EECS 3101

Prof. Andy Mirzaian

Graph Algorithms

STUDY MATERIAL:

- **[CLRS]** chapters 22, 23, 24, 25, 26
- **Lecture Notes**
- **Algorithmics**
  - Dijkstra
  - Minimum Spanning Tree
  - Max-Flow Min-Cut
  - Traveling Salesman

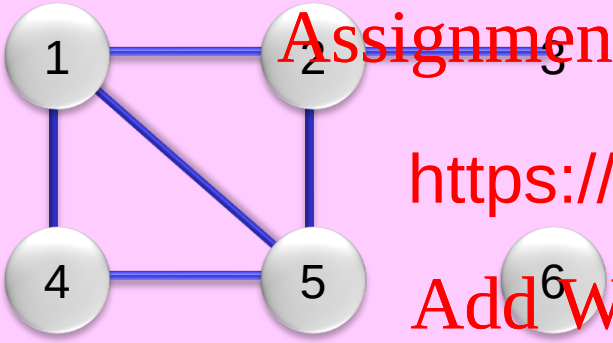
TOPICS

- **Graph Representations**
- **Graph Traversals:**
  - Breadth First Search
  - Depth First Search
- **Un-weighted**
  - Topological
  - Strongly Connected Componen
  - Bi-connected Components
- **Weighted Graphs:**
  - Minimum Spanning Trees
  - Shortest Paths
  - Max Flow
  - Matching

Graph

Graph G = (V, E)
V = V(G) = **vertex set** of G
E = E(G) = **edge set** of G (a set of pairs of vertices of G)

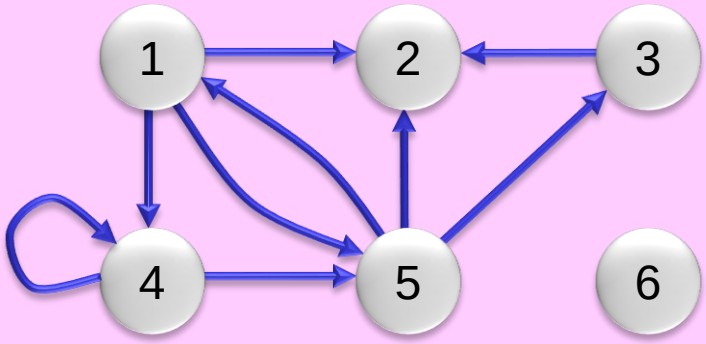
Undirected graph: edges are unordered pairs of vertices:



, 6 }

(1,5), (2,3), (2,5), (3,5), (4,5)}

Directed graph (or digraph): edges are ordered pairs of vertices:



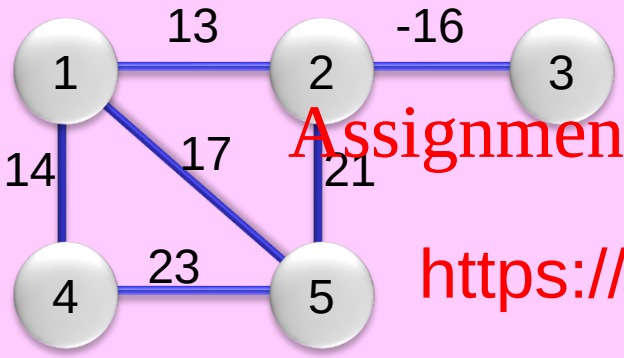
V = { 1, 2, 3, 4, 5, 6 }

E = { (1,2), (1,4), (1,5), (3,2),
(4,4), (4,5), (5,1), (5,2), (5,3)}

Edge Weighted Graph

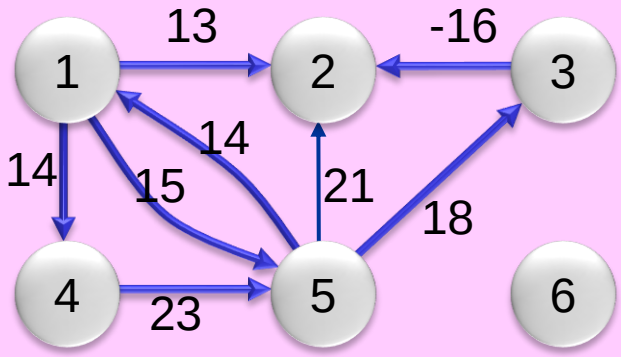
G = (V, E, w)

w: E $\longmapsto \Re$



E = {(1,2,13), (1,4,14), (1,5,17),
(2,3,-16), (2,5,21), (3,5,18), (4,5,23)}

,1) = 17.

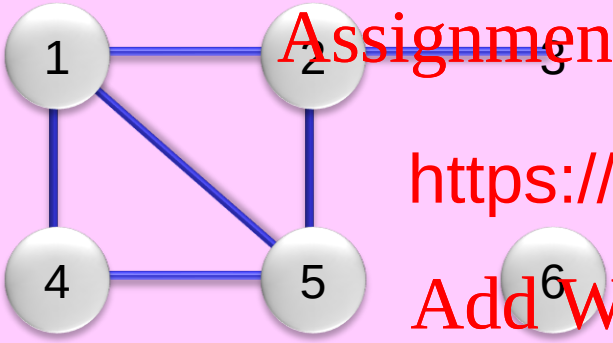


E = { (1,2,13), (1,4,14), (1,5,15), (3,2,-16)
(4,5,23), (5,1,14), (5,2,21), (5,3,18)}

e.g., w(1,5) = 15, w(5,1) = 14.

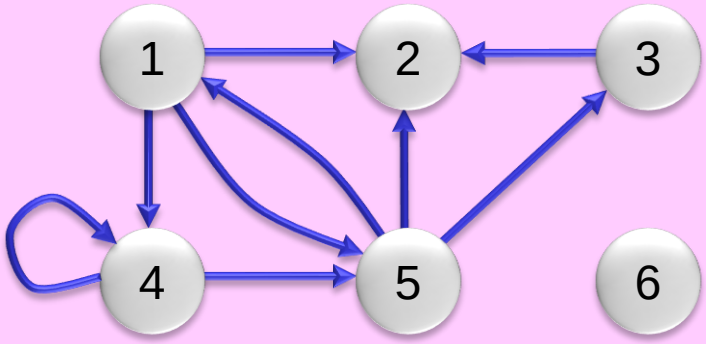
Adjacency Matrix

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E(G) \\ 0 & \text{otherwise} \end{cases}, \quad \text{for } i, j \in V(G).$$



A =

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

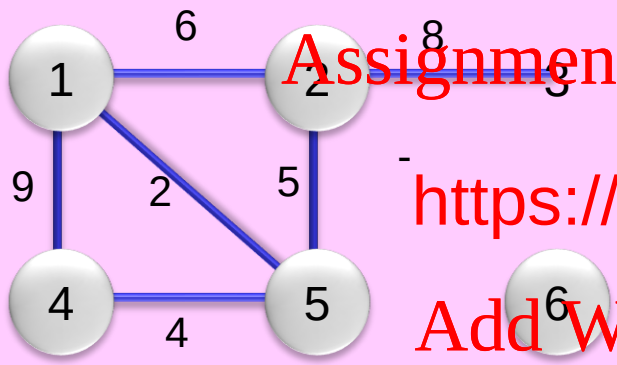


A =

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

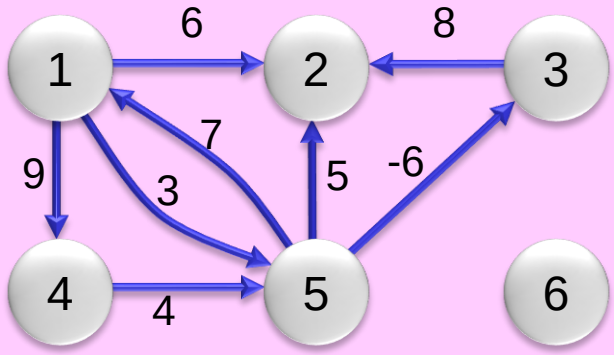
Weighted Adjacency Matrix

$$A[i, j] = \begin{cases} w(i, j) & \text{if } (i, j) \in E(G) \\ 0 & \text{if } i = j, (i, j) \notin E(G) \\ \infty & \text{otherwise} \end{cases}, \quad \text{for } i, j \in V(G).$$



A =

	1	2	3	4	5	6
1	0	6	∞	9	2	∞
2	6	0	8	∞	5	∞
3	∞	8	0	∞	-6	∞
4	∞	∞	∞	0	4	∞
5		5	-6	4	0	∞
6	∞	∞	∞	∞	∞	0

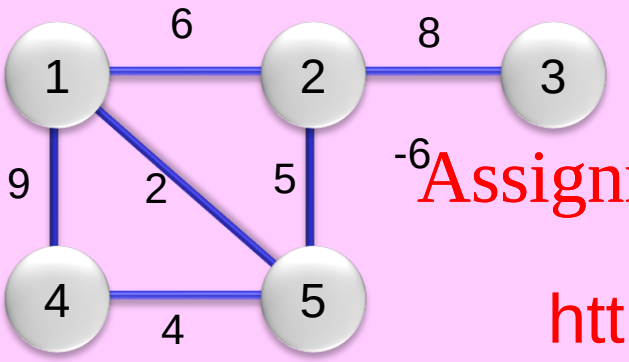


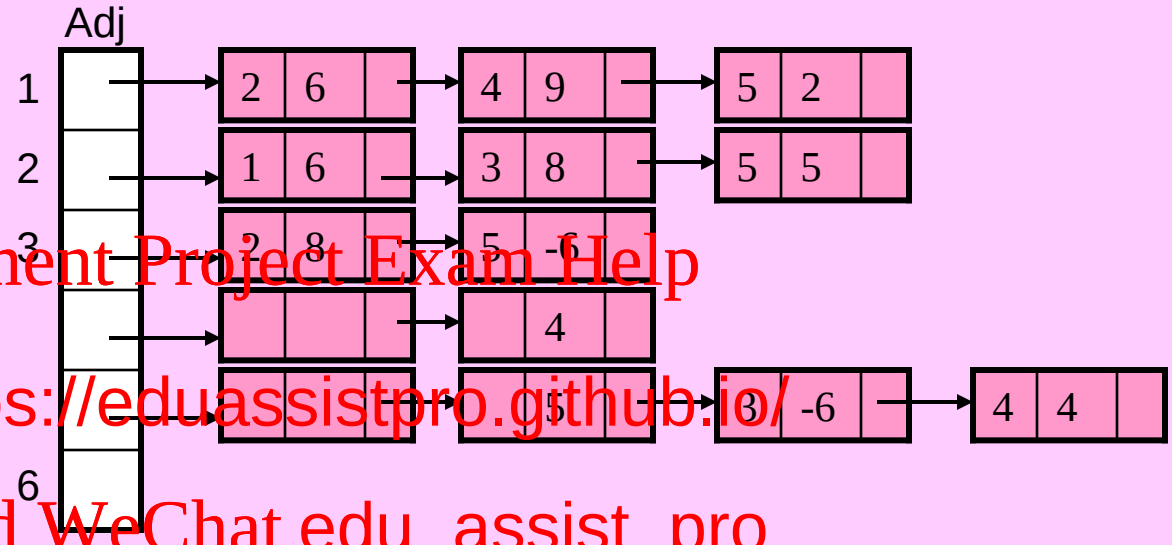
A =

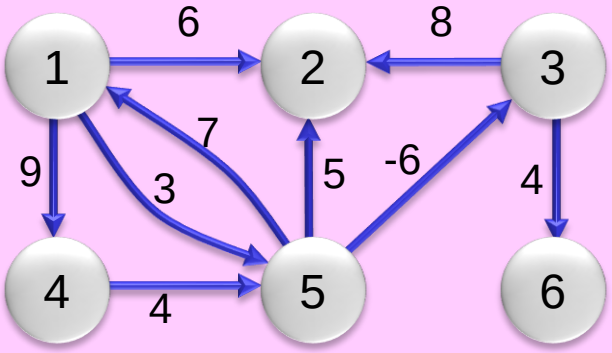
	1	2	3	4	5	6
1	0	6	∞	9	3	∞
2	∞	0	∞	∞	∞	∞
3	∞	8	0	∞	∞	∞
4	∞	∞	∞	0	4	∞
5	7	5	-6	∞	0	∞
6	∞	∞	∞	∞	∞	0

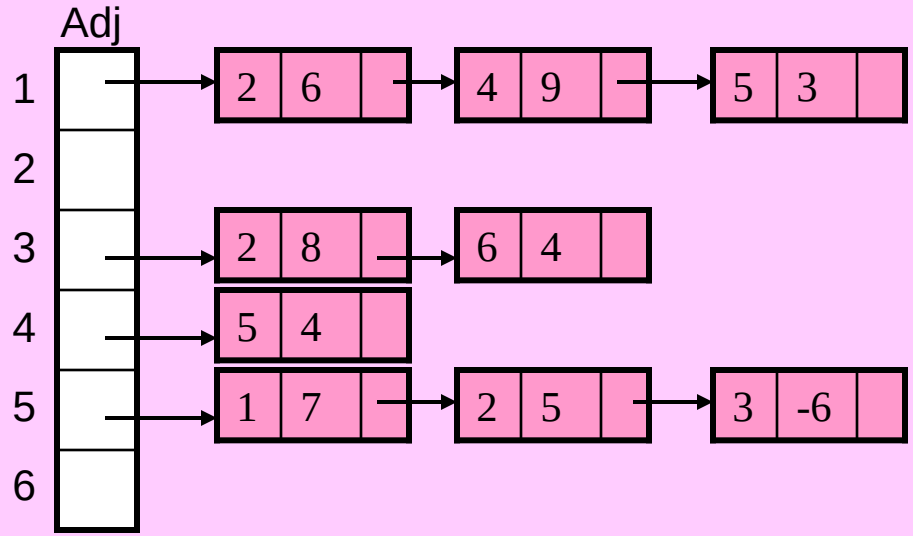
(Weighted) Adjacency List Structure

$$Adj[i] = \{ \langle j, w(i, j) \rangle \mid (i, j) \in E(G) \}, \quad \text{for } i \in V(G).$$





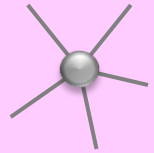




The Hand-Shaking Lemma

Vertex $v \in V(G)$: degree (or valance) , in-degree, out-degree

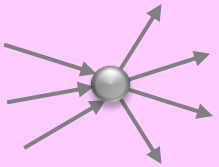
Undirected G: $\deg(v) = |\{ u \mid (v,u) \in E(G) \}| = |\mathrm{Adj}[v]|$



Digraph G: $\mathrm{outdeg}(v) = |\{ u \mid (v,u) \in E(G) \}| = |\mathrm{Adj}[v]|$

$\mathrm{indeg}(v) = |\{ u \mid (u,v) \in E(G) \}|$

$\deg(v) = \mathrm{outdeg}(v) + \mathrm{indeg}(v)$



The Hand-Shaking Lemma:

For any graph (directed or und ave:

$$\sum_{v \in V(G)} \deg(v) = 2 \mid E \mid .$$

For any directed graph we also have:

$$\sum_{v \in V(G)} \mathrm{indeg}(v) = \sum_{v \in V(G)} \mathrm{outdeg}(v) = \mid E \mid .$$

Adjacency Matrix vs Adjacency List Structure

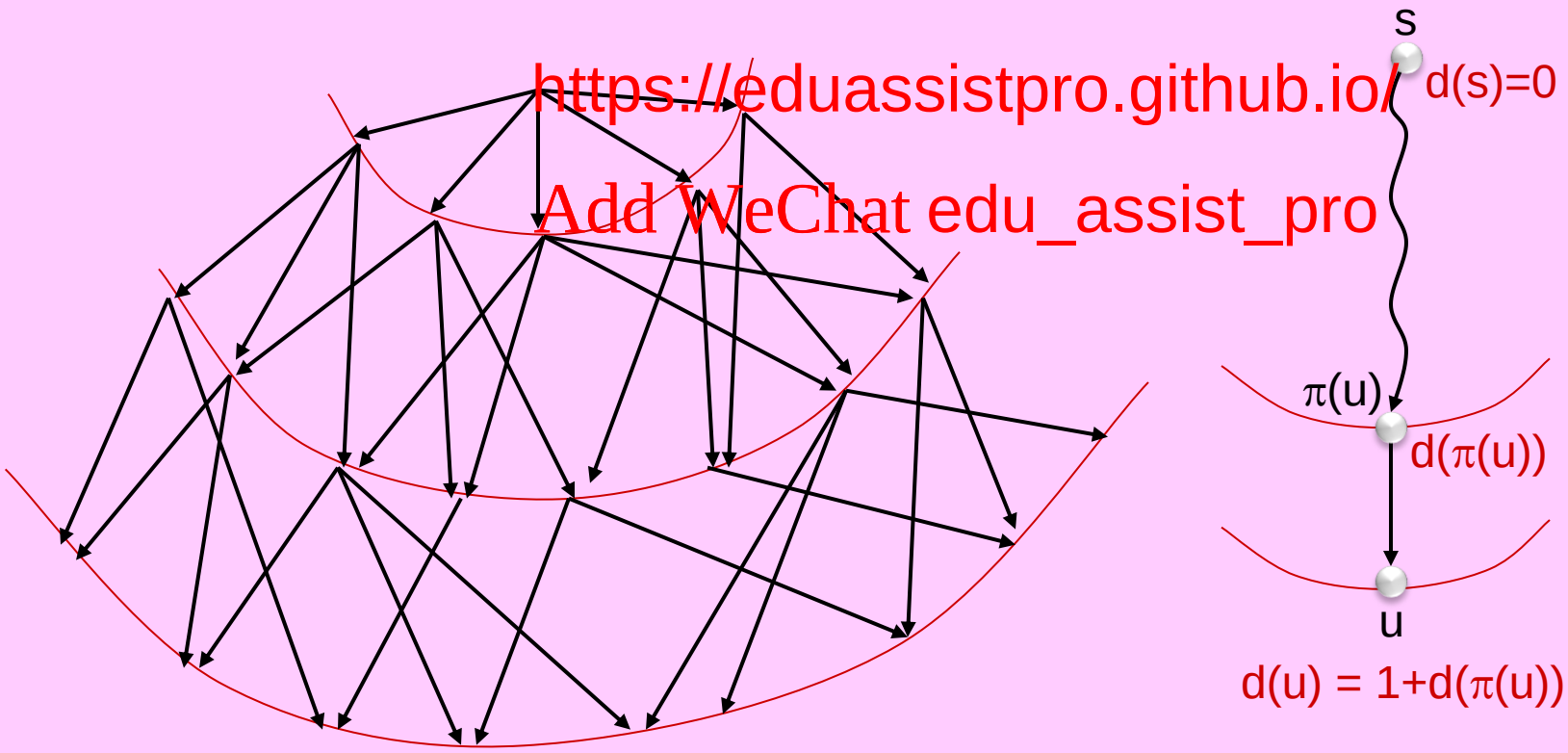
complexity		Adjacency Matrix	Adjacency List
# memory cells	Space	$O(V^2)$	O(V + E)
Initialize struct		$^2)$	O(V + E)
Scan (incident edges of) all vertices	Time	$O(V^2)$	O(V + E)
List vertices adjacent to u∈V(G)		O(V)	O(|Adj[u]|)
Is (u,v) ∈ E(G)?		O(1)	O(|Adj[u]|)

BREAD EARCH

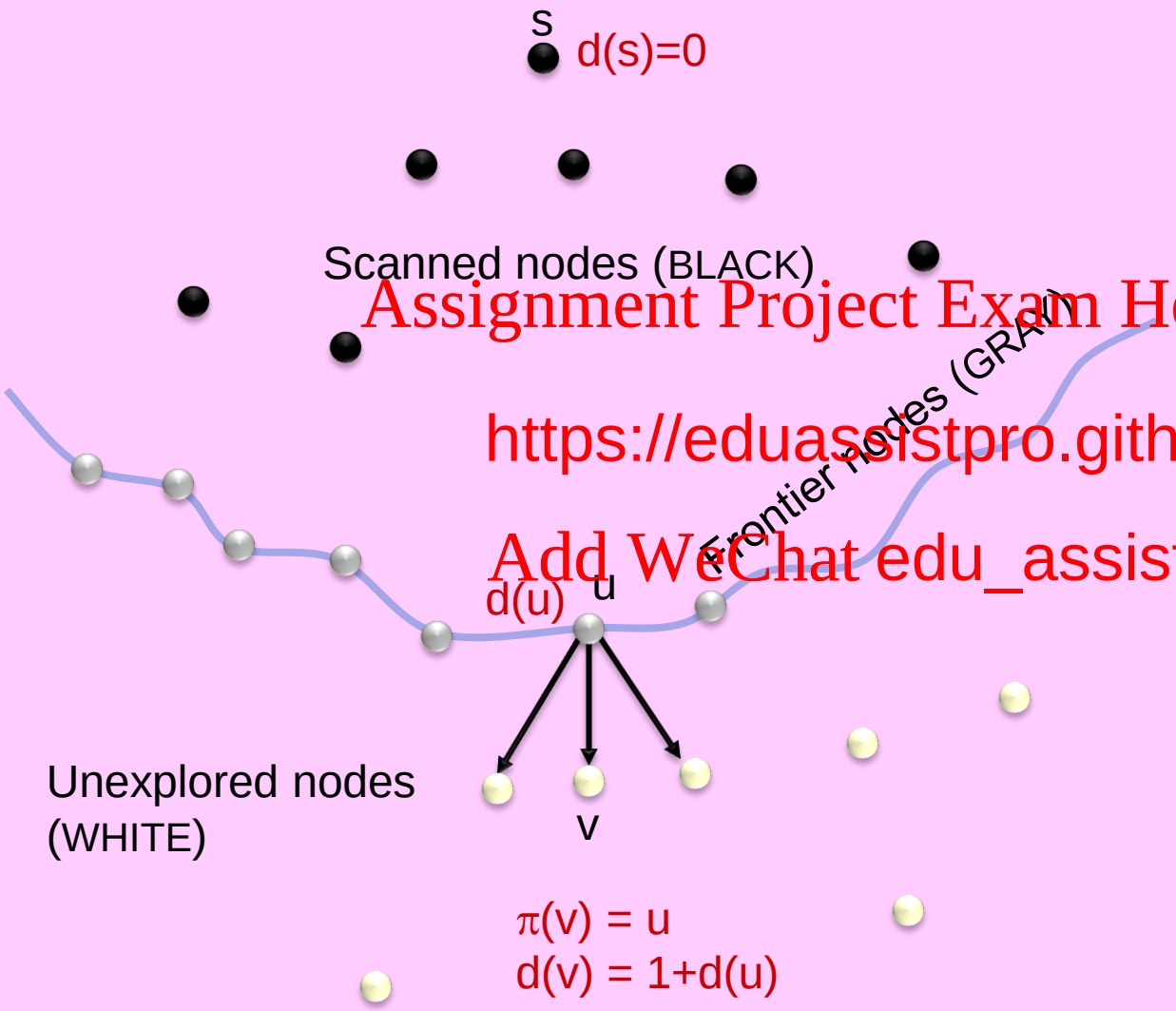
Generalizes level-order traversal on trees.

BFS Properties

- A simple linear-time graph search algorithm with many applications.
- Generalizes the level-order traversal on trees.
- G = (V,E) a given directed or undirected graph.
- Given a source $s \in V$, BFS explores all parts of G that are reachable from s.
- Discovers (visits) each vertex u in order of its un-weighted distance d(u) from s.
- This distance is the length of the un-weighted shortest path from s to u.



How BFS works



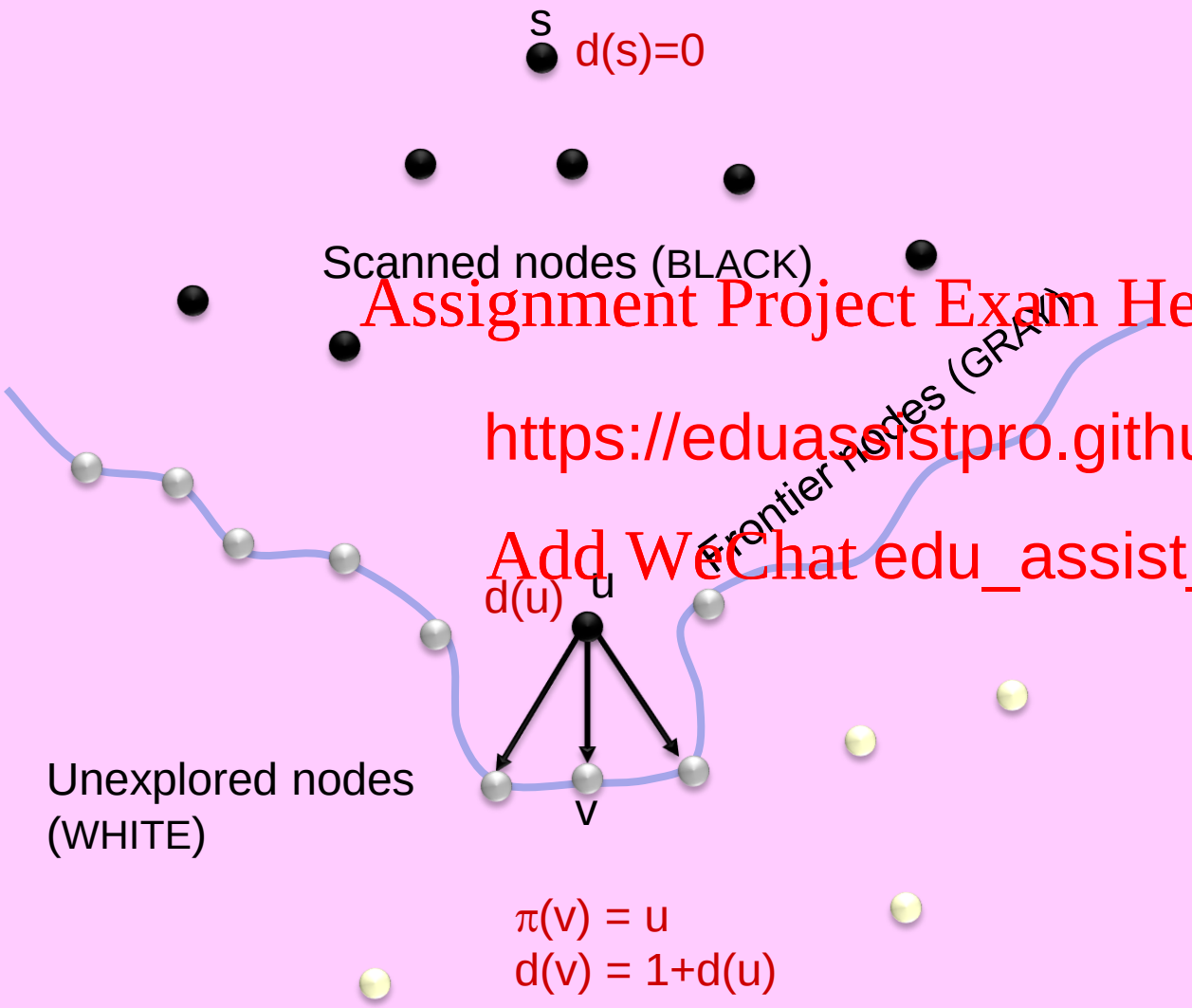
Greedy Choice:
Scan the oldest frontier node.

Q: How?
A: Maintain frontier nodes in a Queue Q (FIFO List).

Iteratively dequeue a vertex u & scan it:
visit every unexplored vertex v adjacent to u.

$\pi(v) = u$
$d(v) = 1+d(u)$

How BFS works



Greedy Choice:
Scan the oldest frontier node.

Q: How?
A: Maintain frontier nodes in a Queue Q (FIFO List).

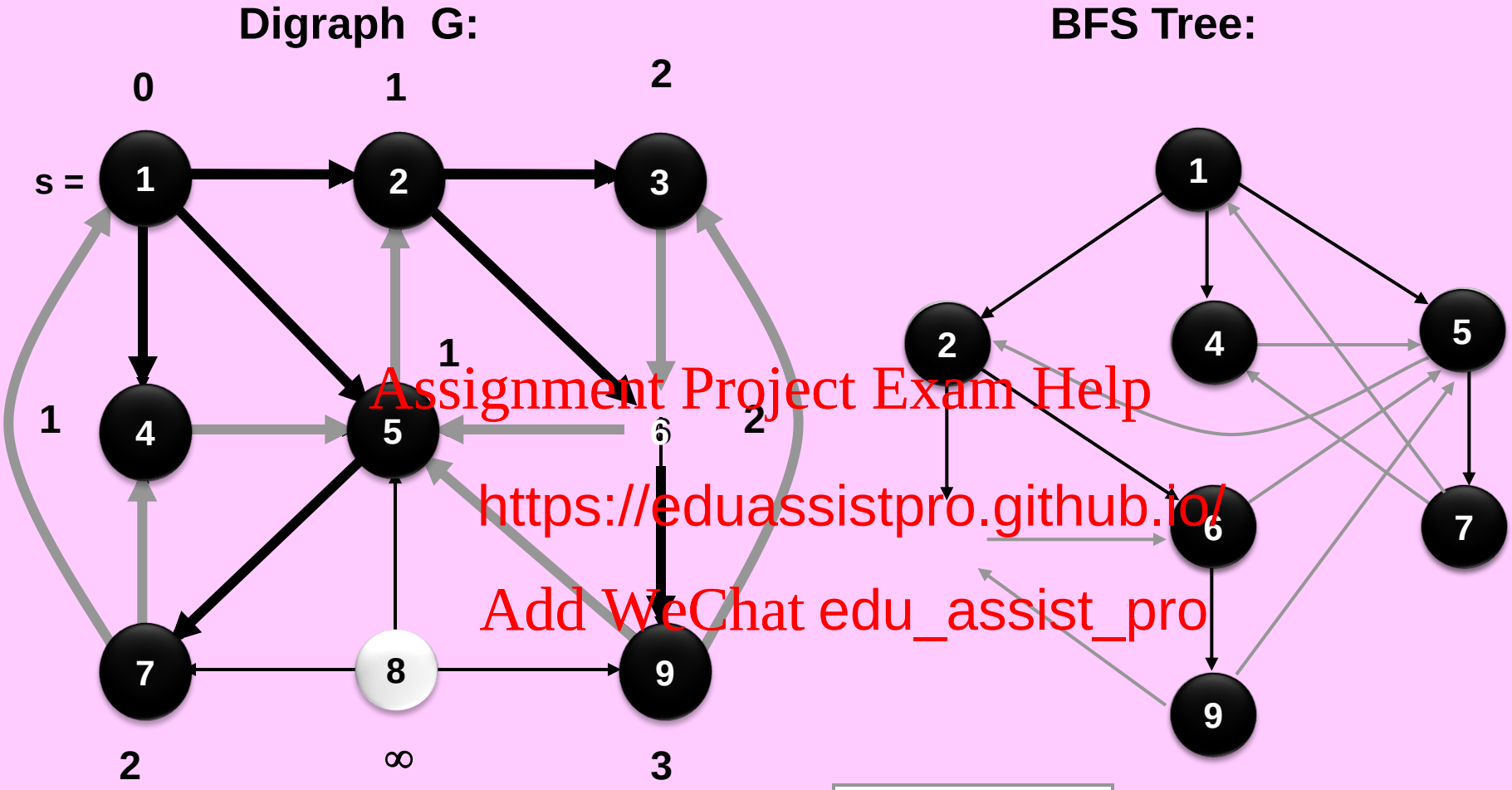
Iteratively dequeue a vertex u & scan it:
visit every unexplored vertex v adjacent to u.

Enqueue newly discovered nodes.

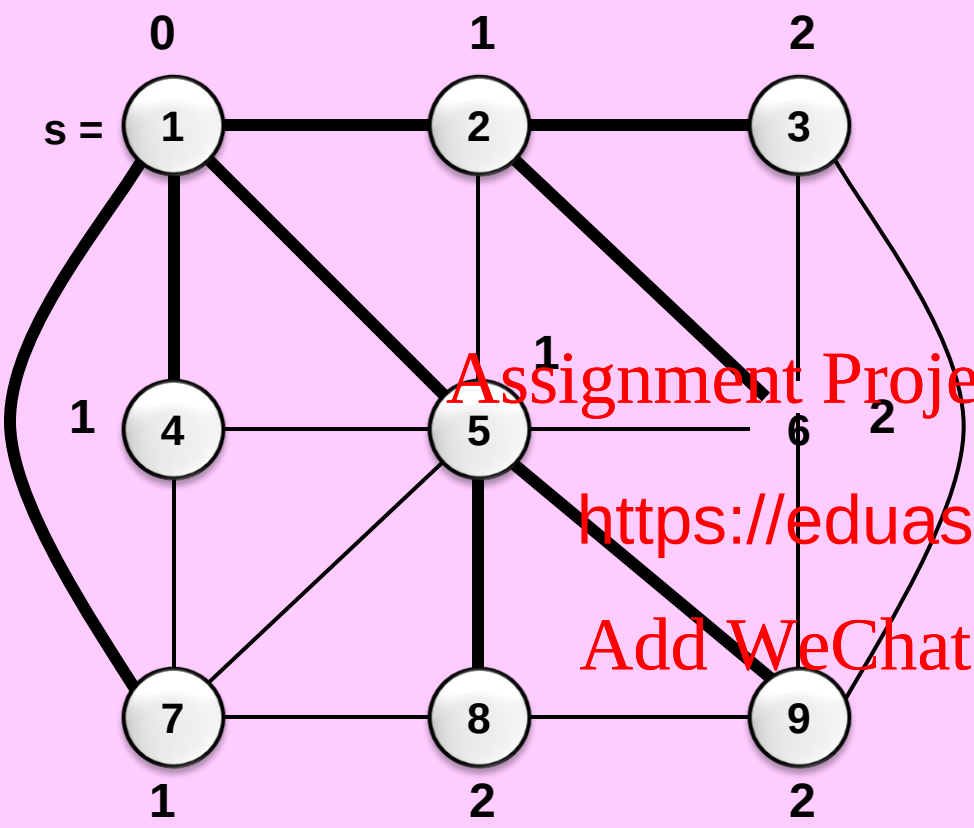
Frontier advances.

BFS Algorithm

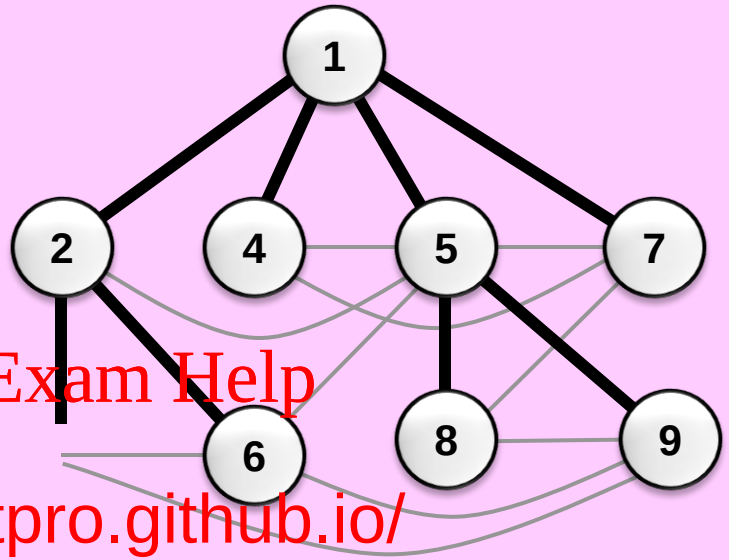
```
Algorithm BFS( G, s )
1.    for each vertex v ∈ V(G) do                    § initialize
          ⟨color[v], d[v], π[v]⟩ ← ⟨WHITE, ∞, nil⟩

2.    ⟨color[s], d[s] ⟩ ← ⟨GRAY, 0⟩                  § s is the first frontier vertex
3.    Q ← ∅;  Enqueue(s, Q)

4.    while Q ≠ ∅ do                                 e frontier
5.        u ← Dequeue(Q)                             iest visited frontier node
6.        for each vertex v ∈ Adj[u] do              plore every edge (u,v)
7.            if color[v] = WHITE then do            plored adjacent node v
8.                ⟨color[v], d[v], π[v]⟩ ← ⟨GRAY, 1+d[u], u⟩
9.                Enqueue(v, Q)                      § v is a new frontier node
10.           end-if
11.       end-for
12.       color[u] ← BLACK                           § u is scanned
13.   end-while
end
```



Undirected graph G:



BFS Tree:



BFS Running Time

- The only way that a node becomes gray & hence enters Q is when it was white (lines 2-3-4, 8-9-10).
- So, each node enters Q at most once (& only if it was reachable from s).
- Thus, a node u is remove at most once (line 6).
- So, a node u is scanned at most once & becomes black (lines 7-13). This takes O(1+|Adj[u]|) time.
- Therefore, the total time is:

$$O\left(V + \sum_{u \in V(G)} (1 + |Adj[u]|)\right) = O\left(V + \sum_{u \in V(G)} 1 + \sum_{u \in V(G)} |Adj[u]|\right) = O(V + E).$$

Algorithm BFS(G, s)

1. **for** each vertex v ∈ V(G) **do**
2. ⟨color[v], d[v], π[v]⟩ ← ⟨WHITE, ∞, nil⟩
3. ⟨color[s], d[s] ⟩ ← ⟨GRAY, 0⟩
4. Q ← ∅; Enqueue(s, Q)
5. **while** Q ≠ ∅ **do**
6. u ← Dequeue(Q)
7. **for** each vertex v ∈ Adj[u] **do**
8. = WHITE **then do**
9.], d[v], π[v]⟩ ← ⟨GRAY, 1+d[u], u⟩
10. Q)
11.
12.
13. color[u] ← BLACK
14. **end-while**

end

More BFS Properties

FACT 2:

For each edge **(u,v)** $\in$ E(G) we have $d[v] \leq 1 + d[u]$.
Furthermore, if G is undirected, then $|\, d[u] - d[v] \,| \leq 1$.

Proof sketch: By the time u becomes black (is scanned), v must have been visited (if not earlier) ...
If G is undirected, then (u,v) = (v,u).

FACT 3: BFS Edge T

Each traversed edge **(** $\in$ r a non-tree edge
with respect to the BFS T there are 3 types of non-tree edges:

1. **BFS tree-edge:** $u = \pi[v]$

2. **Back-edge:** v is a BFS ancestor of u (possibly u itself)

3. **Forward-edge:** v is a BFS proper descendant of u (not a tree edge).
   This case will never happen by FACT 2!

4. **Cross-edge:** u & v are not ancestor-descendant of each other wrt BFS tree.

BFS Tree = Shortest Path Tree rooted at source

FACT 4: For each node v in the BFS tree,
the tree path from root s to v is the un-weighted shortest path from s to v in G.
(Nodes that are not in the BFS tree are not reachable from s.)
More specifically, we have:

$\pi[v]$ = parent of v in the BFS tree
= predecessor of v on its shortest path from s
(nil, if v is not reachable from s),

d[v] = length of short
(∞ if v is not reachable from s).

Algorithm PrintPath(v)
if v = nil **then return**
PrintPath(π[v])
print v

Proof: Let v be a node reachable from s in G. Then, $d[v] = \begin{cases} 0 & \text{if } v = s \\ 1 + d[\pi[v]] & \text{if } v \neq s \end{cases}$

But that is the solution to the shortest path equations:

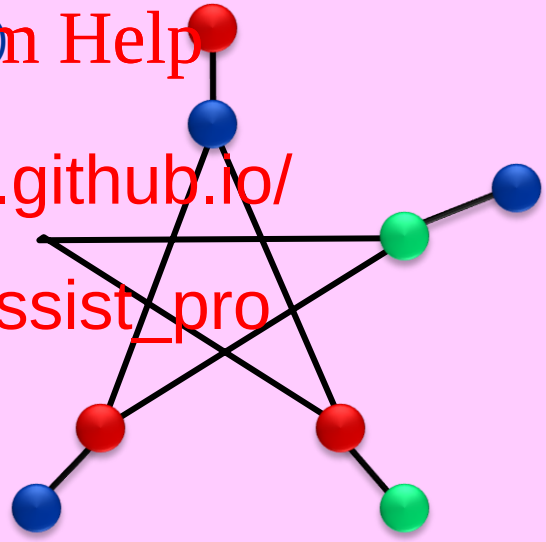
$$d[v] = \begin{cases} 0 & \text{if } v = s \\ \min_u \{1 + d[u] \mid (u, v) \in E(G)\} & \text{if } v \neq s \end{cases}$$

because, by Fact 2: $\forall(u,v) \in E(G)$, we have $d[v] \leq 1+d[u]$.
(The state of nodes not reachable from s is obvious.)

Graph Coloring

G = an undirected graph.

Graph Coloring:
Let k be a positive integer.
G is k-colorable if each node of G can be colored by one of at most k colors such that for each edge (u,v) $\in$E(G), color(u) $\neq$ color(v).
(This problem has lots of many applications.)



A 3-coloring of the Peterson graph
(It is not 2-colorable.)

1-colorable $\Leftrightarrow$ E(G) = $\varnothing$
2-colorable: also easy to detect (see next slides)
3-colorable: is hard to detect (see NP-completeness)

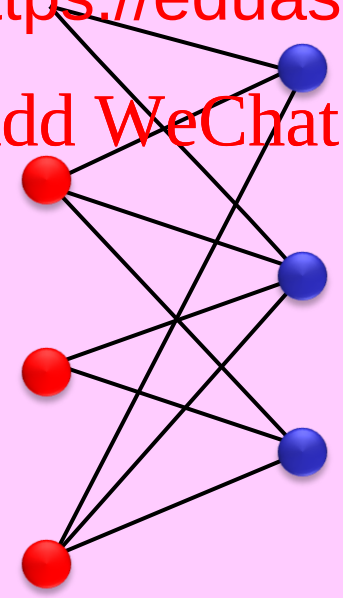
K_n needs n colors.
All planar graphs are 4-colorable.

Bipartite Graphs

G = an undirected graph.

Bipartiteness:
G is bipartite if V(G) can be partitioned into two disjoint subsets X and Y such that for each edge $(u,v) \in E(G)$, either $u \in X$ & $v \in Y$, or $v \in X$ & $u \in Y$.

X Y



FACT 5: Let G be an undirected graph. The following statements are equivalent:

1. G is bipartite,
2. G is 2-colorable,
3. G has no odd-length cycle,
4. In BFS of G, for every edge $(u,v) \in E(G)$, we have $|\, d[u] - d[v] \,| = 1$.

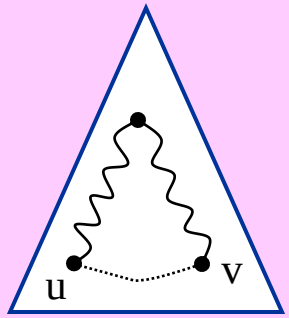
These can be tested in O(V+E) time (constructive proof).

Proof:

[(1) ⇒ (2)]: Let (X,Y) be a bipartition of V(G). Color nodes in X with color 1, and nodes in Y with color 2.

[(2) ⇒ (3)]: If G is 2-colora e alternate colors. Hence, the cycle has even length.

[(3) ⇒ (4)]: By Fact 2 we know that for each edge (u,v), $|d[u] - d[v]| \le 1$. If d[u] = d[v] (i.e., they are at the same BFS tree level), then edge (u,v) plus the tree paths from each of u and v to their lowest common ancestor form an odd-length cycle. A contradiction.



[(4) ⇒ (1)]: Do BFS of each connected component of G: Set $V_1 = \{ u \in V(G) \mid d[u] \text{ is even} \}$, $V_2 = \{ u \in V(G) \mid d[u] \text{ is odd} \}$. (V_1, V_2) is a valid bipartition of G.

DEPT ARCH

Generalizes preorder and postorder traversal on trees.

DFS Properties

- Another simple linear-time graph search algorithm with many applications.
- Generalizes the pre-order and post-order traversals on trees.
- G = (V,E) a given directed or undirected graph.

- DFS creates a forest of node-disjoint trees with non-tree edges between nodes.
  Pick an unvisited node as the root of the next DFS tree and advance the search.

- Greedy Choice:
  within the same DFS tr most recent visited node.

- Recursive DFSvisit(u):
  explores nodes in the sub-tree rooted at u
  (the nodes reachable from u that are still unexplored at the time u gets visited).

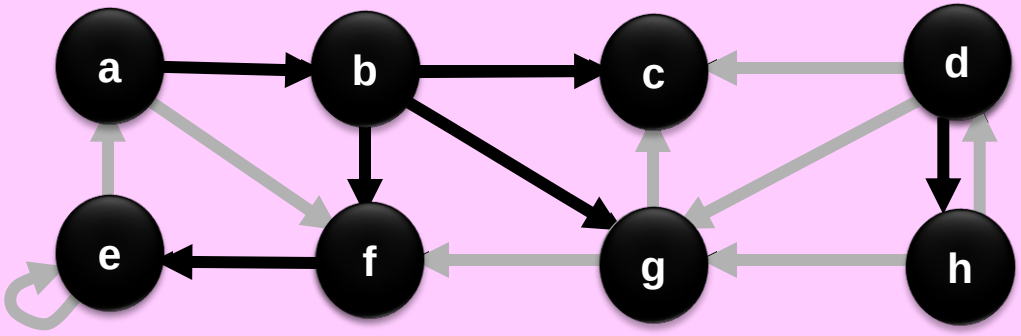
- DFS(u) recursive call is active $\Leftrightarrow$ u is an ancestor of the most recent visited node.

- [d[u], f[u]] = DFS time stamp for node u:
  d[u] = start time of DFSvisit(u),
  f[u] = finishing time of DFSvisit(u).

