Weighted Graphs

Definition 1
A *weighted* (directed or undirected graph) is a pair $(G, W)$ consisting of a graph $G = (V, E)$ and a *weight function* $W : E \to \mathbb{R}$.

In this lecture, we always assume that *weights are non-negative*, i.e., that $W(e) \geq 0$ for all $e \in E$.

Example
Representations of Weighted Graphs (as Matrices)

Adjacency Matrix

\[
\begin{pmatrix}
0 & 2.0 & 0 & 0 & 0 & 9.0 & 5.0 & 0 & 0 \\
2.0 & 0 & 4.0 & 0 & 0 & 0 & 6.0 & 0 & 0 \\
0 & 4.0 & 0 & 2.0 & 0 & 0 & 0 & 5.0 & 0 \\
0 & 0 & 2.0 & 0 & 1.0 & 0 & 0 & 1.0 & 0 \\
0 & 0 & 0 & 1.0 & 0 & 6.0 & 0 & 0 & 3.0 \\
9.0 & 0 & 0 & 0 & 6.0 & 0 & 0 & 0 & 1.0 \\
5.0 & 6.0 & 0 & 0 & 0 & 0 & 0 & 5.0 & 2.0 \\
0 & 0 & 5.0 & 1.0 & 0 & 0 & 5.0 & 0 & 4.0 \\
0 & 0 & 0 & 3.0 & 1.0 & 2.0 & 4.0 & 0 & 0
\end{pmatrix}
\]
Representations of Weighted Graphs (Adjacency List)

Adjacency Lists

```
A  → B  2.0  → G  5.0  → F  9.0
B  → C  4.0  → G  6.0  → A  2.0
C  → D  2.0  → H  5.0  → B  4.0
D  → C  2.0  → H  1.0  → E  1.0
E  → D  1.0  → I  3.0  → F  6.0
F  → A  9.0  → I  1.0  → E  6.0
G  → A  5.0  → B  6.0  → H  5.0  → I  2.0
H  → G  5.0  → C  5.0  → D  1.0  → I  4.0
I  → F  1.0  → G  2.0  → H  4.0  → E  3.0
```
Connecting Sites

Problem

Given a collection of sites and costs of connecting them, find a minimum cost way of connecting all sites.

Our Graph Model

- Sites are vertices of a weighted graph, and (non-negative) weights of the edges represent the cost of connecting their endpoints.
- It is reasonable to assume that the graph is undirected and connected.
- The cost of a subgraph is the sum of the costs of its edges.
- The problem is to find a subgraph of minimum cost that connects all vertices.
Spanning Trees

\( G = (V, E) \) undirected connected graph and \( W \) weight function.
\( H = (V^H, E^H) \) with \( V^H \subseteq V \) and \( E^H \subseteq E \) subgraph of \( G \).

- The \textit{weight} of \( H \) is the number

\[
W(H) = \sum_{e \in E^H} W(e).
\]

- \( H \) is a \textit{spanning subgraph} of \( G \) if \( V^H = V \).

Observation 2

A connected spanning subgraph of minimum weight is a tree.
Minimum Spanning Trees

$(\mathcal{G}, W)$ undirected connected weighted graph

**Definition 3**
A **minimum spanning tree (MST)** of $\mathcal{G}$ is a connected spanning subgraph $\mathcal{T}$ of $\mathcal{G}$ of minimum weight.

The **minimum spanning tree problem**:

Given: *Undirected connected weighted graph* $(\mathcal{G}, W)$
Output: *An MST of* $\mathcal{G}$
Prim’s Algorithm

Idea

“Grow” an MST out of a single vertex by always adding “fringe” (neighbouring) edges of minimum weight.

A fringe edge for a subtree $T$ of a graph is an edge with exactly one endpoint in $T$ (so $e = (u, v)$ with $u \in T$ and $v \notin T$).

Algorithm $\text{PRIM}(G, W)$

1. $T \leftarrow$ one vertex tree with arbitrary vertex of $G$
2. while there is a fringe edge do
3. add fringe edge of minimum weight to $T$
4. return $T$

Note that this is another use of the greedy strategy.
Example

ADS: lects 14 & 15 – slide 9 –
Correctness of Prim’s algorithm

1. Throughout the execution of PRIM, \( T \) remains a tree.

   *Proof:* To show this we need to show that throughout the execution of the algorithm, \( T \) is (i) always connected and (ii) never contains a cycle.

   (i) Only edges with an endpoint in \( T \) are added to \( T \), so \( T \) remains connected.

   (ii) We never add any edge which has *both* endpoints in \( T \) (we only allow a single endpoint), so the algorithm will never construct a cycle.
Correctness of Prim’s algorithm (cont’d)

2. All vertices will eventually be added to $T$.

Proof: by contradiction ... (depends on our assumption that the graph $G$ was connected.)

