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`

Question of the Year: Do you still need time for assignment 3 implementation ? ☺

Greedy Algorithms and Application to Graphs

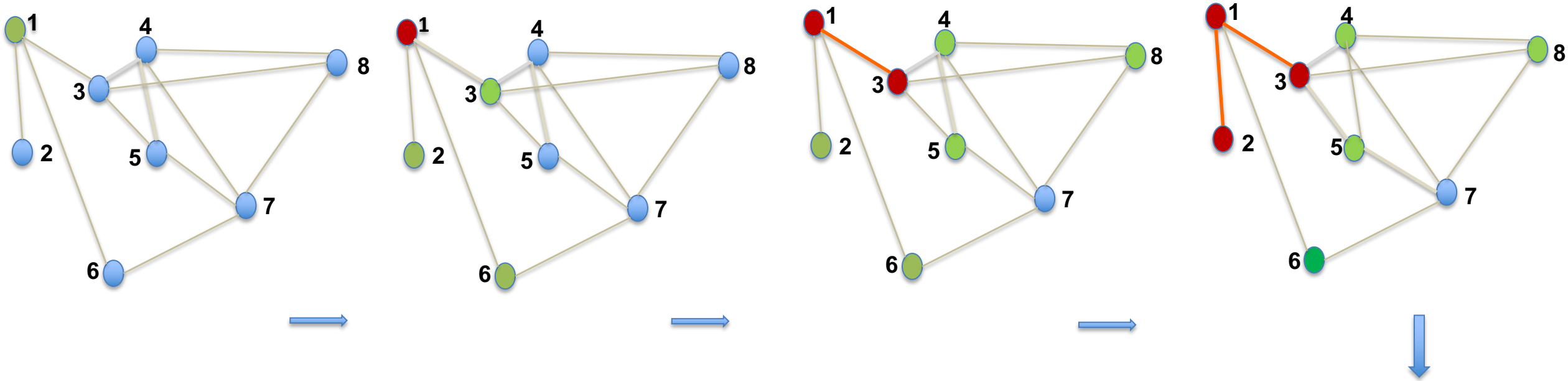
Greedy algorithms:

- A myopic policy of always taking the “best” bite in each step based on limited information.
- NOT always works…
- In many cases it works, and it’s the best policy!

Examples

- Dijkstra’s algorithm is greedy.
- Other greedy algorithms: Prim’s, Kruskal’s for finding MST

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 :-?

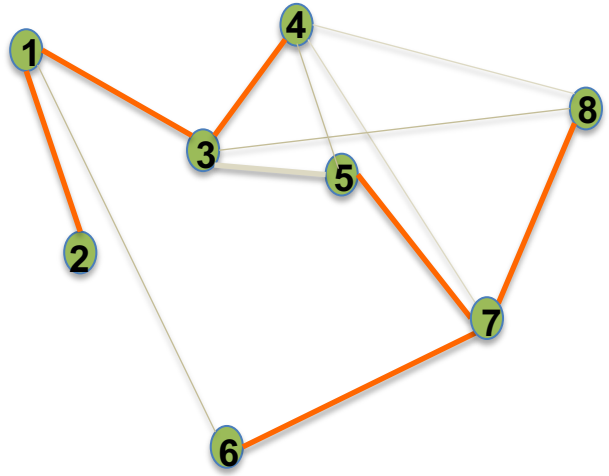
MST - overview

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

- What's a *spanning tree*? How many vertices and edges does it have?
- 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 length of an edge represent its weight. In particular, edges (3,4), (4,5) and (5,3) have the same weight. The green nodes and red edges form one MST.