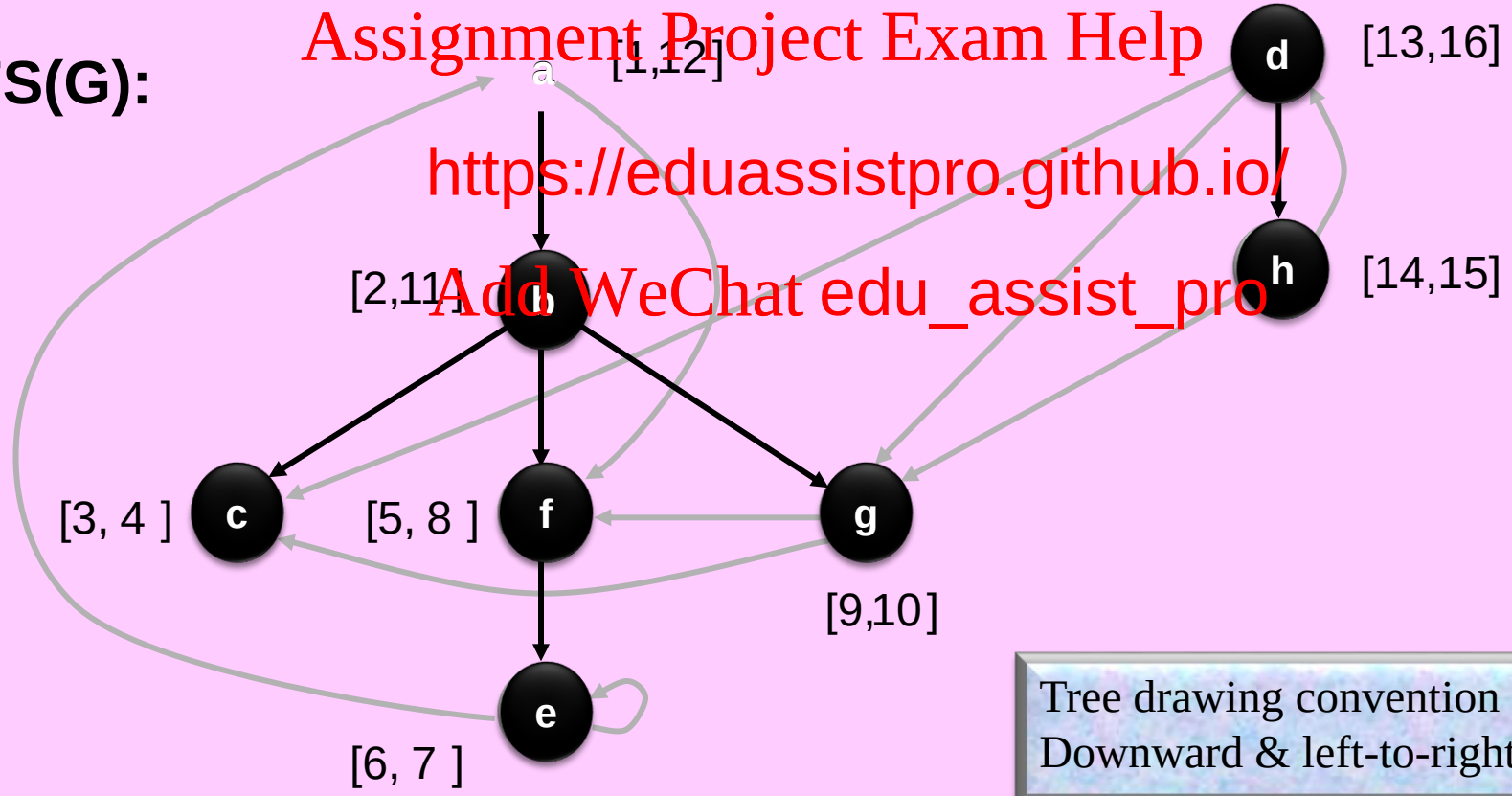
DFS Algorithm

```
Algorithm DFS(G)
1.   time ← 0
2.   for each vertex u ∈ V(G) do ⟨color[u], π[u]⟩ ← ⟨WHITE, nil⟩
3.   for each vertex u ∈ V(G) do
         if color[u] = WHITE then DFSvisit(u)      § u is root of next DFS tree
end

Procedure DFSvisit(u)
4.   d[u] ← time ← time                          § discovery time
5.   color[u] ← GRAY                             §
6.   for each vertex v ∈ Adj[u] do                §          d adjacent node v
7.       if color[v] = WHITE then do              §
8.           π[v] ← u                            § (u,v) is a DFS tree-edge
9.           DFSvisit(v)                         § v is most recent visited node
10.      end-if
11.  end-for
12.  color[u] ← BLACK                            § u is scanned
13.  f[u] ← time ← time + 1                      § DFSvisit(u) finishing time
end
```

G:

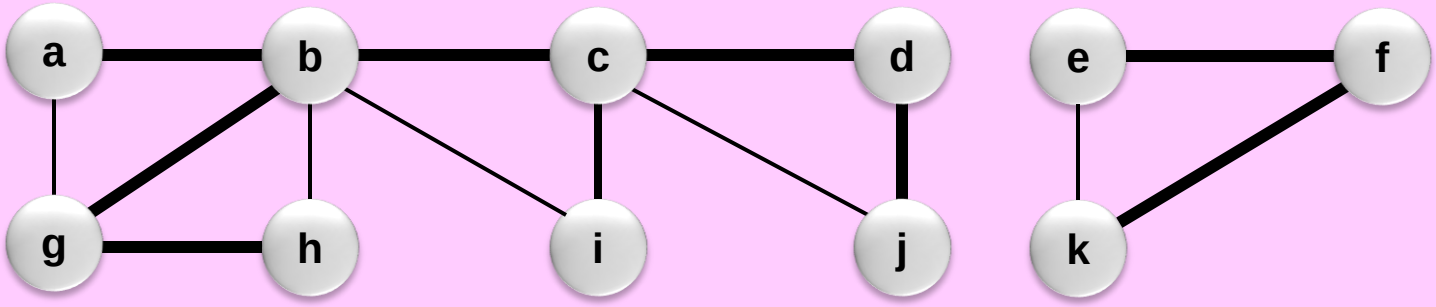


DFS(G):

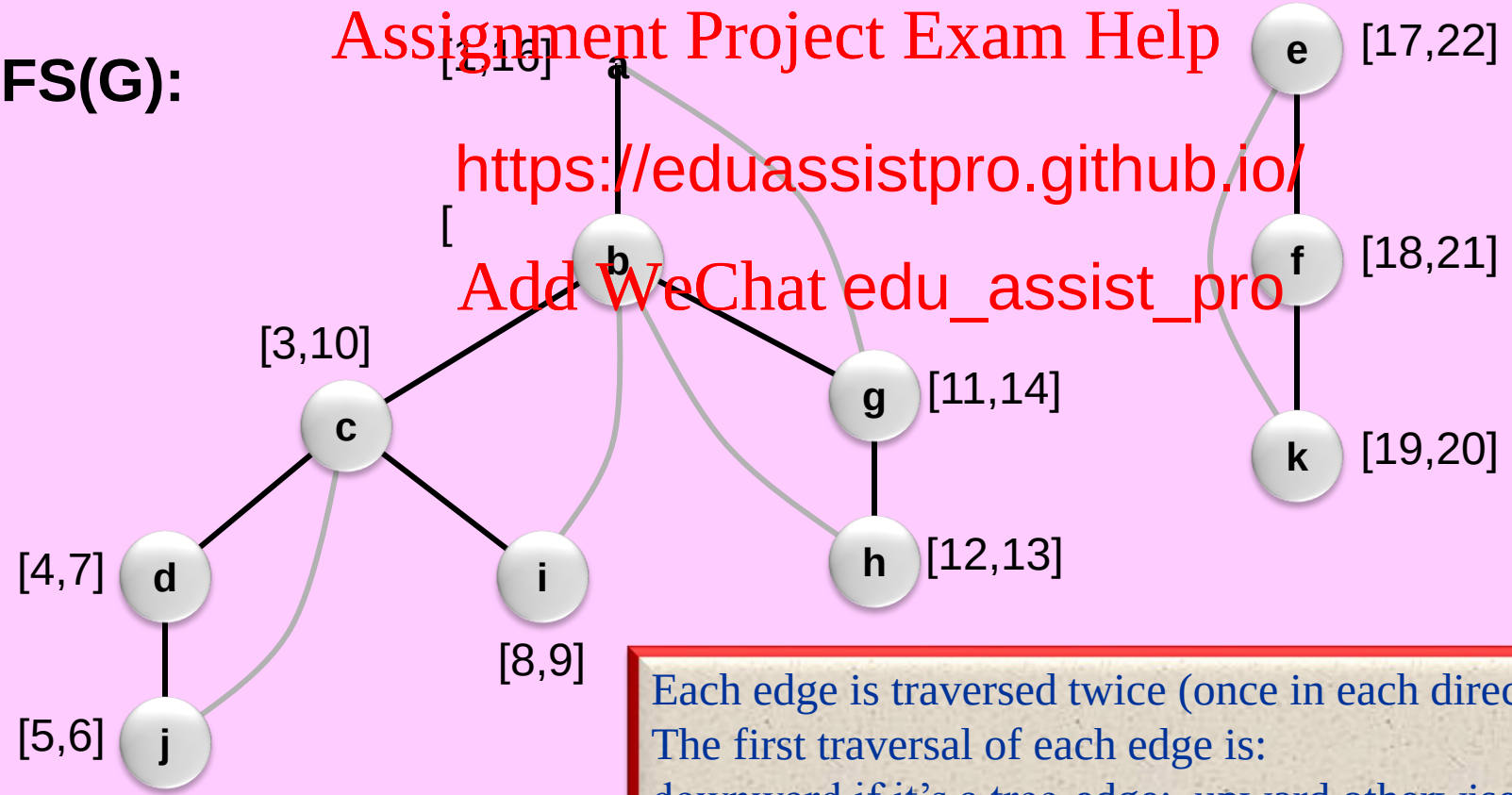


Tree drawing convention
Downward & left-to-right

G:



DFS(G):

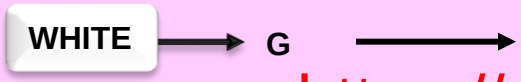


Each edge is traversed twice (once in each direction).
The first traversal of each edge is:
downward if it's a tree-edge; upward otherwise.

DFS Running Time

```
Algorithm DFS(G)
1.  time ← 0
2.  for each vertex u∈V(G) do ⟨color[u], π[u]⟩ ← ⟨WHITE, nil⟩
3.  for each vertex u∈V(G) do
        if color[u] = WHITE then DFSvisit(u)
end
```

```
Procedure DFSvisit(u)
4.  d[u] ← time ← time + 1
5.  color[u] ← GRAY
6.  for each vertex v∈Adj[u] do
7.      if color[v] = WHITE then do
8.          π[v] ← u
9.          DFSvisit(v)
10.     end-if
11. end-for
12. color[u] ← BLACK
13. f[u] ← time ← time + 1
```

- DFSvisit(u) is called exactly once for eac
- Excluding recursive calls, DFSvisit(u) takes O(1+|Adj[u]|) time.
- Therefore, the total time is:

$$O\left(V + \sum_{u \in V(G)} (1 + |Adj[u]|)\right) = O\left(V + \sum_{u \in V(G)} 1 + \sum_{u \in V(G)} |Adj[u]|\right) = O(V + E).$$

The Parenthesis Theorem (PT)

The Parenthesis Theorem (PT):

1. For all nodes u: d[u] < f[u].
   [d[u], f[u]] is activation time interval of DFSvisit(u).

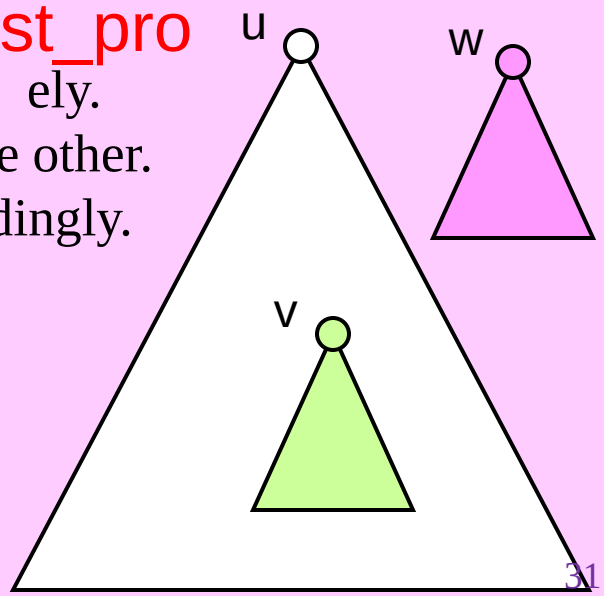
2. For every pair of nodes u & v, their time intervals [d[u],f[u]] and [d[v],f[v]] are either disjoint or one encloses the other. (no partial overlap).

3. u is a DFS ancestor o
   [d[u],f[u]] encloses [f[v] ≤ f[u].

Proof:

Consider DFS sub-trees T_u and T_v rooted at u ely.
T_u and T_v are either completely disjoint, or one includes the other.
Their DFSvisit activation time intervals will behave accordingly.

In the illustrative figure to the right,
[d[u],f[u]] encloses [d[v],f[v]],
but is disjoint from [d[w],f[w]].



The White-Path Theorem (WPT)

The White-Path Theorem (WPT):

Node u is a DFS ancestor of node v if and only if
when u is discovered, there is a path of all white nodes from u to v.

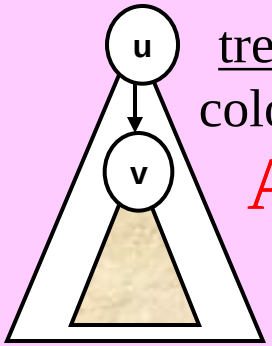
Proof [of ⇒]:
The nodes on the DFS tree path from u to v are all white at the activation time of DFSvisit(u).

Proof [of ⇐]:
P = an all white-node path friled.
Claim: v will be visited before DFSvisit(u) is fini become a descendant of u.
Proof of Claim: by induction on length of P:

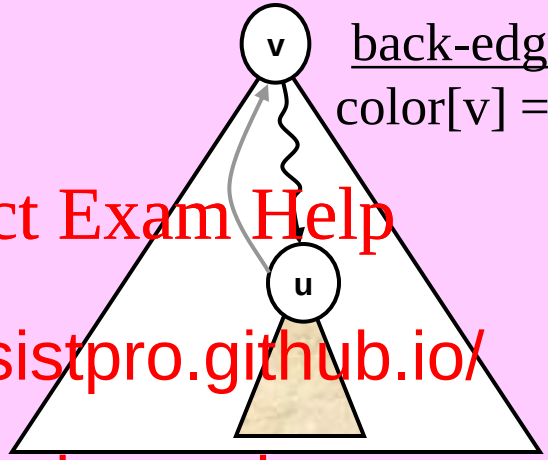
Basis (|P| = 0): Obvious; u=v.
Ind. Step (|P| > 0): Let w be successor of u on P.
During DFSvisit(u), when w∈Adj[u] is considered, if w is still white, it will get visited & becomes a descendant of u.
So, some node on P, other that u, becomes a descendant of u.
Consider the last node x of P that becomes a descendant of u.
The sub-path P' of P from x to v is strictly shorter than P and is all white when x is discovered.
So, by the induction hypothesis, v becomes a descendant of x, which is a descendant of u.
[Note: P is not necessarily the DFS tree path from u to v! Where is that in the proof?]

DFS Edge Classification of Digraphs

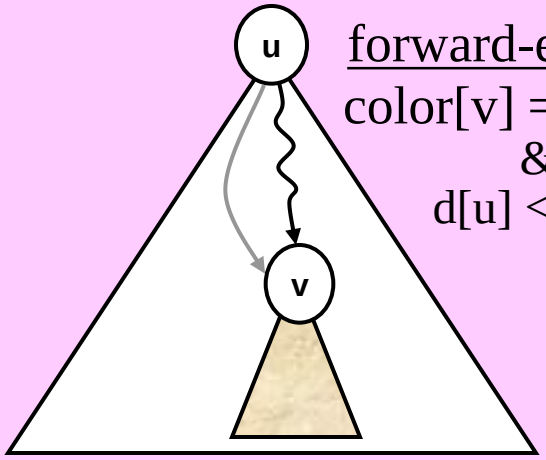
At the time edge $u \to v$ is traversed, we have the following possibilities:



tree-edge:
color[v] = WHITE

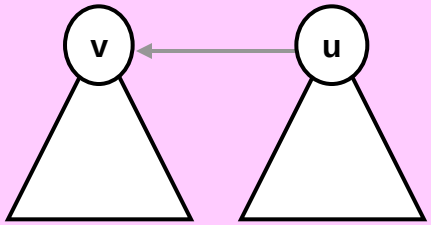


back-edge:
color[v] = GRAY

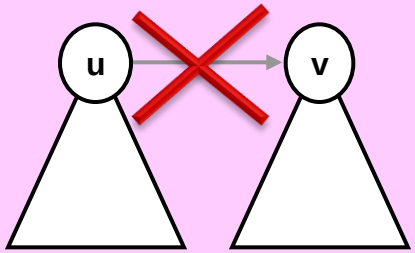


forward-edge:
color[v] = BLACK
&
d[u] < d[v]

onvention
Downward & left-to-right



cross-edge:
color[v] = BLACK
&
d[v] < d[u]

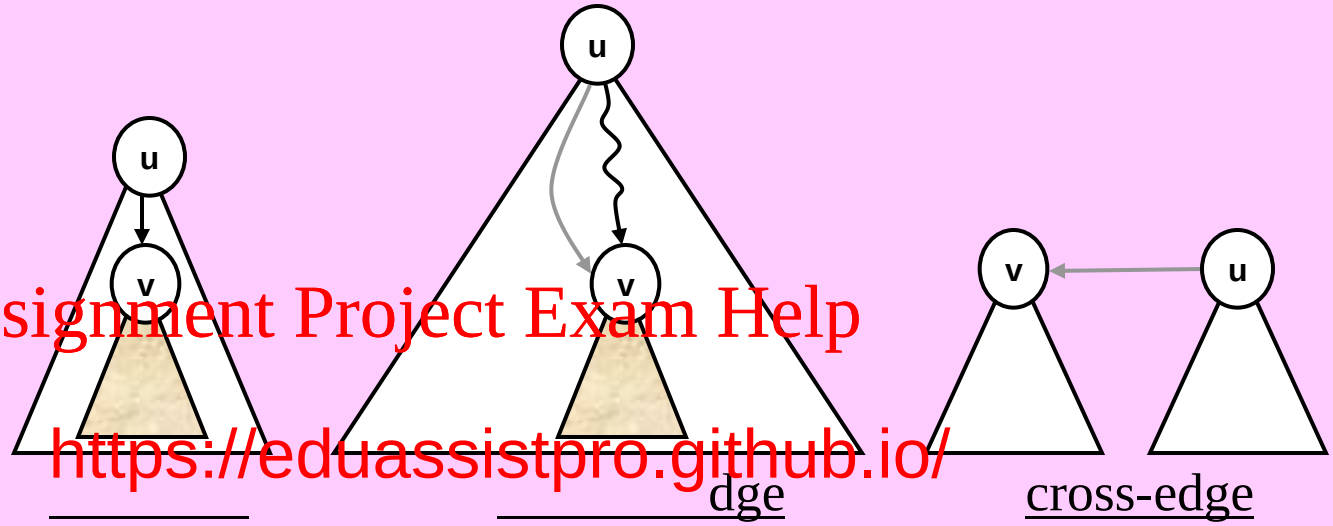


cross-edge:
color[v] = WHITE
Impossible by WPT!

DFS Finishing Times

DiGraph: state of edge u→v at the completion of DFS:

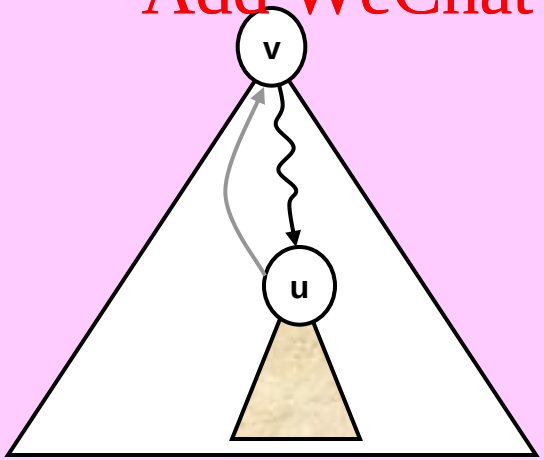
f[u] > f[v]:



dge

cross-edge

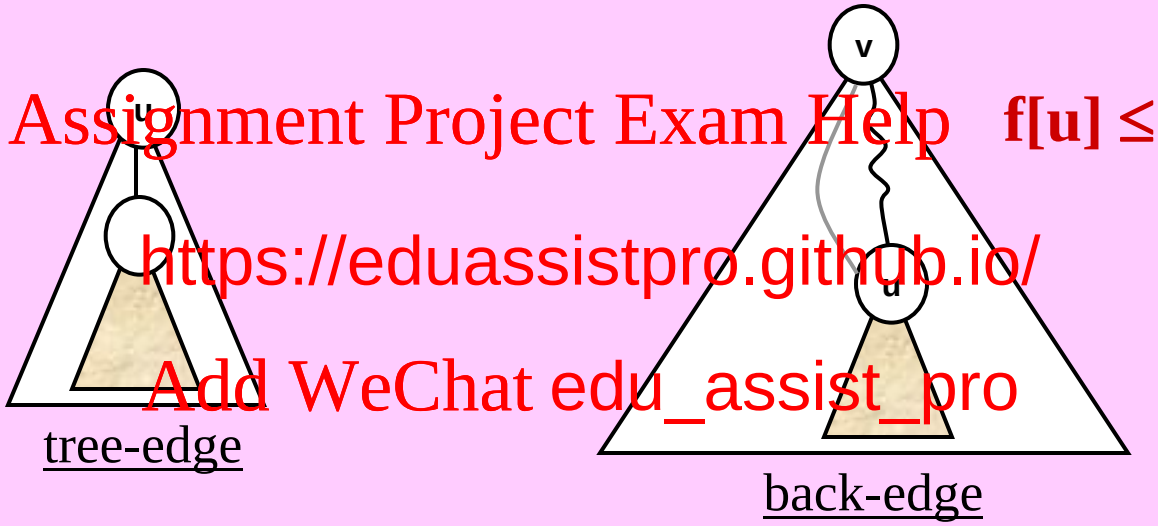
f[u] ≤ f[v]:



back-edge

DFS Edge Classification of Undirected Graphs

Undirected Graph: only 2 types of edges:
tree-edges &
back-edges.



Cross edge: impossible by the White Path Theorem.

So each edge is between an ancestor-descendant pair.
Such non-tree edges are called back-edges by convention
(the first time they are traversed from descendant towards ancestor).

The Cycle Theorem

The Cycle Theorem: [G = a directed or undirected graph]

G has a cycle if and only if DFS(G) has a back-edge.

Proof [of ⇐]:
A back-edge from u to v followed by the DFS tree path from v to u forms a cycle.

Proof [of ⇒]:
Suppose there is a cycle C
Among the nodes on C, let u be the one with .
Let v be the successor of u on C.
Then f[u] ≤ f[v] by the choice of u.
So, edge (u,v) is a back-edge.

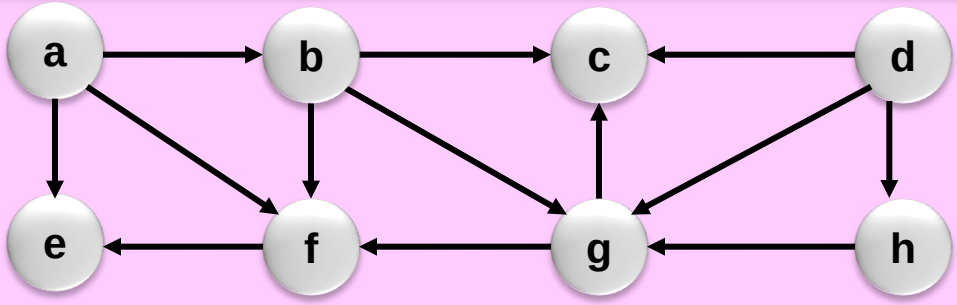
For undirected graphs there is a simpler argument:
without back-edges, we are left with a forest of trees. Trees are acyclic.

TOPO SORT

Linearly Order a Partially Ordered Set

Directed Acyclic Graph (DAG)

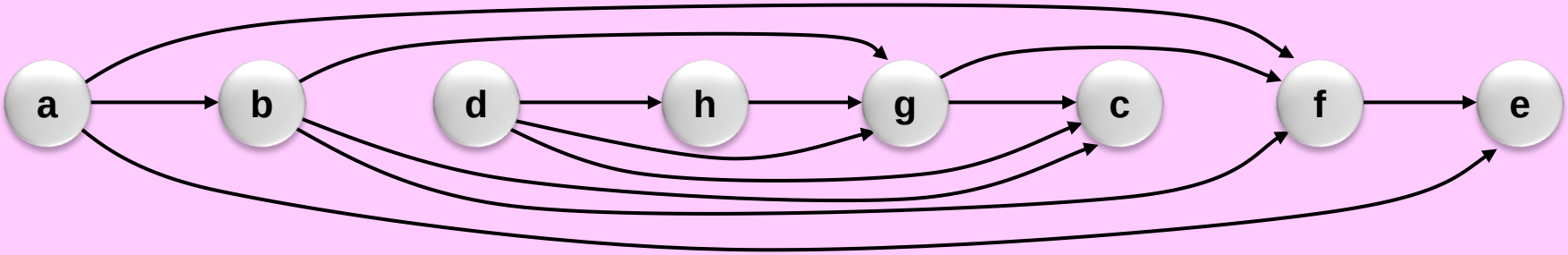
DAG G:



- DAG = digraph with no directed cycles.
- Modeling applications:
  - generaliza
  - reachabilit order on nodes (reflexive,
  - tasks with precedence constr
  - job sequencing
- Topological ordering of a DAG G:

  A linear ordering of V(G) such that

  $\forall(u,v)\in E(G)$, u appears before v in the linear order.

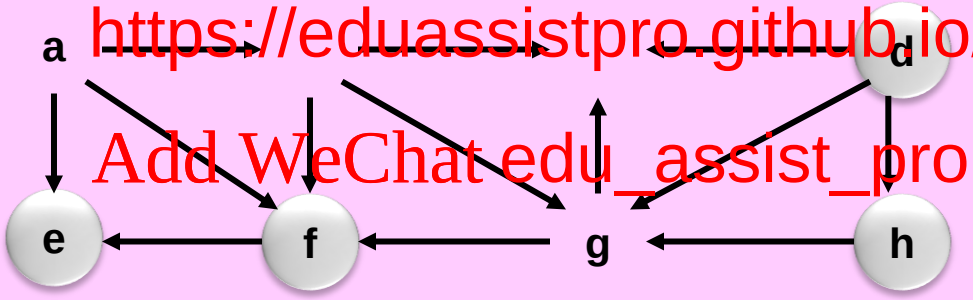


Topological Sort Algorithm 1

FACT 1:
A DAG G has at least one vertex with in-degree 0 (& one vertex with out-degree 0).

Proof:
G has no cycles $\Rightarrow$ all paths in G have finite length (no node can repeat on a path).
Let P be a longest path in G.
Suppose P starts at node u and ends in node v.
indeg(u) = outdeg(v) = 0 a longer path).

Example DAG:



indeg(a) = 0
indeg(d) = 0
outdeg(e) = 0

TOPOLOGICAL SORT ALGORITHM 1:

- Find a node u with in-degree 0.
- Remove u and all its out-edges from G. A smaller DAG remains.
- Add u to the end of the linear order.
- Iterate this process until there is no more node remaining in G.

EXERCISE: Give an O(V+E)-time implementation of this algorithm.

Topological Sort Algorithm 2

FACT 2: Reverse DFS Post-Order, i.e., reverse DFS finishing time is a topological order on a DAG.

Proof:

For any edge $u \rightarrow v$ in digraph G:

$f[u] \leq f[v] \Leftrightarrow u \rightarrow v$ is a back edge.

DAG G has no cycles,

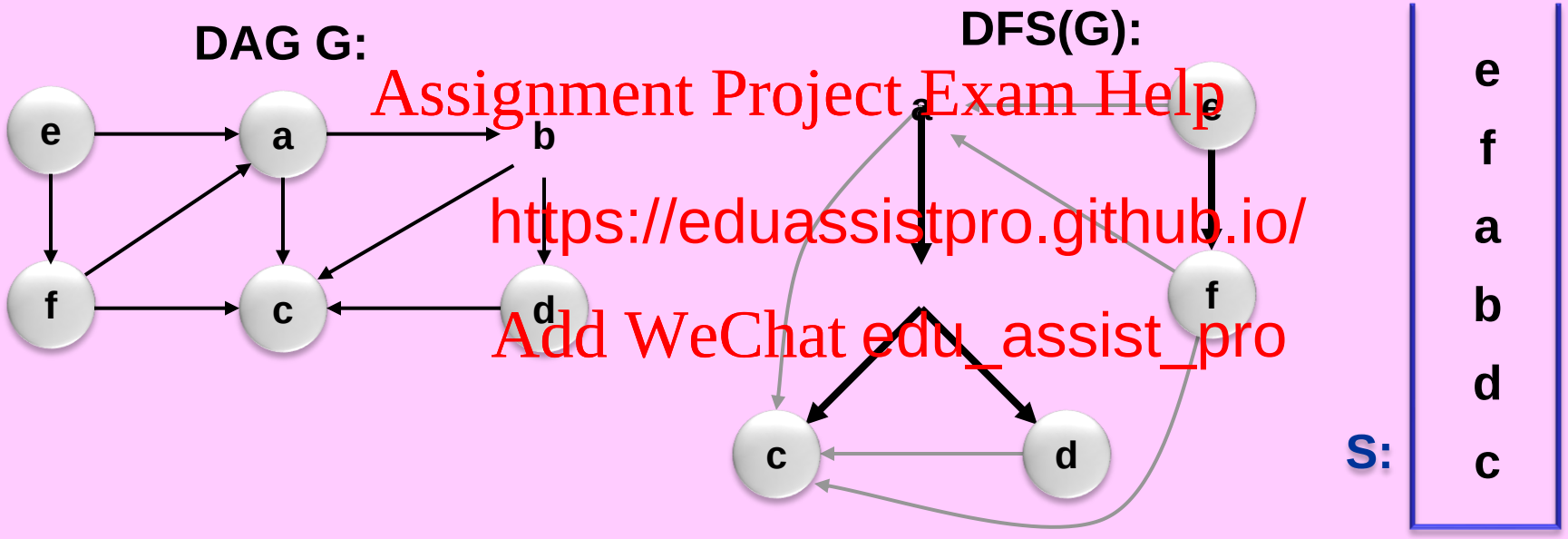
So, for every edge $u \rightarrow v$ in DAG G: $f[u] >$

Linearly order nodes in decreasing order of f[.].

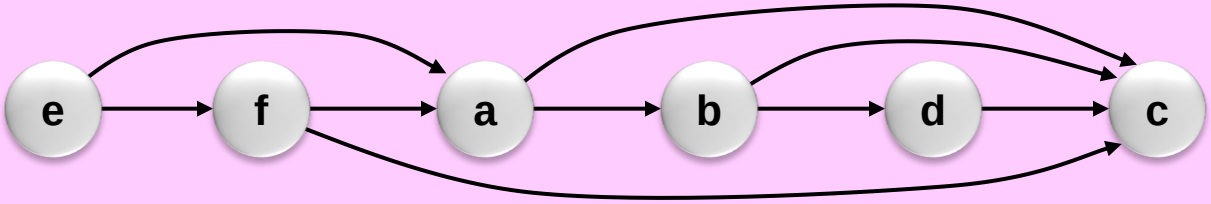
See implementation & example on the next page.

Topological Sort Algorithm 2

In O(V + E) time do a DFS of G with the following modification:
Push nodes into an initially empty stack S when they become black
(i.e., an added line at the end of DFSvisit).
At the end of DFS(G): repeatedly pop(S); nodes come out in topological order.



A topological ordering of G:



Graph Connectivity

- **How well is a graph connected?**
  - Connectivity issues arise in communication & transportation networks.
  - Identify bottlenecks in case parts of the network break down or malfunction.
  - Fault tolerance in distributed networks.

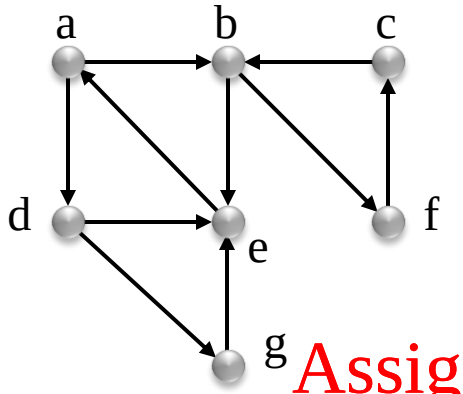
- **Digraph Connectivity:**

  - Semi-connectivity:
    can every pair of nodes have at least unication?

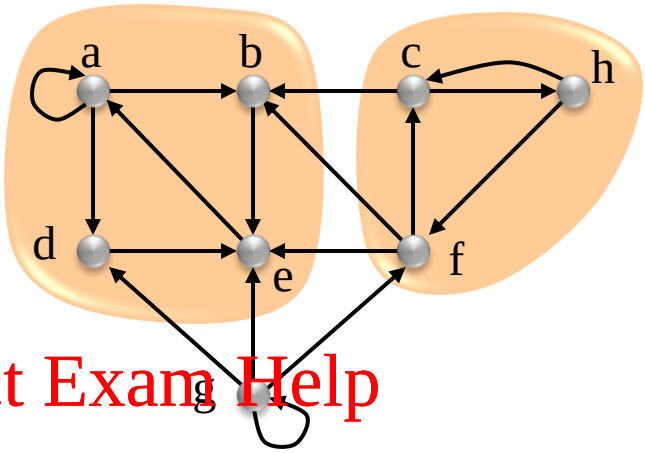
  - Strong connectivity (discussed next)
    can every pair of nodes have two-way communication?

- **Un-directed graph Connectivity:**
  - Edge/Node connectivity (discussed in the following section)
    How many edges/nodes need to be removed to disconnect the graph?

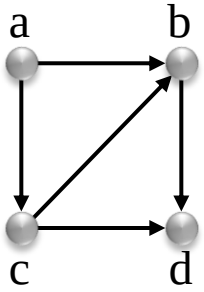
Digraph Strong Connectivity



Strongly Connecte
There is a directed p
from any node to any oth

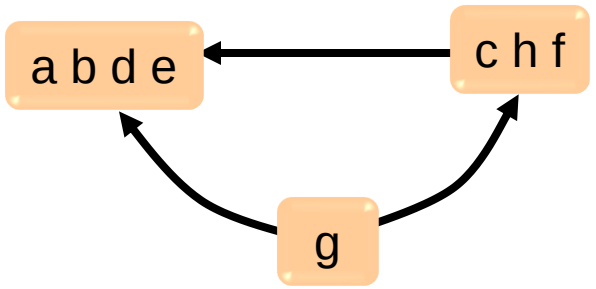


nnected Components



NOT Strongly Connected:
there is no path from d to a.

SCC Component Graph
is a DAG:



Strongly Connected Components

- **Reachability:** $u \rightsquigarrow v$.

  v is reachable from u, i.e., there is a directed path from u to v.

  - Reflexive
  - Transitive

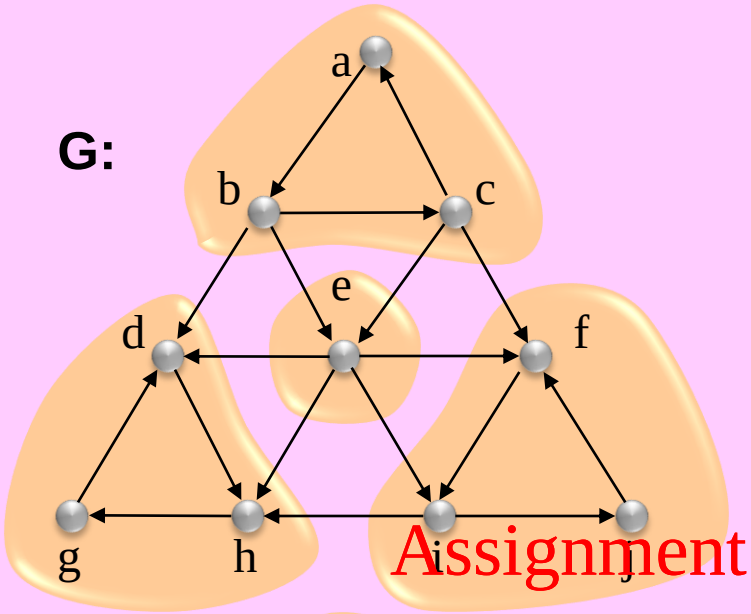
- **Mutual Reachability:** $u \longleftrightsquigarrow v$: u and v are mutually reachable,

  - Reflexive
  - Symmetric $\Rightarrow$ equivalence
  - Transitive

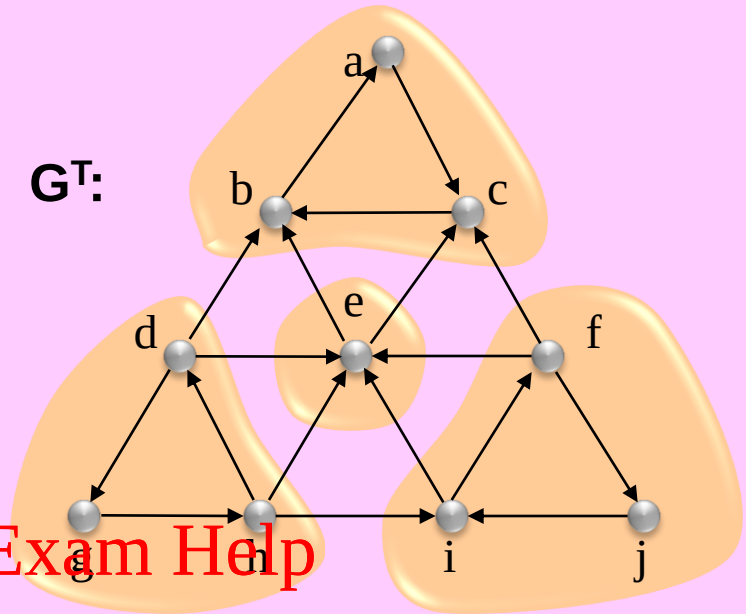
- $\longleftrightsquigarrow$ partitions V(G) into equivalence classes.
- **Strongly Connected Components** of G:

  Subgraphs of G induced by these equivalence classes.
- Remaining edges are called cross-component edges.

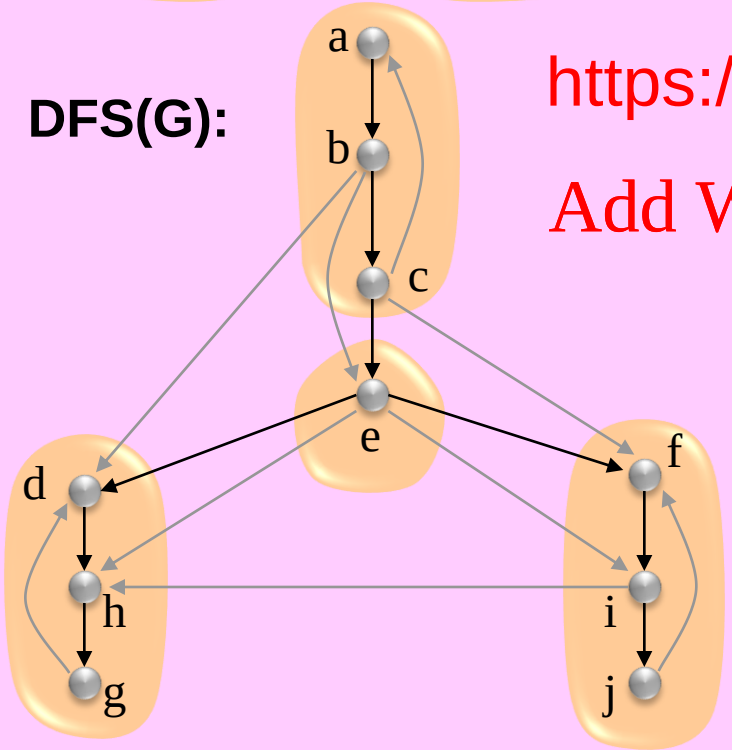
G:



G^T:



DFS(G):



By the WPT
y, node of an SCC
falls within the same DFS tree.

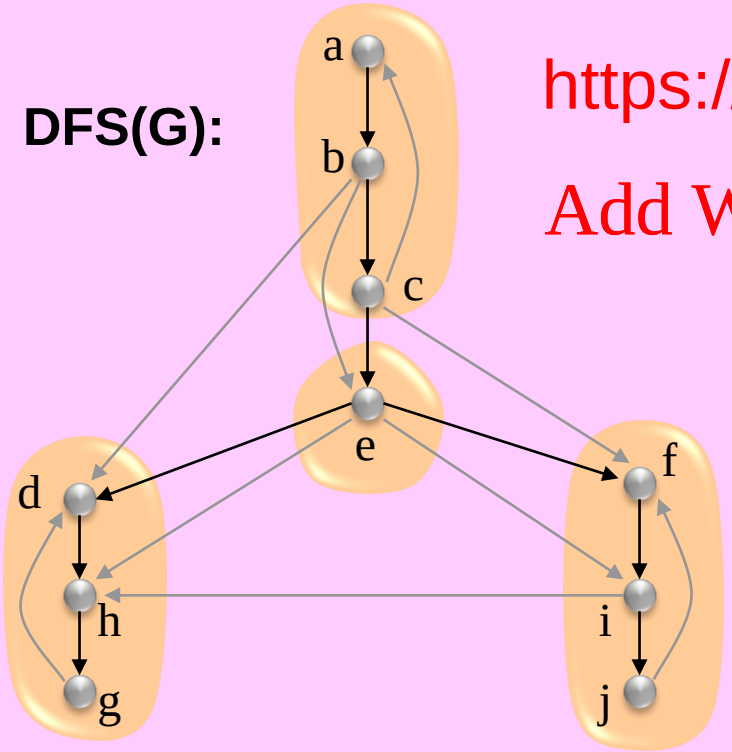
So, each DFS tree is
the union of one or more
SCC's

How can we disentangle them?

- **SCC roof** = the node x with minimum d[x] among all nodes of the SCC (nodes d, f, e, a in the example below).
- **WPT** ⇒ all nodes of an SCC are DFS descendants of its roof.
- ∴ SCC roof x has maximum f[x] among all nodes of the SCC.

- No path can re-enter an SCC after leaving it.
- SCC roofs are scanned (colored BLACK) **in reverse topological order** of the SCC Component Graph.

DFS(G):



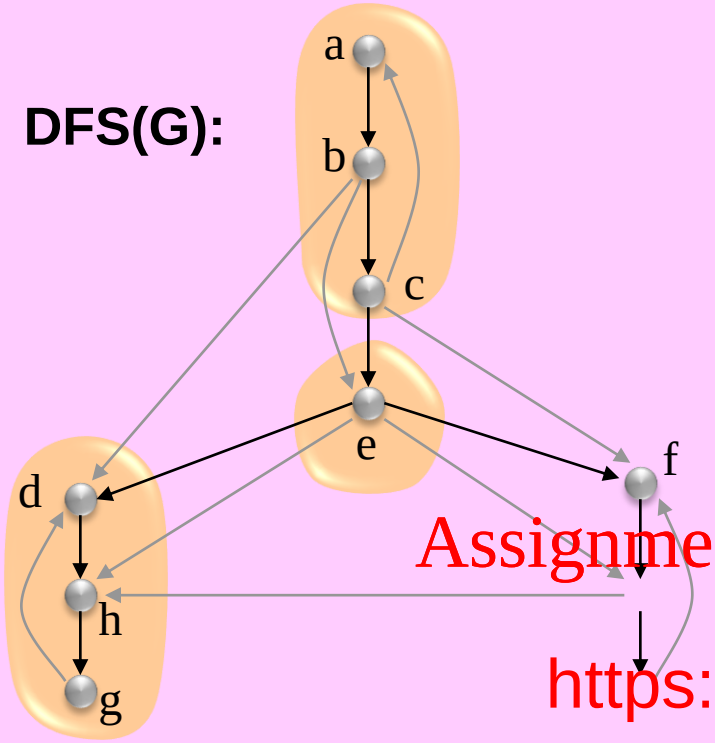
to disentangle SCCs:

c
e
f
i
j
d
h
g

FS, this time on G^T,
rse of G,
but pick DFS tree roots in
reverse of the above
"reverse topological order".

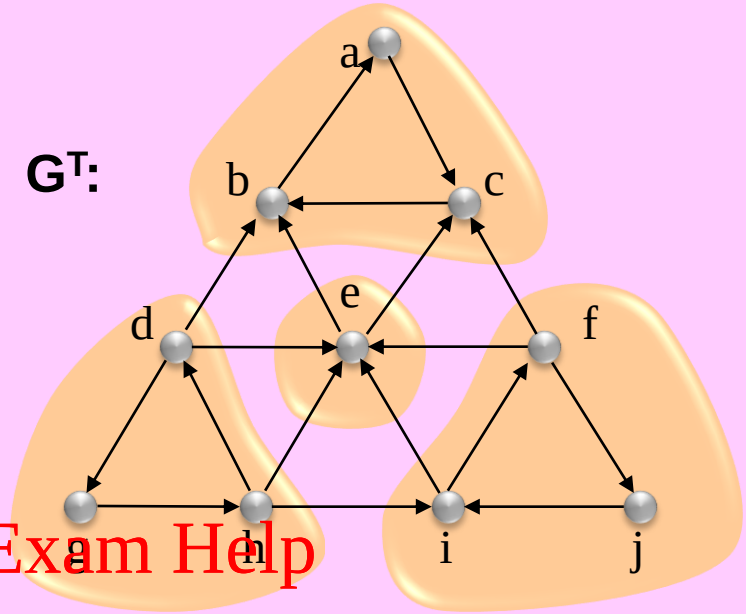
This requires stacking of nodes in the 1st DFS as in Topological Sort Algorithm 2.

DFS(G):

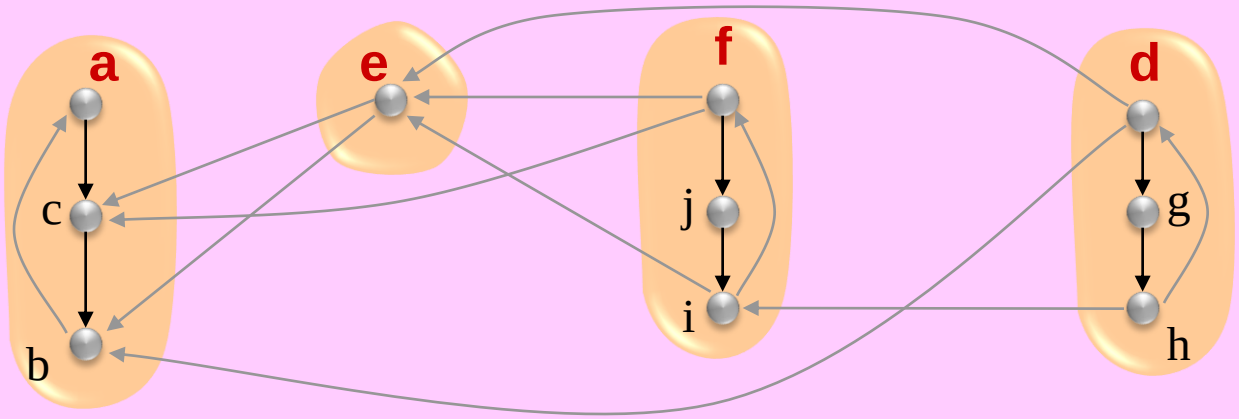


S: a b c e f i j d

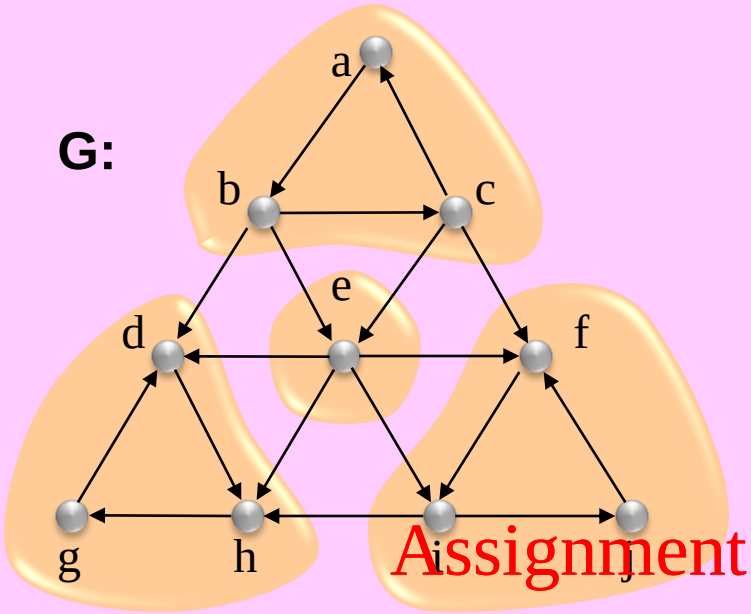
G^T:



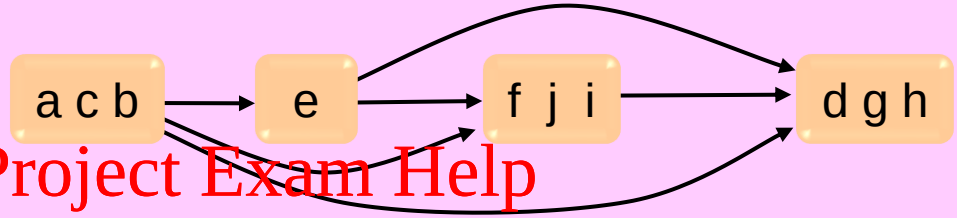
DFS(G^T):



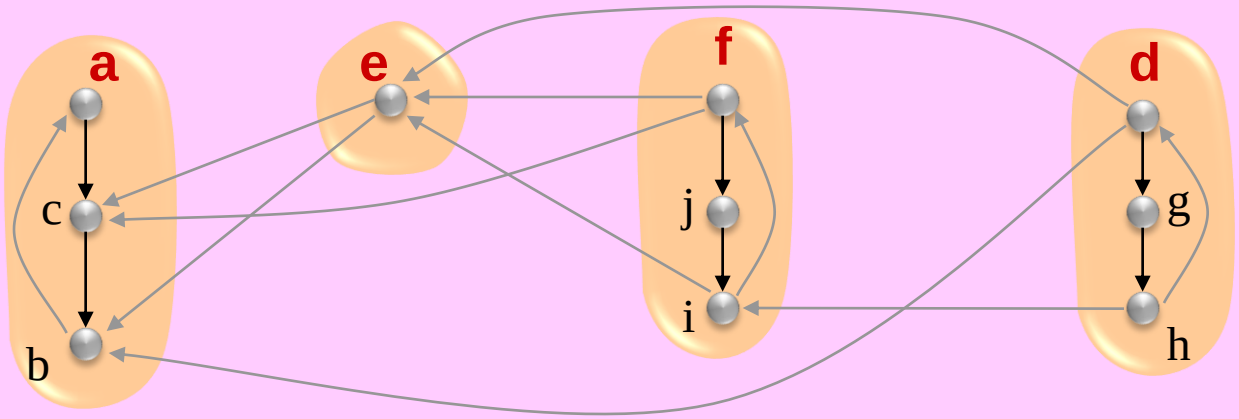
G:



Topological Sort
of the
SCC Component Graph of G:



DFS(G^T):



SCC Algorithm – Kosaraju-Sharir [1978]

Algorithm StronglyConnectedComponents(G)

1. Initialize stack S to empty, and call DFS(G) with the following modifications: push nodes onto stack S when they finish their DFSvisit calls.
   I.e., at the end of the procedure DFSvisit(u) add the statement "Push(u,S)".
   (There is no need to compute d[u] & f[u] values explicitly.)

