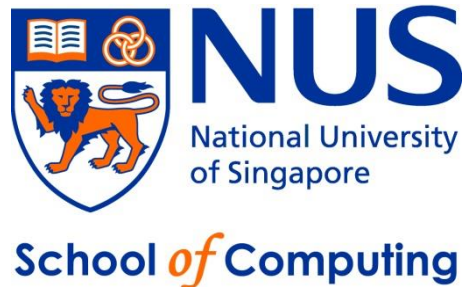


CS2040 – Data Structures and Algorithms

Lecture 14 – Connecting People - MST

chongket@comp.nus.edu.sg



Outline

Minimum Spanning Tree (MST)

- Motivating Example & Some Definitions

Two Algorithms to solve MST (you have a choice!)

- Prim's (greedy algorithm with PriorityQueue)
 - PriorityQueue is discussed in Lecture 08
- Kruskal's (greedy algorithm, uses sorting and UFDS)
 - UFDS is discussed in Lecture 09

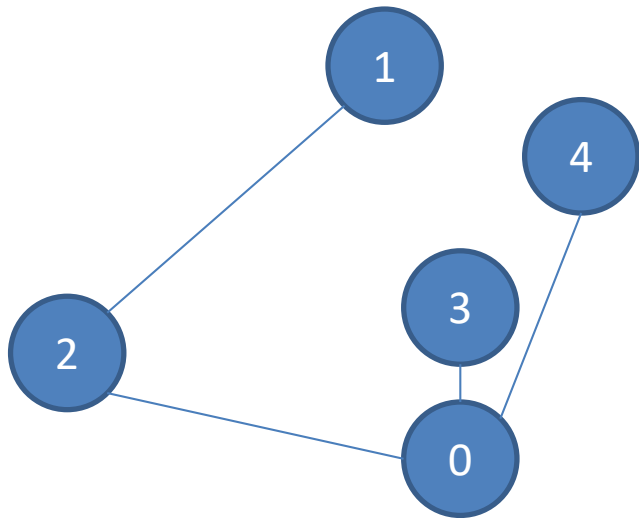
Review

Definitions that we have learned before

- **Tree T**
 - T is a **connected graph** that has V vertices and $V-1$ edges
 - Important: One unique path between any two pair of vertices in T
- **Spanning Tree ST of connected graph G**
 - ST is a tree that spans (covers) every vertex in G
 - Recall the **BFS and DFS Spanning Tree**

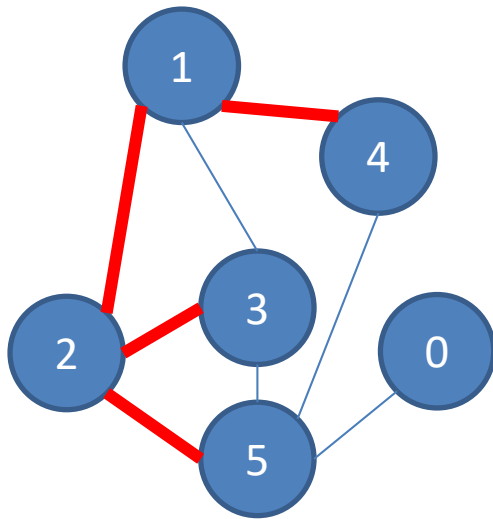
Is This A Tree?

1. Yes, why _____
2. No, why _____



Do the edges highlighted in **red** part form a spanning tree of the original graph?

1. Yes, why _____
2. No, why _____



Motivating Example

Government Project

- Want to link rural villages with roads
- The cost to build a road depends on the terrain, etc
- You only have limited budget
- How are you going to build the roads?



Definitions (1)

- Vertex set **V** (e.g. street intersections, houses, etc)
- Edge set **E** (e.g. streets, roads, avenues, etc)
 - Generally undirected (e.g. bidirectional road, etc)
 - Weighted (e.g. distance, time, toll, etc)
- Weight function **$w(a, b): E \rightarrow R$**
 - Sets the weight of edge from **a** to **b**
- **Weighted Graph G: $G(V, E), w(a, b): E \rightarrow R$**
- **Connected** undirected graph **G**
 - There is a path from any vertex **a** to any other vertex **b** in **G**
- The graph **G** we're concerned with is **connected** **undirected** and **weighted** when dealing with **MST**

More Definitions (2)

- Spanning Tree **ST** of connected undirected weighted graph **G**
 - Let **w(ST)**, weight of **ST**, denotes the total weight of edges in **ST**

