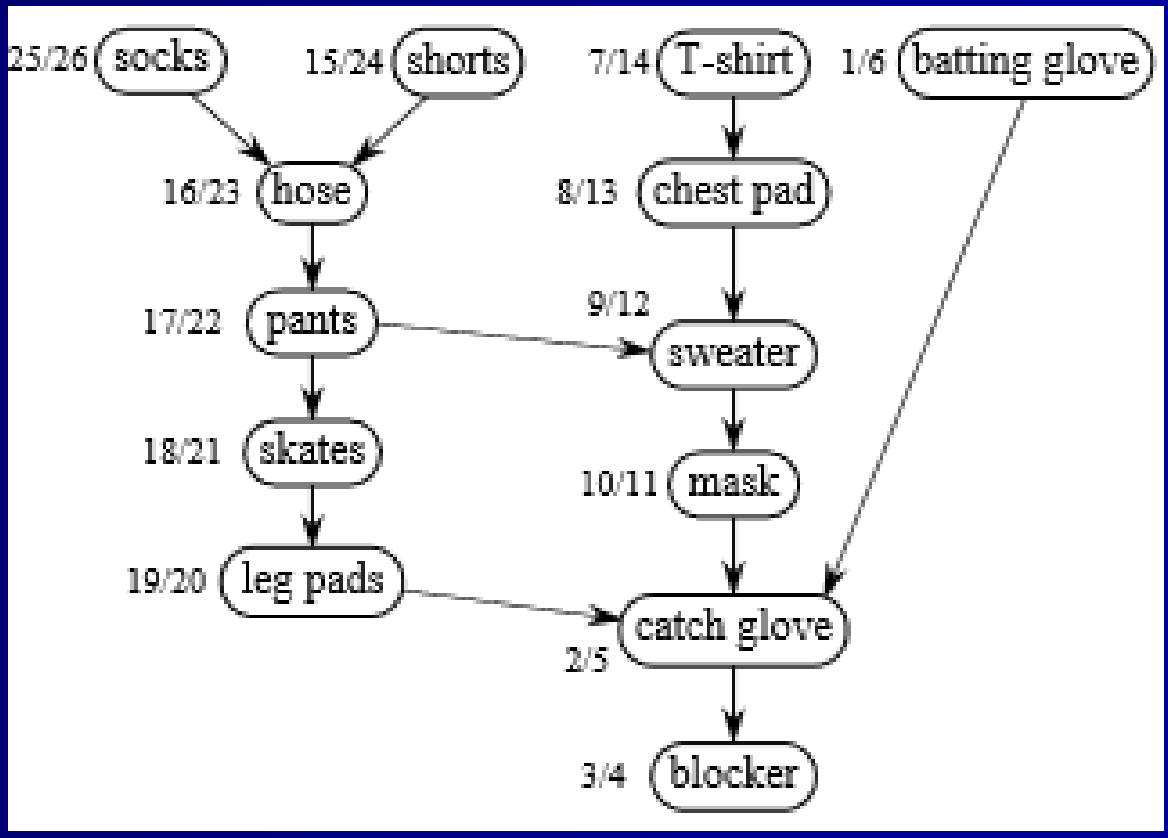
Directed Acyclic Graph (DAG)

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 $b > c$.

Example:
DAG of dependencies for putting on goalie equipment



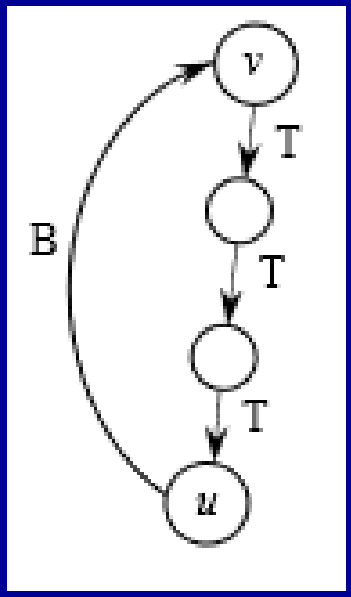
Lemma

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

Proof:

Suppose DFS discovers a back edge (u, v). Then v is ancestor of u, meaning there is a cycle in G

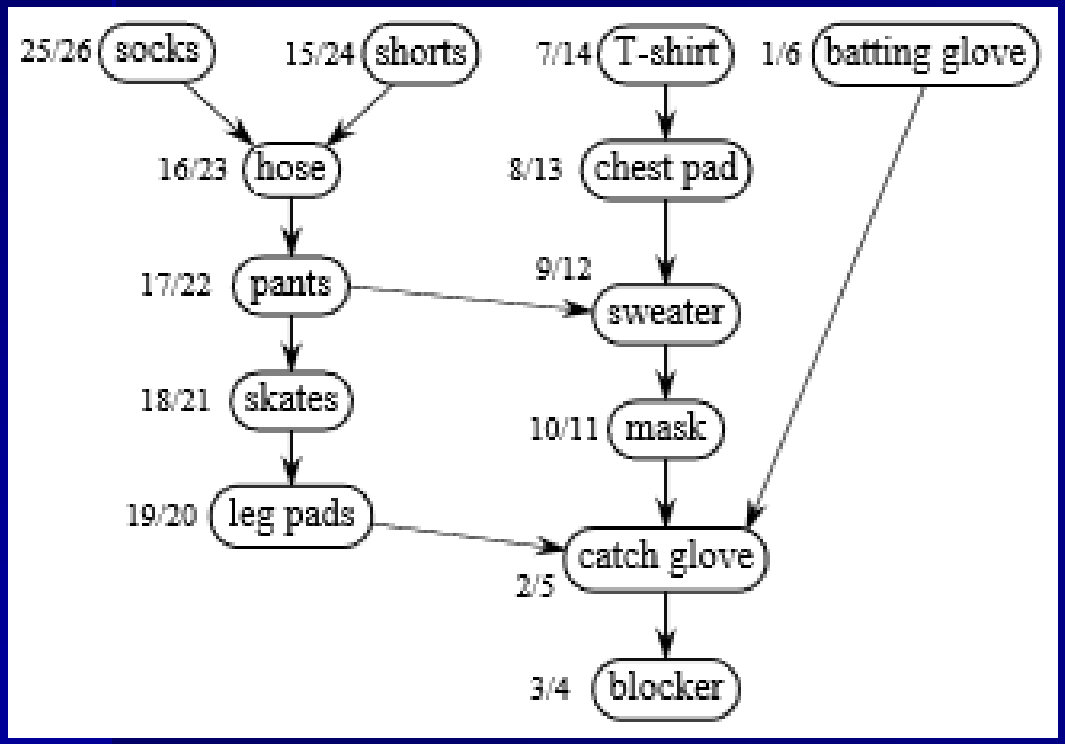
Conversely if there is no cycle in G DFS will not discover an edge returning back to an ancestor



