COMP20003 Workshop Week 12

Welcome to the last workshop!

MST & Greedy Algorithm for building MST
Prim's Algorithm
Kruskal's Algorithm

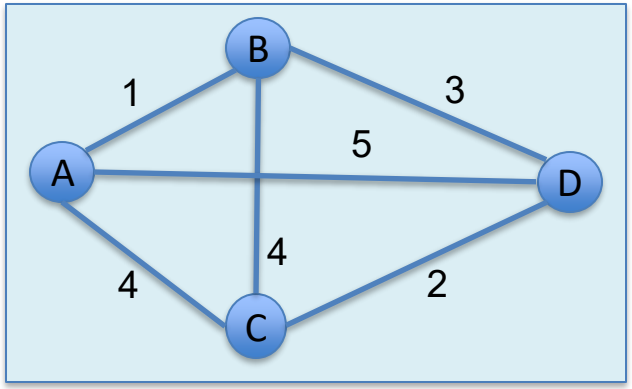
```
LAB: Assignment 3 || Pass exams
```

Example of Applying Graph Algorithms

A city council desires to build a dedicated bicycle network that connect all the city's facilities such as libraries, parks, ... The council know the distance between all pairs of the facilities.

Which algorithm should be used if the council wants to minimize:

- the distance from the council building to all other facilities?
- the distance between any two facilities?
- the cost of building the whole bicycle network, supposing ?

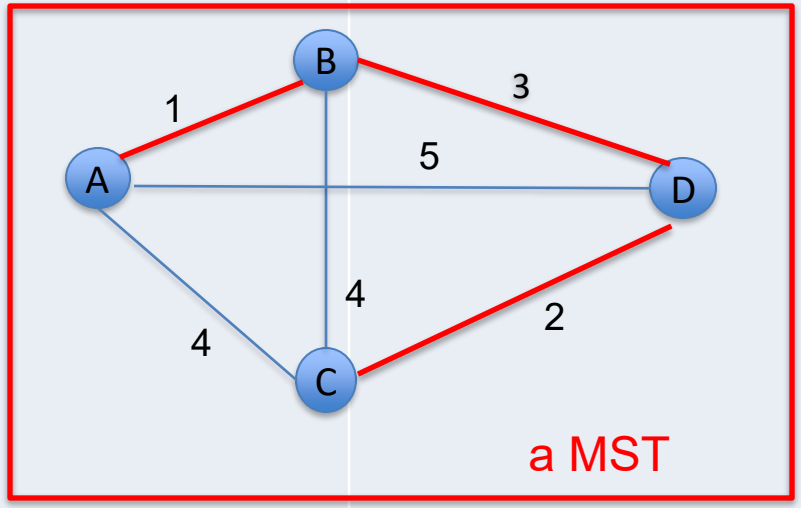


A: City Council Bld
B: Library
C: Sport Centre
D: Park
number: distance in km

Prim's Algorithm vs Dijkstra's Algorithm. Discuss concepts

	Prim's	Dijkstra's
Aim	find a MST (Minimum Spanning Tree)	find SSSP from a vertex s
Applied to	connected weighted graphs	weighted graphs with weights ≥ 0
Works on directed graphs?	No	Yes
Works with negative weights?	?	No
Related concepts for Prim's		

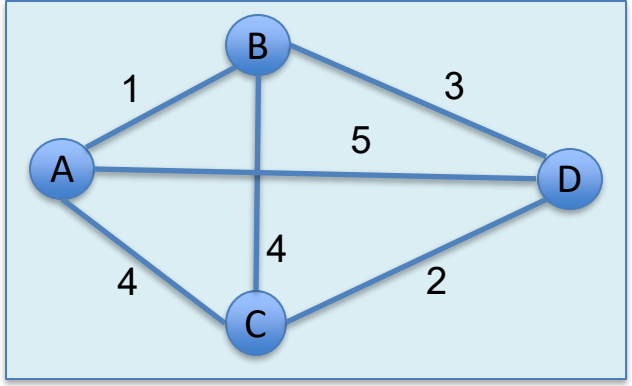
- spanning trees = ?
- MST = ?
- is MST unique?



Dijkstra(G=(V,W,S)	versus	Prim(G=(V,W))
Task: Find SSSP from S (that involves all nodes of a *connected* graph)		**Task: MST (that involves all nodes of a *connected* graph)**
for each v ∈ V : cost[v]:= ∞; prev[v]:= nil cost[S]= 0 PQ:= create_priority_queue(V,cost) with cost[v] as priority of v∈V		for each v ∈ V: cost[v]:= ∞; prev[v]:= nil; **S= any vertex (pick an initial S)** cost[S]:=0 PQ:= create_priority_queue(V, cost) with cost[v] as priority of v∈V
while (PQ is not empty) do u := deleteMin(PQ)		while (PQ is not empty) do u := deleteMin(PQ)

for each neighbour v of u do
 	
if **cost[u] + w(u,v)** < cost[v] then
cost[v]:= **cost[u] + w(u,v)**
 	
update (v, cost[v]) in PQ
prev[v]:= u
		for each neighbour v of u do
		
		if **w(u,v)** < cost[v] then
		cost[v]:= **w(u,v)**
		
		update (v, cost[v]) in PQ
		prev[v]:= u

Aim to minimise the distance from S to any node, that's why **cost[u] + w(u,v)** is considered **cost[v] is min distance from S to V**		Aim to minimise the contribution of individual edge weight to the MST, that's why only **w(u,v)** is considered **cost[v] is min of edge weights to v**



Running Dijkstra(A) to find shortest paths from A to other nodes

At each step, we choose the node with **minimal distance from A**.

step	node with SP found	A	B	C	D
0		0,nil	∞,nil	∞,nil	∞,nil
1	A		**1,A**	4,A	5,A
2	B			**4,A**	4,B
3	C				**4,B**
4	D				

Running Prim's Algorithm to find a MST

At each step, we choose the node with **minimal edge cost**.

We can start from any node. Here, from A, according to the alphabetical order.

step	node added to MST	A	B	C	D
0		0,nil	∞,nil	∞,nil	∞,nil
1					
2					
3					
4					

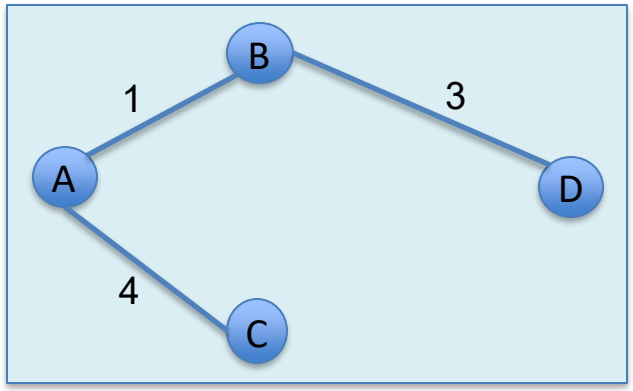
Running Dijkstra(A) to find shortest paths from A to other nodes

step	node with SP found	A	B	C	D
0		0,nil	∞,nil	∞,nil	∞,nil
1	A		**1,A**	4,A	5,A
2	B			**4,A**	4,B
3	C				**4,B**
4	D				

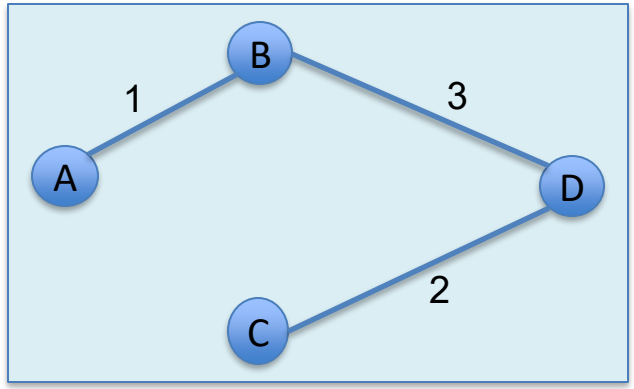
Running Prim's Algorithm to find a MST

step	node added to MST	A	B	C	D
0		0,nil	∞,nil	∞,nil	∞,nil
1	A		**1,A**	4,A	5,A
2	B			4,A	**3,B**
3	D			**2,D**	
4	C				