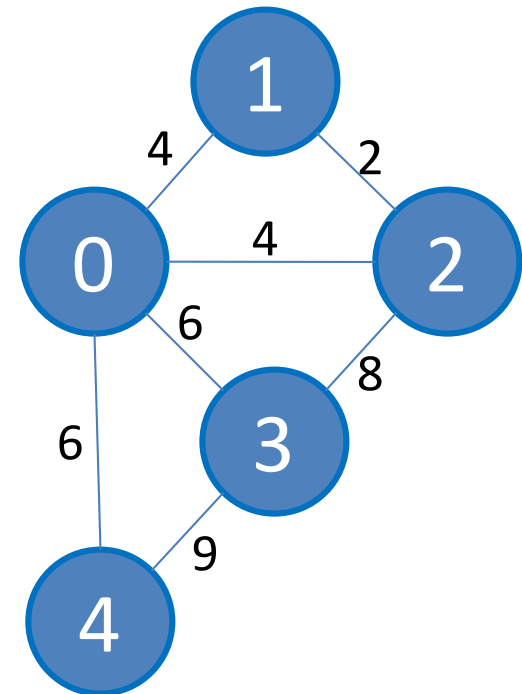
$$w(ST) = \sum_{(a,b) \in ST} w(a,b)$$

- **Minimum Spanning Tree (MST)** of connected undirected weighted graph **G**
 - **MST** of **G** is an **ST** of **G** with the minimum possible **w(ST)**

More Definitions (3)

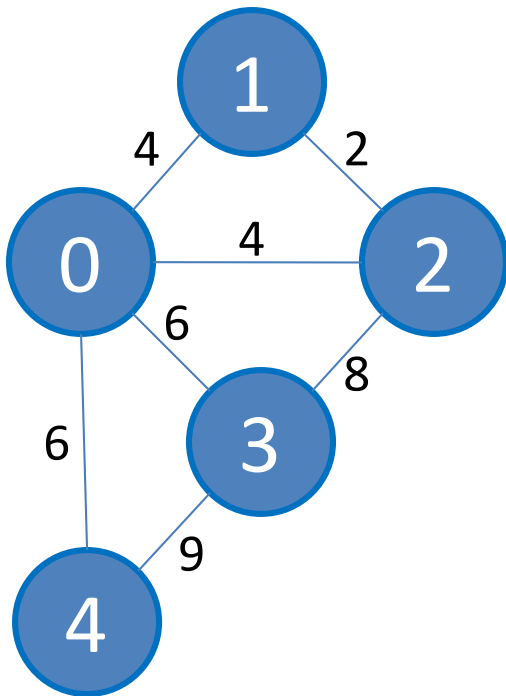
- **The (standard) MST Problem**

- Input: A connected undirected weighted graph $\mathbf{G(V, E)}$
- Select some edges of \mathbf{G} such that the graph forms a spanning tree, but with minimum total weight
- Output: Minimum Spanning Tree (**MST**) of \mathbf{G}

