COMP20003 Workshop Week 12

Welcome to the last workshop!

Good Luck!

1	MST & Greedy Algorithm for building MST
2	Prim's Algorithm
3	Kruskal's Algorithm
	`LAB: Assignment 2 || Pass exams`
	Question of the Year: Do you still need time for assignment 3. Send me a letter Y or N ☺
	Note: Grady is running consultation right now …

MST & Greedy Algorithm

Greedy algorithm:

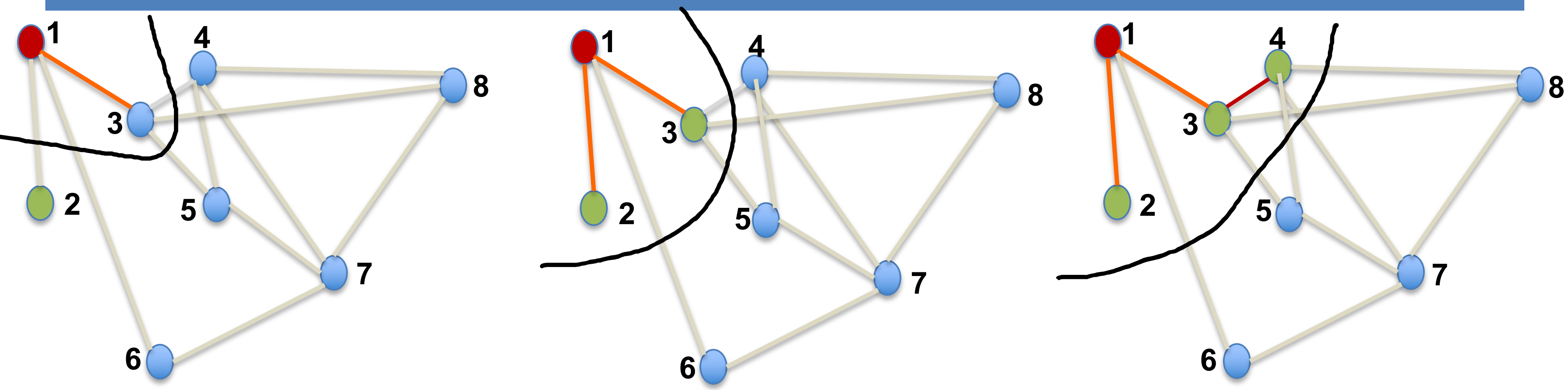
A myopic policy of always taking the “best” bite in each step.

NOT always works…

In many cases it’s the best policy!

Dijkstra’s algorithm is greedy.

Greedy example: Dijkstra's Algorithms



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

MST - overview

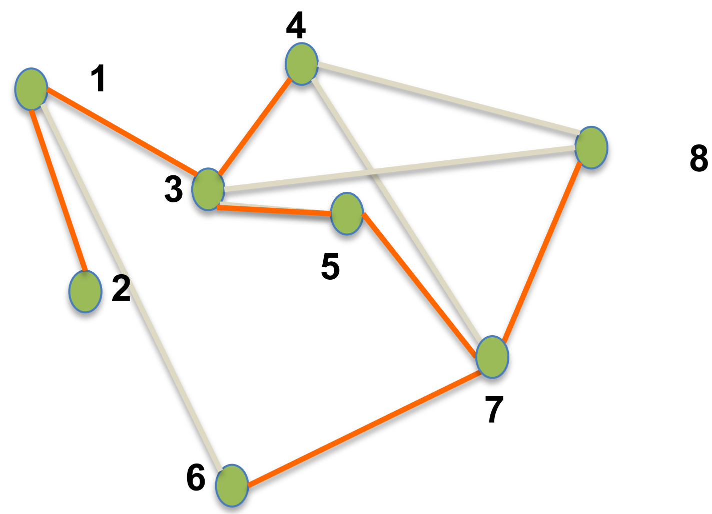
Task: give a *connected*, *weighted* graph `G= (V, E ,w)`, find a MST for G.

What's a spanning tree? How many edges in a spanning tree?

What's a MST? Can `G` have more than one MST?

Further Topics:

- Which algorithms? Complexity = ?
- Which algorithm is better for:
  - dense graphs
  - sparse graphs



In this graph, the visual length of an edge represent its weight. In particular edges (3,4), (4,5) and (5,3) have the same weight.

