Graph Algorithms

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University
Minimum Spanning Trees

- Find a minimum-cost set of edges that connect all vertices of a graph

Applications
- Connecting “nodes” with a minimum of “wire”
  - Networking
  - Circuit design
- Collecting nearby nodes
  - Clustering, taxonomy construction
- Approximating graphs
  - Most graph algorithms are faster on trees
Minimum Spanning Tree

- A tree is an acyclic, undirected, connected graph.
- A spanning tree of a graph is a tree containing all vertices from the graph.
- A minimum spanning tree is a spanning tree, where the sum of the weights on the tree’s edges are minimal.
Minimum Spanning Tree

Graph:

MST:
Minimum Spanning Tree

- **Problem**
  - Given an undirected, weighted graph $G=(V,E)$ with weights $w(u,v)$ for each $(u,v) \in E$
  - Find an acyclic, connected graph $G'=(V,E')$, $E' \subseteq E$, that minimizes $\Sigma_{(u,v) \in E'} w(u,v)$
  - $G'$ is a minimum spanning tree
    - There can be more than one
Minimum Spanning Tree

- Solution #1
  - Start with an empty tree $T$
  - While $T$ is not a spanning tree
    - Find the lowest-weight edge that connects a vertex in $T$ to a vertex not in $T$
    - Add this edge to $T$
  - $T$ will be a minimum spanning tree
- Called Prim’s algorithm (1957)
Prim’s Algorithm: Example
Prim’s Algorithm

- Similar to Dijkstra’s shortest-path algorithm
- Except
  - v.known = v in T
  - v.dist = weight of lowest-weight edge connecting v to a known vertex in T
  - v.path = last neighboring vertex changing (lowering) v’s dist value (same as before)
Prim’s Algorithm

```cpp
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
        if( !w.known )
            if( v.dist + cvw < w.dist )
            {
                // Update w
                decrease( w.dist to v.dist + cvw );
                w.path = v;
            }
    }
}
```

Running time same as Dijkstra: $O(|E| \log |V|)$ using binary heaps.
Prim’s Algorithm: Example

**Graph Representation:**

- **Vertices:** \( v_1, v_2, v_3, v_4, v_5, v_6, v_7 \)
- **Edges and Weights:**
  - \( v_1 \rightarrow v_2: 1 \)
  - \( v_1 \rightarrow v_3: 4 \)
  - \( v_2 \rightarrow v_4: 3 \)
  - \( v_2 \rightarrow v_5: 10 \)
  - \( v_3 \rightarrow v_4: 2 \)
  - \( v_3 \rightarrow v_5: 8 \)
  - \( v_4 \rightarrow v_5: 6 \)
  - \( v_5 \rightarrow v_6: 1 \)

**Known Vertices Table:**

<table>
<thead>
<tr>
<th>( v )</th>
<th>known</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
</tbody>
</table>

**Known Vertices and Edges Table:**

<table>
<thead>
<tr>
<th>( v )</th>
<th>known</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>F</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>F</td>
<td>4</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>F</td>
<td>1</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
</tbody>
</table>
## Prim’s Algorithm: Example

![Graph Diagram]

<table>
<thead>
<tr>
<th>( v )</th>
<th>known</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>T</td>
<td>2</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>T</td>
<td>1</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>F</td>
<td>7</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>F</td>
<td>5</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>F</td>
<td>4</td>
<td>( v_4 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( v )</th>
<th>known</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>T</td>
<td>2</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>T</td>
<td>1</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>F</td>
<td>6</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>F</td>
<td>1</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>T</td>
<td>4</td>
<td>( v_4 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( v )</th>
<th>known</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>T</td>
<td>2</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>T</td>
<td>1</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>T</td>
<td>6</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>T</td>
<td>1</td>
<td>( v_7 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>T</td>
<td>4</td>
<td>( v_4 )</td>
</tr>
</tbody>
</table>
Minimum Spanning Tree

Solution #2
- Start with $T = V$ (no edges)
- For each edge in increasing order by weight
  - If adding edge to $T$ does not create a cycle
  - Then add edge to $T$

- $T$ will be a minimum spanning tree
- Called Kruskal’s algorithm (1956)
Kruskal’s Algorithm: Example

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_1, v_4)</td>
<td>1</td>
<td>Accepted</td>
</tr>
<tr>
<td>(v_6, v_7)</td>
<td>1</td>
<td>Accepted</td>
</tr>
<tr>
<td>(v_1, v_2)</td>
<td>2</td>
<td>Accepted</td>
</tr>
<tr>
<td>(v_3, v_4)</td>
<td>2</td>
<td>Accepted</td>
</tr>
<tr>
<td>(v_2, v_4)</td>
<td>3</td>
<td>Rejected</td>
</tr>
<tr>
<td>(v_1, v_3)</td>
<td>4</td>
<td>Rejected</td>
</tr>
<tr>
<td>(v_4, v_7)</td>
<td>4</td>
<td>Accepted</td>
</tr>
<tr>
<td>(v_3, v_6)</td>
<td>5</td>
<td>Rejected</td>
</tr>
<tr>
<td>(v_5, v_7)</td>
<td>6</td>
<td>Accepted</td>
</tr>
</tbody>
</table>
Kruskal’s Algorithm

```cpp
void Graph::kruskal()
{
    int edgesAccepted = 0;
    DisjSet ds( NUM_VERTICES );
    PriorityQueue<Edge> pq( getEdges() );
    Edge e;
    Vertex u, v;

    while( edgesAccepted < NUM_VERTICES - 1 )
    {
        pq.deleteMin( e ); // Edge e = (u, v)
        SetType uset = ds.find( u );
        SetType vset = ds.find( v );
        if( uset != vset )
        {
            // Accept the edge
            edgesAccepted++;
            ds.unionSets( uset, vset );
        }
    }
}
```

Uses Disjoint Set and Priority Queue data structures.
Kruskal’s Algorithm: Analysis

- Worst case: $O(|E| \log |E|)$
- Since $|E| = O(|V|^2)$, worst case also $O(|E| \log |V|)$
  - Running time dominated by heap operations
- Typically terminates before considering all edges, so faster in practice
Minimum Spanning Tree: Applications

- Feature extraction from remote sensing images (e.g., roads, rivers, etc.)
- Cosmological structure formation
- Cancer imaging
  - Arrangement of cells in the epithelium (tissue surrounding organs)
- Approximate solution to traveling salesman problem
- Most of above use Euclidian MST
  - I.e., weights are Euclidean distances between vertices
Minimum Spanning Trees:

Summary

- Finding set of edges that minimally connect all vertices
- Fast algorithm with many important applications
- Utilizes advanced data structures to achieve fast performance