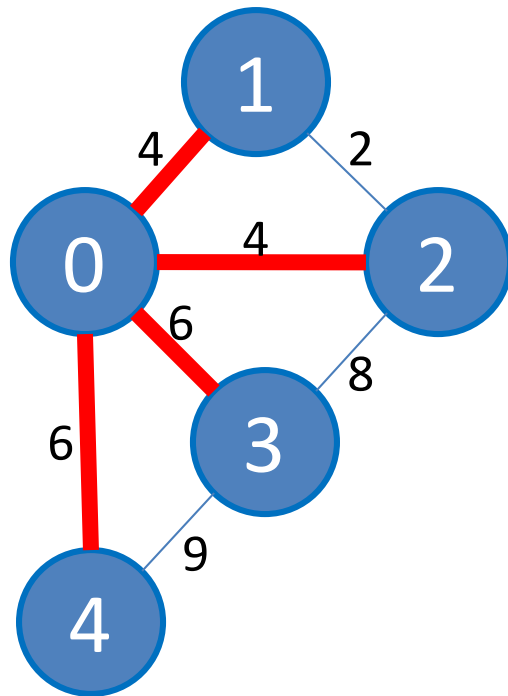


Example

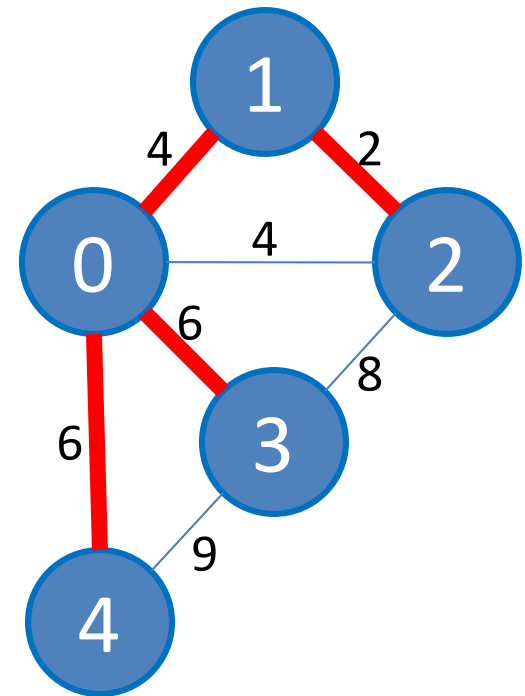
The Original Graph



A Spanning Tree
Cost: $4+4+6+6 = 20$

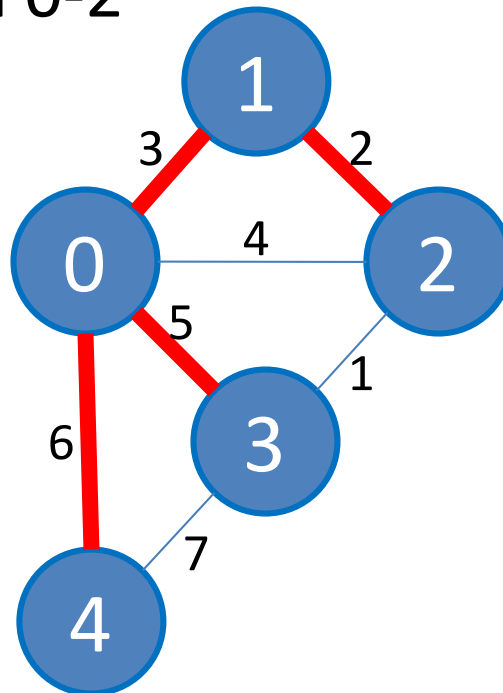


An MST
Cost: $4+6+6+2 = 18$



Do the edges highlighted in **red** part form an MST of the original graph?

1. No, we must replace edge 0-3 with edge 2-3
2. No, we must replace edge 1-2 with 0-2
3. Yes



Brute force/Complete Search Solution?

- Consider all cycles in the graph and break them!
 - For each cycle remove the largest edge
 - If 1 or more edges in a cycle has already been removed previously move on to the next cycle
- Cycle property: For any cycle C in graph $G(V,E)$, if weight of an edge e is larger than every other edge in C , e cannot be included in the MST of $G(V,E)$
- How to get all cycles in the graph?
 - Not so easy except for some special graphs ... (Can you think of a way to do this?)
 - Can have up to $O(V!)$ different cycles!
 - Listing down one by one is slow !

MST Algorithms

MST is a well-known Computer Science problem

Several efficient (polynomial) algorithms:

- Jarnik's/Prim's greedy algorithm
 - Uses PriorityQueue Data Structure taught in Lecture 09
- Kruskal's greedy algorithm
 - Uses Union-Find Data Structure taught in Lecture 10
- Boruvka's greedy algorithm (not discussed here)
- And a few more advanced variants/special cases...

Do you still remember Prim's/Kruskal's algorithms from CS1231/S?

1. Yes and I also know how to ***implement*** them
2. Yes, but I have not try implementing them yet
3. I forgot that particular CS1231/S material... but I know it exists
4. Eh?? These two algorithms were covered before in CS1231/S??
5. I didn't take CS1231/S ☹️

Prim's Algorithm

Very simple pseudo code

1. $T \leftarrow \{s\}$, a starting vertex s (usually vertex 0)
2. enqueue edges connected to s (*only the other ending vertex and edge weight*) into a priority queue PQ that orders elements based on increasing weight
3. **while** there are edges left in PQ
 - take out the front most edge e
 - if** vertex v linked with this edge e is not yet in T
 - $T \leftarrow T \cup v$ (including this edge e)
 - enqueue each edge adjacent to v into the PQ if the other ending vertex of that edge is not already in T
4. T is an MST

MST Algorithm: Prim's

Ask VisuAlgo to perform Prim's from various sources on the sample Graph (CP3 4.10), then try other graphs

In the screen shot below, we show the start of **Prim(0)**

The screenshot shows the VisuAlgo interface for Minimum Spanning Tree. The top bar displays "VISUALGO" and "MINIMUM SPANNING TREE" on the left, and "Exploration Mode" on the right. The main area shows a graph with 5 nodes (0, 1, 2, 3, 4) and weighted edges. Node 0 is highlighted in green. The edges and their weights are: (0,1) weight 4, (0,2) weight 4, (0,3) weight 6, (0,4) weight 9, (1,2) weight 2, (2,3) weight 8, and (3,4) weight 6. A blue arrow points from the text "then try other graphs" to the "Sample Graphs" option in the left sidebar.

On the left sidebar, the options are: Draw Graph, Random Graph, Sample Graphs, Kruskal's Algo, and Prim's Algo. At the bottom of the sidebar, there is a "GO" button.