Edges used



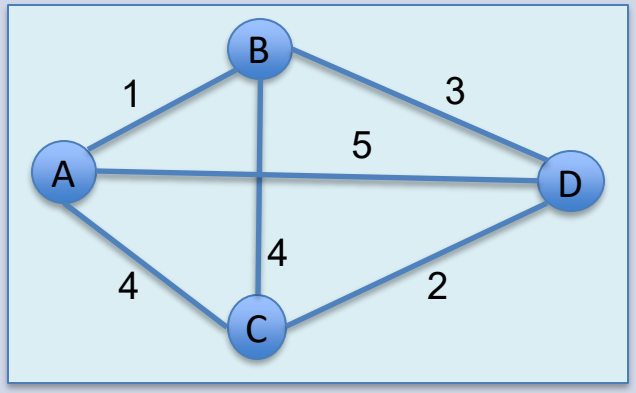
shortest paths from A

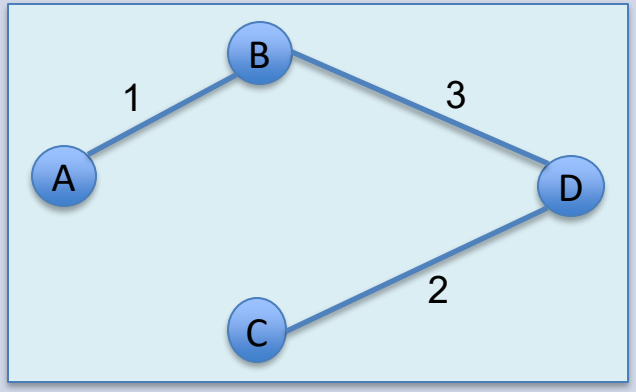


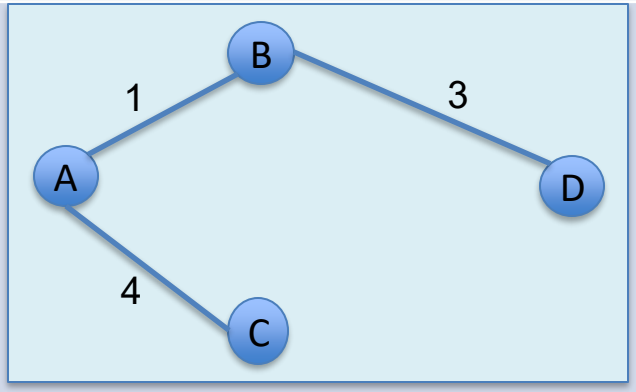
minimum-spanning tree

Prim's Algorithm vs Dijkstra's Algorithm

	Prim's	Dijkstra's
Aim	find a MST	find SSSP from a vertex s
Applied to	connected weighted graphs	weighted graphs with weights ≥ 0
Complexity	O((V+E) log V)	O((V+E) log V)
Works on directed graphs?	No	Yes
Works with negative weights and/or negative cycles	Yes	No
other notes	can be used to find maximum spanning tree by choosing largest-weight edge	**cannot** be used to find longest paths!







Peer Activity: Adapting Prim's Algorithm

Can Prim's algorithm be modified to meet these requirements?

a. Yes, if infinite-weight 'light' edges are accounted for.
b. Yes, if it is repeatedly run on non-MST vertices until exhaustion.
c. No, prior component analysis is required.
d. No, it fails to halt on unconnected graphs.

We want **an algorithm** that:

- works on both connected and unconnected graphs
- finds a set of MSTs spanning each connected component
- doesn't require separate component analysis before its commencement

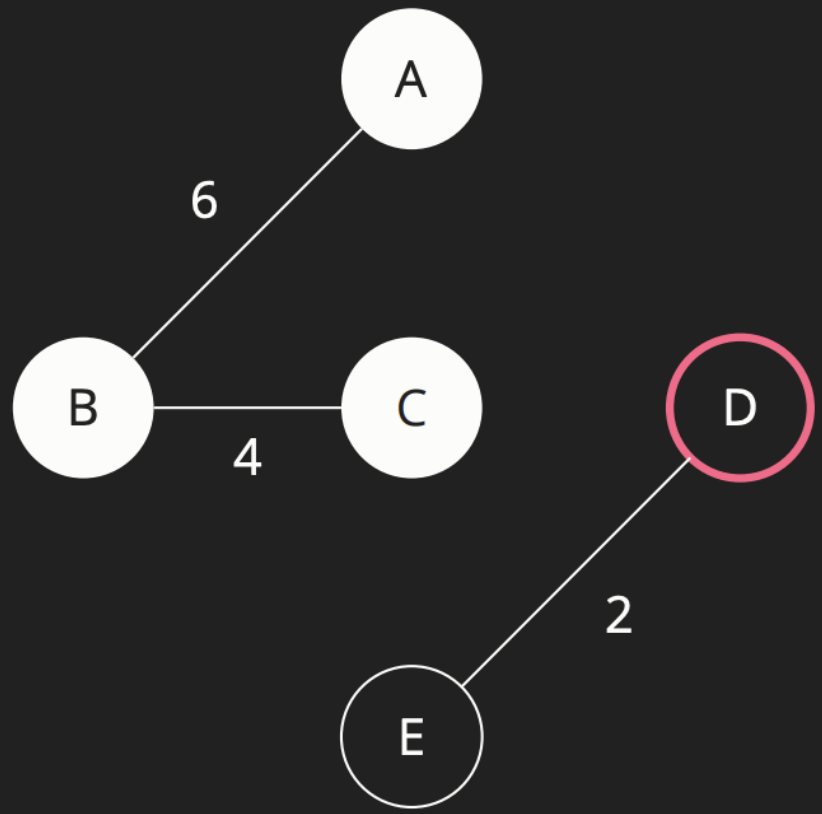
Peer Activity: Adapting Prim's Algorithm

Can Prim's algorithm be modified to meet these requirements?

b. Yes, if it is repeatedly run on non-MST vertices until exhaustion.

Why?

- run Prim's on the source node
- keep track of the remaining nodes that are not yet within any MST
  - run Prim's on one of these nodes
  - repeat until all nodes are within some MST



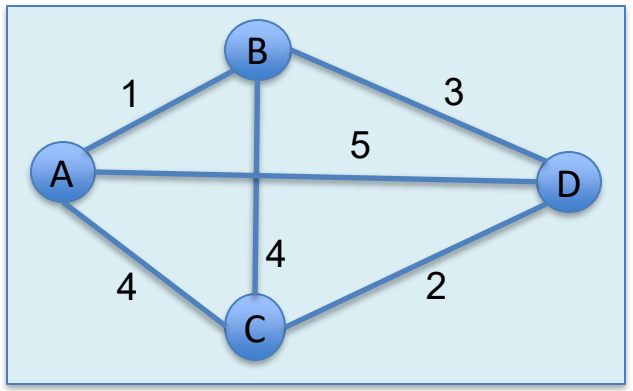
Kruskal's algorithm for MST [not using graph traversal!]

Purpose: Find MST of G= (V,W)

Prim's algorithm: processing node-by-node, ie. adding a new node to MST at each step.

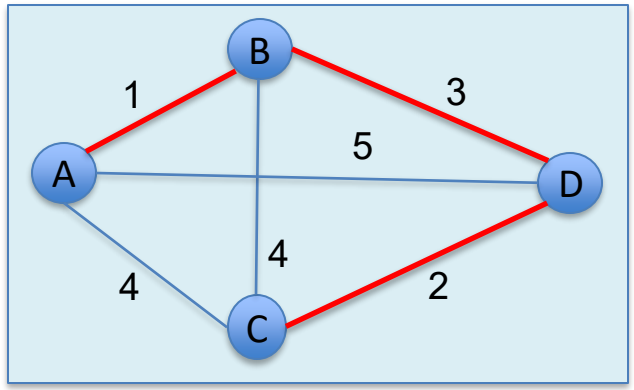
Kruskal's algorithm: operates *edge-by-edge*.

```
0 set MST to empty
3 for each (u,v), in increasing order of weight:
4     if  ( (u,v) does not form a cycle in MST):
5         add edge (u,v) to MST
```



edge weight
(in sorted order)