MST & Greedy Algorithm

Greedy algorithm can be used for the MST task, for example:

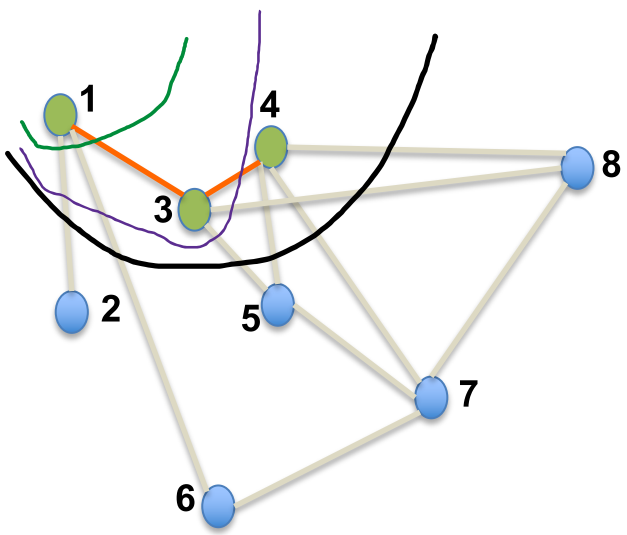
Prim: MST built by taking a vertex at a time	Kruskal: MST built by taking an edge at a time

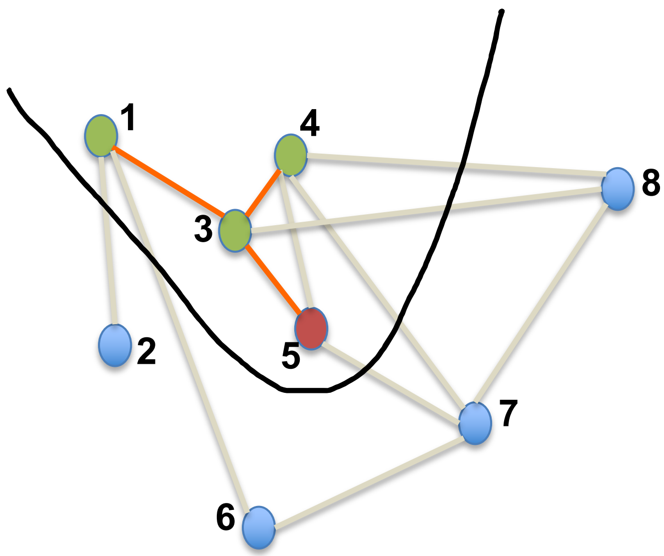
`T= any vertex`
`while ( |T| < |V|):`
`    add to T the vertex that`
`    has least distance to T`
 	
Note: distance between node u and set T is defined as the minimal distance between u and any member of T
	`T= EMPTY SET OF edges`
	`while ( |T| < |V|-1):`
	`    add to T the lightest edge`
	`    that doesn't make cycle in T`

Prim's Algorithm

- Consider a randomly-chosen vertex as the MST-so-far.
- At each stage we, expand the MST-so-far by adding a vertex to that tree (the one that is closest to the so-far MST).

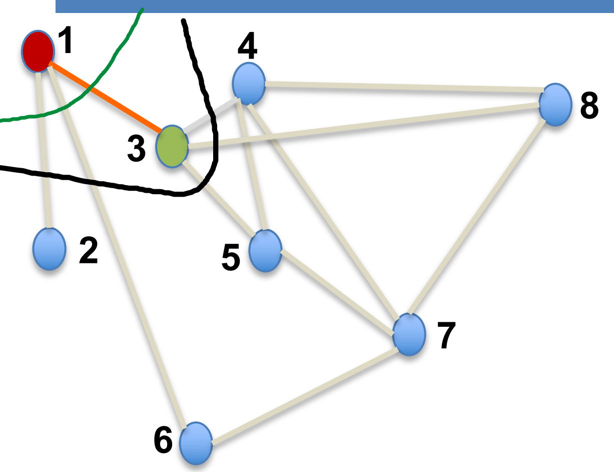
Sounds familiar? Similar to a studied algorithm?

