- Degree seq, Characterization, Recursive Algorithm
- Graph, Count the # of spanning trees, L, M_{11},
  $= \det(M_{11})$, Gaussian Elimination Algorithm for determinant - $3n^3$
  Direct Algorithm - $n!$
- The size problems that can be solved in unit time.

Day 3 - Graph Algorithms

Input ⟷ Representation

1) Correctness
2) Efficiency.

- Accessing the i^{th} element in List : Get (L, i)

Arguments ≅ Domain

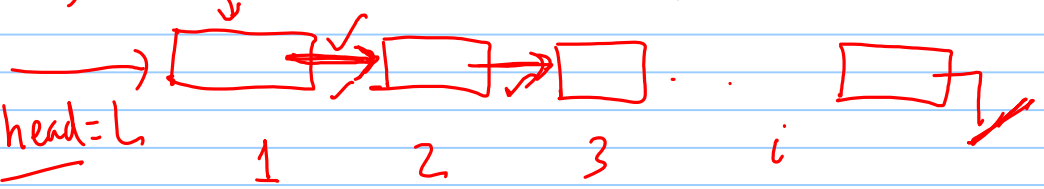
Return Value ≅ Range.

Function

LIST X INTEGER

Return - Value → Range of the function.

Get(L, i) takes i units of time to return the value ✓ O(i) time



One unit of time for a List is one pointer traversal
↳ find address ✓
↳ Check value in Address ✓

Get(L, i)

* L is an Array. ✓
  is a vector each data item has an index L[i] ✓
  Address(L(i)) = Address(L(0)) + i units ✓
  takes 1 unit of time O(1) time

Sort(A) in time $|A| \log_2 |A|$. → Comparison

Dequeue (Queue Q) – element at the head of the queue.

Enqueue (Q) – O(1)

O(1) time

FIFO data structure

pop (stack S) – top of the stack.

push (stack S) – adds to the top.

O(1) time

If you want to check if an element is present, do not use stack or queue

Graphs require Non-linear storage (Visualization or in reality)

Linear means - For each element will defined Pred, Succ.

Vertices

Edges

$G = (V, E)$

Storage - $O(n^2)$
$O(nm)$

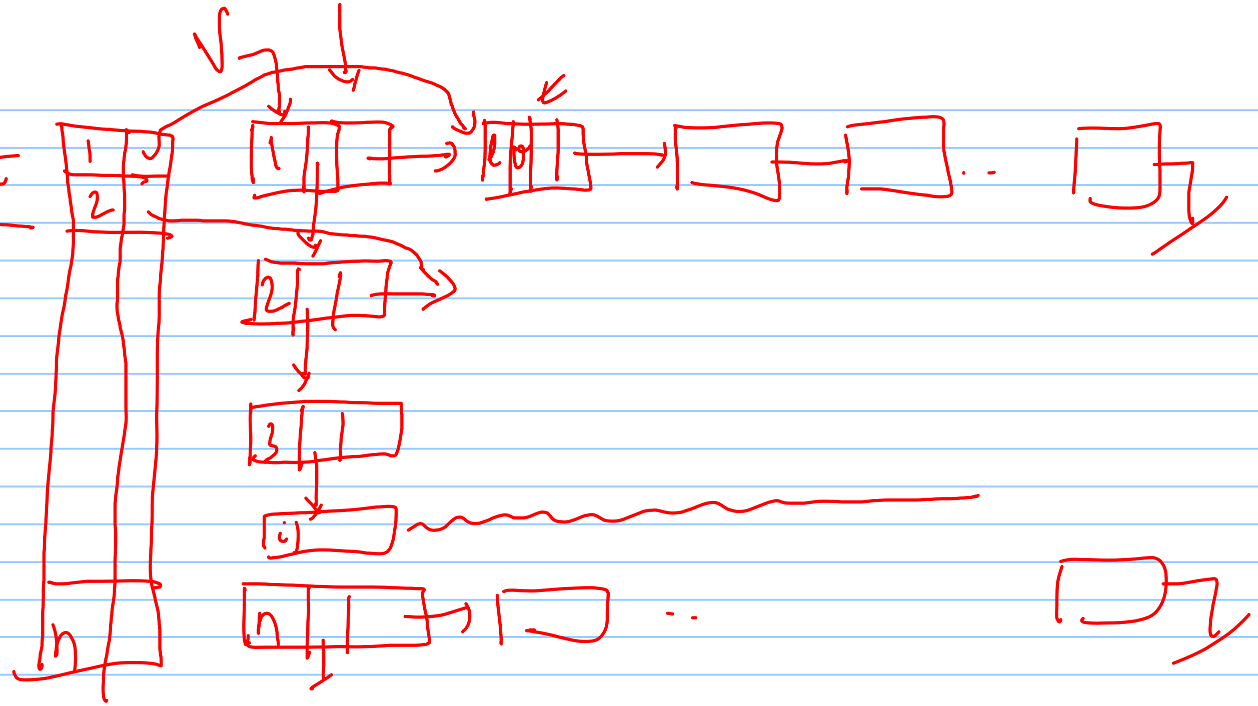
Adjacency Matrix $|V| \times |V|$
- 2 dimensional Array