AB 1
CD 2
BD 3
AC 4
BC 4
AD 5



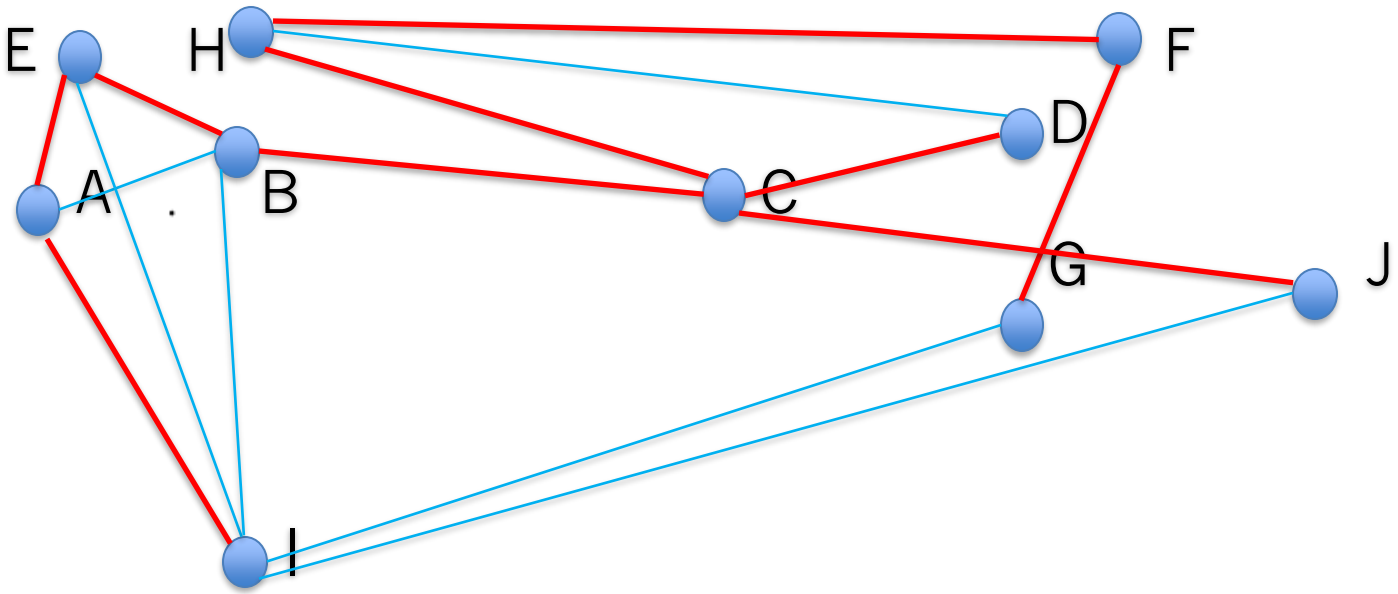
Kruskal's algorithm [not using graph traversal!]

Purpose: Find MST of G= (V,E,w)

Prim's algorithm: processing node-by-node, ie. adding a new node to MST at each step.

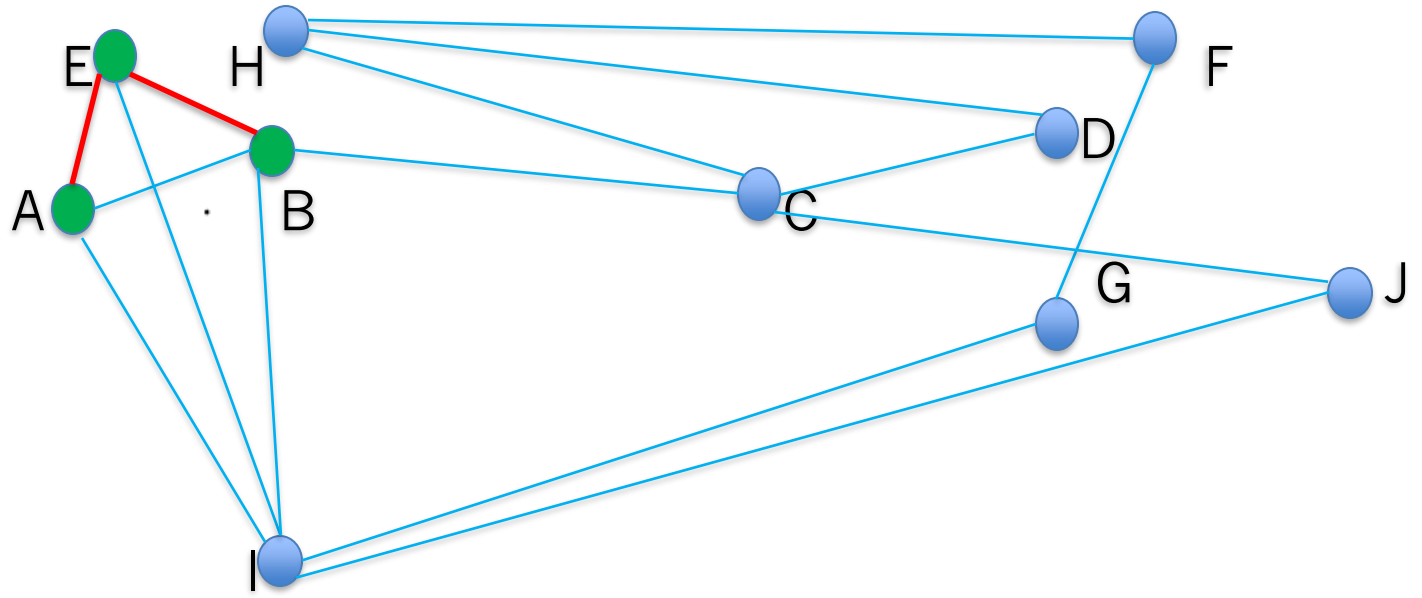
Kruskal's algorithm: operates *edge-by-edge*.

```
0 set MST to empty
3 for each (u,v), in increasing order of weight:
4     if  ( (u,v) does not form a cycle in MST):
5         add edge (u,v) to MST
```



edge	weight
AE	3
BE	4
AB	5
CD	6
FG	6
AI	7
BI	7
BC	8
CH	8
CJ	8
EI	8
DH	9
FH	9
IG	9
IJ	10

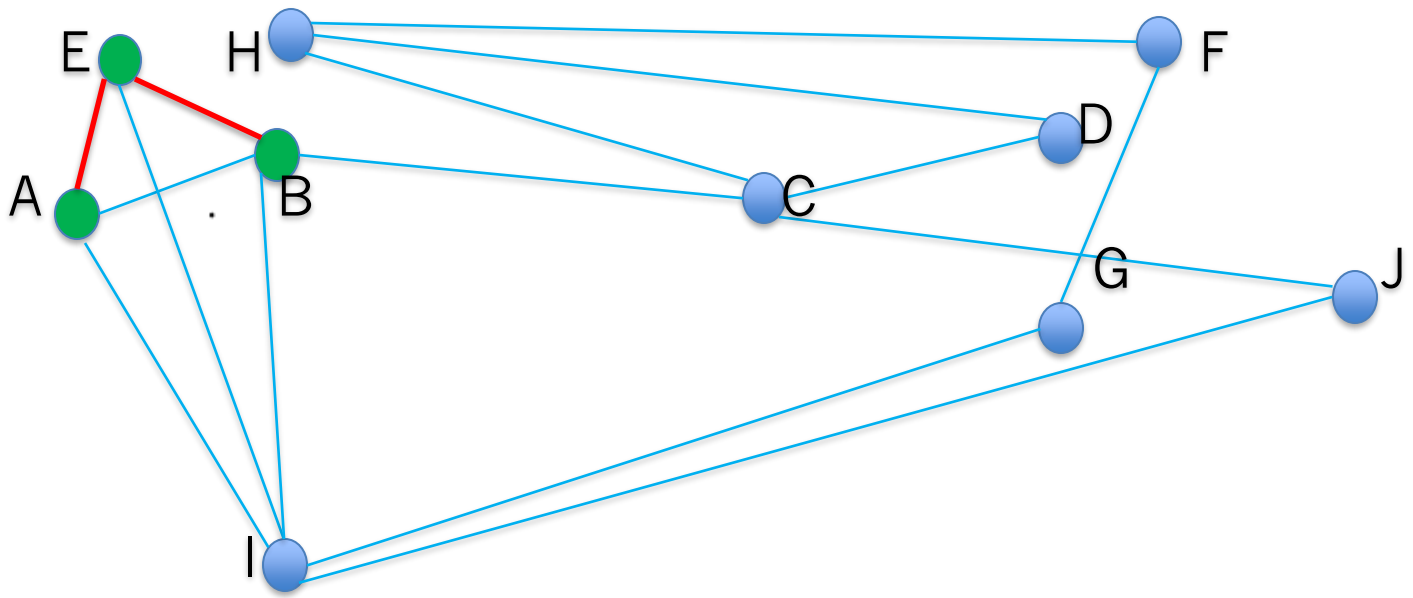
Easy to run manually – but how to implement?



