CSE 3318 Notes 14: Minimum Spanning Trees

(Last updated 1/5/24 11:41 AM)

CLRS 19.3, 21.1-21.2

14.A. Concepts

Given a weighted, connected, undirected graph, find a minimum (total) weight free tree connecting the vertices. (AKA bottleneck shortest path tree)

Cut Property: Suppose S and T partition V such that

1. $S \cap T = \varnothing$
2. $S \cup T = V$
3. $|S| > 0$ and $|T| > 0$

then there is some MST that includes a minimum weight edge $\{s, t\}$ with $s \in S$ and $t \in T$.

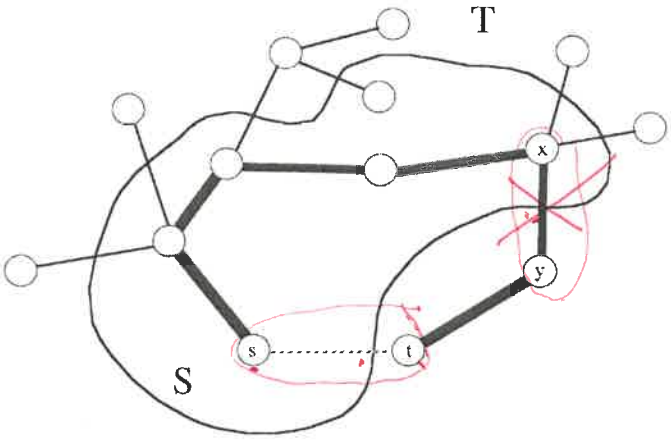
Proof:

Suppose there is a partition with a minimum weight edge $\{s, t\}$.

A spanning tree without $\{s, t\}$ must still have a path between s and t.

Since $s \in S$ and $t \in T$, there must be at least one edge $\{x, y\}$ on this path with $x \in S$ and $y \in T$.

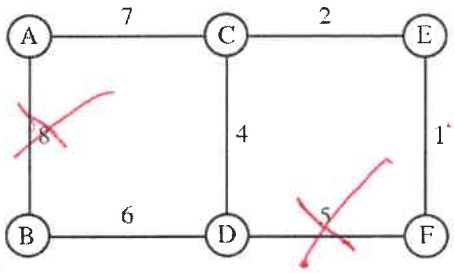
By removing $\{x, y\}$ and including $\{s, t\}$, a spanning tree whose total weight is no larger is obtained. •••



Cycle Property: Suppose a given spanning tree does not include the edge $\{u, v\}$. If the weight of $\{u, v\}$ is no larger than the weight of an edge $\{x, y\}$ on the unique spanning tree path between u and v, then replacing $\{x, y\}$ with $\{u, v\}$ yields a spanning tree whose weight does not exceed that of the original spanning tree.

Proof: Including {u, v} in the set of chosen edges introduces a cycle, but removing {x, y} will remove the cycle to yield a modified tree whose weight is no larger.

The proof suggests a slow approach - iteratively find and remove a maximum weight edge from some remaining cycle:



14.B. PRIM'S ALGORITHM – Three versions

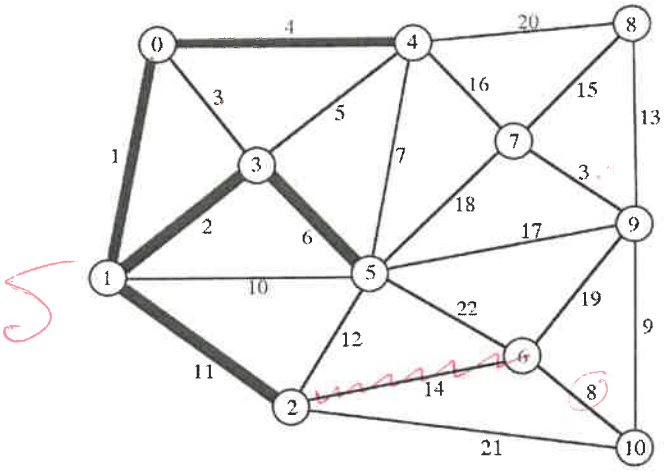
Prim's algorithm applies the cut property by having S include those vertices connected by a subtree of the eventual MST and T contains vertices that have not yet been included. A minimum weight edge from S to T will be used to move one vertex from T to S