On the right, the title "Prim's Algorithm, starting from 0" is displayed. Below it, a green box contains the text: "Add (4,1), (4,2), (6,3), (6,4) to the PQ. The PQ is now (4,1), (4,2), (6,3), (6,4)." Below this, a blue box contains the following pseudocode:

```
T = {s}
enqueue edges connected to s in PQ by weight
while (!PQ.isEmpty)
    if (vertex v linked with e=PQ.remove is not in T)
        T = T U v, enqueue edges connected to v
    else ignore e
T is an MST
```


Easy Java Implementation

You need to use two known Data Structures to be able to implement Prim's algorithm:

1. A priority queue **PQ** (we can use Java PriorityQueue), and
2. A boolean array **taken** (to decide if a vertex has been taken or not)

With these DSes, we can run Prim's in $O(E \log V)$ using Adjacency list

- We only process each edge once (enqueue and dequeue it), $O(E)$
 - Each time, we enqueue/dequeue from a PQ in $O(\log E)$
 - As $E = O(V^2)$, we have $O(\log E) = O(\log V^2) = O(2 \log V) = O(\log V)$
 - Total time $O(E) * O(\log V) = O(E \log V)$

Let's have a quick look at PrimDemo.java

Why Does Prim's Work? (1)

First, we have to realize that **Prim's algorithm** is a **greedy algorithm**

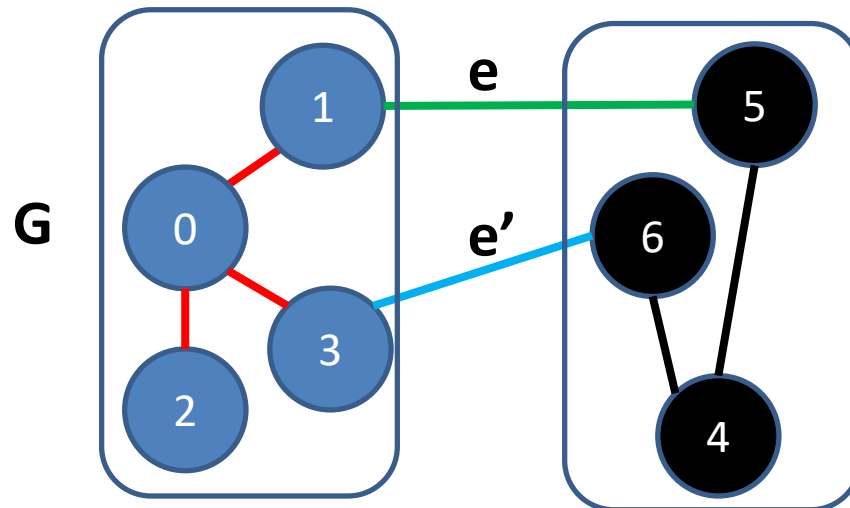
This is because **at each step**, it always try to select the next valid edge e with **minimal weight** (greedy!)

Greedy algorithm is usually simple to implement

- However, it usually requires “proof of correctness”
- Here, we will just see a quick proof

Cut Property of a Connected Graph G

- **Cut of a connected graph:** any partition of vertices of G into 2 disjoint subset (vertices in one set is not in the other). An example is shown below.
- **Cut Set:** The set of edges that cross a cut (e and e' in the example)
- **Cut Property of a connected graph:** For any cut of the graph, if the weight of an edge e in the cut-set is strictly smaller than the weights of other edges of the Cut Set, then e belongs to all MSTs of the graph.

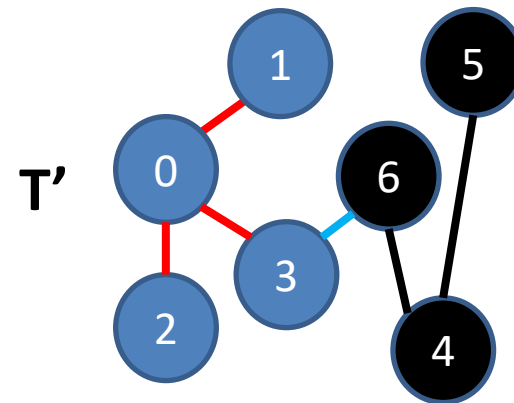
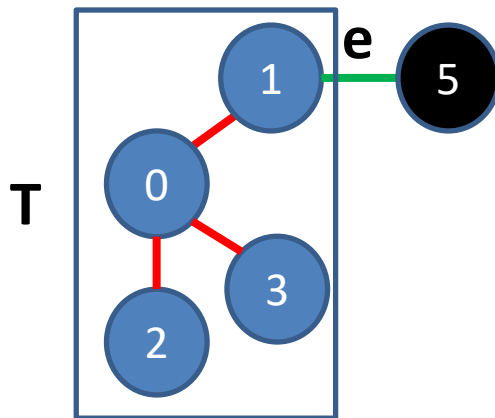


Why Does Prim's Work? (2)

with visual explanation

Proof by contradiction:

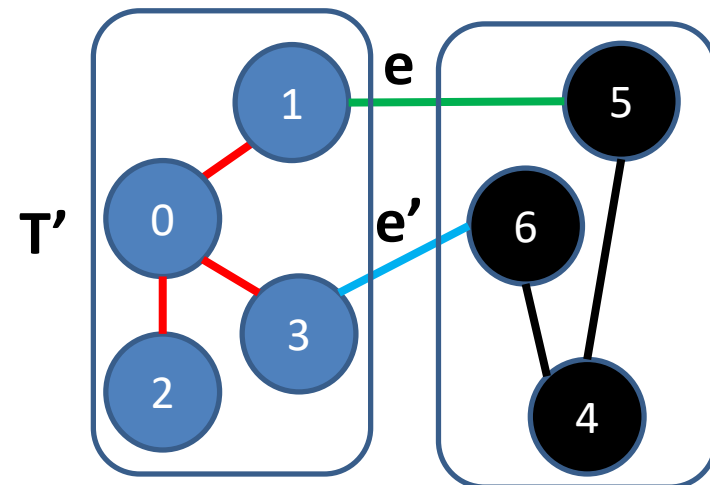
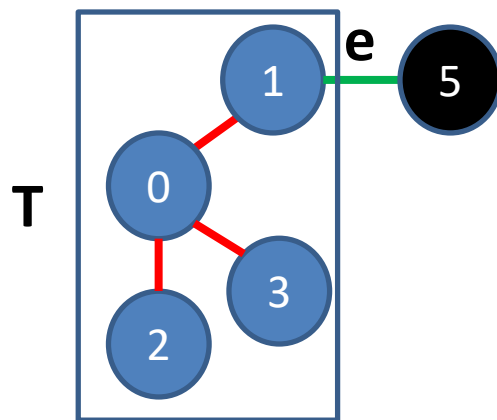
- Assume that edge e is the first edge at iteration k chosen by Prim's which is not in any valid MST.
- Let T be the tree generated by Prim's before adding e .
- Now T must be a subtree of some valid MST T'



Why Does Prim's Work? (3)

with visual explanation

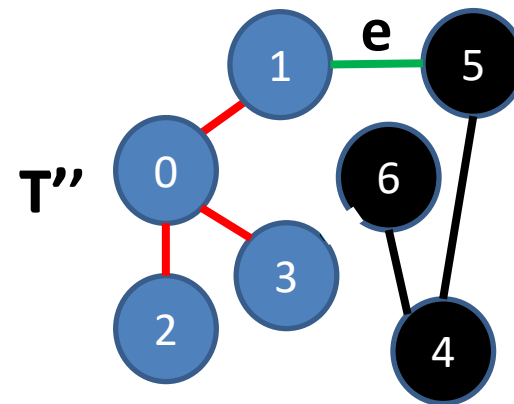
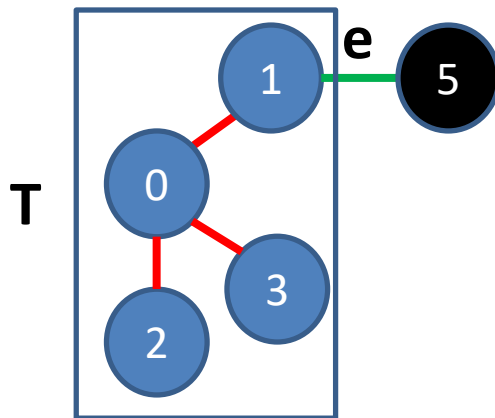
- Adding edge e to T' will now create a cycle.
- Since e has 1 endpoint in T (the valid endpoint) and one endpoint outside T , trace around this cycle in T' until we get to some edge e' that goes back to T
- Removing e and e' will disconnect T' into 2 components ($\{0,1,2,3\}$ which is T and $\{4,5,6\}$ in the example). This is a cut of T' , where $\{e, e'\}$ is the cut set as illustrated



Why Does Prim's Work? (4)

with visual explanation

- By Prim's algorithm **e** and **e'** must be candidate edges at iteration k, but **e** was chosen meaning $w(\mathbf{e}) \leq w(\mathbf{e}')$ by the **cut property**
- Now replacing **e'** with **e** in **T'** must give us tree **T''** covering all vertices of the graph s.t $w(\mathbf{T}'') \leq w(\mathbf{T}')$
- Contradiction that **e** is first edge chosen wrongly



*Prim's variant for Dense Graphs (1)

- For dense graphs where $E = O(V^2)$, time complexity of Prim's is $O(E \log V) = \mathbf{O(V^2 \log V)}$
- We can improve this time complexity ironically by replacing the PQ with a simple array **A** of size V
 - For each $A[v]$, store a pair info $\langle w(u,v), u \rangle$ of the smallest weighted edge (u,v) to v among all edges to v that has been explored as Prim's is executed
 - *Why only track the smallest weighted explored edge to a vertex v ?*

**this variant is not in Visualgo*

Prim's variant for Dense Graphs (2)

Pseudo code is as follows

1. Initialize $A[v] = \langle +\infty, v \rangle$ for all v
2. $A[s] = \langle 0, s \rangle$
3. While not all vertices are in T
 - Scan A to get v where $A[v].\text{first}$ is minimum in A
 - $T \leftarrow T \cup v$ and the adjoining edge
 - $A[v].\text{first} = +\infty$ // will not choose it again
 - for all u adjacent to v // all neighbors of v
 - if (u is not in T && $A[u].\text{first} > w(v, u)$)
 - $A[u] = \langle w(v, u), v \rangle$
4. T is an MST