Suppose that the above 3 green vertices and 2 red edges are in one MST-so-far.

Which edge should be added next? How do we prevent cycles in the implementation?

Implement Kruskal's Algo using Disjoint Sets



edge weight
AE 3
BE 4
AB 5
CD 6
...

Consider each MST in the set of MST-so-far (with red edges) as a set. Currently, the MST has 8 sets: {A,B,E}, {C}, {D}, {F}, {G}, {H}, {I}, {J}.

Edge {A,B} is skipped since A and B are in the same set. (C,D) is added next and join a new set {C,D}.

Initially, each node forms a singleton set (e.g., {A}, {B}, ..., {J} here).
The goal is to merge these sets to a single set while constructing the Minimum Spanning Tree (MST).

Kruskal's Algorithm:
For each edge (u,v) in the increasing order of weight:

1. FIND: Check if u and v belong to the same set:

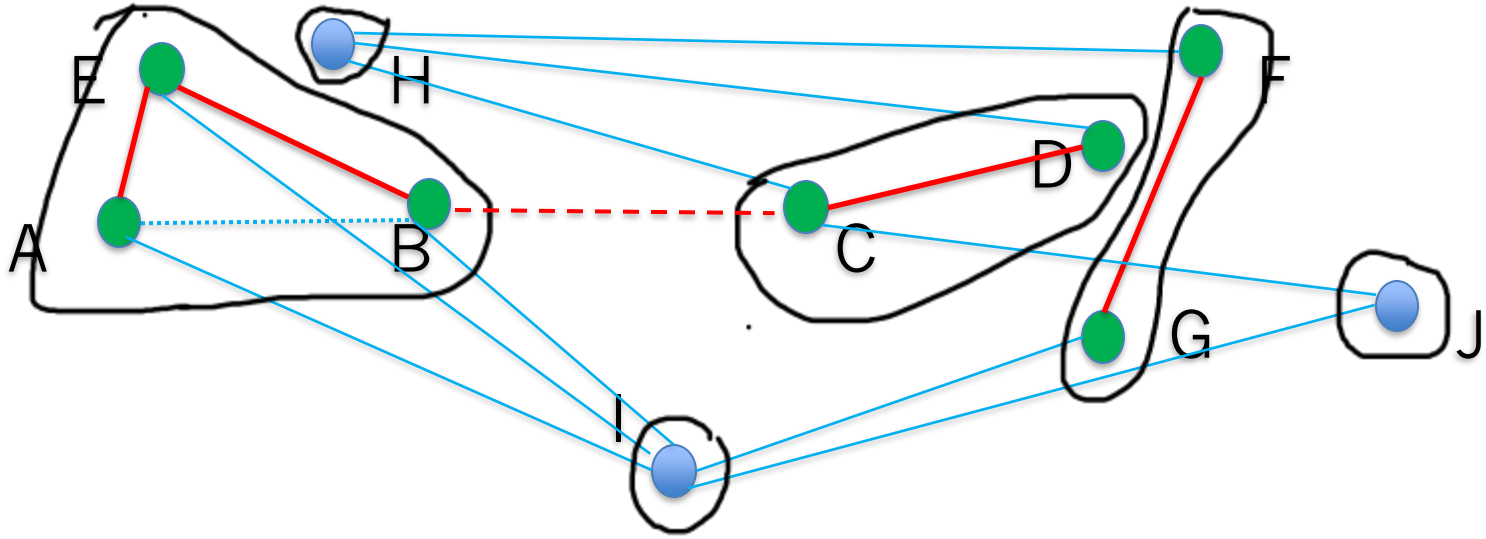
If they are in the same set, the edge is skipped to avoid forming a cycle.

2. UNION: If they belong to different sets:

Add the edge (u,v) to the MST and **union** the two sets containing u and v into one.

The process continues until all nodes are connected in a single set, forming the MST.

Using disjoint sets



Using disjoint sets:

- At the start, each node is a singleton disjoin set ed into ABCDE → we will have 5 disjoint sets

Needed to implement:

- an ID for each set (or MST-so-far): choose a root node and use it as the ID of the set
- Operator Find(u) : find the set that node u belongs to
- Operator Union(u,v) : join the disjoin sets of u and v into a single set

Kruskal's algorithm

Purpose: generate MST with Efficient checking for cycle in finding MST

```
0    set X= empty. X is the MST-so-far
1    E1 = E, but sorted in increasing order of weights
2    for each u: makeset(u) (build single-element set {u})
3    while ( |X| < |V|-1 ) :
3a       set (u,v)= the next edge in E1
4        if  (find(u) ≠ find(v) ):    // if u & v do not belong to a same set
5            add edge (u,v) to X      // .. extends MST by one more edge
6            union(u,v)               // .. and unite the set of u and the set of v
```

- See additional Additional Slides for the work of `makeset(u)`, `find(u)`, `union(u,v)`
- **Complexity:**

→ Step 3-6: can be implemented with O(E) complexity if using optimisations (join-by-rank and path compression)

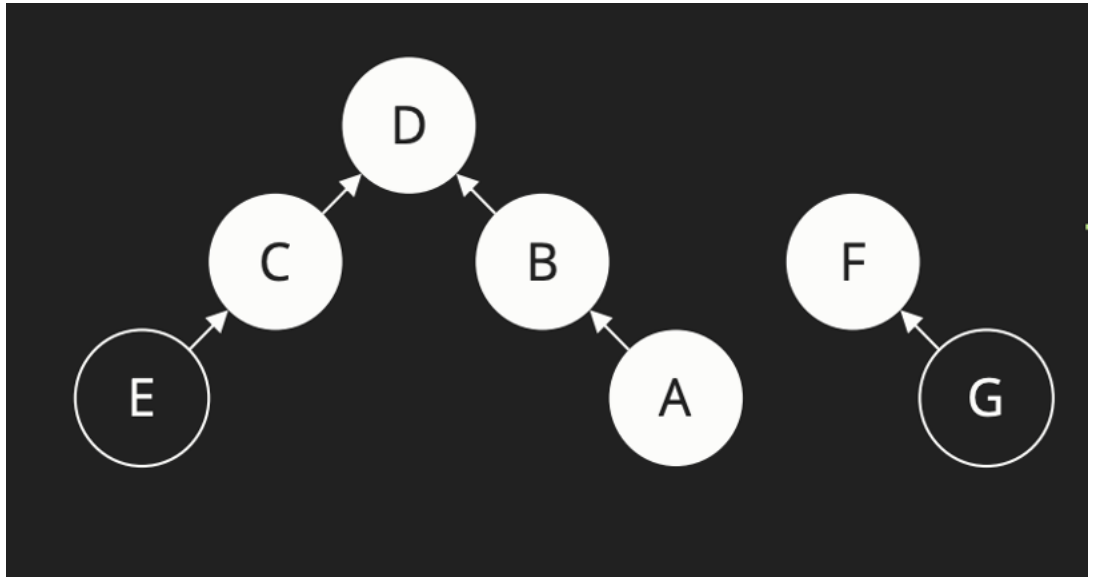
→ The cost of Kruskal's Alg is dominated by ElogE of the sorting phase in step 1

Notes

Complexity of *a single* union and find using disjoint-set:

- find: O(1)
- union(u,v): time for tracing depends on the depth of the tree
  - naïve: O(V)
  - optimization: using union-by-rank (weighted) O(log V)
  - optimization: using union-by-rank + path compression: O(1)

consider union(E,G) in:



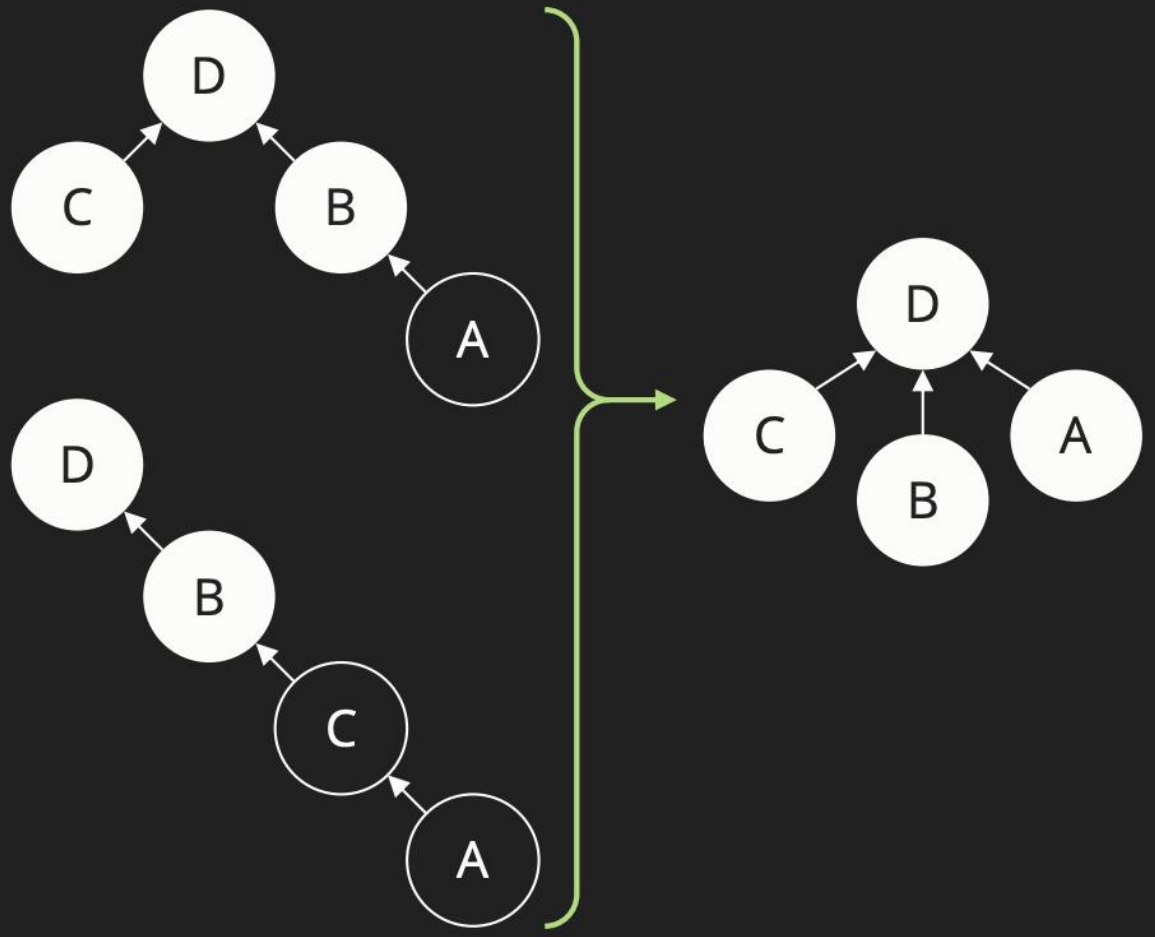
MST: Kruskal's Algorithm: Optimisations

There are several tree-building **optimisations** for Kruskal's algorithm:

1. **Path compression**
    1. new subtrees:

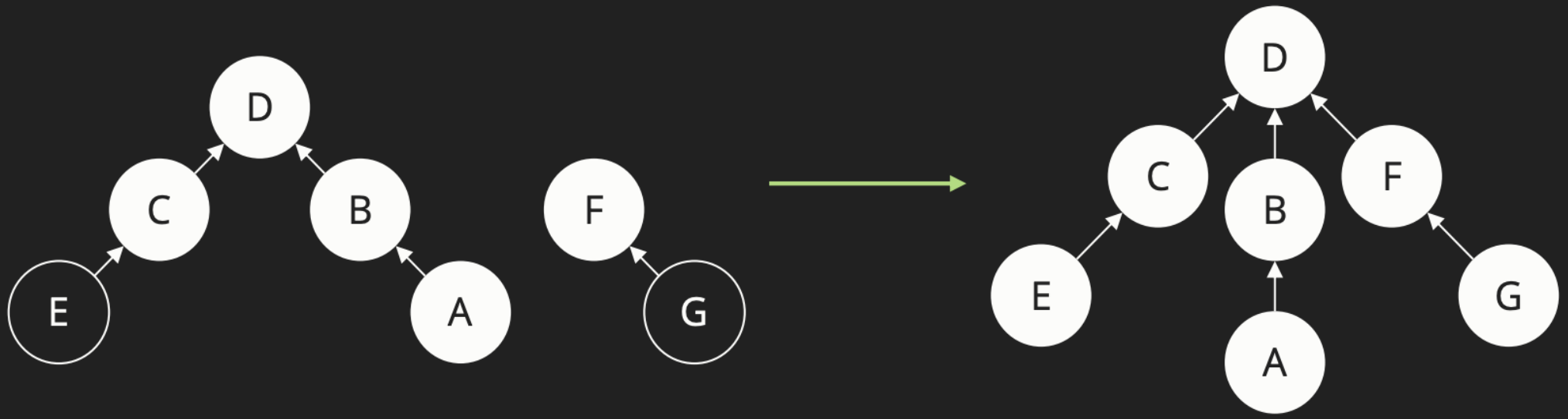
       can be attached directly to the root
    2. when moving up a tree:

       nodes can be connected to the root



MST: Kruskal's Algorithm: Optimisations

2. **Union-by-rank:** always join a smaller tree to a larger tree



Justify the Complexity

Kruskal's: generate MST with Efficient checking for cycle in finding MST

```
0 set X= empty. X is the set of the MST-so-far
1 E1 = E, but sorted in increasing order of weights          O(E log E)
2 for each u:                                                V ×
    add makeset(u) to X                                        O(1)
    (build set {u} and add to X with X= X {u})

3 for each edge (u,v) in E1 (in increasing order of weight): E ×
4   if  (find(u) ≠ find(v) ):                                  O(1)
5     union(u,v)                                               O(1)
6     add (u,v) to X                                           O(1)
```

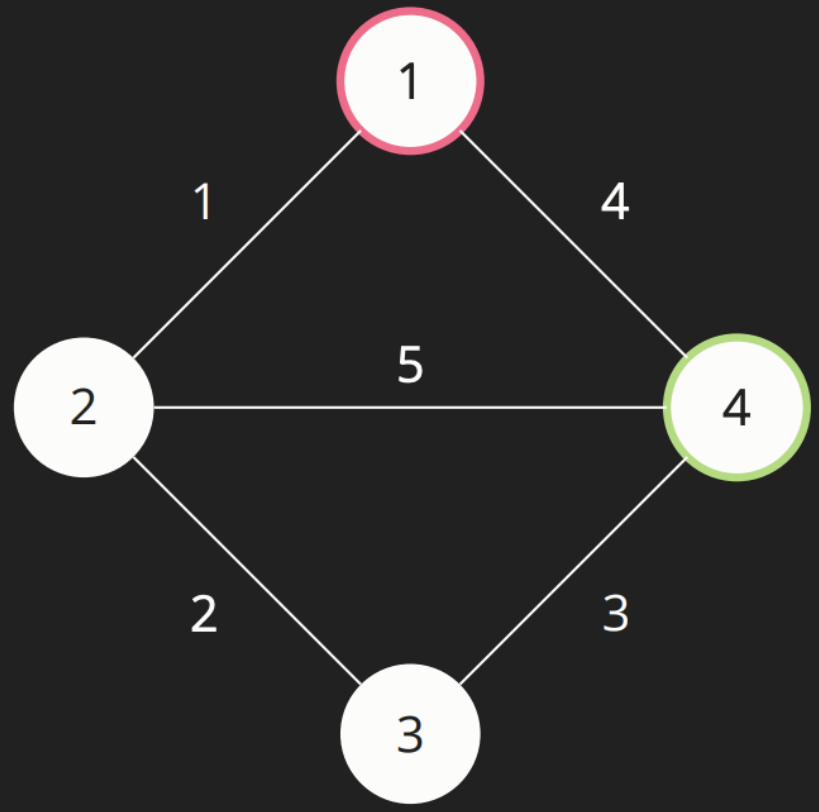
	Prim	Kruskal
General	(E+V) log V	E log E
Dense Graph	E log V	E log E
V<<E, Prim's is faster		
Sparse Graph	V log V	V log V
Kruskal's is faster because of the data structures		

Peer Activity: Minimum Fatness Paths, Pt. 1

Which of these best facilitate finding minimum fatness paths between any two vertices in a connected, weighted graph?

