22. Elementary Graph Algorithms

22.1 # Graph Representation:

Given graph $G = (V, E)$ → May be either Directed or Undirected

- Sparse Graph: $|E|$ is much less than $|V|^2$
- Dense Graph: $|E|$ is close to $|V|^2$

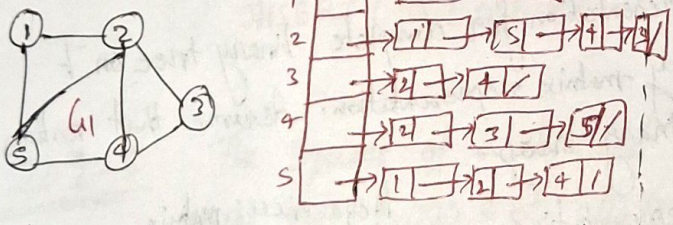
Representation of Graph:
1. Adjacency lists
2. Adjacency matrix

① Adjacency lists:

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

→ Vertex u's list has all vertices v such that $(u,v) \in E$. (directed / undirected).

Ex:- ① Undirected Graph



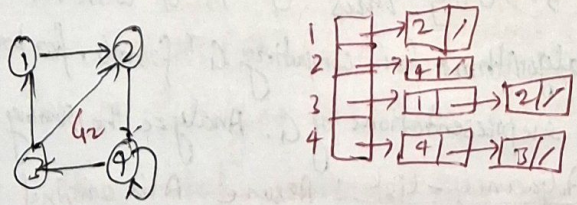
If edge have weights, can put the weights in the lists.; Weight: $w: E \to R$

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

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

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

② Directed Graph



$\Theta(V+E)$ & $\Theta(\text{degree}(u))$

↓ SC. ↓ Time complexity.

* Same asymptotic space & time.

② Adjacency Matrix:

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

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

Ex:- ① Undirected Graph (G_1)

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

② Directed Graph (G_2)

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

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

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

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

Exercises:

22.1-1 Given an adjacency-list representations of a directed Graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Sol":

* Time to compute the outdegree = $\sum_{v \in V} O(\text{out-degree}(v)) = O(|E|+|V|)$

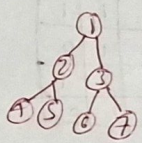
* In-degree, we have to scan through all adjacency lists & keep counters for how many times each vertex has been pointed to. $O(|E|+|V|)$ because we'll visited all nodes & edges.

Thus,

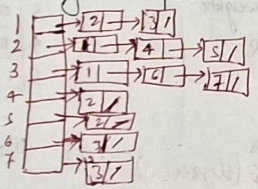
Time to compute the in-degree = $\sum_{v \in V} \Theta(\text{in-degree}(v)) = \Theta(|E|+|V|)$

22.1-2 Given an adjacency-list representation for complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in binary heap.

Sol":



- Adjacency-list representation



- Adjacency-matrix

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	1	1	0	0
3	1	0	0	0	0	1	1
4	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0
7	0	0	1	0	0	0	0

22.1-3 The transpose of a directed graph $G=(V,E)$ is the $G^T=(V,E^T)$ where, $E^T = \{(u,v) \in V \times V : (u,v) \in E\}$. Thus G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G, for both the adjacency-list & adjacency matrix representations of G. Analyze the running time of your algo.

Sol":

- Adjacency Matrix

Transpose the original matrix by looking along every entry above the diagonal & swapping it with the entry that occurs below the diagonal. Time complexity = $O(V^2)$

- Adjacency-list: Assume A is original adjacency lists, $A'[1,\ldots,|V|]$ be the (new adj. list) of the transposed G^T.

$A \to G$

$A' \to G^T$

```
for each vertex u ∈ G.V
    for each vertex v ∈ A[u]
        INSERT(A'[v], u)
```

Time Complexity = $O(E+V)$

22.1-4 Given an adjacency-list representation of a multigraph $G=(V,E)$, describe an $O(V+E)$ time algo. to compute the adjacency-list representation of the "equivalent" undirected graph $G'=(V,E')$, where E' consists of the edges in E with all multiple edges b/w two vertices replaced by single edge & with the self-loops removed.

Sol":

```
EQUIVALENT-UNDIRECTED-GRAPH -
A' → new adj. list
A → 0-initialized array of size|V|
for each vertex u ∈ G.V
    for each v ∈ A[u]
        if each v != u && A[v] != u
            A[v] = u
            INSERT(A'[u], v)
```

Note that A does not contain any element with value u before each iteration of ~~linear~~ inner for-loop. Since That's why we use A[v]=u to mark the existance of an edge (u,v) in the inner for-loop. Since we lookup in adjacency-list A for |V|+|E| times, the time complexity is O(V+E).

22.1-5 The square of a directed graph $G=(V,E)$ is the graph $G^2=(V,E^2)$ such that $(u,v) \in E^2$ iff G contains a path with atmost two edges b/w u & v. Describe efficient algo. for computing G^2 from G for both the adjacency-list & adjacency-matrix representations of G. Analyze the running times of your algo.

Sol":

- Adjacency-list

```
for each v ∈ A[u]
    INSERT(A²[u,v])
    for each w ∈ A[v]
        // edge (u,w) ∈ E²
        INSERT(A²[u], w)
```

Where, A → original adjacency list of A^2 → new adjacency-list

For edge in A we scan at most |V| vertices, we compute A^2 in time $O(|V||E|)$.

After the computation of A^2, we have to remove duplicate edges from the lists, Removing duplicate edges is done in $O(V+E')$ where $E'=O(VE)$ is the no. of edges in A^2 as shown in exercise 22.1-4.

Thus total running time = $O(VE) + O(V+VE) = O(VE)$

- Adjacency-matrix

Let A denote the adjacency-matrix representation of G. The adj. matrix of G^2 is the square of A. Computing A^2 can be done in time $O(V^3)$ (& even faster, theoretically, Strassen's algorithm for example will compute A^2 in $O(V^{2.81})$)

22.1-6 Most graph algo. that take an adj. matrix representation as I/P required time $\Omega(V^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a universal sink — a vertex with in-degree $|V|-1$ & out-degree 0 — in time $O(V)$, given an adjacency-matrix for G.

Sol: Start by examining position (1,1). in the adjacency matrix. When examining position (i,j):

→ If a 1 is encounted, examine position (i+1,j) &

→ If a 0 is encounted, examine position (i,j+1)

once either i or j is equal to |V|, terminate.

If a graph contains a universal sink, then it must be at vertex i.

Suppose, vertex k is a universal sink. Since k is a universal sink, row k will be filled with 0's & column k will be filled with 1's except for M[k,k], which is filled with a 0.

```
IS_CONTAIN_UNIVERSAL_SINK(M)
  i = j = 1
  while i < |V| & j < |V|
  // There's an out-going edge, so examine next row
    if M[i,j] == 1
      i = i+1
  // There's no out-going edge, so see if we could reach the last column of the current row
    else if M[i,j] == 0
      j = j+1
  check if vertex i is a universal sink.
```

This algo runs in $O(V)$ & checking if vertex i is universal sink is done in $O(V)$.

∴ Total running time = $O(V)+O(V) = O(V)$.

22.1-7 The incidence matrix of directed graph $G=(V,E)$ with no self-loops is a $|V| \times |E|$ matrix $B=(b_{ij})$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i, \\ +1 & \text{if edge } j \text{ enters vertex } i, \\ 0 & \text{otherwise} \end{cases}$$

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

Sol:

$$BB^T(i,j) = \sum_{e \in E} b_{ie} b_{ej}^T = \sum_{e \in E} b_{ie} b_{je}$$

→ If $i=j$, then $b_{ie}b_{je} = 1$ (it is 1·1 or (−1)·(−1)) whenever e enters or leaves vertex i, & 0 otherwise

→ If $i \neq j$, then $b_{ie}b_{je} = -1$ when $e=(i,j)$ or $e=(j,i)$ & 0 otherwise.

Thus, $BB^T(i,j) = \begin{cases} \text{degree of } i = \text{in-degree} + \text{out-degree} & \text{if } i=j \\ -(\#\text{ of edges connected } i \text{ \& } j) & \text{if } i \neq j \end{cases}$

22.1-8 Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u,v) \in E$. If all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suppose we alternative have disadvantages compared to the hash table?

Sol: The expected lookup time is $O(1)$, but in the worst case it could take $O(|V|)$.

If we first sorted vertices in each adjacency list then we could perform a binary search so that the worst case lookup time is $O(\log|V|)$, but this has the disadvranges of having a much worse expected lookup time.

22.2 Breadth-First Search

→ Prim's minimum-spanning-tree Algo
→ Dijkstra's Single-source shortest-paths Algo

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

Output: $d[v]$ = distance (smallest # of edges) from s to v, $\forall v \in V$.

- u is v's predecessor.
- u is Ancestor of v & v is descendant of u.
- Set of edges $\{(u.\pi, v) : v \neq s\}$ form a tree.

BFS is so named because it expands the frontier b/w discovered & undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance 'k' from 's' before discovering any vertices at distance 'k+1'.