$A[i, j]$ - Ex
Write a formula for the address of $A[i, j]$.
can be computed in $O(1)$ time.

Incidence matrix - $|V| \times |E|$

1) Adj. queries in $O(1)$ time.
or Incidence $IsAdj(i, j)$ $IsInc(i, e)$ 2) $IsConn(i, j)$??

Adjacency List



$Isadj(i, j)$ — $O(d_i)$ units of time

d_i → degree of i

$Isinc(i, e)$

Graph Search ⇝ Transitive Closure of the edge relation.

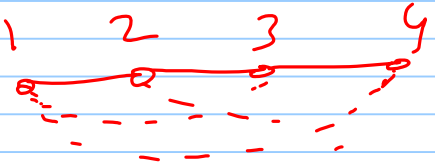
Recall $G = (V, E)$ $E \subseteq V \times V$ ← Directed graph, Self Loops no parallel edges

or $\subseteq$ set of 2 element sub sets of V.

Undirected, no self loops, no parallel edges

Symmetry

E, $E^* = \{(i, j) \mid$ there is a k for which (i, k) is in E and (k, j) is in $E^*\} \cup \{(i, i) \mid i \text{ is in } V\}$



Reflexive Transitive Closure

Input $G = (V, E)$, $i, j \in V$ ✓

Question Does there exist a path from i to j in G? ✓

Is i connected to j in V?

Connectivity Problem

Compute from Adj matrix A, A^n using boolean $\wedge$ for $*$, boolean $\vee$ for $+$

Check $(A^n)_{ij}$.

Thm There is a i-j path $\iff (A^n)_{ij} = 1$

Running time

n^2 inner products ✓

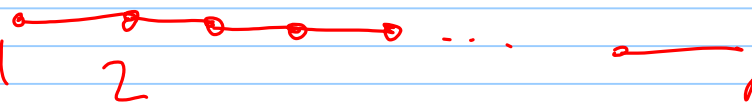
n^3 multiplication ✓

n matrix products $A^2, A^3 \ldots, A^n$

Total n^4 multiplications $n^3 \cdot (n)$

$O(n^4)$ time

$A, A^2, A^4, A^8, A^{16}, \ldots A^{2^r} \leq$ $O(n^3 \log n)$

- Can we do better?
- Computing connectivity between all pairs?
- 

  Starting with Adjacency matrix - A^n is unavoidable

- Research Issues: How fast can matrix multiplication be done? n^3

  Strassen's Algorithm: $n^{2.81}$ multiplications

  $n^{2.37}$ $O(n^3)$

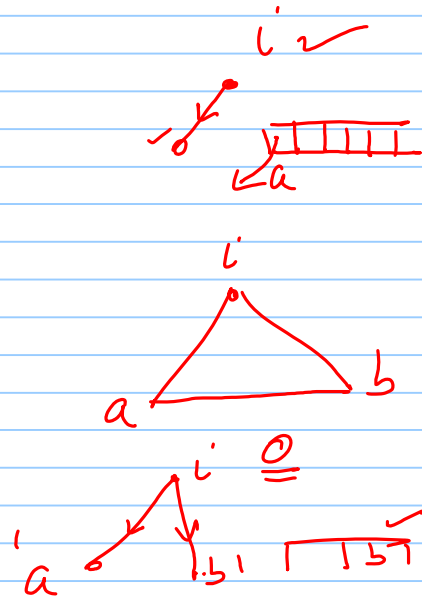
Adjacency List Representation

Breadth-First Search (G, i, J)