a. Original graph representation
b. Dijkstra's auxiliary arrays
c. Floyd-Warshall's auxiliary matrices
d. Minimum spanning tree

fatness of path P: maximum weight of any edge in P



Peer Activity: Minimum Fatness Paths, Pt. 1

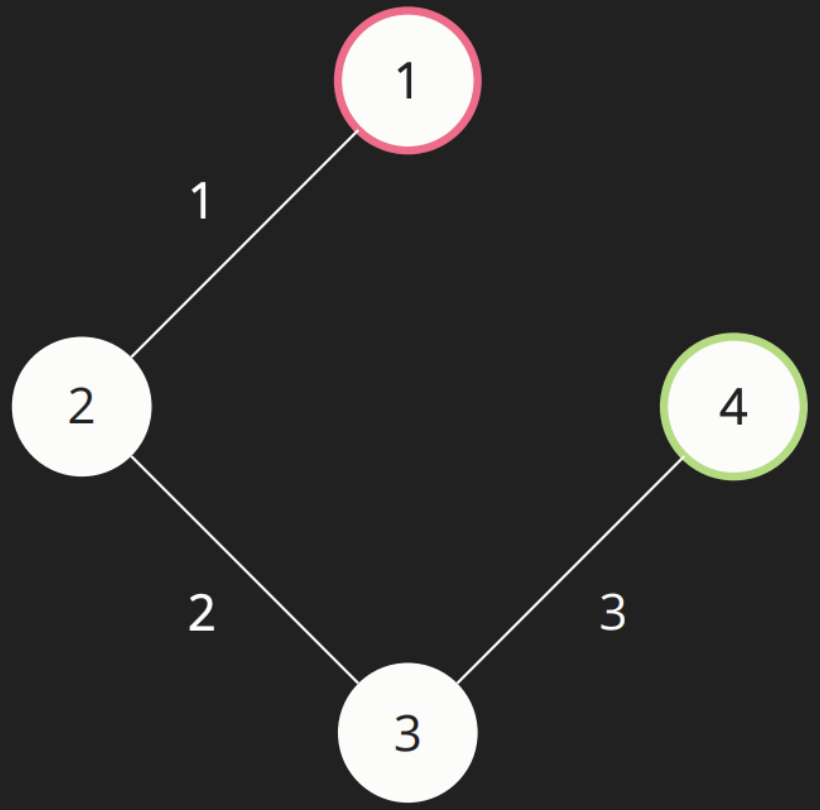
Which of these best facilitate finding minimum fatness paths between any two vertices in a connected, weighted graph?

d. Minimum spanning tree

Why?

- spans all vertices with minimum sum of edge weights
  - minimises fatness between all vertices
  - contrapositive: existence of fatter path within the MST invalidates it

fatness of path P: maximum weight of any edge in P



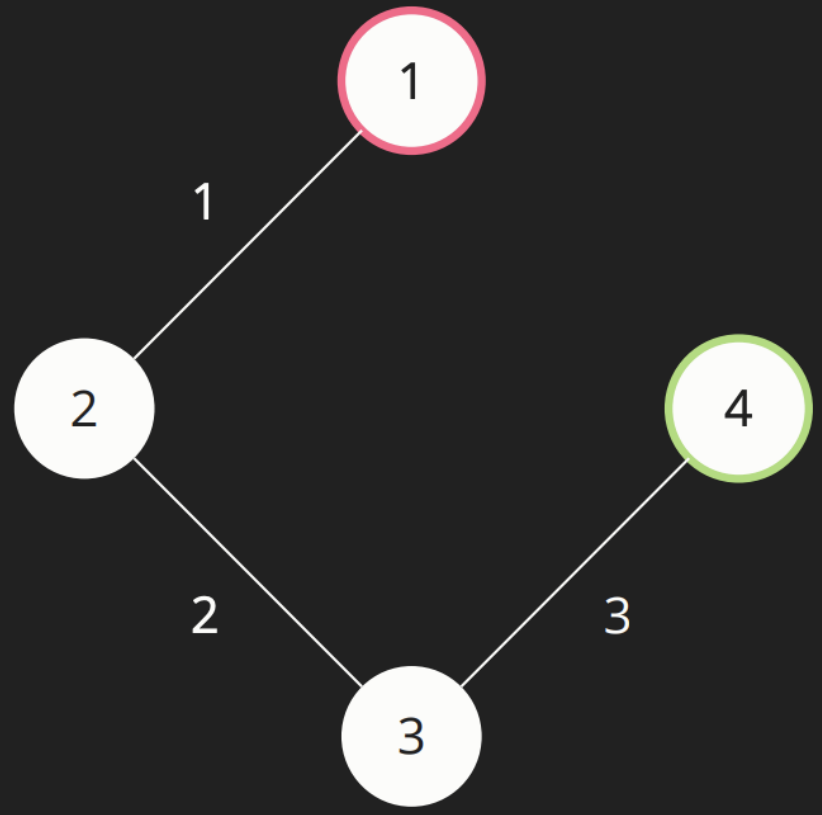
Peer Activity: Minimum Fatness Paths, Pt. 2

Which of these most efficiently finds the minimum fatness path between any two vertices in a weighted minimum spanning tree?

a. Dijkstra's algorithm

b. Floyd-Warshall's algorithm

fatness of path P: maximum weight of any edge in P



Peer Activity: Minimum Fatness Paths, Pt. 2

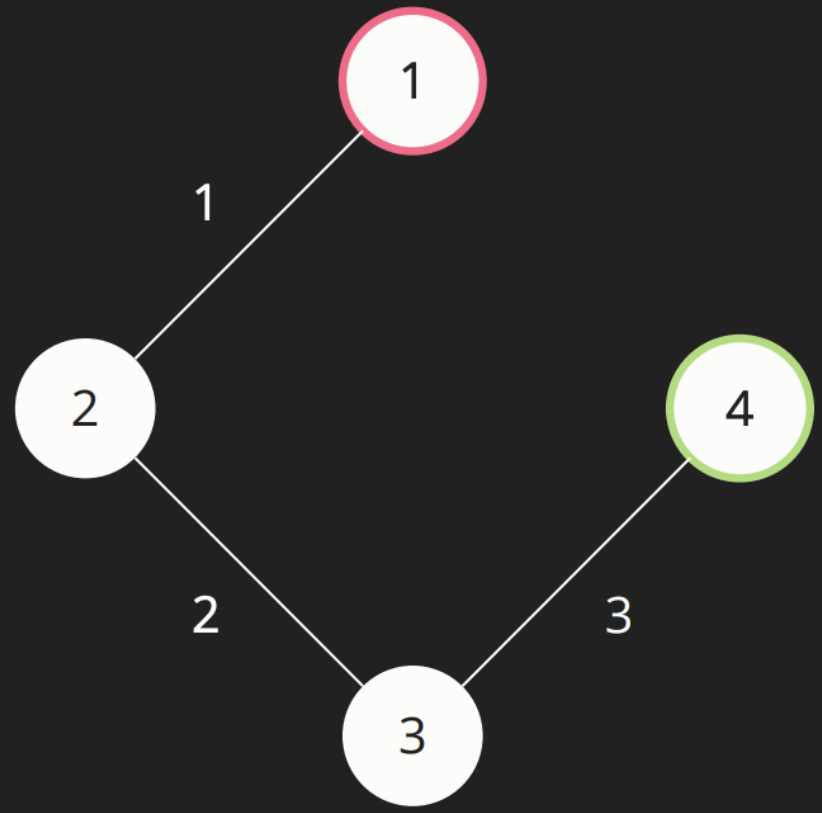
Which of these most efficiently finds the minimum fatness path between any two vertices in a weighted minimum spanning tree?

a. Dijkstra's algorithm

Why?

