>>

Minimum Spanning Trees

∧ >>

❖ Minimum Spanning Trees

Reminder: Spanning tree *ST* of graph $G=(V,E)$

- spanning = all vertices, tree = no cycles
- *ST* is a subgraph of G ($G'=(V,E')$ where $E' \subseteq E$)
- *ST* is connected and acyclic

Minimum spanning tree *MST* of graph G

- *MST* is a spanning tree of G
- sum of edge weights is no larger than any other *ST*

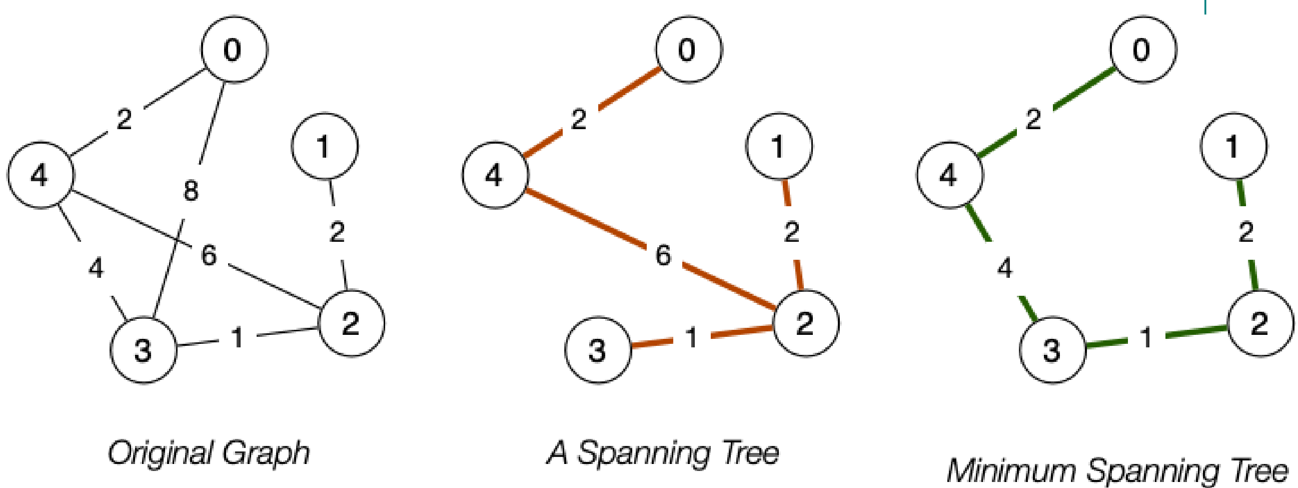
Applications:

- Computer networks, Electrical grids, Transportation networks ...

<< ∧ >>

❖ ... Minimum Spanning Trees

Example:



<< ∧ >>

❖ ... Minimum Spanning Trees

Problem: how to (efficiently) find MST for graph G?

One possible strategy:

- generate all spanning trees
- calculate total weight of each
- MST = ST with lowest total weight

Note that MST may not be unique

- e.g. if all edges have same weight, then all STs are MSTs

<< ∧ >>

❖ ... Minimum Spanning Trees

Brute force solution (using generate-and-test strategy):

```
findMST(G):
|  Input  graph G
|  Output a minimum spanning tree of G
|
|  bestCost=∞
|  for all spanning trees t of G do
|  |  if cost(t) < bestCost then
|  |     bestTree=t
|  |     bestCost=cost(t)
|  |  end if
|  end for
|  return bestTree
```

Not useful in general because #spanning trees is potentially large
(e.g. n^{n-2} for a complete graph with n vertices)

❖ ... Minimum Spanning Trees

Simplifying assumption:

- edges in *G* are not directed (MST for digraphs is harder)

If edges are not weighted

- there is no real notion of *minimum* spanning tree

Our MST algorithms apply to

- weighted, non-directional, connected graphs

<< ^ >>

❖ Kruskal's Algorithm

One approach to computing MST for graph *G* with *V* nodes:

1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added

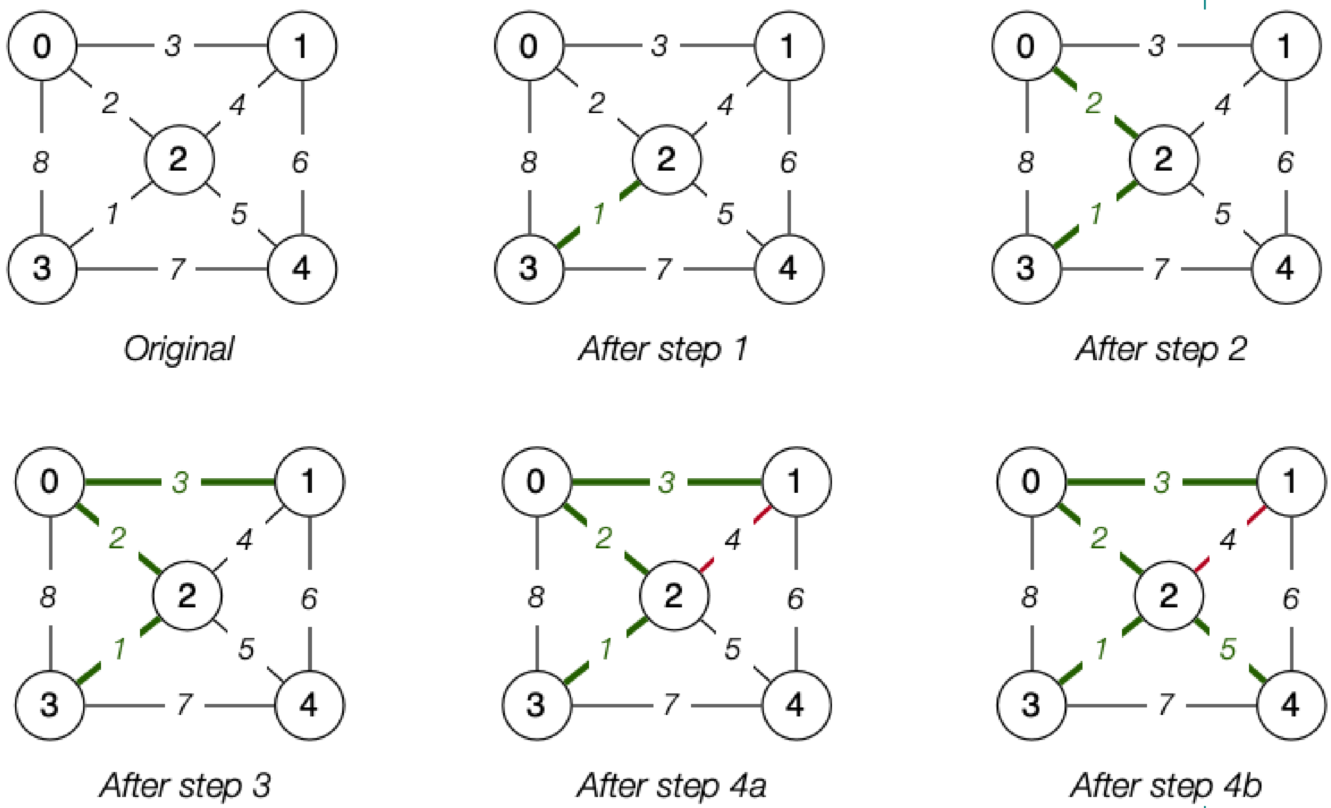
Critical operations:

- iterating over edges in weight order
- checking for cycles in a graph

<< ∧ >>

❖ ... Kruskal's Algorithm

Execution trace of Kruskal's algorithm:



<< ∧ >>

❖ ... Kruskal's Algorithm

Pseudocode:

```
KruskalMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e ∈ sortedEdgeList do
|  |  MST = MST ∪ {e}  // add edge
|  |  if MST has a cyle then
|  |     MST = MST \ {e}  // drop edge
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

❖ ... Kruskal's Algorithm

Rough time complexity analysis ...

- sorting edge list is *O(E·log E)*
- at least *V* iterations over sorted edges
- on each iteration ...
  - getting next lowest cost edge is *O(1)*
  - checking whether adding it forms a cycle: cost = $O(V^2)$

Possibilities for cycle checking:

- use DFS ... too expensive?
- could use *Union-Find data structure* (see Sedgewick Ch.1)

<< ∧ >>

❖ Prim's Algorithm

Another approach to computing MST for graph *G=(V,E)*:

1. start from any vertex *v* and empty MST
2. choose edge not already in MST to add to MST; must be:
   - incident on a vertex *s* already connected to *v* in MST
   - incident on a vertex *t* not already connected to *v* in MST
   - minimal weight of all such edges
3. repeat until MST covers all vertices

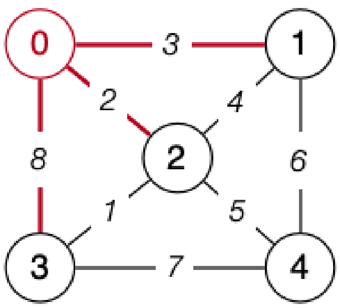
Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

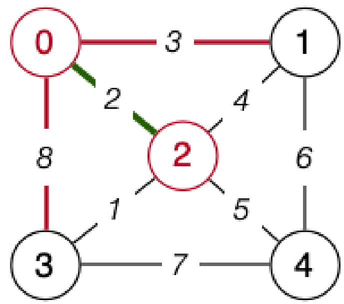
<< ∧ >>

❖ ... Prim's Algorithm

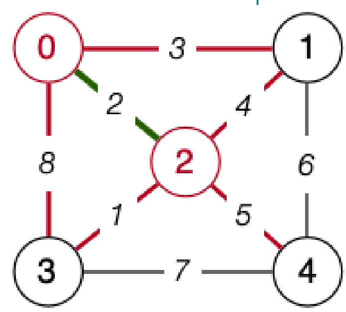
Execution trace of Prim's algorithm (starting at *s*=0):



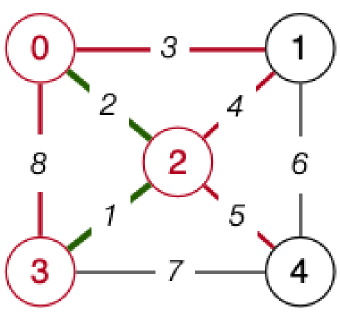
Start of step 1



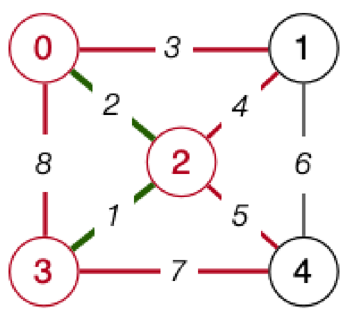
End of step 1



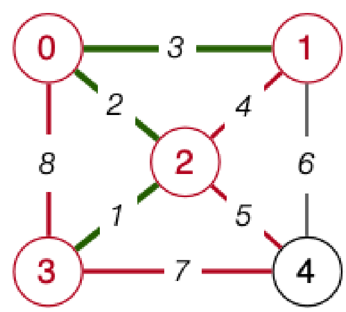
Start of step 2



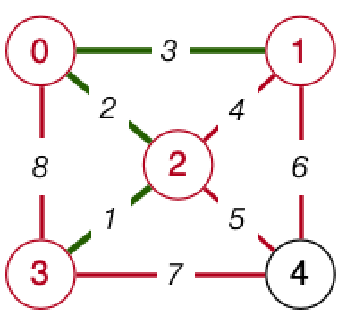
End of step 2



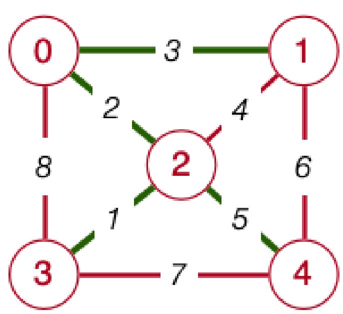
Start of step 3



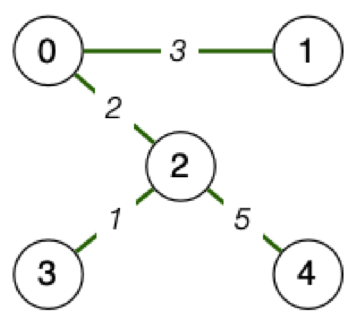
End of step 3



Start of step 4



End of step 4



MST

❖ ... Prim's Algorithm

Pseudocode:

```
PrimMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  usedV={0}
|  unusedE=edges(g)
|  while |usedV| < n do
|  |  find e=(s,t,w) ∈ unusedE such that {
|  |     s ∈ usedV ∧ t ∉ usedV
|  |       ∧ w is min weight of all such edges
|  |  }
|  |  MST = MST ∪ {e}
|  |  usedV = usedV ∪ {t}
|  |  unusedE = unusedE \ {e}
|  end while
|  return MST
```

Critical operation: finding best edge

❖ ... Prim's Algorithm

Rough time complexity analysis ...

- V iterations of outer loop
- in each iteration, finding min-weighted edge ...
  - with set of edges is $O(E) \Rightarrow O(V{\cdot}E)$ overall
  - with priority queue is $O(\log E) \Rightarrow O(V{\cdot}\log E)$ overall

Note:

- have seen stack-based (DFS) and queue-based (BFS) traversals
- using a *priority queue* gives another non-recursive traversal

<< ∧ >>

❖ Sidetrack: Priority Queues

Some applications of queues require

- items processed in order of "key"
- rather than in order of entry (FIFO — first in, first out)

Priority Queues (PQueues) provide this via:

- **`join`**: insert item into PQueue (replacing **`enqueue`**)
- **`leave`**: remove item with largest key (replacing **`dequeue`**)

Will discuss priority queues in more detail in another video

<< ∧

❖ Other MST Algorithms

Boruvka's algorithm ... complexity *O(E·log V)*

- the oldest MST algorithm
- start with V separate components
- join components using min cost links
- continue until only a single component

Karger, Klein, and Tarjan ... complexity *O(E)*

- based on Boruvka, but non-deterministic
- randomly selects subset of edges to consider
- for the keen, here's the paper describing the algorithm

Produced: 4 Jul 2020