1) Level(u) → Initialize all Level values to −1 $O(n)$
2) Parent(u) → Initialize all parents to undef $O(n)$

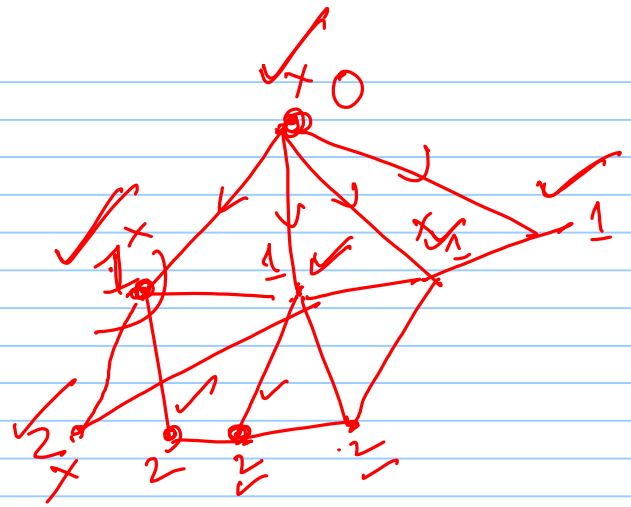
a) Level(i) := 0 ; Parent(i) = undefined
b) Insert N(i) into a queue. Q.

Iterate: Dequeue: Give Parent into ↓ undef, update Level info
Enqueue Nbrs which are not in the queue.



Level(vertex) = Level(parent) + 1

Parent of a vertex = (First time) it is the neighbour of a "visited" vertex.



- Level is given to a vertex when it is visited for the first time. (When they are inserted into Q)
- The vertex from which it is visited is called the parent.
- A vertex dequeued never gets considered again. [crucial for termination/ Halting].
- Each edge is inspected exactly 2 times

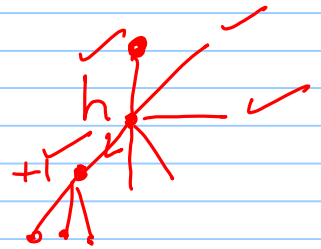
Linear in Input rep.

$n+m$

(2 · # of edges + # of vertices)

Breadth First Search

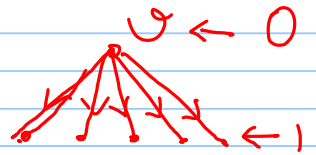
```
main()
{ Visited[1..n] ← 0          Queue Q ← Empty.
  Parent[1..n] ← 0
  Level[1..n] ← -1
  BFS(v)                     Insert *
}
```



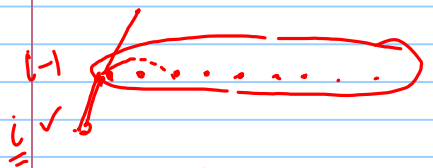

```
BFS(v)
{ Visited(v) = 1, Level(v) = 0, EnQueue(Q, v).
  While (Q is not Empty)
  { h = dequeue(Q)
    For [each u in Adj[h]]
    { if (Visited[u] = 0) { visited(u) = 1, level(u) = level(h) + 1, Parent[u] = h, Enqueue(Q, u) }
    }
  }
```

→ /* At the end of this Loop, h & all its Nbrs are visited, have a level and a parent, & h has left the Q */

1) If BFS(v) is invoked and if u has a path from v, then on termination
   a) Level(u) = length of shortest Path (Min # of edges) from v
   b) Visited(u) = 1.
   c) Parent(u) is the penultimate vertex on the shortest path from v.
      a
      maintain by the algorithm



• Each of these was inserted into the Q, after Level, Visited, & Parent were assigned



• On termination Q is empty.
• These values do not change.

2) Termination : The Queue Q has vertices inserted in non-decreasing order of Level number.

Each vertex is inserted into the Q exactly 1
At the beginning 1 vertex is removed. .. After ≤ max level # of iterations Q = empty.

