Minimum Spanning Trees

Analysis of Algorithms
Minimum Spanning Tree

• The minimum spanning tree (MST) of a weighted connected graph is a subgraph that contains all of the vertices of the original graph and a subset of the edges so that:
  – The subgraph is connected (it *spans* the entire graph)
  – The total weight of the subgraph is the smallest possible
Example Spanning Tree

Is the overall cost minimal?

spanning tree $T$: cost = $50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$
Brute Force MST Algorithm

• We could look at each subset of the edge set and first see if the resulting subgraph is connected, and then see if its total weight is smaller than our best answer so far.

• If there are $M$ edges in the graph, this process will require that we look at $2^M$ different subsets.

• This is a lot more work than is needed!
Kruskal MST Algorithm

• Kruskal’s algorithm concentrates on the edges rather than the vertices
• Sort the edges in order of increasing weight
• Add an edge to the MST as long as it does not form a cycle with edges added in previous passes
• Stop when all of the vertices are connected and there are $N-1$ edges in the MST
Kruskal Example

Original graph: Select edge D-F:
Kruskal Example

Select edge A-B:

Select edge B-E:
Kruskal Example

Select edge A-C:

Select edge A-F:

At this point, either D-G or F-G could be chosen, but not D-B or C-F. Why?
Kruskal Algorithm

• Basic Kruskal algorithm:
  – Sort the edges in order of non-decreasing weight
  – For every edge taken in order of non-decreasing weight
    • Add the edge to the Kruskal tree as long as a cycle is not created
Is Kruskal Correct?

• Proof of correctness of Kruskal:
  – Definition: A cut of a graph is a partition of its vertices into two (nonempty) sets.
  – Definition: A crossing edge connects a vertex in one set with a vertex in the other.
  – Claim: Given any cut, the crossing edge of min weight is in the MST
  – Proof: Assume that min-weight crossing edge $e$ is not in the MST. If we add this edge to the MST, a cycle will be created. Therefore, some other edge $f$ in this cycle must be a crossing edge. But if we remove $f$ and add $e$, we also have a spanning tree. Since weight of $e$ is less than the weight of $f$, this spanning tree is lower weight. This contradicts our assumption that the min-weight crossing edge $e$ is not in the MST, therefore the assumption is incorrect and our claim is correct.
  – Given any cut, we can exchange a smaller weight edge for a larger one, to get a tree with smaller weight.
Prim MST Algorithm

• Prim’s minimum spanning tree algorithm concentrates on **vertices** rather than edges
• Choose a starting vertex, put it in the MST, and then repeatedly choose other vertices to put in the tree
• To find the next vertex, consider all of the edges from the starting vertex, and choose the neighbor vertex of the starting vertex whose edge weight is the smallest
• On each subsequent pass, choose the vertex (among those not yet in the MST) that has the smallest edge weight connected to any of the vertices already in the MST
Prim Example

Original graph: “Fringe” from vertex A: Choose edge A-B
Prim Example

Choose edge B-E:

Choose edge A-C:
Prim Example

Choose edge A-F:

Choose edge F-D:
Prim Example

Choose edge F-G:

Final Prim Tree:
Prim Pseudocode

Select a starting vertex

Build the initial fringe from vertices connected to the starting vertex

While there are vertices left do:
    Choose the edge to the fringe with the smallest weight
    Add the associated vertex to the tree
    Update the fringe:
        Add vertices to the fringe connected to the new vertex
        Update the edges to the fringe so that they are the smallest

End While
Prim Uses a Priority Queue

• Prim MST algorithm: Repeatedly select the next vertex that is closest to the tree we have built so far

• To implement Prim, we need some sort of data structure that will enable us to associate a key (the distance to the tree under construction) with each vertex, and then rapidly select the vertex with the lowest key

• Use a priority queue implemented as a min heap (assuming a sparse graph)
  – We will see how to do this later in the course
QUESTION: Suppose that you are running Prim's algorithm with $N = 8$. The edges added to the MST so far are $(0, 2)$, $(0, 7)$, and $(1, 7)$. What are the keys (edge weights of the fringe) in the priority queue?

a) 17, 38, 32, 40
b) 17, 37, 28, 40
c) 17, 37, 28, 32
d) 0, 19, 26, 17, 37, 28, 40, 16

Assume the graph weights are as follows: $(0, 2)=26$, $(0, 4)=38$, $(0, 6)=58$, $(0, 7)=16$, $(1, 2)=36$, $(1, 3)=29$, $(1, 5)=32$, $(1, 7)=19$, $(2, 3)=17$, $(2, 7)=34$, $(3, 6) = 52$, $(4, 5)=35$, $(4, 7)=37$, $(5, 7)=28$, $(6, 2)=40$, $(6, 4)=93
Analysis of MST Algorithms

• Kruskal always maintains a spanning subgraph that becomes a tree at the last step
  – At each step (after ordering the edges in terms of increasing weight),
    select the lowest weight edge that is not already in the subgraph and add it to the MST, as long as no cycle is formed

