CS 362, Minimum Spanning Trees

Jared Saia
University of New Mexico
Today’s Outline

- Minimum Spanning Trees
- Safe Edge Theorem
- Kruskal and Prim’s algorithms
- Graph Representation
Graph Definition

- A graph is a pair of sets \((V, E)\).
- We call \(V\) the vertices of the graph.
- \(E\) is a set of vertex pairs which we call the edges of the graph.
- In an undirected graph, the edges are unordered pairs of vertices and in a directed graph, the edges are ordered pairs.
- We assume that there is never an edge from a vertex to itself (no self-loops) and that there is at most one edge from any vertex to any other (no multi-edges).
- \(|V|\) is the number of vertices in the graph and \(|E|\) is the number of edges.
Graph Defns

- A graph $G′ = (V′, E′)$ is a subgraph of $G = (V, E)$ if $V′ \subseteq V$ and $E′ \subseteq E$ is a set of edges over the nodes in $V'$.
- If $(u, v)$ is an edge in a graph, then $u$ is a neighbor of $v$.
- For a vertex $v$, the degree of $v$, $deg(v)$, is equal to the number of neighbors of $v$.
- A walk is a sequence of edges, where each successive pair of edges shares a vertex.
- A path is a walk, where the vertices visited are all distinct.
- A graph is connected if there is a path from any vertex to any other vertex.
- A disconnected graph consists of several connected components which are maximal connected subgraphs.
- Two vertices are in the same component if and only if there is a path between them.
Graph Defns

For undirected graphs:

- A *cycle* is a walk visiting at least 3 unique vertices that starts and ends at the same vertex, and where all vertices except the last visited are unique.
- A graph is *acyclic* if no subgraph is a cycle. Acyclic graphs are also called *forests*.
- A *tree* is a connected acyclic graph. It’s also a connected component of a forest.
- A *spanning tree* of a graph $G$ is a subgraph that is a tree and also contains every vertex of $G$. A graph can only have a spanning tree if it’s connected.
- A *spanning forest* of $G$ is a collection of spanning trees, one for each connected component of $G$.
Minimum Spanning Tree Problem

- Suppose we are given a connected, undirected weighted graph.
- That is a graph $G = (V, E)$ together with a function that assigns a weight $w(e)$ to each edge $e$.
- Our task is to find the minimum spanning tree of $G$, i.e., the spanning tree $T$ minimizing the function

$$w(T) = \sum_{e \in T} w(e)$$
Example

A weighted graph and its minimum spanning tree
Applications

- Creating an inexpensive road network to connect cities
- Wiring up homes for phone service with the smallest amount of wire
- Finding a good approximation to the TSP problem
Generic MST Algorithm

Generic-MST(G,w){
    A = {};
    while (A does not form a spanning tree){
        find an edge (u,v) that is safe for A;
        A = A union (u,v);
    }
    return A;
}
Safe edges - Definition

- Let $A$ be any set of edges in $G$ that is a subset of some MST of $G$.
- Definition: An edge $e$ is safe for $A$ if $A \cup \{e\}$ is also a subset of a MST.
Safe edges

- A **cut** \((S, V - S)\) of a graph \(G = (V, E)\) is a partition of \(V\)
- An edge \((u, v)\) **crosses** the cut \((S, V - S)\) if one of its endpoints is in \(S\) and the other is in \(V - S\)
- A cut **respects** a set of edges \(A\) if no edge in \(A\) crosses the cut.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.
Theorem 1 Let $A$ be a set of edges included in some minimum spanning tree. Then an edge $e$ is safe for $A$ if $e$ is a light edge crossing some cut that respects $A$. 
Proof

- Let $T$ be a MST that includes some set of edges $A$
- Assume that $T$ does not contain the light edge $e = (u, v)$
- Since $T$ is connected, it contains a unique path from $u$ to $v$ and at least one edge $e'$ on this path crosses the cut that respects $A$
- Note that $w(e) \leq w(e')$ by assumption
- Removing $e'$ from $T$ and adding $e$ gives us a new spanning tree $T'$
- $T'$ has total weight no more than $T$ and thus $T'$ must also be a MST. QED.
Proof that every safe edge is in some MST. The red edges are the set $A$. 
Corollary

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, and let $C = (V_c, E_c)$ be a connected component (tree) in the forest $G_A = (V, A)$. If $(u, v)$ is a light edge connecting $C$ to some other component in $G_A$, then $(u, v)$ is safe for $A$.