2. Construct the adjacency list structure of G^T from that of G.

3. Call DFS(G^T) with th ate DFS-roots in
   stack-S-order, i.e., in nstead of
   "**for** each vertex u∈V(G^T) **do if** colo **hen do DFSvisit**(u)"
   perform the following:

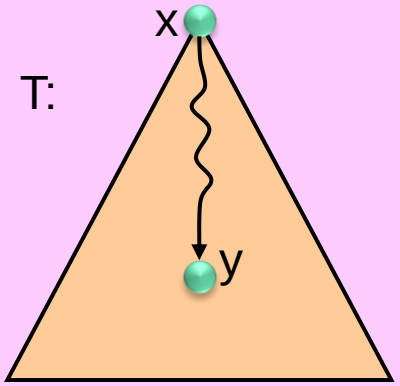
   while S ≠ ∅ **do**
   u ← Pop(S)
   if color[u] = WHITE **then do DFSvisit**(u)
   end-while

4. Each DFS-tree of step 3 (plus all edges between its nodes) forms an SCC.
   SCCs are discovered in topological order of the SCC Component Graph of G.

end

FACT 1: Each DFS tree T of DFS(G^T) consists of a **single** SCC.

Proof:



1) Among nodes in T, root x has max f[.] (wrt 1st DFS(G)).
   It came out of stack S before any other node in T.

2) Among the nodes in T, let y be the one with min d[.]
   (wrt 1st DFS(G)).

3) P = the tree path in T (and G^T) from x to y.

4) P^T is a path in G fro

5) (2) $\Rightarrow$ P^T was an y was discovered.

6) (5) + WPT $\Rightarrow$ x is a descendant of y

7) (6) + P^T $\Rightarrow$ f[x] $\le$ f[y] (wrt 1st DF

8) (1) + (7) $\Rightarrow$ f[x] = f[y] $\Rightarrow$ x = y.

9) (1)+(2)+(8) $\Rightarrow$ $\forall$node u$\in$T: d[x] $\le$ d[u] < f[u] $\le$ f[x] (wrt 1st DFS(G)).

10) (9) + P^T $\Rightarrow$ $\forall$node u$\in$T: u is a descendant of x in 1st DFS(G).

11) (10) $\Rightarrow$ $\forall$node u$\in$T: $x \rightsquigarrow u$ in G.

12) $\forall$node u$\in$T: $x \rightsquigarrow u$ in G^T.

13) (11) + (12) $\Rightarrow$ $\forall$node u$\in$T: u is in the same SCC as x. □

FACT 1: Each DFS tree T of $DFS(G^T)$ consists of a **single** SCC.

FACT 2: In $DFS(G^T)$, the DFS trees (i.e., SCCs) are discovered in topological order of the SCC Component Graph of G.

Proof:

In $DFS(G^T)$, each DFS tree is an SCC (by Fact 1).

Cross-component edges, i[T] tree to a node in an earlier DFS tree (i.e., from right t

Reverse cross-component edges of G^T to con , from left to right). □

THEOREM: The algorithm correctly finds the SCCs in topological order of the SCC Component Graph of G in O(V + E) time.

Bi-CONNECTED COMPONENTS

We must all hang together,

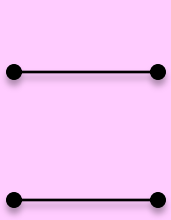
or assuredly we shall hang separately.

Benjamin Franklin

Node Connectivity

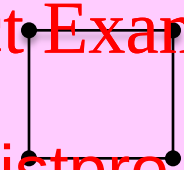
undirected graph G = (V, E) & integer k ≥ 0.

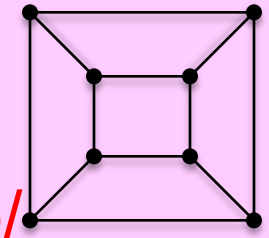
G is **k-connected** (or **k-node-connected**):
if removal of fewer than k nodes (& all their incident edges) cannot disconnect G.



0-connected
but not
1-connected

1-connected
but not
2-connected





3-connected
but not
4-connected

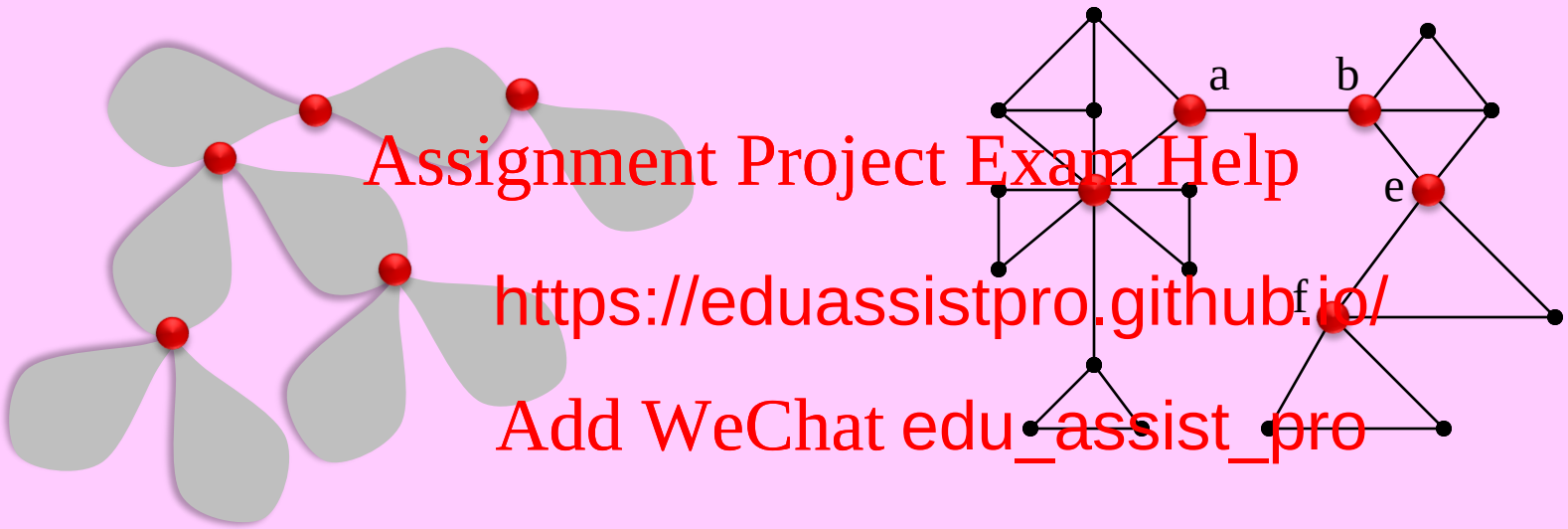
FACT 1: G is k-connected if and only if between every pair of its nodes there are at least min{ $k, |V|$-1} node-disjoint paths (excluding their common end points).

Balinski's Theorem [1961]:
The node-edge skeleton of a k dimensional polytope is k-connected.

BiConnectivity

Suppose G = (V, E) is connected.

An **articulation point** is a node whose removal will disconnect G.



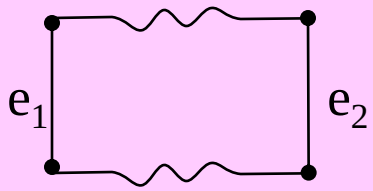
FACT 3: $x \in V$ is an articulation point of G if there are two distinct nodes u & v in G distinct from x, such that every path between u & v passes through x.

FACT 4: G is biconnected $\Leftrightarrow$ G has no articulation points.

Biconnected Components

- **An equivalence relation on E:**

  $e_1 , e_2 \in E$: $e_1 \equiv e_2$ if and only if

  1. $e_1 = e_2$, or
  2. There is a simple cycle that contains e_1 and e_2 .

  reflexive
  symmetric
  transitive

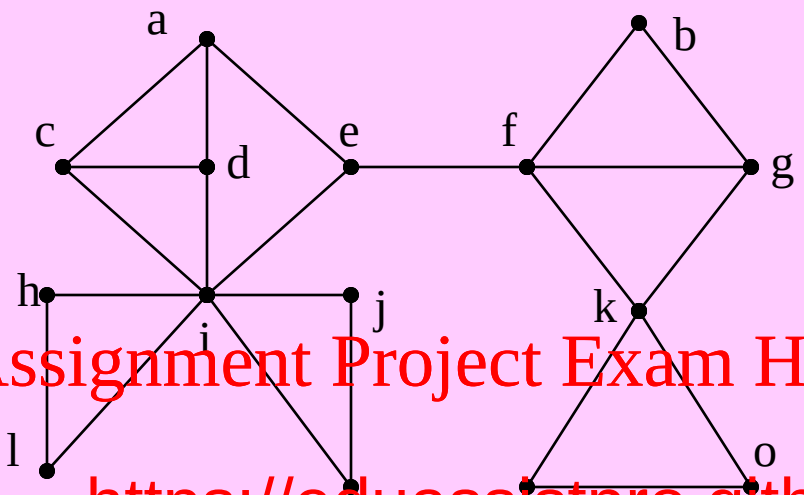
- Equivalence relation e classes

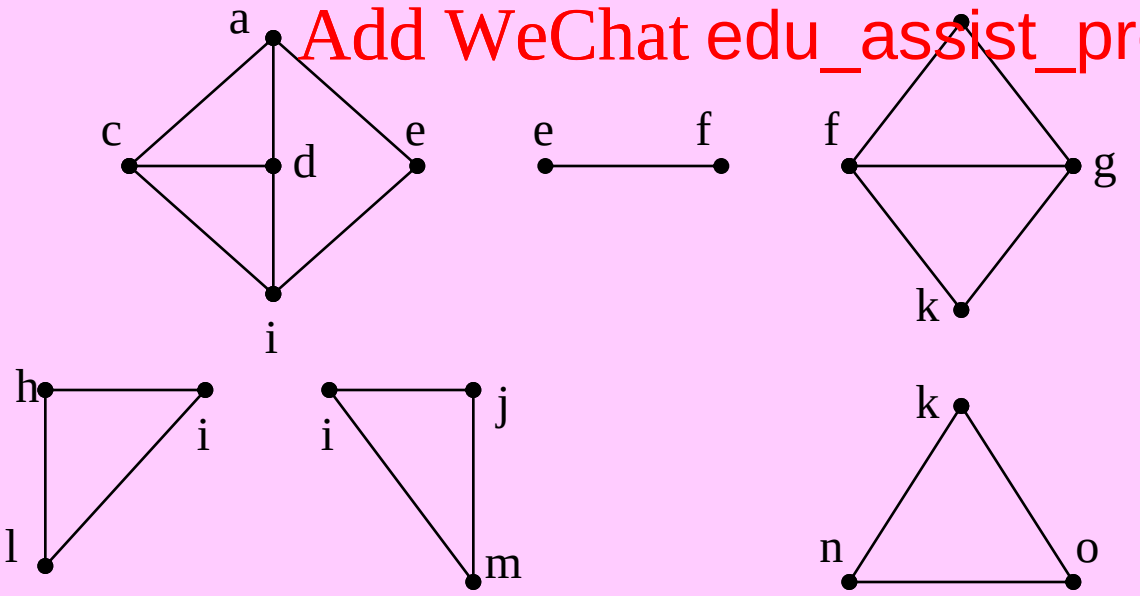
  $E_1 , E_2 , \ldots , E_k$, for some k

- V_i = set of vertices of E_i , i = 1..k.

- **Biconnected Components** of G: $G_i = (V_i , E_i)$, i = 1..k.

Example



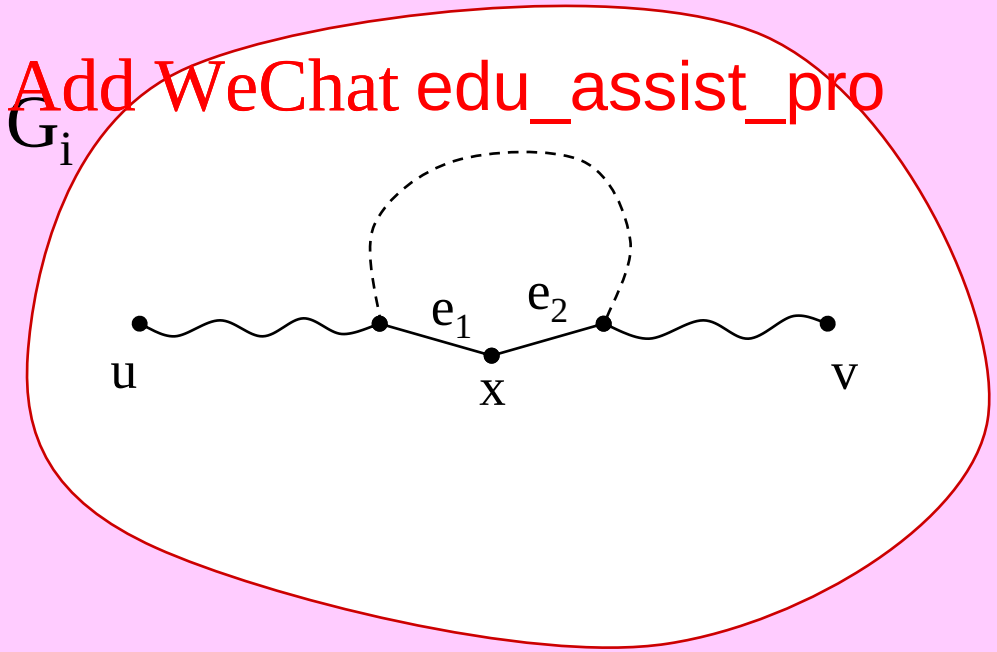


Biconnected Components & Articulation Points

FACT 5: Let $G_i = (V_i , E_i)$, i = 1..k, be biconnected components of G. Then,

1. G_i is biconnected , for i = 1..k,
2. $V_i \cap V_j$ has at most one node, for $i \neq j$,
3. node $x \in V$ is an articulation point of $G \Leftrightarrow x \in V_i \cap V_j$ for some $i \neq j$.

Proof of (1): G_i cannot h

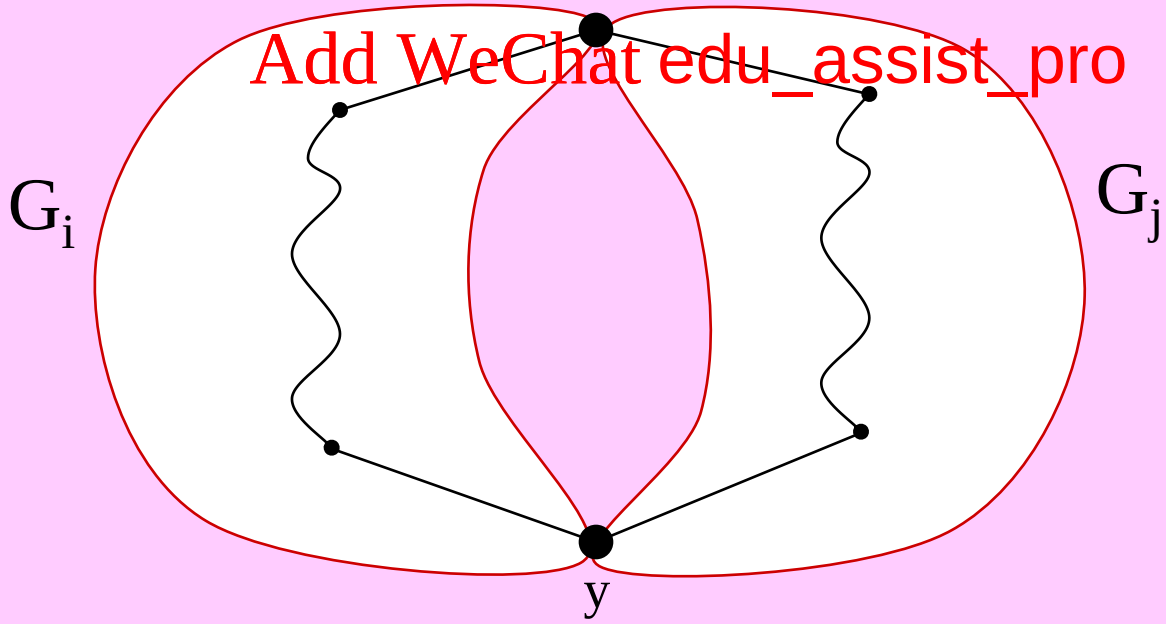


Biconnected Components & Articulation Points

FACT 5: Let $G_i = (V_i , E_i)$, i = 1..k, be biconnected components of G. Then,

1. G_i is biconnected , for i = 1..k,
2. $V_i \cap V_j$ has at most one node, for $i \neq j$,
3. node $x \in V$ is an articulation point of $G \Leftrightarrow x \in V_i \cap V_j$ for some $i \neq j$.

Proof of (2): G_i and G_j ca in common, say x and y:



Biconnected Components & Articulation Points

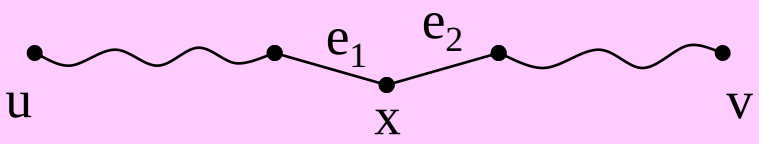
FACT 5: Let $G_i = (V_i , E_i)$, i = 1..k, be biconnected components of G. Then,

1. G_i is biconnected , for i = 1..k,
2. $V_i \cap V_j$ has at most one node, for $i \neq j$,
3. node $x \in V$ is an articulation point of $G \Leftrightarrow x \in V_i \cap V_j$ for some $i \neq j$.

Proof of (3): [$\Rightarrow$]:

Suppose x is an articul y path between u & v.

Consider one such path, and let e_1 and e_2 es on the path incident to x.



Then, $e_1 \not\equiv e_2$.

So, $e_1 \in E_i$ and $e_2 \in E_j$ for some $i \neq j$.

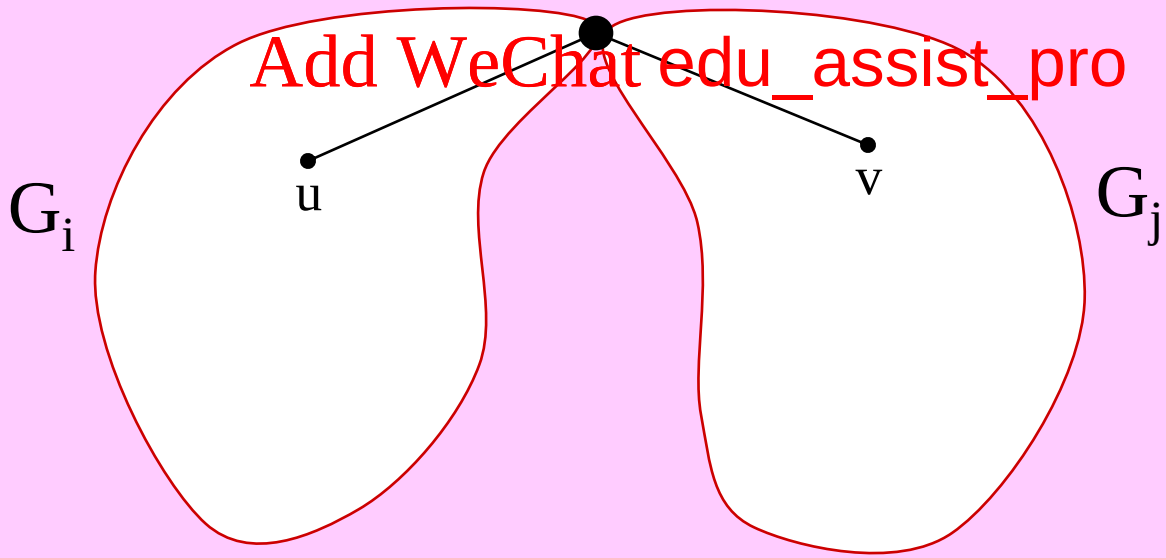
Therefore, $x \in V_i \cap V_j$ for some $i \neq j$.

Biconnected Components & Articulation Points

FACT 5: Let $G_i = (V_i , E_i)$, i = 1..k, be biconnected components of G. Then,

1. G_i is biconnected , for i = 1..k,
2. $V_i \cap V_j$ has at most one node, for $i \neq j$,
3. node $x \in V$ is an articulation point of $G \Leftrightarrow x \in V_i \cap V_j$ for some $i \neq j$.

Proof of (3): [$\Leftarrow$]:

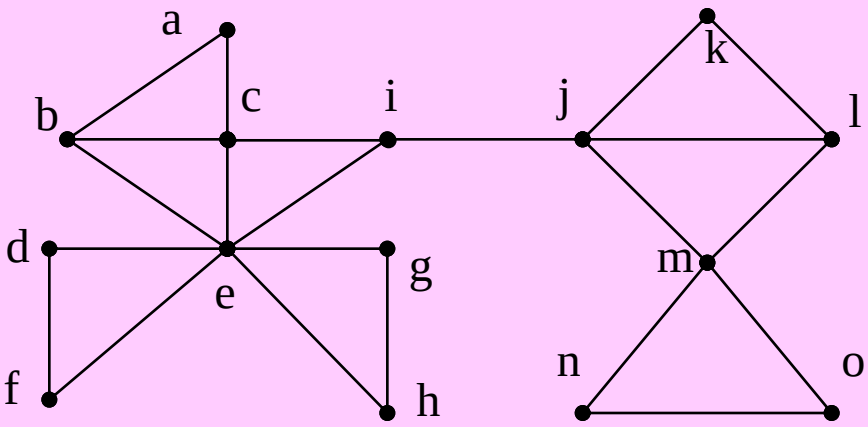


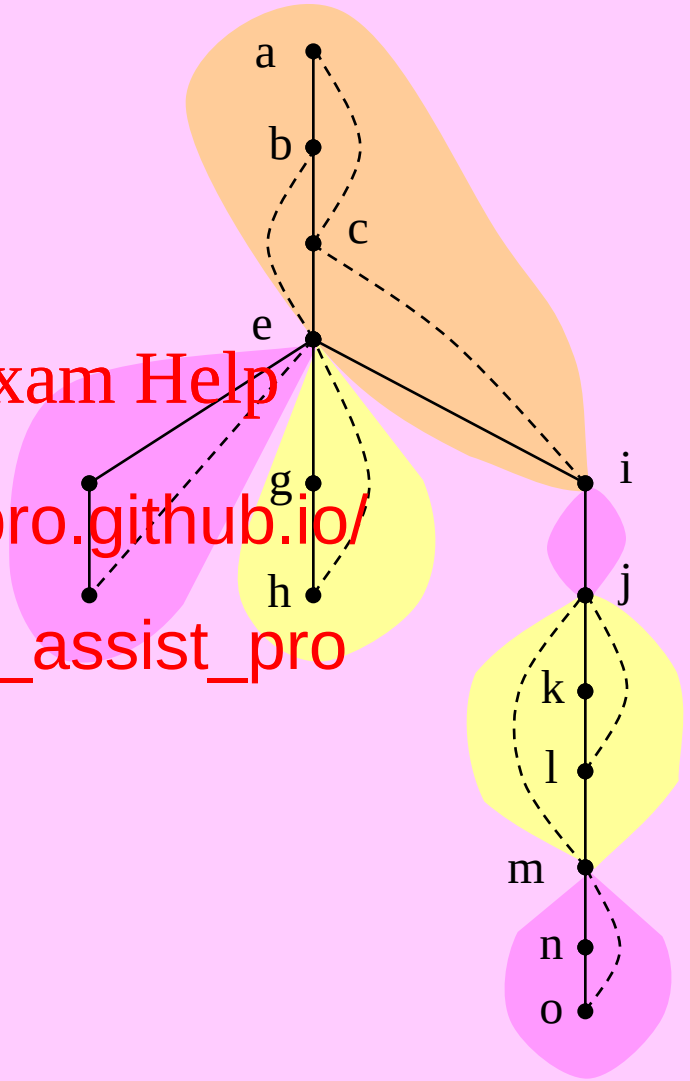
Apply DFS

DFS is particularly useful in finding the biconnected components of a graph, because in the DFS there are no cross edges; only tree- and back-edges.

A back-edge (u,v) forms a simple cycle with the DFS tree path between u and v.

Hence, all edges of that c belong to the same biconn



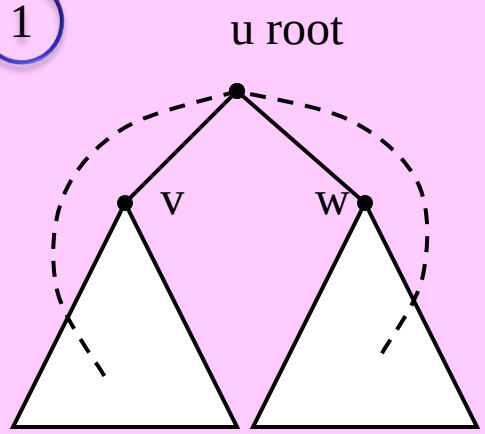


Articulation Points & DFS

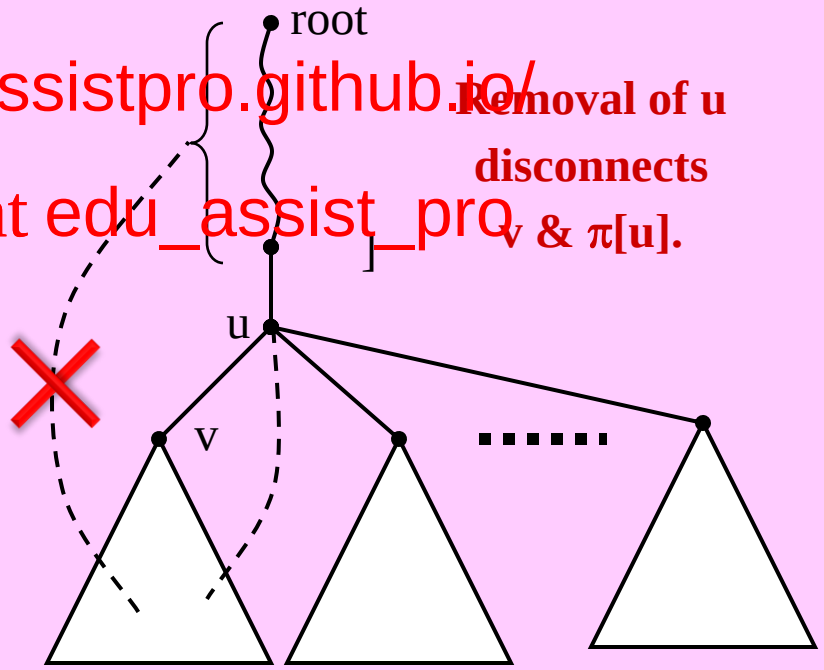
FACT 6: Node u is an articulation point ⇔ either 1 or 2 holds:

1. u is a DFS root with at least 2 children, or
2. u is not root, but it has a child v such that there is no back-edge between any descendant of v (including v) and any ancestor of parent π[u] of u (including π[u]).

1



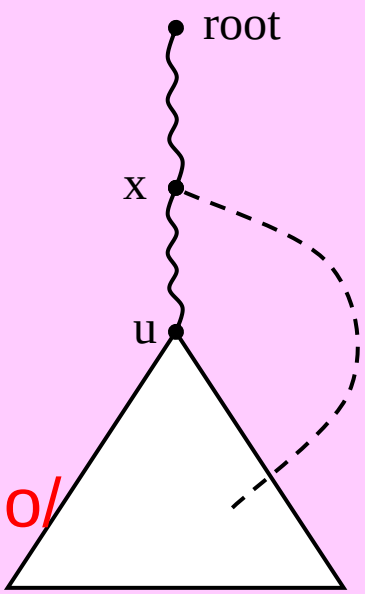
No cross-edges. Removal of u disconnects v & w.



Removal of u disconnects v & π[u].

DFS "low" Computation

- Do a DFS and for each node $u \in V(G)$ compute:
- pre[u] = DFS pre-order numbering of node u.
- $\text{low}[u] \stackrel{\text{def}}{=} \min \left\{ \text{pre}[x] \;\middle|\; \begin{array}{l} x = u, \quad \text{or} \\ \exists \text{ back} - \text{edge between } x \\ \text{and some descendant of } u \end{array} \right\}$.

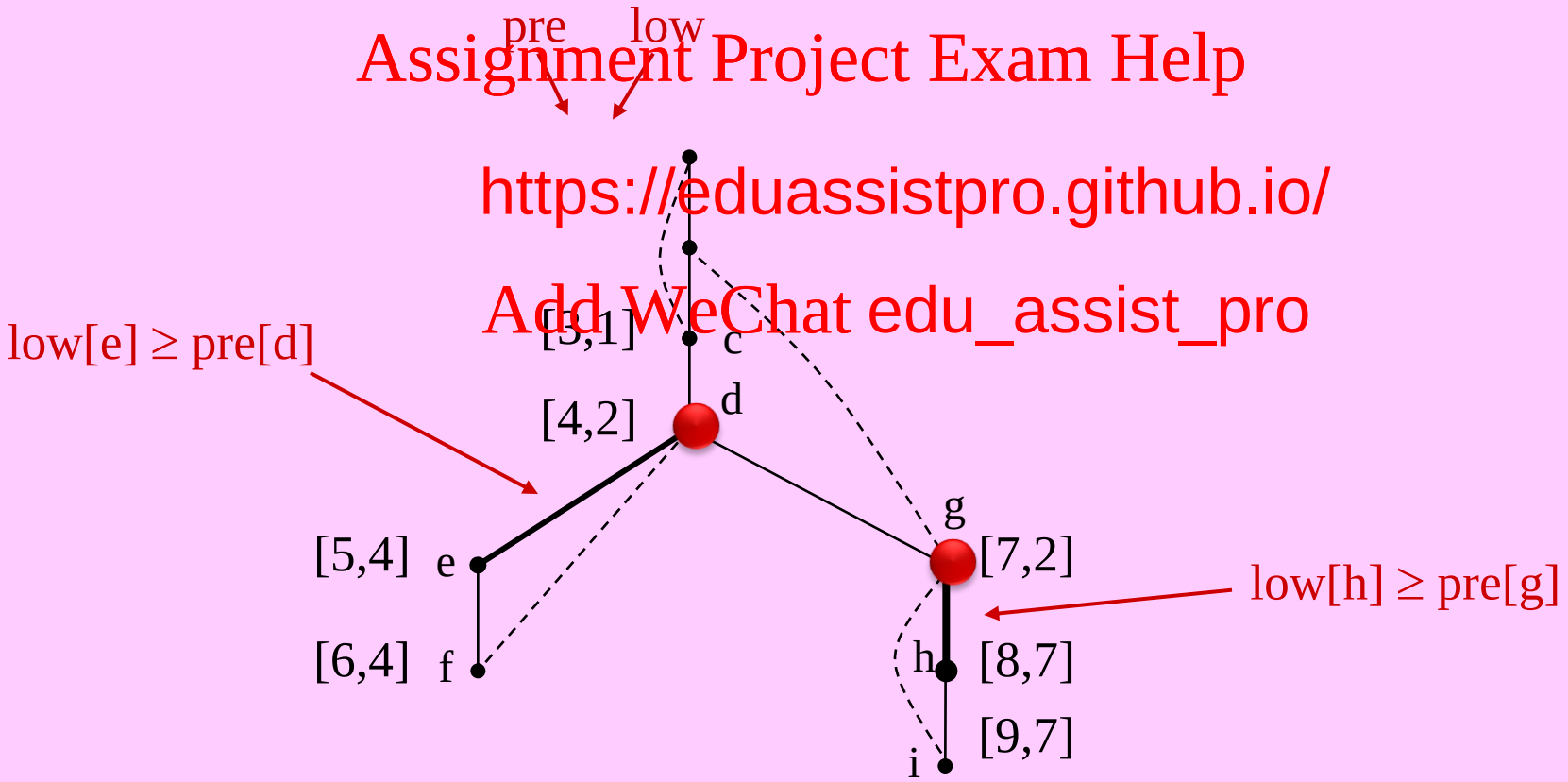


FACT 7: $\text{low}[u] = \min \left(\begin{array}{l} \{ \text{pre}[u] \} \\ \cup \{ \text{pre}[x] \mid \exists \text{ back} - \text{edge } (u, x) \} \\ \cup \{ \text{low}[c] \mid c \text{ is a DFS} - \text{child of } u \} \end{array} \right)$.

Articulation Points & DFS "low" numbers

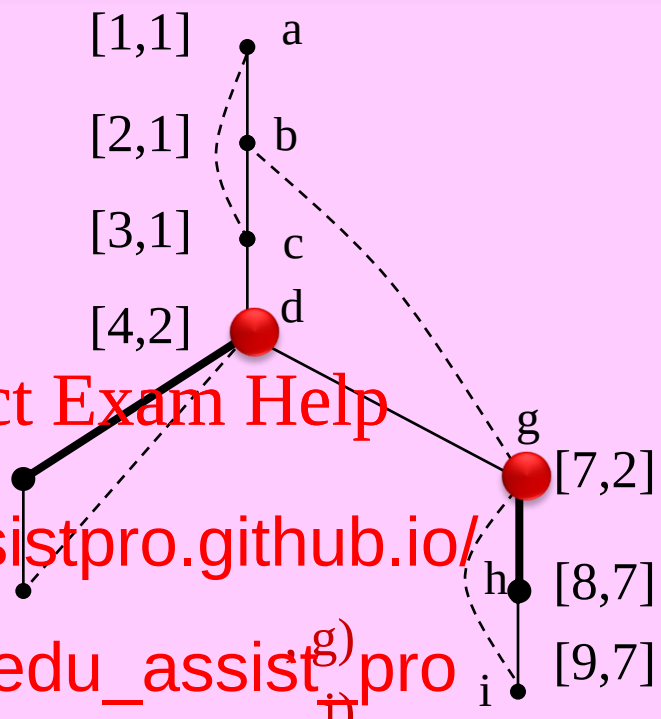
FACT 8: Node u is an articulation point ⇔ either 1 or 2 holds:

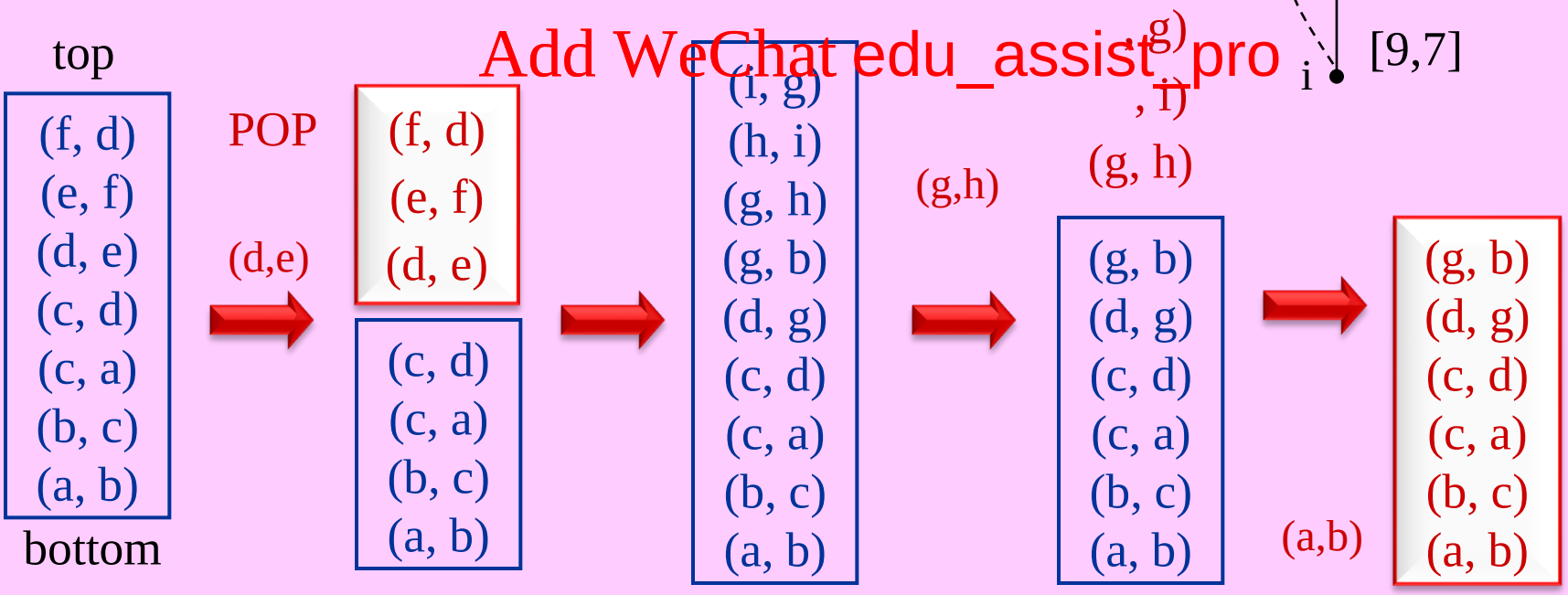
1. u is a DFS root with at least 2 children, or
2. (u is not root &) ∃ a child v of u such that low[v] ≥ pre[u].



Stack up edges

- Stack up each edge (u,v) at its 1st traversal, i.e., if $v \neq \pi[u]$ & $pre[v] < pre[u]$.
- Articulation points discovered in post-order of child-node.
- Pop-up the stack up to (u,v), inclusive, $u = \pi(v)$ & $low(v) \geq p$





BiConnectivity Algorithm – Hopcroft [1972]

Algorithm BiConnectivity (G) **§ O(V + E) time**

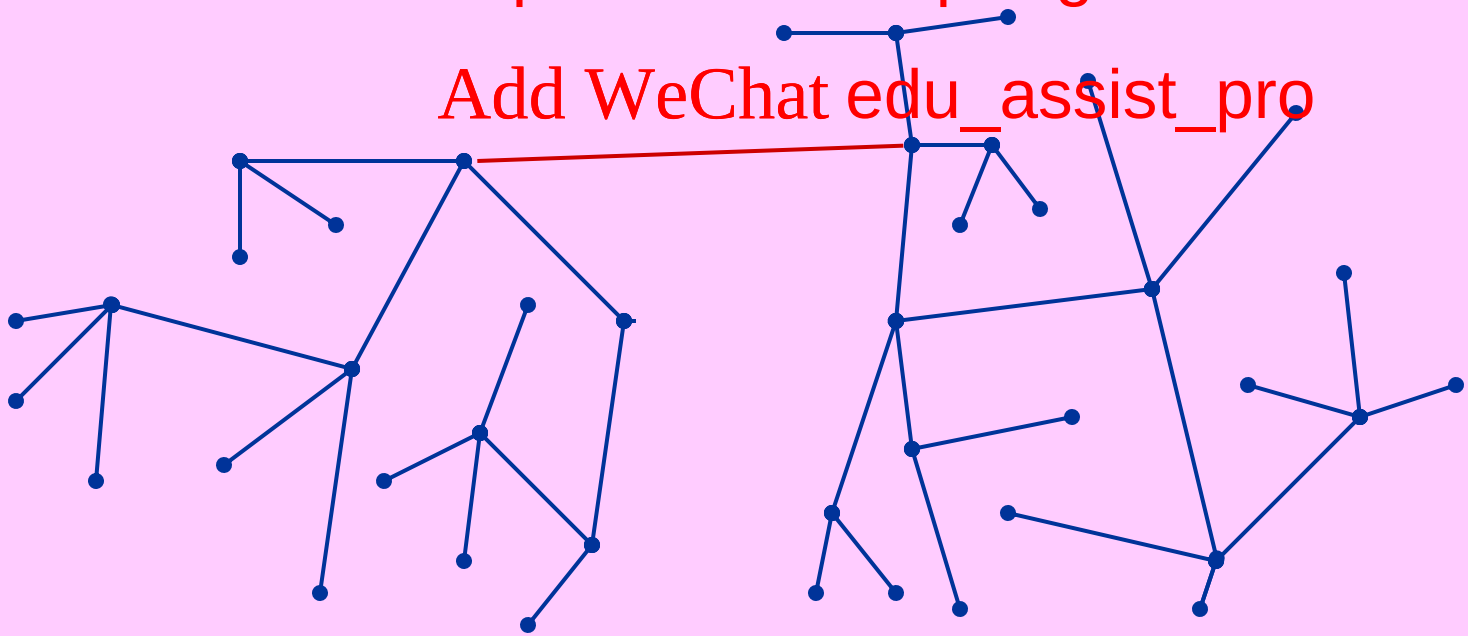
```
1.   S ← ∅ ;  time ← 0                                   § assume no isolated nodes
2.   for each vertex u ∈ V(G) do  ⟨pre[u], π[u]⟩ ← ⟨0, nil⟩
3.   for each vertex u ∈ V(G) do  if pre[u] = 0 then SearchBC(u)
end

Procedure SearchBC(u)
4.   pre[u] ← low[u] ← time ← time + 1
5.   for each vertex v ∈
6.        if  v ≠ π[u] & pre                          S)        § 1st traversal of (u,v)
7.        if pre[v] = 0  the
8.             π[v] ← u                                        § (u,v) is a tree-edge
9.             SearchBC(v)
10.            if low[v] ≥ pre[u]  then PrintB
11.            low[u] ← min { low[u] , low[v] }
12.       else if v ≠ π[u]  then low[u] ← min { low[u] , pre[v] }   § (u,v) is a back-edge
13.  end-for
end

Procedure PrintBC(u,v)
14.  print ( "new biconnected component with following edges is found:")
15.  Iteratively pop S up to and including edge (u,v) and print them
end
```

What is an undirected tree?

Let T be an undirected graph on n nodes.

The following statements are equivalent:

1. T is a tree.
2. T is connected and acyclic.
3. Between each pair of nodes in T there is a unique simple path.
4. T is connected and has n –1 edges.
5. T is acyclic and has n – 1 edges.
6. T is minimally connec sconnect T.
7. T is maximally acyclic creates a simple cycle.



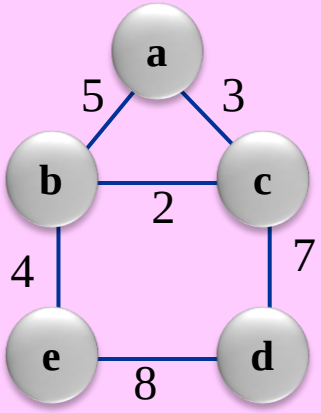
The MST Problem

INPUT: A weighted connected undirected graph G = (V, E, w).

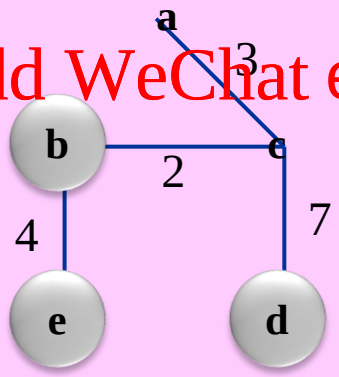
OUTPUT: A Minimum Spanning Tree (MST) T of G.

Feasible Solutions: Any spanning tree of G, i.e., a subgraph of G that is a tree and spans (i.e., includes) V.

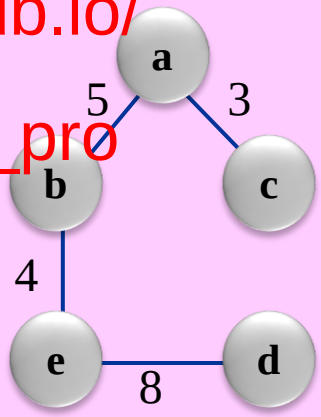
Objective: Minimize Cost(T) = sum of the weights of edges in T.



G



Minimum spanning tree T
Cost(T) = 3+2+4+7 = 16



Another spanning tree T'
Cost(T') = 3+5+4+8 = 20

MST Preliminaries

Applications:

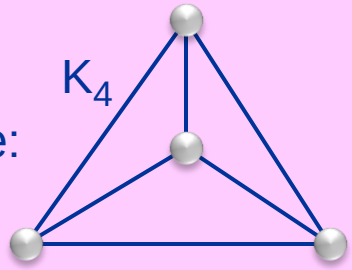
- Optimum cost network interconnection (connected & acyclic)
- Minimum length wiring to connect n points
- Sub-problem to many other problems

FACT:

- A connected undirected g tree (e.g., the DFS tree).
- So, a connected weight T.

Caley's Theorem: K_n, the complete graph on n nodes, has n^{n-2} spanning trees.

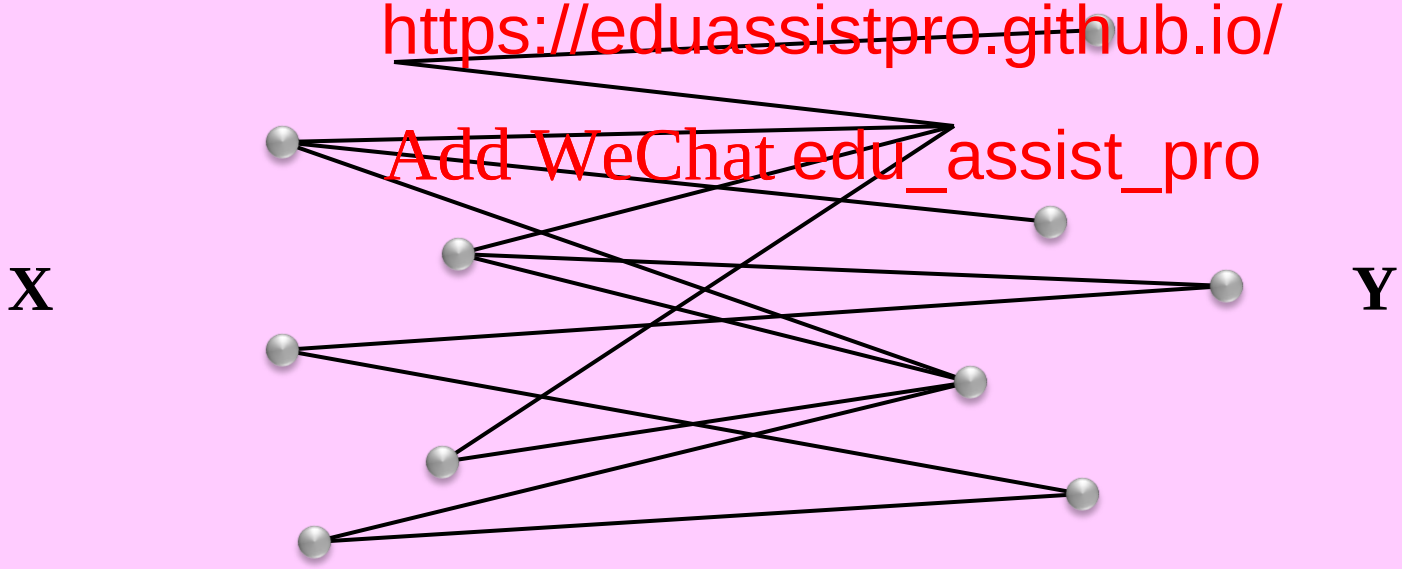
For example:



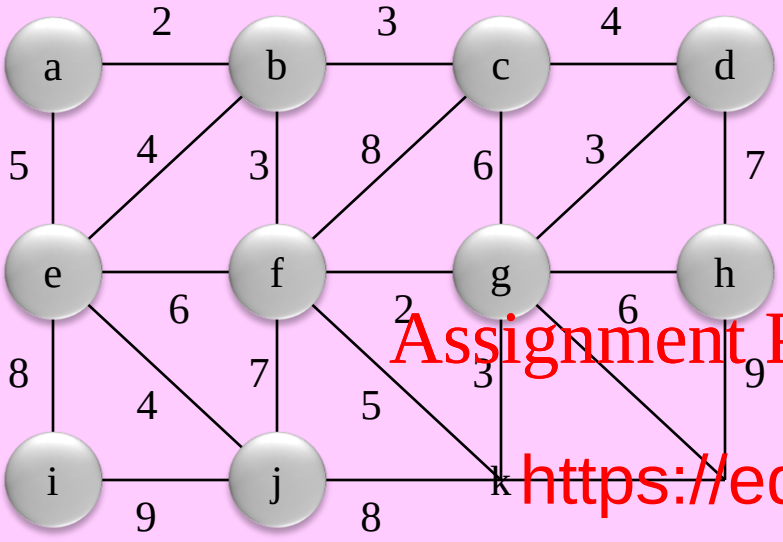
has $4^2 = 16$ spanning trees.

A CUT

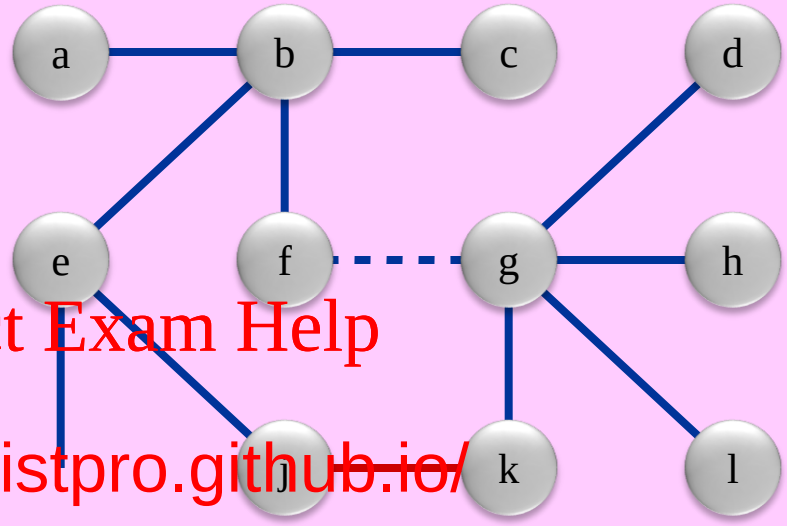
- If S is a set and e an element, then S+e denotes S ∪ {e}, & S-e denotes S – {e}.
- An edge e is incident to subgraph (or subset of vertices) H,
  if exactly one end of e is in H.
- A **cut** C = (X,Y) is a partitioning of V(G) into two non-empty subsets X and Y.
  Cross-edges of C are edges in E(G) that are incident to X (and also to Y).