Topological Sort

- A topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering
- We can use DFS for performing topological sort

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

Topological Sort, Pseudocode

TOPOLOGICAL-SORT(G)

call DFS(G) to compute finishing times $v.f$ for all $v \in G.V$
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.

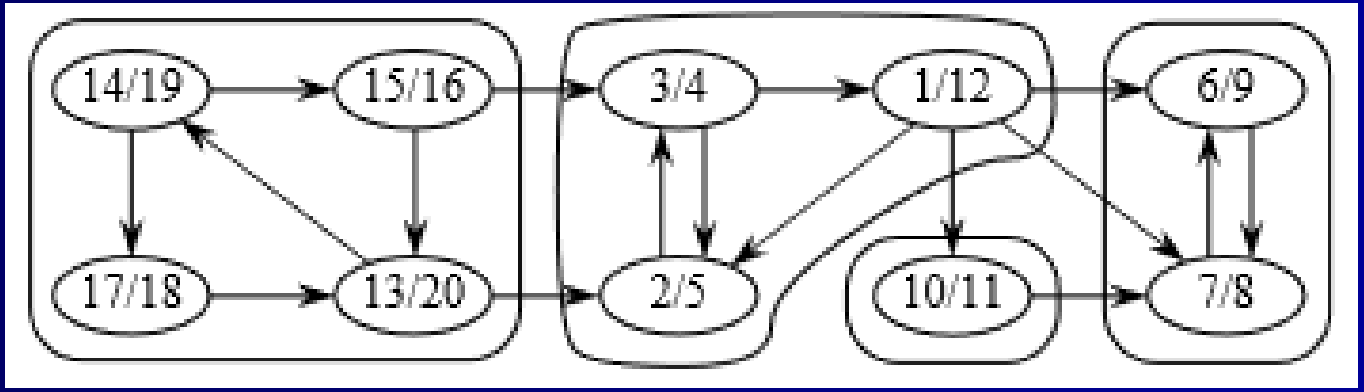
Time

$\Theta(V + E)$.

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 $v \rightsquigarrow u$.



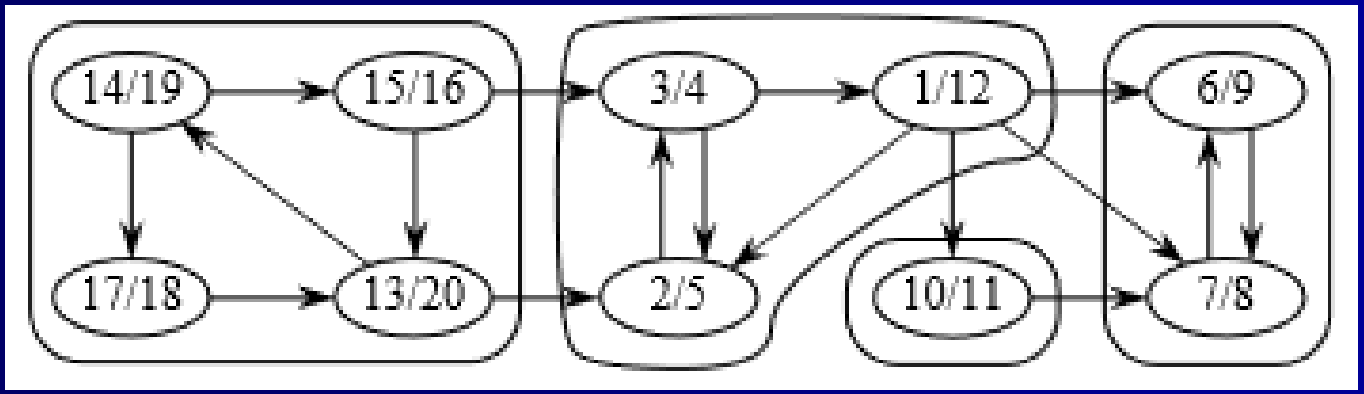
- Many algorithms that work with directed graphs begin after decomposing graphs into strongly connected components (SCC)
- Algorithms run separately on each SCC and then combine the solutions

