**Weighted Graphs**

- Some applications require us to consider a cost or weight
  - costs/weights are assigned to edges
- Often use a geometric interpretation of weights
  - low weight - short edge
  - high weight - long edge
- Weights are not always geometric
  - Some weights can be negative
    - this can make some problems more difficult!
    - Assume in our graphs we have non-negative weights
Example: Weighted Graphs

- Example: “map” of airline flight routes
  - vertices = airports
  - edge = flights
  - weights = distance/time/price
WEIGHTED GRAPH IMPLEMENTATION

- Adjacency Matrix Representation
  - change 0 and 1 to float/double
  - Need a special float constant to indicate NO_EDGE
  - Can’t use 0. It may be a valid weight

- Adjacency Lists Representation
  - add float weight to each node

- This will work for directed or undirected graphs
ADJACENCY MATRIX WITH WEIGHTS

Weighted Digraph

Adjacency Matrix
Adjacency List Representation with Weights

Weighted Digraph

Adjacency Lists
WEIGHTED GRAPH PROBLEMS

- Minimum spanning tree
  - find the minimal weight set of edges that connect all vertices in a weighted graph
    - might be more than one minimal solution
  - we will assume undirected graph
  - we will assume non-negative weights

- Shortest Path Problem
  - Find minimum cost path to from one vertex to another
  - Edges may be directed or undirected
  - We will assume non-negative weights
MINIMAL SPANNING TREE PROBLEM

- Origins
  - Otakar Boruvka, electrical engineer in 1926
  - most economical construction of electric power network

- Some modern applications of MST:
  - network layout: telephone, electric, computer, road, cable

- Has been studied intensely, still looking for faster algorithms
Minimum Spanning Trees (MST)

- Reminder: Spanning tree $ST$ of graph $G(V,E)$
  - $ST$ is a subgraph of $G$
    - $(G'(V,E'))$ where $E'$ is a subset of $E$
  - $ST$ is connected and acyclic

- Minimum spanning tree $MST$ of graph $G$
  - $MST$ is a spanning tree of $G$
    - sum of edge weights is no larger than any other $ST$

- Problem: how to (efficiently) find MST for graph $G$?
Kruskal’s MST Algorithm

- One approach to computing MST for graph $G(V,E)$:
  - Start with empty MST
  - Consider edges in increasing weight order
  - Add edge if it does not form a cycle in MST
  - Repeat until $V - 1$ edges are added

- Critical operations:
  - Iterating over edges in weight order
  - Checking for cycles in a graph
EXECUTION TRACE OF KRUSKAL’S MST

Initially

After step 1

After step 2

After step 3

After step 4a

After step 4b
EXERCISE: TRACE KRUSKAL’S ALGORITHM
Kruskal’s Algorithm: Minimal Spanning Tree

- **Implementation 1: Two main parts:**
  - sorting edges according to their length \( (E \times \log E) \)
  - check if adding an edge would create a cycle
    - Could check for cycles using DFS ... but too expensive
    - use Union-Find data structure from Sedgewick ch.1
      - If we use this the cost of sorting dominates so over all
        - \( E \log E \)

- **Implementation 2: Using a pq instead of full sort**
  - Create a priority queue using weights as priority
  - Allows us to remove edges from pq in weighted order
  - \( O(E + X \times \log V) \), with \( X \) = number of edges shorter than the longest edge in the MST
PRIM’S ALGORITHM: MINIMAL SPANNING TREE

Another approach to computing MST for graph $G(V,E)$:

- start from any vertex $s$ and empty MST
  - choose edge not already in MST to add to MST
    - must not contain a self-loop
    - must connect to a vertex already on MST
    - must have minimal weight of all such edges
  - check to see whether adding the new edge brought any of the non-tree vertices closer to the tree
  - repeat until MST covers all vertices

Critical operations:
- checking for vertex being connected in a graph
- finding min weight edge in a set of edges
- updating min weights in a set of edges
**Prim’s MST Algorithm**

- **Idea:**
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree

```
• 0 – 1 (32)
• 0 – 2 (29)
• 0 – 5 (60)
• 0 – 6 (51)
• 0 – 7 (31)
• 1 – 7 (21)
• 3 – 4 (34)
• 3 – 5 (18)
• 4 – 5 (40)
• 4 – 6 (51)
• 4 – 7 (46)
• 6 – 7 (25)
```
**Prim’s MST Algorithm**

- **Idea:**
  - Starting from a sub-graph containing only one vertex, we successively add the shortest vertex connecting the sub-graph with the rest of the nodes to the tree.
  - Edges in pink are in the fringe.

```
0 – 1 (32)
0 – 2 (29)
0 – 5 (60)
0 – 6 (51)
0 – 7 (31)
1 – 7 (21)
3 – 4 (34)
3 – 5 (18)
4 – 5 (40)
4 – 6 (51)
4 – 7 (46)
6 – 7 (25)
```
**Prim’s MST Algorithm**

**Idea:**
- Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree.
- Edges in pink are in the fringe.
- Edges in black bold are in the MST.