MST Properties



G



T = MST of G

- **T + (j,k)** has a unique simple cycle.
  What is the maximum edge weight on that cycle?
  What is the range of values for w(j,k) while T remains an MST?

- **T – (f,g)** is disconnected.
  Its two connected components induce a cut in G.
  What is the minimum weight cross-edge of that cut?
  What is the range of values for w(f,g) while T remains an MST?

A Generic Greedy Strategy

- Assume initially all edges of G are **white** (undecided).
- We will iteratively select a white edge and make a permanent greedy decision, either coloring it **blue** (accepted) or **red** (rejected).
- This is done according to the following two rules, whichever applies:

- **BLUE Rule:** Consider a cut C with no blue cross-edges.
  Then select a minimum weight white cross-edge of C and color it blue.

- **RED Rule:** Conside
  Then select a maximu lor it red.

- **Red-Blue Invariant:** At all times, there i that includes all blue edges and excludes all red edges.

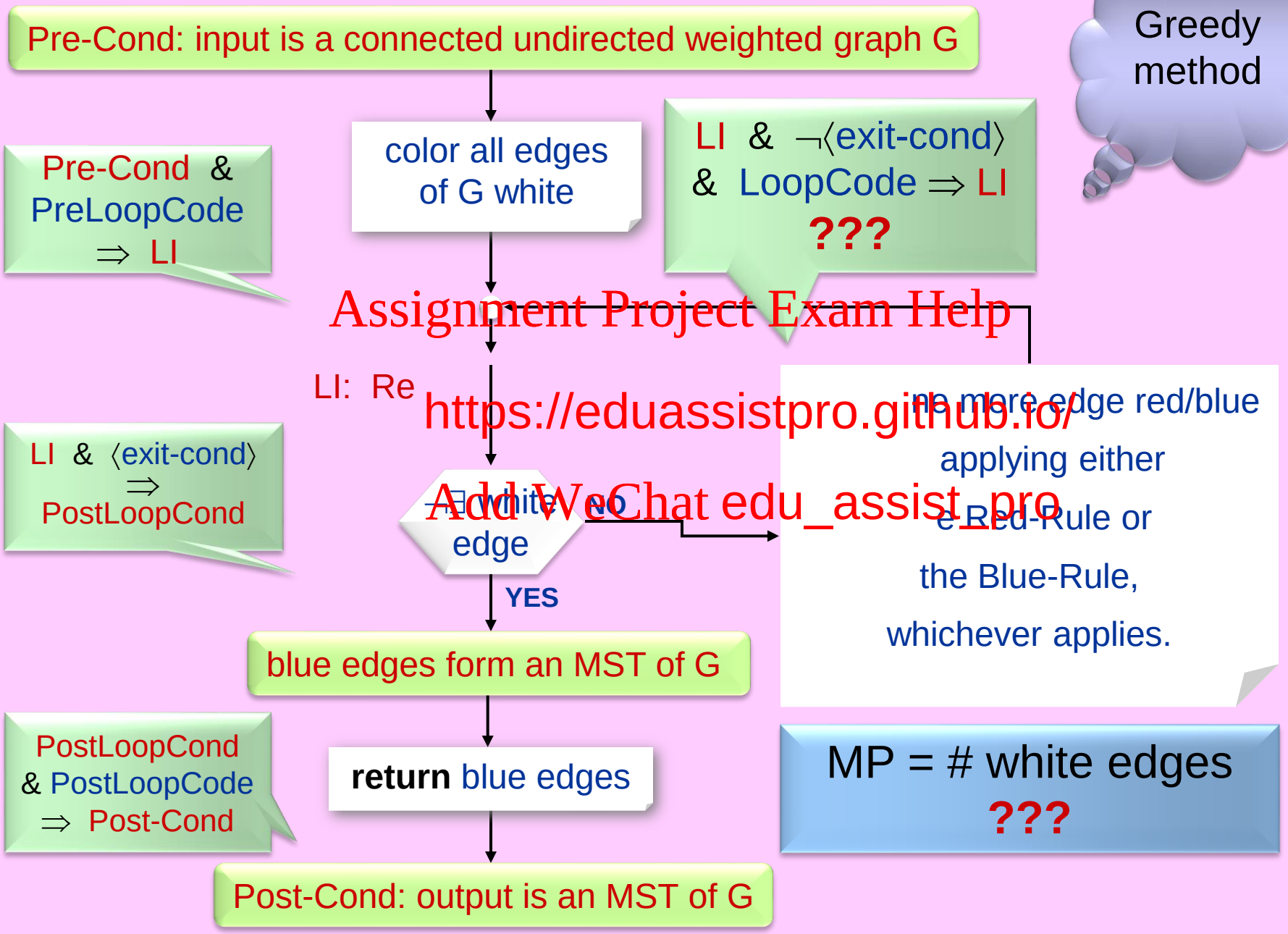
Proofs coming up

- **Progress:** either no white edge remains, or at least one of the two rules applies.

- In |E(G)| iterations all edges are colored blue/red. The blue edges form the MST.

- There are many MST algorithms, and virtually all of them are specializations of this generic greedy strategy depending on how the two rules are applied.

Generic MST Algorithm



Must prove 3 things:

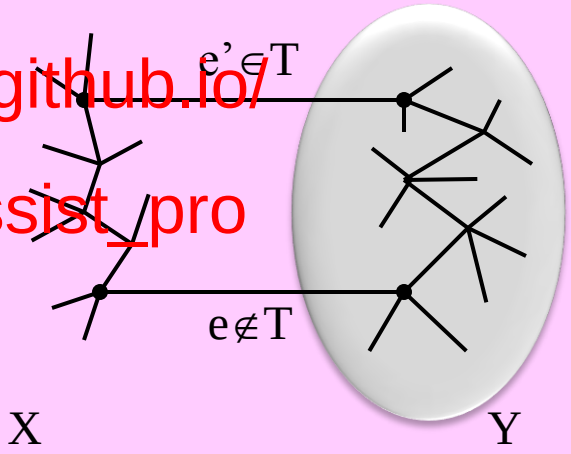
LI & $\neg\langle$exit-cond$\rangle$ & Blue-Rule applied $\Rightarrow$ **LI**

Progress: If there is a white edge, then at least one of the 2 rules is applicable

LI & ¬⟨exit-cond⟩ & Blue-Rule applied ⇒ LI

Proof:

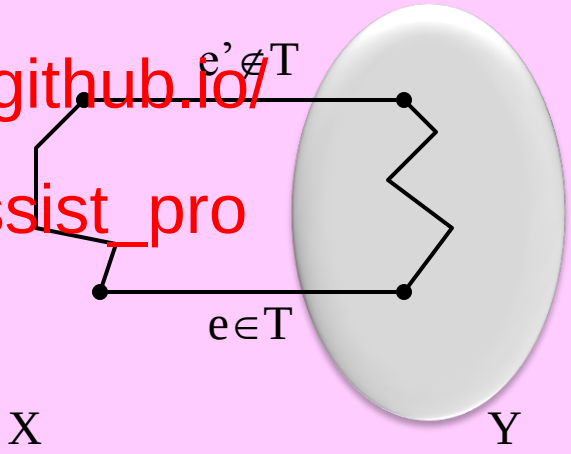
- There is an MST T that includes all blue edges and excludes all red edges.
- Suppose Blue-Rule is applied to cut (X,Y) that contains no blue cross-edges, and minimum weight white edge e of cut (X,Y) is colored blue.
- If e ∈ T, then T still satisfies the LI.
- Suppose e ∉ T.
- T + e contains a unique simple cycle C.
  There must be at least one other (white) ed
  e' ∈ T∩C that crosses the cut (X,Y).
  (Why does e' exist & is white?)
  So, w(e) ≤ w(e').
- T' ← (T + e) – e' now satisfies LI:
  - T' is also a spanning tree of G.
  - T' includes all blue edges (e among them) and excludes all red edges.
  - Cost(T') = Cost(T) + w(e) – w(e') ≤ Cost(T).
    So, T' is also an MST.



LI & ¬⟨exit-cond⟩ & Red-Rule applied ⇒ LI

Proof:

- There is an MST T that includes all blue edges and excludes all red edges.
- Suppose Red-Rule is applied to cycle C that contains no red edges, and maximum weight white edge e of C is colored red.
- If e∉T, then T still satisfies the LI.
- Suppose e∈T.
- T – e is disconnected and induces a cut (X, Y). There must be at least one other (white) edge e' ∈ C that crosses the cut (X,Y). (Why does e' exist & is white?) So, w(e') ≤ w(e).

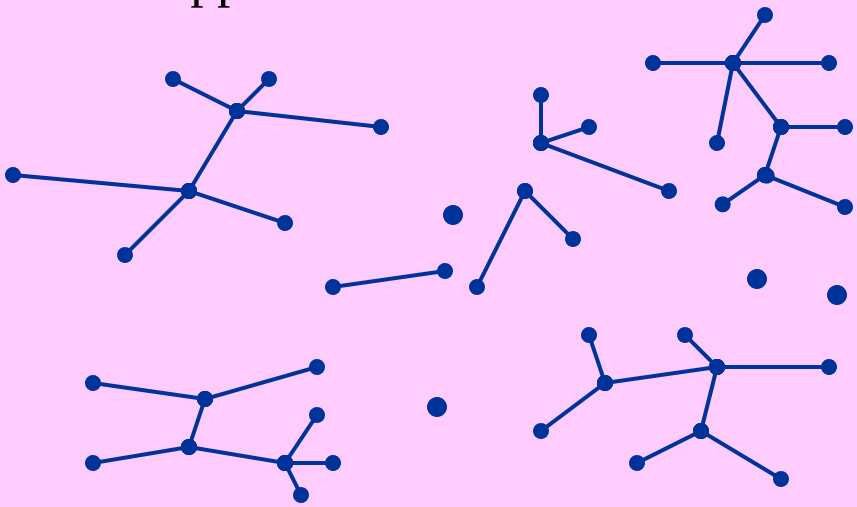


- T' ← (T + e') – e now satisfies LI:
  - T' is also a spanning tree of G.
  - T' includes all blue edges and excludes all red edges (e among them).
  - Cost(T') = Cost(T) + w(e') – w(e) ≤ Cost(T). So, T' is also an MST.

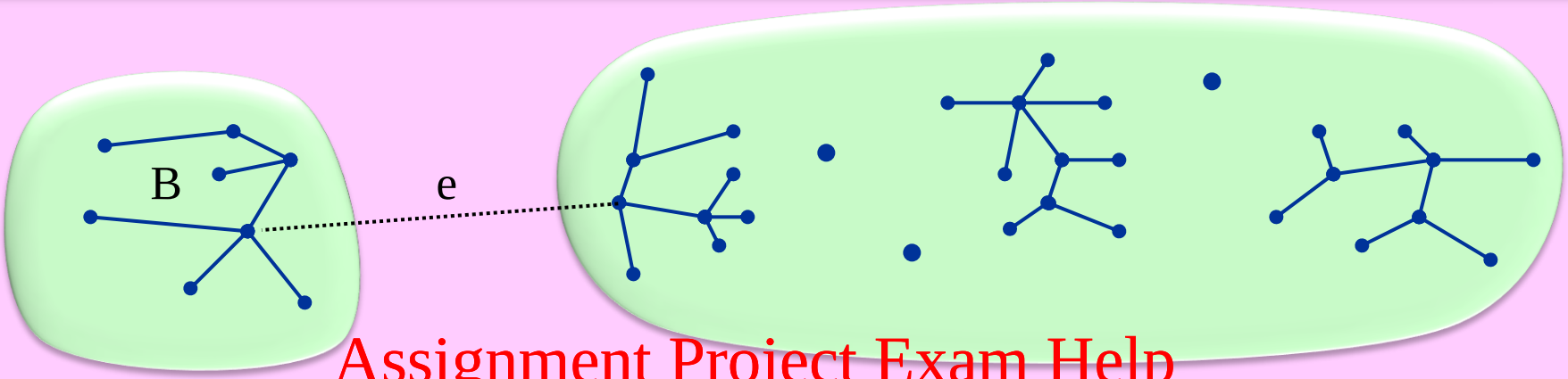
Progress: If there is a white edge, then at least one of the 2 rules is applicable

Proof:

- Suppose there is a white edge e = (u,v).
- If u & v are connected by a blue path, say P, then Red-rule applies to cycle P+e.
- Otherwise, define: X = u by blue paths. Y =
- Cut (X,Y) contains at least one white cross-edge. Why? So, the Blue-rule can be applied to this cut.



Short Bridge



The Blue Spannin lored blue so far.

This subgraph is a n of trees), and

each connected component of it (a tre the component.

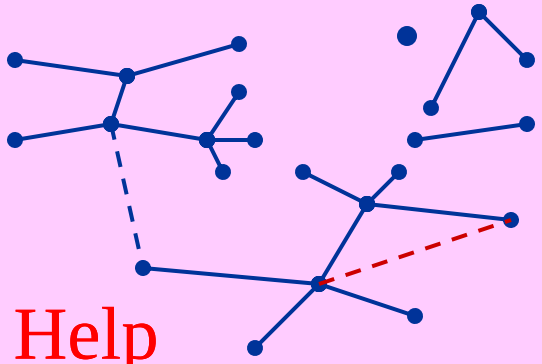
A white edge e is the short bridge of a blue component B if:

- e is incident to blue component B, and
- among all (white) edges incident to B, e has minimum weight.

Short Bridge Rule: A short bridge of any blue component can be colored blue.

MST Construction: Two Classic Algorithms

Kruskal [1956] based on **Borůvka [1926]:**

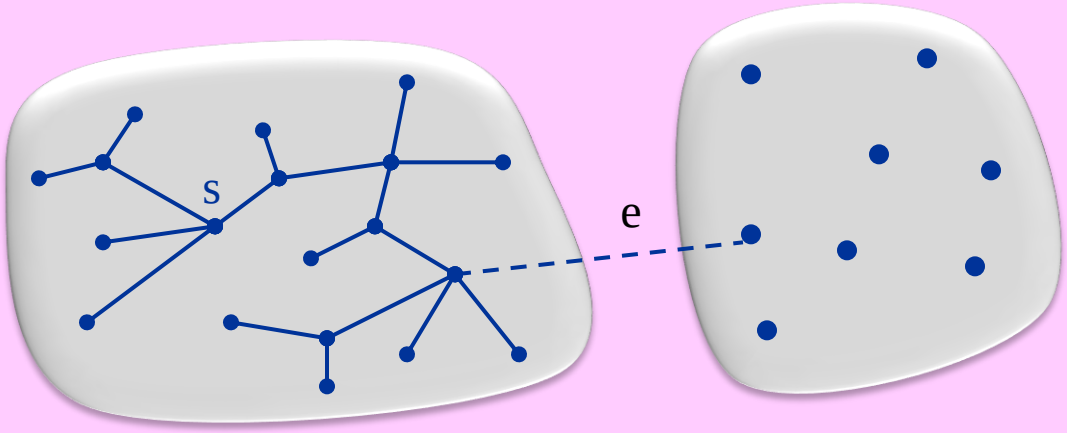


- e = min weight white edge.
- **If** both ends of e are in the same blue component
  then color e red (by the Red-Rule),
  else color e blue (e is a short bridge).

Prim [1957] based on

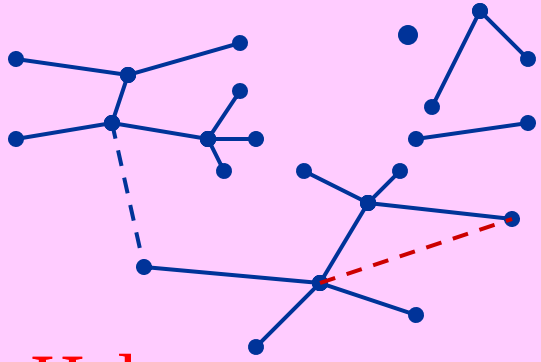
- s = an initially selected starting vertex.

- e = the short bridge incident to the blue t contains s.
- Color e blue.



Kruskal's MST Algorithm

- e = min weight white edge.
- **If** both ends of e are in the same blue component
  then color e red (by the Red-Rule),
  else color e blue (e is a short bridge).



Disjoint Set Union Data Structure:

Maintains the partitioni ue components under the following operation

- MakeSet(v) Create a singleton blue c {v}.
- FindSet(v) Return the name of the b that contains node v.
- Union(A,B) Replace the two component sets A and B by A∪B.

- This data structure is studied in EECS 4101 and [CLRS 21].
- O(E) such operations on V elements take O(E α(E,V)) time in total.
- α is inverse of Ackermann's function and is very very slow growing.
  On practical instances $\alpha < 5$ (but it eventually increases to ∞).

Algorithm KruskalMST(G)

```
1.  for each vertex v ∈ V(G) do MakeSet(v)          § initialize blue forest
2.  SORT edges E(G) in non-decreasing order of weight  § all edges white
3.  T ← ∅                                           § MST blue edges

4.  for each edge (u,v) ∈ E(G), in sorted order do   § min weight white edge
5.      A ← FindSet(u)                  § find blue components that contain u & v
        B ← FindSet(v)
6.      if A ≠ B then do                § if A = B, then color (u,v) red
7.          T ← T ∪ {(u,v                          is short bridge of A (& B)
8.          Union(A,B)                             A & B
9.      end-if
10. end-for

11. return T                            § The MST blue edges
end
```


Step 2 Sort: $O(E \log E) = O(E \log V^2) = O(E \log V)$ time.

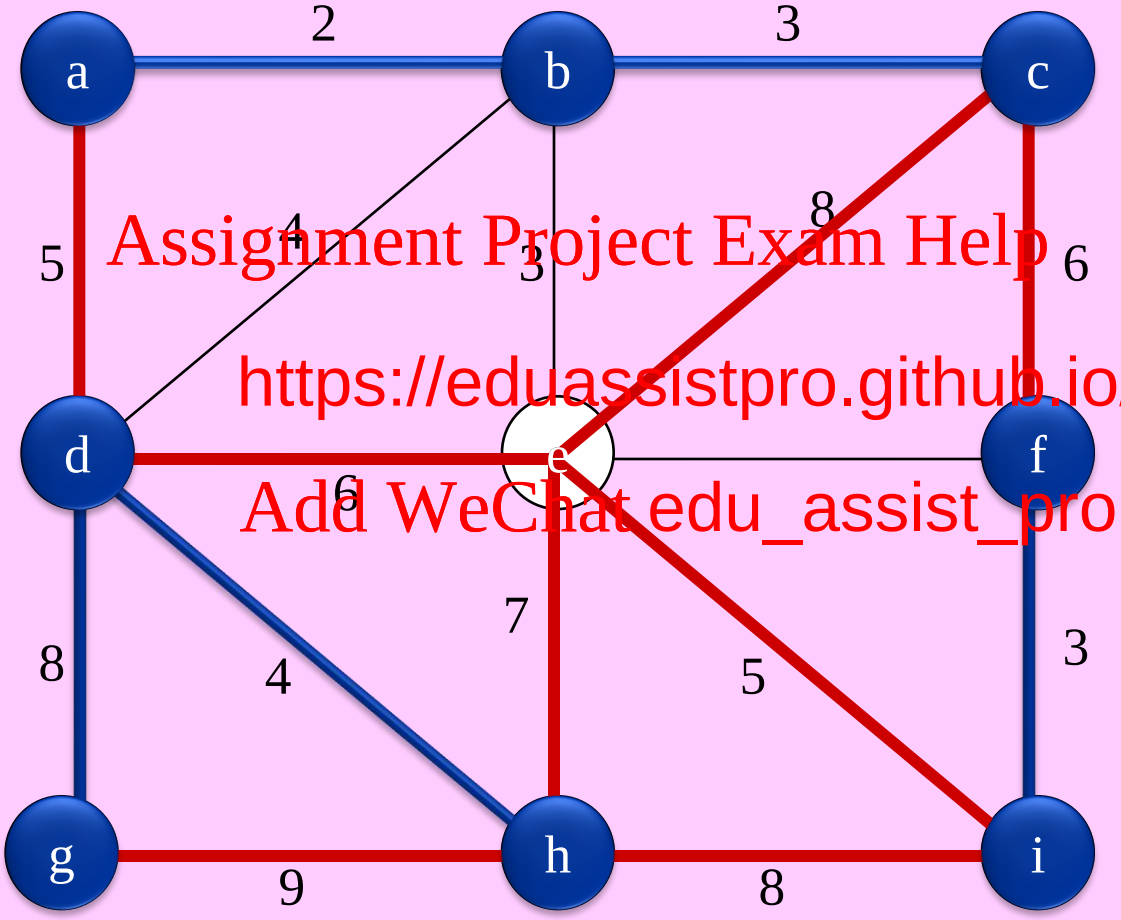
Disjoint Set Union operations: V MakeSets, 2E FindSets, V–1 Unions: $O(E\alpha(E,V))$ time.

Total Time = $O(E \log V)$.

Example

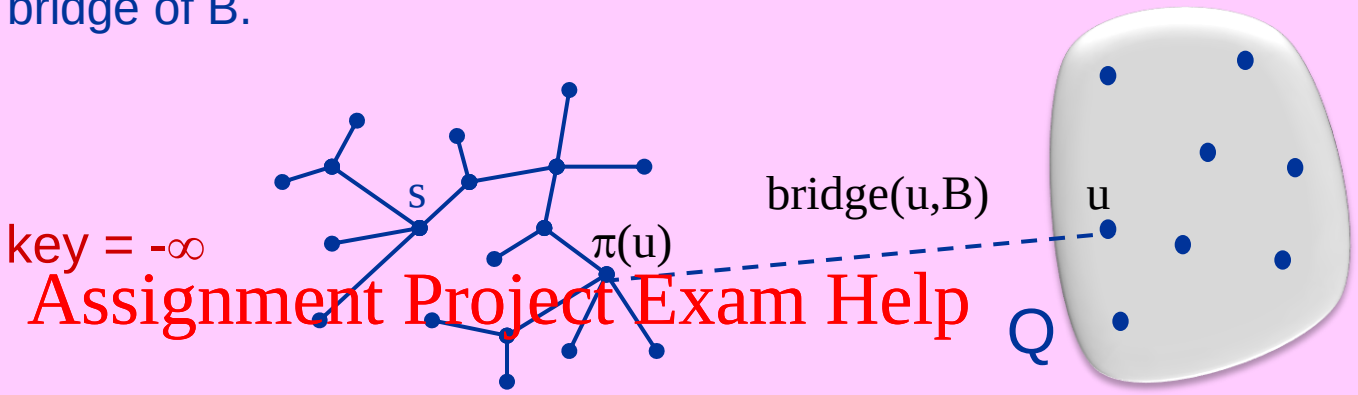
Edges sorted by weight

2	ab
2	ef
3	bc
3	be
3	fi
4	bd
4	dh
5	ad
5	ei
6	cf
6	de
7	eh
8	ce
8	dg
8	hi
9	gh



Prim's MST Algorithm

- B = blue component that contains the selected starting node s.
- e = the short bridge of B.
- Color e blue.



Each blue component ode in Q = V(G) – B.

Q = min priority queue of nodes.

For each u ∈ Q define:

bridge(u,B) = min weight white edge incident to both u and B

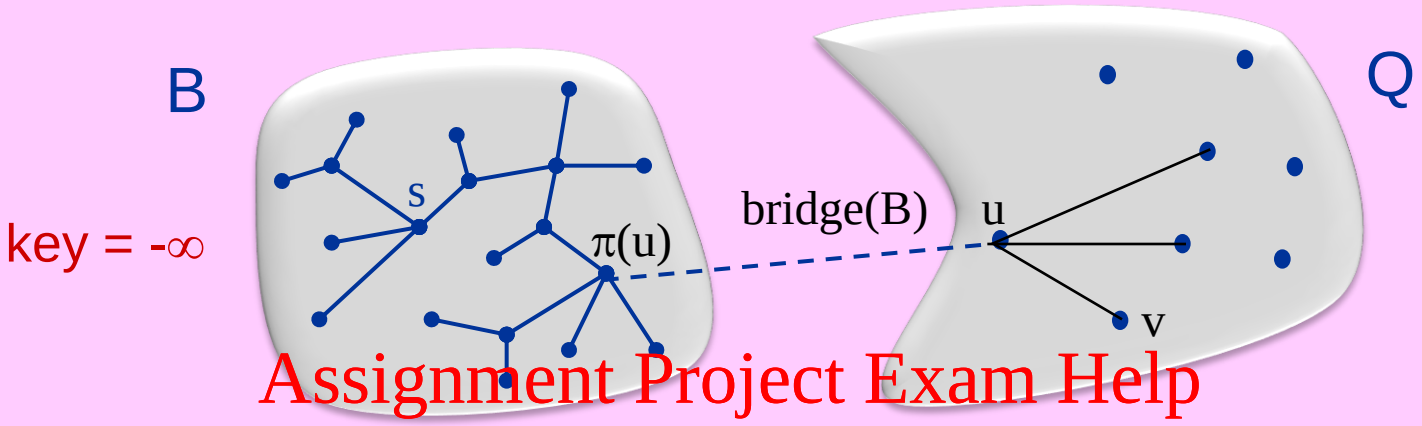
π(u) = the end node of bridge(u,B) in B (nil if no such edge)

key[u] = priority of u = w(bridge(u,B)) = w(π(u), u) (+∞ if no such edge)

$u^* = \operatorname{argmin}_u \{ w(\text{bridge}(u,B)) \mid u \in Q \}$

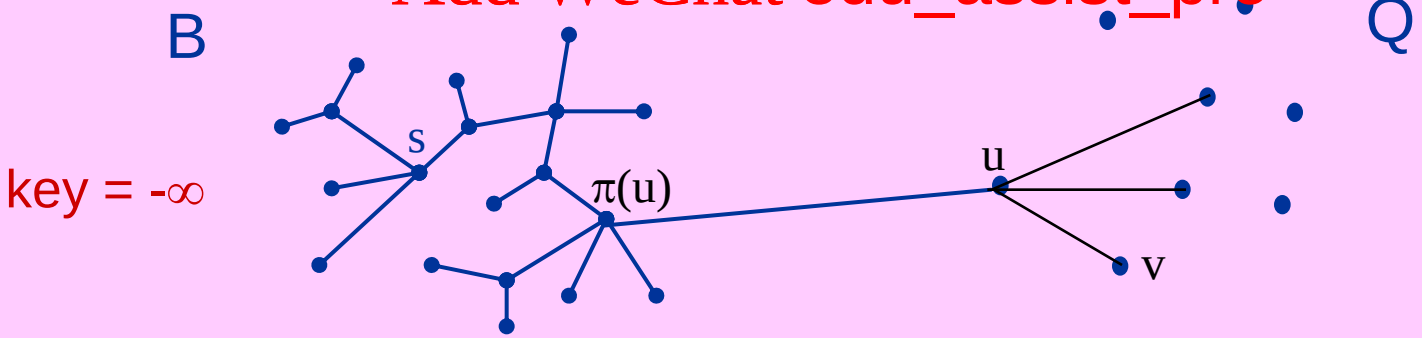
e = bridge(B) = (π(u*), u*) = the short bridge incident to B.

One Prim Step



teMin(Q)

- ∞



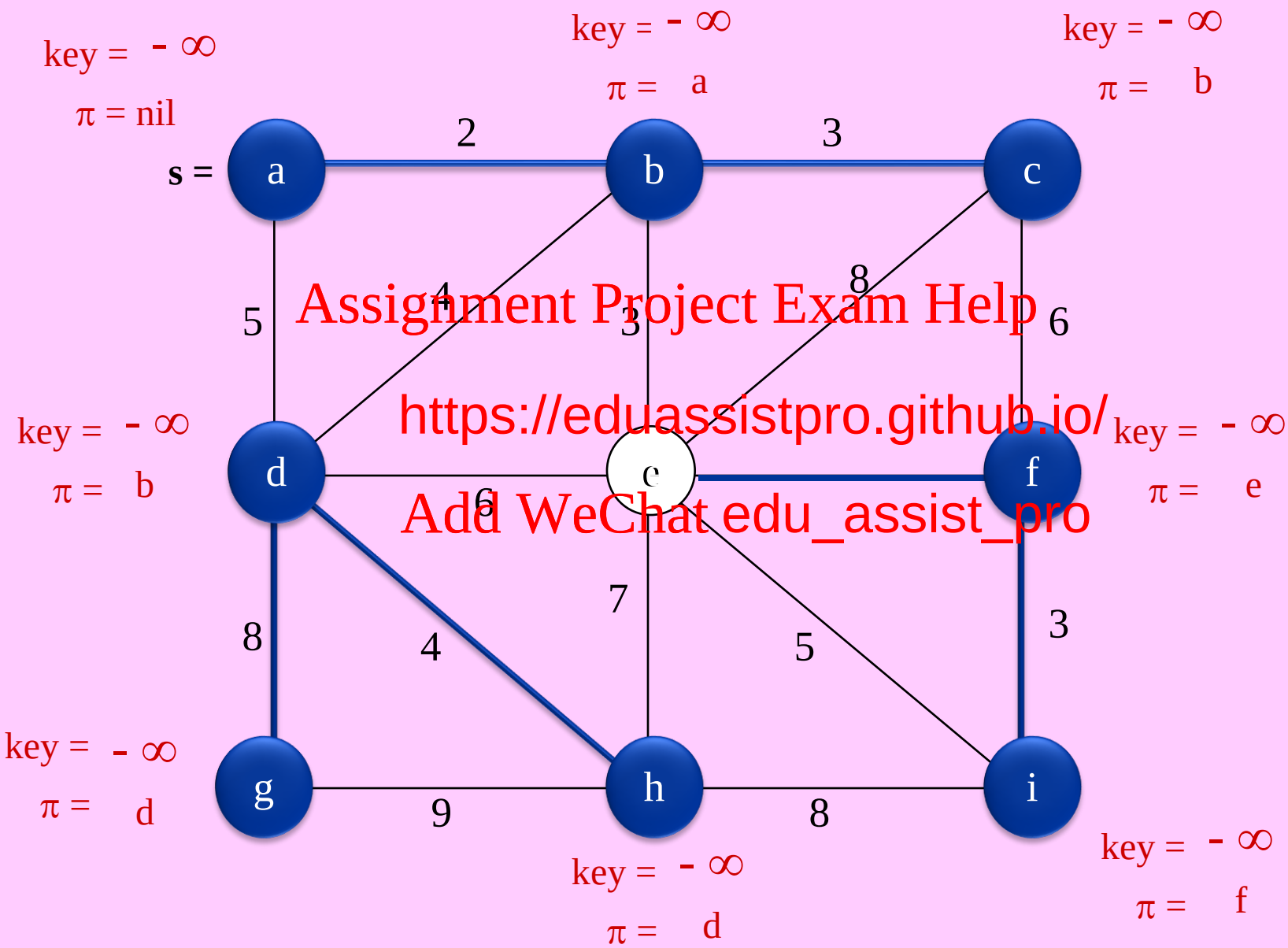
Update key[v] , π[v] ; and UpHeap(v,Q) , for each v ∈ Adj[u]∩Q

Algorithm PrimMST(G , s∈V(G))

```
1.   for each vertex u ∈ V(G) do key[u] ← +∞
2.   key[s] ← 0 ;   π[s] ← nil                    § first node to be selected
3.   Q ← ConstructMinHeap(V(G), key)              § key[u] = priority(u)
4.   while Q ≠ ∅ do                               § |V| iterations
5.      u ← DeleteMin(Q);   key[u] ← -∞           § O(log V) time
6.      for each v ∈ Adj[u] do
7.         if key[v] > w(u,v) then do             § key[v] ≠ -∞
8.            key[v] ← w(u,v)
9.            π[v] ← u                           § O( |Adj[u]| log V) time
10.           UpHeap(v,
11.        end-if
12.     end-for
13.  end-while
14.  T ←  { (u, π[u]) |  u ∈ V(G) –{s} }          § MST edges
15.  return T
end
```


Total time = { O(E log V) with standard binary heap
O(E + V log V) with Fibonacci Heaps (EECS 4101 & [CLRS 20]).

Example



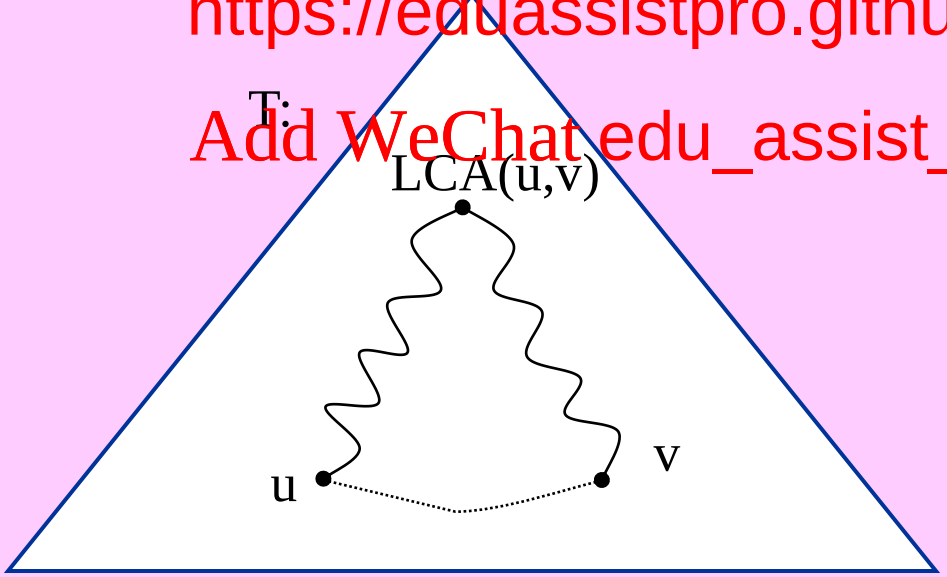
MST Verification

FACT: Let T be a spanning tree of G = (V,E,w).

T is an MST of G if and only if

for every edge (u,v) $\in$ E – T,

w(u,v) $\geq$ weight of every edge along the unique simple path in T between u & v.



Bibliography: MST

- **Kruskal [1956]** (Borůvka [1926]): O(E log V) time
- **Prim[1957]** (Jarník [1930]): O(E log V) , O(E + V log V) with Fib.Heap
- **Cheriton-Tarjan [1976]**
  O(E log log V) time on general graphs,
  O(V) time on planar graphs with arbitrary edge weights
  [studied in EECS 6114]

AAW

- **Karger, Klein, Tarjan [** dmized MST algorithm
- **Chazelle [2000]:** O(E α(E,V)) time deter lgorithm

- **Pettie-Ramachandran [2002]:** Decision tr T algorithm

- **Buchsbaum et al. [2006], King [1997]**: O(E) time MST verification algorithm.
- **Euclidean MST** of n points in the plane:
  O(n log n) time via Voronoi-Delaunay structures [studied in EECS 6114]

P.T.O.

Bibliography: MST

- A.L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R.E. Tarjan, J.R. Westbrook, "Linear-time pointer-machine algorithms for path evaluation problems on trees and graphs," **arXiv:cs.DS/0207061v2**, October 2006. [An earlier version in STOC'88, pp: 279-288.]
- B. Chazelle, "The Soft Heap: an approximate priority queue with optimal error rate," J. ACM, 47(6):1012-1027, 2000.
- B. Chazelle, "A minimum spanning tree algorithm with inverse Ackermann type complexity," J. ACM, 47(6):1028-1047, 2000.
- D. Cheriton, R.E. Tarjan "F
SIAM J. Computing (5):72
- H. Kaplan, U. Zwick, "A simpler implementation a zelle's Soft Heaps," SODA, pp:477-485, 2009.
- D.R. Karger, P.N. Klein, R.E. Tarjan, "A randomiz rithm to find minimum spanning trees," J. ACM (42):321-328, 1995.
- V. King, "A simpler linear time algorithm for minimum spanning tree verification," Algorithmica, 18: 263-270, 1997.
- S. Pettie, V. Ramachandran "An optimal minimum spanning tree algorithm," J. ACM 49(1):16-34, 2002.

SHORTEST PATHS

Well, ya turn left by the fire station in the village and take the old post road by the reservoir and... no, that won't do.
Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on won't work either.
East Millinocket, here from here.

Bert and I and Oth n East (1961)

Hey farmer! Where does this road go?
Been livin' here all my life, it ain't gone nowhere yet.
Hey farmer! How do you get to Little Rock?
Listen stranger, you can't get there from here.
Hey farmer! You don't know very much do you?
No, but I ain't lost.

Michelle Shocked, *"Arkansas Traveler"* (1992)

The Shortest Path Problem

- Given **weighted digraph** G = (V,E,w) with w: E $\longmapsto$ $\Re$ (possibly < 0).
- **Cost or distance** of a path P in G: **d(P)** = sum of weights of edges on P.
- **A shortest (s,t) path:** a path from vertex s to vertex t in G, whose cost is minimum among all paths in G from s to t.
- **Shortest Path Problem:** Find a shortest path from s to t in G for each member (s,t) of a given collection of vertex pairs.
- **Four versions of the S**

Single-Pair: Find a shortest path from a a given vertex t in G.

Single-Source: Given a source vertex s, fin from s to v, $\forall v \in V(G)$.

Single-Sink: Given a sink vertex t, find a shortest path from v to t, $\forall v \in V(G)$.

All-Pairs: For every pair of vertices s and t, find a shortest path from s to t in G.

° Single-Source and Single-Sink are directional duals of each other.

° Single-Pair algorithms typically solve (at least partially) Single-Source or Single-Sink.

° All-Pairs can be solved by |V(G)| iterations of Single-Source.

° So, Single-Source is a fundamental problem on which we will concentrate.

° However, we will also discuss All-Pairs separately.

Single-Source Shortest Paths

Negative Cost Cycles

FACT 1: Suppose node t is reachable from node s.
∃ a shortest path from s to t if no path from s to t contains a negative cost cycle.
If there is any shortest path from s to t, there is one that is simple.

Proof:
If some s-t path contains a negative cycle, we can produce an arbitrarily short s-t path by repeating the cycle enough times.

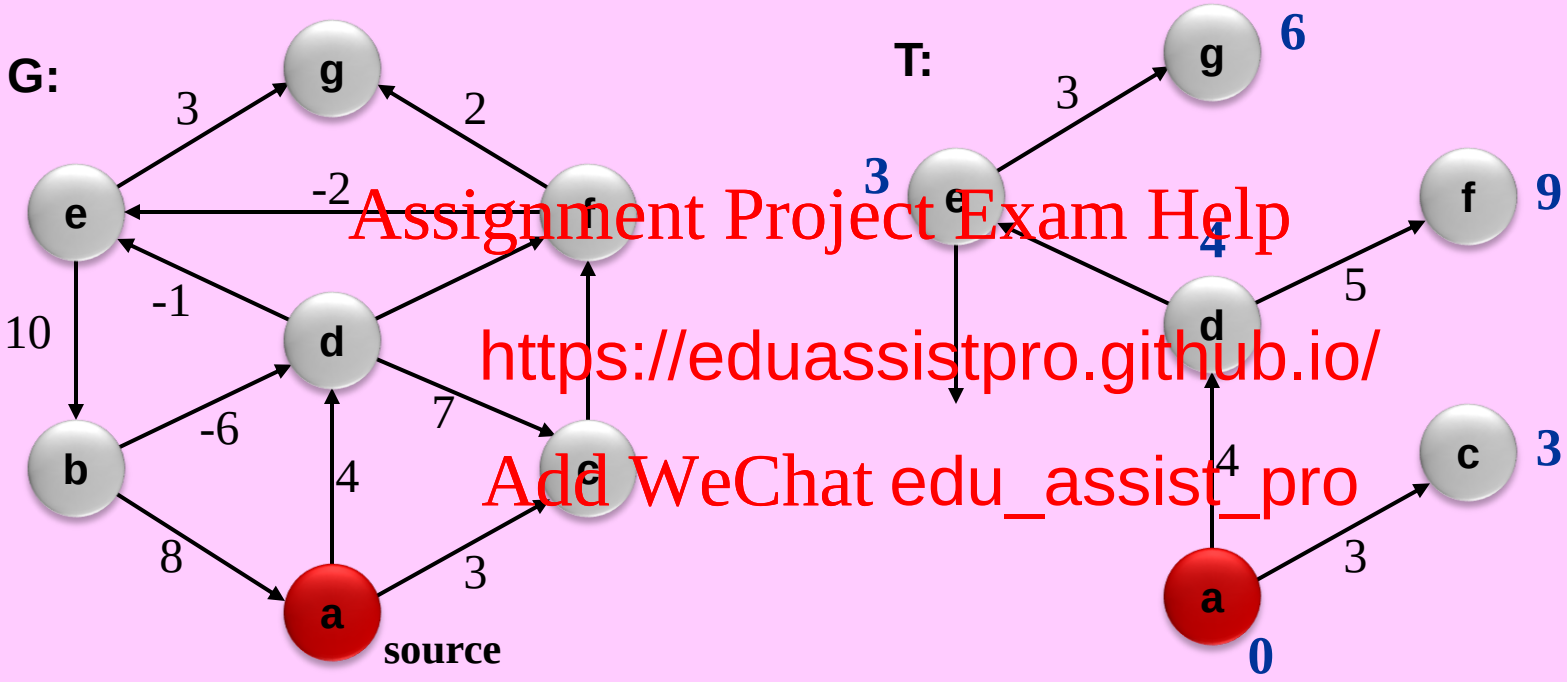
If no s-t path contains a neg th simple, without increasing its cost, by removing (or by

In the presence of **negative cycles**, asking for a **simple** shortest path is NP-Complete.

Optimum Sub-Structure Property (OSSP) of dynamic programming **does NOT hold**.

Shortest Path Trees

Shortest Path Tree T: A compact way to represent single-source shortest paths. This is a tree rooted at the source s and spans all nodes reachable from s, all of whose paths are shortest paths in G.



FACT 2: G contains shortest paths from source s to all reachable nodes if and only if G contains a shortest path tree rooted at s.

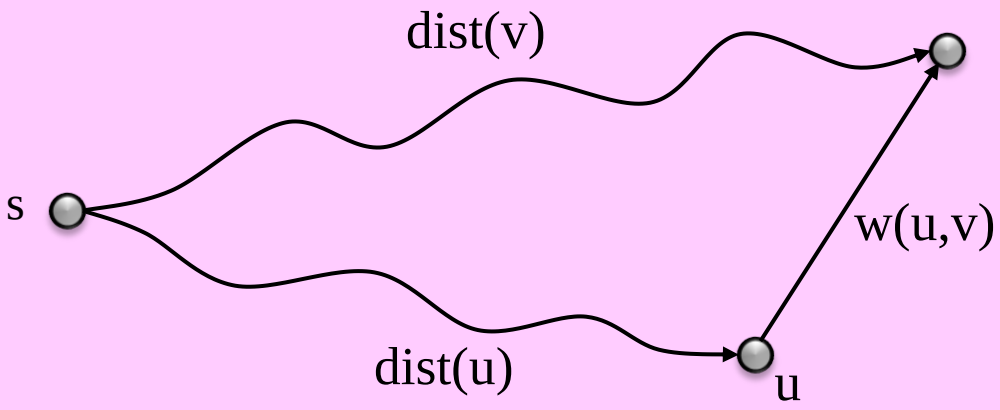
Proof: Completed later, by giving an algorithm that finds either a shortest path tree or a negative cycle reachable from s.

Characterizing Shortest Path Trees

T = a subgraph of G that is a tree rooted at source s and spans all nodes reachable from s.
dist(v) = cost of the unique path in T from s to v.

FACT 3: T is a shortest path tree if and only if for every edge (u,v) of G,
dist(u) + w(u,v) ≥ dist(v). [Verifiable in O(V+E) time.]

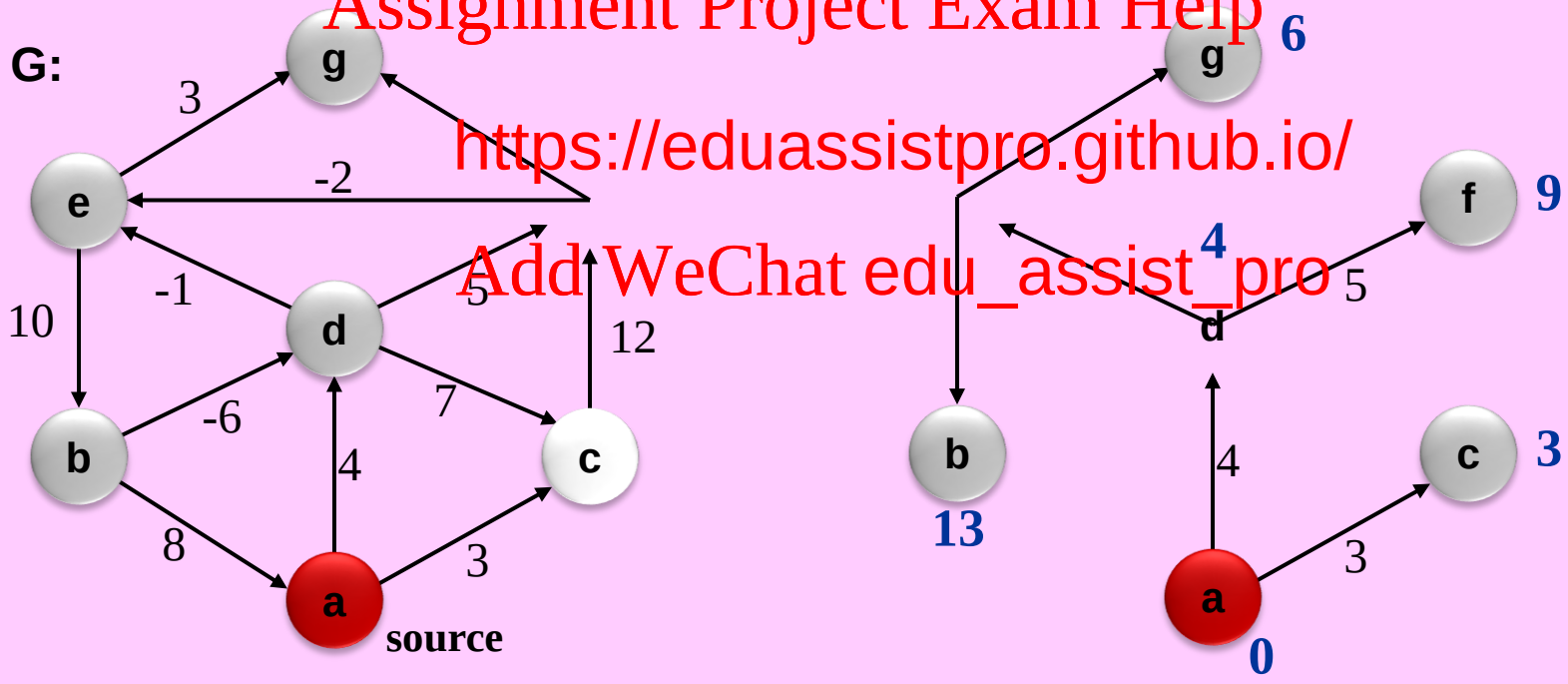
Proof: dist(u) + w(u,v) < dist(v) ⟹ the path in T from s to v is not shortest.
Conversely, suppose dis y edge (u,v) in G.
Let P be any path from n the # edges on P, we
can show d(P) ≥ dist(v). e.



Characterizing Shortest Path Trees

T = a subgraph of G that is a tree rooted at source s and spans all nodes reachable from s.
dist(v) = cost of the unique path in T from s to v.

FACT 3: T is a shortest path tree if and only if for every edge (u,v) of G,
dist(u) + w(u,v) ≥ dist(v). [Verifiable in O(V+E) time.]



The Recurrence [Bellman 1958]

T = shortest path tree rooted at s.
dist(v) = cost of the unique path in T from s to v.
π(v) = parent of v in T (nil if no parent).
R(s) = the set of nodes in G that are reachable from s.

FACT 4: G has a shortest path tree rooted at s if and only if the followin

1. $\pi(s) = nil$
   $dist(s) = 0$

2. $\forall v \in R(s) - \{s\}$:
   $\pi(v) = \text{argmin}_u \{ dist(u) + w(u,v) \mid (u,v) \in E(G) \}$
   $dist(v) = \min \{ dist(u) + w(u,v) \mid (u,v) \in E(G) \} = dist(\pi(v)) + w(\pi(v), v)$

3. $\forall v \in V(G) - R(s)$:
   $\pi(v) = nil$
   $dist(v) = +\infty$.