• Prim always maintains a tree that becomes spanning at the last step
  – At each step select the highest priority (lowest weight) vertex not already in the tree and add it to the MST

• Making selections this way are examples of a general class of algorithms: greedy algorithms
Analysis of MST Algorithms

• Kruskal’s runtime is proportional to $M \log M$ for sorting the edges plus $M \alpha(N)$ for cycle checking where $\alpha$ is the inverse Ackerman function
  – Sorting the edges dominates the runtime
• Prim’s runtime is proportional to $M \log N$
  – If using a priority queue implemented as a binary heap
• Question: Which is faster, Kruskal or Prim?
• Unsolved problem: Does a general-purpose linear-time MST algorithm exist?
Kruskal Implementation Details

• How to tell if there is a cycle?
  – Before adding a candidate edge to the MST, perform a depth-first search and look for a back edge
    • Total runtime of Kruskal using DFS is proportional to $M (N+M)$
    • $M \log M$ for sorting the edges plus $M (N+M)$ for cycle-checking
    • Can we do any better? Yes!
  – Maintain a set of subtrees (i.e., maintain a forest of trees)
    • Use the union-find data structure to determine if the two vertices of the candidate edge are in the same subtree
    • If the two vertices of the candidate edge are in the same subtree they will have the same root vertex, therefore reject the edge because adding it to the MST would create a cycle
Cycle Detection

• Suppose that at some stage of Kruskal, the next candidate edge is \((u, v)\). Then there are two possibilities:
  - \(u\) and \(v\) are not in the same subtree, therefore adding the edge will not create a cycle, or
  - \(u\) and \(v\) are in the same subtree, therefore adding the edge will create a cycle (reject the edge)
  - In either case, checking \(\text{root}(u)\) and \(\text{root}(v)\) will let us know if \(u\) and \(v\) are already in the same subtree
Cycle Detection

• Therefore, we need a data structure that allows us to quickly find the subtree to which the two vertices belong
  – If they are not in the same subtree, then we must merge the two subtrees into a single subtree

• The appropriate data structure for this problem is the *partition* (also called disjoint subsets)
  – A partition of a set is a collection of disjoint subsets that covers then entire set
Union-Find Data Structure

• Initially, the partition consists of $N$ subsets of size 1
  – Each vertex in the graph is a subtree of size 1
• Whenever an edge is added to the MST, two subtrees are joined together and the total number of subtrees decreases by one
• The operations needed to do this are:
  – $\text{find}(u)$ and $\text{find}(v)$: return the roots of the subtrees that vertices $u$ and $v$ belong to
  – $\text{union}(t_1, t_2)$: update the partition by merging two subtrees, $t_1$ and $t_2$, into a single subtree, assuming $u$ and $v$ have different roots
Naïve Partitions

• One simple way to represent a partition is to choose one element of each subset to be the root.

• For example, consider the partition with roots 0, 1, and 4:

\[ \{0, 2 \mid 1, 3, 5 \mid 4, 6, 7\} \]

• This partition could be implemented using an array as:

<table>
<thead>
<tr>
<th>v</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(v)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Naïve Partitions

- The operation **find** is straightforward – we can decide whether $u$ and $v$ are in the same partition just by comparing $p[u]$ and $p[v]$
- Thus, **find** has runtime that is constant, regardless of the number of vertices
Updating the Partition

• Suppose now that we wish to update the partition by merging (union) the first two subsets:
  \{0, 1, 2, 3, 5 \mid 4, 6, 7\}

  \[
  \begin{array}{cccccccc}
  \text{v} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  p(v) & 0 & 0 & 0 & 0 & 4 & 0 & 4 & 4 \\
  \end{array}
  \]

• To do this, we might potentially have to run through the entire array to update each vertex’s root: this takes runtime proportional to N

• Can we improve the runtime of the union operation?
Disjoint Sets Forest

- Consider implementing the partition as a set of subtrees using linked lists, where each vertex points to its root
Disjoint Sets Forest

• Option 1: the union of two subsets can be accomplished by adjusting the pointers so they point to the new root
  – For example, the union of (0, 2) and (1, 3, 5) yields:

```
0
  2
1
  3  5
→
0
  2  1  3  5
```

• Can we achieve the same thing by adjusting just a single pointer?
Disjoint Sets Forest

• Option 2: we could union two subsets by linking a single pointer instead of all of them:

\[\begin{array}{c}
\begin{array}{c}
0 \\
2
\end{array}
\begin{array}{c}
1 \\
2 & 3 & 5
\end{array}
\end{array}\rightarrow
\begin{array}{c}
\begin{array}{c}
0 \\
2
\end{array}
\begin{array}{c}
1 \\
3 & 5
\end{array}
\end{array}\]

• But now we have lost the ability to quickly find the root of the subtree
Disjoint Sets Forest

• For option 2 the union has constant runtime because we are only changing a single pointer

• However, we have lost the ability to quickly find the root
  – The runtime of the \textbf{find} operation is now proportional to N, so we seemed to have gained little over the array implementation

