These are computers and costs of direct connections. What is a cheapest way to network them?
(Edge-)Weighted Graph

Many useful graphs have numbers attached to edges. Think of: each edge has a price tag. (Usually $\geq 0$. Some cases have $< 0$.)

A weighted (edge-weighted) graph consists of:

- a set of vertices
- a set of edges
- a map from edges to numbers
Storing a Weighted Graph

Adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency list:

<table>
<thead>
<tr>
<th>is adjacent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  (B,4), (C,2)</td>
</tr>
<tr>
<td>B  (A,4), (C,1), (D,5)</td>
</tr>
<tr>
<td>C  (A,2), (B,1)</td>
</tr>
<tr>
<td>D  (B,5)</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>
Common Task #1 on Weighted Graphs

Shortest path:

- Input two vertices, find a simple path between them (if any).
- Minimize the sum of the weights of the edges used.

Between $A$ and $D$:

$\langle A, C, B, D \rangle$ is a shortest path. Total weight 8.

$\langle A, B, D \rangle$ is not a shortest path. Total weight 9.
Common Task #2 on Weighted Graphs

Minimum spanning tree:

- Find a spanning tree (if any).
- Minimize the sum of the weights of the edges used.
Finding a Minimum Spanning Tree Incrementally

I have a partial answer to minimum spanning tree. Can I add the cheapest candidate and still get a partial answer?
A Theorem about Minimum Spanning Trees

Assume

1. have a minimum spanning tree, its edges are $T$
2. have $S \subseteq T$
3. the vertices are partitioned into $V_1, V_2$
4. every edge $\{x, y\} \in S$: either $x, y \in V_1$ or $x, y \in V_2$
   “no edge in $S$ crosses the partition”
5. have edge $e = \{u, v\}$, and $u \in V_1, v \in V_2$
   “$e$ crosses the partition”
6. every edge crossing the partition has weight $\geq w(e)$

then

- there exists a minimum spanning tree, its edges are $T'$
- $\{e\} \cup S \subseteq T'$
Proof Sketch

If $e \in T$ already, choose $T' = T$, done.

If not:

7. by 1, some edge $e_T \in T$ crosses the partition
8. choose $T' = (T \setminus \{e_T\}) \cup \{e\}$
   (take $T$ but replace edge $e_T$ by $e$)
9. check: $T'$ are the edges of some spanning tree
10. by 6&7, total weight of $T' \leq$ total weight of $T$
    by 1&9, $T'$ are the edges of some minimum spanning tree
11. by 2&4&8, $\{e\} \cup S \subseteq T'$
Remark on The Theorem

This is an improvement over Theorem 7.10 of the textbook.

Theorem 7.10 is not enough. It does not guarantee: the “some minimum spanning tree” that has $e$ also reuses $S$.

Reusing $S$ is important because it stands for a partial answer, you can’t ignore it.

I found the improved theorem in another textbook\textsuperscript{1}. It is longer, but it is necessary to guarantee reuse of partial answers.

\textsuperscript{1}by Cormen, Leiserson, Rivest, and Stein
Prim’s Algorithm: Idea

Prim’s algorithm finds a minimum spanning tree by something like breadth-first search, but with a twist:

The queue is changed to a priority queue (the min version).

Priority of vertex = smallest edge weight between the vertex and visited vertices. ($\infty$ if no such edge.)

Also, remember which edges have those smallest weights, they will likely be part of the final answer.
(Or remember the predecessor.)
Prim’s Algorithm: Small Example

![Graph with nodes A, B, C, D and edges labeled with weights 2, 1, 4, 5]

<table>
<thead>
<tr>
<th>priority</th>
<th>vertex</th>
<th>edge</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>
### Prim’s Algorithm: Small Example

**Figure:**
- **Vertices:** A, B, C, D
- **Edges:**
  - {A, C} with weight 2
  - {A, B} with weight 4
  - {B, D} with weight 5

**Table:**

<table>
<thead>
<tr>
<th>priority</th>
<th>vertex</th>
<th>edge</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>C</td>
<td>{A, C}</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>{A, B}</td>
<td>A</td>
</tr>
<tr>
<td>∞</td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prim’s Algorithm: Small Example

A --- 2 --- C
  |
  v
B

4

vertex: B, D
edge: \{C, B\}
pred: C

priority

vertex

edge

pred

1

\infty

B

D

C
Prim’s Algorithm: Small Example

![Graph Diagram]

<table>
<thead>
<tr>
<th>priority</th>
<th>vertex</th>
<th>edge</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>D</td>
<td>{B, D}</td>
<td>B</td>
</tr>
</tbody>
</table>
Prim’s Algorithm: Small Example

- **Vertices**: A, B, C, D
- **Edges**:
  - A to B:优先级4
  - B to C:优先级1
  - C to D:优先级5

**Table**:

<table>
<thead>
<tr>
<th>priority</th>
<th>vertex</th>
<th>edge</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prim’s Algorithm

S = empty graph
Q = new PriorityQueue()
start = pick a vertex
Q.insert(0, (start, null))
for each vertex v ≠ start: Q.insert(∞, (v, null))
while not Q.empty():
    (u, e) = Q.removeMin()
    add vertex u, edge e to S
    for each z adjacent to u:
        if z in Q && w({u,z}) < priority of z in Q:
            change Q:
            priority of z is now w({u,z})
            value of z is now (z, {u,z})
            // predecessor of z is now u
return S
Sketch Correctness of Prim’s Algorithm

Before each iteration:

- $S$ is part of some minimum spanning tree
- $V_2 = \text{vertices in } Q \text{ (unvisited vertices)}$
- $V_1 = \text{vertices in } S \text{ (visited vertices)}$
- no edge in $S$ crosses the partition (they are all on $V_1$’s side)
- $e$ crosses the partition
- $w(e)$ is minimum among all edges crossing the partition ($Q$ keeps track of them)

Then by the theorem, $\{e\} \cup S$ is part of some minimum spanning tree. It is correct to update $S$ by adding $e$.