Two Greedy Algorithms for finding MST

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

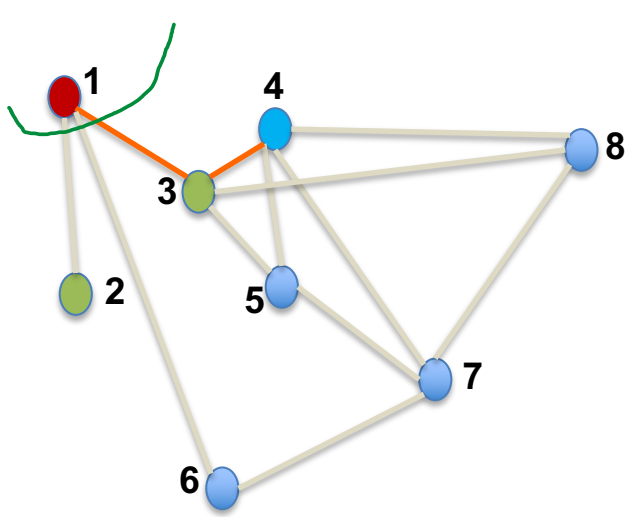
Prim's Algorithm: build MST by selecting a vertex at a time (using graph traversal)	**Kruskal's Algorithm: build MST by selecting an edge at a time (not using graph traversal)**

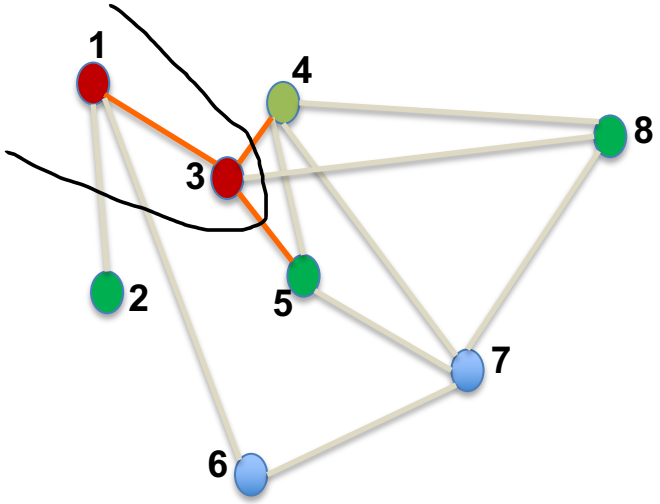
`T= any vertex`
`while ( |T| < |V|):`
`  add to T the vertex that`
`  has least distance to T`
 	
Note: distance between node u and set T is defined as the minimal distance between u and any member of T
	`T= EMPTY SET OF edges`
	`while ( |T| < |V|-1):`
	`  add to T the lightest edge`
	`  that doesn't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

Sounds familiar 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).*

Prim's algorithm: operates vertex-by-vertex

Given a (connected) weighted graph G

Prim(G): Find a MST of G

```
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
```

Dijkstra(G,s): find shortest paths from s

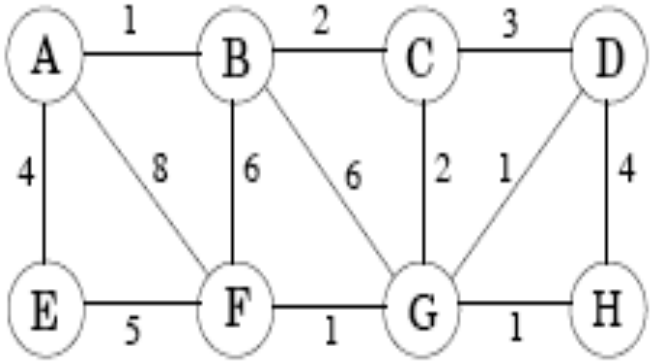
```
for each u in V:
  dist[u]=∞
  prev[u]=nil
  done[u]= FALSE  // =1 if shortest path found

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
```

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.

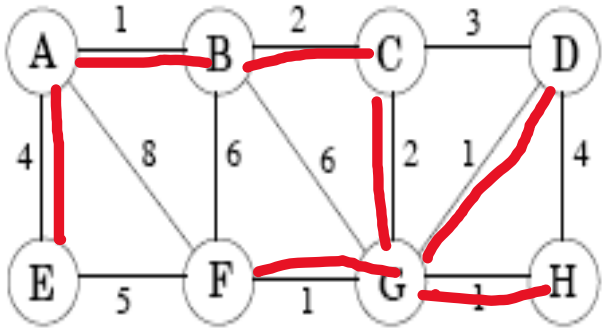


(a) Run Prim's algorithm; whenever there is a choice of nodes, always use alphabetic ordering (e.g., start from node *A*). Draw a table showing the intermediate values of the `cost` array.