Proof: The cut $(V_C, V - V_C)$ respects $A$, and $(u, v)$ is a light edge for this cut. Therefore $(u, v)$ is safe for $A$. 
Two MST algorithms

- There are two major MST algorithms, Kruskal’s and Prim’s.
- In Kruskal’s algorithm, the set $A$ is a forest. The safe edge added to $A$ is always a least-weighted edge in the graph that connects two distinct components.
- In Prim’s algorithm, the set $A$ forms a single tree. The safe edge added to $A$ is always a least-weighted edge connecting the tree to a vertex not in the tree.
Q: In Kruskal’s algorithm, how do we determine whether or not an edge connects two distinct connected components?

A: We need some way to keep track of the sets of vertices that are in each connected components and a way to take the union of these sets when adding a new edge to \( A \) merges two connected components.

What we need is the data structure for maintaining disjoint sets (aka Union-Find) that we discussed last week.
Kruskal’s Algorithm

MST-Kruskal(G,w){
    for (each vertex v in V)
        Make-Set(v);
    sort the edges of E into nondecreasing order by weight;
    for (each edge (u,v) in E taken in nondecreasing order){
        if(Find-Set(u)! = Find-Set(v)){
            A = A union (u,v);
            Set-Union(u,v);
        }
    }
    return A;
}
Example Run

Kruskal’s algorithm run on the example graph. Thick edges are in $A$. Dashed edges are useless.
• Correctness of Kruskal’s algorithm follows immediately from the corollary
• Each time we add the lightest weight edge that connects two connected components, hence this edge must be safe for $A$
• This implies that at the end of the algorithm, $A$ will be a MST
• The runtime for Kruskal’s alg. will depend on the implementation of the disjoint-set data structure. We’ll assume the implementation with union-by-rank and path-compression which we showed has amortized cost of $\log^* n$

• Let $m = |E|$ and $n = |V|$. 
Runtime?

- Time to sort the edges is $O(m \log m)$
- Total time for the $n$ calls to Make-Set; and $O(m)$ calls to Find-Set and Set-Union is $O((n + m) \log^* n)$
- Since $G$ is connected, $m \geq n - 1$ and so $O((n + m) \log^* n) = O(m \log^* n) = O(m \log m)$
- Total additional work done in the for loop is $O(m)$
- Thus total runtime of the algorithm is $O(m \log m)$
- Since $m \leq n^2$, we can rewrite this as $O(m \log n)$
Prim’s Algorithm

- In Prim’s algorithm, the set $A$ maintained by the algorithm forms a single tree.
- The tree starts from an arbitrary root vertex and grows until it spans all the vertices in $V$.
- At each step, a light edge is added to the tree $A$ which connects $A$ to an isolated vertex of $G_A = (V, A)$.
- By our Corollary, this rule adds only safe edges to $A$, so when the algorithm terminates, it will return a MST.
Example Run

Prim’s algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in $A$, an arrow points along $A$’s safe edge, and dashed edges are useless.
An Implementation

- To implement Prim’s algorithm, we keep all edges adjacent to $A$ in a heap
- When we pull the minimum-weight edge off the heap, we first check to see if both its endpoints are in $A$
- If not, we add the edge to $A$ and then add the neighboring edges to the heap
- If we implement Prim’s algorithm this way, its running time is $O(m \log m) = O(m \log n)$
- However, we can do better
Prim’s Algorithm

- We can speed things up by noticing that the algorithm visits each vertex only once.
- Rather than keeping the edges in the heap, we will keep a heap of vertices, where the key of each vertex $v$ is the weight of the minimum-weight edge between $v$ and $A$ (or infinity if there is no such edge).
- Each time we add a new edge to $A$, we may need to decrease the key of some neighboring vertices.
Prim’s