Labeling Method [Ford 1956]

Tentative shortest path tree **dist** and π. Initialization and iterative improvement:

Initialize:

$dist(s) = 0$

$dist(v) = +\infty \quad \forall v \in V(G) - \{s\}$

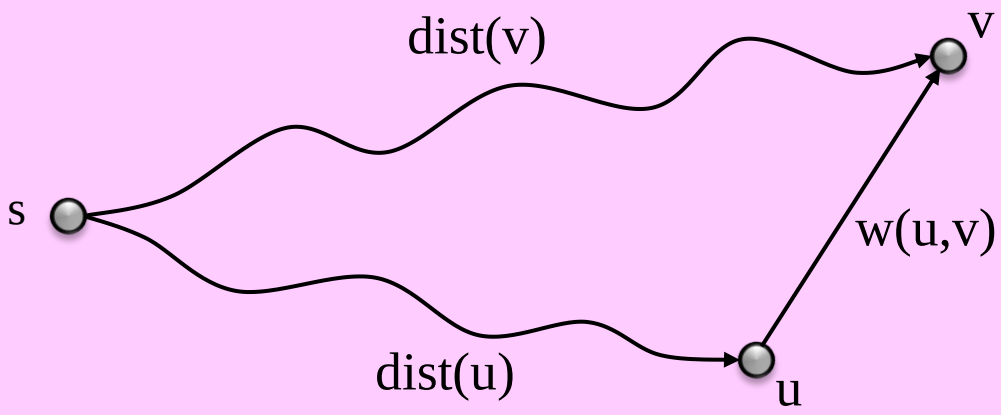
$\pi(v) = nil \quad \forall v \in V(G)$

Labeling Step:

Select an edge (u,v) with $dist(u) + w(u,v) < dist(v)$.

Label v: $dist(v) \leftarrow dist(u) + w$

$\pi(v) \leftarrow u$.



[CLRS] calls this a Relaxation Step.

Labeling Method Invariants

FACT 5:

Ford's labeling method maintains the following invariants for any $v \in V(G)$:

(1) If dist(v) is finite, then there is an s-v path of cost dist(v).

(2) If $\pi(v) \neq$ nil, then $dist(\pi(v)) + w(\pi(v), v) \leq dist(v)$ remains true, and is met with equality when the method terminates.

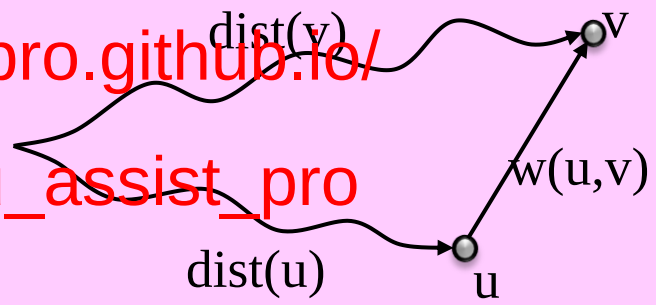
(3) If P is any s-v path, then $dist(v) \leq$ dist(v) when Ford's method terminates.

Proof sketch:

(1) By induction on #

(2) By induction on # labeling steps.

(3) By induction on # edges on P.



FACT 6:

(1) When the labeling method terminates,
dist(v) = cost of a shortest s-v path, if v is reachable from s, and
dist(v) = ∞ otherwise.

(2) If there is a negative cycle reachable from s, the method never terminates.

Tentative Shortest Path Tree Invariants

Define: $\pi^0(v) = v$, $\pi^{k+1}(v) = \pi(\pi^k(v))$ for any vertex v and any integer $k \geq 0$.

That is, $\pi^0(v), \pi^1(v), \pi^2(v), \ldots$ are v, parent of v, grand parent of v, …

FACT 7:

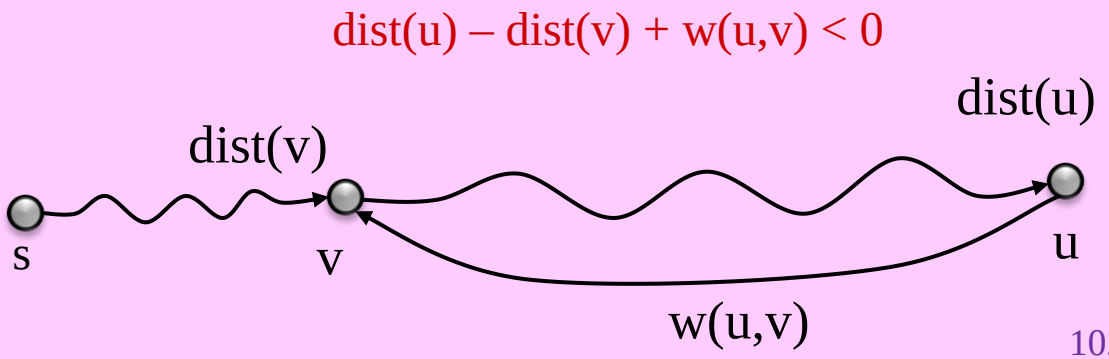
(1) Ford's labeling method maintains the invariant that either the edges $(\pi(v), v)$, for v such that $\pi(v) \neq$ nil, form a tree r uch that dist(v) < ∞, or there is a vertex v .

(2) If at some time during the labeling meth $\pi^k(v) = v$ for some vertex v and integer k then the corresponding cycle in G has ne

Proof sketch:

(1) By ind. on # labeling steps.

(2) Consider a labeling step that creates such a cycle of parent pointers:



Termination?

FACT 8:

If and when Ford's labeling method terminates, the parent pointers define a shortest path tree from the source s to all vertices reachable from s.

- If there is a negative cycle reachable from s, the method never terminates.
- Even if it does termin ly many labeling steps (an upper bound of $2^{|E|}$

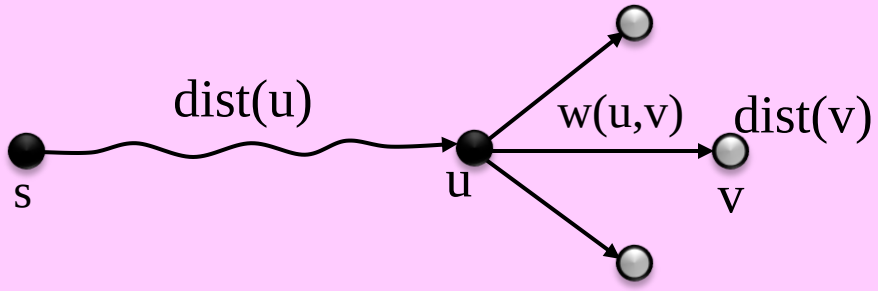
- Instead of verifying that the general label minates, we shall study efficient refinements of it.

Labeling & Scanning Method

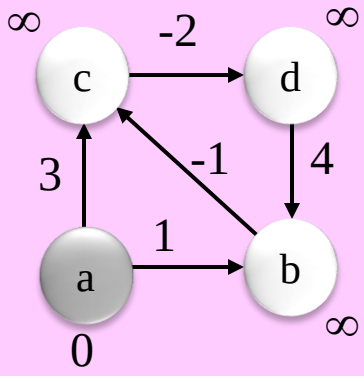
- This first refinement eliminates unnecessary edge examinations.
- Maintains a partition of vertices into 3 states: **unlabeled, labeled, scanned**.
- The labeled or scanned vertices are exactly those with finite dist values.
- The possible state transitions are:

- **Initialization:** s is labeled and every o nlabeled.

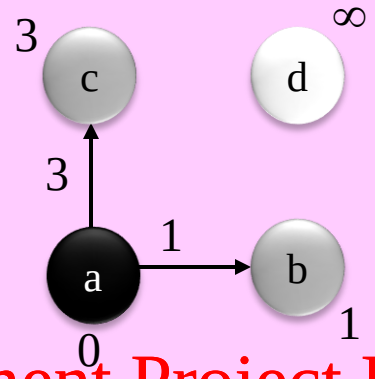
- **Scanning Step:** Select a labeled vertex u and scan it, thereby converting it to the scanned state, by applying the labeling step to every edge (u,v) such that dist(u) + w(u,v) < dist(v), thereby converting v to the labeled state.



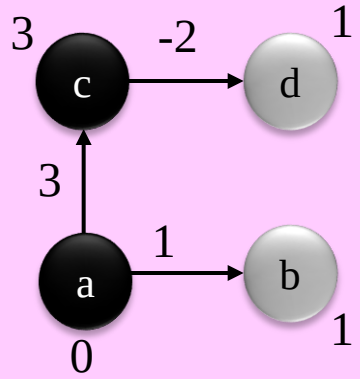
Example



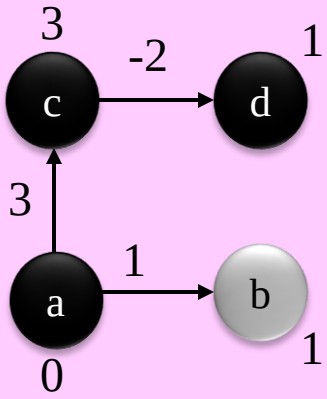
(0) Initial



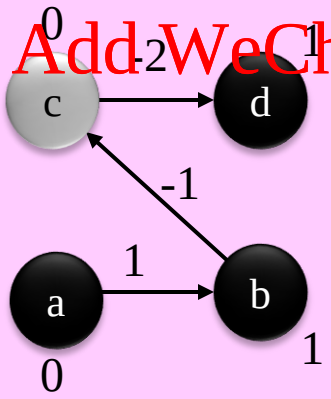
(1) Scan a



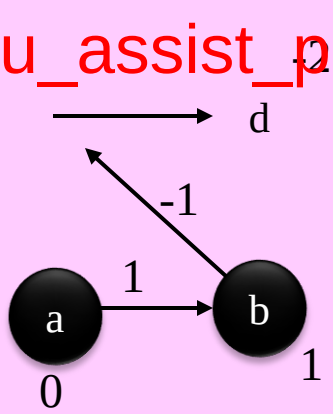
(2) Scan c



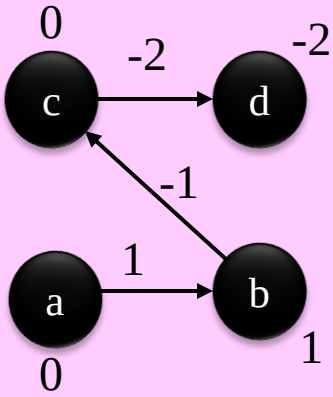
(3) Scan d



(4) Scan b



(5) Scan c



(6) Scan d

No labeled vertex remains. Scanning & Labeling Method terminates.

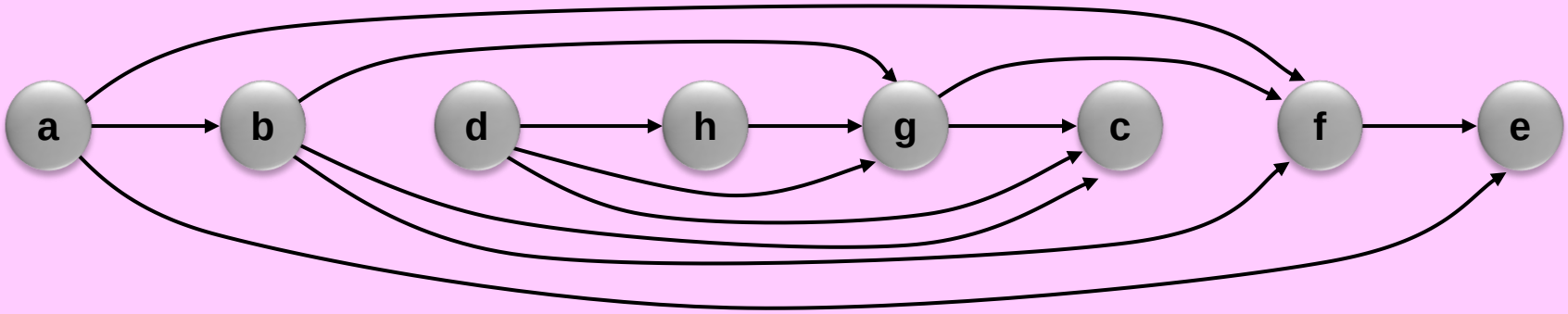
Efficient Scanning Orders

Three cases considered:

1. [DAG]: G is a Directed Acyclic Graph,
2. [$w \geq 0$]: G may have cycles but all edge weights are non-negative,
3. [General]: G may have cycles and edge weights may be negative.

We will consider these 3 …… P.T.O.

1. G is a DAG [Topological]

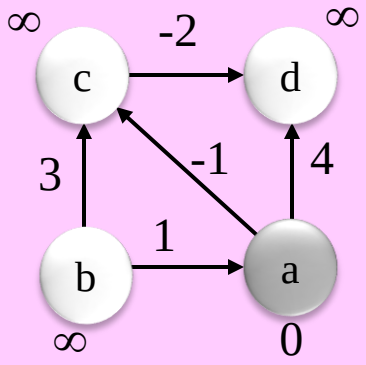


- Topological order is an appropriate scanning order.
- Order the vertices that at if u→v is an edge of G, u appears before v in t

- As we have learned, we can topologically of DAG G in O(V+E) time by doing a DFS of G and ordering th erse-postorder.

- Then apply the scanning step once to each vertex in topological order.
- Since the scanning order is topological, once a vertex is scanned, it never returns to the labeled state.
- So, one scan per vertex reachable from s suffices.
- The running time of the algorithm is $O\left(\sum_{u \in V(G)} (1 + |\text{Adj}[u]|)\right) = O(V + E)$.

Example

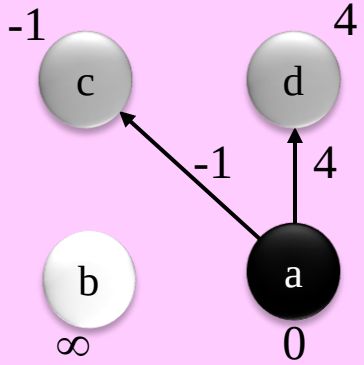


(0) Initial

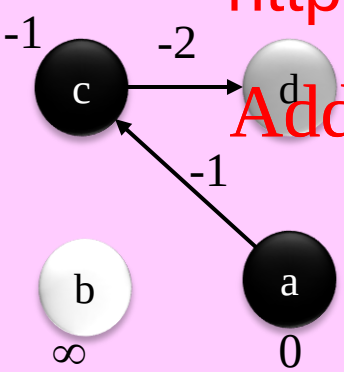
source = a.

Topological Order: b, a, c, d.

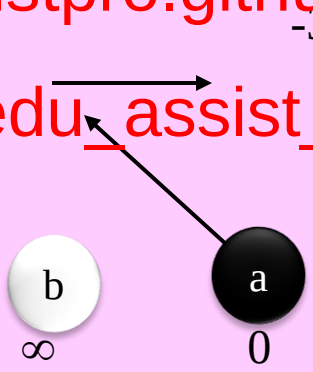
b is not reachable from a.



(1) Scan a



(2) Scan c



(3) Scan d

No labeled vertex remains. Scanning & Labeling Method terminates.

2. $w \geq 0$ [Dijkstra]

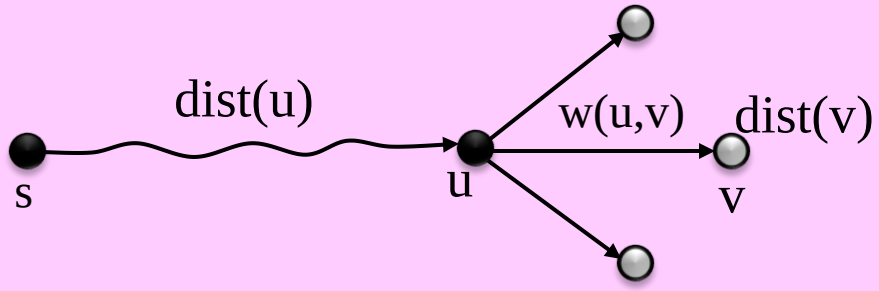
- One scan per vertex also suffices if G is arbitrary (with possible cycles) but has no negative weight edges.
- **Dijkstra [1959]** proposed the appropriate **GREEDY** scanning order:

  Shortest First: Among labeled vertices, always scan one whose tentative distance is **minimum**.

FACT 9:

Suppose every edge in
Scanning shortest first
for every scanned vertex u and every non
u) ≤ dist(v)
led/unlabeled) vertex v.

Proof sketch: By induction on the # scanning steps,
using non-negativity of edge weights.



2. w ≥ 0 [Dijkstra]

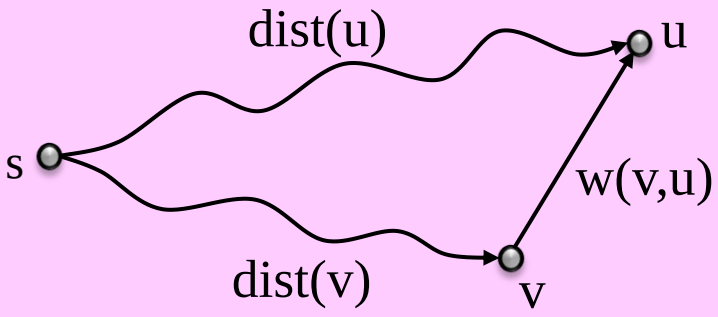
FACT 10:

Suppose every edge in G has non-negative weight.
Scanning shortest first maintains the invariant that:
once a vertex u is scanned, dist(u) is the cost of a shortest path from s to u.

Proof:

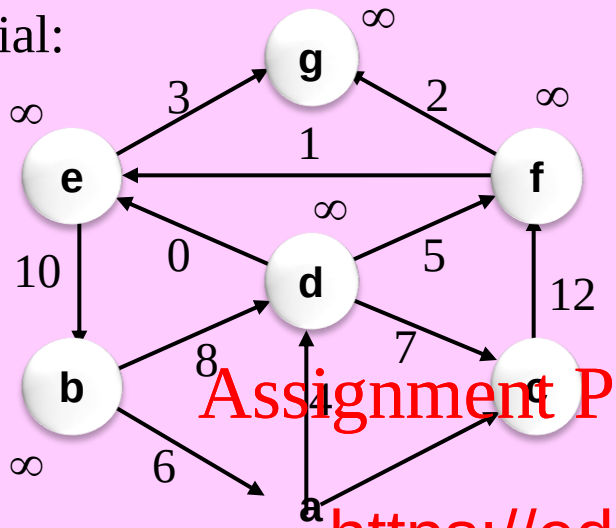
By Fact 9: dist(u) ≤ di
for every s scanned vertex v.

This means that vertices are scanned in non-d r by shortest distance from s, and that a vertex, once scanned, due to non-negativity of edge weights, cannot become labeled.

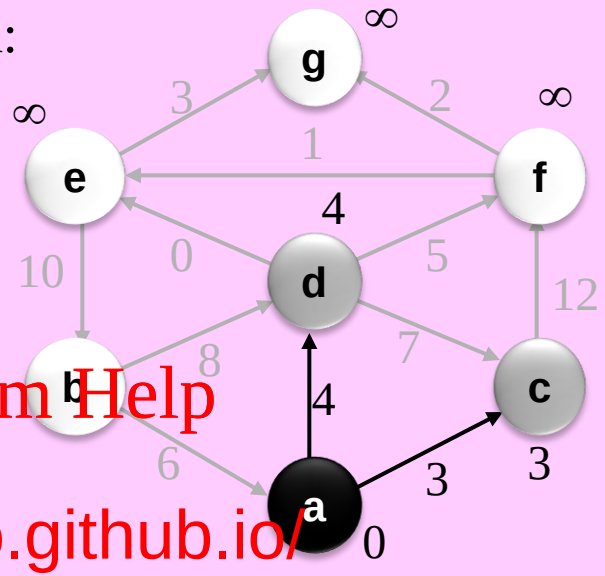


Example

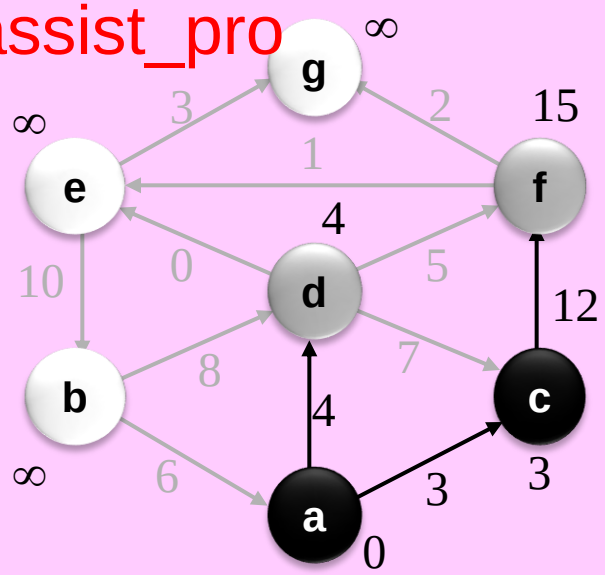
0) Initial:



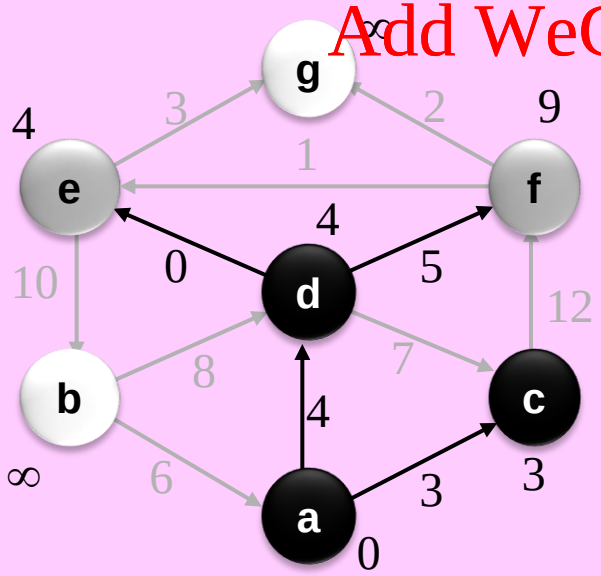
1) Scan a:



2)

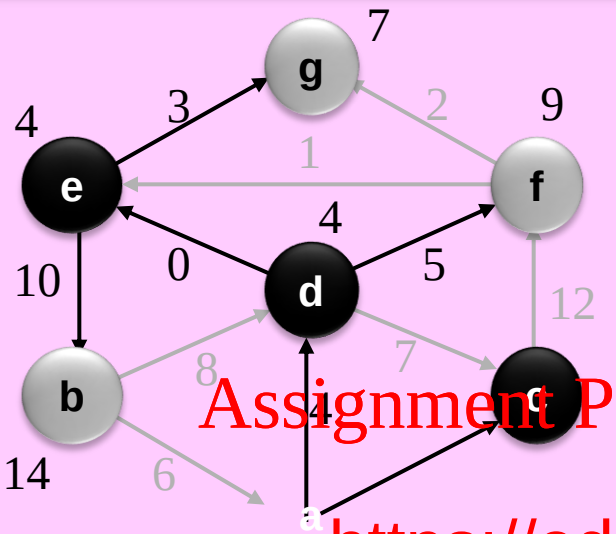


3) Scan d:

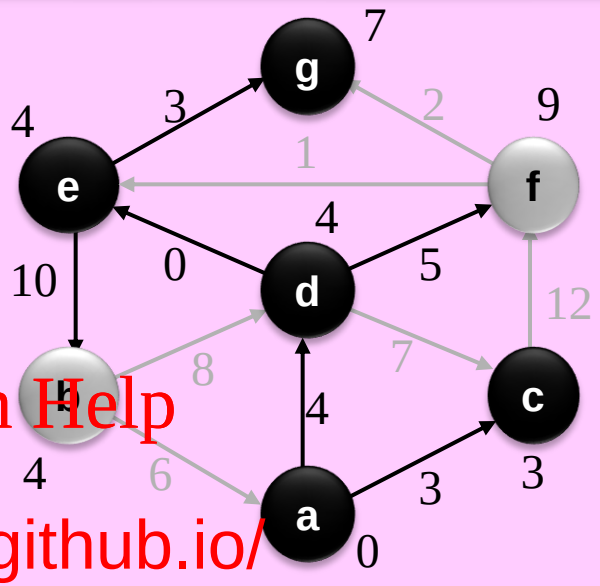


Example

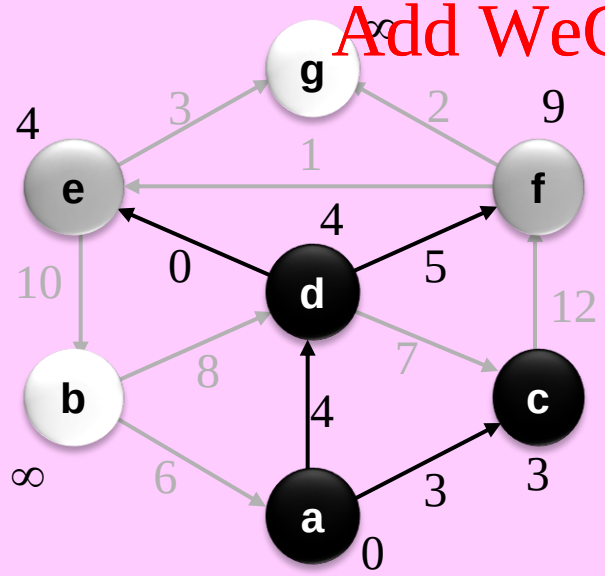
4) Scan e:

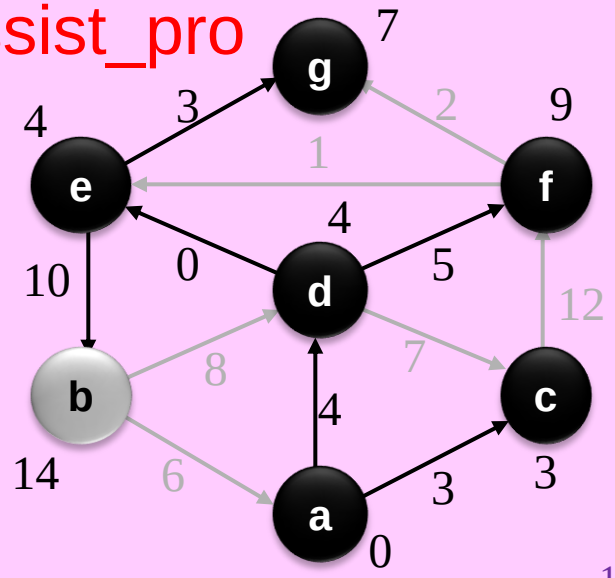


5) Scan g:



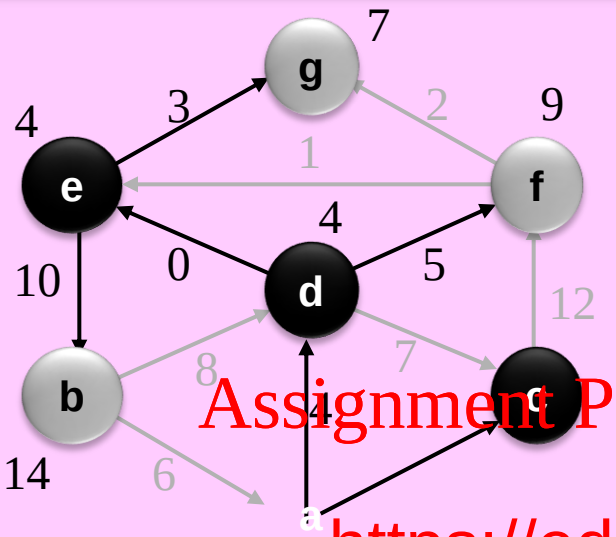
3) Scan d:



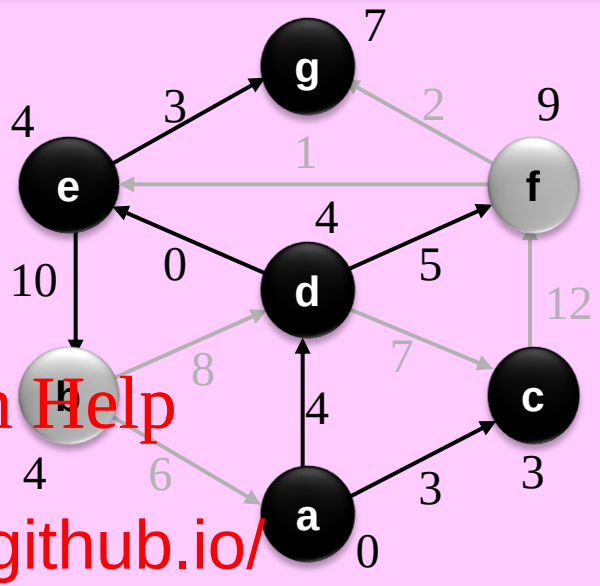


Example

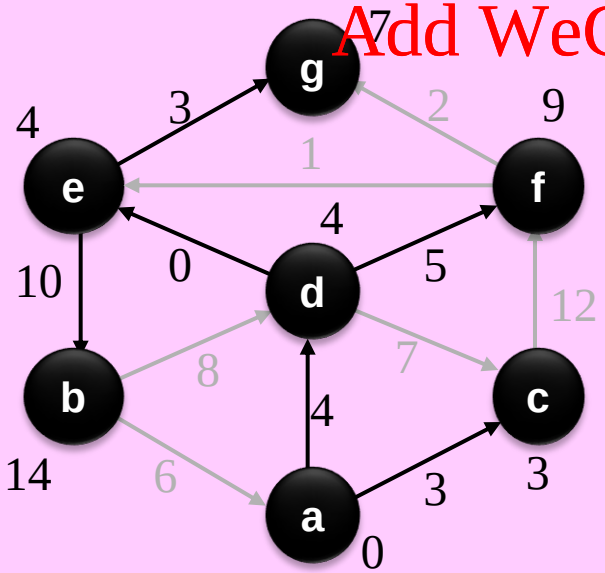
4) Scan e:



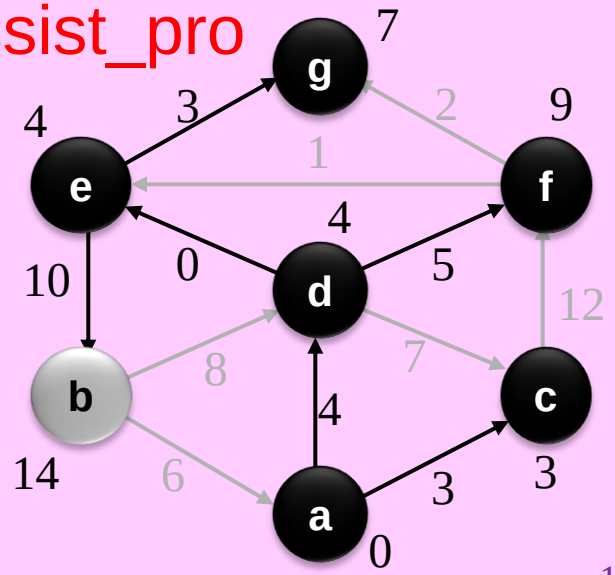
5) Scan g:



7) Scan b:



DONE



Algorithm Dijkstra (G , s∈V(G))

```
1.   for each vertex u ∈ V(G) do dist[u] ← +∞
2.   dist[s] ← 0 ;  π[s] ← nil                          § first node to be scanned
3.   Q ← ConstructMinHeap(V(G), dist)                  § dist[u] = priority(u)
4.   while Q ≠ ∅ do                                    § |V| iterations
5.      u ← DeleteMin(Q)                               § O(log V) time
6.      for each v ∈ Adj[u] do
7.          if dist[v] > dist[u] + w(u,v) then do      } O( |Adj[u]| log V) time
8.             dist[v] ← d
9.             π[v] ← u
10.            UpHeap(v,
11.         end-if
12.     end-for
13.  end-while
end
```


Total time = { O(E log V) with standard binary heap ; O(E + V log V) with Fibonacci Heap

```
Procedure PrintSP(u)
      if  u = nil  then return
      PrintSP(π[u])
      print u
end                          § O(V) time
```

3. General [Bellman-Ford]

- In this 3rd and last case, G is a general digraph (possibly with cycles) and some edge weights may be negative.
- **Bellman [1958]** proposed the appropriate **GREEDY** scanning order:
  Breadth First: Among labeled vertices, scan the one least recently labeled.

- A looser interpretation
  We divide the labeling some arbitrary order by we check every edge in them if applicable.

- By induction on the # edges on an s-v path s, we can show that by the end of stage k, dist(v) ≤ d(P).
- Any simple path has at most |V(G)| – 1 edges.
  So, if there is a shortest path tree, it will be found by the end of stage |V(G)| –1.

  If in stage |V(G)| a labeling step is still applicable,
  we conclude there must be a negative cycle reachable from s.

Algorithm BellmanFord (G , s∈V(G)) **§ O(VE) time**

```
1.   for each vertex u ∈ V(G) do dist[u] ← +∞
2.   dist[s] ← 0 ;   π[s] ← nil

3.   for stage ← 1 .. |V(G)| –1 do
4.      for each edge (u,v) ∈ E(G), in arbitrary order do
5.         if dist[v] > dist[u] + w(u,v) then do
6.            dist[v] ← dist[u] + w(u,v)
7.            π[v] ←
8.         end-if
9.      end-for
10.  end-for

11.  for each edge (u,v) ∈ E(G), in arbitrary order do      § stage |V(G)|
12.        if dist[v] > dist[u] + w(u,v) then
13.           return there is a negative cycle reachable from s
end
```


Exercise: If line 13 detects existence of a negative cycle, find one such cycle in O(V) time.

All-Pairs Shortest Paths

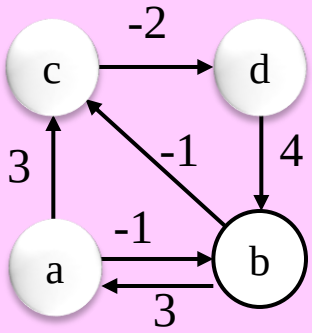
All-Pairs by Single-Source

- All-Pairs Shortest Paths: |V| iterations of Single-Source Shortest Paths.
- If all edge weights are non-negative, |V| Dijkstra iterations.
  Time = O(VE log V) improved to O(VE + V^2 log V) by Fibonacci Heap.
- If some edge weights may be negative, |V| Bellman-Ford iterations.
  Time = $O(V^2E)$.
- **Edmonds-Karp [1972** improved method:
  Preprocessing tran eserve shortest pa
  make all ed
  [How? For details, see the next slide.]

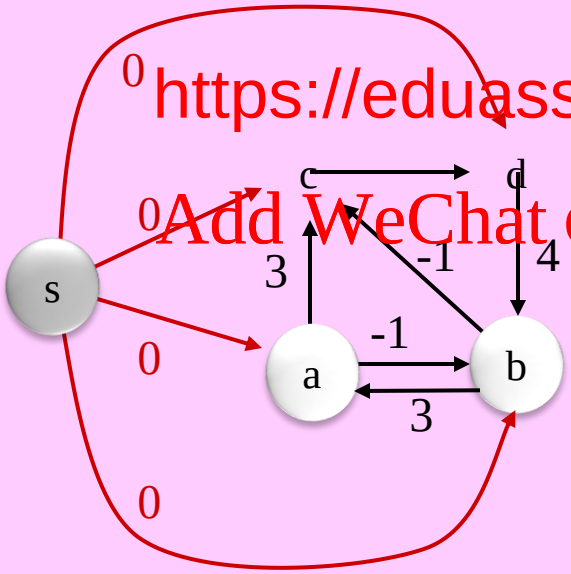
  - Preprocessing: Bellman-Ford in O(VE) time.
    Afterwards, apply |V| iterations of Dijkstra.
    Total time of O(VE + V^2 log V).

Edmonds-Karp & Johnson Transformation

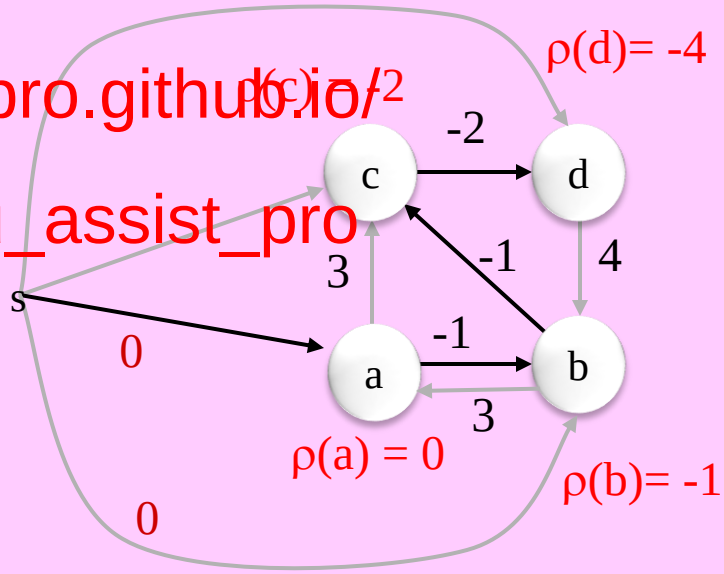
1. Start with the given weighted digraph G = (V, E, w).
2. Add to G a new vertex $s \notin V$ and a 0 weight edge from s to each vertex $v \in V$.
3. Apply **Bellman-Ford** to the augmented graph with source s:
   either find shortest path distances, denoted **ρ(v)**, from s to each $v \in V$,
   or discover a negative cycle in G. In the latter case abort.



G



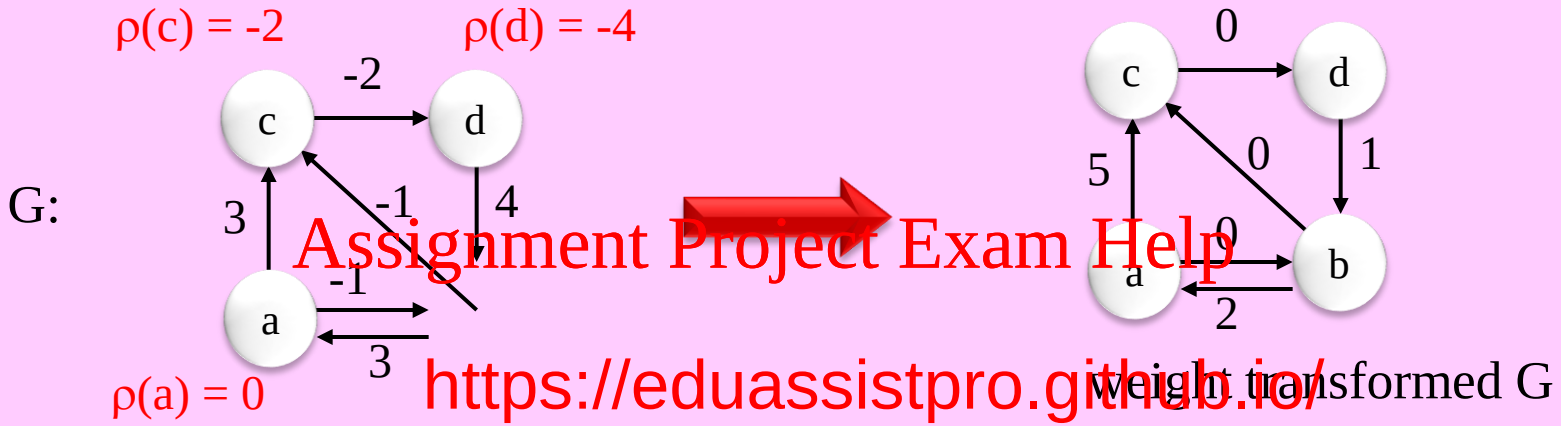
Augmented G



Single-source shortest dist in Augmented G

Edmonds-Karp & Johnson Transformation

- Transform edge weights: $\hat{w}(u,v) = w(u,v) + \rho(u) - \rho(v)$ for all $(u,v) \in E(G)$.



- **Property 1:** $\forall (u,v) \in E(G)$ we have:

  $\hat{w}(u,v) \geq 0$ (since $\rho(v) \leq w(u,v) + \rho(u)$)

- **Property 2:** For any path P from any vertex u to any vertex v in G:

  $\hat{d}(P) = d(P) + \rho(u) - \rho(v)$

  where $\hat{d}(P)$ is the cost of path P according to the modified edge weights $\hat{w}$.

- This means shortest paths are preserved under the modified edge weights, since cost of every path from u to v is shifted by the same amount $\rho(u) - \rho(v)$.

Floyd-Warshall [1962]

- **Floyd** modified transitive closure algorithm of **Warshall** to obtain a simplified $O(V^3)$-time algorithm for all-pairs shortest paths based on dynamic programming.
- Let $V(G) = \{1,2, \ldots, n\}$. For every pair of nodes u, v define:

  $d^{(k)}(u,v)$ = Minimum cost among all those paths from node u to node v whose intermediate nodes (excluding u and v) are in {1..k}.
  That is, the te node is at most k.

  $\pi^{(k)}(u,v)$ = predecesso

- **FACT:** $d^{(n)}(u,v)$ = Shortest path cost fro

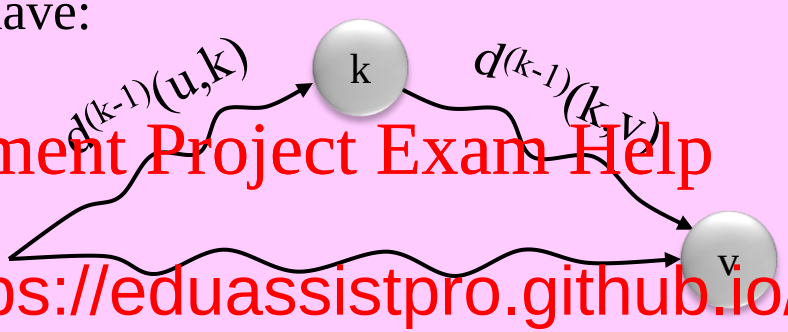
  $d^{(0)}(u,v)$ = A[u,v] , where A is the weighted adjacency matrix (assume no self-loops exist).

- DP: How to go from $d^{(k-1)}$ to $d^{(k)}$, for k = 1,2, …, n?

Floyd-Warshall [1962]

- DP: How to go from $d^{(k-1)}$ to $d^{(k)}$, for k = 1,2, …, n?
- **If $d^{(k-1)}(k,k) < 0$**, then there is a **negative cycle** & k is the max node of it. Abort.
- **Otherwise:** for all u, v we have:

OSSP holds:



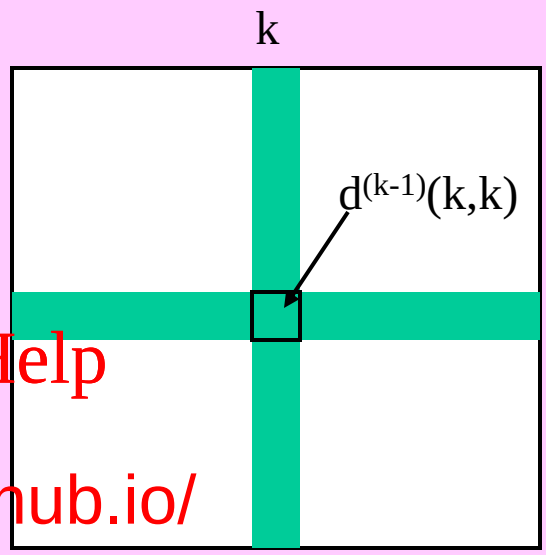
$$d^{(k)}(u,v) = \min \{ d^{(k-1)}(u,v), \; d^{(k-1)}(u,k) + d^{(k-1)}(k,v) \}$$

$$\pi^{(k)}(u,v) = \pi^{(k-1)}(u,v) \text{ OR } \pi^{(k-1)}(k,v) \quad k \neq v$$

Floyd-Warshall [1962]

$d^{(k)}(u,v) = \min\{ d^{(k-1)}(u,v),\ d^{(k-1)}(u,k) + d^{(k-1)}(k,v)\}$

$d^{(k-1)}(k,k) \geq 0$



$d^{(k)}(u,k) = \min\{ d^{(k-1)}(u,k),\ d^{(k-1)}(u,k) + d^{(k-1)}(k,k)\}$
$= d^{(k-1)}(u,k)$

$d^{(k)}(k,v) = \min\{ d^{(k-1)}(k,v),\ d^{(k-1)}(k,k) + d^{(}$
$= d^{(k-1)}(k,v)$

- This means row k and column k do not change in iteration k-1 $\rightarrow$ k.
- Since all other entries are updated based on their "old" values & row k and column k, we can re-use the same matrices d and π.
- That is, drop the **super-script** from these matrices.

Algorithm FloydWarshall (G) § $O(V^3)$ time, $O(V^2)$ space

```
1.   for each (u,v) ∈ V(G) × V(G) do                  § k = 0 (or nil)
2.       d[u,v] ← A[u,v]                              § weighted adj. matrix
3.       π[u,v] ← nil
4.   for each (u,v) ∈ E(G) do  π[u,v] ← u

5.   for each k ∈ V(G) do                             § O(V) iterations
6.       if d[k,k] < 0 th                                  cle
7.       for each (u,v) ∈                             § O(V²) iterations
8.           if   d[u,v]
9.                d[u,v] ← d[u,k] + d[k,v]
10.               π[u,v] ← π[k,v]                     § k ≠ v
11.          end-if
12.      end-for
13.  end-for
14.  return d and π
end
```

The Dynamic Programming Algorithm

- **Floyd-Warshall**'s algorithm is a special case of **Gauss-Jordan** elimination.

- **Gauss-Jordan** elimination has many other applications, including
  - Solving systems of linear equations
  - Converting a finite automaton into a regular expression
  - Doing global anal

- **Tarjan [1981]** showed all these problems treated together in an appropriate unified setting.

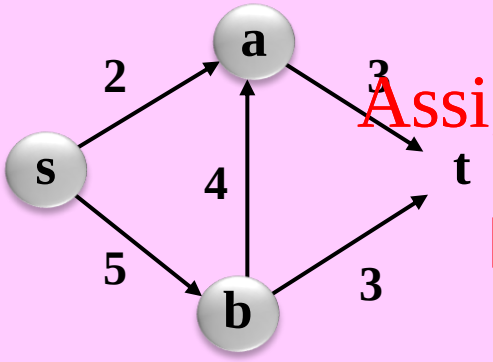
Bibliography: Shortest Paths

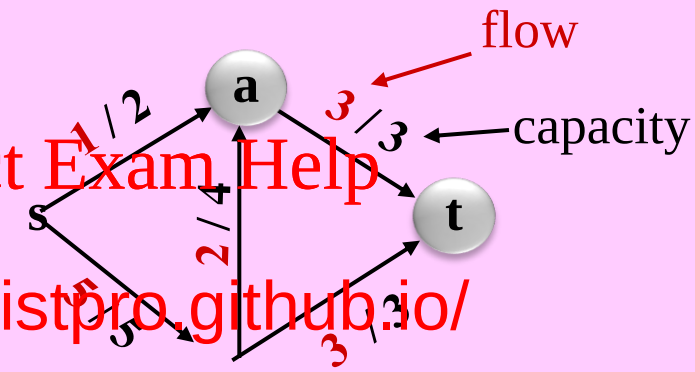
- R.E. Bellman "On a routing problem," Quart. Appl. Math. 16: 87-90, 1958.
- E.W. Dijkstra, "A note on two problems in connexion with graphs," Numer. Math.1:269-271, 1959.
- Jack Edmonds, R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," J. ACM 19:248-264, 1972.
- R.W. Floyd, "Algorithm 97: Shortest path," CACM 5:345, 1962.
- L.R. Ford, Jr., "Network flow theory," Paper P-923, RAND Corp., Santa Monica, CA., 1956.
- L.R. Ford, Jr., D.R. Fulkers niversity Press, 1962.
- D.B. Johnson, "Efficient al ivorks," JACM 24(1):1-13, 1977.
- P. Klein, S. Rao, M. Rauch, S. Subramanian, "Faste orithms for planar graphs," 26th STOC, pp: 27-37, 1994.
- R.E. Tarjan, "A unified approach to path problems," JACM 28: 577-593, 1981.
- M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," J. ACM 46(3):362-394, 1999.
- S. Warshall, "A theorem on Boolean matrices," JACM 9: 11-12, 1962.

Flow Network

Weighted digraph G = (V, E, c, s, t):

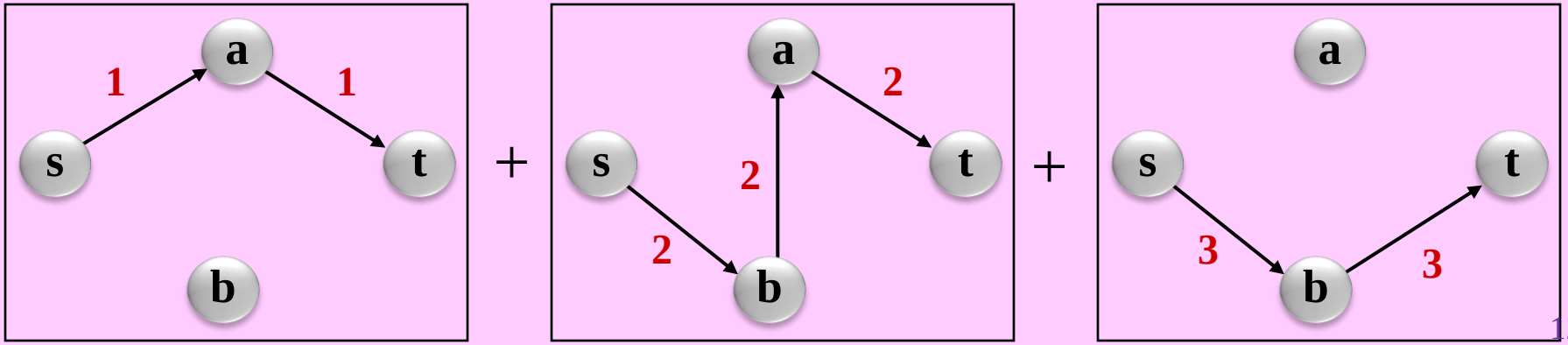
s = source node (in-degree = 0)
t = sink node (out-degree = 0)
edge capacity: c(e) > 0 $\forall e \in E$





M = 1 + 5 = 3 + 3 = 6

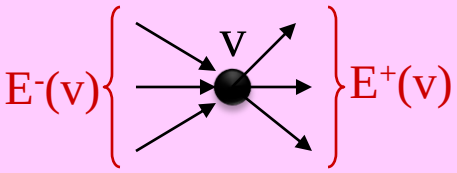
Flow Decomposition into s-t path flows:



The max Flow Problem

Notation: $E^+(v)$ = the set of edges out of vertex $v \in V$

$E^-(v)$ = the set of edges into vertex $v \in V$.