→ $(u,v) \in E$ & vertex u is black, then vertex v is either gray or black. ↳ discovered
Undiscovered vertices white.

→ Gray vertices may have some adjacent white vertices; they represent the frontier b/w discovered & undiscovered vertices.

```
BFS(G,s)        // G=(V,E), s=source vertex.
1  for each vertex u ∈ G.V - {s}   // u ∈ V
2      u.color = WHITE             // Undiscovered.
3      u.d = ∞                     // ∞ distance
4      u.π = NIL                   // if u has no predecessor (Ex:- u=s or u has not been discovered)
5  s.color = GRAY                  // frontier b/w discovered & undiscovered
6  s.d = 0
7  s.π = NIL
8  Q = ∅                           // Queue to manage the set of gray vertices   → O(V)
9  ENQUEUE(Q,s)                    → O(1)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)              → O(1)
12     for each v ∈ G.Adj[u]       → O(E)
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q,v)
18 u.color = BLACK.
```

Initialization

Sum of the lengths of all adj. list.

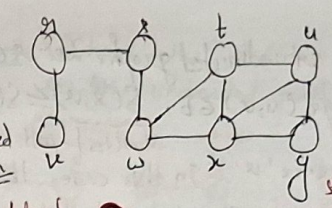
→ iterate as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency list fully examined.

Total Time = $O(1) + O(1) + O(V) + O(E)$

T.C. = $O(V+E)$

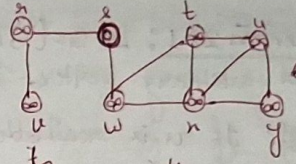
↓ linear in size of Adj. list.

Ex:- Given Undirected Graph



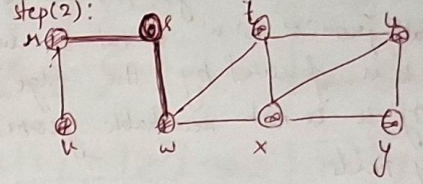
black → ●
Gray → ◎ or ▨
White → ○

Step(1):



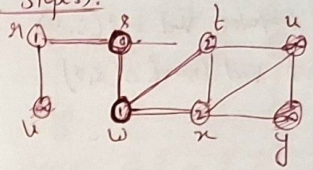
Q [s]

Step(2):



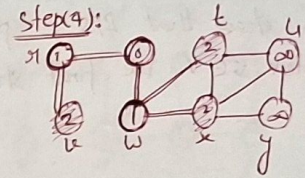
Q | r | w |
  | 1 | 1 |

Step(3):



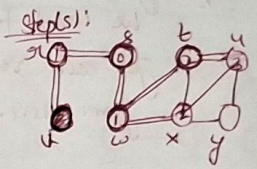
Q | r | t | x |
  | 1 | 2 | 2 |

Step(4):



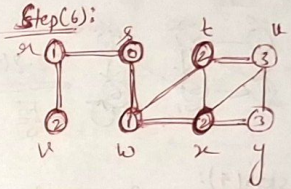
Q | t | x | v |
  | 2 | 2 | 2 |

Step(5):



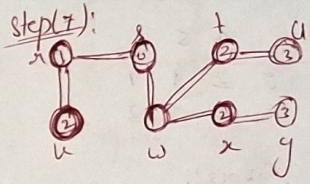
Q | x | v | w |

Step(6):



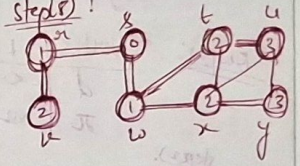
Q | v | w | y |

Step(7):



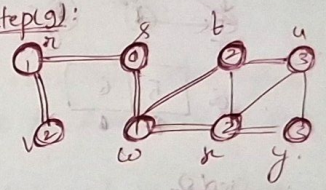
Q | w | y |

Step(8):



Q | y |

Step(9):



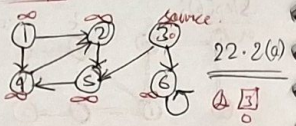
Lemma 22.1: Let $G=(V,E)$ be directed or undirected graph, let $s \in V$ be an arbitrary vertex. Then, for any edge $(u,v) \in E$, $\delta(s,v) \leq \delta(s,u)+1$.
(shortest-path distance)

Proof: If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u,v) & thus the inequality holds. If u is not reachable from s, then $\delta(s,u)=\infty$ & the inequality holds.

We want to show that BFS properly computes $v.d = \delta(s,v)$ for each vertex $v \in V$. We first show that $v.d$ bounds $\delta(s,v)$ from above.

Exercises

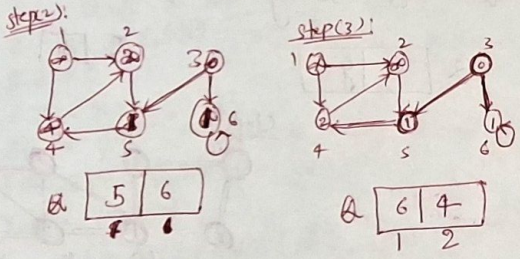
22.2-1 Show the d & π values that result from running BFS on the directed graph of Figure 22.2(a), using vertex 3 as the source.

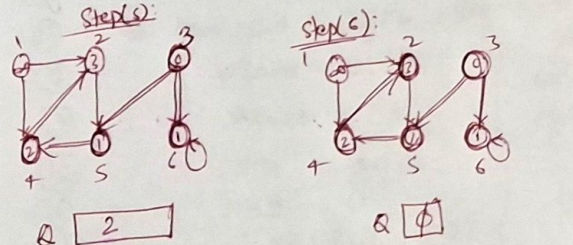


Output:

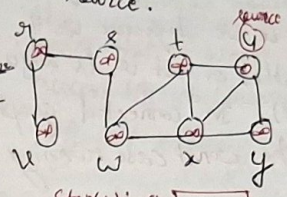
Vertex	1	2	3	4	5	6
d	∞	3	0	2	1	1
π	NIL	4	NIL	5	3	3

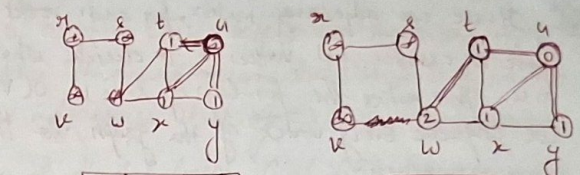
Sol":

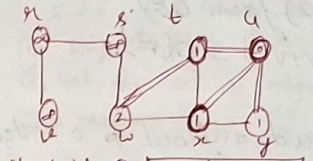


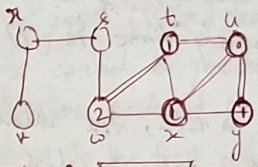


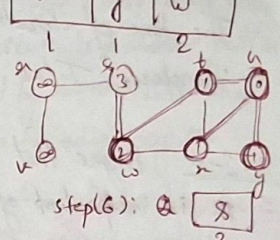
22.2-2 Show the d & π values that result from running BFS on the undirected graph of figure 22.3, using vertex u as the source.

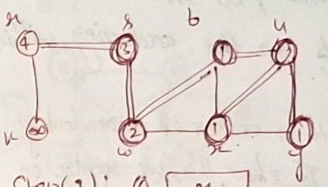


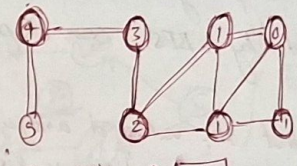


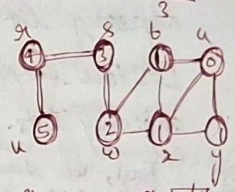












Vertex	z	s	t	u	v	w	x	y
d	4	3	1	0	5	2	1	1
π	s	w	t	NIL	z	t	u	u

22.2-3 Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if lines 5 & 14 were removed.

Sol": The GRAY color for the pedagogical purpose to distinguish b/w the GRAY nodes (which are enqueued) & the BLACK nodes (which are dequeued). Therefore, it suffices to use a single bit to store each vertex color.

22.2-4 What is the running time of BFS if we represent its I/p graph by an adjacency matrix & modify the algorithm to handle this form of I/p?

Sol^n If we use adjacency matrix, for each vertex u we dequeue we'll have to examine all vertices v to decide whether or not v is adjacent to u. This makes the for-loop of line 12 $O(V)$. In connected graph we enqueue every vertex of the graph, so the worst case running becomes $O(V^2)$.

The time of iterating all edges becomes $O(V^2)$ from $O(E)$.

Therefore, Total running time is $O(V+V^2) = O(V^2)$.

22.2-5 Argue that in a BFS, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using Fig. 22.3 as an example, show that the BFS tree computed by BFS can depends on the ordering within adjacency lists.

Sol^n First, we show that the value d assigned to a vertex is dependent of the order that entries appear in adjacency lists. To show this, we rely on theorem 22.5, which proves correctness of BFS. In particular, the theorem states that $v.d = \delta(s,v)$ at the termination of BFS. Since $\delta(s,v)$ is property of the underlying graph, for any adjacency list representation of the graph (including any reordering of the adjacency lists), $\delta(s,v)$ will not change. Since the d values are equal to $\delta(s,v)$ & $\delta(s,v)$ is invariant for any ordering of the adjacency list, d is also not dependent of the ordering of the adjacency list.