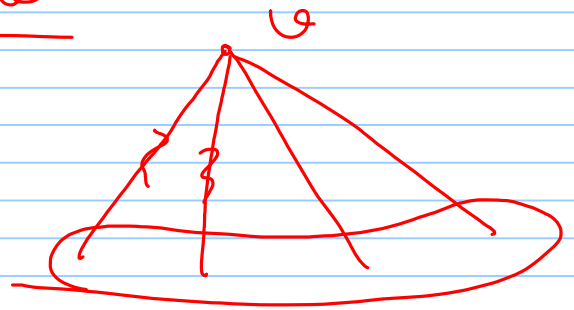
Time: • Each vertex, visited, Parent, level is updated ≤ 2 times (Initialization, one in the while loop)

These are in arrays → i^{th} elt is accessed in $O(1)$ time

Total time is $O($# of vertices reachable from $v)$

• Each edge in the adj. list is accessed either 0 times or 2 times.

2 · # of edges incident on vertices reachable from v



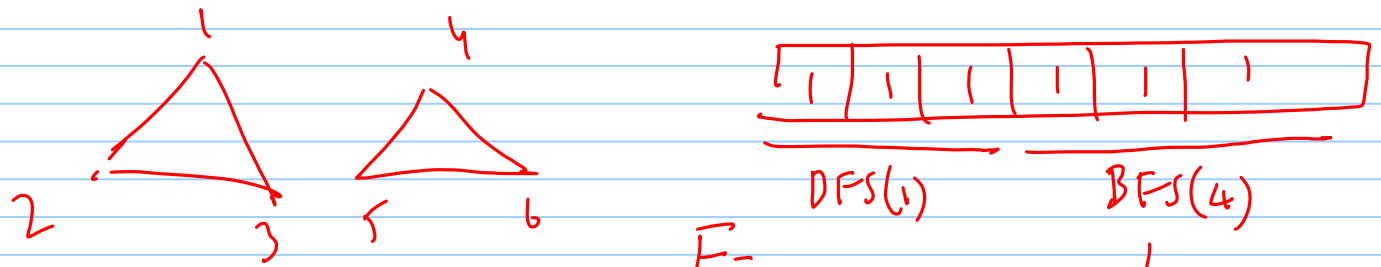
- Main code

```
For (each vertex v)
  { If (visited[v] = 0) BFS(v)
  }
```

This also terminates

$(\#$ of vertices $+ n +$ 2 · total # of edges$)$

- If each vertex is reachable from v, there is exactly 1 BFS call
- # of BFS = # of connected components for undirected graphs.

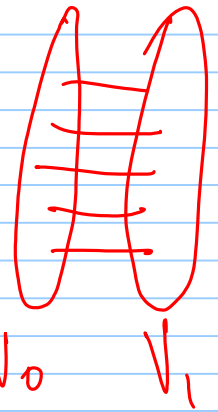


Properties: Consider the edges $F = \{(parent(u), u) \mid u \text{ in } V\}$. has $n-k$ edges

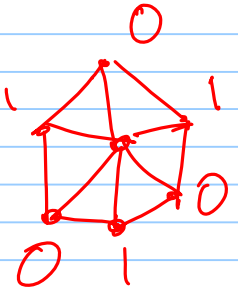
k is the # of connected components.

- F is an acyclic set of edges. $Level(u) = Level(parent(u)) + 1$
- If $k=1$, F is called a BFS spanning tree / Forest ($k > 1$) has all "distance" info from the "root" v.

Colouring the Vertices with 2 colours 0 & 1



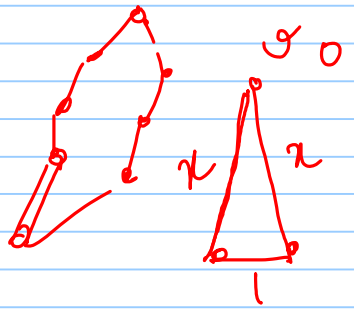
- Possible If & only If NO Odd cycle.



Introduction to Algorithms CLRS

- Design an efficient Algorithm

BFS - colour(v) = 0 if level(v) is even
= 1 if level(v) is odd.



Report yes: If each edge is properly coloured ✓

No: otherwise.

