Graph Algorithms:

Minimum (Maximum) Spanning Tree (Ch 23, Ch 21, L #9)

Graphs in this section are undirected i.e. all edges are undirected. [The directed version is described in CS 6381]. If there are parallel edges, all but one can be removed without loss of generality. No self loops \( (u, u) \) (edges of the form \( (u, u) \)).

**Input:** A connected undirected Simple Graph \( G = [V; E] \) with edge weights \( w(e), e \in E \). These weights may be +ve, -ve, or 0.

**Definition:** Connected: \( \exists \) a path from \( i \) to \( j \) \( \forall i, j \in V \).

**Simple:**

**Tree:** Connected subgraph with no cycles

**Sp. Tree:** A tree that contains all vertices

Examples, on next page
Problem: Find a spanning tree such that \( \sum w(e) \) is minimum (maximum)\footnote{one of the two}.

We assume min from now on.

**Algorithms**

1. Improvement algorithm
2. Boruvka algorithm (1926)
3. Kruskal Algorithm A \( \{ \) \( \}\) (1956)
4. \( \)
5. Jarnik (1930) - Prim (1957) - Dijkstra (1959)

\[\text{Algorithm} \quad \text{Commonly Used Name}\]

Please Do NOT use this (Confusion with Shortest Path alg)
Since all these algorithms are implemented on machines, we need to specify input in the appropriate form [No "figures" or "pictures"]

For our purposes, best is to specify a "list" of vertices $V$; and for each vertex, an adjacency list $\text{adj}(u); \forall u \in V$. This is particularly useful if the graph is sparse i.e $|E| = o(n^2); n = |V|$. 

**Spanning Tree Problems**: If $G$ is undirected and hence $(u, v) \in E$ is listed both in $\text{adj}(u)$ as $v \in \text{adj}(u)$ and in $\text{adj}(v)$ as $u \in \text{adj}(v)$. When we do Shortest Path and Maxflow, a directed edge $(u \rightarrow v)$ will only be there as in $v \in \text{adj}(u)$.

We are using "figures" for easier understanding. The graph used for all algorithms is
**Improvement Algorithm**

Start with an arbitrary spanning tree (obtained for example by using DFS, BFS, or a good guess obtained in some manner). For example:

![Graph Diagram]

If this is not the Min. wt. Sp. tree, then a Min. wt. Sp. tree must include some edge (one or more) that are not present here. Our idea is to try to include one at a time and see if there is improvement. For example say we try edge (D,E) shown above in green

But adding (D,E) to the tree, creates a (unique) cycle and we can not have cycles in a tree. To "break" the cycle, we "remove" edge in the cycle with "largest" weight — in this case (C,D) to get an "improved" tree.
New Improved tree.

A saving of 5 since we added an edge with weight 2 and removed one with weight 7.

We repeat this process until an improvement of this type is not possible.

Several questions arise in this setting:

1. Does this process stop? In how many steps?
   Is the algorithm polynomially bounded?

2. When the algorithm stops, do we have an optimal solution?

Answer to both questions is in the affirmative.

On #1: When an edge is "removed" at any step, this edge need not be considered at any later steps. This makes the algo. poly-time.

On #2: Proof is complicated and done in CS6381.

Most powerful Fourth paradigm on creating algorithms.
Boruvka Algorithm.

Start with only nodes V. Each is a connected component by itself at this point.

At any step, for each connected component, select an edge with one end inside & one outside with minimum edge weight among such edges. Add all (possibly) of them at this step: In this algorithm, many edges are added & at each step. The algorithm stops when we have only one component (Hopefully at this point we have a Spanning Tree).

Example:

Step 1:

<table>
<thead>
<tr>
<th>Node</th>
<th>Nearest Neighbor Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AB</td>
</tr>
<tr>
<td>B</td>
<td>AB</td>
</tr>
<tr>
<td>C</td>
<td>CF</td>
</tr>
<tr>
<td>D</td>
<td>DE</td>
</tr>
<tr>
<td>E</td>
<td>DE</td>
</tr>
<tr>
<td>F</td>
<td>CF</td>
</tr>
<tr>
<td>G</td>
<td>EG</td>
</tr>
<tr>
<td>H</td>
<td>FH</td>
</tr>
<tr>
<td>I</td>
<td>GI</td>
</tr>
</tbody>
</table>

Greedy choice: irreversible.
Step 2: Component

\[
\begin{align*}
A & \quad B \\
C & \quad F & \quad H \\
DEG & \\
\end{align*}
\]

Min. w.t. edge to outside

\[
AC \text{ or } BE \underline{\text{(Must Choose one Arbitrarily)}}
\]

\[
EH \
EH \
\]

---

First Step

Second Step

---

Single Component $\Rightarrow$ Spanning Tree (Min)

---

Data Structure for Implementation

Complicated one due to A.C. Yao. Has the best Complexity with Yao's implementation.