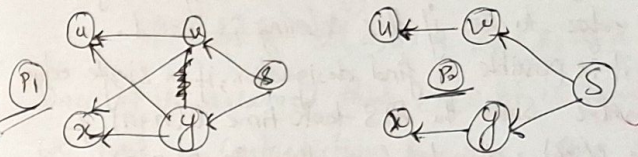
Now, to show that π does depend on the ordering of the adjacency lists, we will be using Figure 22.3 as a guide.

First, we note that in the given worked out procedure, we have that in the adjacency list for w, t precedes x. Also in the worked out procedure, we have that $u.\pi = t$.

Now, suppose instead that we had x preceding t in adjacency list of w. Then, it would get added to the queue before t, which means that it would u as it's child before we have a chance to process the children of t. This will means that $u.\pi = x$ in this different ordering of adjacency list of w.

22.2-6 Give an example of directed Graph $G=(V,E)$, a source vertex $s \in V$, & a set of tree edge $E_\pi \subseteq E$ such that for every vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π can not be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

Sol^n Let G be the graph shown in the first picture (1), $G_\pi = (V, E_\pi)$ be the graph shown in the second picture (2) & s be the source vertex.



We could see that E_π will never be produced by running BFS on G.

→ If y precedes w in the Adj[s], we'll dequeue y before w, So, $u.\pi$ & $x.\pi$ are both y. However, this is not the case.

→ If w precedes y in the Adj[s]. We'll dequeue w before y. So $u.\pi$ & $x.\pi$ are both w, which again isn't true.

Nonetheless, the unique simple graph in G_π from s to any vertex is a shortest path in G.

22.2-7 There are two types of professional wrestlers: "babyfaces" ("good guys") & "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose, we have n professional wrestlers & we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n+r)$-time algo. that determines whether it is possible to designate some of the wrestlers as babyfaces & the

reminders as heels such that each rivalry is b/w a babyface & a heel. If it is possible to perform such a desinact? your algo should produce it.

Sol'n: This problem is basically just a obfuscated version of 2 coloring. We will try to color the vertices of this graph of rivalries by 2 colors. "babyfaces" & "heel". To have that no 2 babyfaces & no two heels have rivalry is the same as saying that the coloring is proper. To 2 color, we perform BFS of each connected component to get the d values for all each vertex. Then, we give all the odd ones one color say "heel" & all even d values a different color. We know that no other coloring will succeed where this one fails since if we gave any other coloring, we would have that a vertex v has the same color as v.π since v & v.π must have diff. parities for their d-values. Since, we know that there is no better coloring, we just need to check each edge to see if this coloring is valid. If each edge works, it is possible to find designation, if a single edge fails, then it is not possible. Since the BFS took time $O(n+r)$ & the checking took time $O(r)$, the total runtime is $O(n+r)$.

*22.2-8 The diameter of a tree $T=(V,E)$ is defined as $\max_{u,v \in V} \delta(u,v)$ that is, the largest of all shortest-path distances in the tree. Give an efficient algo. to compute the diameter of a tree & analyze the running time of your algo.

Sol'n: Suppose that a & b are the endpoints of the path in the tree which achieve the diameter & without loss of generality assume that a & b are the unique pair which do so. Let s be any vertex ≠ T. We claim that the result of ~~the~~ a single BFS will return either a or b (or both) as the vertex whose distance from s is greatest.

To see this, suppose to the contrary that some other vertex x is shown to be further from s.

$$d(s,a) < d(s,x) \quad \& \quad d(s,b) < d(x,x).$$

Let c denote, the vertex on the path from a to b which minimizes $d(s,c)$. Since the graph is in fact a tree, we must have

$$d(s,a) = d(s,c) + d(c,a) \quad \& \quad d(s,b) = d(s,c) + d(c,b).$$

(If there were another path, we could form a cycle). Using the triangle inequality & inequalities & equalities mentioned above we must have

$$d(a,b) + 2d(s,c) = d(s,c) + d(c,b) + d(s,c) + d(c,a).$$
$$< d(s,x) + d(s,c) + d(c,b).$$

It claim that $d(x,b) = d(s,x) + d(s,b)$. If not, then by the triangle inequality we must have a strict less-than. In other words, there is some path from x to b which does not go through c. This gives the contradiction, because it implies there is a cycle formed by concatenating these paths. Then we have

$$d(a,b) < d(a,b) + 2d(s,c) < d(x,b).$$

Since it is assumed that $d(a,b)$ is maximal among all pairs, we have a contradiction. Therefore, since trees have $|V|-1$ edges, we can run BFS a single time $O(V)$ to obtain one of the vertices which is the endpoint of the longest simple path contained in the graph. Running BFS again will show us where the other one is, so we can solve the diameter for trees in $O(V)$.

22.2-9 Let $G=(V,E)$ be a connected, undirected graph. Give an $O(V+E)$-time algo. to compute a path of G that traverses each edges in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

Sol'n: First, the algo. computes a minimum spanning tree of the graph. It can also done by performing BFS & restricting to the edges b/w v & v.π for every v. To avoid it's not double counting.

edges, fix any ordering $\leq$ on the vertices before hand. Then, we will construct the sequence of steps by calling MAKE-PATH(s), where, s was the root used for the BFS.

```
MAKE-PATH(u)
    for each v ∈ Adj[u] but not in the tree such that u ≤ v
        · go to u & back to u
    for each v ∈ Adj[u] but not equal to u.π
        · go to v
          perform the path prescribed by MAKE-PATH(v)
    go to u.π.
```

22.3 Depth-First Search

→ Topological Sort
→ DAG
→ Strongly connected components

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

Output: 2 timestamps on each vertex:

- $u.d$: $d[v]$ = discovery time
- $u.f$: $f[v]$ = finishing time

- WHITE = undiscovered
- GRAY = discovered, but not finished
- BLACK = finished
- $1 \leq d[v] < f[v] \leq 2|V|$.

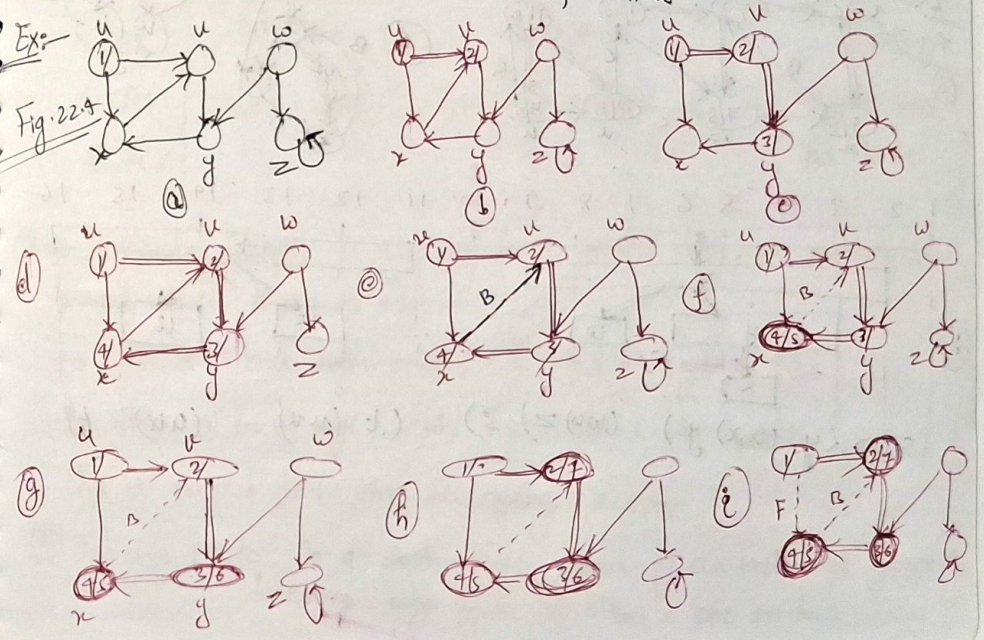
(timestamps)