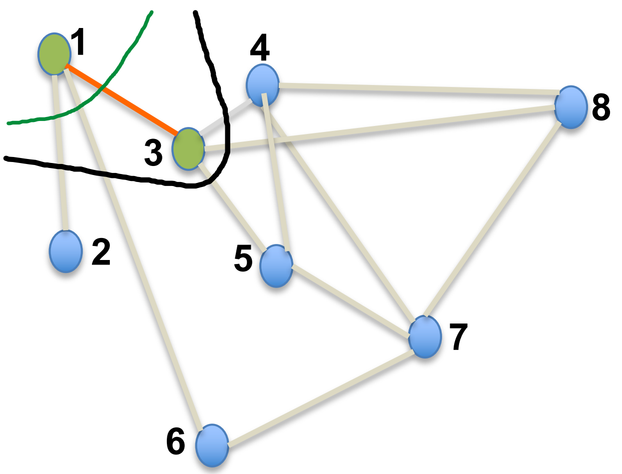




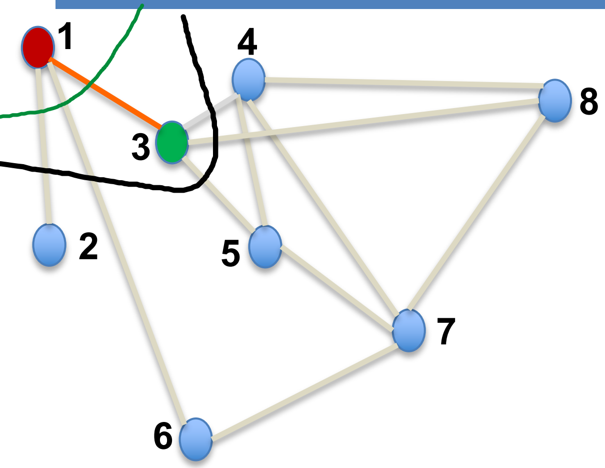
Note: *in the graph: the visual length of an edge represents its weight. For example, edge (3,4) has the smallest weight, and the next is (3,5).*

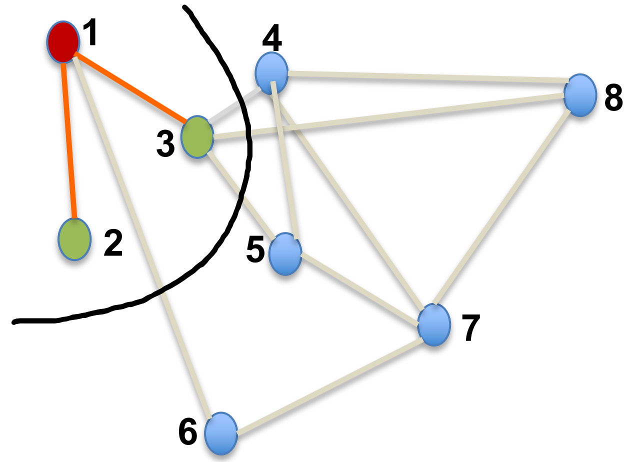
Comparing: Dijkstra’s & Prim's Algorithms