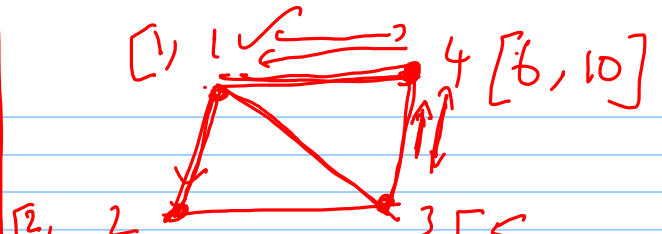
$O(\#Edges + \#Vertices)$

Depth First Search

- Recursive specification, implementation.
- Visited [1 .. n] ← 0
  Parent [1 .. n] ← −1
  First [1 .. n] } Timestamps ← −1
  Last [1 .. n] } ← −1
  Depth [1 .. n] ← −1
  Time = 0

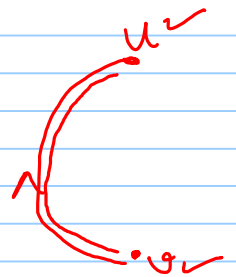
For (each v) ✓
→ {If (visited [v] = 0) dfs (v)}

```
DFS(v)
{ time = time + 1;
  If (visited(v) = 0)
    { visited(v) = 1
      First(v) = time.
      For (u in Adj(v)) DFS(u)
    } Last(v) = time; return;
else /* v is already visited */
    { time = time + 1;
      return
    }
}
```



DFS(1)
time 1
DFS(2)
time 2
DFS(1)
time 3
time 4
return
DFS(3)
time 5
DFS(4)
time 6

time 7
time 8
time 9
time 10
returns



```
For (u in Adj(v))
  { If (Visited[u]==0)
      { Parent[u] = v
        depth[u] = depth[v]+1
      }
    DFS(u);
  }
```

1) At the end of DFS, each vertex has atmost one parent. If a vertex u does not have a parent, then depth[u] = 0.

2) The intervals [First[v], Last[v]] form a LAMINAR FAMILY.

   $u \neq v$, then either [f[u], L[u]] & [F[v], L[v]] are (i) Disjoint

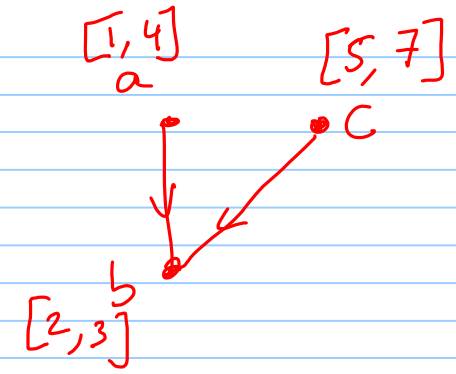
   or

   (ii) Strictly contained in the other.

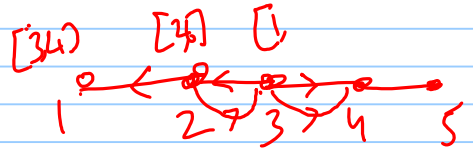
3) If $[f(u), L(u)] \subset [f(v), L(v)]$ then v is an ancestor of u $\approx$ $v\, v_1 \ldots v_r\, u$ is a path such that $P[u] = v_r$ $P[v_r] = v_{r-1}$ $\ldots$ $P[v_1] = v$

   $d[u] = d[v_{r+1}] + 1$ $\ldots$ $d[v_1] = d[v] + 1$

4) If $[f(u), l(u)]$, $[f(v), l(v)]$ are disjoint then they do not have a common ancestor ✗



→ X →



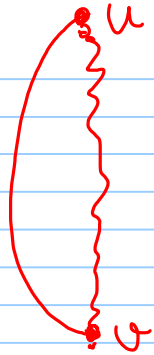
i) • they have a common ancestor and there is no path from u to v other the possible path u to ancestor(u) to v

OR

b) They don't have a common ancestor

5) Consider $Z = \{v \mid d[v] = 0\}$ ✓

Consider $C(v) = \{u \mid u \text{ is a descendant of } v\}$.