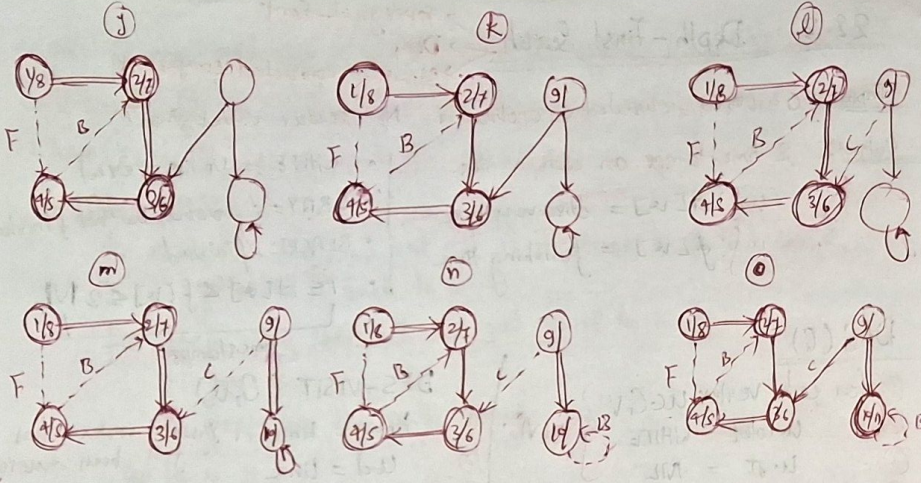
```
DFS(G)
1  for each vertex u ∈ G.V                 } Θ(V)
2      u.color = WHITE
3      u.π = NIL
4  time = 0 // global time counter
5  for each vertex u ∈ G.V                 } Θ(V)
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

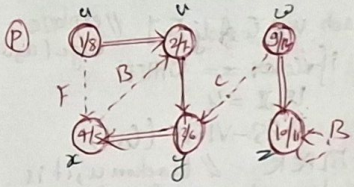
Total running time = $\Theta(V) + \Theta(V) + \Theta(E) = \Theta(V+E)$

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

Ex:



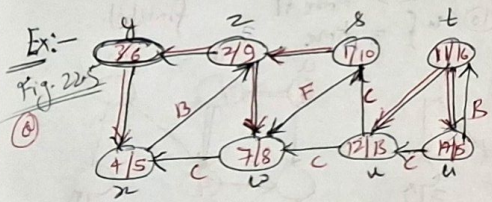


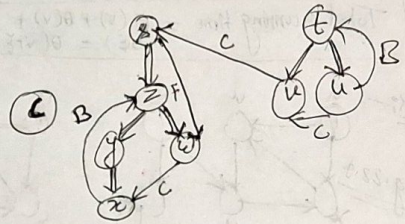


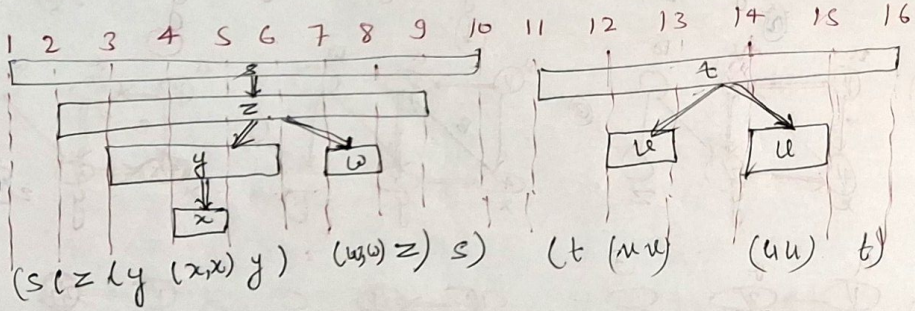
(u/v) → discovered time / finishing time

F, B, & C → Forward, back & cross edge

Ex:- Fig. 22.5







Exercises

22.3-1 Make a 3-by-3 chart with row & column labels WHITE, GRAY & BLACK. In each cell (i,j) indicate whether, at any point during a DFS of a directed graph, there can be an edge from a vertex of color i to a vertex color j. For each possible edge, indicate what edge types it can be. Make a second such chart for DFS of an undirected graph.

Soln: N/c to Theorem 22.7 (parenthesis theorem), there are 3 cases of relationship b/w intervals of vertex u & v:

① $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ ⇒ $[u.d, u.f]$ & $[v.d, v.f]$ are entirely disjoint. (neither of u & v is a descendant of each other)

② $[u.d, u.f] \subset [v.d, v.f]$ OR → $d[u] < d[v] < f[v] < f[u]$ (u is a descendant of v)

③ $[v.d, v.f] \subset [u.d, u.f]$ OR → $d[v] < d[u] < f[u] < f[v]$ (v is a descendant of u)

① ⇒ Directed Graph: We can use the edge classification given by exercise 22.3-5 to simplify the problem.

from \ to	WHITE	GRAY	BLACK
WHITE	All kinds	Cross, Back	Cross
GRAY	Tree, Forward	Tree, Forward, Back	Tree, Forward, Cross
BLACK	—	Back	All kinds.

② ⇒ Undirected Graph: Starting from directed chart, we remove the forward edge & the cross edge, &

→ When a back edge exist, we add a tree ~~graph~~ edge;

→ When a tree edge exist, we add a back edge.

This is correct for the following reasons:

④ Theorem 22.10: In a depth-first search of an undirected graph every edge of G is either a tree or back edge

(b) If (u,v) is a tree edge from u's perspective, (u,v) is also a back edge from v's perspective.

from \ to	WHITE	GRAY	BLACK
WHITE	—	Tree, Back	Tree, Back
GRAY	Tree, Back	Tree, Back	Tree, Back
BLACK	Tree, Back	Tree, Back	—

22.3-2: Show how depth-first search work on the graph of Fig. 22.6 Assume that the for loop of lines 5-7 of the DFS procedure considers the vertices in aphabetical orders, & assume that each adjacency list is ordered alphabetically. Show the discovery & finishing times for each vertex, & show the classification of each edge.

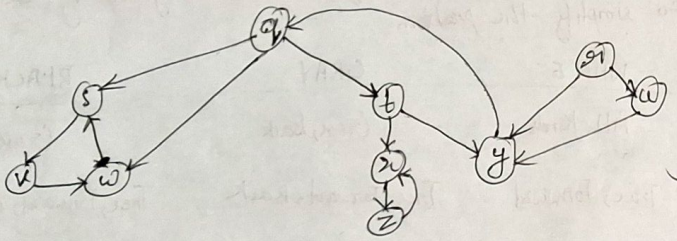
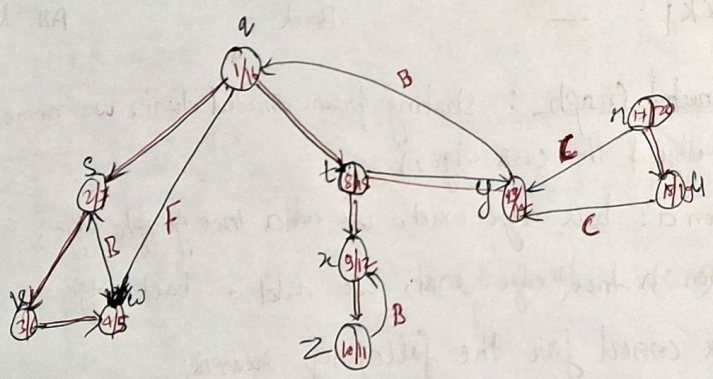


Fig. 22.6

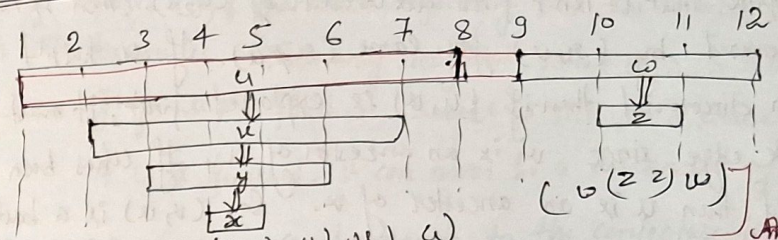
Soln



Vertex	Discovered	Finished
q	1	16
r	17	20
s	2	7
t	8	15
u	18	19
v	3	6
w	4	5
x	9	12
y	13	14
z	10	11

- Tree edges: $(q,s),(s,v),(v,w),(q,t),(t,x),(x,z),(t,y),(r,u)$.
- Back edges: $(w,s),(z,x),(y,q)$
- Forward edges: (q,w)
- Cross edges: $(r,y),(u,y)$.

22.3-3 Show the parenthesis structure of the DFS of figure 22.4



$(u\,(v\,(y\,(x\,x)\,y)\,v)\,u)\;\;(w\,(z\,z)\,w)$

Ans.

22.3-4 Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure would produce the same result if line-3 of DFS_VISIT was removed

Ans: change line 3 to color = BLACK & remove line-8. Then, the algo. would produce the same result.

22.3-5 Show that edge (u,v) is

(a) a tree edge or forward edge iff $u.d < v.d < v.f < u.f$

(b) a back edge iff $v.d \le u.d < u.f \le v.f$, &

(c) a cross edge iff $v.d < v.f < u.d < u.f$.

Ans: (a) u is an ancestor of v

(b) u is a descendant of v

(c) v is visited before u.

22.2-6 : Show that in an undirected graph, classifying an edge (u,v) as a tree edge or back edge according to whether (u,v) or (v,u) is encountered first during the DFS is equivalent to classifying it according to the ordering of the four types in the classification scheme.

Sol^n : By theorem 22.10, every edge of an undirected graph is either a tree or a back edge. First suppose that v is first discovered by exploring edge (u,v). Then by definition, (u,v) is a tree edge. Moreover, (u,v) must have been discovered before (v,u) because once (v,u) is explored, v is necessarily discovered. Now, suppose that v isn't first discovered by (u,v). Then it must be discovered by (x,v) for some $(x \neq u)$. If u hasn't yet been discovered then if (u,v) is explored first, it must be back edge since v is an ancestor of u. If u has been discovered then u is an ancestor of v. So, (v,u) is a back edge.

22.3-7: Rewrite the procedure DFS, using an stack to eliminate recursion.

Pseudocode:

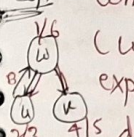
```
DFS-STACK(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if color == WHITE
        DFS-VISIT-STACK(G,u)
```