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

note: in the table, - is my shorthand for ∞,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								

Kruskal's algorithm [not using graph traversal!]

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

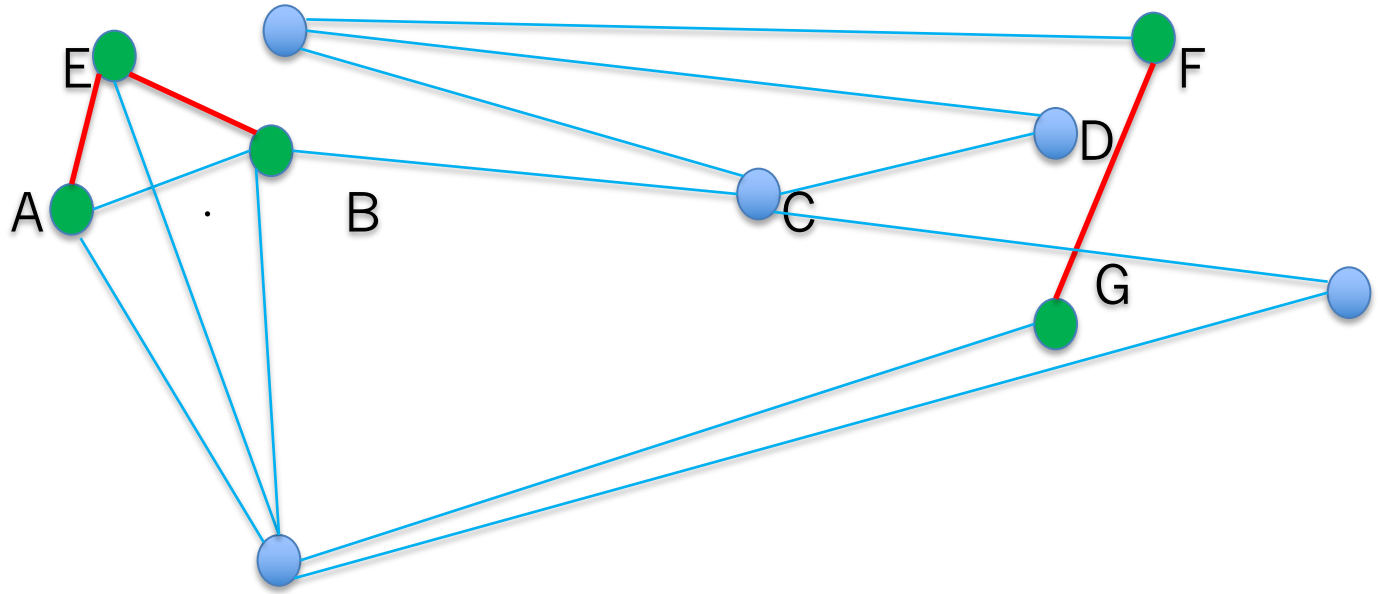
Prim's algorithm: processing node-by-node, ie. adding a new node to MST at each step.

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

```
0 set MST-so-far to empty
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
```

- Easy to run by hand. But how do we implement?