Transpose of a Graph

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

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

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

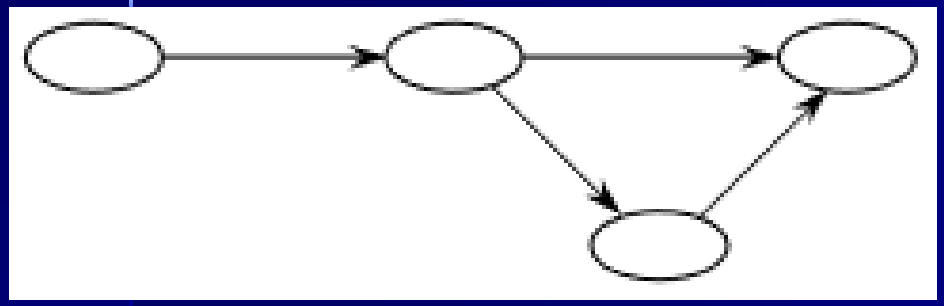


Observation

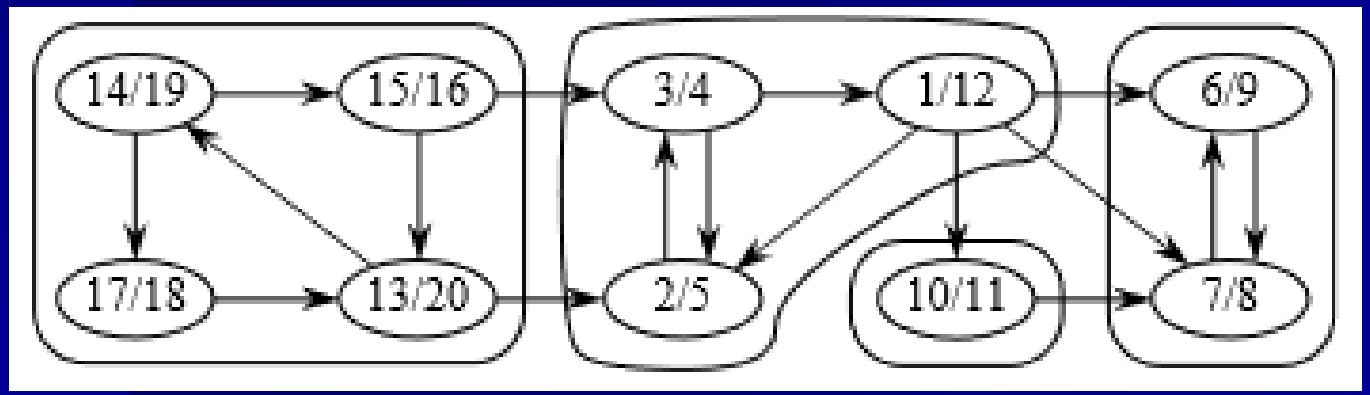
G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T.)

Component Graph

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



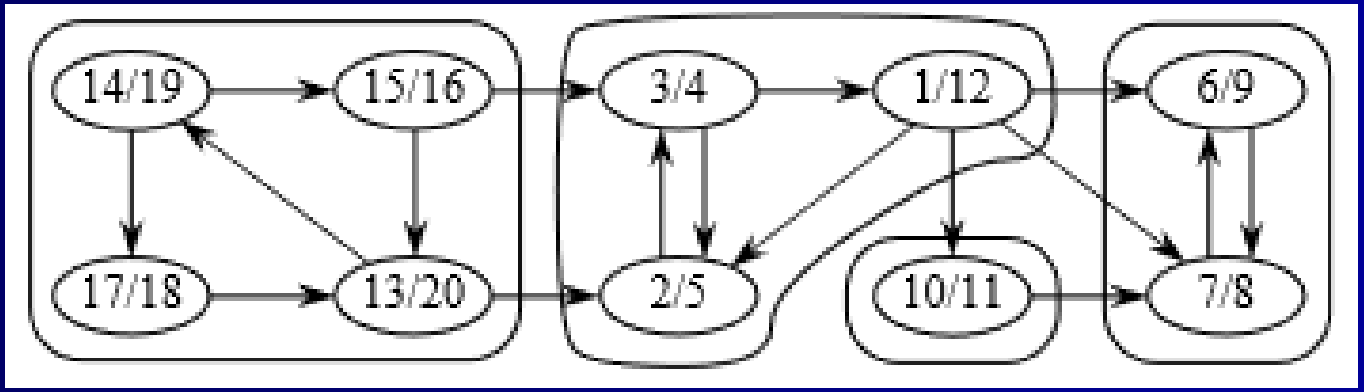
- **For example, the component graph above represents the graph below**



Lemma

G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof Suppose there is a path $v' \rightsquigarrow v$ in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)



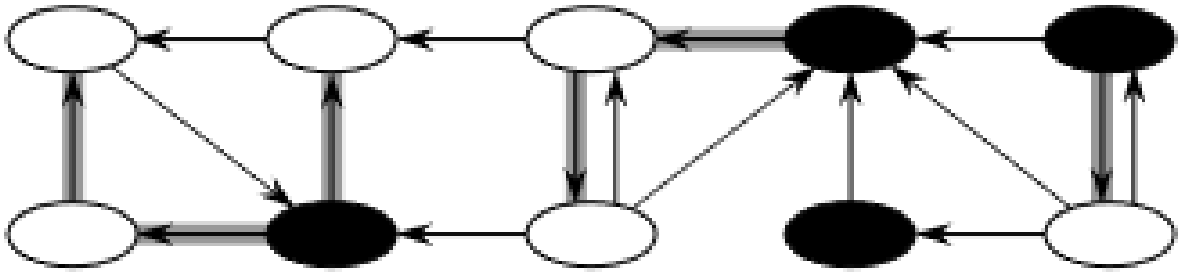
SCC Pseudocode

SCC(G)

call DFS(G) to compute finishing times $u.f$ for all u
compute G^T
call DFS(G^T), but in the main loop, consider vertices in order of decreasing $u.f$
(as computed in first DFS)
output the vertices in each tree of the depth-first forest formed in second DFS
as a separate SCC

Example:

1. Do DFS
2. G^T
3. DFS (roots blackened)



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

SCC Pseudocode, Working

SCC(G)

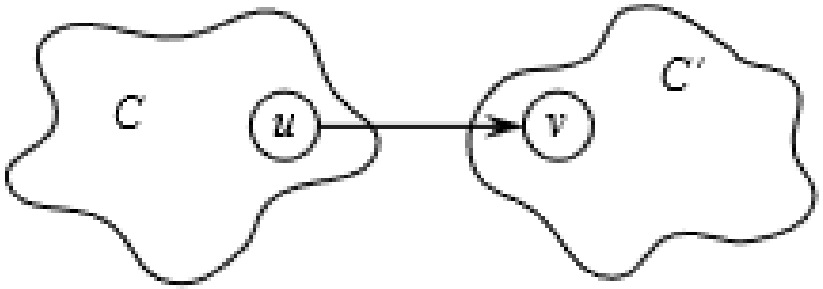
call DFS(G) to compute finishing times $u.f$ for all u
compute G^T
call DFS(G^T), but in the main loop, consider vertices in order of decreasing $u.f$
(as computed in first DFS)
output the vertices in each tree of the depth-first forest formed in second DFS
as a separate SCC

To prove that it works, first deal with 2 notational issues:

- Will be discussing $u.d$ and $u.f$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min_{u \in U} \{u.d\}$ (earliest discovery time)
  - $f(U) = \max_{u \in U} \{u.f\}$ (latest finishing time)

Lemma, Case 1

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



Then $f(C) > f(C')$.