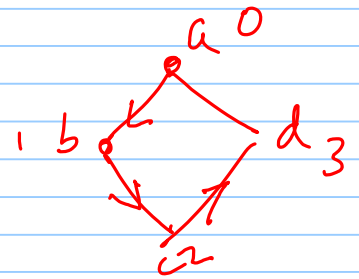


For each v in Z

a) $C(u) \cap C(v) = \phi$ for $u \neq v$ in Z.

b) $[f[u], l[u]] \cap [f[v], l[v]] = \phi$ for $u \neq v$ in Z.

Classification of edges



a) (u, v) when $d[v] = d[u] + 1$ are called TREE ARCS
v $P[v] = u$

b) If u, v are in the same $C(w)$ for w in Z,
: v is a descendant of u and (v, u) is an edge (v, u) is a back Arc

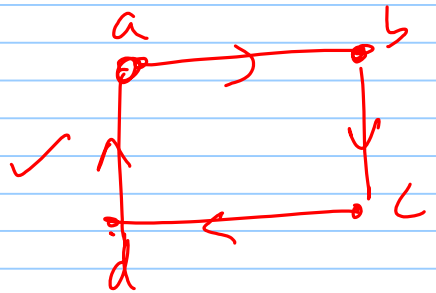
(u, v) Back arc $\Rightarrow$ v is a descendant of u AND (u, v) is an edge

$v \to v_1 \to v_2 \ldots v_r \to u \to v$

A CYCLE

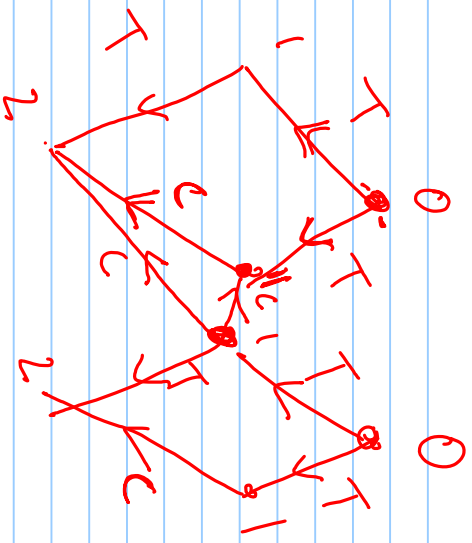
If C is a cycle $\Rightarrow$ one of the edges is a back arc

If u is a descendant of v & (v, u) is an edge, then (v, u) is called a FORWARD ARC



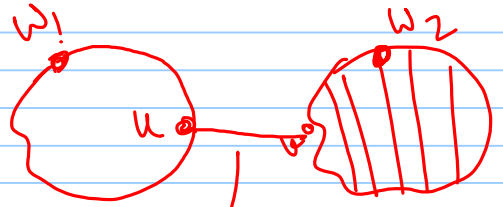
Consider x in $C(u)$
y in $C(v)$
an arc (x, y) is
Called a CROSS ARC

actually An edge which is not a TA, BA, FA is called a CA



Undirected Graphs → Only Tree arcs, Back Arcs

- Does G have a bridge (removal disconnects u & v)
- (u, v) is a bridge if the graph $G' = (V, E \setminus \{u,v\})$ is such that u and v are in $C(w_1)$, $C(w_2)$. $w_1 \neq w_2$

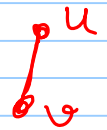


G'

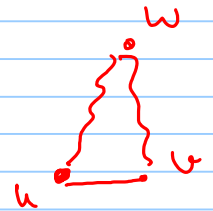
BRIDGE

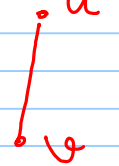
(u, v) is a Back Arc

- Either the edge is visited from parent to child



- When the edge is considered the end point v is not a child of u ⇒ v is an ancestor u & v are in C(w) for some w ∴ d(w) ≥ 0





Repeat for each edge $e = (u, v)$

DFS (G after removing e, from u)

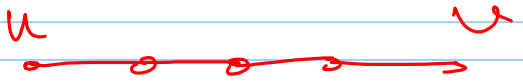
Check $[f(u), l(u)]$, $[f(v), l(u)]$

If Disjoint e is a bridge.

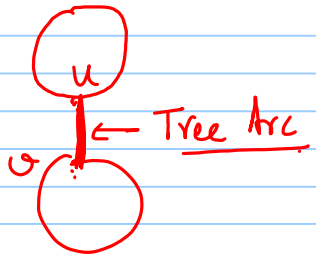
Inefficient We do too many DFS, 1 per edge.