- minimum spanning trees are sparse graphs $\rightarrow |E| \approx |V|$
  - Dijkstra's $\rightarrow O(|V|^2 \log |V|)$
  - Floyd-Warshall $\rightarrow \Theta(|V|^3)$

fatness of path P: maximum weight of any edge in P



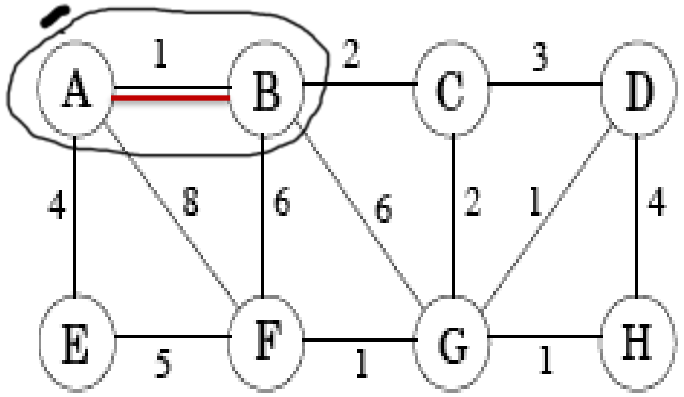
Lab Time:

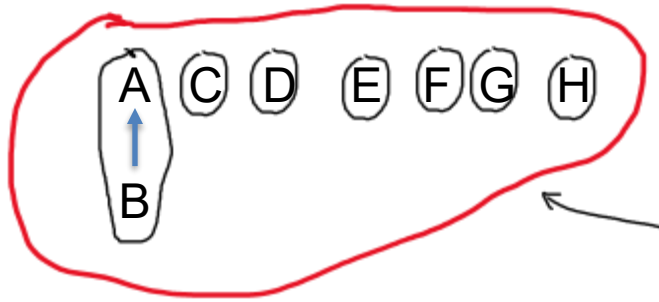
- W12.10 (Peer Activities)
- finish ass3, or
- go through some questions, especially short questions, in past-exam papers,
- give questions to the in-lazy-mode Anh

Thank You and Good Luck!

Additional Slides

Kruskal's Algorithm Example: Kruskal's disjoint set after selecting the first (smallest) edge



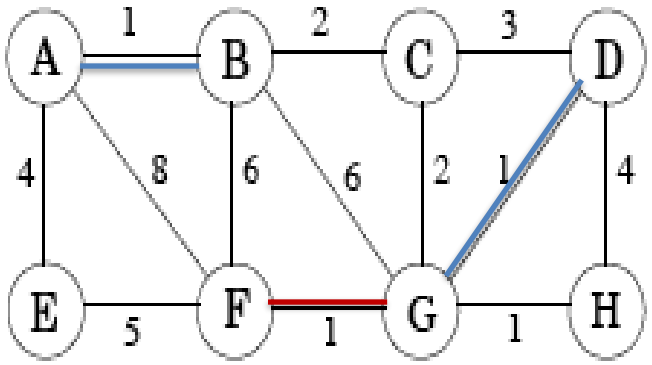


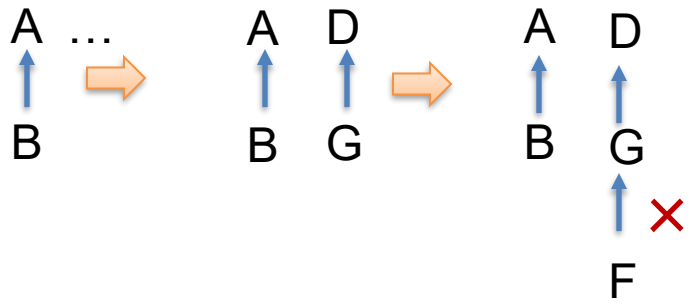
the MST-so-far, which is a set of tree, each tree is a disjoint set.

Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7

Note: number in table body represents tree ID at each step. Tree ID for a node = ID of the root of the tree the node belongs to.
If a tree ID is the same as node ID, then the node is the root of its tree
(*another choice is using –k for recognizing the node ID is a root node and the tree has k nodes in total, a positive value shows the root of the node ID*)

Example (Kruskal's)



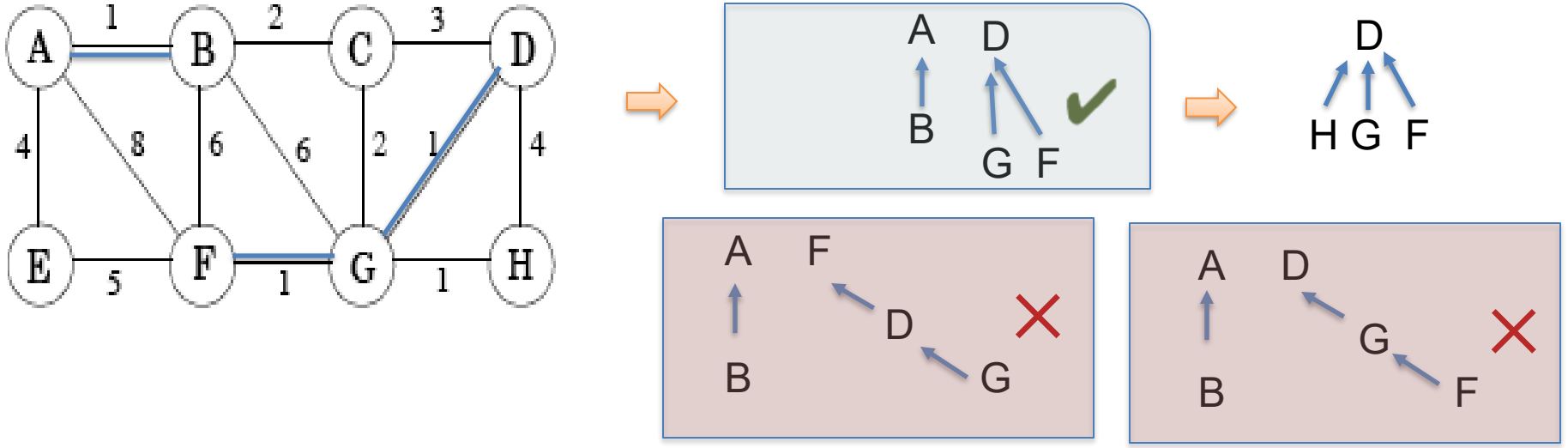


Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	**3**	4	**?**	3	7

Note: number in table represents tree ID, if a tree ID is the same as node ID, then the node is the root of its tree

Optimisation 1: weighted (aka. join-by-rank)

1) join smaller tree to bigger tree (*weighted*, or *join-by-rank* optimization)
2) when joins, joins to the root, ie. id[F]= id[G] instead of id[F]= G



Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	3	4	6?	3	7
G---F	0	0	2	3	4	**3**	3	7

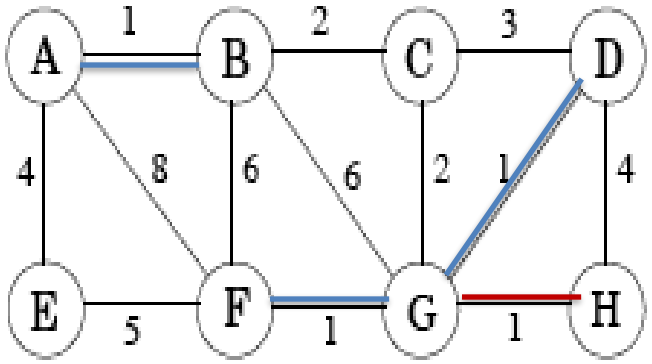
Note: to be able to decide which tree is bigger, in the table for each root node we store the number of nodes, say k, as –k.
As the beginning, all nodes have value -1.
Then, a negative value means a root, a non-negative value shows the reference to root.

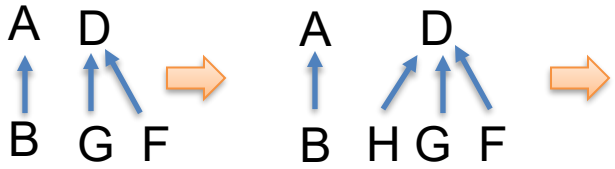
See live lecture for details.