Proof Two cases, depending on which SCC had the first discovered vertex during the first DFS.

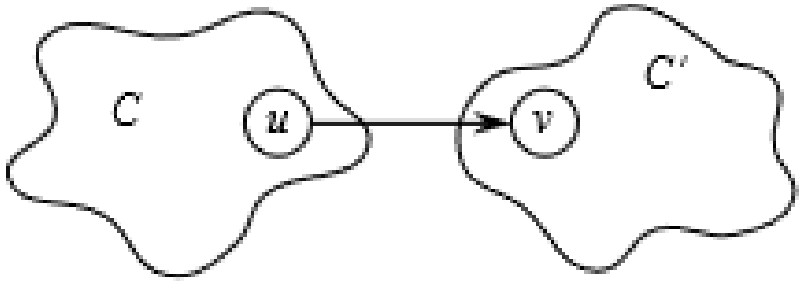
- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $x.d$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

  By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

  By the parenthesis theorem, $x.f = f(C) > f(C')$.

Lemma, Case 2

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



Then $f(C) > f(C')$.

- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $y.d$, all vertices in C' are white and there is a white path from y to each vertex in C' $\Rightarrow$ all vertices in C' become descendants of y. Again, $y.f = f(C')$.

  At time $y.d$, all vertices in C are white.

  By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.

  So no vertex in C is reachable from y.

  Therefore, at time $y.f$, all vertices in C are still white.

  Therefore, for all $w \in C$, $w.f > y.f$, which implies that $f(C) > f(C')$.

Two Corollaries

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof $(u, v) \in E^T \Rightarrow (v, u) \in E$. Since SCC's of G and G^T are the same, $f(C') > f(C)$. ■ (corollary)

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$, and suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^T.

Proof It's the contrapositive of the previous corollary. ■

Now the Understanding of SCC

When we do the second DFS, on G^T, start with SCC C such that $f(C)$ is maximum. The second DFS starts from some $x \in C$, and it visits all vertices in C.

Corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^T.

Therefore, DFS will visit *only* vertices in C.

Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C. DFS visits all vertices in C', but the only edges out of C' go to C, *which we've already visited.*

Therefore, the only tree edges will be to vertices in C'.

We can continue the process.

Each time we choose a root for the second DFS, it can reach only

- vertices in its SCC—get tree edges to these,
- vertices in SCC's *already visited* in second DFS—get *no* tree edges to these.

Minimum Spanning Tree (MST)

- A spanning tree of a connected Graph G is a sub-graph of G which covers all the vertices of G.
- A minimum-cost spanning tree is one whose edge weights add up to the least among all the spanning trees.
- A given graph may have more than one spanning tree.
- A MST has $|V|$-1 edges and has no cycles
- DFS & BFS give rise to spanning trees, but they don't consider weights.

MST, Example

- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that

  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

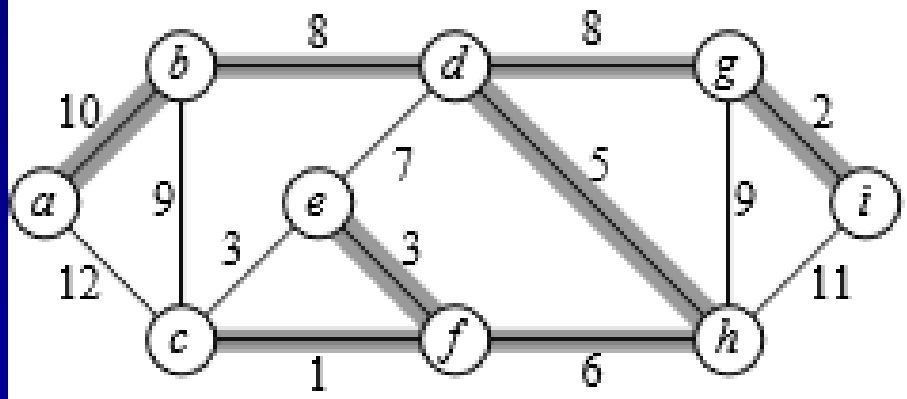
Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that

  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

Example of such a graph *[edges in MST are shaded]* :



In this example, there is more than one MST. Replace edge (e, f) in the MST by (c, e). Get a different spanning tree with the same weight.

Growing a MST

- We will build a set A of edges.
- Initially, A has no edges.
- As we add edges to A, maintain a loop invariant:

  Loop invariant: A is a subset of some MST.

- Add only edges that maintain the invariant. If A is a subset of some MST, an edge (u, v) is ***safe*** for A if and only if $A \cup \{(u, v)\}$ is also a subset of some MST. So we will add only safe edges.

Loop Invariant

A statement that allows a program to execute repeatedly. An invariant of a loop is a property that holds before (and after) each repetition. Loop invariants help to keep an algorithm correct

Generic MST Algorithm

```
GENERIC-MST(G, w)
  A = ∅
  while A is not a spanning tree
      find an edge (u, v) that is safe for A
      A = A ∪ {(u, v)}
  return A
```

How do we find *safe edges*?

Finding a Safe Edge (Greedy Algorithm)

Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?

Intuitively: Let $S \subset V$ be any set of vertices that includes c but not f (so that f is in $V - S$). In any MST, there has to be one edge (at least) that connects S with $V - S$. Why not choose the edge with minimum weight? (Which would be (c, f) in this case.)