- Suppose $w$ is a vertex that never gets added to $T$ (as usual, in proof by contradiction, we suppose the opposite of what we want).
- Let $v = v_0 e_1 v_1 e_2 ... v_n = w$ be a path from some vertex $v$ inside $T$ to $w$ (we know such a path must exist, because $G$ is connected). Let $v_i$ be the first vertex on this path that never got added to $T$.
- After $v_{i-1}$ was added to $T$, $e_i = (v_{i-1}, v_i)$ would have become a fringe edge. Also, it would have remained as a fringe edge unless $v_i$ was added to $T$.
- So eventually $v_i$ must have been added, because Prims algorithm only stops if there are no fringe edges. So our assumption was wrong. So we must have $w$ in $T$ for every vertex $w$. 

ADS: lects 14 & 15 – slide 11 –
Correctness of Prim’s algorithm (cont’d)

3. Throughout the execution of PRIM, \( T \) is contained in some MST of \( G \).

   Proof: (by Induction)
   
   ▶ Suppose that \( T \) is contained in an MST \( T' \) and that fringe edge \( e = (x, y) \) is then added to \( T \) by PRIM. We shall prove that \( T + e \) is contained in some MST \( T'' \) (not necessarily \( T' \)).
   
   ▶ case (i): If \( e \) is contained in \( T' \), our proof is easy, we simply let \( T'' = T' \).
   
   ▶ case (ii): Otherwise, if \( e \not\in T' \), consider the unique path \( P \) from \( x \) to \( y \) in \( T' \) (\( P \) is the pink path in the example overleaf). Then \( P \) contains \textit{exactly one} fringe edge \( e' = (x', y') \) (same names in example).
Correctness of Prim's algorithm (cont'd)

Define $T''$ to be $T' + (x,y) - (x',y')$  
("drop $(x',y')$ and add $(x,y)$")
3. case (ii) cont’d

- Then $W(e) \leq W(e')$.
  (otherwise $e'$ would definitely have been added before $e$)
- Let $T'' = T' + e - e'$.
- $T''$ is a tree.
  Why? Well, we drop $e' = (x', y')$, which splits the global MST $T$ into two components: $T'_x$, and the other subtree $T'_y = T' \setminus T'_x$.
  We know $x$ and $y$ are now in different components after this split, because we have broken the unique path $P$ between $x$ and $y$ in $T'$.
  Hence we can add $e = (x, y)$ to re-join $T'_x$, and $T'_y$, without making a cycle.
  $T''$ has the same vertices as $T'$, thus it is a spanning tree.
- Moreover, $W(T'') = W(T') + W(e) - W(e')$, and because we know $W(e) \leq W(e')$, this gives $W(T'') \leq W(T')$, thus $T''$ is also a MST.
Towards an Implementation

Improvement

- Instead of fringe edges, we think about adding fringe vertices to the tree.
- A fringe vertex is a vertex $y$ not in $\mathcal{T}$ that is an endpoint of a fringe edge.
- The weight of a fringe vertex $y$ is

$$\min\{W(e) \mid e = (x, y) \text{ a fringe edge}\}$$

(ie, the best weight that could “bring $y$ into the MST”)

- To be able to recover the tree, every time we “bring a fringe vertex $y$ into the tree”, we store its parent in the tree.

We will store the fringe vertices in a priority queue.
Priority Queues with Decreasing Key

A Priority Queue is an ADT for storing a collection of elements with an associated key. The following methods are supported:

- **Insert**\((e, k)\): Insert element \(e\) with key \(k\).
- **Get-Min\(): Return an element with minimum key; an error occurs if the priority queue is empty.
- **Extract-Min\(): Return and remove an element with minimum key; an error if the priority queue is empty.
- **Is-Empty\(): Return *true* if the priority queue is empty and *false* otherwise.

To update the keys during the execution of Prim, we need priority queues supporting the following additional method:

- **Decrease-Key**\((e, k)\): Set the key of \(e\) to \(k\) and update the priority queue. It is assumed that \(k\) is smaller than or equal to the old key of \(e\).
Implementation of Prim’s Algorithm

**Algorithm** \( \text{PRIM}(G, W) \)

1. Initialise parent array \( \pi \):
   \[ \pi[v] \leftarrow \text{NIL} \text{ for all vertices } v \]
2. Initialise weight array:
   \[ \text{weight}[v] \leftarrow \infty \text{ for all } v \]
3. Initialise inMST array:
   \[ \text{inMST}[v] \leftarrow \text{false} \text{ for all } v \]
4. Initialise priority queue \( Q \)
5. \( v \leftarrow \text{arbitrary vertex of } G \)
6. \( Q.\text{INSERT}(v, 0) \)
7. \( \text{weight}[v] = 0; \)
8. while not \( Q.\text{IS-EMPTY}() \) do
9. \( y \leftarrow Q.\text{EXTRACT-MIN}() \)
10. \( \text{inMST}[y] \leftarrow \text{true} \)
11. for all \( z \) adjacent to \( y \) do
12. \( \text{RELAX}(y,z) \)
13. return \( \pi \)