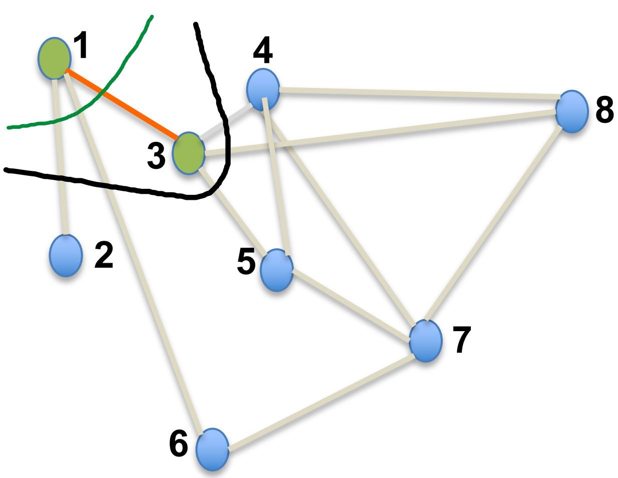


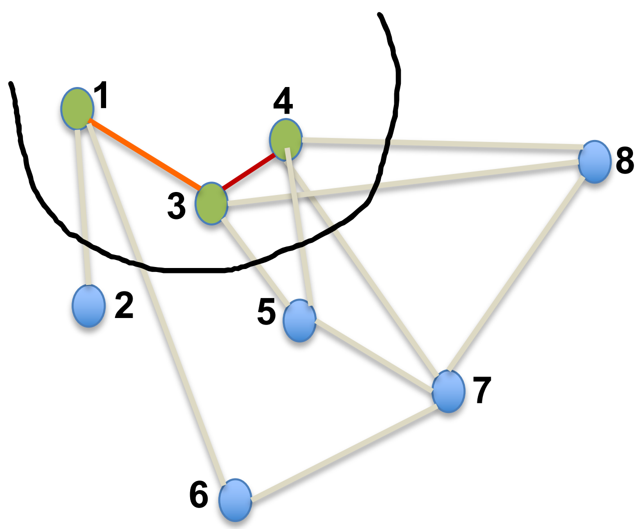


Comparing: Dijkstra’s & Prim's Algorithms

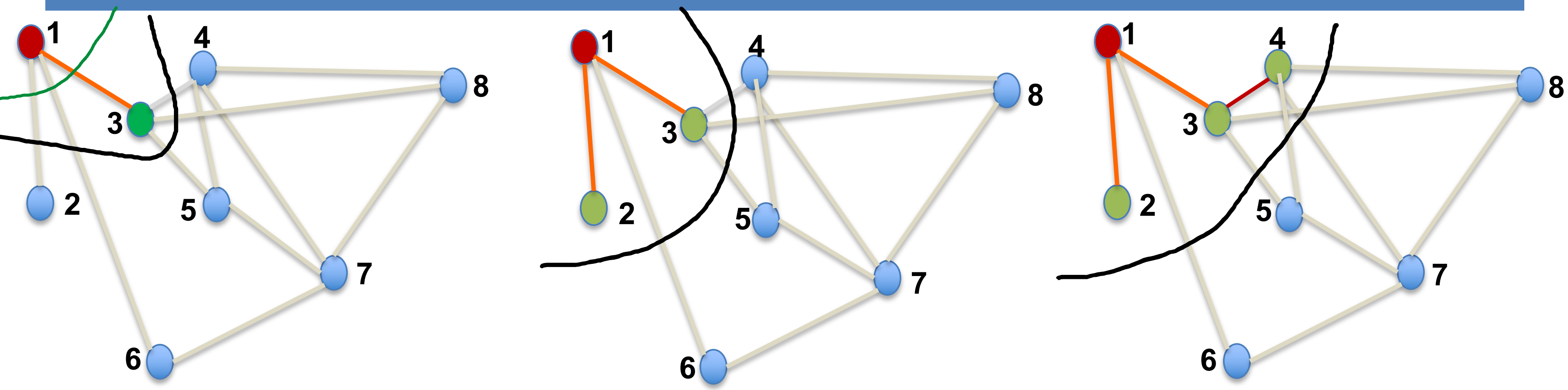




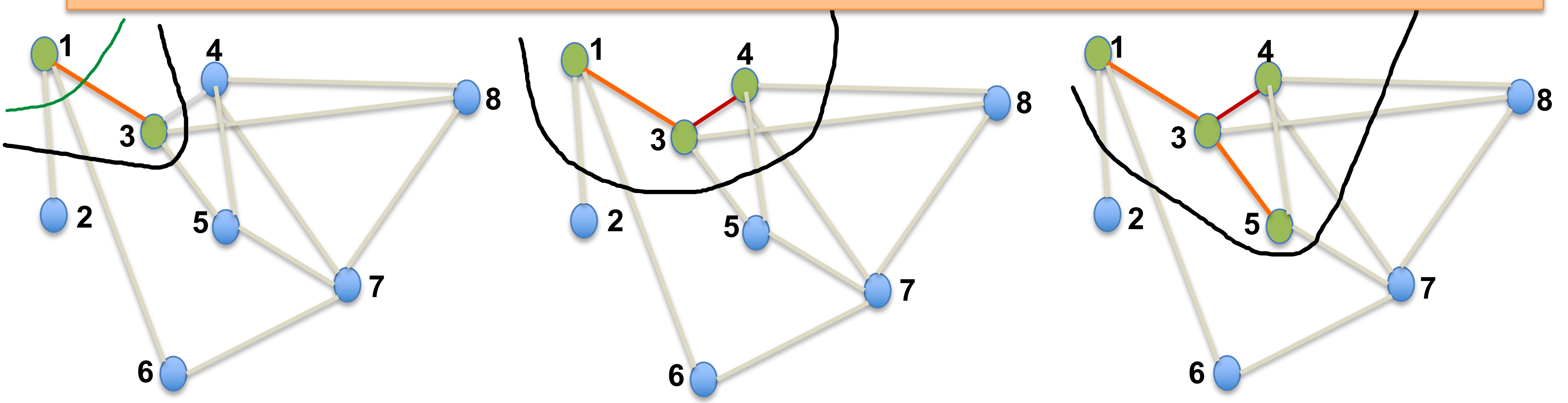




Comparing: Dijkstra’s & Prim's Algorithms



- Dijkstra’s: choose node with the shortest distance to the red node
- Prim’s: choose node with the shortest distance to any of the green nodes



Prim's algorithm: operates vertex-by-vertex

Given a (connected) weighted graph G

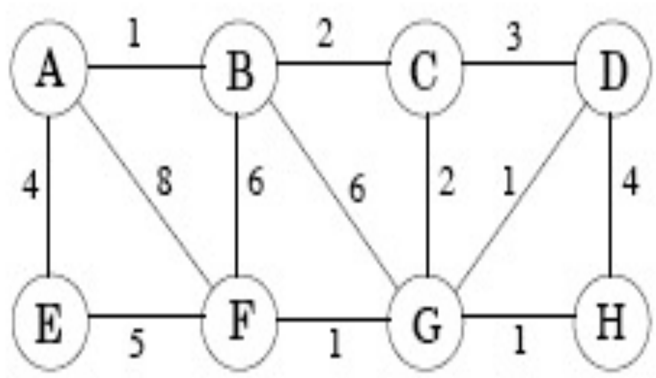
Prim(G): Find a MST of G	Dijkstra(G,s): find shortest paths from s