Theorem

Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Some Definitions before the Proof

Let $S \subset V$ and $A \subseteq E$

- A ***cut*** $(S, V - S)$ is a partition of vertices into disjoint sets V and $S - V$.
- Edge $(u, v) \in E$ ***crosses*** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut ***respects*** A if and only if no edge in A crosses the cut.
- An edge is a ***light edge*** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edge crossing it.

Theorem

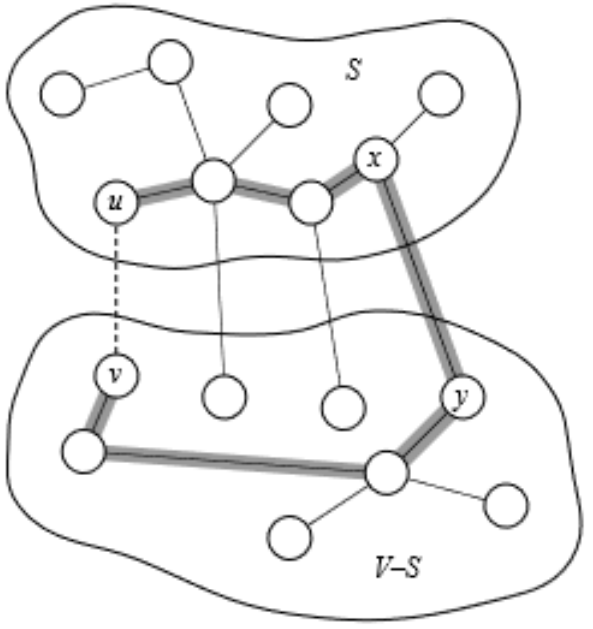
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Proof Let T be an MST that includes A.

If T contains (u, v), done.

So now assume that T does not contain (u, v). We'll construct a different MST T' that includes $A \cup \{(u, v)\}$.

Recall: a tree has unique path between each pair of vertices. Since T is an MST, it contains a unique path p between u and v. Path p must cross the cut $(S, V - S)$ at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), must have $w(u, v) \leq w(x, y)$.



To form T' from T:

- Remove (x, y). Breaks T into two components.
- Add (u, v). Reconnects.

So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

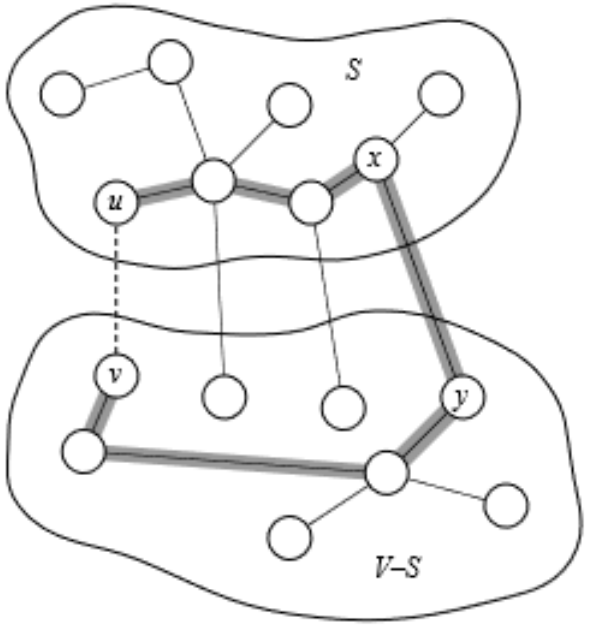
T' is a spanning tree.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) , \end{aligned}$$

since $w(u, v) \leq w(x, y)$. Since T' is a spanning tree, $w(T') \leq w(T)$, and T is an MST, then T' must be an MST.

Need to show that (u, v) is safe for A:

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
- $A \cup \{(u, v)\} \subseteq T'$.
- Since T' is an MST, (u, v) is safe for A.



Generic MST (Closing)

So, in GENERIC-MST:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V|-1$ safe edges, we're down to just one component.

Corollary

If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V - V_C)$), then (u, v) is safe for A.

This idea naturally leads to the algorithm known as Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's Algorithm

- An algorithm to find the minimum spanning tree of connected graph.
- It makes use of the previously mentioned properties.

Step 1: Initially all the edges of the graph are sorted based on their weights.

Step2: Select the edge with minimum weight from the sorted list in step 1.Selected edge shouldn't form a cycle. Selected edge is added into the tree or forest.

Step 3: Repeat step 2 till the tree contains all nodes of the graph.

- This algorithm works because when any edge is rejected it will be heavier than the already existing edge(s).

…. Kruskal's Algorithm

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \to \mathbb{R}$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Joseph Bernard Kruskal, Jr. (/ˈkrʌskəl/; January 29, 1928 – September 19, 2010) was an American mathematician, statistician, computer scientist and psychometrician.
Source: http://miblogdeteoriadegraficas.blogspot.com/2015/04/biografia-joseph-kruskal.htm