Flow Feasibility Constraints:

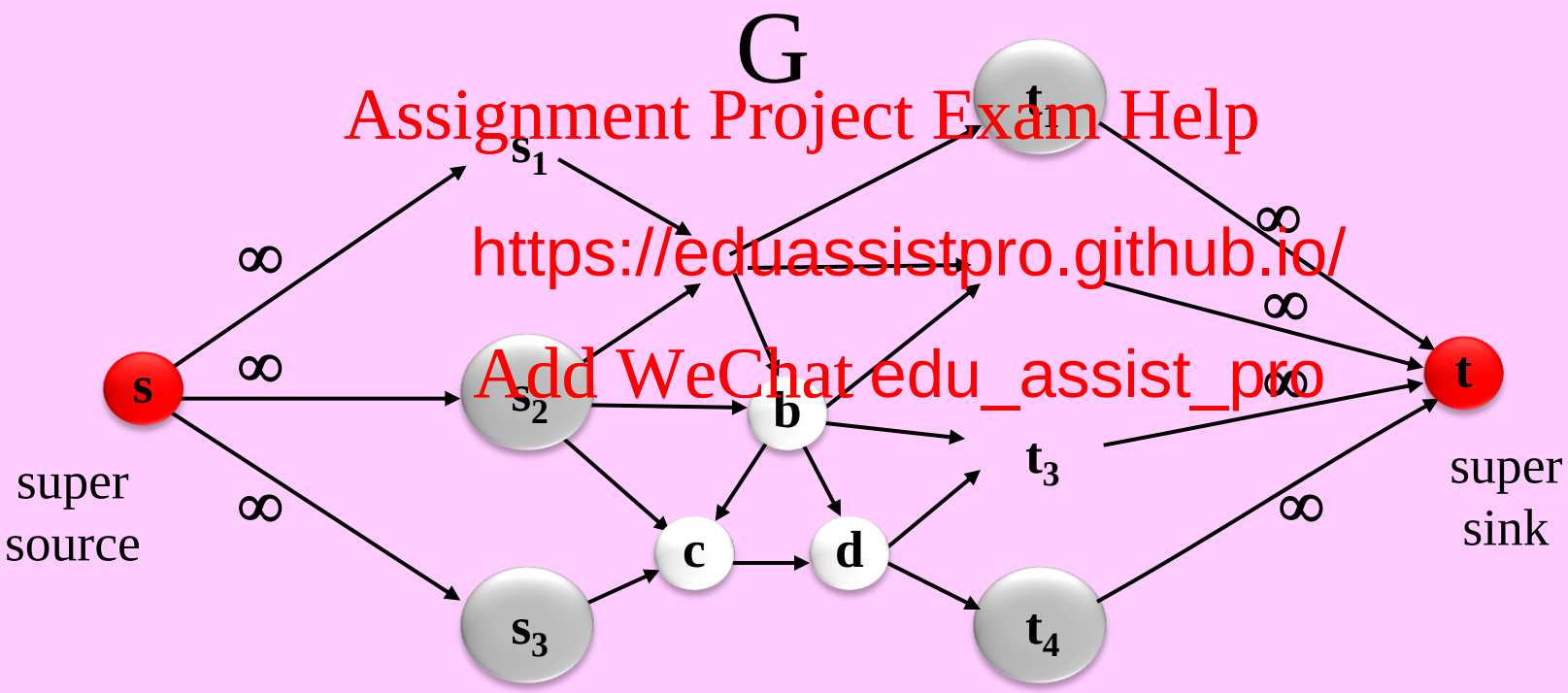
Capacity Con $0 \le f(e) \le c(e) \quad \forall e \in E$

Flow Conservation: $\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e) \quad \forall v \in V - \{s,t\}$

Maximize Flow value: $v(f) = \sum_{e \in E^+(s)} f(e) = \quad e \in E^-$

Multi-Source Multi-Sink

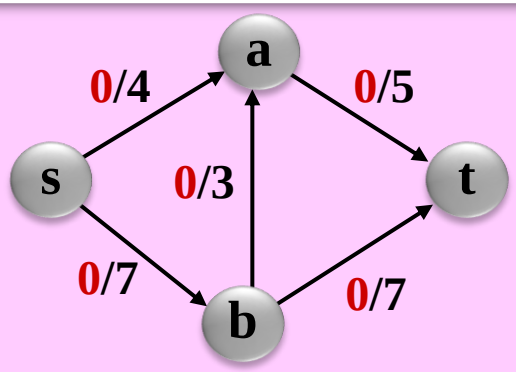
Transform to single-source single-sink:

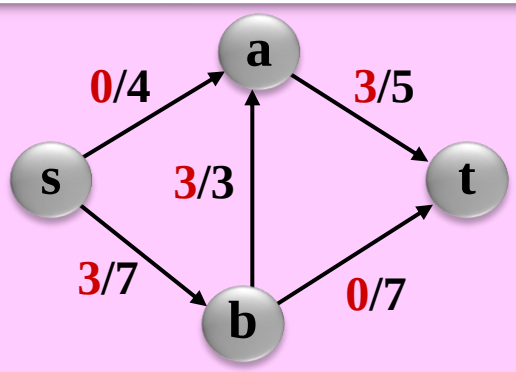


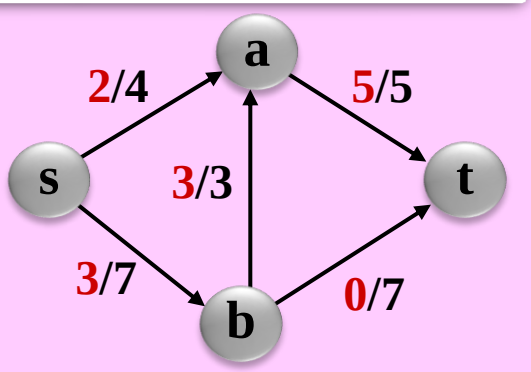
Applications

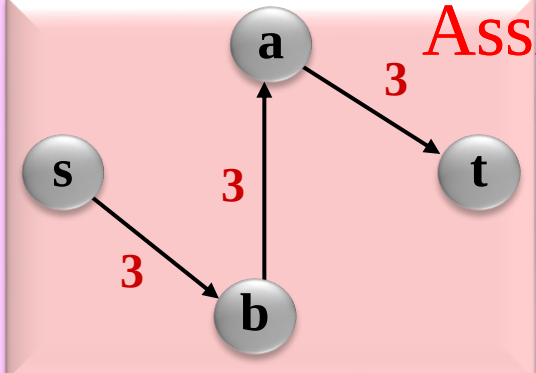
- Electrical circuits
- Hydraulic systems
- Traffic movements
- Te orks
- Fre
- Min-cost flow
- Graph Matching
- Matrix rounding
- …

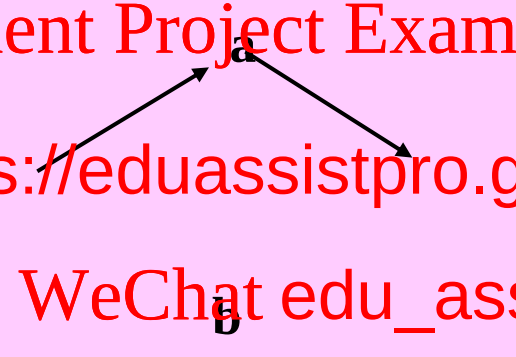
A Heuristic: flow composition by s-t path flows

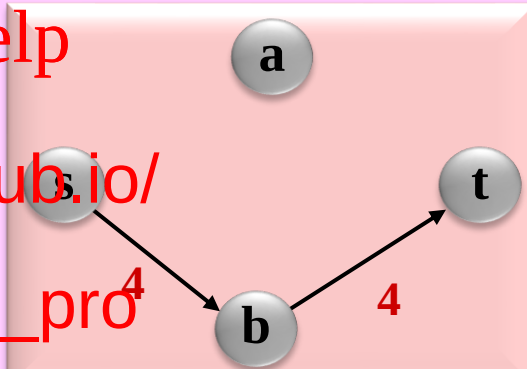


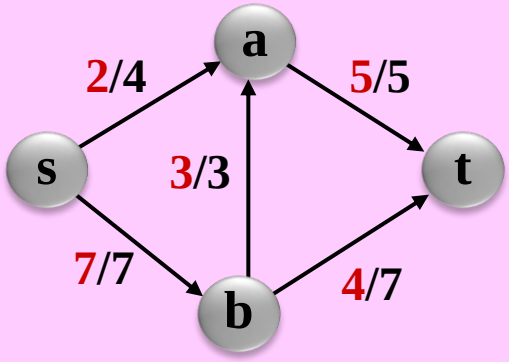






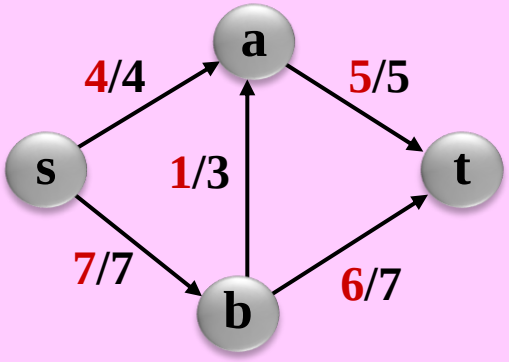






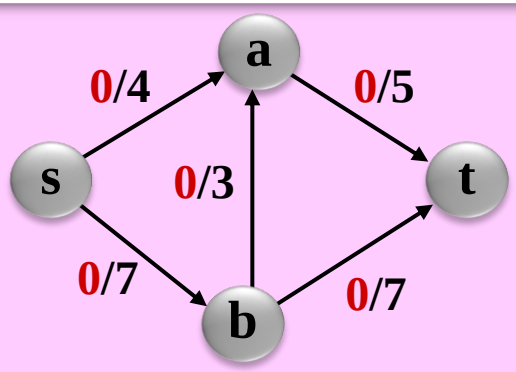
Flow value = 9

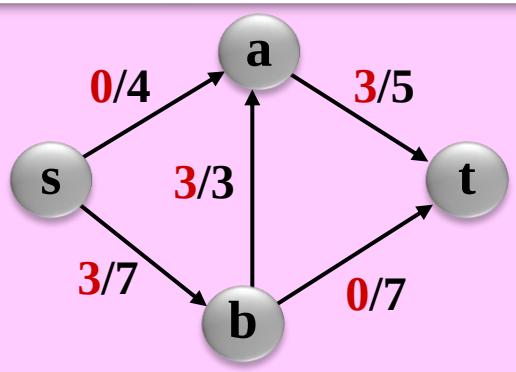
?

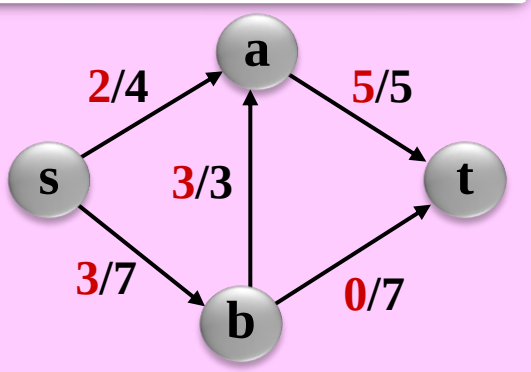


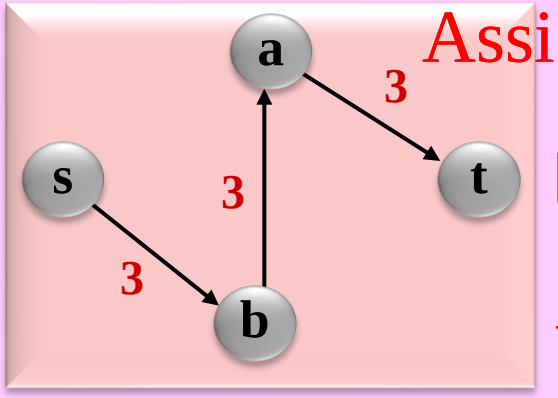
Max Flow value = 11

A Heuristic: flow composition by s-t path flows

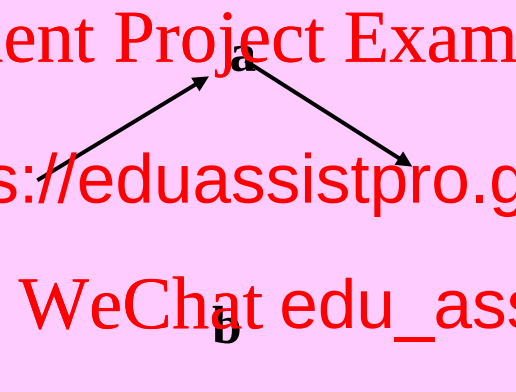




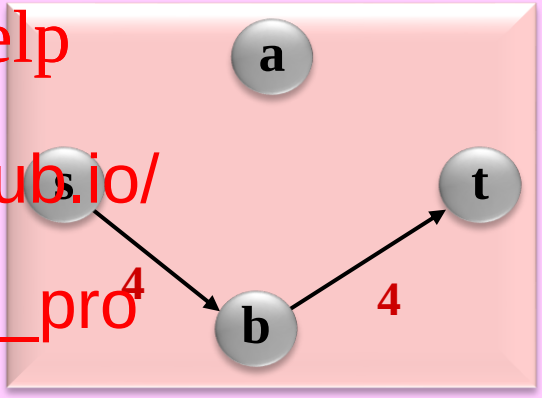


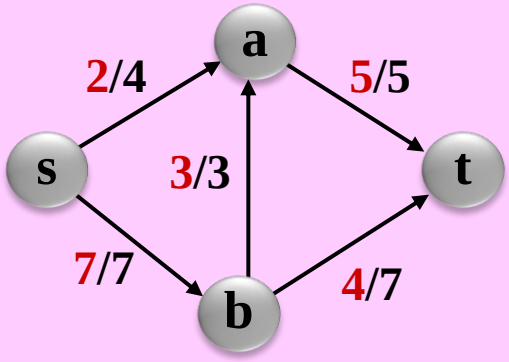


+



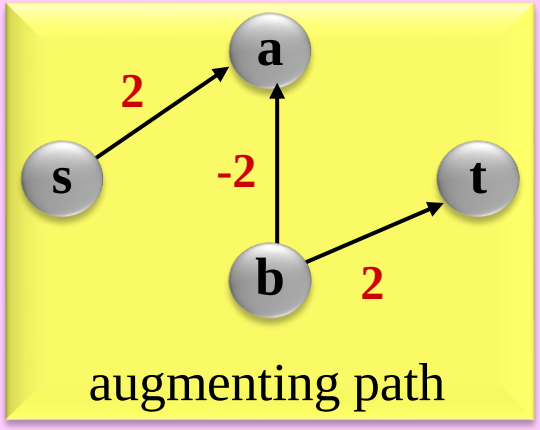
+



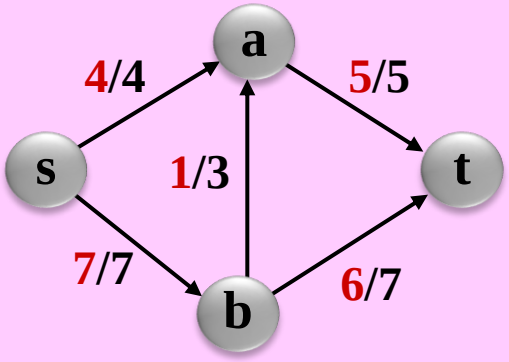


Flow value = 9

+



augmenting path



Max Flow value = 11

Augmenting Path

- f = a feasible flow in G
- p is an **augmenting path** with respect to flow f in G if:
  - p is an s-t path in G if we ignore edge directions,
  - every forward edge e on p is un-saturated, i.e., f(e) < c(e),
  - every backward edge e on p carries positive flow, i.e., f(e) > 0.

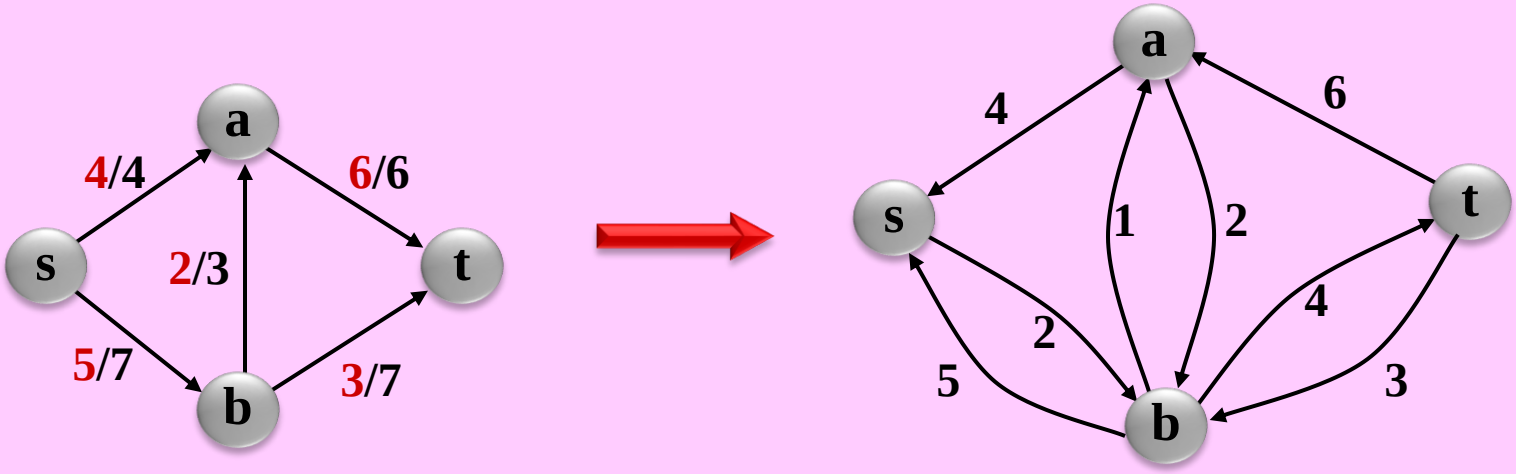
Residual capacity of "edge" (u,v) on p:

$\hat{c}(u,v) = c(u,v) - f(u,v)$ if (u,v) is a **forward** edge on p

$\hat{c}(u,v) = f(v,u)$ if (v,u) is a **backward** edge on p

The Residual Graph G(f)

To help find augmenting paths, we define **The Residual Graph G(f):**



An Augmenting Step

- There is an augmenting path in G wrt f ⇔ there is a directed s-t path in G(f).
- p = an augmenting path with respect to f in G.
- **bottleneck capacity of p:**

$$\Delta(p) = \min\left(\{\hat{c}(u,v) \mid (u,v) \in E(G) \text{ is a forward edge on } p\} \cup \{\hat{c}(u \quad) \quad (\quad) \quad (\quad) \quad \text{ward edge on } p\}\right)$$

- **Augment flow f along p:**

$$f(e) \leftarrow \begin{cases} f(e) + \Delta(p) & \text{if e is a forward edge on p} \\ f(e) - \Delta(p) & \text{if e is a backward edge on p} \\ f(e) & \text{otherwise} \end{cases} \qquad \forall e \in E(G)$$

Algorithm FordFulkerson(G=(V,E,c,s,t))

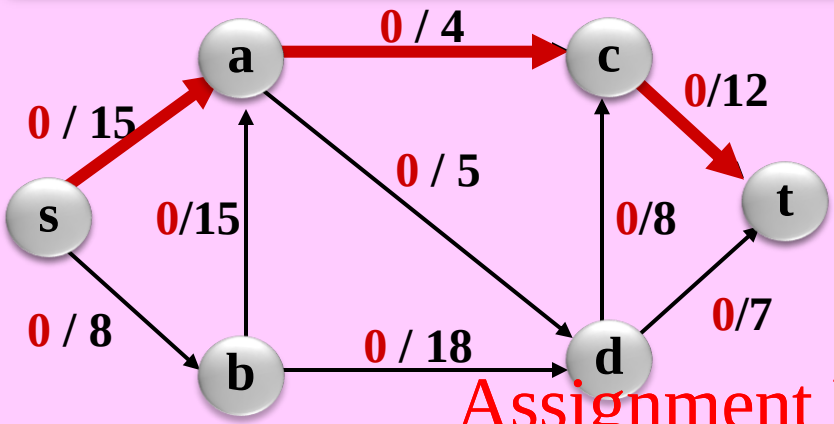
```
1.   for each e ∈ E do f(e) ← 0                    § an initial feasible flow
2.   optimal ← false
3.   while not optimal do                          § each iteration in O(E) time
4.         Construct residual graph G(f)
5.         if ∃ an s-t path p in G(f)              § do a graph search in G(f)
6.               then augment f along p
7.               else optimal ← true               § no path is coming
8.   end-while
9.   return f
end
```

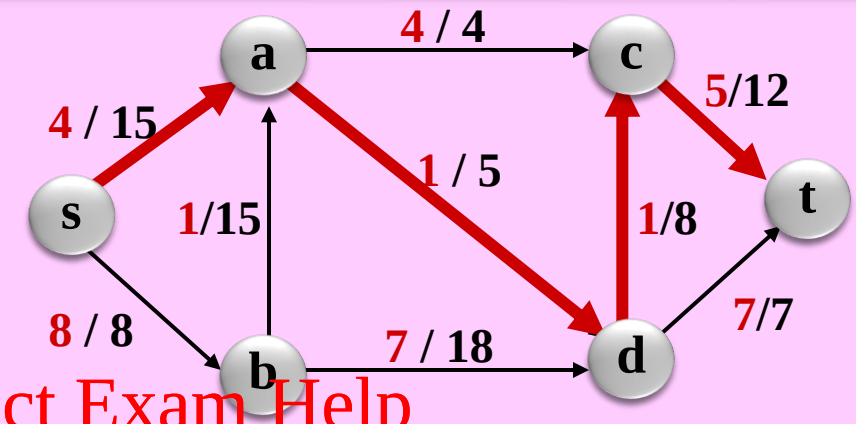

Ford-Fulkerson [1956-196

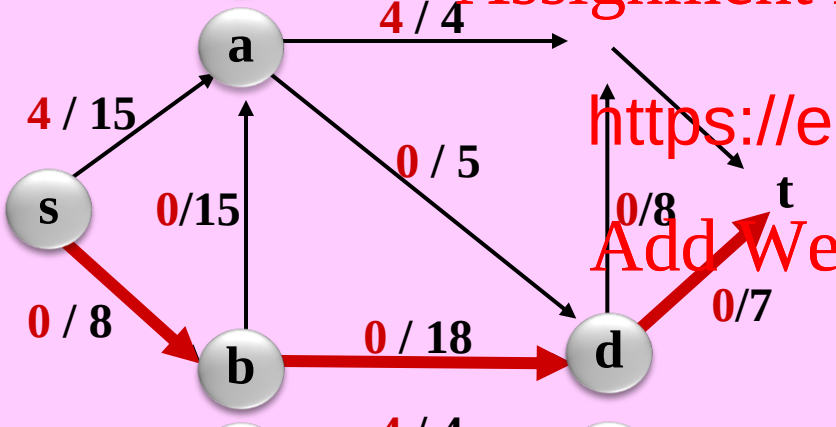
Fundamental Questions:

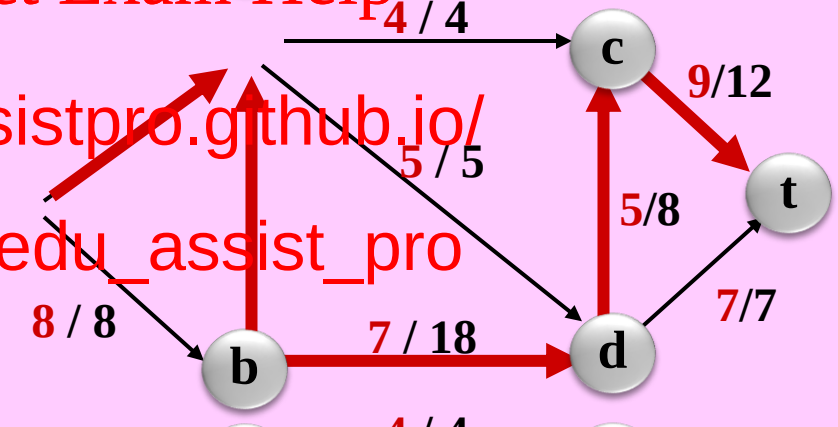
- Termination
- Correctness: optimality upon termination
- Efficiency: # of augmenting iterations.

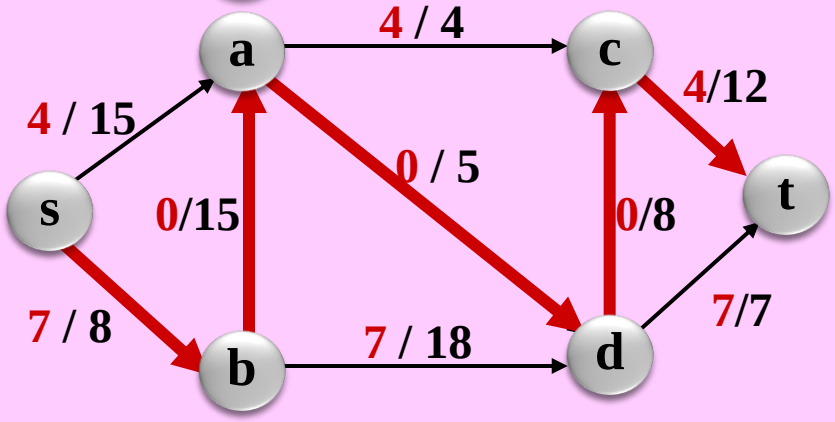
Example

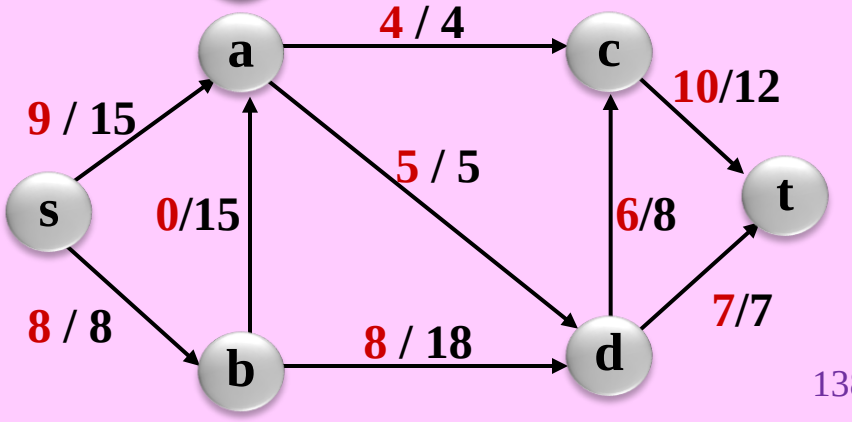






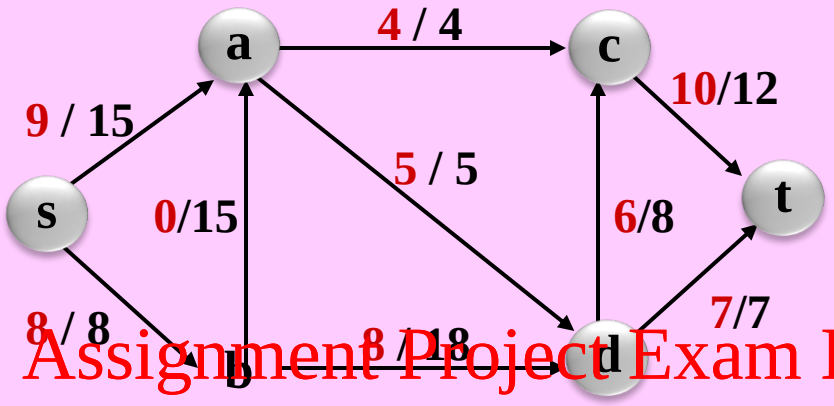




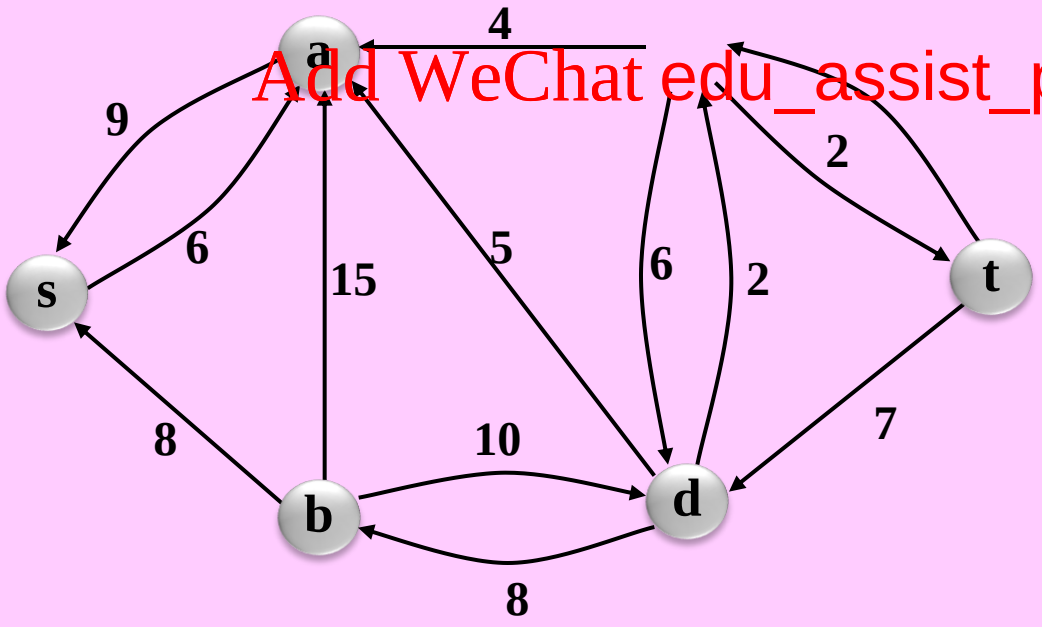


Example

G, f:



G(f):



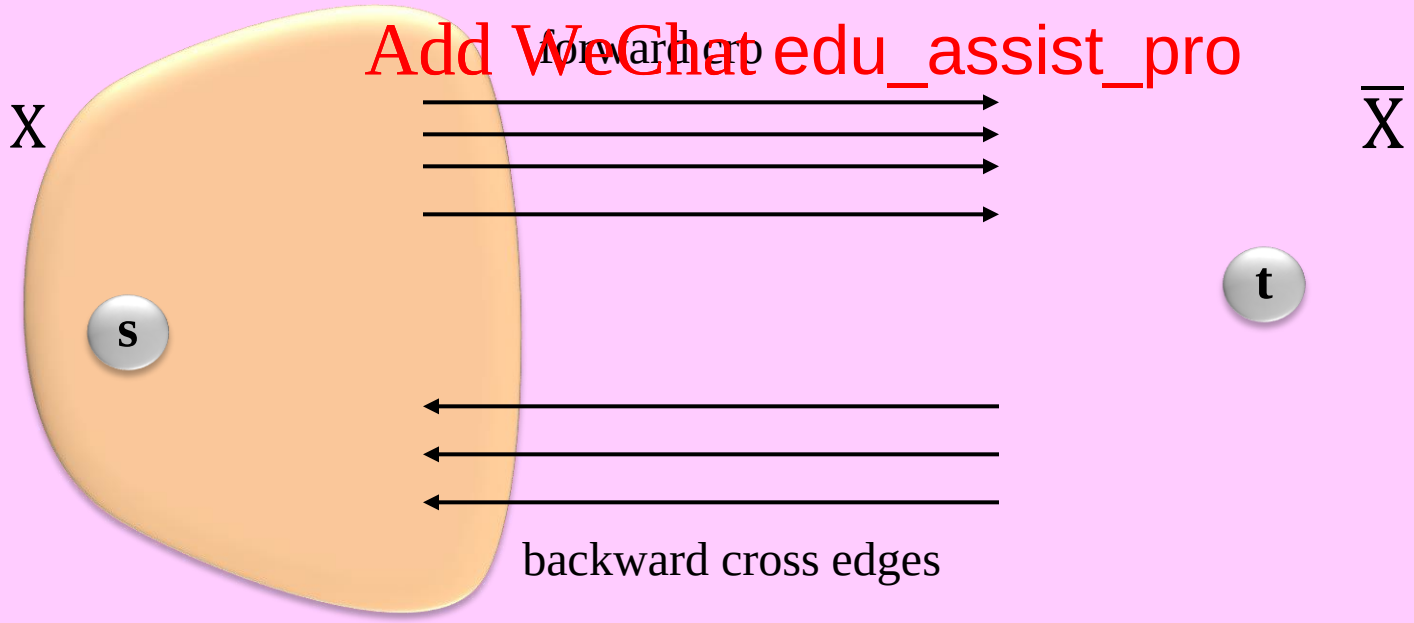
nodes reachable from s

st-CUT

An st-cut is a partitioning of the vertex set V into a subset X and its complement $\bar{X} = V - X$ such that $s \in X$ and $t \in \bar{X}$.

Cut Capacity: sum of capacities of **forward** cross edges of the cut.

$$c(X, \bar{X}) = \sum_{(u,v) \in E} c(u, v).$$



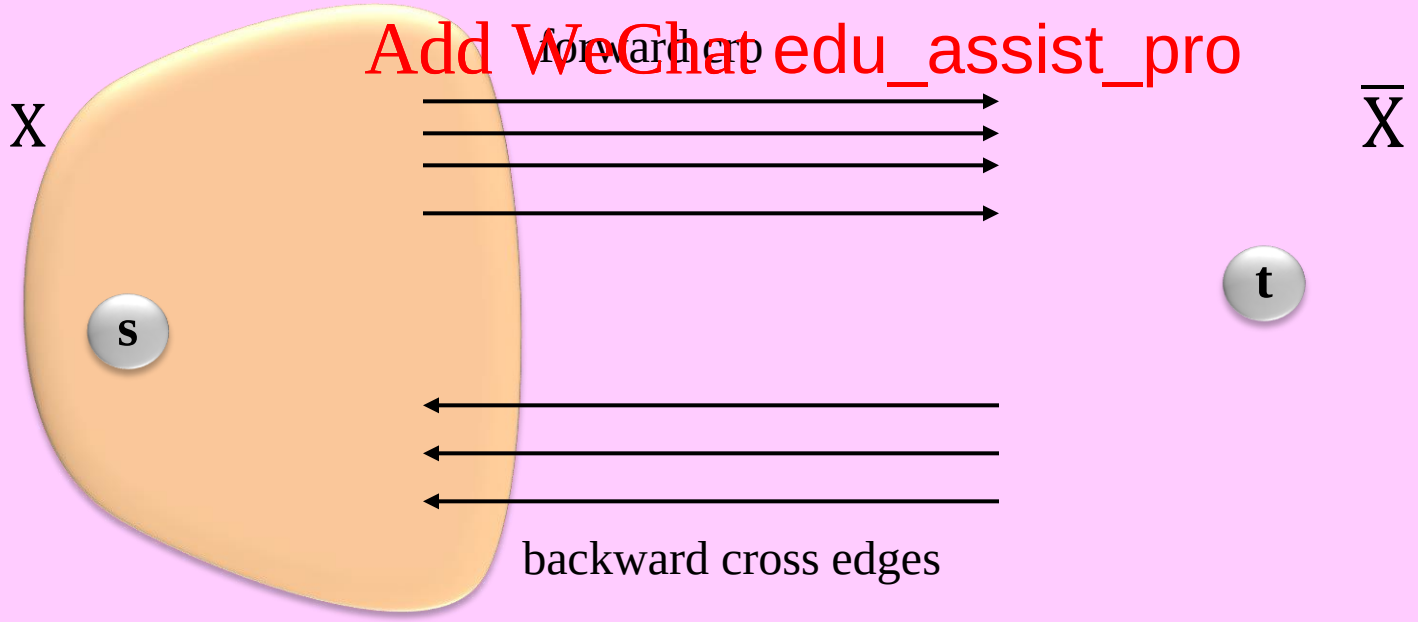
Flow across a cut

FACT: For any feasible flow f and any st-cut $(X, \bar{X})$, the flow value v(f) satisfies:

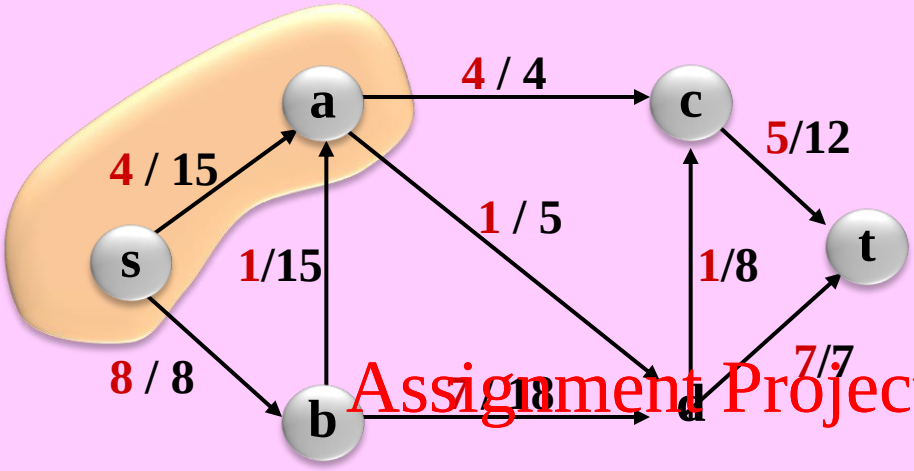
$v(f)$ = (total forward cross flow) – (total backward cross flow)

$$v(f) = \sum_{\substack{(u,v)\in E \\ u\in X, v\in \bar{X}}} f(u,v) \; - \sum_{\substack{(u,v)\in E \\ u\in \bar{X}, v\in X}} f(u,v)\,.$$

Proof: By induction on law.

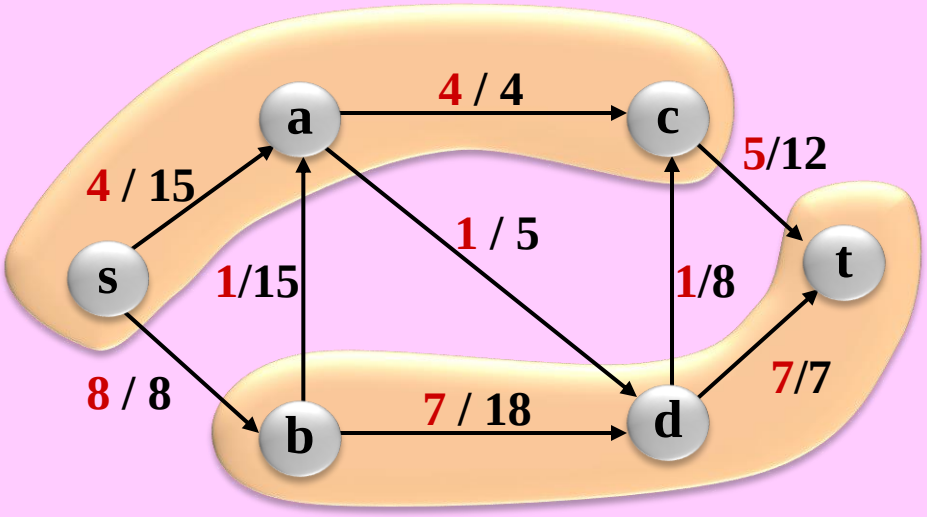


Example



$v(f) = 12 = (4 + 1 + 8) - (1)$

$C(X_1, \overline{X}_1) = 17 = 8 + 5 + 4$



$v(f) = 12 = (8 + 1 + 5) - (1 + 1)$

$, \overline{X}_2) = 25 = 8 + 5 + 12$

Max-Flow Min-Cut Theorem

Theorem [Ford-Fulkerson 1962]:

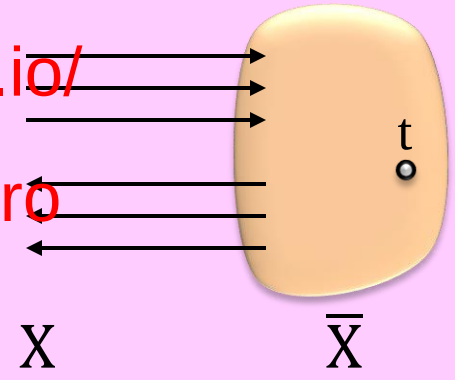
Let f be any feasible flow, and $(X, \overline{X})$ be any st-cut. Then

(a) $v(f) \leq C(X, \overline{X})$,

(b) f is a max flow and $(X, \overline{X})$ is a min capacity st-cut if and only if $v(f) = C(X, \overline{X})$.

Proof:

(a) $v(f)$ = (total forward $-$
$\leq$ (total forward capacity) -0
$= C(X, \overline{X})$.



(b) At the last iteration of Ford-Fulkerson's algorithm, let
$X \leftarrow$ the set of nodes reachable from s in G(f).
$(X, \overline{X})$ is an st-cut with the following properties:
all its forward cross edges are saturated (f = c) and
all its backward cross edges are empty (f = 0).
2nd line of proof of (a) is now met with =.

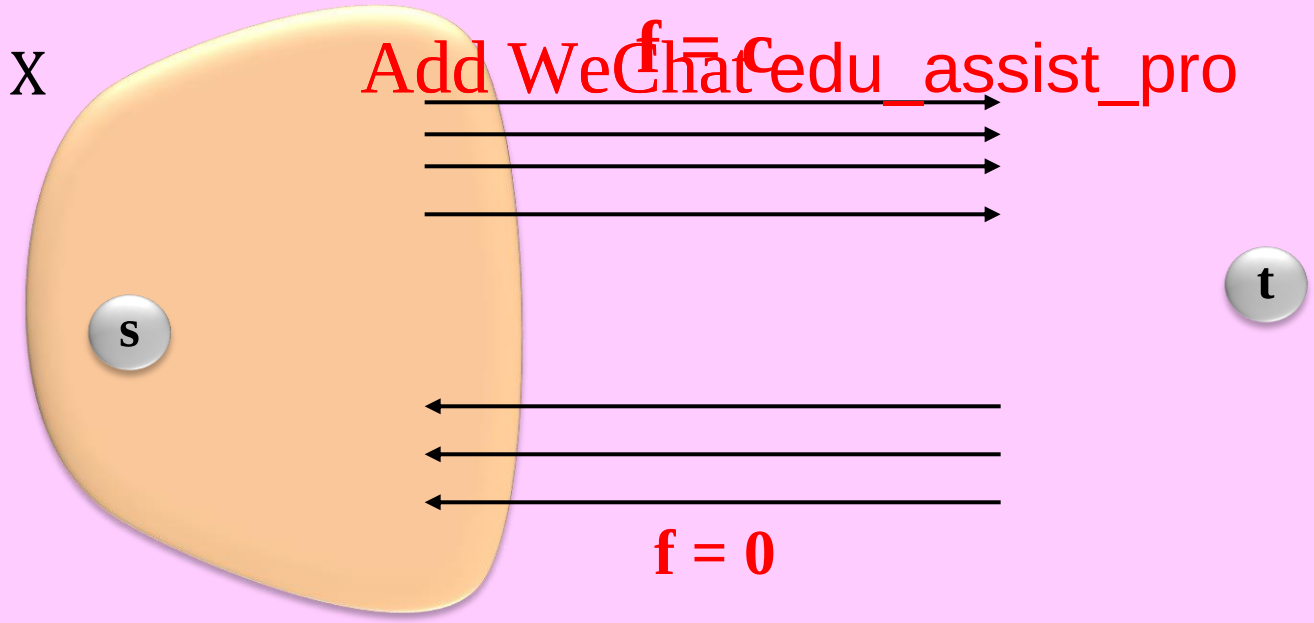
Max Flow Min Cut

Optimality: For any feasible flow f & any st-cut $(X, \overline{X})$:

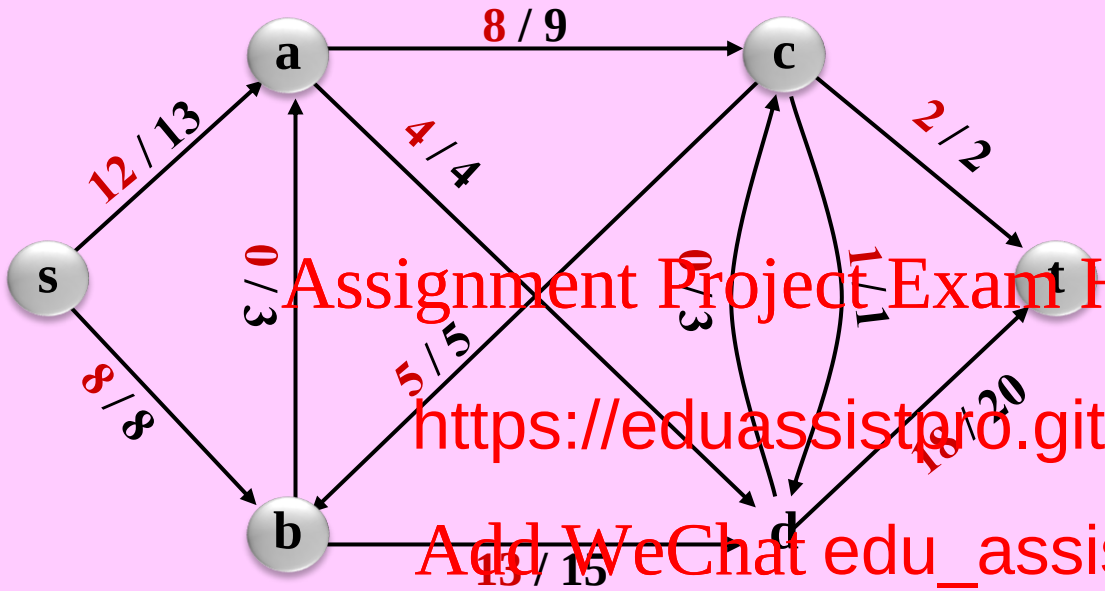
f is a **max-flow** & $(X, \overline{X})$ is a **min-cut**

if and only if

all forward cross edges of the cut are saturated (f = c), &
all backward cr e empty (f=0).



Example



Max-Flow & Min-Cut:
Forward cross edges are saturated.
Backward cross edges are empty.

Max flow value = 12 + 8 = 20

= Min st-cut capacity = 8 + 4 + 5 + 1 + 2 = 20.

Integrality Theorem:

If all edge capacities are integers, then there is always an all integer max flow, and the max flow obtained by Ford-Fulkerson's algorithm has this property.

Proof:

By induction on the # augmenting iterations: f remains an all integer flow.

Basis: Initial flow value is all zero integers.

Induction: Residual s
(they are city minus flow value).

Therefore, the augmenting also an integer.

Termination

Corollary [to the Integrality Theorem]:

If all edge capacities are **integers**, then Ford-Fulkerson's algorithm terminates after at most v* iterations, where v* is the max flow value.

In this case, the running time of the algorithm is O(v* E).

Proof: Upper-bound: $\Delta \geq 1$ per iteration (by the Integrality Theorem).

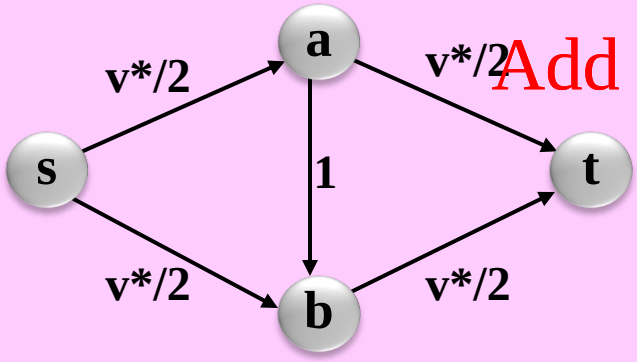
Worst-case tight:

WLOGA v* is even (e.g

een augmenting paths:

t & s→b←a→t.

ation.



However,

2 iterations would suffice if we chose

s→a→t & s→b→t.

Remark:

If some edge capacities are **irrationals**, Ford-Fulkerson's algorithm may never terminate. It may even converge to a strictly sub-optimal flow!

Edmonds-Karp Improvement

Edmonds-Karp [1972]:

Use BFS to pick the shortest augmenting path,
i.e., the s-t path in G(f) with fewest # of edges on it.

Notation:

$\delta_f(v)$ = shortest distance from s to v in G(f), for $v \in V$.

Edmonds-Karp Improvement

FACT 1: $\delta_f(v)$ is non-decreasing in each iteration, for all $v \in V$.