```
DFS-VISIT-STACK(G,u)
S = ∅
PUSH(S,u)
time = time + 1
u.d = time
u.color = GRAY
while ¬ STACK-EMPTY(S)
    u = TOP(S)
    v = FIRST-WHITE-NEIGHBOR(G,u)
    if v == NIL
        // u's adjacency list has been fully explored
        POP(S)
        time = time + 1
        u.f = time
        u.color = BLACK
    else // u's adj. list hasn't been fully explored
        v.π = u
        time = time + 1
        v.d = time
        v.color = GRAY
        PUSH(S,v)
```

```
FIRST-WHITE-NEIGHBOR(G,u)
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        return v
return NIL
```

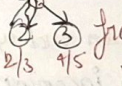
22.3-8: Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, & if $u.d < v.d$ in a DFS of G, then v is a descendant of u in the DFS-forest produced.

Sol^n Consider a graph with 3 vertices u, v & w, & with edges (w,u), (u,w) & (w,v). Suppose that DFS first explores w, & that w's adjacency list has u before v. We next discovers u. Then only adjacent vertex is w, but w is already gray so, u finishes. Since v is not yet a descendant of u & u is finished. v can never be a descendant of u.



22.3-9: Give a counterexample to the conjecture that if a directed graph G contains a path u to v, then any DFS must result in $v.d \leq u.f$.

Proof: Consider the directed graph on the vertices $\{1,2,3\}$ & having the edges $(1,2)$, $(1,3)$, $(2,1)$ then there is a path from 2 to 3. However, if we start a DFS at 1 & process 2 before 3, we will have $2.f \neq 3 \leq 4 = 3.d$ which provides a counterexample to the given conjecture.



22.3-10: Modify the pseudocode for DFS so that it prints out every edge in the directed graph G, together with its type. Show that modifications, if any, you need to make if G is undirected.

Ans If G is undirected we don't need to make any modification

```
DFS-VISIT-PRINT(G,u)
    time = time+1
    u.d = time
    u.color = GRAY
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            print "(u,v) is a tree edge.";
            v.π = u
            DFS-VISIT-PRINT(G,v)
        else if v.color == GRAY
            print ("(u,v) is back edge.");
        else if u.d > v.d
            print ("(u,v) is a forward edge.");
        else
            print ("(u,v) is cross edge.");
    u.color = BLACK
    time = time+1
    u.f = time.
```

22.3-11: Explain how a vertex u of a directed graph can end-up in a DFS tree containing only u, even though u has both incoming & outgoing edges.

Ans: Suppose that we have a directed graph on the vertices $\{1,2,3\}$ & having edges $(1,2)$ & $(2,3)$. Then 2 has both incoming & outgoing edges.

If we pick first root to be 3, that will be in its own DFS tree. Then, we pick our second root to be 2, since the only thing it points to has already been marked BLACK, we won't be exploring it. Then, picking the last root to be 1, we don't grow up the fact that 2 is along a DFS tree even though it has both an an incoming & outgoing edge in G.

22.3-12: Show that we can use a DFS of an undirected graph G to identify the connected components of G, & that the depth-first forest contains as many trees as G has connected components. More precisely, show that to modify DFS so that it assigns to each vertex v an integer label $v.cc$ b/w 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ iff u & v are in the same connected component.

The modification work as follows: each time the if-condition of line 8 is satisfied in DFC-CC, we have a new root of a tree in the forest, so, we update it cc label to be the new value of k. In the recursive calls of DFS-VISIT-CC, we always update a descendant's connected component to agree with its ancestors.

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

```
DFS-VISIT-CC(G,u)
    time = time+1
    u.d = time
    u.color = GRAY
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.cc = u.cc
            v.π = u
            DFS-VISIT-CC(G,v)
    u.color = BLACK
    time = time+1
    u.f = time
```

*22.3-13: A directed graph $G = (V,E)$ is singly connected if $u \rightsquigarrow v$ implies that G contains atmost one simple path from u to v for all vertices $u,v \in V$. Give an efficient algo to determine whether or not a directed graph is singly connected.

Ans: This can be done in time $O(|V|+|E|)$. To do this, first perform a topological sort of the vertices. Then, we will contain for each vertex a list of it's ancestors with in-degree 0. We

We compute these lists for each vertex in the order starting from the earlier ones topologically.

Then, if we ever have a vertex that has the same degree 0 vertex appearing in the lists of two of its immediate parents, we know that graph is not singly connected. However, if at each step we have that at each step all of the parents have disjoints sets of degree 0 vertices as ancestors, the graph is singly connected. Since for each vertex, the amount of time required is bounded by the no. of vertices times the in-degree of the particular vertex, the total runtime is bounded by $O(|V| \cdot |E|)$.

22.4: Topological Sort

Directed acyclic Graph

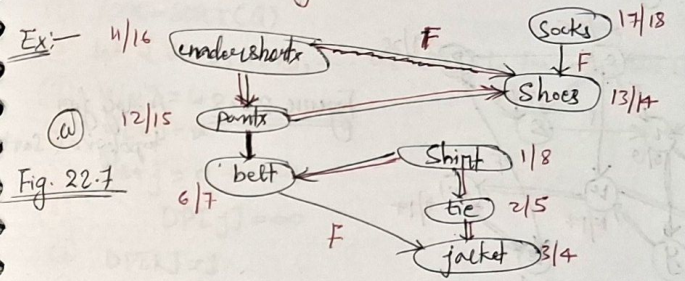
→ A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible).

TOPOLOGICAL-SORT(G)

① call DFS(G) to compute finishing times $v.f$ for each vertex v. } $\Theta(V+E)$
② as each vertex is finished, insert it onto the front of a linked list } $O(1)$
③ return the linked list of vertices.

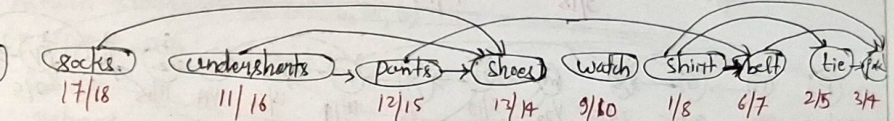
Total time complexity = $\Theta(V+E)$ Ans.

Ex:-



(a)

Fig. 22.7



(a) Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery & finishing times from a depth-first search are shown next to each vertex.

(b) The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.

Exercises

22.4-1: Showing the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of fig. 22.8, under the assumption of Exercise 22.3-2.

22.4-2: Give a linear-time algo. that takes as input a directed acyclic graph $G = (V, E)$ and 2 vertices s and t, & returns the no of simple paths from s to t in G. For example, the directed acyclic graph of fig. 22.8 contains exactly four simple paths from vertex p to vertex v: pov, poryv, poryv, posryv, & posryv. (your algo. needs only to count the simple paths, not list them).

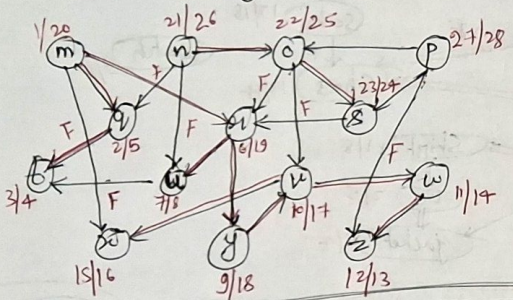
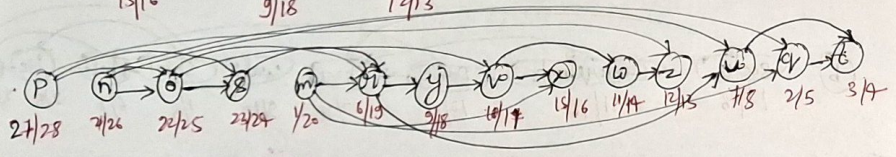


Figure 22.8 A dag for topological sort.



Ans. 22.4-1

label	m	q	t	r	u	y	v	w	z	x	n	o	p	s
discovered	1	2	3	6	7	8	10	11	12	15	21	22	27	23
finished	20	5	4	19	8	18	17	14	13	16	26	25	28	24

Ans. 22.4-2: The algorithm works as follows: The attribute u.paths of node u tells the number of simple paths from u to v, where we assume that v is fixed throughout the entire process.

First of all, a topological sort should be conducted & list the vertex b/w u, v as $\{v[1], v[2], \ldots v[k-1]\}$. To count the number of paths, we should construct a solution from v to u. Let's call u as v[0] & v as v[k], to avoid overlapping subproblems, the number of paths b/w v_k & u should be remembered & used as k decrease to 0. Only in this way can we solve the problem in $\Theta(V+E)$.

An bottom-up, iterative version is possible only if the graph uses adjacency matrix so whether v is adjacency to u can be determined in $O(1)$ time. But building a adjacency matrix would cost $\Theta(|V|^2)$, so never mind.