$u$ is dequeued from $Q$. No edge in new $S$ crosses the new partition.

$Q$ is updated for new crossing edges.
Prim’s Priority Queue Requirement

Prim’s algorithm requires these extra operations on priority queues:

- find whether a vertex is in Q; if yes:
  - find its priority
  - change its priority and value

There is a way to track positions of nodes in a heap as they move.

- find whether a vertex is in Q: $\Theta(1)$
- find its priority, change its value: $\Theta(1)$
- change its priority: $\Theta(\log |V|)$ because you still have to bubble up or down
Prim’s Algorithm Time

If you use the easy way to store clusters:

- every vertex is enqueued and dequeued once: $\Theta(\log |V|)$ each
- every edge may trigger a change of priority: $\Theta(\log |V|)$ each
- the rest is $\Theta(1)$ per vertex

Total $\Theta((|V| + |E|) \cdot \log |V|)$ time worst case.
Kruskal’s Algorithm: Idea

Kruskal’s algorithm finds a minimum spanning tree by successive mergers.

1. At first, each vertex is its own small company (cluster in textbook).
2. Find an edge of minimum weight, use it to merge two companies into one.
3. Do it again...
4. In general, find an edge of minimum weight that crosses two companies; merge them into one.

(This will require: from a vertex, look up which company it belongs to.)
Kruskal’s Algorithm: A Few Example Steps
Kruskal’s Algorithm: A Few Example Steps
Kruskal’s Algorithm: A Few Example Steps
Kruskal’s Algorithm: A Few Example Steps
Kruskal’s Algorithm: A Few Example Steps
Kruskal’s Algorithm

S = empty graph
Q = new PriorityQueue()
for each edge e: Q.insert(w(e), e)
for each vertex v: v.cluster = {v}
while S has fewer than |V|-1 edges:
    {u,v} = Q.removeMin()
    if u.cluster ≠ v.cluster:
        add u, v, {u,v} to S
        merge u.cluster and v.cluster
return S
Sketch Correctness of Kruskal’s Algorithm

Before each iteration:

- \( S \) is part of some minimum spanning tree
- \( V_1 = u \)'s cluster
- \( V_2 = \) other vertices (includes \( v \)'s cluster)
- no edge in \( S \) crosses the partition
- \( \{u, v\} \) crosses the partition
- \( w(\{u, v\}) \) is minimum among all edges crossing the partition (minimum among all unconsidered edges, which are in \( Q \))

Then by the theorem, after adding \( \{u, v\} \) to \( S \), new \( S \) is part of some minimum spanning tree.

\( u \)'s and \( v \)'s clusters are merged. No edge in new \( S \) crosses future partitions.
Storing Clusters: Easy Way

The textbook suggests two ways to implement clusters.

Easy way:

- each cluster is a linked list
- each vertex has a pointer to its owning cluster
- $u.cluster \neq v.cluster$ is pointer equality, $\Theta(1)$ time
- merging two clusters is merging two linked lists, BUT:
  a lot of vertices need their pointers updated

Luckily, if you move the smaller list to the larger one, then:

- whenever $v.cluster$ needs update, cluster size doubles
- $v.cluster$ is updated at most $\log |V|$ times, ever
Kruskal’s Algorithm Time

- each edge is enqueued and dequeued once: \( \Theta(\log |E|) \) each
- each vertex faces cluster updates: \( \Theta(\log |V|) \) each
- the rest is \( \Theta(1) \) per vertex or edge

Total \( \Theta(|V| \cdot \log |V| + |E| \cdot \log |E|) \) time worst case.

Note \( \log |E| \in O(\log |V|) \).

\( O((|V| + |E|) \cdot \log |V|) \) time.
Storing Clusters: Advanced Way

An advanced way to store clusters: “union-find”. Supports:

- from a member, **find** its owning cluster
- **union** two clusters

Basic idea:

- Each cluster is a directed tree. Members are vertices.
- Edges go from children to parents. (So, only parent pointers.)
Union-Find: Find

From member find cluster: find root.

- d’s root is g. i’s root is g. d and i are in the same cluster.
- b’s root is b. d and b are in different clusters.

Improvement: “path compression”: after finding a member’s root, change member’s parent pointer to point to root directly. Future finds will be fast.
Union-Find: Union

Union two clusters: have one cluster’s root point to a member of the other cluster.

Union c’s cluster with h’s cluster: have c’s root point to h.

Improvement: merge smaller cluster into larger cluster.
Trailers

Kruskal’s algorithm is faster using union-find.

Speed of union-find will be solved in the next few weeks!

Graph problems and algorithms will return in C63 and C78!