Proof: Consider an arbitrary augmenting iteration $f \to f'$.

Need to show that $\delta_{f'}(v) \geq \delta_f(v)$, $\forall v \in V$. We use induction on $d = \delta_{f'}(v)$.

Basis ($d = 0$): $\delta_{f'}(s) = 0$. Source s is the only possible node.

Induction ($d > 0$): Suppose $\delta_{f'}(v) = d$. Let $u = \pi(v)$ on the shortest s-v path in G(f').
So, $(u,v) \in E(G(f'))$ and $\delta_{f'}(u) = d - 1 < d$.

Case 1) $(u,v) \in E(G(f))$

$\delta_{f'}(v) = \delta_{f'}(u) + 1$ [$s \rightsquigarrow u \to v$ path in G(f')]

$\geq \delta_f(u) + 1$ [by the ind sis]

$\geq \delta_f(v)$ [by the triangle inequality in G(f)] .

Case 2) $(u,v) \notin E(G(f))$:

Flow along (v,u) must have increased in this iteration.

So, (v,u) is the last edge on the shortest s-u path in G(f).

$\delta_{f'}(v) = \delta_{f'}(u) + 1$ [$s \rightsquigarrow u \to v$ is shortest s-u path in G(f')]

$\geq \delta_f(u) + 1$ [by the induction hypothesis]

$= \delta_f(v) + 2$ [$s \rightsquigarrow v \to u$ is shortest s-u path in G(f)].

Edmonds-Karp Improvement

FACT 2: An edge (u,v) can be an augmenting bottleneck edge at most $|V|/2 - 1$ times.

Proof: Suppose (u,v) is the augmenting bottleneck edge in G(f) in some iteration i.
So, $\delta_f(v) = \delta_f(u) + 1$.
(u,v) is "filled up" and removed from the residual graph.
How can (u,v) er augmentation again?

Suppose (u,v) is the augmenting bottleneck e me later iteration j > i.
This can only happen if for some intermediat $< k < j$.
(v,u) appears on an augmenting path, say in G(f').
So, $\delta_{f'}(u) = \delta_{f'}(v) + 1$.

Therefore, $\delta_{f'}(u) = \delta_{f'}(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2$.

So, each time (u,v) is a bottleneck edge again, $\delta(u)$ increases by at least 2.
But $\delta(u) \leq |V| - 2$, since $u \in V - \{s,t\}$.
So, # times (u,v) can be a bottleneck edge $\leq (|V| - 2)/2 = |V|/2 - 1$.

Edmonds-Karp Improvement

Theorem: Edmonds-Karp Max-Flow algorithm takes $O(VE^2)$ time.

Proof: Each augmentation has a bottleneck edge.

An edge can become bottleneck of an augmentation O(V) times.

So, there are O(

Each augmenti

Total time is as claimed.

Algorithm MaxFlowMinCut (G=(V,E,c,s,t))

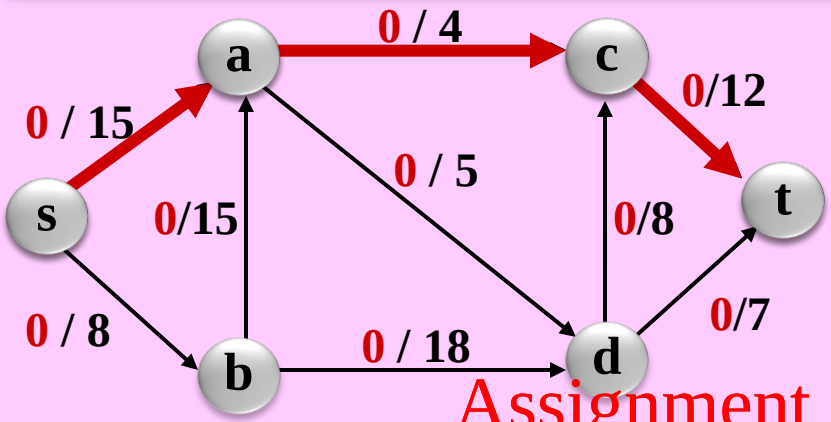
1. **for** each e ∈ E **do** f(e) ← 0
2. optimal ← false
3. **while** not optimal **do**
4. Construct residual graph G(f)
5. Do a **BFS** on G(f) from source s :
   X ← nodes reachable from s in G(f)
6. **if** t ∈ X
7. **then** p ←
8. augm
9. **else** opti
10. **end-while**
11. **return** (max-flow f , min-cut ($X, \bar{X}$)

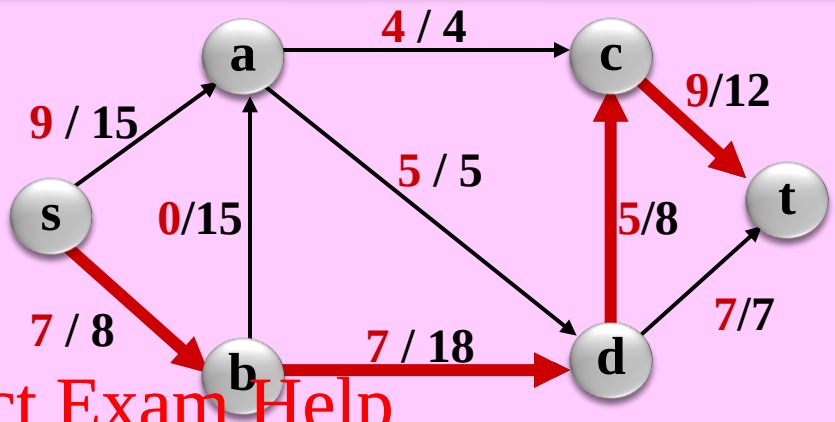
end

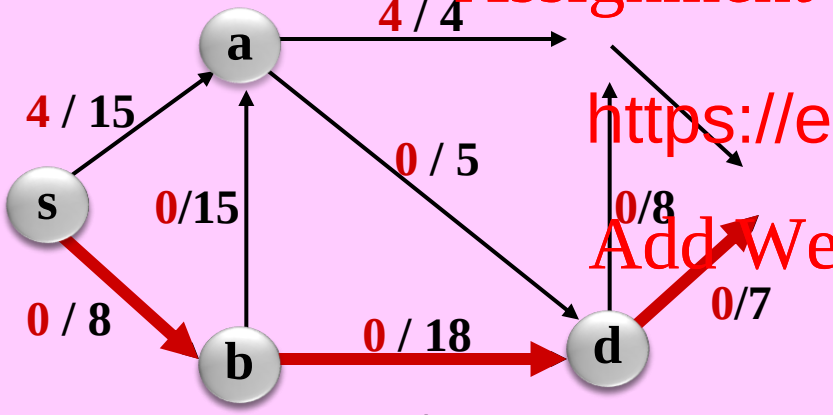
§ Ford-Fulkerson
§ +
§ Edmonds-Karp

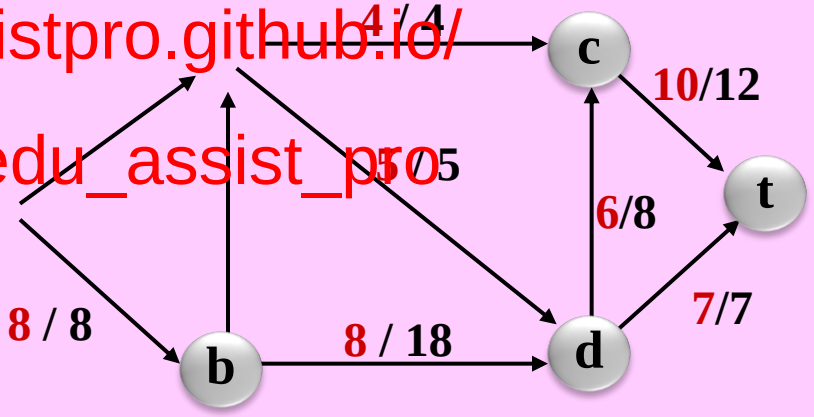
§ **O(VE²) time**

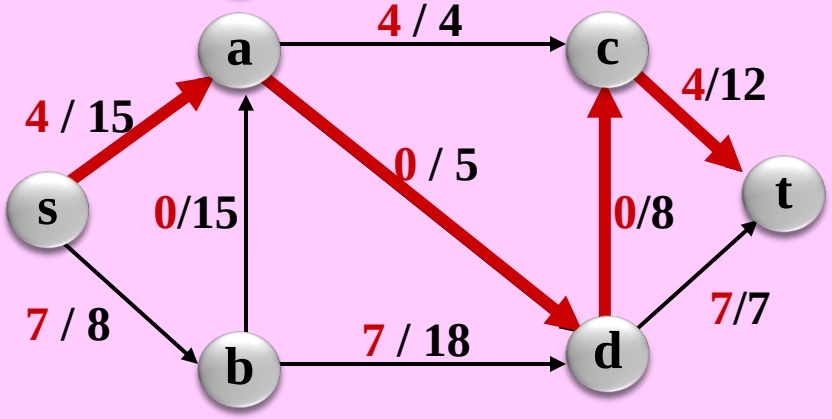
Example











Max-Flow & Min-Cut:
Forward cross edges are saturated.
Backward cross edges are empty.

Bibliography: Max Flow

- **Ahuja, Magnanti, Orlin[1993]** is a comprehensive book in this area.
- **Edmonds-Karp [1972]:** gave the first polynomial time max-flow algorithm based on Ford-Fulkerson's algorithm with BFS augmentation: $O(VE^2)$ time.
- **Dinits [1970]:** independently suggested BFS augmentation. In addition, he proposed a faster algorithm by augmenting along many paths simultaneously, known as "blocking flows": $O(EV^2)$ time.
- **Karzanov [1974]**: gav gorithm on dense graphs based on the notion of preflows: $O(V^3)$ ti
- **Melhotra, Kumar, Maheshvari [1978]:** ga algorithm with the same time complexity: $O(V^3)$ time.
- **Sleator, Tarjan [1983]:** gave the fastest known max-flow algorithm on sparse graphs by devising an efficient link-&-cut tree data structure: $O(E\ V \log V)$ time.
- **Borradaile, Klein [2009]:** give a faster max-flow algorithm on planar graphs.

P.T.O.

Bibliography: Max Flow

- R.K. Ahuja, T.L. Magnanti, J.B. Orlin, "Network Flows: Theory, Algorithms, and Applications," Prentice Hall, 1993.
- G. Borradaile, P. Klein, "An O(n log n) algorithm for maximum st-flow in a directed planar graph," JACM 56(2), 9:1-9:30, 2009.
- E.A. Dinits, "Algorithm for solution of a problem of maximum flow in a network with power estimation," Soviet Mathematics Doklady, 11(5):1277-1280, 1970.
- Jack Edmonds, R.M. Karp, mic efficiency for network flow problems," J. ACM 19:24
- L.R. Ford, Jr., D.R. Fulkers "Flows in Networks niversity
- A.V. Karzanov, "Determining the maximal flow in ethod of preflows," Soviet Mathematics Doklady, 15:434-437, 1974.
- V.M. Melhotra, M.P. Kumar, S.N. Maheshvari, "An O($|V|^3$) algorithm for finding maximum flows in networks," Information Processing Letters, 7(6):277-278, 1978.
- C. Papadimitriou, K. Steiglitz, "Combinatorial Optimization: Algorithms and Complexity," 1st edition Prentice Hall, 1982; 2nd edition, Courier Dover, 1998.
- D.D. Sleator, R.E. Tarjan, "A Data Structure for Dynamic Trees," Journal of Computer and System Sciences, 26(3):362-391, 1983.

Graph Matching

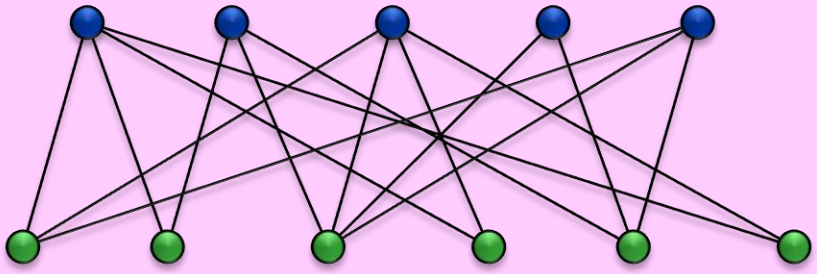
$M \subseteq E(G)$ is a **matching** in undirected graph G if no two edges in M share a common vertex.

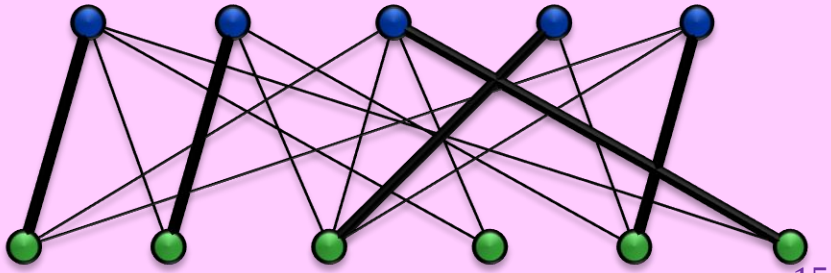
G: a non-bipartite graph:

M_1

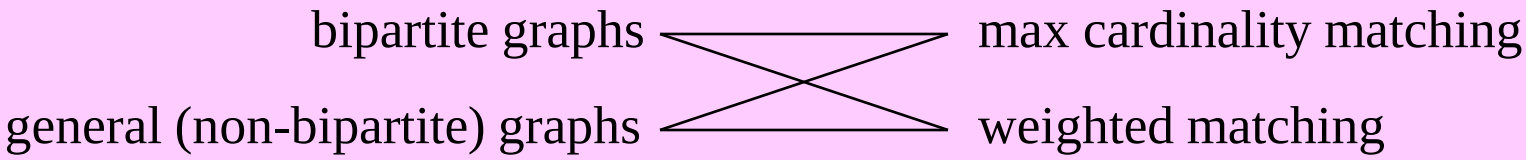
a perfect matching

G: a bipartite graph:





Graph Matching Problems



- All 4 problems can be solved in polynomial time.
- The most general of them is **weighted matching on general graphs**.
  It can be solved in O(
- We will show **Max Cardinality** matchin

  - **Bipartite Matching:** an O(VE)-time algorithm
    by a transformation to the Max Flow Problem,

  - **Non-Bipartite Matching:** sketch an $O(V^2E)$-time algorithm.

Some Applications

Bipartite Matching:

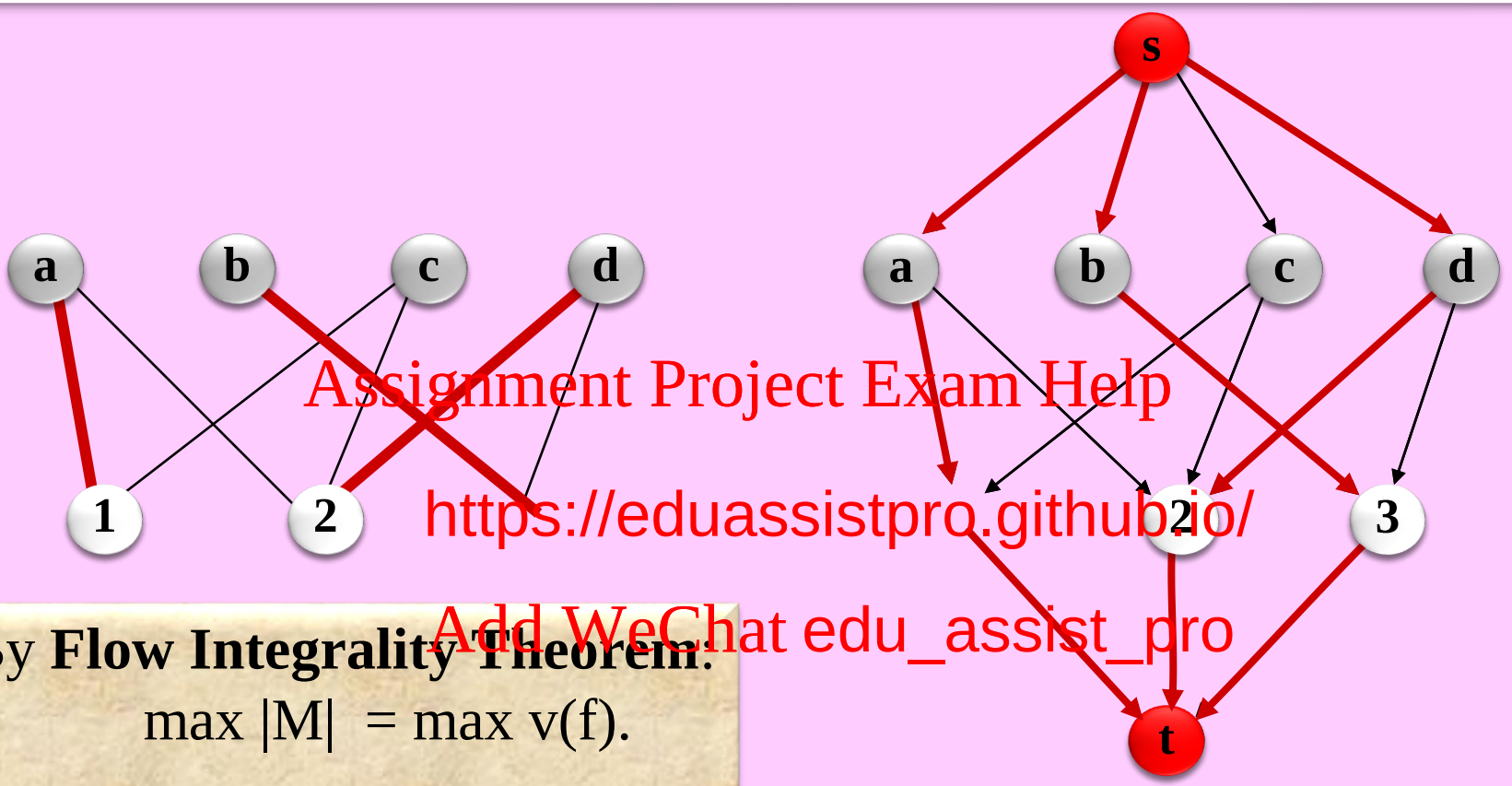
- assign jobs to workers:
  maximize employment (unweighted) or profit (weighted).
- 2 processor scheduling with job conflict constraints:
  minimize completion time.

Non-Bipartite Matchin

- 2 processor sch nstraints:
  minimize completion time.

- Traveling Salesman Problem.
- The Chinese Postman Problem
  (MST + Shortest Paths + Min Weight Perfect Matching).
- Covering and Packing Problems.
- Assign 2n students to n 2-bed dorm-rooms
  with student acquaintance-pair constraints.

Max Cardinality Bipartite Matching



By **Flow Integrality Theorem**:

$$\max |M| = \max v(f).$$

On this special flow network

Ford-Fulkerson's algorithm

takes O(VE) time.

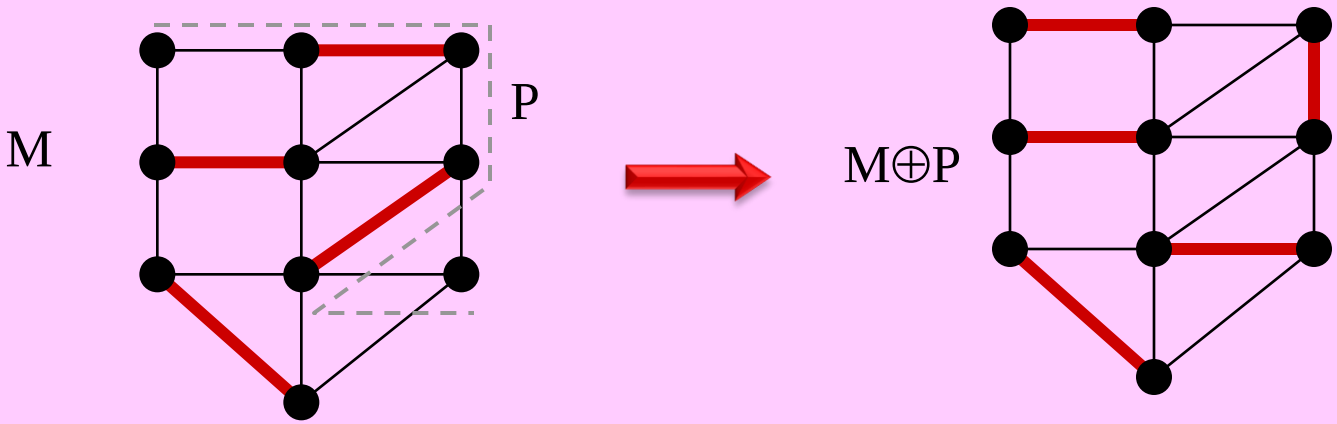
all edge capacities = 1

Augmenting Paths

Suppose M is a matching in graph G = (V, E).

- A node of G is **matched** if it is incident to some edge in M. Otherwise, it is **exposed**.
- An **alternating** path (cycle) with respect to M in G is a path (cycle) such that the edges along the path (cycle) alternate between being in M and being in E – M.
- An **augmenting path** two exposed nodes. (By necessity, an aug
- Augmenting M along an augmenting pat g M to $M \oplus P = (M - P) \cup (P - M)$ (i.e., edges ut not both)

Note that $|M \oplus P| = |M| + 1$.



The Augmenting Path Theorem

THEOREM: A matching M in a graph G has maximum cardinality if and only if there is no augmenting path in G with respect to M.

Proof:

[⇒]: Obvious.

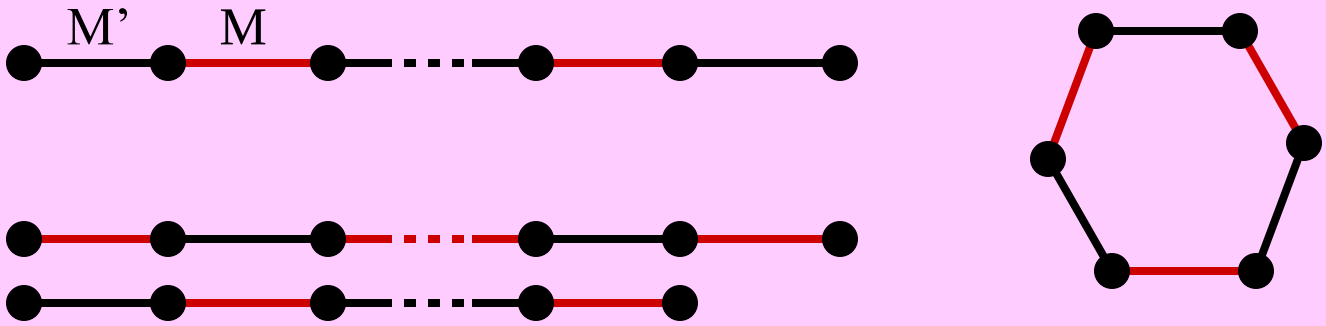
[⇐]: Suppose there is a m > |M|.

Consider M⊕M'. E at most 2 -

So M⊕M' consists alternating paths and cycles.

Since there are more edges in M' than i ne of the connected components in M⊕M' must be a path P ends in an edge in M'.

P is an augmenting path in G with respect to M.



Generic Max-Cardinality Matching Method

Algorithm MaxCardinalityMatching(G = (V,E))

1. M ← ∅
2. optimal ← false
3. **while** not optimal **do** § O(V) iterations
4. Search for an augmenting path P in G wrt M § This is the key step
5. **if** P found
6. **then** M ←
7. **else** optim
8. **end-while**
9. **return** M

end

- In the following pages we sketch a method to do step 4 in O(VE) time.
- This results in an $O(V^2E)$-time max-cardinality matching algorithm.
- The fastest known method takes $O(V^3)$ time.

Matched & Exposed Nodes

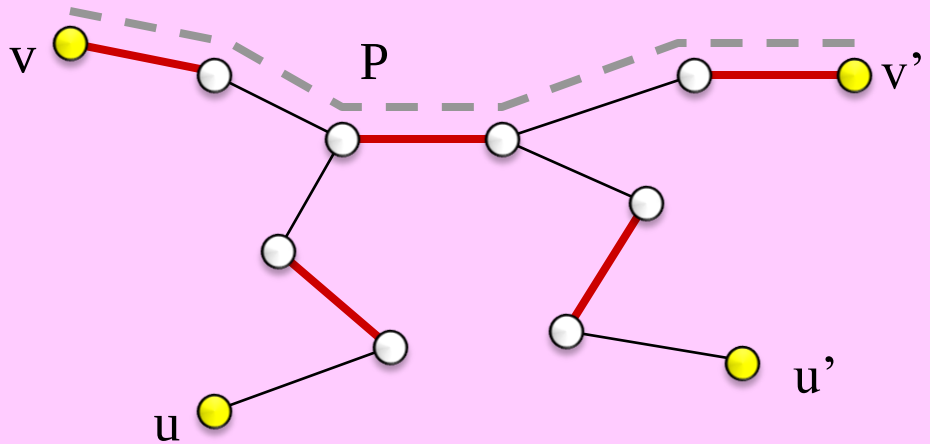
FACT 1: If a node becomes matched, it will remain matched.

A **u-augmenting-path** is an augmenting path where exposed node u is one end of it.

FACT 2: If at some iteration there is no u-augmenting-path, then there will never be a u-augmenting-path later on.

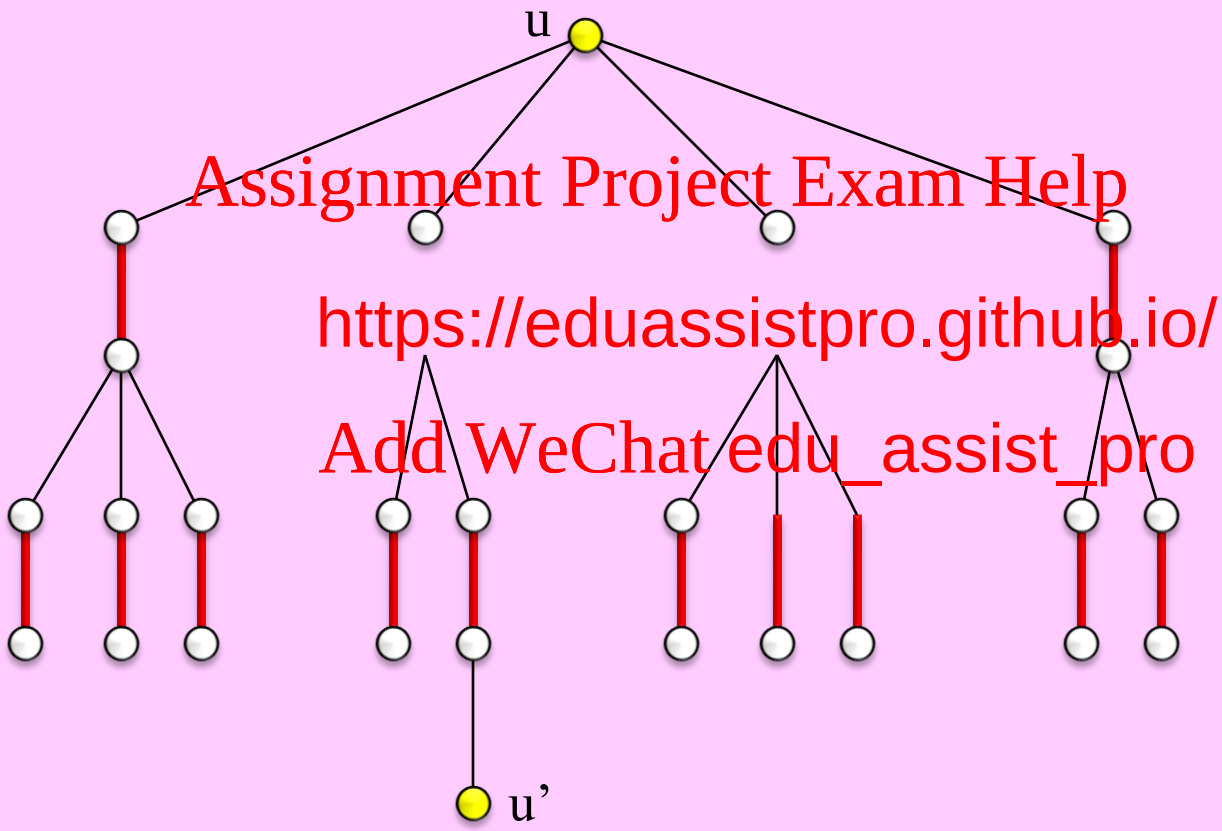
Proof sketch: Suppose n M.
Suppose
There ca M⊕P.

So, in our search, we can test each exposed node u once, and only once, as the starting node of an augmenting path.

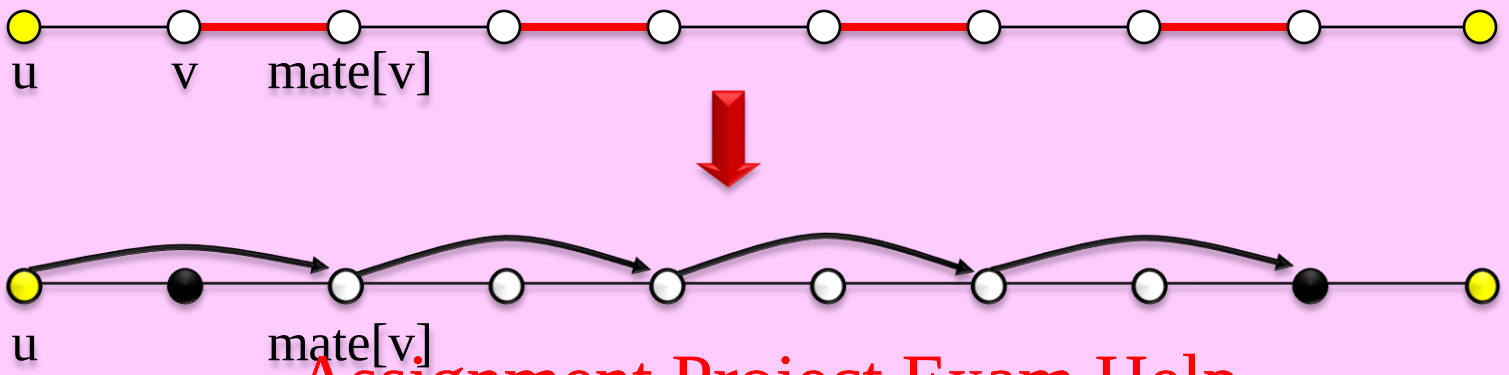


Search for a u-augmenting path

Grow a tree of alternating paths rooted at vertex u:



Augmenting Path Search as Digraph Search



x

y mate[y]

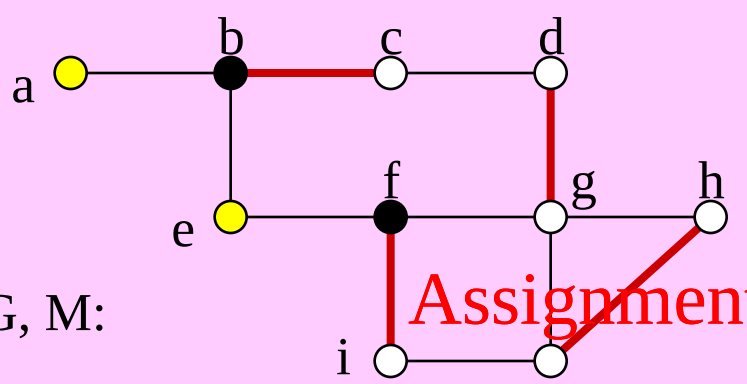
mate[y]

$(x,y) \in E - M \longrightarrow (x, mate[y]) \in E'$

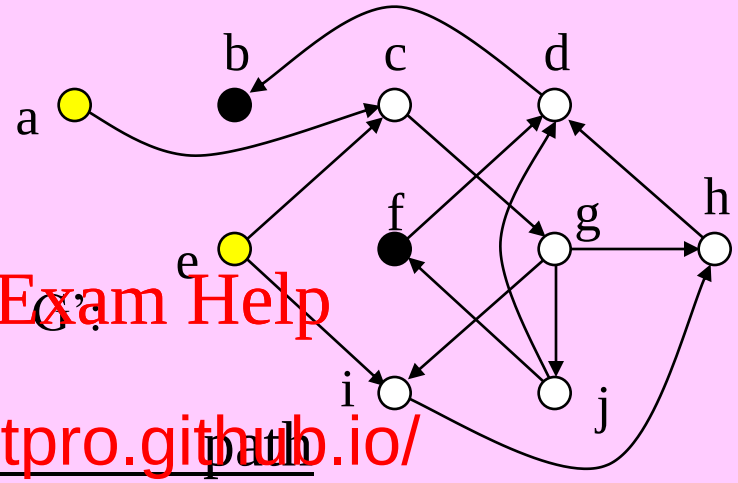
Now in the **auxiliary digraph** G' = (V, E'), search for a directed path from a node marked "exposed" (yellow) to a "target" node (solid black: adjacent to an exposed node).

Odd Cycles & Blossoms

Existence of odd length cycles creates further complications:



G, M:

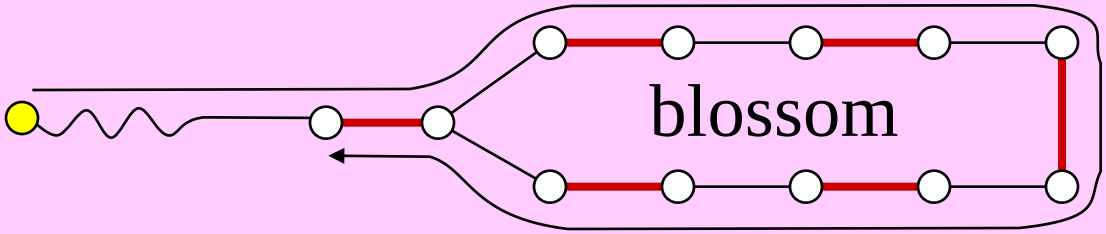


G':

augmenting path

(a, b, c, d, g, h, j, i, f, e) ↔ (a, b, c, d, g, h, j, i, f)

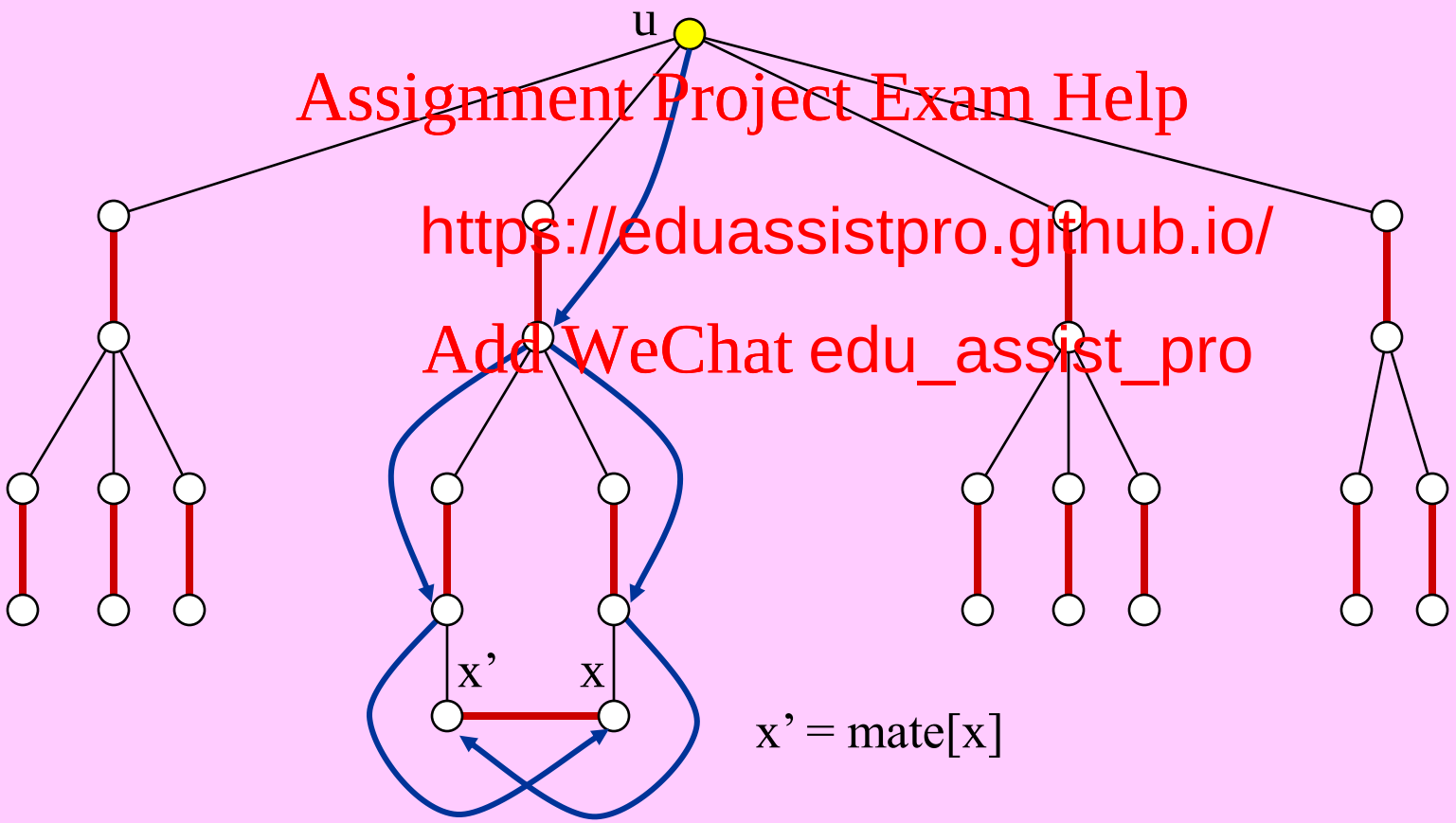
? ← d, b)



Blossom: An odd length cycle B with each of its nodes incident to an edge in M∩B, except for one node (base of the blossom) that has a mate outside B.

Discovering a Blossom

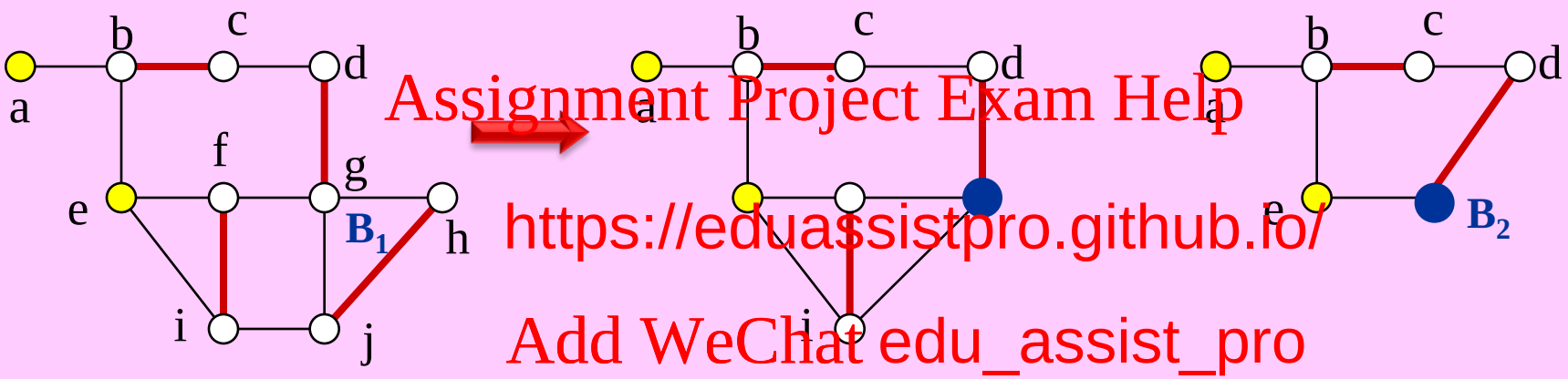
- When search of G' from u visits a node x' whose mate x has been visited.
- Backtrack up the search tree to lowest common ancestor of (x, x') to find base of the blossom.



Blossom Shrinking

Blossom Shrinking:

When a blossom B is discovered, we shrink it to a new single node called B that inherits all incident edges to the blossom nodes. This can be done in O(E) time.



FACT 3: Shrinking a blossom found during the search for a u-augmenting path preserves existence (non-existence) of augmenting paths.

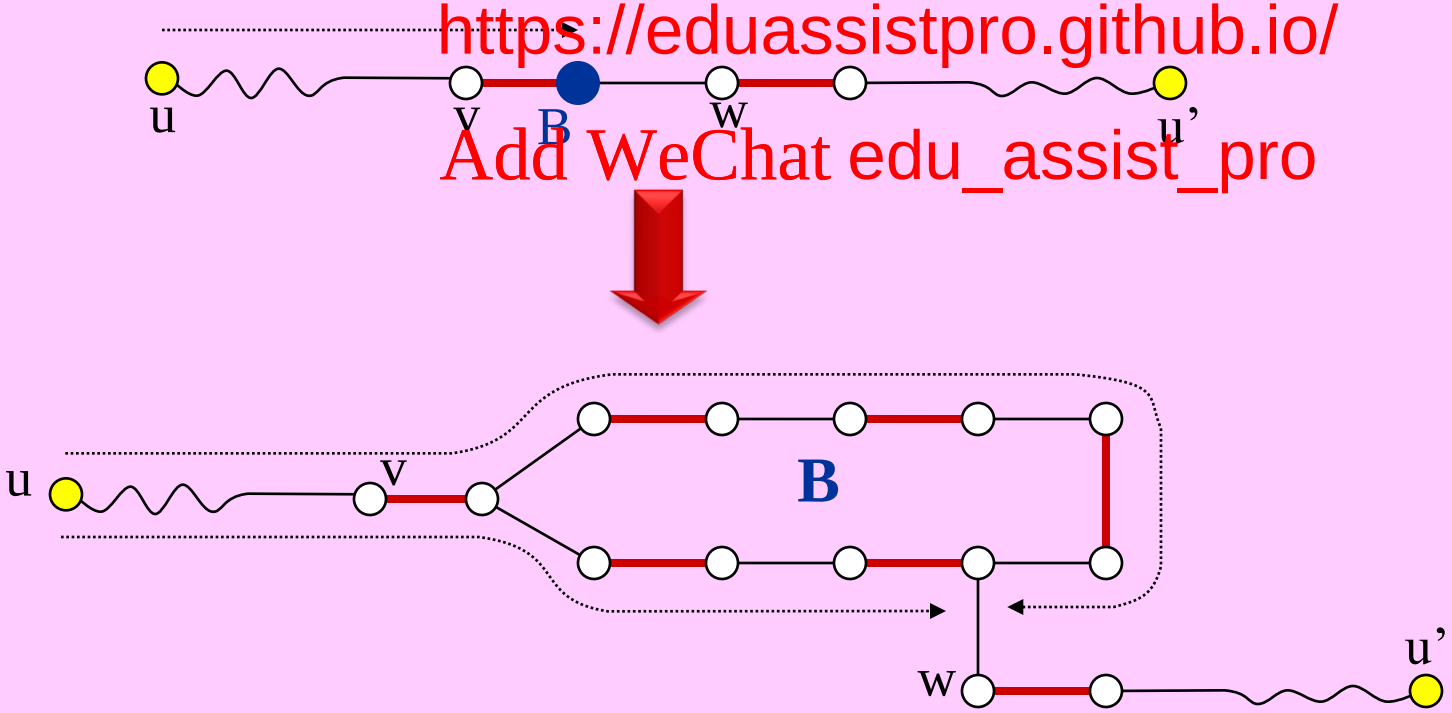
FACT 4: There can be at most O(V) blossoms (possibly nested) in an augmenting iteration.

Blossom Expansion

Blossom Expansion:

When an augmenting path P is found, some of its nodes might be shrunken blossom nodes.

Expand these blossoms back to their original state (in reverse order of nesting) and re-route P appropriately through the expanded blossoms until the expanded version of P is an augmenting path in G with no un-expanded blossom nodes.



Max Cardinality Matching Algorithm

Algorithm MaxCardinalityMatching(G = (V,E))

1. M ← ∅ § matching edges
2. U ← V § untested exposed nodes
3. **while** U ≠ ∅ **do** § O(V) iterations
4. remove a node u from U
5. Search for an augmenting path P in G w.r.t. M using the auxilia inking/expansion
6. **if** P found **the**
7. remove the
8. M ← M ⊕ P
9. **end-if**
10. **end-while**
11. **return** M

end

THEOREM: The above algorithm takes $O(V^2E)$ time.

Bibliography: Graph Matching

Recommended books in the area:

- L. Lovász, M.D. Plummer, "Matching Theory," Volume 121 of Annals of Discrete Mathematics. North Holland, 1986.
- C. Papadimitriou, K. Steiglitz, "Combinatorial Optimization: Algorithms and Complexity," 1st edition, Prentice Hall, 1982; or 2nd edition, Courier Dover, 1998.
- E.L. Lawler, "Combinatorial Optimization: Networks and Matroids," Holt, Rinehart & Winston, 1976. [gives the first $O(V^3)$-time non-bipartite matching algorithm]

Algorithms for bipartite matching have been known for some time:

- M. Hall Jr., "An algorithm Math. Monthly 63:716-717, 1956.
- L.R. Ford, Jr., D.R. Fulkers https://eduassistpro.github.io/ niversity Press, 1962.

The simple transformation of bipartite matching to ma inted out by

- S. Even, R.E. Tarjan, "Network flow and testing gr SIAM J. Computing 4(4):507-512, 1975.

The first polynomial time algorithm for non-bipartite matching is from

- Jack Edmonds, "Paths, trees, flowers," Canadian Journal of Mathematics, 17:449-467, 1965.
- Jack Edmonds, "Matching and a polyhedron with 0-1 vertices," J. Res. NBS, 69B:125-130, 1965. [gives an algorithm for the weighted matching problem]

Faster than $O(V^3)$-time algorithm for the geometric version of the problem was first discovered by

- P. Vaidya, "Geometry helps in matching," STOC, pp: 422-425, 1988.

1. Given the adjacency list structure of a graph G = (V, E), give an O(V+E)-time algorithm that reorders the vertices in each adjacency list in sorted order (say numerically, or alphabetically).

2. **[CLRS, Exercise 22.1-3, p. 592]** The **transpose** of a digraph $G = (V , E)$ is the digraph $G^T = (V , E^T)$, where $E^T = \{ (v,u) \mid v \in V, u \in V, (u,v) \in E \}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G, for both the adjacency list and adjacency matrix representations of G. Analyze the running times of your algorithms.

3. **[CLRS, Exercise 22.1-5, p. 593]** The **square** of a digraph $G = (V , E)$ is the digraph $G^2 = (V , E^2)$ such that $(u,w) \in E^2$ if and only if for some $v \in V$, both $(u,v) \in E$ and $(v,w) \in E$. That is, G^2 contains an edge from u to w whenever G contains a path from u to w with exactly two edges. Describe efficient algorithms for computing G^2 from G for both the adjacency list and adjacency matrix rep times of your algorithms.

4. **[CLRS, Exercise 22.1-6** presentation is used, most graph algorithms require time $\Omega(V^2)$, but there are som w that determining whether a digraph G contains a **universal sink** (a vertex w 1 and out-degree 0) can be determined in time O(V), given the adjacency ma

5. **[CLRS, Exercise 22.1-7, p. 593]** The **vertex-edge incidence matrix** of a digraph G = (V, E) is a matrix $B[V,E] = (b_{ue})$ such that

$$b_{ue} = \begin{cases} -1 & \text{if edge } e \text{ leaves vertex } u, \\ +1 & \text{if edge } e \text{ enters vertex } u, \\ 0 & \text{otherwise.} \end{cases}$$

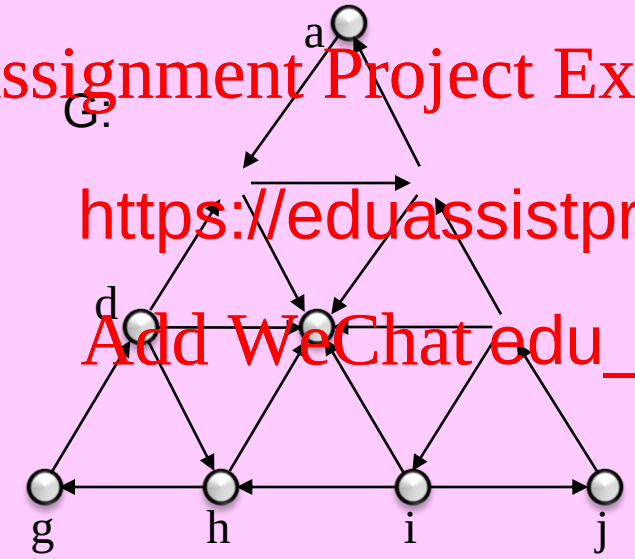
Describe what the entries of the matrix BB^T represent, where B^T is the transpose of B.

6. **[BFS & DFS Example]** Consider the example digraph G shown below.
We are given the adjacency list structure of G with each adjacency list alphabetically ordered.

(a) Show the BFS structure of G starting at the source vertex a.
Show the d and π values of each vertex.

(b) Do the same as in part (a), except that start at the source vertex j.