We will break up the algorithm into two parts, Prim-Init and Prim-Loop

Prim(V,E,s) {
    Prim-Init(V,E,s);
    Prim-Loop(V,E,s);
}

26
Prim-Init

Prim-Init(V,E,s){
    for each vertex v in V - {s}{
        if ((v,s) is in E){
            edge(v) = (v,s);
            key(v) = w((v,s));
        }else{
            edge(v) = NULL;
            key(v) = infinity;
        }
    }
    Heap-Insert(v);
}
Heap-Insert(s);
}
Prim-Loop

Prim-Loop(V,E,s){
    A = {};
    for (i = 1 to n - 1){
        v = Heap-ExtractMin();
        add edge(v) to A;
        for (each edge (u,v) in E){
            if ((u,v) is not in A AND key(u) > w(u,v)){
                edge(u) = (u,v);
                Heap-DecreaseKey(u,w(u,v));
            }
        }
    }
    return A;
}
The runtime of Prim’s is dominated by the cost of the heap operations Insert, ExtractMin and DecreaseKey.

- Insert and ExtractMin are each called $O(n)$ times.
- DecreaseKey is called $O(m)$ times, at most twice for each edge.
- If we use a Fibonacci Heap, the amortized costs of Insert and DecreaseKey is $O(1)$ and the amortized cost of ExtractMin is $O(\log n)$.

Thus the overall run time of Prim’s is $O(m + n \log n)$.

This is faster than Kruskal’s unless $E = O(n)$. 
Note

- This analysis assumes that it is fast to find all the edges that are incident to a given vertex
- We have not yet discussed how we can do this
- This brings us to a discussion of how to represent a graph in a computer
Graph Representation

There are two common data structures used to explicitly represent graphs

- Adjacency Matrices
- Adjacency Lists
Adjacency Matrix

- The adjacency matrix of a graph $G$ is a $n \times n$ matrix of 0’s and 1’s
- For an adjacency matrix $A$, the entry $A[i, j]$ is 1 if $(i, j) \in E$ and 0 otherwise
- For undirected graphs, the adjacency matrix is always symmetric: $A[i, j] = A[j, i]$. Also the diagonal elements $A[i, i]$ are all zeros
Example Graph

\begin{center}
\begin{tikzpicture}
  \node[vertex] (a) at (0,0) {$a$};
  \node[vertex] (b) at (1,1) {$b$};
  \node[vertex] (c) at (1,-1) {$c$};
  \node[vertex] (d) at (2,0) {$d$};
  \node[vertex] (e) at (2,-1) {$e$};
  \node[vertex] (f) at (3,0) {$f$};
  \node[vertex] (g) at (4,1) {$g$};
  \node[vertex] (h) at (4,-1) {$h$};
  \node[vertex] (i) at (5,0) {$i$};

  \draw (a) -- (b) -- (c) -- (a);
  \draw (b) -- (d) -- (c);
  \draw (d) -- (e) -- (f) -- (d);
  \draw (f) -- (g) -- (h) -- (i) -- (g);
\end{tikzpicture}
\end{center}
Example Representations

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Adjacency matrix and adjacency list representations for the example graph.
• Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge.
• We can also list all the neighbors of a vertex in $\Theta(n)$ time by scanning the row corresponding to that vertex
• This is optimal in the worst case, however if a vertex has few neighbors, we still need to examine every entry in the row to find them all
• Also, adjacency matrices require $\Theta(n^2)$ space, regardless of how many edges the graph has, so it is only space efficient for very dense graphs
Adjacency Lists

- For *sparse* graphs — graphs with relatively few edges — we’re better off with adjacency lists
- An adjacency list is an array of linked lists, one list per vertex
- Each linked list stores the neighbors of the corresponding vertex
Adjacency Lists

• The total space required for an adjacency list is $O(n + m)$
• Listing all the neighbors of a node $v$ takes $O(1 + \text{deg}(v))$ time
• We can determine if $(u, v)$ is an edge in $O(1 + \text{deg}(u))$ time by scanning the neighbor list of $u$
• Note that we can speed things up by storing the neighbors of a node not in lists but rather in hash tables
• Then we can determine if an edge is in the graph in expected $O(1)$ time and still list all the neighbors of a node $v$ in $O(1 + \text{deg}(v))$ time