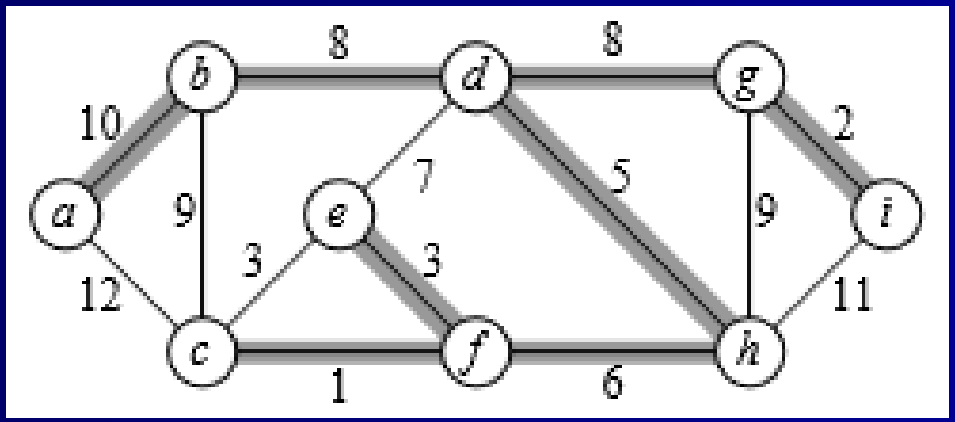
Pseudocode for Kruskal's Algorithm

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

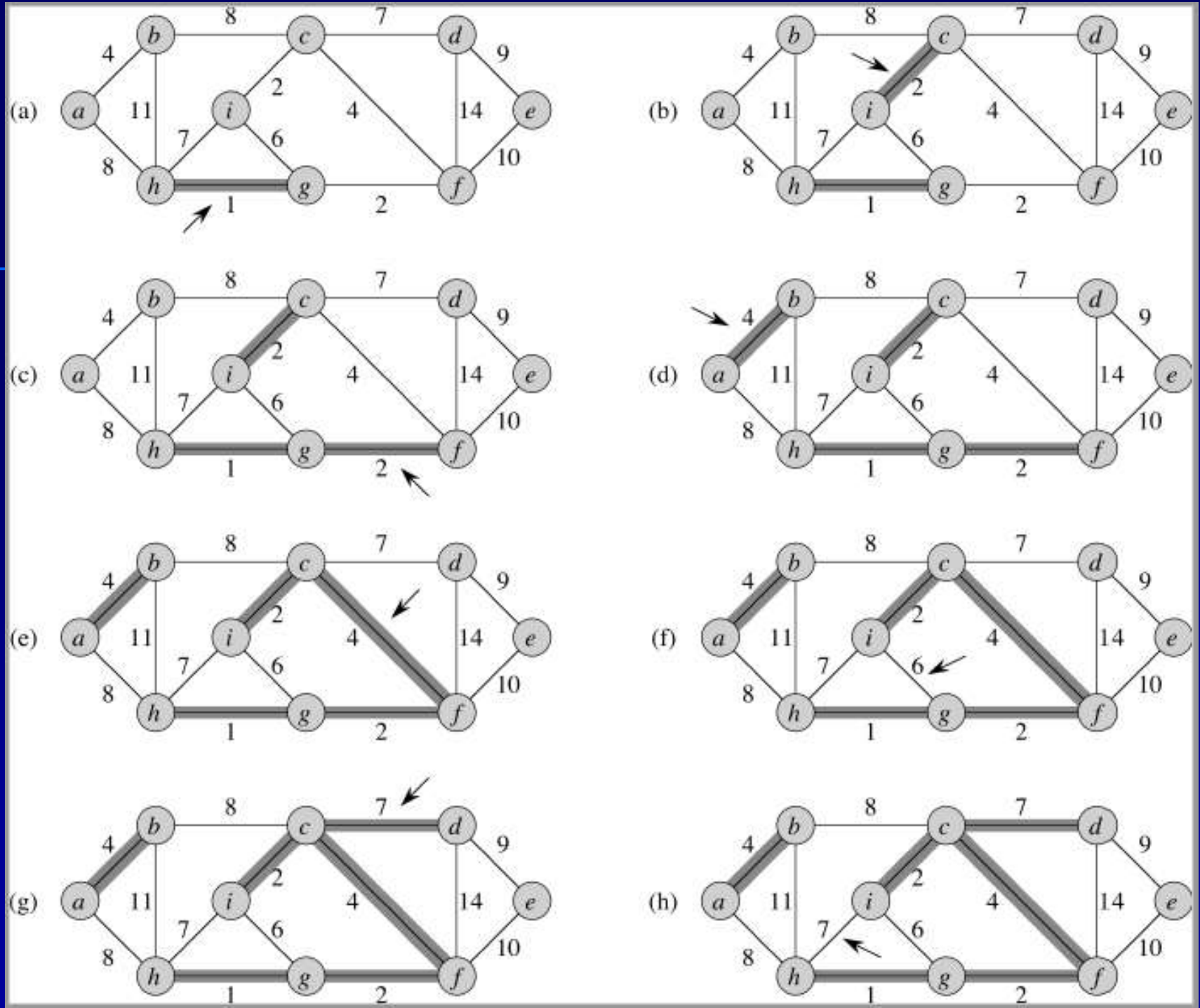
Kruskal's Algorithm, Example

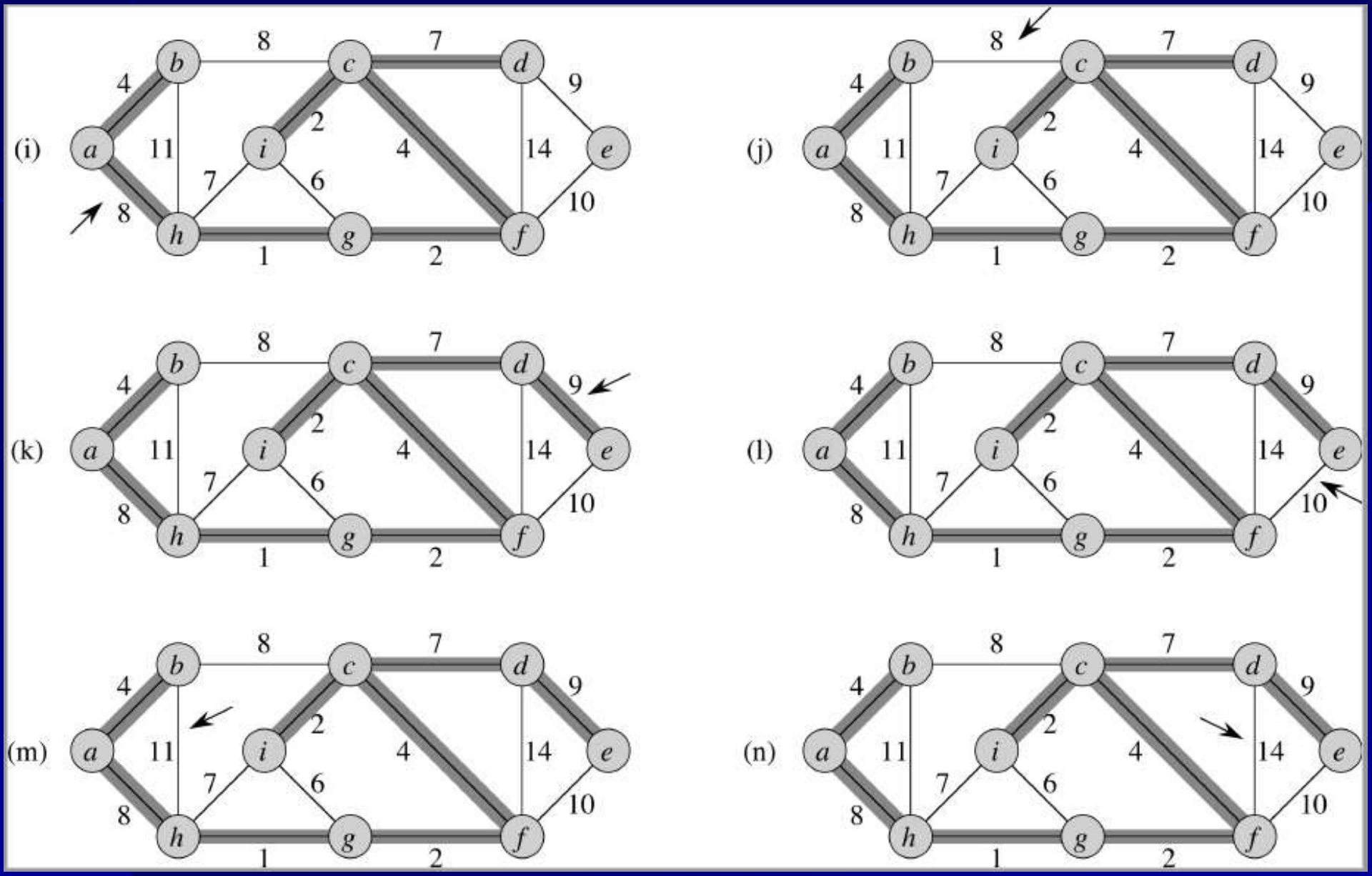
To see how Kruskal's algorithm works we repeat the example above:

- Edges are sorted in the order of their weights
- An edge is rejected if its end points belong to an existing tree as it may create a cycle
- Suppose we had examined (c, e) before (e, f). The algorithm would have found (c, e) safe and rejected (e, f)



(c, f)	:	safe
(g, i)	:	safe
(e, f)	:	safe
(c, e)	:	reject
(d, h)	:	safe
(f, h)	:	safe
(e, d)	:	reject
(b, d)	:	safe
(d, g)	:	safe
(b, c)	:	reject
(g, h)	:	reject
(a, b)	:	safe


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
7
6
10
1
2


(i)
(j)
(k)
(l)
(m)
(n)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2