1. "Memoryless" – Only saves partial MST and current partition.
   (https://ranger.uta.edu/~weems/NOTES3318/primMemoryless.c)

Place any vertex x ∈ V in S.
T = V – {x}
while T ≠ ∅
    Find the minimum weight edge {s, t} over all t ∈ T and all s ∈ S. (Scan adj. list for each t)
    Include {s, t} in MST.
    T = T – {t}
    S = S ∪ {t}

Since no substantial data structures are used, this takes $\Theta(EV)$ time.

Which edge does Prim's algorithm select next?

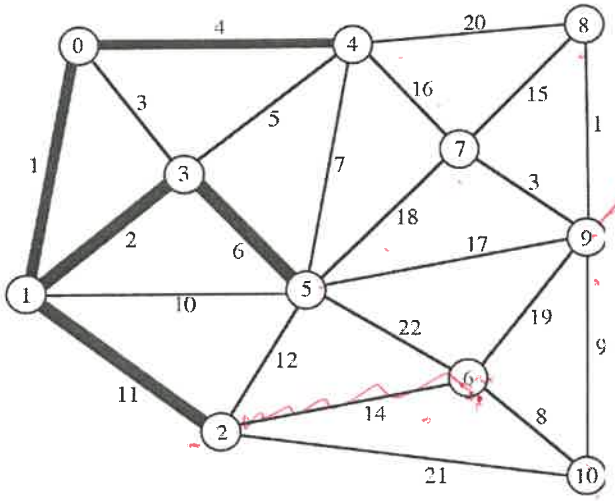


2. Maintains T-table that provides the closest vertex in S for each vertex in T.
   (`https://ranger.uta.edu/~weems/NOTES3318/primTable.c` traverses adjacency lists)

Eliminates scanning all T adjacency lists in every phase, but still scans the adjacency list of the last vertex moved from T to S.

Place any vertex x ∈ V in S.
T = V – {x}
for each t ∈ T
    Initialize T-table entry with weight of {t, x} (or ∞ if non-existent) and x as best-S-neighbor
while T ≠ ∅
    Scan T-table entries for the minimum weight edge {t, best-S-neighbor[t]}
        over all t ∈ T and all s ∈ S.
    Include edge {t, best-S-neighbor[t]} in MST.
    T = T – {t}
    S = S ∪ {t}
    for each vertex x in adjacency list of t
        if x ∈ T and weight of {x, t} < T-weight[x]
            T-weight[x] = weight of {x, t}
            best-S-neighbor[x] = t

What are the T-table contents before and after the next MST vertex is selected?



	After
6	14 (2)
7	16 (4)
8	20 (4)
9	17 (5)
10	21 (2)

Analysis:

Initializing the T-table takes Θ(V).
Scans of T-table entries contribute $\Theta(V^2)$.
Traversals of adjacency lists contribute Θ(E).
$\Theta\left(V^2 + E\right) = \Theta(V^2)$ overall worst-case.

3. Replace T-table by a min-heap.
   (https://ranger.uta.edu/~weems/NOTES3318/primHeap.cpp)

The time for updating for best-S-neighbor increases, but the time for selection of the next vertex to move from T to S improves.

Place any vertex x ∈ V in S.
T = V – {x}
for each t ∈ T
    Load T-heap entry with weight (as the priority) of {t, x} (or ∞ if non-existent) and x as best-S-neighbor
`minHeapInit`(T-heap) // a `fixDown` at each parent node in heap
while T ≠ ∅
    Use `heapExtractMin` /* `fixDown` */ to obtain T-heap entry with the minimum weight edge over all t ∈ T and all s ∈ S.
    Include edge {t, best-S-neighbor[t]} in MST.
    T = T – {t}
    S = S ∪ {t}
    for each vertex x in adjacency list of t
        if x ∈ T and weight of {x, t} < T-weight[x]
            T-weight[x] = weight of {x, t}
            best-S-neighbor[x] = t
            `minHeapChange`(T-heap) // `fixUp`

Analysis:

Initializing the T-heap takes Θ(V).
Total cost for `heapExtractMins` is Θ(V log V).
Traversals of adjacency lists and `minHeapChanges` contribute Θ(E log V).
$\Theta(E \log V)$ overall worst-case, since E > V.