<pre>for each u in V:
  cost[u]=∞
  prev[u]=nil
  done[u]= FALSE // =1 if in MST
s= any vertex in V
cost[s]=0
H= makePQ(V)
while (H ≠ ∅):
  u= deleteMin(H)
  done[u]= TRUE   // add u to MST
  for each v adjacent to u:
    if (!done[v]
        && cost[v]> w(u,v)):
      cost[v]= w(u,v)   // ↓ in H
      prev[v]= u</pre>
	<pre>for each u in V:
	  dist[u]=∞
	  prev[u]=nil
	  done[u]= FALSE // =1 if shortest path found
	
	dist[s]=0
	H= makePQ(V)
	while (H ≠ ∅):
	  u= deleteMin(H)
	  done[u]= TRUE // shortest path to u found
	  for each v adjacent to u:
	    if (!done[v]
	        && dist[v]> dist[u]+w(u,v)):
	      dist[v]= dist[u]+w(u,v) // ↓ in H
	      prev[v]= u</pre>

Complexity of Prim's: same as Dijkstra's, O((E+V) log V)

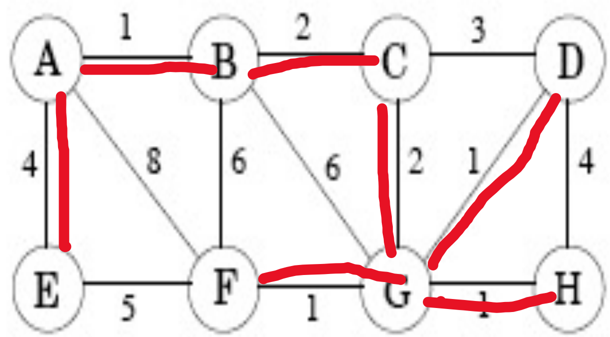
Example

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).

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								

Kruskal's algorithm

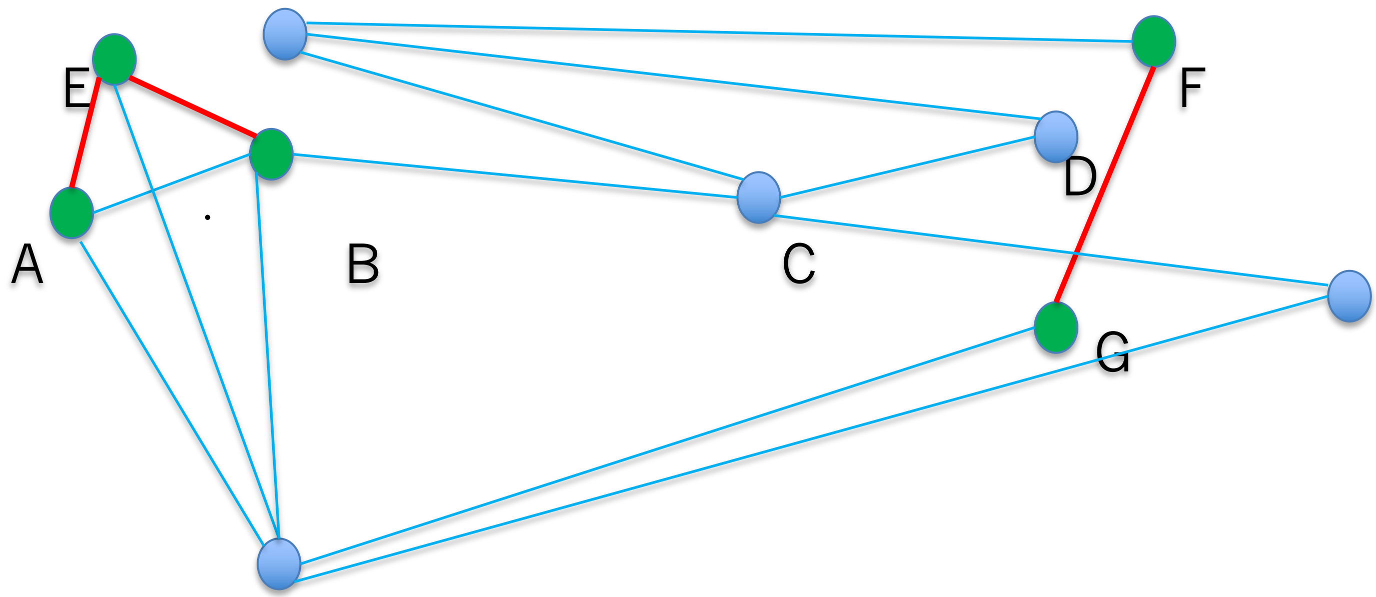
Purpose: Find MST of G= (V,E,w)