• There are two heuristics that can be used to speed things up at the cost of a little extra data: \textit{union-by-rank} and \textit{path compression}
Disjoint Sets Forest

• Question: Suppose that in the array implementation, we *do not* perform the update operation (updating each vertex’s root) after the union of two subsets, and the array is as such:

<table>
<thead>
<tr>
<th>v</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[v]</td>
<td>0</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

• What are the roots of the subtrees of vertex $v = 3$ and vertex $v = 7$, respectively?
Union-by-Rank Heuristic

• Let the rank of a vertex be the height of the tree it belongs to
  – Initially each vertex is the root of its own tree with rank = 0

• When a union operation is done, the root of the shorter tree (smaller rank) is made to point to the root of the taller tree (higher rank)
  – The resulting tree therefore does not increase its height unless both trees are the same height in which case the height increases by one
Union-by-Rank Heuristic

Union-by-rank: the root of the *shorter* sub-tree is linked to the root of the longer sub-tree. This will tend to keep the *height* of newly formed tree short.

```c
void union (int u, int v) {
    i = find(u);
    j = find(v);
    if (i.rank > j.rank) {
        j.parent = i;
    } else {
        i.parent = j;
        if (i.rank == j.rank)
            j.rank = j.rank + 1;
    }
}
```

Union-by-weight: the root of the *smaller* sub-tree is linked to the root of the larger sub-tree. This will tend to keep the *size* of newly formed tree small.
Union-by-Rank Heuristic

Question: Given the following array, what will be the entries in the array after union(3, 6), using the union-by-rank heuristic?

<table>
<thead>
<tr>
<th>v</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[v]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

What will be the entries in the array if instead of union-by-rank, you use the union-by-weight heuristic?
Path Compression Heuristic

• Notice that when we perform $\text{find}(v)$ we have to follow a path from $v$ to the root of the subtree containing $v$

• Since it is recursive, why not simply make all of the elements point directly to the root of the subtree when the recursion unwinds?

• This is reminiscent of the naïve algorithm, where we made every element point directly to the root, but here it is much cheaper because we only alter pointers that we needed to look at anyway
Path Compression Heuristic

Find the root of the subtree for a given vertex, and while you are at it, make all vertices in that subtree point to the root when it is found.

```c
int find(int v) {
    if (v != v.parent) {
        v.parent = find(v.parent);
    }
    return v.parent;
}
```

Note that `find` is a *two-pass* procedure: it makes one pass up the subtree to find the root, and a second pass down the subtree to update each vertex so that it points directly to the root. Each call returns `v.parent`. If `v` is the root, then the line `v.parent = find(v.parent)` is not executed and the root is returned. Otherwise, this line is executed and a pointer to the root is returned. This line updates `v` to point directly to the root, and this is what is returned to the caller.
Analysis of Union-Find

- Union-by-rank with path compression takes time proportional to $M \alpha(N)$, where $\alpha$ is the inverse Ackerman function
  - Compare this to $M+N$ for cycle-checking using depth-first search
  - Using the heuristics with a billion data items reduces processing time of cycle checking from 30 years to 6 seconds
  - Moral: a good algorithm sometimes enables a solution
  - A supercomputer would not help much here
Kruskal Using Union-Find

Sort the edges in non-decreasing order by weight
Initialize a partition structure (make each vertex its own root)
Choose the next edge \((u, v)\) in the order returned by the sort

\[
\text{includedCount} = 0
\]

\[
\text{while includedCount} < N \text{ do}
\]

\[
\text{root1} = \text{find}(u)
\]

\[
\text{root2} = \text{find}(v)
\]

\[
\text{if (root1} \neq \text{root2) then}
\]

\[
\text{add edge} \ (u, v) \text{ to the spanning tree so far}
\]

\[
\text{includedCount} = \text{includedCount} + 1
\]

\[
\text{union(root1,root2)}
\]

\[
\text{end if}
\]

\[
\text{end while}
\]
Question: How many connected components result after performing the following sequence of union operations on a set of 10 items, using the union-by-rank heuristic? Which items are in each connected component?

(1,2), (3,4), (5,6), (7,8), (7,9), (2,8), (0,5), (1,9)
Dynamic Connectivity

Given a set of N objects, **find out if there is a path connecting two given objects**

- Actually finding that path is a different problem

Some applications of dynamic connectivity:

- Pixels in a digital photo
- Computers in a network
- Friends in a social network
- Transistors in a computer chip
- Elements in a mathematical set
- Variable names in Fortran program
- Metallic sites in a composite system
Data Mining Using Union-Find

• Partition a set of objects into clusters such that different clusters are far apart and the objects within each cluster are close together
  – Routing in mobile ad hoc networks
  – Categorizing documents for web searches
  – Searching medical image databases for similar images
  – Clustering “sky objects” into stars, quasars, galaxies
  – Identifying patterns in gene expression

• Start by forming N clusters of one object each
  – \textbf{Find} the pair of objects from different clusters that have the minimum distance between them and \textbf{union} them into a new cluster
  – Repeat until there are exactly k clusters (k spanning trees)

• This procedure is precisely Kruskal’s algorithm