Optimisation 1: weighted

1) join smaller tree to bigger tree (*weighted*, or *join-by-rank* optimization)

2) when joins, joins to the root, ie. id[F]= id[G] instead of id[F]= G

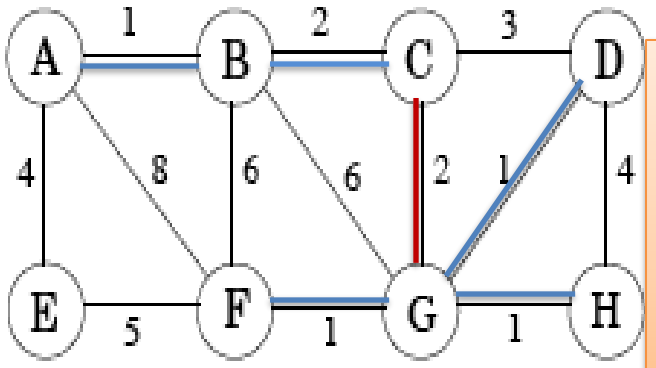




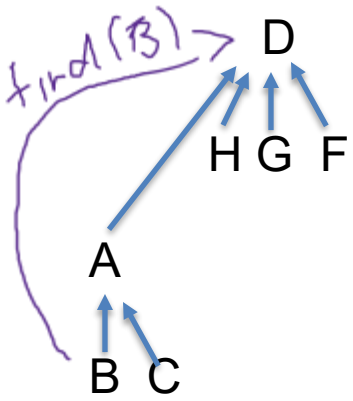
Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	3	4	**3**	3	7
G---H	0	0	2	3	4	3	3	3

Optimisation 2: Path compression

when doing find(x), also compress the path by join x directly to find(x)

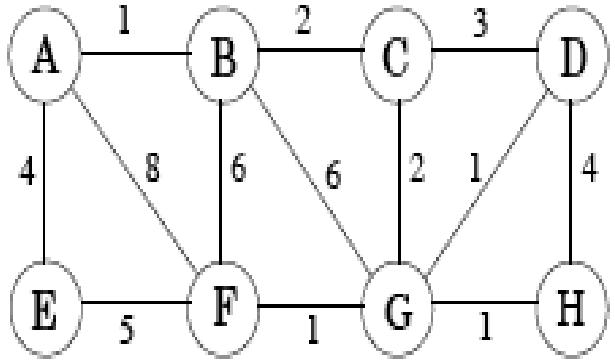


- Now, the pathlen of B,C is >1
- In the future, if we call find(B) we will have find(B)= D, and at that time we will make B point directly to D. That is called *path compression*.

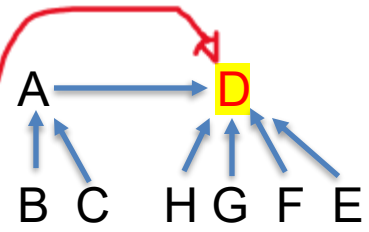


Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	**0**	2	3	4	5	6	7
D---G	0	0	2	3	4	5	**3**	7
G---F	0	0	2	3	4	**3**	3	7
G---H	0	0	2	3	4	3	3	**3**
B---C	0	0	**0**	3	4	3	3	3
C--G	**3**	0	0	3	4	3	3	3

at the end: MST has single root (id. node that Node ID == Tree ID)



After adding A---E to the MST-so-far, the latter has enough V-1 egdes, and we stop. Note that in our tree-array, there is only a single tree (only one root) at this stage.



Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	3	4	**3**	3	7
G---H	0	0	2	3	4	3	3	3
B---C	0	0	0	3	4	3	3	3
C---G	3	0	0	3	4	3	3	3
A---E	3	0	0	3	**3**	3	3	3

An alternative method is start with value -1 in the first row. At any time, a value –k at node i means that "i is a root node, and there are k nodes in the tree i". A positive value m at node i means i belongs to the tree rooted at node m.

See a detailed example in lecture slide Week 11.

Justify the Complexity

Kruskal's: generate MST with Efficient checking for cycle in finding MST

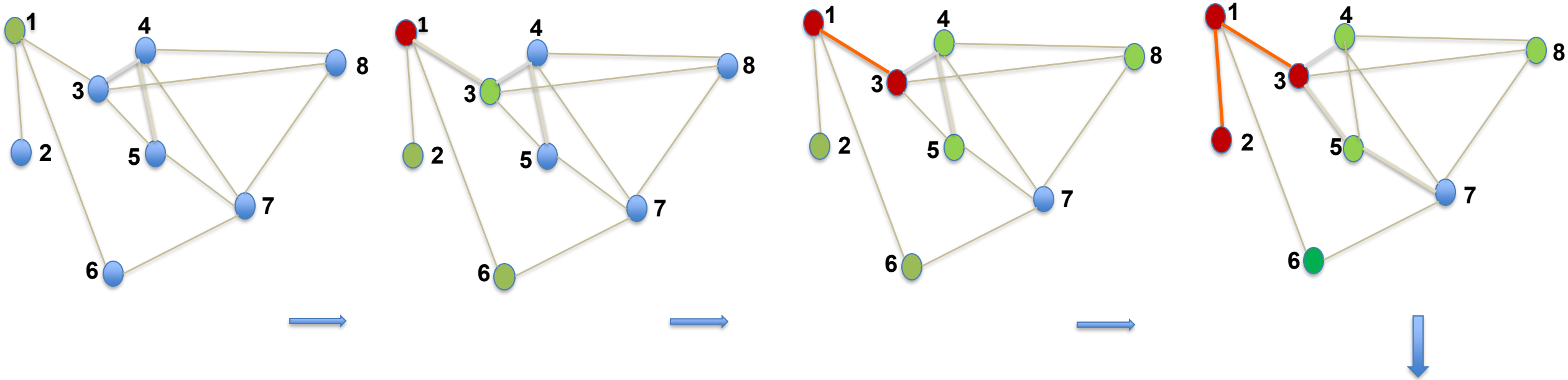
```
0 set X= empty. X is the set of the MST-so-far
1 E1 = E, but sorted in increasing order of weights          O(E log E)
2 for each u:                                                V×
    add makeset(u) to X                                        O(1)
    (build set {u} and add to X with X= X {u})

3 while (|X|<|V|-1) :                                        V×      ???
    (u,v)= next edge in E1                                     O(1)
4   if  (find(u) ≠ find(v) ):                                  O(1)
5     union(u,v)                                               O(1)
6     add (u,v) to X                                           O(1)
```

True or False:

- Loop (3) is O(V), and hence we don't need to fully sort E1, just make E1 a minheap. As the result, step 1 is O(E), step 3-6 is O(V log E)?
- We can use distribution counting and make Kruskal's to make step 1 be O(E)
- Since E= $O(V^2)$ in the worst case, Kruskal's is O(E log V) just like Prim's.

Greedy example: Dijkstra's Algorithms from node 1

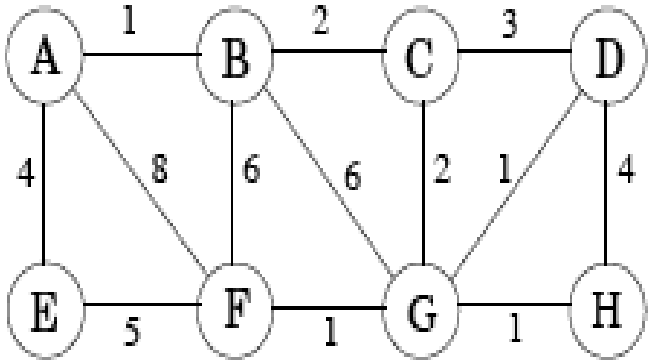


Note: In the graph, the length of an edge represents the edge's weight, smaller length means smaller weight. Green node= node with *some found distance* from the source 1. Red node= node with *found shortest distance* from the source 1. Blue node= node with distance = ∞

Would you apply the greedy policy when applying for a job after graduation :-?

Example of Tracing the Prim's Algorithm

Suppose we want to find the minimum spanning tree of the following graph.

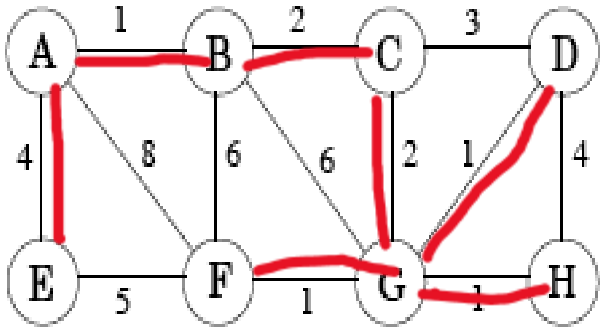


(a) Run Prim's algorithm; whenever there is a choice of nodes, always use alphabetic ordering (e.g., start from node *A*). Draw a table showing the intermediate values of the `cost` array.

done	a	b	c	d	e	f	g	h
	0,nil	-	-	-	-	-	-	-

note: in the table, - denotes the pair ∞,nil

Example



	a	b	c	d	e	f	g	h
	0,nil	-	-	-	-	-	-	-
a		1,a	-	-	4,a	8,a	-	-
b			2,b	-	4,a	6.b	6,b	-
c				3,c	4,a	6,b	2,c	-
g				1,g	4,a	1,g		1,g
d					4,a	1,g		1,g
f					4,a			1,g
h					4,a			
a								