Prim's variant for Dense Graphs (3)

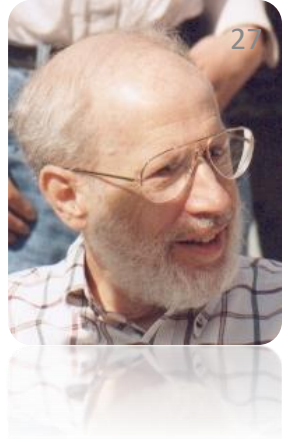
Time complexity of Prim's variant for dense graphs

1. Initialize $A[v] = \langle +\infty, v \rangle$ for all v
2. $A[s] = \langle 0, s \rangle$
3. While not all vertices are in T // $O(V)$ iterations
 - scan A to get v where $A[v].\text{first}$ is minimum in A // $O(V)$
 - $T \leftarrow T \cup v$ and the adjoining edge
 - $A[v].\text{first} = +\infty$
 - for all u adjacent to v // $O(V)$ due to $O(V)$ neighbors
 - if (u is not in T && $A[u].\text{first} > w(v, u)$)
 - $A[u] = \langle w(v, u), v \rangle$
4. T is an MST

Total Time Complexity = $O(V^2)$, better than $O(V^2 \log V)$ for standard Prim's algorithm

Coming up next: Kruskal's algorithm

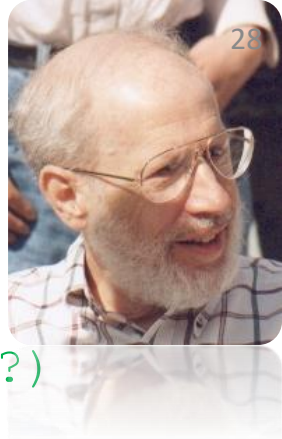
Kruskal's Algorithm



Very simple pseudo code

```
sort the set of E edges by increasing weight  
T  $\leftarrow$  {}  
while there are unprocessed edges left  
    pick an unprocessed edge e with min cost  
    if adding e to T does not form a cycle  
        add e to T  
T is an MST
```

Kruskal's Implementation (1)



```
sort the set of E edges by increasing weight //  $O(?)$   
T  $\leftarrow$  {}  
while there are unprocessed edges left //  $O(E)$   
    pick an unprocessed edge e with min cost //  $O(?)$   
    if adding e to T does not form a cycle //  $O(?)$   
        add e to the T //  $O(1)$   
T is an MST
```

To sort the edges:

- We use **EdgeList** to store graph information
- Then use “any” sorting algorithm that we have seen before

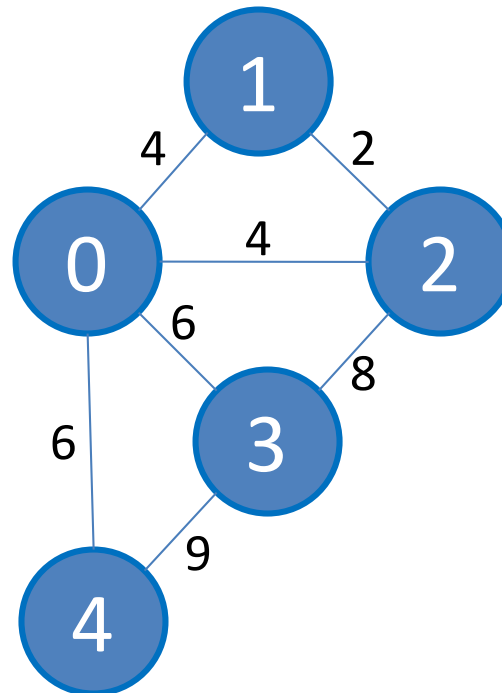
To test for cycles:

- We use **Union-Find Disjoint Sets**

Sorting Edges in Edge List

Adjacency Matrix/List that we have learned previously are *not suitable* for edge-sorting task!

To sort **EdgeList**, we use *one liner* Java **Collections.sort** ...



i	w	u	v
0	2	1	2
1	4	0	1
2	4	0	2
3	6	0	3
4	6	0	4
5	8	2	3
6	9	3	4

MST Algorithm: Kruskal's

Ask VisuAlgo to perform Kruskal's on the sample Graph (CP3 4.10), then try other graphs

In the screen shot below, we show the start of **Kruskal**
(there is no parameter for this algorithm)

The screenshot displays the VisuAlgo interface for the Minimum Spanning Tree (MST) algorithm. The top bar shows the VisuAlgo logo and the title "MINIMUM SPANNING TREE". The right side of the bar indicates "Exploration Mode".