![Graph with Prim's MST Algorithm](image)

- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 – 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)
**Weighted Graph: Minimal Spanning Tree II**

- **Idea:**
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree.
  - Edges in pink are in the fringe.
  - Edges in black bold are in the MST.

```
0 – 1 (32)
0 – 2 (29)
0 – 5 (60)
0 – 6 (51)
0 – 7 (31)
1 – 7 (21)
3 – 4 (34)
3 – 5 (18)
4 – 5 (40)
4 – 6 (51)
4 – 7 (46)
6 – 7 (25)
```
**Weighted Graph: Minimal Spanning Tree II**

**Idea:**

- Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree.
- Edges in pink are in the fringe.
- Edges in black bold are in the MST.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 1</td>
<td>(32)</td>
</tr>
<tr>
<td>0 – 2</td>
<td>(29)</td>
</tr>
<tr>
<td>0 – 5</td>
<td>(60)</td>
</tr>
<tr>
<td>0 – 6</td>
<td>(51)</td>
</tr>
<tr>
<td>0 – 7</td>
<td>(31)</td>
</tr>
<tr>
<td>1 – 7</td>
<td>(21)</td>
</tr>
<tr>
<td>3 – 4</td>
<td>(34)</td>
</tr>
<tr>
<td>3 – 5</td>
<td>(18)</td>
</tr>
<tr>
<td>4 – 5</td>
<td>(40)</td>
</tr>
<tr>
<td>4 – 6</td>
<td>(51)</td>
</tr>
<tr>
<td>4 – 7</td>
<td>(46)</td>
</tr>
<tr>
<td>6 – 7</td>
<td>(25)</td>
</tr>
</tbody>
</table>
**Weighted Graph: Minimal Spanning Tree II**

- **Idea:**
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree.
  - Edges in pink are in the fringe.
  - Edges in black bold are in the MST.

![Graph Diagram]

- Edges and their weights:
  - 0 – 1 (32)
  - 0 – 2 (29)
  - 0 – 5 (60)
  - 0 – 6 (51)
  - 0 – 7 (31)
  - 1 – 7 (21)
  - 3 – 4 (34)
  - 3 – 5 (18)
  - 4 – 5 (40)
  - 4 – 6 (51)
  - 4 – 7 (46)
  - 6 – 7 (25)
PRIM’S MST ALGORITHM

Idea:

- Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree.
- Edges in pink are in the fringe.
- Edges in black bold are in the MST.

Edges:

- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 – 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)
**Prim’s MST Algorithm**

- **Idea:**
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree.
  - Edges in pink are in the fringe.
  - Edges in black bold are in the MST.

- Edges and their weights:
  - 0 – 1 (32)
  - 0 – 2 (29)
  - 0 – 5 (60)
  - 0 – 6 (51)
  - 0 – 7 (31)
  - 1 – 7 (21)
  - 3 – 4 (34)
  - 3 – 5 (18)
  - 4 – 5 (40)
  - 4 – 6 (51)
  - 4 – 7 (46)
  - 6 – 7 (25)
Prím’s MST Algorithm

Idea:
- Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
- Edges in pink are in the fringe
- Edges in black bold are in the MST

0 – 1 (32)
0 – 2 (29)
0 – 5 (60)
0 – 6 (51)
0 – 7 (31)
1 – 7 (21)
3 – 4 (34)
3 – 5 (18)
4 – 5 (40)
4 – 6 (51)
4 – 7 (46)
6 – 7 (25)
**Prim’s Algorithm**

- Prim’s algorithm is just a graph search –
  - instead of depth first (using a stack) or breadth first (using a queue),
    - we choose a shortest first strategy using a priority queue

- It can be implemented to run in
  - \(O(E \times \log V)\) steps
    - if the steps listed above are implemented efficiently (using adjacency lists and heap),
  - \(O(V^2)\) for adjacency matrix