Kruskal's Algorithm, Analysis

Initialize A:	$O(1)$
First **for** loop:	$\lvert V\rvert$ MAKE-SETs
Sort E:	$O(E \lg E)$
Second **for** loop:	$O(E)$ FIND-SETs and UNIONs

- Assuming the implementation of disjoint-set data structure, already seen in Chapter 21, that uses union by rank and path compression:

  $$O((V + E)\,\alpha(V)) + O(E \lg E)\,.$$

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.
- $|E| \leq |V|^2 \Rightarrow \lg|E| = O(2 \lg V) = O(\lg V)$.
- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

Useful Heuristics I

Union by rank: make the root of the smaller tree (fewer nodes) a child of the root of the larger tree.

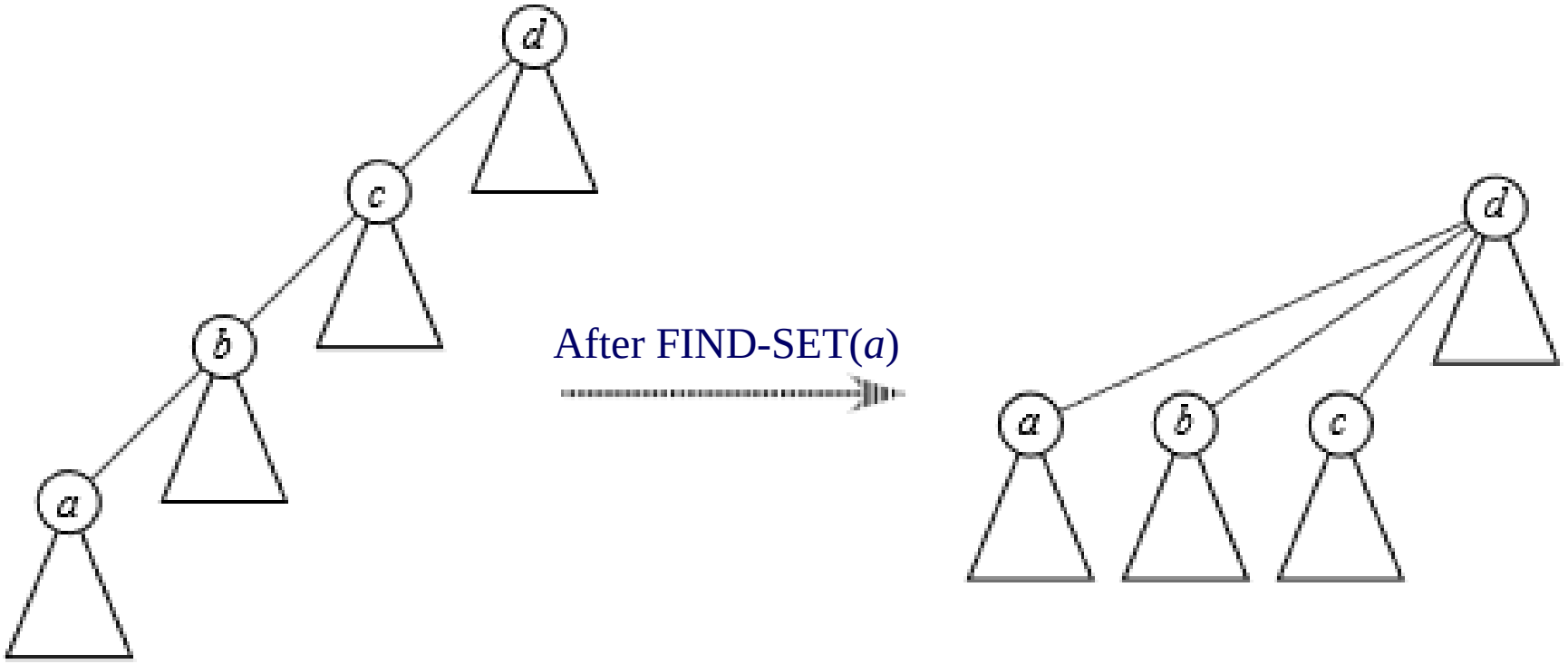
- Don't actually use *size*.
- Use *rank*, which is an upper bound on height of node. $O(\lfloor \lg n \rfloor)$
- Make the root with the smaller rank into a child of the root with the larger rank.