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

Kruskal’s algorithm: operates edge-by-edge.

```
set MST-so-far to empty
for each (u,v), in increasing order of weight:
    if  ( (u,v) does not form a cycle in MST-so-far):
         add edge (u,v) to MST
```

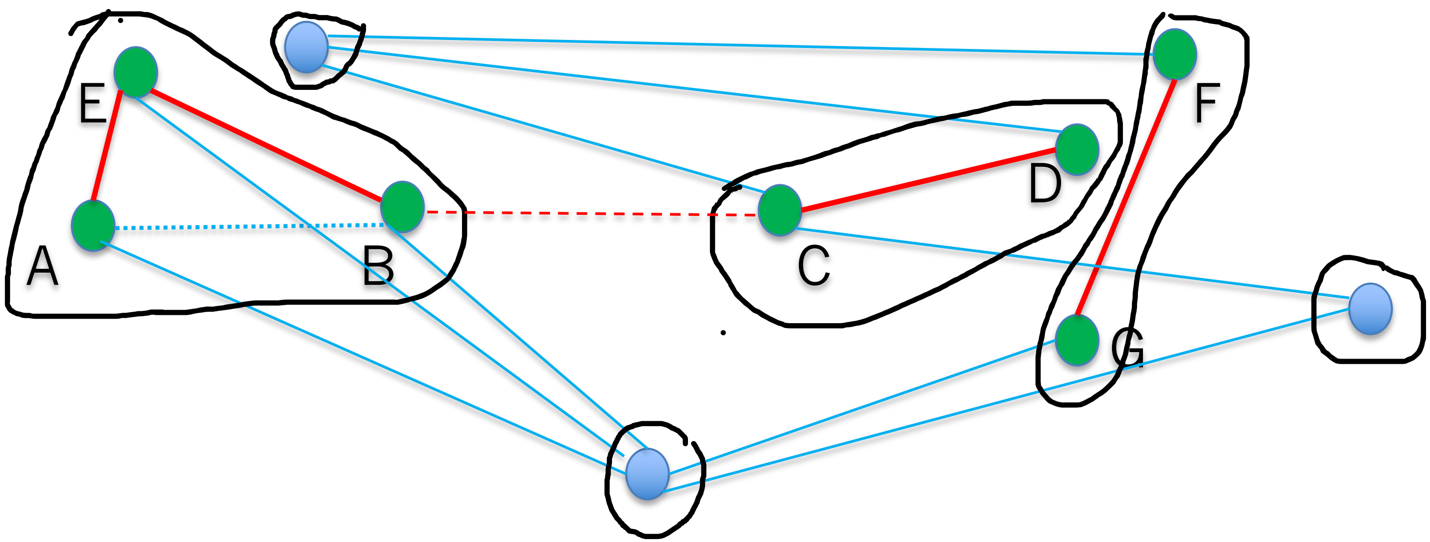
How do we implement?



Suppose that the above 5 green vertices and 3 red edges are in our MST-so-far. The next lightest are AB (should be rejected), then CD.

How can we recognize that inclusion of AB would create a cycle in the MST-so-far?

Using disjoit sets



How can we recognize that inclusion of AB would create a cycle in the MST-so-far?

Think about disjoint sets:

- After adding (C,D) to MST, we have 6 disjoint sets.
- After adding (B,C) the sets ABE and CD are joined into ABCDE → we will have 5 disjoint sets

Needed:

- an ID for each set ?
- Operator Find(u)   : find the set the set a node u belongs to
- Operator Union(u,v) : join the disjoin sets of u and v into a single set

Kruskal's algorithm

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

Implement step 3-5 using disjoint set:

- Before the loop: Make |V| disjoint subsets, each contains a single node of V.
- In the loop body:
  - u and v not belong to a same set ⇔ (u,v) does not form a cycle in the MST.
- Step 3: The algorithm stops when we have V-1 edges in MST

Operations:

`makeset(u)`- return a tree that contains single `u`

`find(u)` – return the root (means, ID) of the tree that contains `u`;

`union(u,v)` – joins trees containing `u` and `v`.