- See lecture code for an implementation
EXERCISE: TRACE PRIM’S ALGORITHM

Diagram of a network with weighted edges.
Shortest Paths

- Weight of a path $p$ in graph $G$
  - sum of weights on edges along path ($\text{weight}(p)$)
- Shortest path between vertices $s$ and $t$
  - a simple path $p$ where $s = \text{first}(p)$, $t = \text{last}(p)$
  - no other simple path $q$ has $\text{weight}(q) < \text{weight}(p)$
- Problem: how to (efficiently) find $\text{shortestPath}(G,s,t)$?
  - Assumptions: weighted graph, no negative weights.
EXERCISE:

- What is the minimum spanning tree?
- What is the shortest path from 0 to 3?
- What is the least hops path (shortest unweighted path) from 0 to 2?
**Shortest Path Algorithms**

- Shortest-path is useful in a wide range of applications
  - robot navigation
  - finding routes in maps
  - routing in data/computer networks

- Flavours of shortest-path
  - source-target (shortest path from $s$ to $t$)
  - single-source (shortest paths from $s$ to all other $V$)
  - all-pairs (shortest paths for all $(s,t)$ pairs)
Dijkstra’s Algorithm
Single Source Shortest Paths

Weighted Digraph

Shortest paths from $s=0$
**Dijkstra’s Algorithm:**
**Single Source Shortest Path**

- **Given:**
  - weighted digraph/graph $G$, source vertex $s$

- **Result:**
  - shortest paths from $s$ to **all** other vertices
  - $\text{dist}[]$: $V$-indexed array of distances from $s$
  - $\text{st}[]$: $V$-indexed array of predecessors in shortest path

- **Note:** shortest paths can be viewed as tree rooted at $s$
**Edge Relaxation**

- Relaxation along edge $e$ from $v$ to $w$
  - $\text{dist}[v]$ is length of some path from $s$ to $v$
  - $\text{dist}[w]$ is length of some path from $s$ to $w$
  - if $e$ gives shorter path $s$ to $w$ via $v$, then update $\text{dist}[w]$ and $\text{st}[w]$

- Relaxation updates data on $w$ if we find a shorter path to $s$.

```java
if (\text{dist}[v] + e\text{.weight} < \text{dist}[w]) {
    \text{dist}[w] = \text{dist}[v] + e\text{.weight};
    \text{st}[w] = v;
}
```
**Dijkstra’s Algorithm**

- **Data:**
  - $G$, $s$, $dist[\cdot]$, $st[\cdot]$, and a $pq$ containing the set of vertices whose shortest path from $s$ is not yet known

- **Algorithm:**
  - initialise $dist[\cdot]$ to all $\infty$, except $dist[s]=0$
  - Initialise $pq$ with all $V$, with $dist[v]$ as priority
  - $v = \text{deleteMin from } pq$
    - Get e’s that connect $v$ to $w$ in $pq$
    - relax along $e$ if new dist is better
  - repeat until $pq$ is empty
**Execution Trace of Dijkstra’s Algorithm**

Initially

After 0 visited

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dist</strong></td>
<td>0</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td><strong>st</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>pq</strong></td>
<td>{0,1,2,3,4}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dist</strong></td>
<td>0</td>
<td>0.3</td>
<td>inf</td>
<td>0.7</td>
<td>inf</td>
</tr>
<tr>
<td><strong>st</strong></td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td><strong>pq</strong></td>
<td>{1,2,3,4}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
...Execution Trace of Dijkstra’s Algorithm

After 1 visited

After 2 visited

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist</td>
<td>0</td>
<td>0.3</td>
<td>0.7</td>
<td>0.7</td>
<td>inf</td>
</tr>
<tr>
<td>st</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>pq</td>
<td>{2,3,4}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist</td>
<td>0</td>
<td>0.3</td>
<td>0.7</td>
<td>0.7</td>
<td>1.2</td>
</tr>
<tr>
<td>st</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>pq</td>
<td>{3,4}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
...Execution Trace of Dijkstra's Algorithm

After 3 visited

After 4 visited

dist
[0] [1] [2] [3] [4]
0   0.3  0.7  0.7  1.2
st  -   0   1   0   2
pq  {4}

dist
[0] [1] [2] [3] [4]
0   0.3  0.7  0.7  1.2
st  -   0   1   0   2
pq  {}
Dijkstra’s Results

- After the algorithm has completed:
  - Shortest Path distances are in dist array
  - Actual path can be traced back from endpoint via the predecessors in the st array
Assume we have just completed running Dijkstra’s algorithm with starting vertex v. Write code to print out the path from vertex v to w or “No path” if the path does not exist. (It is ok to print it in reverse order.)
TRACE EXECUTION OF DIJKSTRA’S ALGORITHM FROM STARTING VERTEX 2