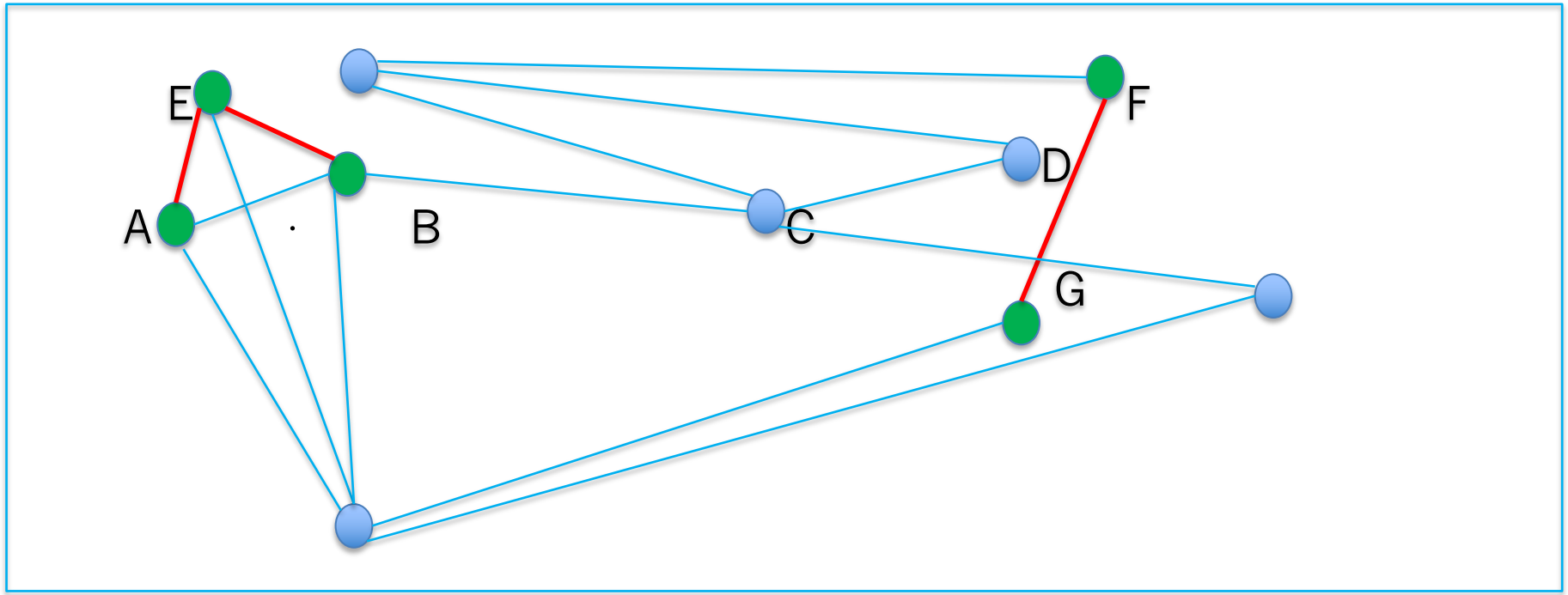
Easy to run manually – but how to implement?



Suppose that the above 5 green vertices and 3 red edges are in our MST-so-far.

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

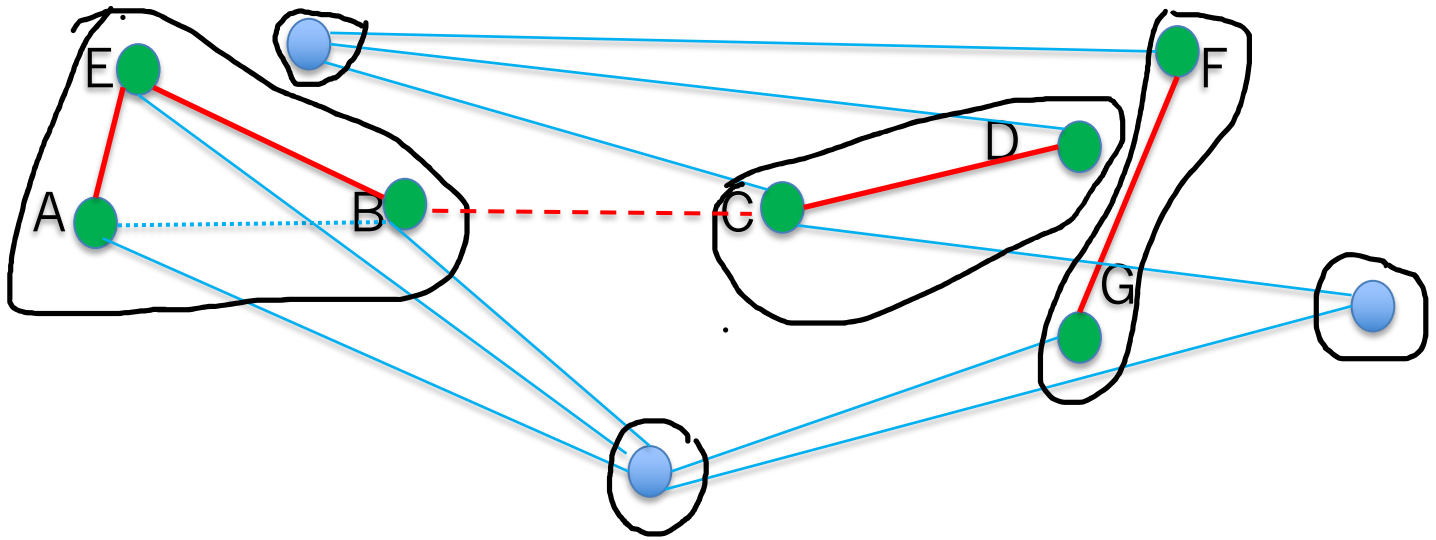
but how to implement?