```
SIMPLE-PATHS(G, u, v)
1  TOPO-SORT(G)
2  let {v[1], v[2], ... v[k-1]} be the vertex b/w u & v
3  v[0] = u
4  v[k] = v
5  for j = 0 to k-1
6      DP[j] = ∞
7  DP[k] = 1
8  return SIMPLE-PATHS-AUX(G, DP, 0)

SIMPLE-PATHS-AUX(G, DP, i)
1  if i > k
2      return 0
3  else if DP[i] != ∞
4      return DP[i]
5  else
6      DP[i] = 0
7      for v[m] in G.adj[v[i]] & 0 < m ≤ k
8          DP[i] += SIMPLE-PATHS-AUX(G, DP, m)
9  return DP[i]
```

22.4-3: Give an algo. that determines whether or not a given undirected graph $G = (V, E)$ contains a simple cycle. Your algo. should run $O(V)$ time, independent of $|E|$.

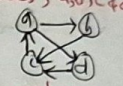
```
SIMPL-CYCLE (G):
① Input: G = (V, E) as an adjacency list
② Output: True if G contains a cycle, False otherwise.
③ visited = array of size |V|, initialized to False
④ DFS (v, parent) // traversal of Graph G.
⑤     visited[v] = True
⑥     for each neighbor in G[v]:
⑦         if visited[neighbor] is false:
                // Recursively visit the neighbor.
⑧             if DFS (neighbor, v) is True:
⑨                 return True
⑩         else if neighbor != parent:
                // if the neighbor is visited & is not the parent, there's a cycle
⑪             return True
⑫     return False
⑬ for each vertex v in V:    // check every vertex, as the graph might be disconnected
⑭     if visited[v] is False:
⑮         if DFS (v, -1) is True:    // start DFS with no parent (-1)
⑯             return True
⑰ return False
```

→ Time complexity: The algorithm performs a DFS traversal, where each vertex is visited once. Since the time complexity of visiting all vertices is $O(V)$, the algo. runs in $O(V)$ time.

→ If a back edge is found (i.e. visiting a vertex that has already been visited but is not the parent), this indicates a cycle.

22.4-4 Prove or disprove: If a directed graph G contains cycles, then TOPOLOGICAL-SORT(G) produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

Ans: This is not true. Consider the graph G consisting of vertices, a, b, c & d. Let the edges be (a,b), (b,c), (a,d), (d,c), (c,a).



Suppose that we start the DFS of TOPOLOGICAL-SORT at vertex C. Assuming that b appears before d in the adjacency list of a, the order, from latest to earliest, of finish times is c, a, d, b.

The "bad" edges in this case are (b,c) & (d,c). However, if we had instead ordered them by a, b, d, c then the only bad edges would be (c,a). Thus TOPOLOGICAL-SORT doesn't always minimizes the number of "bad" edges.

Definition:

- A topological sort of a directed graph $G = (V, E)$ is an ordering of its vertices such that for every directed edge $(u, v) \in E$, vertex u appears before v in the ordering.
- A bad edge is an edge (u, v) where u appears before v in the topological ordering; In a directed acyclic graph (DAG), no bad edge exist, as the topological sort provides a valid ordering.

22.4-5: Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it & remove it & all of its outgoing edges from the graph. Explain how to implement this idea so that it run in time $O(V+E)$. What happens to this algo. if G has cycles.

Ans: The approach to perform topological sorting by ~~repeatedly~~ repeatedly finding a vertex of in-degree 0, outputting it & removing it & it outgoing edges can indeed be implemented to run in $O(V+E)$ time. There is a following pseudo code:

```
TOPOLOGICAL-SORT(G):
I/p : G=(V,E), a DAG
o/p : Topologically sorted of vertices or detect cycle.
in-degree = array of size |V|, initialized to 0.
# step(1): calculate in-degree of each vertex
for each u in V:                                  } O(E)
    for each neighbor v of u:
        indegree[v] += 1
# step(2): Enqueue all vertices with in-degree 0
queue = new empty queue                           } O(V)
for each vertex u in V:
    if indegree[u] == 0:
        queue.enqueue(u)
topo_order = [] # This will store the topologically sorted order
# step(3): process the vertices
While queue is not empty:                         } O(V+E)
    u = queue.dequeue()
    topo_order.append(u)
    for each neighbor v of u:
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.enqueue(v)
# step(4): check if all vertices were processed
if len(topo_order) != |V|:
    return "Graph contains a cycle"
else
    return topo_order.
```

Overall time complexity O(V+E)

What happens if the Graph contains cycles?

→ If the graph contains a cycle, there will be no way to remove all vertices from the graph because vertices involved in the cycle will always have an in-degree greater than 0 (since they depend on each other).

→ As a result, the queue will become empty before all vertices have been processed.

→ After processing the vertices with in-degree 0, if the number of vertices in the topologically sorted order is less than V. (i.e. some vertices are still unprocessed), this indicates the presence of a cycle.

Therefore, in the presence of a cycle, the algo. will terminate without producing a valid topological order & will correctly detect that the graph contains a cycle.

22.5 : Strongly connected Components

The following linear-time ($\Theta(V+E)$ time) algo. computes the strongly connected components of directed graph $G=(V,E)$ using 2 DFS → one on G & one on G^T.

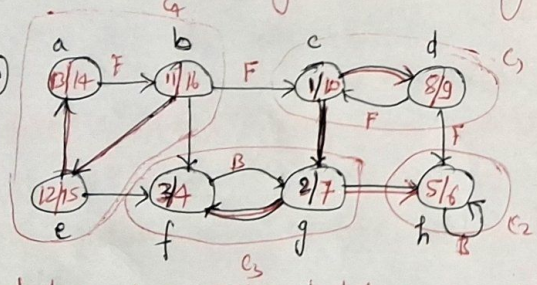
STRONGLY-CONNECTED-COMPONENTS(G)

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

The idea behind this algo. comes from key property of the connected graph $G^{SCC} = (V^{SCC}, E^{SCC})$, which we define as follows.

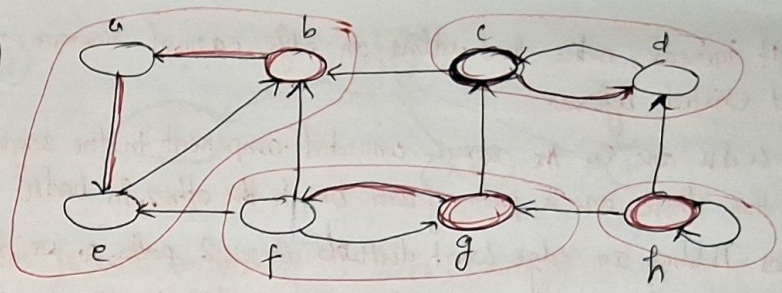
→ Suppose that G has SCCs $C_1, C_2, C_3, \dots, C_k$.
→ vertex set $V^{SCC} \rightarrow \{v_1, v_2 \dots v_k\}$, if it contains a vertex v_i for each SCC C_i of G.
→ $\therefore (v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x,y) for some $x \in C_i$ & some $y_i \in C_j$.
→ looked at another way, by contracting all edges whose incident vertices are within the same SCC of G, the resulting graph is G^{SCC}.

Fig. 22.9 (a)



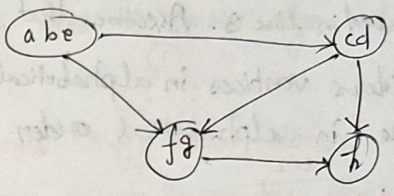
A directed graph G. Each shaded region is a SCC of G.

(b)



The graph G^T, the transpose of G, with the depth-first forest computed in line 3 of SCC, shown the tree edge as ⟶. Each SCC corresponds to one DFS-tree. Vertices b, c, g, h which are ⬭ the roots of the depth-first trees produced by the DFS of G^T.

(c)



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

Exercises

22.5-1 : How can the number of SCC of graph change if a new edge is added?

It can either stay the same or decrease. To see that it is possible to stay the same, just suppose you add some edge to a cycle. To see that it possible to decrease, suppose that your original graph is on three vertices & is just a path passing through all of them & the edge added completes their path to a cycle. To see that it

it cannot increase, notice that adding an edge cannot remove any path that existed before.

So, if u & v are in the same connected component in the original graph, then there are a path from one to the other, in both directions. Adding an edge won't disturb these 2 paths, so we know that u & v will still be in the same SCC in the graph after adding an edge. Since no components can be split apart, this means that the number of them cannot increase since they form a partition of the set of vertices.

22.5-2 : Show how the procedure S-C-C works on the graph of Figure 22.6, Specially, show the finishing times computed in the line 1 & the forest produced in line 3. Assume that the loop of lines 5-7 of DFS considers vertices in alphabetical order & that the adjacency lists are in alphabetical order.

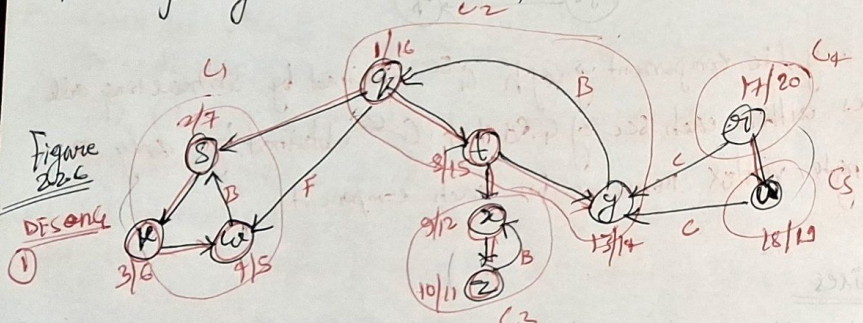
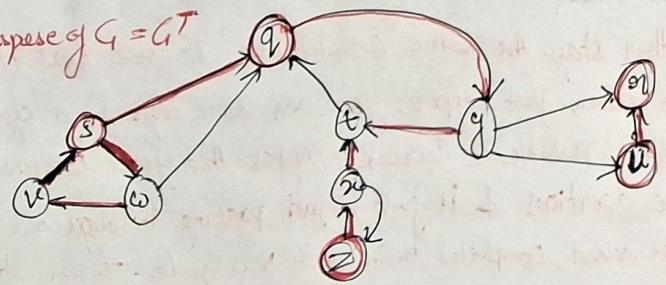
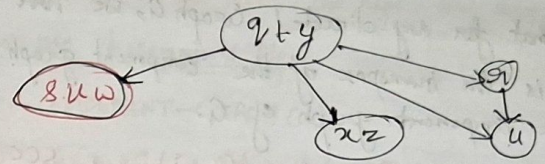


Figure 22.6

② Transpose of $G = G^T$

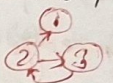


③



The forest consists of 5 trees, each of which is a chain. We'll list the vertices of each tree in order from root to leaf: $r, u, q-y-t, x-z$, & $s-w-v$

22.5-3 : Professor Bacon claims that the algo. for SCCs would be simpler if it used the original (instead of the transpose) graph in the second DFS & scanned the vertices in order of increasing finishing times. Does this simpler algo. always produce correct results?



Professor Bacon's suggestion doesn't work out. As an example, suppose that our graph is on the three vertices $\{1, 2, 3\}$ & consists of the edges $(2,1), (2,3), (3,2)$. Then, we should end up with $\{2,3\}$ & $\{1\}$ as our SCC's. However, a possible DFS starting of 2 could explored 3 before 1, this would mean that the finish time of 3 is lower than of 1 & 2. This means that when we first perform the DFS starting at 3. However, a DFS starting at 3 will be able to reach all other vertices. This means that the algorithm would return that entire graph is a single SCC, even though this is clearly not the case since there is neither a path from 1 to 2 or from 1 to 3.

22.5-4: Prove that for any directed graph G, we have $(G^T)^{SCC} = G^{SCC}$. That is the transpose of the component graph of G^T is the same as the component graph of G.

First observe that C is a SCC of G iff it is a SCC of G^T. Thus the vertex sets of G^{SCC} & $(G^T)^{SCC}$ are the same, which implies the vertex sets of $((G^T)^{SCC})^T$ & G^{SCC} are the same. It suffices to show that their edge sets are the same. Suppose (v_i, v_j) is an edge in $((G^T)^{SCC})^T$. Then (v_j, v_i) is an edge in $(G^T)^{SCC}$. Thus there exist $x \in C_j$ & $y \in C_i$ such that (x, y) is an edge of G^T, which implies (y, x) is an edge of G. Since components are preserved, this means that (v_i, v_j) is an edge in G^{SCC}. For the opposite implication, we simply note that for any G we have $(G^T)^T = G$.

22.5-5: Given a $O(V+E)$-time algo. to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge b/w two vertices in the component graph your algo. produces.

① Find SCC: use Kosaraju's algorithm or Tarjan's algo. to find the all SCCs. Both run in $O(V+E)$-time.

② Build the component Graph: → Treat each SCC as a single vertex.
→ Create an edge b/w 2 SCCs if there is at least one edge b/w them in the original graph.
→ Ensure there is at most one edge b/w any 2 SCCs.

Algorithm use Kosaraju's Algo.