---

Prim's Algorithm (First Creator Jarnik) (Greedy)

Select a node (arbitrarily) & find cheapest edge incident at this node. Now two nodes are connected. Find cheapest edge connecting either of these two to a third node. Include that edge and repeat the process. [Local Greediness]
In our example starting (arbitrarily) with $D*$.

**Step 1** in **Blue**

Choice: $DE, DF, DC$

**Step 2** in **Red**

Choice: $BD, DF$

Old: $DC, DF$

New $BE, EH, EG$

**Step 3** in **Green**

Choice:

Old: $DC, DF$, $BE, EG$

New $EH, HI, HG$
Step 4: in black

Old: DC
   BE, EG
   HI, HG

New: CF

Steps in blue (2)

Old: BE, EG
   HI, HG

New (blue(2)): CB, AC

Tie: AC, BE

Step 6: in red (2)

Old: BE
   HI
   AC (blue(2))
   CB (not)

New: GE (red(2))

Tie: AC, BE: Must choose one
(Arbitrarily)
BE
Step 1: in Green(2)
old: H I (Green(1))
   AC (Blue(2))
   GI (Red(2))
new: AB (Green(2))

Steps: in Black(2)
old: H I (Green(1))
   GI (Red(2))

Min. Sz. Free.

Datastructure: Discussed later.

Kruskal Algorithms A and B
Both "global" greedy but from opposite direction. Next page
Kruskal Algorithm A:

Notation for Conveniece

Edge included in sp. tree colored blue and those excluded colored red. Those not yet processed will be colored black.

Algorithm:

"Process" edges in increasing order of weights. At any point, in processing the next edge, color it blue (include) if doing so does not create a blue cycle (a cycle of all blue edges); if adding it (coloring it blue) would create a blue cycle, then color it red.

For our "numerical" example, edges will be processed in order:

CF, DE*, FH, AB*, EH, DF, CD*, EG, BE*, AC, GI, HI, BC, GH.

[For all these algorithms and the problem itself the actual weights do not matter; only their order matters, i.e. algorithms in this section only do comparisons.]

The evolution of algorithm is shown in next page.* indicates how "ties" are broken (arbitrary)
Numbering indicates order of processing. Blue edges \( \rightarrow \) form a Min. Sp. Tree (because of how ties were broken).

Data structure will be discussed later along with Complexity.

Kruskal algorithm B:

Process edges in decreasing order of weights.
At any point, in processing next edge, color it red (remove the edge \( \rightarrow \) do not include) if doing so does not disconnect the graph; if it disconnects, Color it blue.

[The attempt here is to keep the remaining graph connected at all times]

"Numerical" example on next page.
Removing edges $GH$, $BC$, $HI$, does not disconnect the graph. So they are colored red in that order. An attempt at this point to remove GI disconnects the graph; so it is colored blue. Numbers indicate order of processing of the edges.

For a given order both algorithms yield the same tree at the end.

Implementation and Complexity

We will do this for Prim's Algorithm and Kruskal Algorithm A.

For Prim algorithm: At any step we have a sub-tree (connected) of the spanning tree and we need the next edge: Min $w(t)$ among those that connect a node of the sub-tree to a node not in the sub-tree at each step.
At any step, we keep track of the "nearest" node in the subtree for each node outside using key values for each node in the form of a min heap. Recall we arbitrarily choose D as our starting node.

Step 1: \[ D: \begin{array}{cccccccc} A & B & C & D & E & F & G & H & I \\ \infty & \infty & \infty & 7 & -2 & 6 & \infty & \infty & \infty \end{array} \]

First heap

Extract Min: \( E: 2 \)

Subtree

New node in sub-tree E. Look at its edges.
Step 2:

Heap Structure Evolution

4. Violates Min heap bubble up

Ext. Min

Sub tree

New node → H
The process continues this way.

Each step involves the sub tree growing by one node.

# of extract mins: \((n-1)\) \(n = |V|\)

Work / \(\text{work} : \Omega(lg n)\)

# of possible key value changes: \(|E|\)

Work / edge key value change : \(\Omega(lg n)\)

\(\therefore\) Total work: \(\Theta(|E| lg |V| + |V| lg |V|)\)

\(\therefore \Theta(|E| lg |V|)\)

---

Proof of correctness: later, for Prim's Algorithm

Data structure for Kruskal Algorithm: A UNION-FIND in the next lecture.