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 \(W : E \rightarrow \mathbb{R}\).

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

Example
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
\( \mathcal{G} = (V, E) \) undirected connected graph and \( W \) weight function.

\( \mathcal{H} = (V^H, E^H) \) with \( V^H \subseteq V \) and \( E^H \subseteq E \) subgraph of \( \mathcal{G} \).

► The weight of \( \mathcal{H} \) is the number \( W(\mathcal{H}) = \sum_{e \in E^H} W(e) \).

► \( \mathcal{H} \) is a spanning subgraph of \( \mathcal{G} \) 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 \( \mathcal{T} \) of a graph is an edge with exactly one endpoint in \( \mathcal{T} \) (so \( e = (u, v) \) with \( u \in \mathcal{T} \) and \( v \notin \mathcal{T} \)).

Algorithm \( \text{PRIM}(\mathcal{G}, W) \)
1. \( \mathcal{T} \leftarrow \) one vertex tree with arbitrary vertex of \( \mathcal{G} \)
2. \textbf{while} there is a fringe edge \textbf{do}
3. \hspace{1em} add fringe edge of minimum weight to \( \mathcal{T} \)
4. \textbf{return} \( \mathcal{T} \)

Note that this is another use of the greedy strategy.
Correctness of Prim's algorithm

1. Throughout the execution of Prim, \( T \) remains a tree.
   
   \textit{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 \( T \).
   
   \textit{Proof:} by contradiction ... (depends on our assumption that the graph \( G \) was connected.)
   
   ▶ Suppose \( w \) is a vertex that \textit{never} gets added to \( T \) (as usual, in proof by contradiction, we suppose the \textit{opposite} of what we want).
   
   ▶ Let \( v = v_0e_1v_1e_2...v_n = w \) 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 \textit{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 \( w \).

Correctness of Prim's algorithm (cont'd)

3. Throughout the execution of Prim, \( T \) is contained in some MST of \( G \).
   
   \textit{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 \( e \notin T' \), consider the unique path \( P \) from \( x \) to \( y \) in \( T' \). Then \( P \) contains exactly one fringe edge \( e' = (x', y') \).
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 $T$ 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:

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

- $\text{Decrease-Key}(e, k)$: 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 PRIM(G, W)
1. Initialise parent array π:
   π[v] ← nil for all vertices v
2. Initialise weight array:
   weight[v] ← ∞ for all v
3. Initialise inMST array:
   inMST[v] ← false for all v
4. Initialise priority queue Q
5. v ← arbitrary vertex of G
6. Q.insert(v, 0)
7. while not Q.is-empty() do
   8. y ← Q.extract-min()
   9. for all z adjacent to y do
      10. if weight[z] > weight[y] then
           11. weight[z] ← weight[y]
           12. π[z] ← y
           13. Q.decrease-key(z, weight[y])
14. return π
```

Analysis of PRIM’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

\[
\Theta(n \cdot T_{\text{extract-min}}(n))
\]

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

\[
\Theta(m \cdot T_{\text{relax}}(n, m))
\]

Analysis of Prim’s algorithm (RELAX)

- Decreasing the time needed to execute \text{INSERT} and \text{DECREASE-KEY}, the execution of \text{RELAX} requires time \(\Theta(1)\).
- \text{INSERT} is executed once for every vertex, which requires time

\[
\Theta(n \cdot T_{\text{INSERT}}(n))
\]

- \text{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

\[
T_{\text{PRIM}}(n, m) = \Theta(n(T_{\text{EXTRACT-MIN}}(n) + T_{\text{INSERT}}(n)) + mT_{\text{DECREASE-KEY}}(n))
\]

Priority Queue Implementations

- \text{Array}: Elements simply stored in an array.
- \text{Heap}: Elements are stored in a binary heap (see Inf2B (ADS note 7), [CLRS] Section 6.5)
- \text{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>INSERT</td>
<td>(\Theta(1))</td>
<td>(\Theta(lg n))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>(\Theta(n))</td>
<td>(\Theta(lg n))</td>
<td>(\Theta(lg n))</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>(\Theta(1))</td>
<td>(\Theta(lg n))</td>
<td>(\Theta(1)) (amortised)</td>
</tr>
</tbody>
</table>
Running-time of Prim

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

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 \( m \in \mathcal{O} \left( \frac{n^2}{\log n} \right) \), 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 Prims algorithm).

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?

Reading Assignment

[CLRS] Chapter 23.