Which version is the fastest?

	Theory	Sparse $(E = O(V))$	Dense $(E = \Omega(V^2))$
1.	$\Theta(EV)$	$\Theta(V^2)$	$\Theta(V^3)$
2.	$\Theta(V^2)$	$\Theta(V^2)$	$\Theta(V^2)$
3.	$\Theta(E \log V)$	$\Theta(V \log V)$	$\Theta(V^2 \log V)$

May 1, Wed 1-4, Extra OH
May 3, Fri 2-4:30, Final

14.C. UNION-FIND TREES TO REPRESENT DISJOINT SUBSETS

Abstraction:

Set of n elements: 0 . . n - 1

Initially all elements are in n different subsets

`find(i)` - Returns integer ("leader") indicating which subset includes `i`

`i` and `j` are in the same subset ⇔ `find(i)==find(j)`

`unionFunc(i,j)` - Takes the set union of the subsets with leaders `i` and `j`.

Results of previous `finds` are invalid after a `union`.

Implementation 1: "Colored T-Shirts" (https://ranger.uta.edu/~weems/NOTES3318/uf1.c)

Initialization:

```
for (i=0; i<n; i++)
  id[i]=i;
```

Θ(n)

```
find(i):

    return id[i];
```

Θ(1)

```
unionFunc(i,j):

    for (k=0; k<n; k++)
      if (id[k]==i)
        id[k]=j;
```

Θ(n)

0	1	2	3	4
0	1	2	3	4

union(2, 3)
union(4, 1)
union(3, 1)

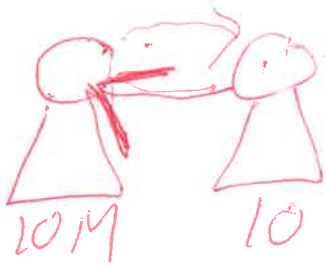
Implementation 2: Trees with Parent Pointers (https://ranger.uta.edu/~weems/NOTES3318/uf2.c)

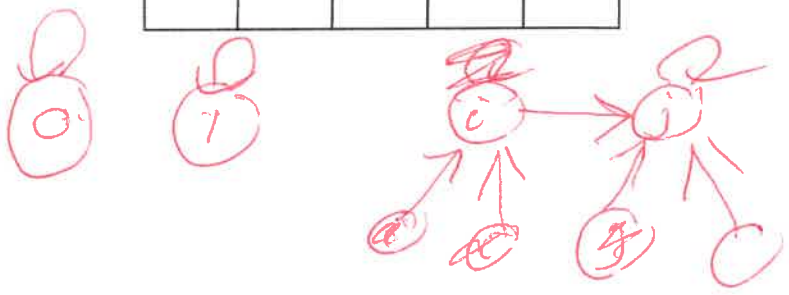
```
find(i):

    while (id[i]!=i)
      i=id[i];
    return i;

unionFunc(i,j):

    id[i]=j;
```

Θ(n)

Θ(1)