Kruskal's algorithm

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

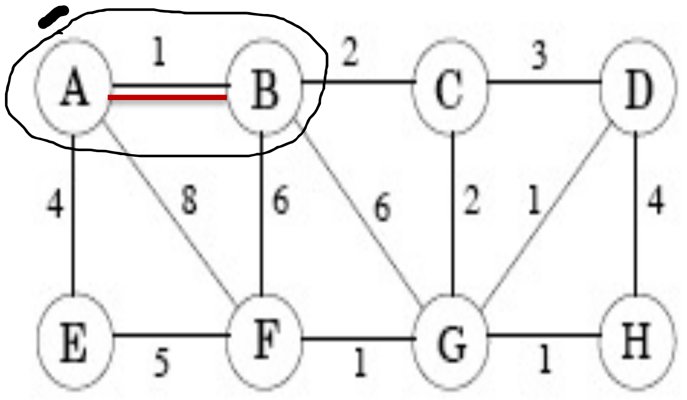
```
0  set X= empty. X is the MST-so-far
1  E1 = E, but sorted in increasing order of weights
2  for each u: makeset(u) (build single-element set {u})
3  while ( |X| < |V|-1 ) :
3a   set (u,v)= the next edge in E1
4    if  (find(u) ≠ find(v) ):   // u & v do not belong to a same set
5       add edge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

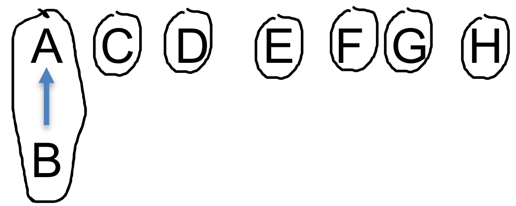
Watch online lecture for how to actually implement `makeset(u)`, `find(u)`, `union(u,v)`

→ We can implement with O(V+ElogV) or even O(V+E) for steps 2-6

→ The cost of KA is dominated by ElogE of the sorting phase in step 0

Example (Kruskal's)

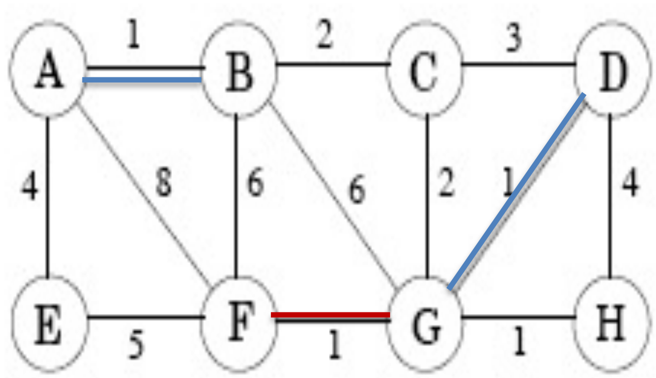


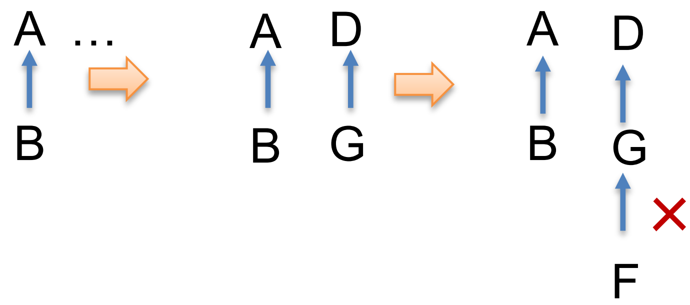


Edges in MST	A	B	C	D	E	F	G	H
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7

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

Example (Kruskal's)





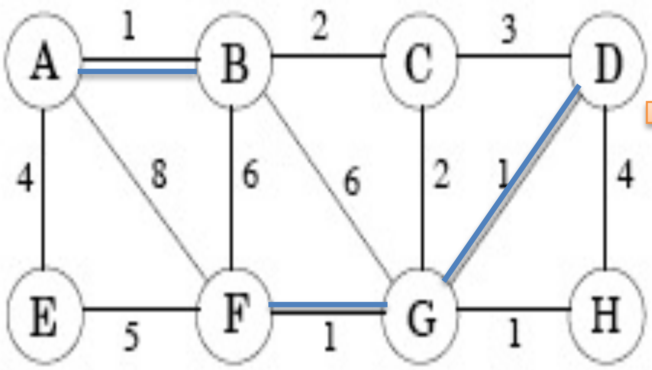
Edges in MST	A	B	C	D	E	F	G	H
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	**3**	4	**?**	3	7

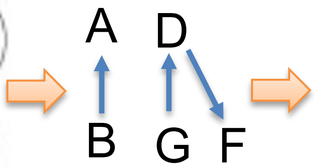
Note: number in table represents tree ID, if a tree ID is the same as node ID, then the node is the root of its tree

Rules:

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

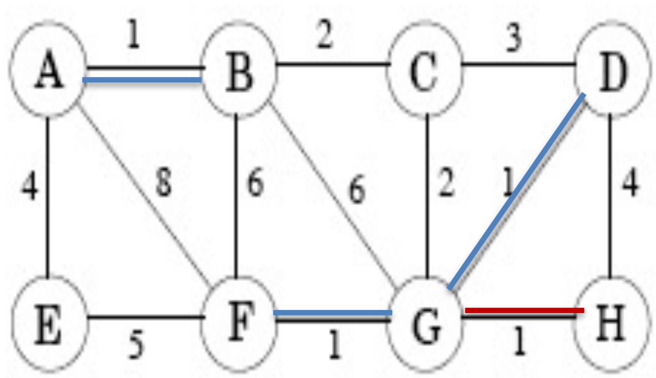
2) when joins, joins to the root, ie. id[F]= id[G]

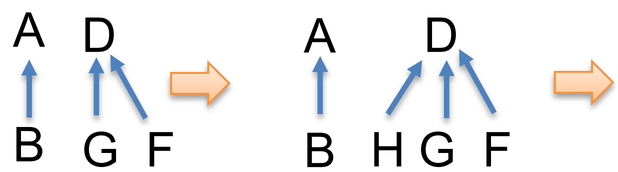




Edges in MST	A	B	C	D	E	F	G	H