```
COMPUTE-COMPONENT-GRAPH(G):
    I/P: G=(V,E), a directed graph.
    O/P: Component graph of G (DAG of SCCs).
    # Step(1): Find Strongly connected components using Kosaraju's Algo
    # Find DFS to compute finishing times
    DFS1(G, v, visited, stack):
        visited[v] = True
        for each neighbour u of v in G:
            if not visited[u]:
                DFS1(G, u, visited, stack)
        stack.push(v) # Push v to the stack once all its neighbours are visited.
    # Second DFS on the transposed graph to find SCCs
    DFS2(G^T, v, visited, current_scc):
        visited[v] = True
        current_scc.append(v) # add v to the current SCC
        for each neighbour u of v in G^T:
            if not visited[u]:
                DFS2(G^T, u, visited, current_scc)
    # Main function to find SCCs
    FIND-SCCs(G):
        stack = new empty stack
        visited = array of size |V|, initialized to False
        # First DFS
        for each vertex v in V:
            if not visited[v]:
                DFS1(G, v, visited, stack)
        # Transpose of G
        G^T = transpose(G)
```

```
# DFS2
visited = array of size |V|, initialized to False
SCCs = [ ]   # list of all SCCs
While stack is not empty:
    v = stack.pop()
    if not visited[v]:
        current_scc = [ ]
        DFS2 (GT, v, visited, current_scc)
        SCCs.append (current_scc)
return SCCs
```

#step2: Build the component Graph

```
BUILD-COMPONENT-GRAPH (G, SCCs):
    # map each vertex to its SCC
    scc_map = array of size |V|
    for each i in range (len(SCCs)):
        for each vertex v in SCCs[i]:
            scc_map[v] = i   # Vertex v belongs to SCC i
    component_graph = new empty graph (C, E)
    C = {0, 1, ---> len(SCC) - 1}   # The vertices are SCCs
    # create a set of edges to avoid duplicates
    edge_set = new empty set
    # for each edge (u,v) in the original Graph G
    for each edge (u, v) in E:
        if scc_map[u] != scc_map[v]:
            if (scc_map[u], scc_map[v]) not in edge_set:
                component_graph.add edge (scc_map[u], scc_map[v])
                edge_set.add ((scc_map[u], scc_map[v]))
    return componet_graph
```

```
# main procedure
SCCs = FIND-SCCs (G)  ---> O(V+E)
Component_graph = BUILD-COMPONENT-GRAGH (G, SCCs) ---> O(V+E)
return component_graph
```

22.5-6: Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (a) G' has a SCC as G, (b) G' has the same component graph as G & (c) E' is as small as possible. Describe a fast algo to compute G'.

(1) Find the SCCs of G → [Kosaraju's algo. / Tarjan algo.] → $O(V+E)$.

(2) Construct a new graph G' by preserving the intra-SCC edges & adding only the necessary edge btw SCCs → [DAG] → $O(V+E)$
- Intra-SCC edges, keep enough edges to ensure that each SCC is still strongly connected.
- Inter-SCC edges, add edges only btw SCCs that are directly connected in G.

(3) The reduced Graph G' should: → $O(E)$
- Keep the minimal set of edges within each SCC to preserve strong connectivity (e.g. use DFS tree within each SCC).
- Preserve the direct edge btw diff. SCCs in the Component Graph.

```
Minimal_scc_graph (G):
    I/p: G = (V, E), a directed graph
    O/p: G' = (V, E'), a minimal graph with the same SCCs & Component graph as G.
    # Step(1): Find SCCs using Kosaraju's Algo.
    SCCs = find_SCCs(G)
    # Step(2): Build component Graph (Direct SCC-to-SCC edges)
    Component_graph = New empty graph
    scc_map = map each vertex to its SCC in SCCs
    for each edge (u,v) in G:
        if scc_map[u] != scc_map[v]
            component_graph.add_edge (scc_map[u], scc_map[v])
    # Step(3): Minimizes Intra-SCC Edges
    ~~Minimal_scc_graph~~:
    Minimal_intra_scc_edges = [ ]
```