0	1	2	3	4
0	1	2	3	4





Implementation 3: (https://ranger.uta.edu/~weems/NOTES3318/uf3.c)

unionFunc forces leader of smaller subset to point to leader of larger subset

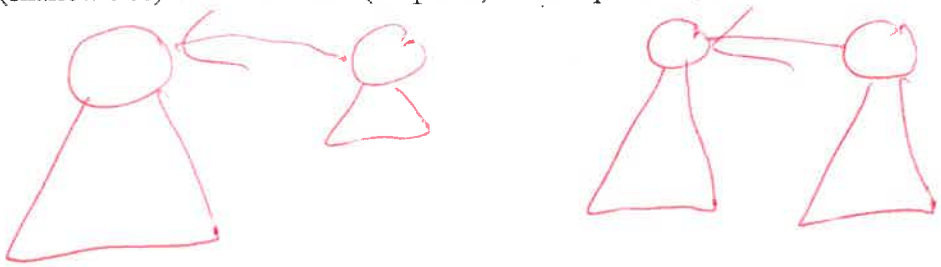
Initialization:

```
for (i=0; i<n; i++)
{
  id[i]=i;
  sz[i]=1;
}
```

```
find(x):

      for (i=x;
           id[i]!=i;
           i=id[i])
        ;
      root=i;
      // path compression - make all nodes on path
      // point directly at the root
      for (i=x;
           id[i]!=i;
           j=id[i],id[i]=root,i=j)
        ;
      return root;
```

```
unionFunc(i,j):

      if (sz[i]<sz[j])
      {
        id[i]=j;
        sz[j]+=sz[i];
      }
      else
      {
        id[j]=i;
        sz[i]+=sz[j];
      }
```

Best-case (shallow tree) and worst-case (deep tree) for a sequence of unions?



14.D. KRUSKAL'S ALGORITHM – A Simple Method for MSTs Based on Union-Find Trees

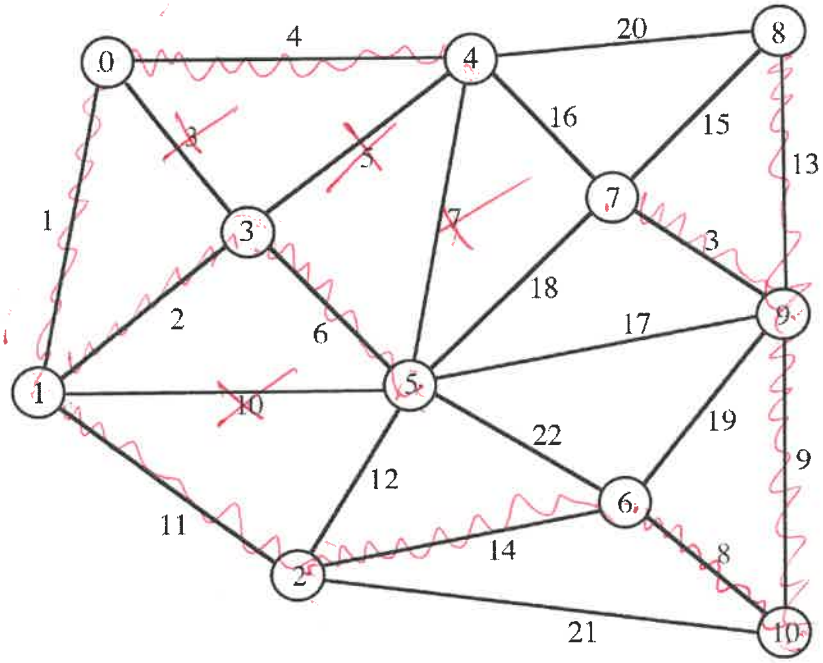
(https://ranger.uta.edu/~weems/NOTES3318/kruskal.c)

Sort edges in ascending weight order.

Place each vertex in its own set.

Process each edge {x, y} in sorted order:

```
a=FIND(x)
b=FIND(y)
if a ≠ b
        UNION(a,b)
        Include {x, y} in MST
```



1	{0, 1}	12	{2, 5}
2	{1, 3}	13	{8, 9}
3	{0, 3}	14	{2, 6}
3	{7, 9}	---	---
4	{0, 4}	15	{7, 8}
5	{3, 4}	16	{4, 7}
6	{3, 5}	17	{5, 9}
7	{4, 5}	18	{5, 7}
8	{6, 10}	19	{6, 9}
9	{9, 10}	20	{4, 8}
10	{1, 5}	21	{2, 10}
11	{1, 2}	22	{5, 6}

Time to sort, $\Theta(E \log V)$, dominates computation