So running time is

$m \times$ Time(DFS)

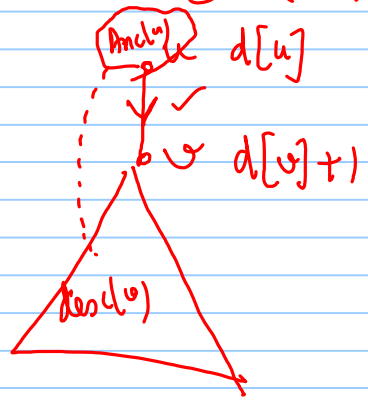


- Identify All Bridges using 1 DFS on G.
- If an edge is a bridge then it is a tree Arc [In any DFS when bridge is considered for the first time, atu the "other vertex" is visited for the first time ≈ TREE ARC]



1) DFS

2) Consider each e a tree Arc (u,v) — e is Bridge $\Leftrightarrow$ for

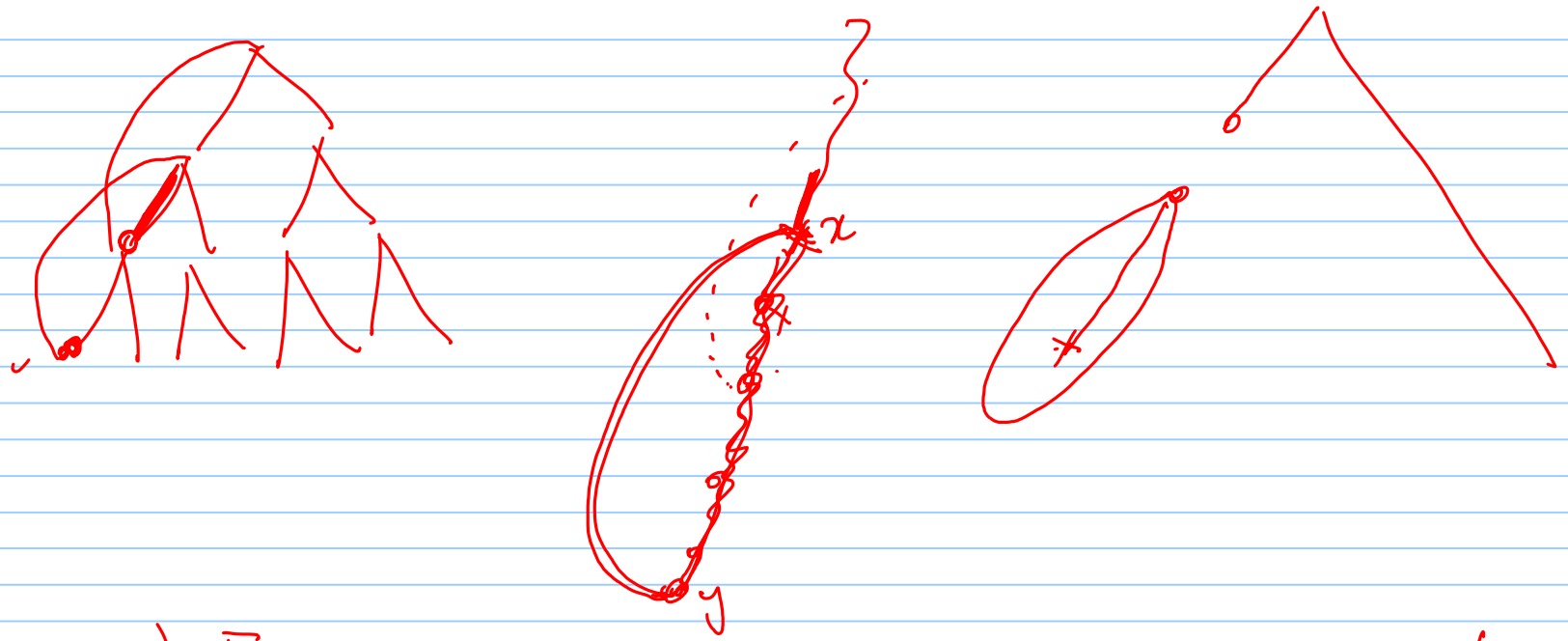


Check for an edge (x,y) where x is a descendant of v & y an ancestor of u

& (x,y) a back Arc.