```
for each SCC in SCCs:
    tree_edges = DFS-tree(scc)
    minimal-intra-scc-edges.append(tree_edges)
# Combine the minimal intra-scc edges & inter-scc edges into G'
G-prime = new graph
G-prime.add_edges(minimal-intra-scc-edges)
G-prime.add_edges(component-graph.edges)
return G-prime.
```

Time Complexity:

① Finding SCCs: Using Kosaraju's algo or Tarjan algo. takes $O(V+E)$

② Building the Component Graph: Since we iterate over all the edges once & add at most one edge b/w SCCs, this takes $O(E)$.

③ Minimizing intra-SCC edges: For each SCC, we perform a DFS to find a spanning tree, which takes $O(V+E)$. If SCCs combined

Thus overall time complexity is $O(V+E)$

Hence, the resulting graph G' will have the following properties:

→ It has the same SCCs as G.

→ It has some component graph (DAG of SCCs) as G

→ It contains the minimal number of edges necessary to maintain these properties.

22.5-7: A directed graph $G=(V,E)$ is semiconnected if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algo. to determine whether or not G is semiconnected. Prove that you algo. is correct & analyze its running time.

Algorithm:

① Run STRONG-CONNECTED-COMPONENTS(G). → $O(V+E)$

② Take each SCC as a virtual vertex & create a new virtual graph G'.

③ Run Topological-Sort(G') $O(V+E)$

④ check if for all consecutive vertices (v_i, v_{i+1}) in a topological sort of G', there is an edge (v_i, v_{i+1}) in graph G'. If So, the original graph is semiconnected. otherwise, it isn't.

Proof:

It is easy to show that G' as a DAG. Consider consecutive vertices v_i & v_{i+1} in G'. If there is no edge from v_i to v_{i+1}, we also conclude that there is no path from v_{i+1} to v_i since v_i finished after v_{i+1}. From the definition of G', we conclude that, there is no path from any vertices in G who is represented as v_i in G' to those represented as v_{i+1}. Thus, there is not path from any vertices in G who is represented as v_i in G' to those represented as v_{i+1}. Thus, G is not semi-connected. If there is an edge b/w all consecutive vertices, we claim that there is an edge b/w any two vertices. Therefore, G is semi-connected.

Running time: $O(V+E)$

Chapter-22 Problems

22-1 Classifying edges by BFS

A DFS forest classifies the edges of graph into tree, back, forward & cross edges. A BFS tree can also be used to classify the edges reachable from the source of the search into the same four categories.

(a) Prove that in a BFS of an undirected graph, the following properties hold:

→ (1) There is no back edges & no forward edges. [level-by-level]

- If we found a back edge, this means that there are two vertices, one a descendant of other, but there is already a path from the ancestor to the child that doesn't involve moving up the tree. This a contradiction since the only children in the BFS tree are those that are a single edge away, which means there cannot be any other paths to that child because that would make it more than a single edge away.
- To see that there are no forward edges, we do a similar procedure. A forward edge would mean that from a given vertex we notice it has a child that has already been processed, but this cannot happen because all children are only one edge away, & for it to of already been processed, it would need to have gone through some other vertex first.

* No back edges can exist in BFS tree because BFS always explores the shortest path first & a back edge would imply a shorter path that was not found.

* No forward edges can exist in BFS tree because BFS processes vertices level-by-level, ensuring that there are no "skipped" vertices that would make forward edges possible.

* Cross edges can exist, but they don't violate the BFS structures

→ (2) For each tree edge (u,v), we have $v.d = u.d+1$

An edge is placed on the list to be processed if it goes to a vertex that has not yet been considered. This means that the path from that vertex to the root must be at least the distance from current vertex plus 1. It also at most that since we can just take the path that consists of going to the current vertex & taking its path to the root.

→ (3) For each cross edge (u,v), we have $v.d \leq u.d$ or $v.d = u.d+1$

- We know that cross edge cannot be going to a depth more than or less, otherwise it would be used as a tree edge when we were processing that earlier element. • It also cannot be going to a vertex that was much further away from the root. Since, the depths of vertices in the cross edge cannot be more than one apart, the conclusion follows by possibly interchanging the roles of u & v, which we can do because the edges are unordered.

(b) Prove that in BFS search of a directed graph, the following properties hold:

→ (1) There are no forward edges.

To have a forward edge, we would need to have already processed a vertex using more than one edge, even though there is a path to it using a single edge. Since BFS always considers shortest path first, this is not possible.

→ (2) For each tree edge (u,v), we have $v.d = u.d+1$

Suppose that (u,v) is a tree edge. Then, this means that there is a path from the root to u of length $u.d+1$ by just appending (u,v) on to

the path from the root to u. To see that there is no shorter path, we just note that we would of processed u sooner, & so wouldn't currently have a tree edge if there were.

③ For each cross edge (u,v), we have $v.d \leq u.d+1$

To see this, all we need to do is note that there is some path from the root to v of length $u.d+1$ obtained by appending (u,v) to $v.d$. Since there is a path of that length, it serves as an upper bound on the minimum length of all such paths from the root to v.

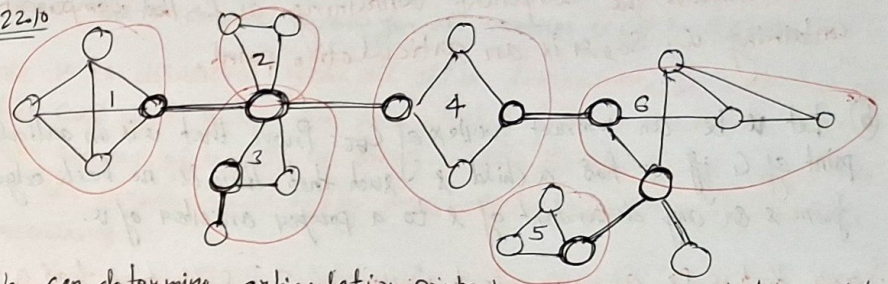
④ For each back edge (u,v), we have $0 \leq v.d \leq u.d$

It is trivial that $0 \leq v.d$, since it is impossible to have path from the root to v of negative length. The more interesting inequality is $v.d \leq u.d$. We know that there is some path from v to u. Considering of tree edges, this is the defining property of (u,v) being a back edge. This means that is $v_0, v_1, \ldots v_k$, u is path (It is unique because the tree edges form a tree). Then, we have that $u.d = v_k.d+1 = v_{k-1}+2 = \cdots = v_1.d+k =$ $v.d+k+1$. So, we have that $u.d > v.d$. In fact, we just showed that we have the stronger conclusion, that $0 \leq v.d \leq u.d$.

22-2 Articulation points, bridges, & biconnected components

Let $G=(V,E)$ be a connected, undirected graph. An articulation point of G is a vertex whose removal disconnects G. A bridge of G is an edge whose removal disconnects G. A biconnected component of G is an edge whose a maximal set of edges such that any 2 edges in the set lie on a common simple cycle.

Fig. 22.10



We can determine articulation points, bridges & biconnected components using DFS. Let $G_{\pi} = (V, E_{\pi})$ be a DF-tree of G.

(a) Prove that the root of G_{π} is an articulation point of G iff it has at least two children in G_{π}.

First suppose that root r of G_{π} is an articulation point. Then the removal of r from G would cause the graph disconnect, so r has at least 2 children in G. If r has only on child u in G_{π} then it must be the case that there is a path from u to each of r's other children. Since removing r disconnection of the graph, there must exist vertices u & w such that the only paths from u to w contain r.

To reach r from u, the path must first reach on of r's children. This child is connected to u via a path which doesn't contain r.

To reach w, the path must also leave r through one of its children, which is also reachable by v. This implies that there is a path from u to w which doesn't contain r a contradiction. Now, suppose r has at least 2 children u & v in G_π. Then there is no path from u to v in G which doesn't go through r, since otherwise u would be an ancestor of v. Thus, removing r disconnects the component containing u & the component containing v. So, r is an articulation point.

(b) Let v be an nonroot vertex of G_π. Prove that v is an articulation point of G iff v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

Suppose that v is an non-root vertex of G_π & that v has a child s such that neither s nor any of s's descendant have back edges to a proper ancestor of v. Let w be an ancestor of v & remove v from G. Since we are in the undirected case, the only edges in the graph are tree edges no back edges, which means that every edge incident with s takes us to a descendant of s & no descendants have back edges, so at no points can we move up the tree by taking edges.

Therefore, r is unreachable from s, so the graph is disconnected & v is an articulation point.

Now, suppose that for every child of v there exists a descendant of that child which has a back edge to a proper ancestor of v. Remove v from G. Every subtree of v is a connected component. Within a given subtree, find the vertex which has a back edge to a proper ancestor of v. Since the set T of vertices which aren't descendants of v form a connected, after the deletion we have that every subtree of v is connected to T. Thus, the graph remains connected after deletion of v so v is not a articulation point.

(c) Let $v.low = \min\{v.d, w.d : (u,w) \text{ is a back edge for some descendant } u \text{ of } v\}$.

Show how to compute $v.low$ for all vertices $v \in V$ in $O(E)$ times

Since v is discovered before all of its descendants, the only back edges which could affect $v.low$ are ones which go from a descendant of v to a proper ancestor of v. If we know $u.low$ for every child u of v, then we can compute $v.low$ easily since all the information is coded in its descendants.

Thus, we can write the algorithm recursively: If v is a leaf in G_π then $v.low$ is the minimum of $v.d$ & $w.d$ where (u,w) is a back edge. If v is not a leaf, v is minimum of $v.d, w.d$ where (u,w) is a back edge, & $u.low$, where u is child of v. Computing $v.low$ for vertex is linear in its degree. Thus the sum of the vertices's degree gives twice of the number of edges, so the total runtime is $O(E)$.

(d) Show how to compute all articulation points in $O(E)$ time.

First apply the algorithm of part (c) in $O(E)$ to compute $v.low$ for all $v \in V$. If $v.low = v.d$ iff no descendant of v has a back edge to a proper ancestor of u, iff v is not an articulation point.

Thus, we need only check $v.low$ versus $v.d$ to decide in constant time whether or not u is an articulation point, so the runtime is $O(E)$.