**Algorithm** \( \text{RELAX}(y, z) \)

1. \( w \leftarrow W(y, z) \)
2. if \( \text{weight}[z] = \infty \) then
3. \( \text{weight}[z] \leftarrow w \)
4. \( \pi[z] \leftarrow y \)
5. \( Q.\text{INSERT}(z, w) \)
6. else if \( (w < \text{weight}[z] \text{ and} \quad \text{not (inMST}[z])) \) then
7. \( \text{weight}[z] \leftarrow w \)
8. \( \pi[z] \leftarrow y \)
9. \( Q.\text{DECREASE KEY}(z, w) \)
Analysis of Prims’s algorithm

Let $n$ be the number of vertices and $m$ the number of edges of the input graph.

- Lines 1-7, 13 of Prim require $\Theta(n)$ time altogether.
- $Q$ will extract each of the $n$ vertices of $G$ once. Thus the loop at lines 8-12 is iterated $n$ times.

Thus, disregarding (for now) the time to execute the inner loop (lines 11-12) the execution of the loop requires time

$$\Theta\left(n \cdot T_{\text{Extract-Min}}(n)\right)$$

- The inner loop is executed at most once for each edge (and at least once for each edge). So its execution requires time

$$\Theta\left(m \cdot T_{\text{Relax}}(n, m)\right).$$
Analysis of **Prim’s algorithm** (**Relax**)

- Decreasing the time needed to execute **Insert** and **Decrease-Key**, the execution of **Relax** requires time $\Theta(1)$.
- **Insert** is executed once for every vertex, which requires time
  \[ \Theta(n \cdot T_{\text{Insert}}(n)) \]
- **Decrease-Key** is executed at most once for every edge. This can require time of size
  \[ \Theta(m \cdot T_{\text{Decrease-Key}}(n)) \]

Overall, we get

\[
T_{\text{Prim}}(n, m) = \Theta\left(n\left(T_{\text{Extract-Min}}(n) + T_{\text{Insert}}(n)\right) + mT_{\text{Decrease-Key}}(n)\right)
\]
Priority Queue Implementations

- **Array**: Elements simply stored in an array.
- **Heap**: Elements are stored in a binary heap (see Inf2B (ADS note 7), [CLRS] Section 6.5)
- **Fibonacci Heap**: Sophisticated variant of the simple binary heap (see [CLRS] Chapters 19 and 20)

<table>
<thead>
<tr>
<th>method</th>
<th>Array</th>
<th>Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INSERT</strong></td>
<td>Θ(1)</td>
<td>Θ(lg n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td><strong>EXTRACT-MIN</strong></td>
<td>Θ(n)</td>
<td>Θ(lg n)</td>
<td>Θ(lg n)</td>
</tr>
<tr>
<td><strong>DECREASE-KEY</strong></td>
<td>Θ(1)</td>
<td>Θ(lg n)</td>
<td>Θ(1) (amortised)</td>
</tr>
</tbody>
</table>
Running-time of $\text{Prim}$

$$T_{\text{Prim}}(n, m) = \Theta \left( n \left( T_{\text{Extract-Min}}(n) + T_{\text{Insert}}(n) \right) + m T_{\text{Decrease-Key}}(n) \right)$$

Which Priority Queue implementation?

- With array implementation of priority queue:
  $$T_{\text{Prim}}(n, m) = \Theta(n^2).$$

- With heap implementation of priority queue:
  $$T_{\text{Prim}}(n, m) = \Theta((n + m) \log(n)).$$

- With Fibonacci heap implementation of priority queue:
  $$T_{\text{Prim}}(n, m) = \Theta(n \log(n) + m).$$

($n$ being the number of vertices and $m$ the number of edges)
Remarks

- The Fibonacci heap implementation is mainly of theoretical interest. It is not much used in practice because it is very complicated and the constants hidden in the $\Theta$-notation are large.
- For dense graphs with $m = \Theta(n^2)$, the array implementation is probably the best, because it is so simple.
- For sparser graphs with $m \in O\left(\frac{n^2}{\lg n}\right)$, the heap implementation is a good alternative, since it is still quite simple, but more efficient for smaller $m$.
  Instead of using binary heaps, the use of $d$-ary heaps for some $d \geq 1$ can speed up the algorithm (see [Sedgewick] for a discussion of practical implementations of Prim's algorithm).
Reading Assignment

[CLRS] Chapter 23.

Problems

1. Exercises 23.1-1, 23.1-2, 23.1-4 of [CLRS]
2. In line 3 of Prim’s algorithm, there may be more than one fringe edge of minimum weight. Suppose we add all these minimum edges in one step. Does the algorithm still compute a MST?
3. Prove that our implementation of Prim’s algorithm on slide 6 is correct - ie, that it computes an MST. What is the difference between this and the suggested algorithm of Problem 4?