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	3	4	6?	3	7
G---F	0	0	2	3	4	**3**	3	7

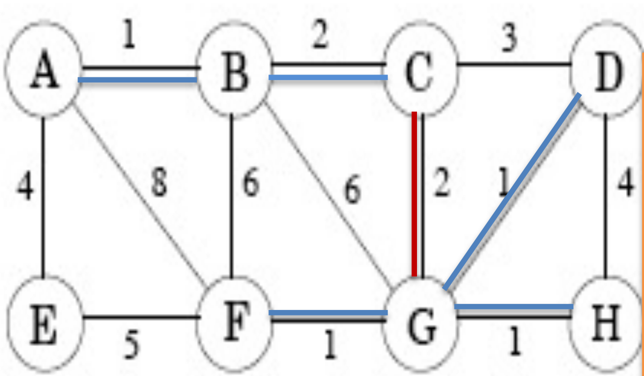
Rules:
1) join smaller tree to bigger tree (*weighted*, or *join-by-rank* optimization)
2) when joins, joins to the root



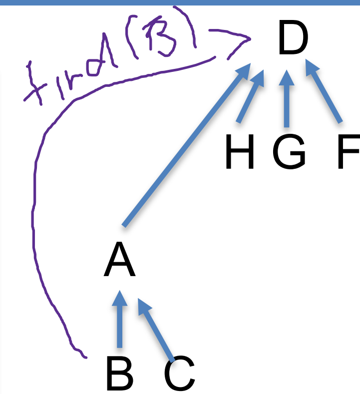


Edges in MST	A	B	C	D	E	F	G	H
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	3	4	**3**	3	7
G---H	0	0	2	3	4	3	3	3

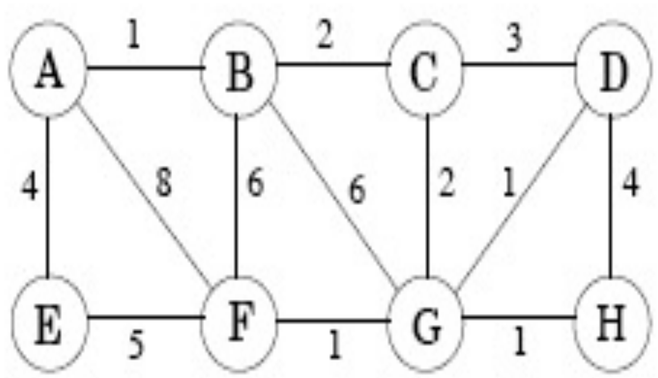
Rules:
3) Path compression



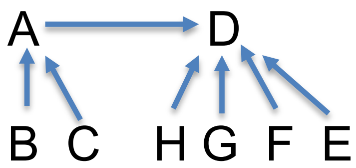
- Now, the pathlen of B,C is >1
- In the future, if we call find(B) we will have find(B)= D, and at that time we will make B point directly to D. That is called *path compression*.



Edges in MST	A	B	C	D	E	F	G	H
	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



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 at this stage.



Edges in MST	A	B	C	D	E	F	G	H
	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

Notes

In the lecture, weighted and path compression was mentioned, but probably without much of details.

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

- find: O(1)
- union: time for tracing depends on the depth of the tree
  - naïve: O(V)
  - weighted: O(log V)
  - weighted + path compression: O(1)

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.

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		

Lab Time:

Learning experience from the semester?

LAB:

- finish ass2, or
- go through some questions, especially short questions, in past-exam papers,
- give questions to the in-lazy-mode Anh

Good luck!