At any time, we consider the graphs
(V, E= already-included-into-MST edges),
and define each connected component as a set. The graph is a collection of disjoint sets. Then:

edge `(x,y)` forms a cycle in MST-so-far ⇔ `x` and `y` belong to a same disjoint set

Using disjoint sets



Think about disjoint sets:

- At the start, each node is a singleton disjoin set
- When adding an edge, say (C,D), to MST, we join together the sets containing C and D
- 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

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

Notes

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)
  - optimization: using join-by-rank (weighted) O(log V)
  - optimization: using join-by-rank + 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 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 Activities – W12.10

Q1: MST is a concept on connected graphs, and the Prim's algorithm can only be applied to weighted connected graphs. Can we modify the Prim's algorithm for building *minimum spanning forest* (ie. set of MSTs) for any weighted graphs?

Extension: How about the Kruskal's algorithm?

In connected weighted graphs, the *fatness* of a path u→v is defined as the maximum weight of all the edges in this path.

Q2: From the provided data structure, which one is the best (ie. easiest input) for the task of finding minimum fatness path between any pair of vertices?

Q3: Given a MST for a (connected weighted) graph. From the 2 provided algorithms, which one is the most (time-)efficient for the task of finding (minimum fatness) paths between all pair of vertices ?

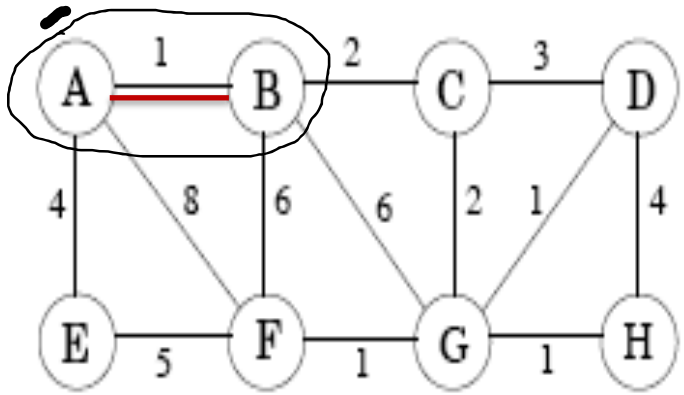
Lab Time:

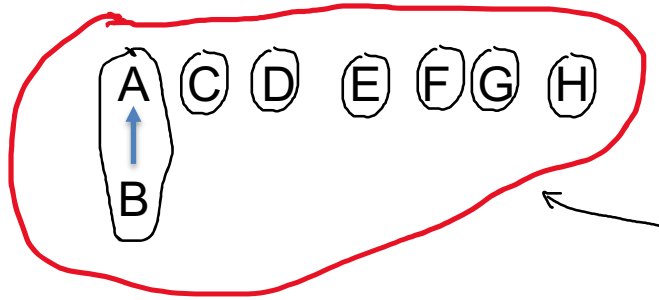
- W12.10 (Peer Activities)
- finish ass3, or
- go through some questions, especially short questions, in past-exam papers,
- give questions to the in-lazy-mode Anh

Good luck!