On the left, a sidebar menu lists the following options: "Draw Graph", "Random Graph", "Sample Graphs", "Kruskal's Algo", and "Prim's Algo". A blue arrow points from the text "then try other graphs" in the preceding paragraph to the "Sample Graphs" option.

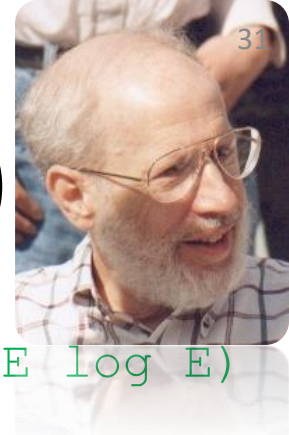
The central area shows a graph with five vertices labeled 0, 1, 2, 3, and 4. The edges and their weights are: (0,1) weight 4, (0,2) weight 4, (0,3) weight 6, (0,4) weight 9, (1,2) weight 2, (1,3) weight 8, (2,3) weight 8, and (3,4) weight 6. Vertices 1 and 2 are highlighted in green, and the edge (1,2) is highlighted in green, indicating it is the current edge being considered.

On the right, the "Kruskal's Algorithm" section displays the following text:

Adding edge (1,2) with weight 2 does not form a cycle, so add it to T. The current weight of T is 2.

```
Sort E edges by increasing weight
T = empty set
for (i=0; i<edgeList.length; i++)
    if adding e=edgeList[i] does not form a cycle
        add e to T
    else ignore e
T is an MST
```

Kruskal's Implementation (2)



```
sort the set of E edges by increasing weight //  $O(E \log E)$ 
T  $\leftarrow \{\}$ 
while there are unprocessed edges left //  $O(E)$ 
    pick an unprocessed edge e with min cost //  $O(1)$ 
    if adding e to T does not form a cycle //  $O(\alpha(V)) = O(1)$ 
        add e to the T //  $O(1)$ 
T is an MST
```

To sort the edges, we need $O(E \log E)$

To test for cycles, we need $O(\alpha(V))$ – small, assume constant $O(1)$ for $V \leq 1M$

In overall

- Kruskal's runs in $O(E \log E + E \alpha(V))$ // $E \log E$ dominates!
- As $E = O(V^2)$, thus Kruskal's runs in $O(E \log V^2) = O(E \log V)$

Let's have a quick look at KruskalDemo.java

Why Does Kruskal's Work? (1)

Kruskal's algorithm is also a **greedy algorithm**

Because **at each step**, it always try to select the next unprocessed edge **e** with **minimal weight** (greedy!)

Simple proof on how this greedy strategy works

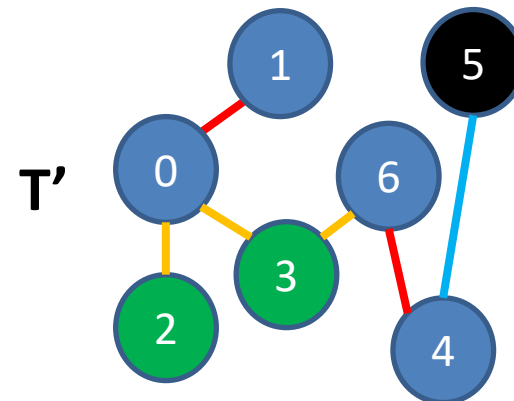
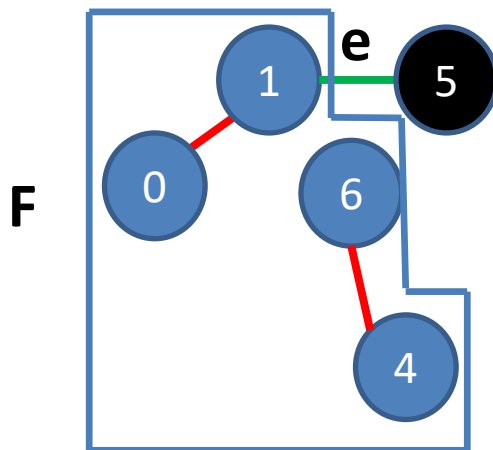
- Almost the same as that for Prim's

Why Does Kruskal's Work? (2)

with visual explanation

Proof by contradiction:

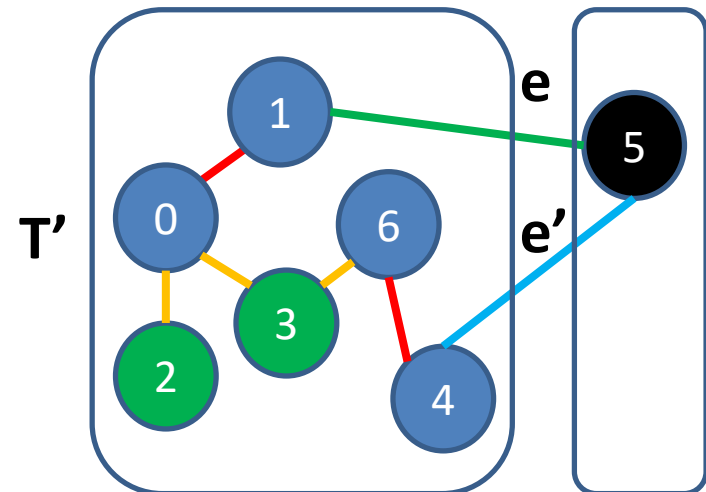
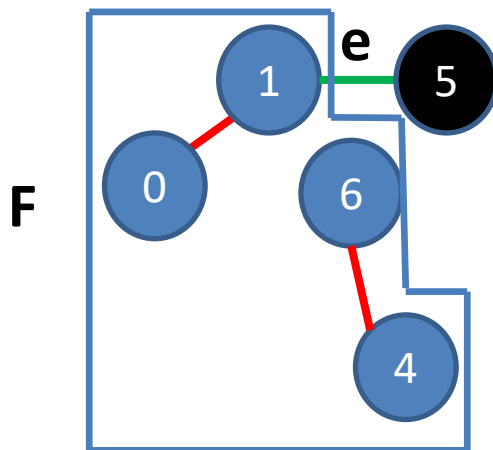
- Assume that edge **e** is the first edge at iteration k chosen by Kruskal's which is not in any valid MST.
- Let **F** be the forest generated by Kruskal's before adding **e**.
- Now **F** must be a part of some valid MST **T'**



Why Does Kruskal's Work? (3)

with visual explanation

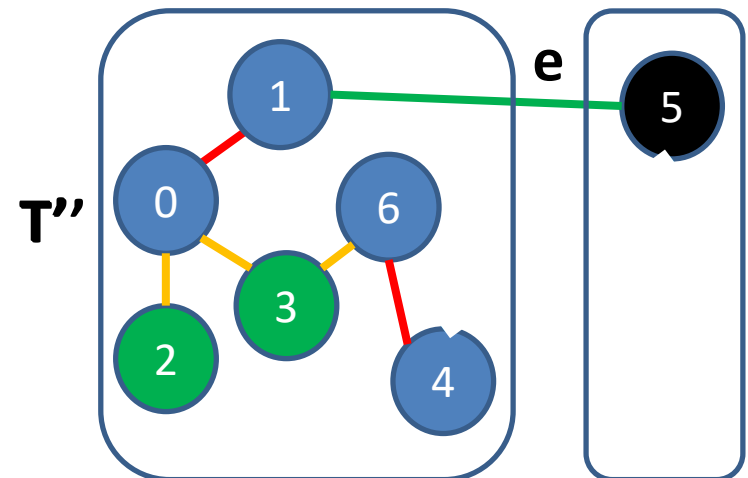
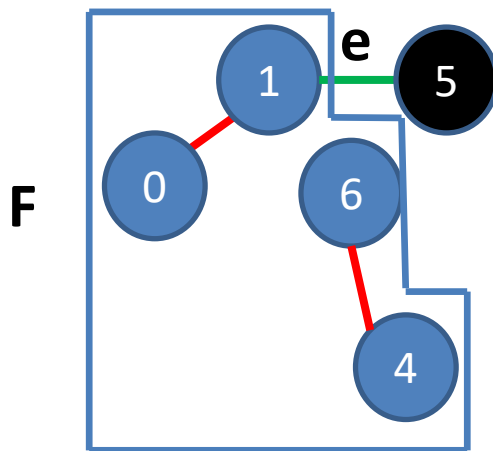
- Putting e into T' will create a cycle.
- Tracing the cycle using e to exit F , at some point we must come across an edge e' leading back to F
- Removing e and e' will create 2 components of T' ($\{0,1,2,3,4,6\}$ and $\{5\}$ in the example). This forms a cut where $\{e, e'\}$ is the cut set.



Why Does Kruskal's Work? (4)

with visual explanation

- At iteration k , both e and e' are candidate (they are not chosen and do not form a cycle if chosen).
- Since e was chosen, $w(e) \leq w(e')$ by the **cut property**
- Now replacing e' with e in T' must give us tree T'' covering all vertices of the graph s.t $w(T'') \leq w(T')$
- Contradiction that e is first edge chosen wrongly



If given an MST problem, I will...

1. Use/code Kruskal's algorithm
2. Use/code Prim's algorithm
3. No preference...

Summary

Introduce the MST problem (covered briefly in CS1231/S)

Discuss 2 algorithms

- Prim's algorithm (uses PriorityQueue ADT) & a variant for dense graphs where $E=O(V^2)$ (uses an array instead of PQ)
- Kruskal's algorithm (uses Edge List and UFDS)
- Can view the above 2 algorithms as making use of the Cut Property as opposed to the Cycle Property to construct the MST of any given connected weighted graph

You *may* learn MST/Prim's/Kruskal's again in CS3230