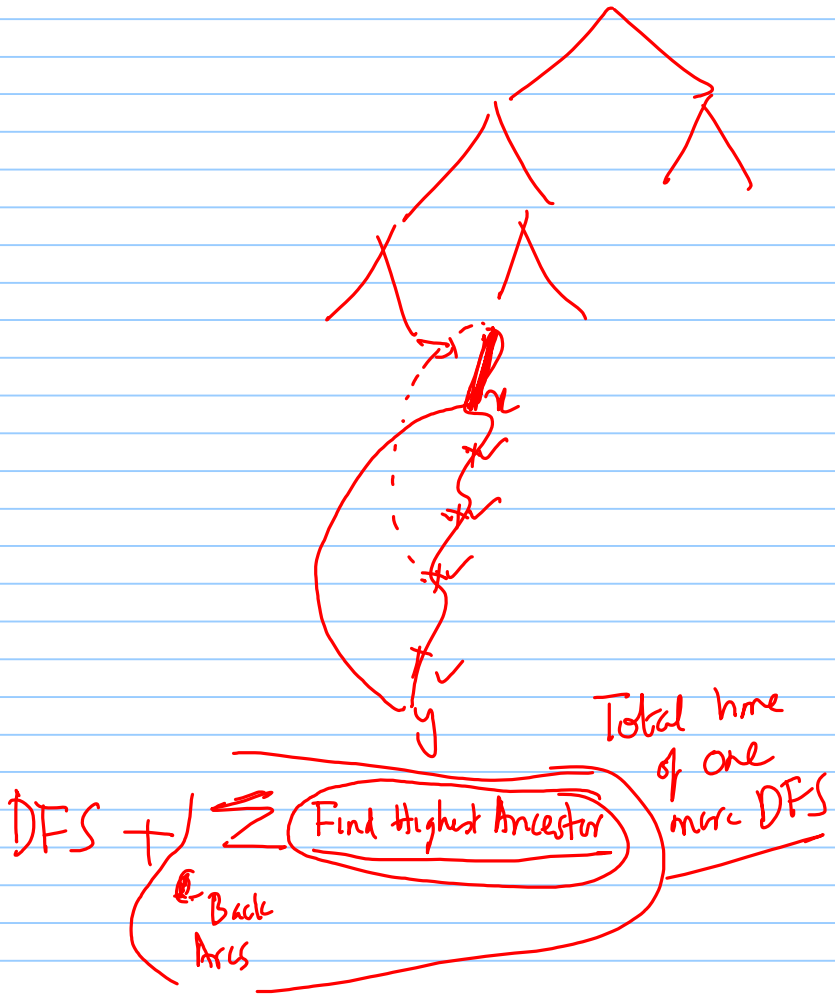
If none — e is a bridge.

3) For each vertex v — Track the back arc (v,y): $d(y) - d(v)$ is the maximum & associate with $v \rightarrow d(y)$.



Track -- 1) For each v, the back arc to the highest ancestor / ancestor of least depth.

2) For each Back Arc - Check if one vertex has an ancestor



(y, x) is a back Arc

Consider the cycle x ~ Tree arc ~ y - - - Back Arc

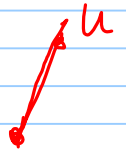
Compute $\min_{w}$ Highestancestor(w)

If this is lesser than $d[x]$ then

$(parent(x), x)$ is not a bridge for any vertex in this cycle.

Because Parent(x) does not get disconnected from any vertex in the cycle by removal of $(P(x), x)$.

DFS + $\sum$ Find highest Ancestor
Back Arcs



2 DFS

Time to update visited + 4. |E|

$n + 4m$

CLRS.

1) Parent info
2) Depth info
3) Interval Infor

Computed by DFS

- Edge classification, TA, FA, BA, CA
  under graphs TA, BA
- Addition information
  - Highest back arc for each vertex ✓
  - For each back arc (x,y) the highest ancestor for each vertex in the cycle formed by back arc.
    - If this is of lesser depth than depth of y
  - No bridge.