Additional Slides

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



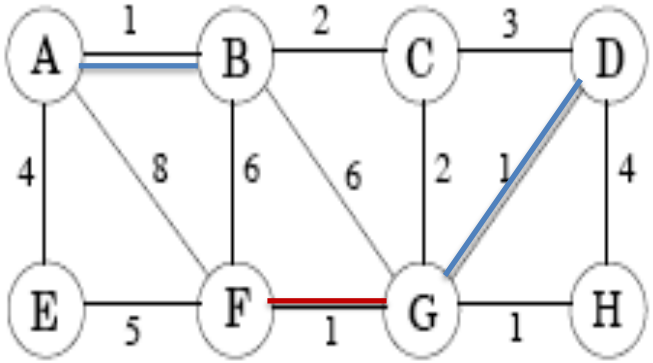


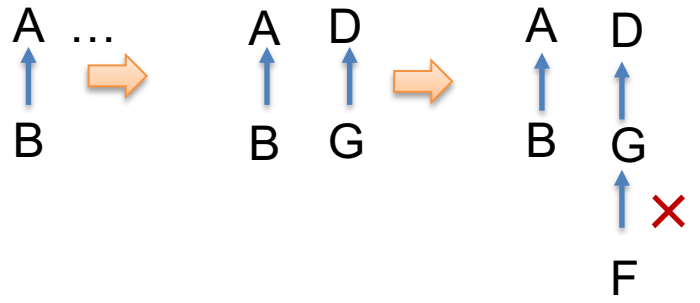
the MST-so-far, which is a set of tree, each tree is a disjoint set.

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

Example (Kruskal's)



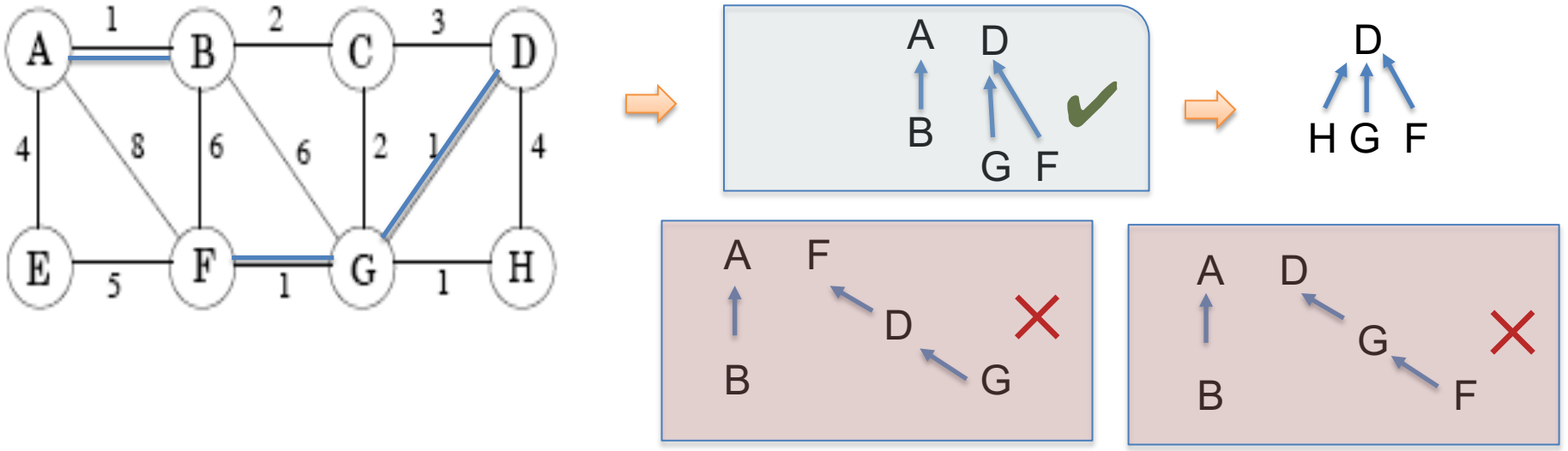


Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	**3**	4	**?**	3	7

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

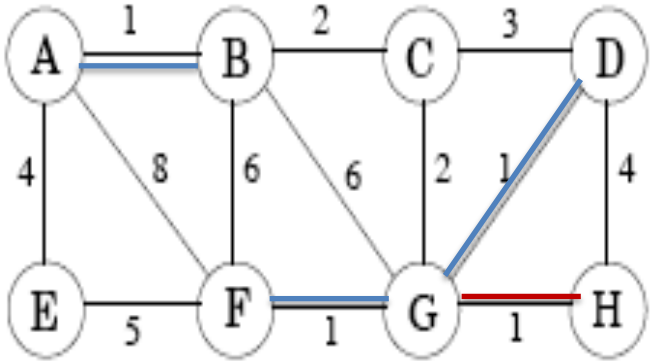
Note: to be able to decide which tree is bigger, in the table for each root node we store the number of nodes, say k, as –k.
As the beginning, all nodes have value -1.
Then, a negative value means a root, a non-negative value shows the reference to root.