Notes

To implement union by rank, each element is associated with a rank. Initially a set has one element and a rank of zero. If two sets are unioned and have the same rank, the resulting set's rank is one larger; otherwise, if two sets are unioned and have different ranks, the resulting set's rank is the larger of the two. Ranks are used instead of height or depth because path compression will change the trees' heights over time

Useful Heuristics II

- ***Path compression:*** ***Find path*** = nodes visited during FIND-SET on the trip to the root. Make all nodes on the find path direct children of root.



Each node has two attributes, p (parent) and *rank*.

Pseudocode for Union-by-rank and Path Compression

```
MAKE-SET(x)
    x.p = x
    x.rank = 0

UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
    if x.rank > y.rank
        y.p = x
    else x.p = y
        // If equal ranks, choose y as parent and increment its rank.
        if x.rank == y.rank
            y.rank = y.rank + 1

FIND-SET(x)
    if x ≠ x.p
        x.p = FIND-SET(x.p)
    return x.p
```

Union-by-rank and Path Compression: Running Time

If use both union by rank and path compression, $O(m\,\alpha(n))$.

n	$\alpha(n)$
0–2	0
3	1
4–7	2
8–2047	3
2048–$A_4(1)$	4

What is $A_4(1)$?

- It is $\gg 10^{80} \approx$ number of atoms in observable universe
- n is the number of MAKE-SET operations
- m is the total number of MAKE-SET, UNION, and FIND-SET operations
- More on the next slide

A Very Quick Growing Function

For integers $k \geq 0$ and $j \geq 1$, $A_k(j)$ is defined as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)} & \text{if } k \geq 1 \end{cases}$$

Where

$A_{k-1}^{(0)} = j$ and $A_{k-1}^{(i)} = A_{k-1}(A_{k-1}^{(i-1)}(j))$ for $i \geq 1$

The function $A_k(j)$ strictly increases with both j and k

$A_k(j)$ is similar to Ackermann's function, which is another fast growing function.

Growing Function and Inverse

For $j = 1$, calculated values for $A_k(1)$ are

$$A_0(1) = 2;\ A_1(1) = 3; A_2(1) = 7;$$
$$A_3(1) = 2047;\ A_4(1) = 16^{512} \gg 10^{80}$$

Inverse of $A_k(n)$, for $n \geq 0$, is defined by

$$\alpha(n) = min\{k: A_k(1) \geq n\}$$

In other words, $\alpha(n)$ is the lowest level k for which $A_k(1)$ is at least n. Thus from the values of $A_k(1)$

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

So $\alpha(n) \leq 4$ for all practical purposes

Shortest Paths

- The problem is to find shortest paths among nodes.
- There are 3 different versions of this problem as shown below:
  1. **Single source single destination shortest path (SSSP)**
  2. **Single source all destinations shortest path (SSAP).**
  3. **All pairs shortest path.**
- For un-weighted graphs:
  - ✓ **BFS can be used for 1.**
  - ✓ **BFS can used to for 2.**
  - ✓ **Running BFS from all nodes is presently the best algorithm for 3.**

Dijkshitra's Algorithm

- This is an algorithm for finding the shortest path in a weighted graph.
- It finds the shortest path from a single source node to all other nodes.
- The shortest path from a single node to all destinations is a tree.
- The following applet gives a very good demonstration of the Dijkshitra's Algorithm. It finds the shortest distances from 1 city to all the remaining cities to which it has a path.

Applet

…. Dijkshitra's Algorithm

```
DIJKSTRA (G, w, s)
1.  INITIALIZE-SINGLE-SOURCE(G, s)
2.  S ← ø
3.  Q ← V[G]
4.  while Q ≠ ø
5.        do u ← EXTRACT-MIN(Q)
6.           S ← S U {u}
7.           for each vertex v ∈ Adj[u] do
8.                if d[v] > d[u] + w (u, v)
9.                then d[v] ← d[u] + w (u, v)
10.                       π [v] ← u
```

The algorithm repeatedly selects the vertex *u* with the minimum shortest-path estimate, adds *u* to S, and relaxes all edges leaving *u*.

…. Analysis of Dijkshitra's Alg.

- For a Graph G=(V, E) and n = |V| & m=|E|
  - ✓ When **Binary Heap** is used
    - Complexity is **O((m+n) log n)**
  - ✓ When **Fibonacci Heap** is used
    - Complexity is **O(m + n log n)**

HW

- 22.1-6, 22.1-7
- 22.2-3, 22.2-5
- 22.3-4, 22.3-5, 22.3-8, 22.3-12
- 22.4-3
- 23.1-1, 23.1-4, 23.1-6