(c) Do a DFS of G (also assume DFS root scanning order is alphabetical).
Show the π values by our drawing convention, as well as the DFS starting and finishing time stamps d and f for each vertex. Classify tree-, back-, forward-, and cross- edges.



7. **[CLRS, Exercise 22.2-5, p. 602]** Argue that in a BFS from a given source vertex, the value d[u] assigned to each vertex u is independent of the order in which the vertices appear in each adjacency list. However, show that the BFS tree structure (the π values) may depend on the said ordering.

8. **[CLRS, Exercise 22.2-6, p. 602]** Give an example digraph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

9. **[CLRS, Exercise 22.2-8, p. 602]** The **diameter** of an un-weighted undirected tree $T = (V, E)$ is the largest of all shortest-path distances in the tree. Design and analyze an efficient algorithm to compute the diameter of a given tree.

10. **[CLRS, Exercise 22.2-9, p. 602]** We are given a connected undirected graph $G = (V, E)$.
    (a) Give an O(V+E)-time algorithm to compute a path in G that traverses each edge in E exactly twice, once in each d uence of edges along such a path.
    (b) Describe how you ca e given a large supply of pennies.

11. **[CLRS, Exercise 22.3-7, p. 611]** Rewrite the pro a stack to eliminate recursion.

12. **[CLRS, Exercise 22.3-8, p. 611]** Give a counter njecture that if there is a path from u to v in a digraph G, and if $d[u] < d[v]$ in a DFS of G, then v is a descendant of u in the DFS forest produced.

13. **[CLRS, Exercise 22.3-9, p. 612]** Give a counter-example to the conjecture that if there is a path from u to v in a digraph G, then any DFS must result in $d[v] \leq f[u]$.

14. **[CLRS, Exercise 22.3-12, p. 612]** Give an O(V+E)-time algorithm that outputs the vertex set of each connected component of a given undirected graph $G = (V, E)$.

15. Tree Broadcast:

The CEO of a large company has an urgent and important message, and she needs to notify every employee by phone. The company hierarchy can be described by a tree T, rooted at the CEO, in which each other node v has a parent node u where u is the direct superior officer of v, and v is a **direct subordinate** of u. To notify everyone of the message, the CEO first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. (Note that each person in this process can only call direct subordinates on the phone.) We can picture this process as being divided into **rounds**. In one round, each person who has already learned the message can call one of his or her direct subordinates on the phone. [Interpret T as a digraph (each edge directed from parent to child) given by its adjacency list structure. So, Adj[x] is the list of direct subordinates of employee x.] The number of rounds it takes for everyone to be notified depends on the order in which each person calls their direct subordinates. For example, in Figure (1) below, it will take only two rounds if A starts by calling B, but it will take three rounds if A starts by calling D.

a) Consider the example Figure (2). Show an optimum solution by labeling each edge by its round number. What is the required minimu

b) Define **MinRounds(x)** to be urted from the time employee x learns the message, to broadcast th odes in the subtree rooted at x). Develop a recurrence that expresses MinRounds(x) in terms of th dinates MinRounds(y), ∀y∈Adj[x]. [Hint: In Figure (3) suppose MinRounds(B) = 17, MinRounds( ) = 15. In what order should A call its direct subordinates B, C, D? What is MinRounds(A)?]

c) Using part (b), design an efficient algorithm that takes as input a broadcast tree T (rooted at root[T]) given by its adjacency list structure, and outputs the minimum number of rounds needed to broadcast a message in T.

d) Analyze the worst-case time complexity of your algorithm in (c) as a function of n (# nodes in T).

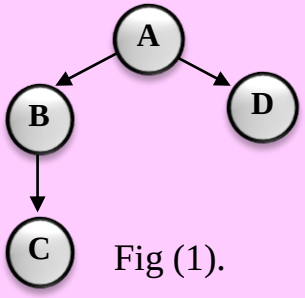


Fig (1).

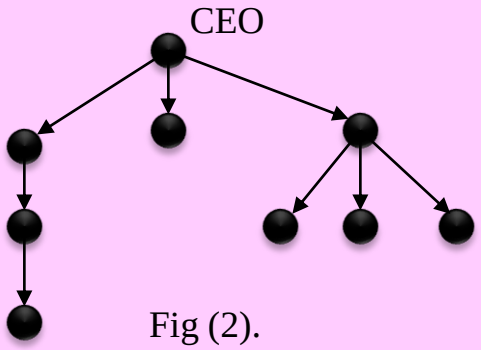


Fig (2).

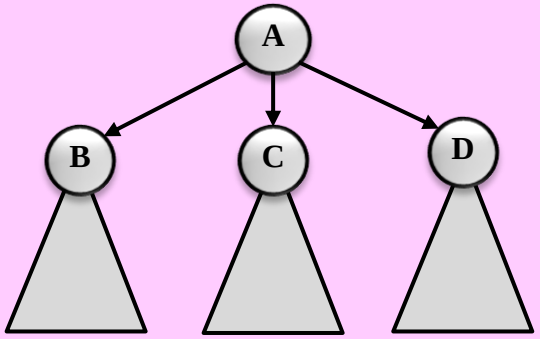


Fig (3).

16. **[CLRS, Exercise 22.4-1, p. 614]** Run the Topological Sort Algorithm of the example DAG of Fig (a) below and show the pertinent results.

17. A **Hamiltonian path** in a directed graph G=(V,E) is a directed simple path that visits each vertex of G exactly once. (Note that we are referring to a Hamiltonian **path**, not a **cycle**.) In general, the problem of deciding whether a given digraph has a Hamiltonian path is known to be NP-complete.
    (a) Does the graph shown in Fig (b) below contain a Hamiltonian path?
    If yes, show one by highlighting the edges on the path.
    (b) If the graph G happens to be a DAG, how does finding a topological sort of G help in finding a Hamiltonian path of G?

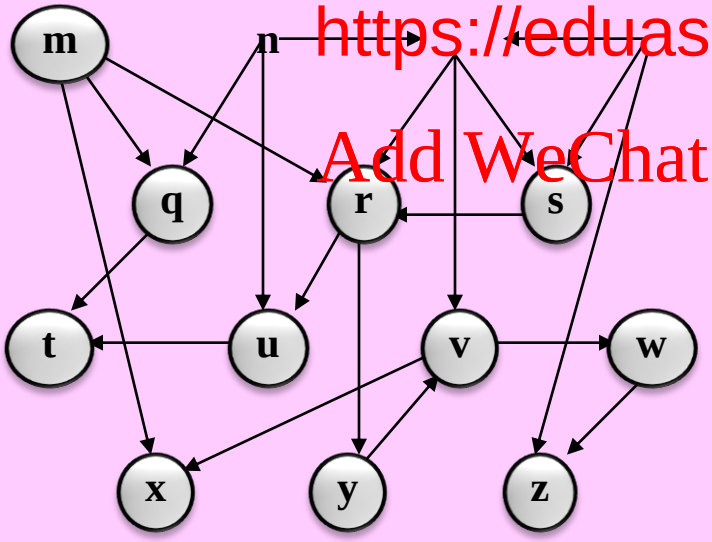


Fig (a)

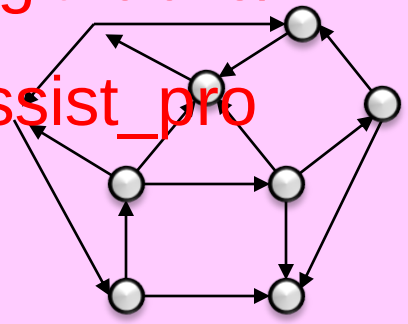


Fig (b)

18. **[CLRS, Exercise 22.4-2, p. 614]** Give a linear-time algorithm that takes as input a DAG G and two vertices s and t, and returns the number of paths from s to t in G. For example, in the digraph in Fig (a) on the previous page, there are exactly 4 paths from vertex p to vertex v: pov, poryv, posryv, psryv. (Your algorithm needs to give the number of paths, not list them.)

19. **[CLRS, Exercise 22.4-3, p. 615]** Give an algorithm that determines whether or not a given undirected graph G = (V, E) contains a cycle. Your algorithm should run in O(V) time, independent of |E|.

20. **[CLRS, Exercise 22.4-4, p. 615]** Prove or disprove: If a digraph G contains cycles, then TopologicalSort(G) produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the orderi

21. **[CLRS, Exercise 22.4-5, p. 615]** Another way to al sorting on a DAG G = (V, E) is to repeatedly find a vertex of in-degree 0, output it, all its outgoing edges from the graph. Explain how to implement this idea so that i +E).
What happens to the algorithm if G has cycles?

22. **[CLRS, Problem 22-3, p. 623]** An **Euler tour** of a connected graph G = (V, E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.
G is Eulerian if it has an Euler tour.
    (a) Show that a connected digraph G is Eulerian if and only if in-degree(v) = out-degree(v) $\forall v \in V$.
    (b) Show that a connected undirected graph G is Eulerian if and only if degree(v) is even $\forall v \in V$.
    (c) Describe an O(E)-time algorithm to find an Euler tour of G if one exists, once assuming G is directed, and once assuming G is undirected.
    [Hint: Merge edge-disjoint cycles.]

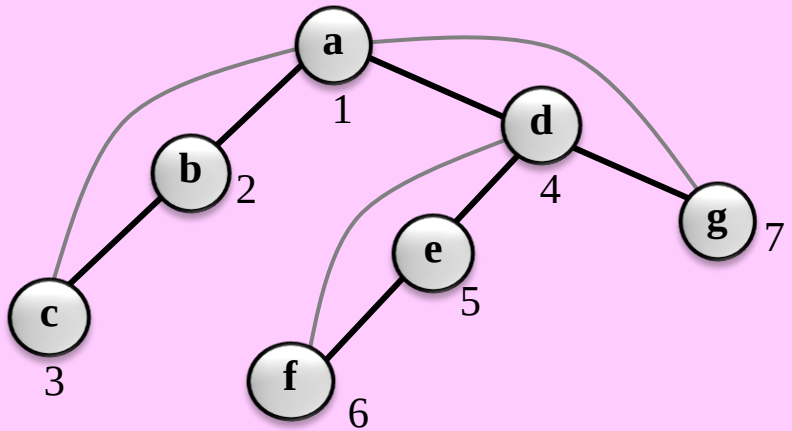
23. The algorithm below is **purported** to find an Euler tour in a given Eulerian undirected graph G.

Algorithm EulerTour (G)
Pre-Cond: G is an undirected Eulerian graph.
Post-Cond: Outputs a cycle in G
1. Perform a DFS traversal of G and number the vertices in DFS-preorder.
2. Re-initialize all vertices and edges of G as unused.
3. Produce a cycle as follows:
   Start from the vertex with preorder number 1 (computed in step 1), and
   repeatedly go to the vertex with highest preorder number possible along an unused edge.
   Stop when all edges incident to the current vertex are used.

end

For example, if the figure then in step 3 the algorithm will produce the cycle ⟨ d an Euler tour of the graph.

Give a counter-example to show that the above al ays capable of finding an Euler tour, even though there exists one. Therefo oes not work!
[Hint: the smallest counter-example has less than 10 vertices.]



24. Show how the Strongly Connected Components (SCC) algorithm as presented in this lecture slide works on the graph shown below. (Assume adjacency lists are alphabetically ordered.)
(a) Show the stack and the DFS-structure after step 1.
(b) Show the DFS-structure of the transpose graph G^T after step 3.
(c) Show the SCC Component graph of G.

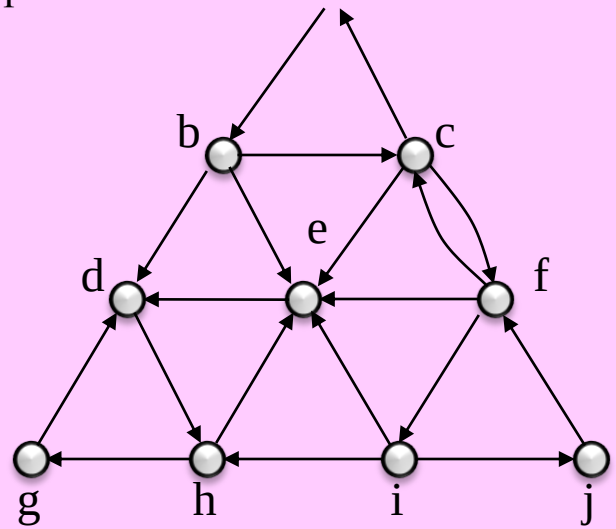
25. [CLRS, Exercise 22.5-5, p. 620] Give an O(V+E)-time algorithm to compute the SCC Component graph of a digraph G = (V, E). Make sure that there is at most one edge between two vertices in the component graph that your algorithm produces.

26. We say a digraph is weakly connected if its undirected version (i.e., ignoring the directions on the edges) is a connected undirected graph.
Let G be a digraph in which for every vertex v, indegree(v) = outdegree(v). Then show that G is strongly connected if and only if it is weakly connected.

27. [CLRS, Exercise 22.5-6, xplain how to create another digraph G' = (V, E') such

(i) The strongly con the same vertex partition of V,
(ii) G' has the same SCC Component graph
(iii) E' is as small as possible. [Note: E' is a subset of E.]

Describe an efficient algorithm to compute G'.

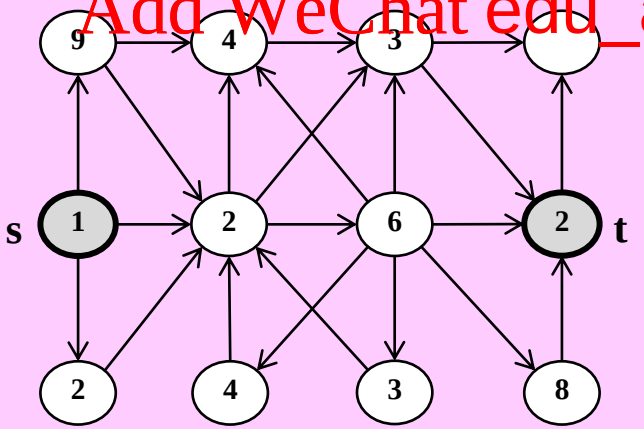


28. Treasure Hunt:

You are given a directed graph G = (V , E), where each node v∈V in the graph has a certain amount of money m(v) > 0 sitting on it. You are also given vertices s, t ∈ V . The goal is to start at s and follow a directed path in G to reach t, picking up as much money along the way as you can. You may visit the same node more than once, but once you have picked up the money at that node, it is gone. Define M(G, m, s, t) to be the maximum money that can be picked up on any path in G from s to t (paths need not be simple).

a) Show M(G, m, s, t) and a corresponding optimum st-path for the instance shown in the Figure below.

b) Design and analyze an efficient algorithm to compute M(G, m, s, t). (You are not required to compute the optimum st-path.) For full credit, your alg . [Hint: think about SCC.]

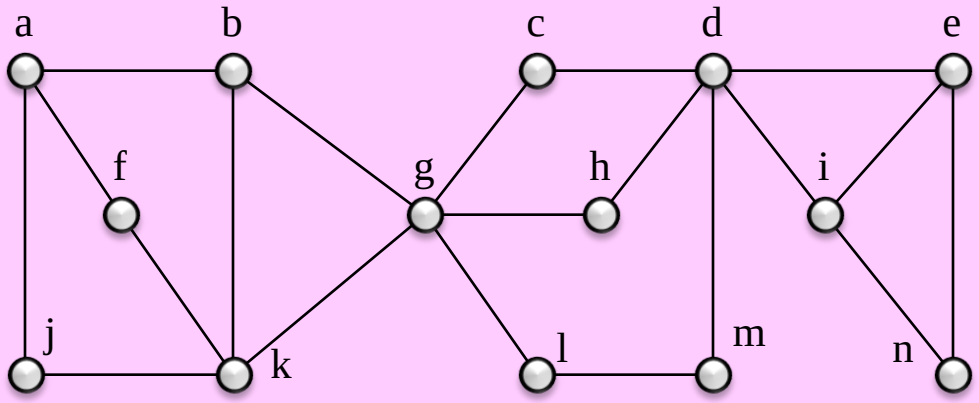


29. The City Hall has decided to convert every down-town street to a one-way street without restricting traffic accessibility, i.e., we should still be able to drive from any city block to any other city block without ever leaving the down-town area.
The problem is how to assign one-way directions to the down-town streets to achieve the City Hall's objective.
This can be formulated as a graph problem as follows: We are given an undirected connected graph G = (V, E). We want to assign a (one way) direction to each edge of G in such a way that the resulting directed graph G' = (V , E') is strongly connected.
In this case we say G' is a **strong orientation** of G.
We say G is **strongly orientable**, if it has at least one strong orientation.

(a) The graph shown below is strongly orientable. Show one of its strong orientations by assigning an appropriate direction to each of its edges.

(b) Show that a connected ble if and only if it has no **bridges**. A bridge is a the graph.
[Note: make sure to pr if.]

(c) Using DFS and part (b), design and analyze an to construct a strong orientation of a given undirected graph G, if th se, report that no strong orientation of G exists, and as a witness outpu
Justify the correctness and time complexity of your algorithm.

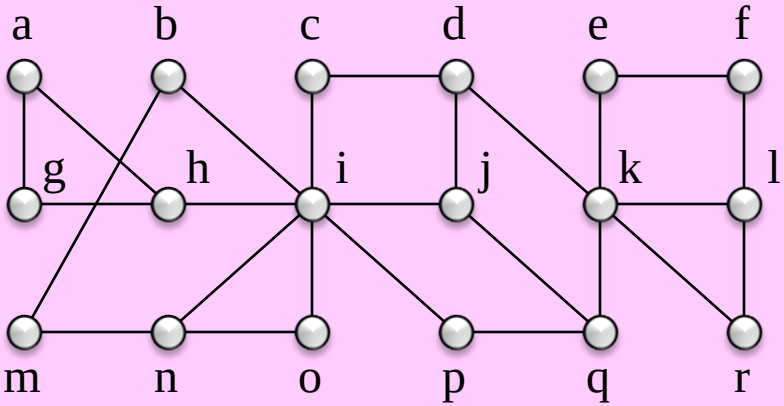


30. **[CLRS, Exercise 22.5-7, p. 621]** A digraph G = (V, E) is said to be **semi-connected** if, for all pairs of vertices u, v ∈V, at least one of u or v is reachable from the other.
Design and analyze an efficient algorithm to determine whether or not G is semi-connected.

31. Given a digraph G = (V, E) and three of its vertices x, y, z ∈V, give an efficient algorithm that determines whether or not G has a cycle that includes vertices x and y but excludes vertex z.

32. Trace the BiConnectivity algorithm on the example graph G shown below.
Show the articulation points and the biconnected components of G.

33. Let G = (V, E) be a biconnected graph. Let (v,w)∈E. Start with the graph G' = ({v,w}, {(v,w)}).
A **petal** is a simple path in odes that are also in G'.
Describe an on-line algori f G' and (s,t) is an edge of G not
in G'. AddPetal(s,t) finds s the nodes and edges of that
petal to G'. The execution of any on-line sequenc rations should take O(E) time.



34. Biconnectification Problem: given a connected undirected graph G, add a minimum number of new edges to G to make it biconnected.

Let us first consider the simpler case when G is a tree. Let L be the set of leaves of G.

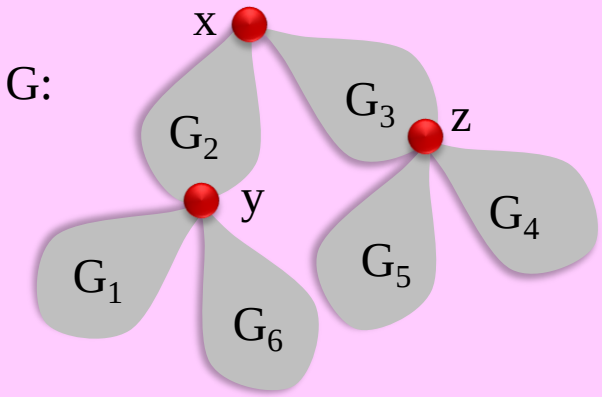
(a) Show that in this case there is always a solution where the added edges are between nodes in L.
(b) Show that there is a solution where the added edges form a spanning tree of L.
Hence, the number of new edges is $|L| - 1$.

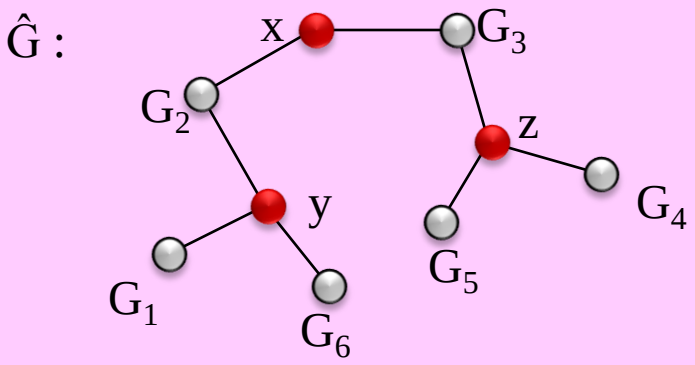
Now consider the general case. Suppose the biconnected components of G are G_i, for i = 1..k, and let A denote the set of articulation points of G.
Define the skeleton graph $\hat{G}$ of G as follows: (See the illustrative figure below.)

$V(\hat{G}) = \{G_i \mid i = 1..k\} \cup A$
$E(\hat{G}) = \{(G_i, x) \mid x$

(c) Show that $\hat{G}$ is a tree.

(d) Design & analyze a linear-time algorithm to sol ing (a-c) above.





35. Consider an arbitrary iteration of the generic MST algorithm that uses the Red and Blue Rules with the Red-Blue Invariant. The set B of blue edges at that point forms a forest of one or more blue components. By the Red-Blue Invariant we know that there is an MST T that contains B. Assume we still have more than one blue component. Form a cut C by partitioning these blue components into two non-empty subsets. Consider an arbitrary edge e of T that crosses the cut C. Give a counter-example to the conjecture that e must be a min-weight cross edge of the cut C.

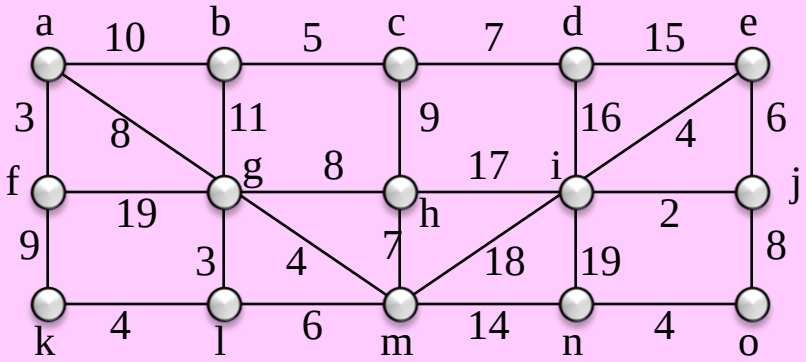
36. Is the following divide-&-conquer algorithm guaranteed to find a MST of a given graph G? : Divide V(G) in two arbitrary subsets of roughly equal size X and Y. Let G' and G'' be the subgraphs of G induced by X and Y, respectively. Recursively find an MST T' of G' and an MST T'' of G''. Let e be the minimum weight edge of G incident to X and to Y.
Return T = T' ∪ T'' ∪ {e} as the MST of G.

37. **[CLRS, Exercise 23.1-4,** { e∈E(G) | e is a minimu ph G such that the edge set does not form a MST of G.

38. **[CLRS, Exercise 23.1-6, p. 630]** Show that a grap MST if, for every cut of G, there is a unique minimum-weight crossing edge of
Show that the converse is not true by giving a counter-example.

39. **[CLRS, Exercise 23.1-9, p. 630]** Let T be a MST of G, and let V' be a subset of V(G). Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a MST of G'.

40. **[CLRS, Exercise 23.2-4, 23.2-5, p. 637]**
    (a) Suppose that all edge weights in a graph are integers in the range 1 to |V|. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range 1 to W for a given positive integer W? Express the running time in terms of the graph size and W.
    (b) Answer these same questions with respect to Prim's algorithm.



41. We want to explore some MST properties of the graph shown above.
    (a) Show the sequence of edges in the order that they are added to the MST using Kruskal's algorithm. Highlight the chosen MST edges in the graph, and indicate in what order they are added to the MST (e.g., by labeling them 1, 2, 3, ...).
    (b) Show the sequence of to the MST using Prim's algorithm, starting fro
    (c) Suppose we decide to real number from -∞ to +∞. What is the interval of weight values for this e spanning tree in part (a) above (as a set of edges) remains an MST of t ain your answer.
    (d) Suppose we decide to change the weight of ed l number from -∞ to +∞. What is the interval of weight values for this edge, for which the spanning tree in part (a) above remains an MST of the graph. Explain your answer.

42. Let T be a MST of G = (V, E, w). Suppose the weight w(e) of one of the edges e∈E is altered.
    (a) Characterize the complete real valued range for w(e) for which T still remains a MST of G. Consider two cases depending on whether e ∈T or e ∉T.
    (b) Using the characterization in part (a), describe an algorithm to update the MST T if the weight w(e) of a given edge e∈E changes to a new given value w'.

43. [CLRS, Problem 23-3, p. 640] Bottleneck Spanning Tree (BST) Problem:
Let T be any spanning tree of a given weighted connected undirected graph G.
The *bottleneck value* of T is defined to be the maximum edge weight among edges of T.
We say T is a *bottleneck spanning tree (BST)* of G if bottleneck value of T is minimum among all spanning trees of G. The **Bottleneck Spanning Tree Problem** is to compute a BST of G.

(a) Show a graph G with two of its bottleneck spanning trees such that only one of them is a minimum spanning tree.

(b) Show that any minimum spanning tree is always a bottleneck spanning tree.
[This shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show that one can be found in linear time.]

(c) Give a linear-time alg l number b, determines whether
bottleneck value of a T of G is given.)
[Hint: consider the edges of G that have weig ey contain a spanning tree?]

(d) Use your algorithm for part (c) as a subroutin r-time algorithm for the
Bottleneck Spanning Tree Problem. [Hint: use prune-&-search and the hint for part (c)]

44. Let T be an arbitrary spanning tree of a weighted connected undirected graph G = (V, E, w).
Let $L(T) = \langle e_1, e_2, \ldots, e_{n-1} \rangle$ be the list of edges in T sorted in non-decreasing order of weight, that is, $w(e_1) \le w(e_2) \le \ldots \le w(e_{n-1})$, where $n = |V(G)|$.
Let T' be another spanning tree of G with $L(T') = \langle e'_1, e'_2, \ldots, e'_{n-1} \rangle$ sorted in the same manner.

(a) Prove that if T is a MST of G, then $w(e_k) \le w(e'_k)$, for all k = 1..n-1.

(b) Show that if T and T' are both MSTs of G, then $w(e_k) = w(e'_k)$, for all k = 1..n-1.

45. Let us say a connected weighted undirected graph G = (V, E, w) is a **near-tree** if $|E| \leq |V| + 8$. Design and analyze an algorithm with running time O(V) that takes a near-tree G and returns a MST of G. (You may assume that all the edge weights are distinct.)

46. Suppose you are given a directed graph G = (V, E) with arbitrary real valued costs on its edges and a sink vertex $t \in V$. Edge costs may be **negative**, **zero**, or **positive**. Assume that you are also given finite values d(v) for all $v \in V$. Someone claims that, for each vertex $v \in V$, the quantity d(v) is the cost of the shortest path from vertex v to the sink t.

(a) Give a linear-time algorithm that verifies whether this claim is correct.

(b) Assume in part (a) th
Now you are asked to $t' \in V$.
Design and analyze an O(E log V) time algor g the shortest path distances d′(v) for all vertices $v \in V$ to the new sink t.
(Remember that G may have negative cost ed

47. Show that the single source longest paths in a weighted DAG can be computed in linear time.

48. A **tournament** is a directed graph formed by taking the complete undirected graph and assigning arbitrary directions on the edges, i.e. a graph G = (V,E) such that for each $u,v \in V$, exactly one of (u,v) or (v,u) is in E. Show that every tournament has a Hamiltonian path, that is, a path that visits every vertex exactly once.

49. Color Constrained Single Source Shortest Paths:
Design and analyze an efficient algorithm for the following problem:
Input: A digraph G = (V, E) in which each node v∈V is allocated a color c(v)∈{red,blue}; and a source node s∈V.
Output: For each node v∈V, find a path from s to v that contains the minimum number of red nodes (including s and v).

50. Another Color Constrained Single Source Shortest Paths:
Modify Dijkstra's algorithm to solve the problem:
Input: A weighted digraph G = (V, E, w), in which each node v∈V is allocated a color c(v)∈{red,blue}, a source node s∈V, and a positive integer K.
Output: For each vertex v∈V, find the shortest path from s to v subject to the constraint that it does not contain more tha
[Hint: For each vertex v∈ ... tions of d[v] and π[v] in Dijkstra's algorithm from ... v and r, in order to account for not only (the weight of) the path from s to v, but also ... he number of red nodes on that path.]

51. [CLRS, Exercises 24.3-8 & 24.3-9, p. 664] Modify Dijkstra's algorithm so that it runs faster when all the edge weights are restricted to being in the integer range [0..W], where W is a given positive integer.
(a) Give a O(W V + E) time algorithm.
(b) Give a O((V+E) log W) time algorithm. [Hint: implement the priority queue by buckets.]

52. We are given the adjacency matrix of an un-weighted directed graph G = (V, E). For each pair of vertices s,t∈V, compute the number of simple paths in G from s to t.

53. **[CLRS, Exercise 24.3-6, p. 663]** We are given a digraph G = (V, E) on which each edge (u,v) ∈ E has an associated value r(u,v), which is a real number in the range 0 ≤ r(u,v) ≤ 1 that represents the reliability of a communication channel from vertex u to vertex v. We interpret r(u,v) as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Thus, the reliability of a path is the product of the reliabilities of its edges. Give an efficient algorithm to find the most reliable path between two given vertices.

54. **[CLRS, Exercise 25.3-4, p. 705]** Suppose instead of the Edmonds-Karp & Johnson's edge weight transformation, we use the following simpler and faster strategy:
Let $w^* = \min \{ w(u,v) \mid (u,v) \in E(G) \}$.
Define $\hat{w}(u,v) = w(u,v) - w^*$ for all edges $(u,v) \in E(G)$.
What is wrong with this edge weight transformation?

55. **A Variation of the Sing**
Input: a digraph G with profit labels on nodes an ges; and a source s∈V(G).
Define **Cost(path P)** := sum of costs of edges on rofits of nodes on P.
Design and analyze an efficient algorithm for this SP problem.
[Hint: Transform it to the standard SSSP problem.]

56. Single-source bottleneck paths problem (SSBPP): Given a weighted digraph G =(V, E, w), define the **bottleneck cost** of a path to be the maximum of the weights of the edges on the path. The optimum bottleneck path from a vertex s to a vertex t is a path from s to t in G with minimum bottleneck cost (i.e., the heaviest edge on the path is as light as possible). The problem is: given a source vertex s∈V find the optimum bottleneck paths from s to each vertex in G.

(a) Consider the example digraph shown below.
  (i) What is the bottleneck cost of the path (a, e, f)?
  (ii) What is the optimum bottleneck path from vertex a to f?
  (iii) What is the bottleneck cost of that path?

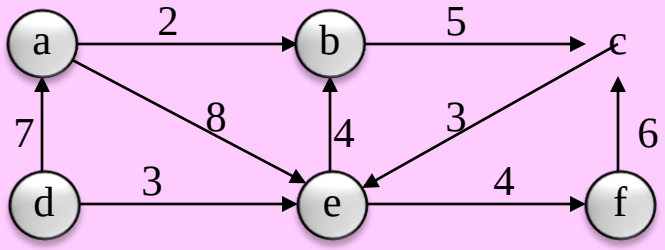
(b) Describe how to modify the initialization and the labeling step of Ford's single-source shortest paths method for the solution of SSBPP.

(c) Would the existence of negative cost edges create any complication in part (b)? Explain.

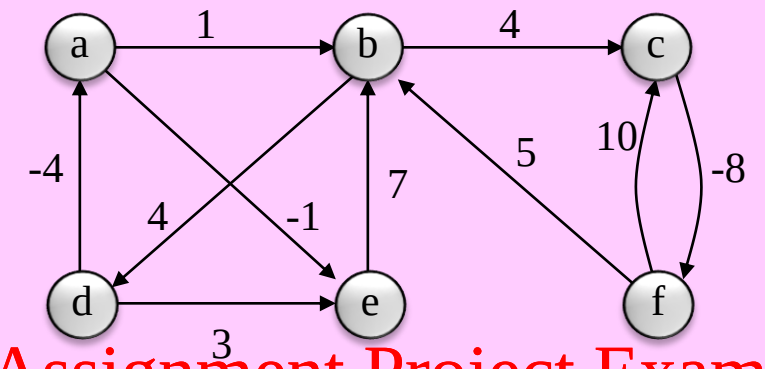
(d) Describe an algorith G is a DAG.

(e) Design and analyze a P on a general digraph.

[Hint: Modify Dijkst



57. [CLRS, Exercise 25.2-1, p. 699] Run the Floyd-Warshall algorithm on the weighted digraph shown below. Show the d and π matrices after each iteration of the outer k-loop.



58. Transitive Closure: The sure of a digraph G = (V, E), given its adjacency list stru

(a) Describe an O(VE) tim

(b) Describe an O(VE) time algorithm for a genera

[Hint: compute strongly connected components]

59. Even length paths: We are given the 0/1 adjacency matrix A[V, V] of an n vertex directed graph G = (V, E). Assume all diagonal entries of A are 0 (i.e., G has no self-loops).

We want to compute the 0/1 matrix P[V, V], such that for all vertices u and v in G,

P[u,v] = 1 if and only if there is a path of **even** length from vertex u to v in G, where the length of a path is the number of edges on it. Design and analyze an efficient algorithm for this problem.

[Hint: also think about odd length paths.]

60. [CLRS, Problem 24-2, p. 678] Nesting Boxes

A d-dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ **nests** within another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation $\pi[1..d]$ such that $x_{\pi(1)} \le y_1, x_{\pi(2)} \le y_2, \ldots, x_{\pi(d)} \le y_d$.

(a) Argue that nesting relation is transitive.

(b) Describe an efficient method to determine whether or not one box nests inside another.

(c) Suppose that you are given a set of n d-dimensional boxes $\{B(1), B(2), \ldots, B(n)\}$.
Describe an efficient algorithm to determine the longest sequence $\langle B(i_1), B(i_2), \ldots, B(i_k)\rangle$ of boxes such that $B(i_j)$ nests within $B(i_{j+1})$ for $j = 1, 2, \ldots, k-1$.
Express the running time of your algorithm in terms of n and d.

61. [CLRS, Problem 24-3, p. 679] Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of currency into more than one unit of currency. For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 yen, and 1 yen buys 0.0091 U.S. dollar. Then, by converting currency, a trader can buy 46.4×2.5×0.0091 = 1.0556 U.S. dollars, thus turning a profit of 5.56 percent.

Suppose that we are given n currencies c(1), c(2), ..., c(n) and an n × n table R of exchange rates, such that one unit of currency c(i) buys R[i,j] units of currency c(j).

(a) Design and analyze an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c(i_1), c(i_2), \ldots, c(i_k)\rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

(b) Design and analyze an efficient algorithm to print out such a sequence if one exists.

62. [CLRS, Exercise 26.2-11, p. 731] Edge Connectivity

The **edge connectivity** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how the edge connectivity of an undirected graph G = (V, E) can be determined by running a maximum flow algorithm on at most |V| flow networks, each having O(V) vertices and O(E) edges.

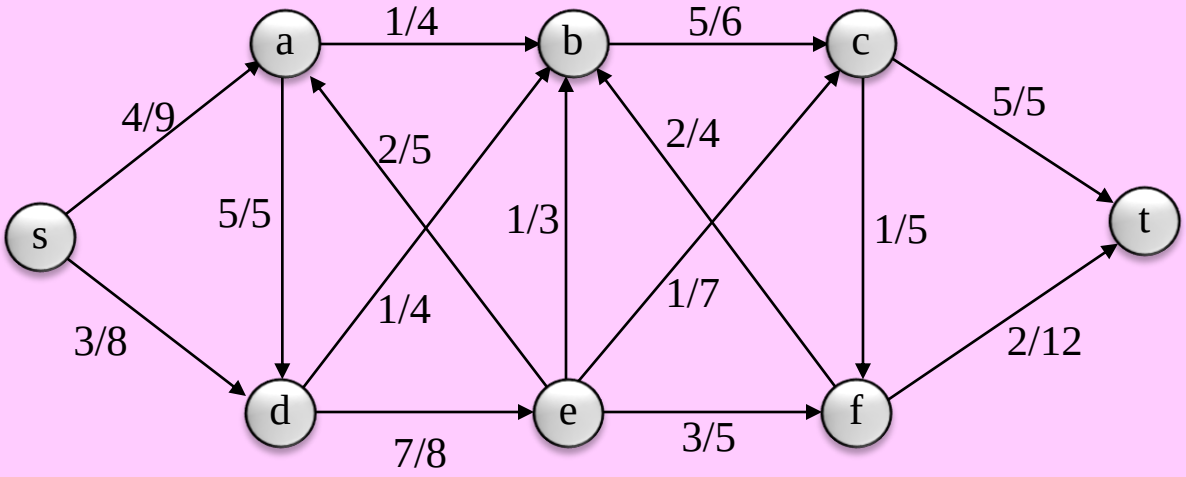
63. Max Flow Example: Consider the flow network shown below with source s and terminal t. The labels on edges are of the form "f / c" where f is the flow and c is the capacity of the edge. This flow is feasible.

Start from this given feasible flow and augment it to a max-flow using augmenting iterations of Ford-Fulkerson's max-flow-min-cut algorithm.
(In each iteration you are f

(a) Clearly show each aug

(b) Show the final max-fl w from s to t?

(c) Show the minimum st-cut. What is the min-cut



64. Max Flow Min Cut:

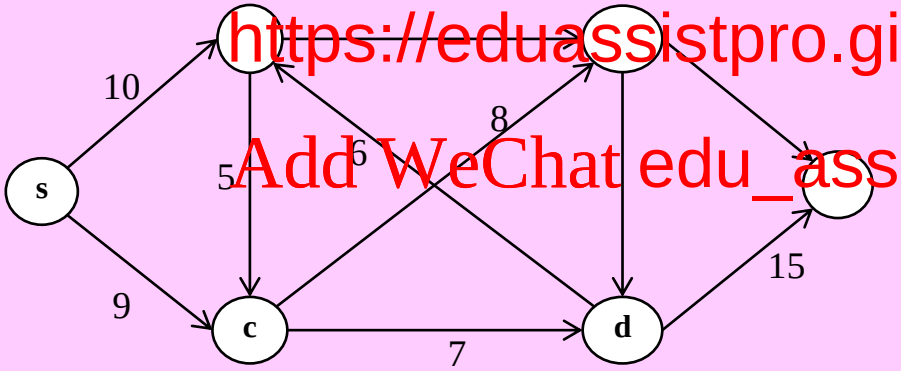
a) Below we have an instance of a flow network with the source s and terminal t and edge capacities shown.
Show the max flow and the min cut on the same figure.
Also give the value of that flow and the capacity of that st-cut.

Now answer parts (b) – (d) below for a general Flow Network G = (V, E, c, s, t):

b) Given a flow f for a flow network G, how would you algorithmically determine whether f is a feasible flow for G in O(|V| + |E|) time?

c) Given a feasible flow f for a flow network G, how would you algorithmically determine whether f is a max flow for G in O(|V| + |E|) time?

d) Given a max flow f for a flow network G, how would you algorithmically determine a min capacity st-cut for G in O(|V| + |E|) time?



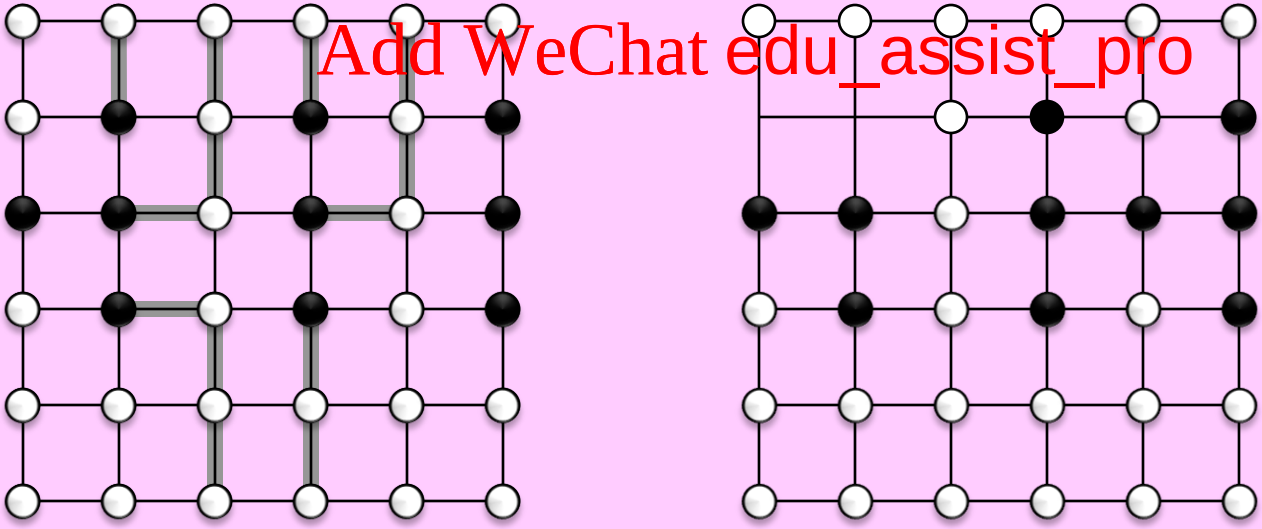
65. [CLRS, Problem 26-1, p. 760] Escape Problem

An n×n **grid** is an undirected graph consisting of n rows and n columns of vertices, as shown below. We denote the vertex in row i and column j by (i,j). All vertices in a grid have exactly 4 neighbors, except for the boundary vertices, which are points (i,j) for which i=1, i=n, j=1 or j=n.

Given $m \le n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the **escape problem** is to determine whether or not there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Fig (a) below has an escape, but the one in Fig (b) does not.

(a) Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced (or transformed) to an ordinary flow network of comparable size.

(b) Describe and analyze an efficient algorithm to solve the escape problem.



(a) (b)

66. [CLRS, Problem 26-2, p. 761] Minimum Path Cover

A path cover of a digraph G = (V, E) is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including 0. A **minimum path cover** of G is a path cover containing the fewest possible paths.

(a) Give an efficient algorithm to find a minimum path cover of a DAG G = (V, E).

[Hint: Construct the graph G' = (V', E'), where

$V' = \{s, t\} \cup \{x_u \mid u \in V\} \cup \{y_v \mid v \in V\}$,

$E' = \{(s, x_u) \mid u \in V\} \cup \{(y_v, t) \mid v \in V\} \cup \{(x_u, y_v) \mid (u,v) \in E\}$,

and run a max flow algorithm.]

(b) Does your algorithm w

xplain

67. [CLRS, Problem 26-4, p. 762] Updating Maxim

Let G = (V, E) be a flow network with source s, si pacities. Suppose that we are given a maximum flow in G.

(a) Suppose that the capacity of a single edge (u,v)∈E is increased by 1. Give an O(V+E) time algorithm to update the max flow.

(b) Suppose that the capacity of a single edge (u,v)∈E is decreased by 1. Give an O(V+E) time algorithm to update the max flow.

68. [CLRS, Exercise 26.3-4, p. 735] Perfect Matching

A **perfect matching** is a matching in which every vertex is matched. Let G = (V, E) be an undirected bipartite graph with vertex bipartition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the neighborhood of X as

$$N(X) = \{ y \in V \mid (x,y) \in E \text{ for some } x \in X \},$$

that is, the set of vertices adjacent to some member of X. Prove **Hall's Theorem**: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

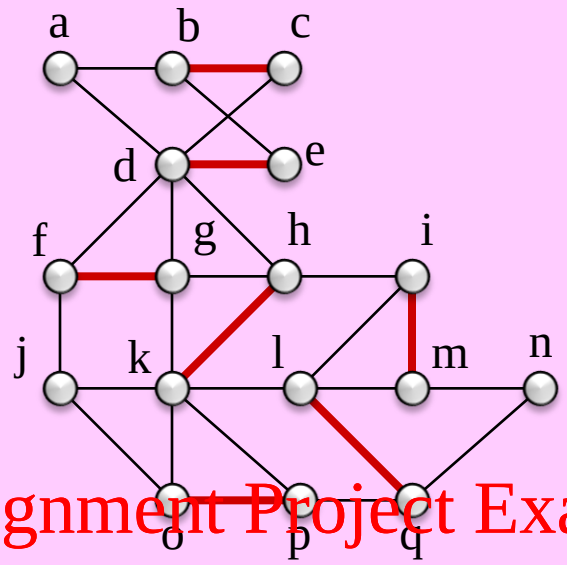
69. A Dancing Problem: There are n boys and n girls. Each boy knows exactly k girls, and each girl knows exactly k boys, for some positive integer k.
Assume "knowing" is symmetric: boy b knows girl g $\Leftrightarrow$ g knows b.
We can represent this by t hich each boy and each girl is represented by a vertex, an tween boy b and girl g if and only if b and g know each

$\forall v \in V(G)$:

(a) Show that G has a perfect matching.
Interpretation: we can pair boys and girls into n pairs for a paired dancing contest.
[Hint: use the max flow integrality theorem.]

(b) Show that E(G) is the disjoint union of exactly k perfect matchings.
Interpretation: We can have k dancing rounds, where in each round every one dances with a dancing partner of the opposite sex that he/she knows, and no one dances with the same partner in two different rounds.

(c) Suppose we change the above equality
"$\forall v \in V(G)$: degree(v) = k" to "$\forall v \in V(G)$: degree(v) $\geq$ k".
In the latter case, demonstrate one such graph G that has no perfect matching.



70. Does the matching shown i inality?
If not, augment it to a max
(a) You may find augmenting paths by visual insp
(b) Now find augmenting paths using BFS in the a
and blossom shrinking & expansion.

71. Maximum Cardinality Matching in a tree
Design and analyze an O(n)-time algorithm that finds a maximum cardinality matching in a given tree with n nodes. Carefully prove correctness & optimality.

72. A **node cover** C of a graph G = (V, E) is any subset of V such that every edge in E is incident to some node in C. Suppose G is a bipartite graph, M is a maximum cardinality matching, and C is a minimum cardinality node cover of G. Show that |M| = |C|.

73. Perfect Matching: Reduction from general graphs to degree 3 graphs

Consider the following reduction of the matching problem from general graphs to degree 3 graphs. Given a graph G with n nodes, construct a graph G' as follows. For every node i of G take the smallest complete binary tree with at least n leaves and insert a node in the middle of every edge; let T_i be the resulting tree. The graph G' has a tree T_i for every node i of G, and for every edge (i, j) of G, G' has an edge connecting the j^{th} leaf of T_i to the i^{th} leaf of T_j .

a) Show that, for any i, T_i minus any one of its leaf nodes contains a perfect matching.
b) Show that G has a perfect matching if and only if G' has a perfect matching.
c) Verify that the above reduction takes polynomial time.

74. Suppose that a given graph G = (V, E) has no isolated nodes.
An **edge cover** of G is a sub incident to some edge in C.

a) Show that if C is a min M is a maximum cardinality matching of G, then |
b) Give an efficient algorithm to find a minimum over of a graph.

75. If G = (V, E) is a graph with V = {1,2, …, n}, let us **atrix** of G, T(G), to be the n×n matrix defined as follows:

$$T(G)_{ij} = \begin{cases} +x_{ij} & \text{if } (i,j) \in E \ \& \ i > j \\ -x_{ij} & \text{if } (i,j) \in E \ \& \ i < j \\ 0 & \text{otherwise} \end{cases}$$

where x_{ij}'s are indeterminates (i.e., variables).

Show that G has a complete matching (i.e., a matching with at most one unmatched node) if and only if $\det(T(G)) \neq 0$.

76. An undirected graph $G = (V, E)$ is called *d-regular* if all its vertices have degree d.

a) Show that any d-regular bipartite graph with $d > 0$ has a perfect matching.

b) Show a 3-regular graph that has no perfect marching.

77. Let J = {J_1 , J_2 , …, J_n} be a set of jobs to be executed on two processors with all jobs requiring the same amount of time (say 1). Suppose that there is a directed acyclic graph P = (J, E), called the **precedence graph**, such that if (J_1 , J_2)∈E, then J_1 must be executed before J_2.

The **Two-Processor Scheduling Problem with Precedence Constraints** is:

Given P, find an optimal sc …, T} such that

(i) For all t ≤ T, |{J_k

(ii) if (J_1 , J_2)∈E the

1 2

(iii) T is as small as possible

a) Consider the undirected graph G_P = (J, E_P), where (J_1 , J_2)∈E_P if and only if there is no path from J_1 to J_2 , nor one from J_2 to J_1 , in P. Suppose that M is a maximum cardinality matching of G_P. Show that the smallest T achievable must obey

$$T \geq |J| - |M|.$$

b) Show that there is always a schedule with $T = |J| - |M|$.

c) Give an efficient algorithm for finding the optimal schedule.