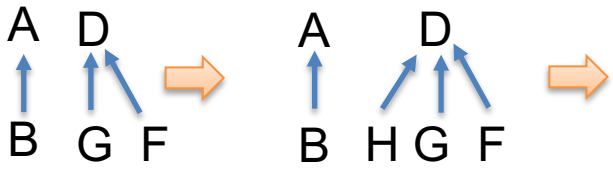
See live lecture for details.

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

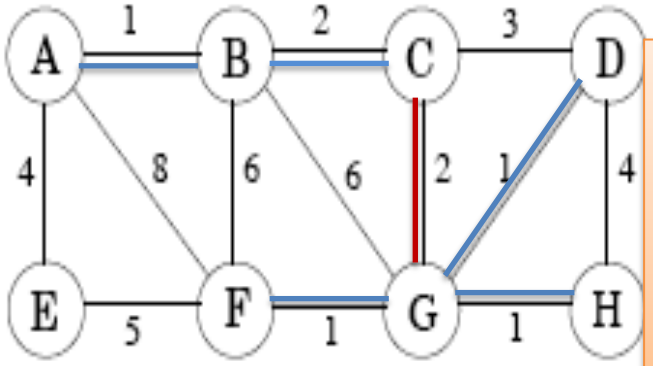




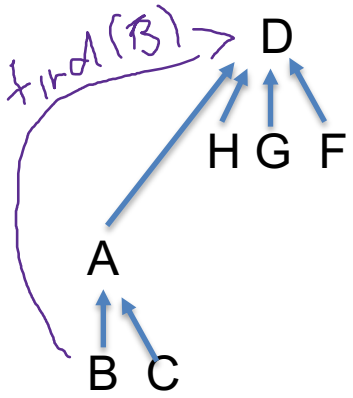
Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	0	2	3	4	5	6	7
D---G	0	0	2	3	4	5	3	7
G---F	0	0	2	3	4	**3**	3	7
G---H	0	0	2	3	4	3	3	3

Optimisation 2: Path compression

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

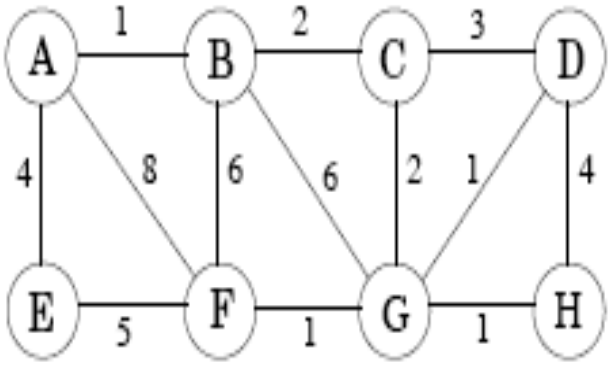


- Now, the pathlen of B,C is >1
- In the future, if we call find(B) we will have find(B)= D, and at that time we will make B point directly to D. That is called *path compression*.

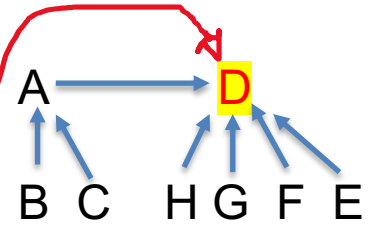


Edges in MST	A 0	B 1	C 2	D 3	E 4	F 5	G 6	H 7
	0	1	2	3	4	5	6	7
A---B	0	**0**	2	3	4	5	6	7
D---G	0	0	2	3	4	5	**3**	7
G---F	0	0	2	3	4	**3**	3	7
G---H	0	0	2	3	4	3	3	**3**
B---C	0	0	**0**	3	4	3	3	3
C--G	**3**	0	0	3	4	3	3	3

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

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.