Algorithms and Data Structures: Minimum Spanning Trees I and II (Prim)

31st Oct & 4th Nov, 2014

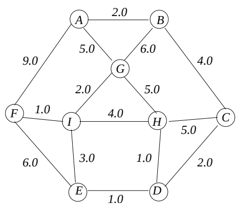
Weighted Graphs

Definition 1

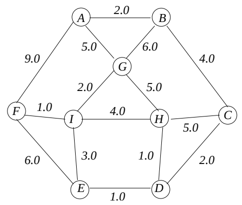
A *weighted* (directed or undirected graph) is a pair $(\mathcal{G}, W)$ consisting of a graph $\mathcal{G} = (V, E)$ and a *weight function* $W : E \to \mathbb{R}$.

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

Example



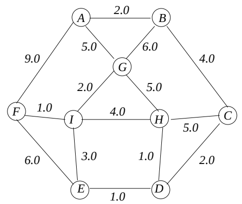
Representations of Weighted Graphs (as Matrices)



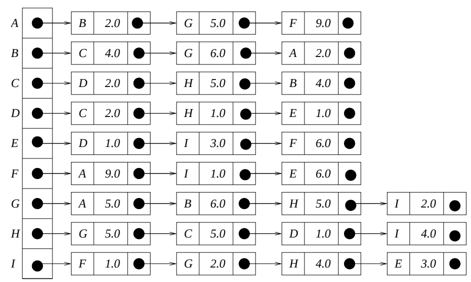
Adjacency Matrix

$$
\begin{pmatrix}
0 & 2.0 & 0 & 0 & 0 & 9.0 & 5.0 & 0 & 0 \\
2.0 & 0 & 4.0 & 0 & 0 & 0 & 6.0 & 0 & 0 \\
0 & 4.0 & 0 & 2.0 & 0 & 0 & 0 & 5.0 & 0 \\
0 & 0 & 2.0 & 0 & 1.0 & 0 & 0 & 1.0 & 0 \\
0 & 0 & 0 & 1.0 & 0 & 6.0 & 0 & 0 & 3.0 \\
9.0 & 0 & 0 & 0 & 6.0 & 0 & 0 & 0 & 1.0 \\
5.0 & 6.0 & 0 & 0 & 0 & 0 & 0 & 5.0 & 2.0 \\
0 & 0 & 5.0 & 1.0 & 0 & 0 & 5.0 & 0 & 4.0 \\
0 & 0 & 0 & 0 & 3.0 & 1.0 & 2.0 & 4.0 & 0
\end{pmatrix}
$$

Representations of Weighted Graphs (Adjacency List)



Adjacency Lists



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.

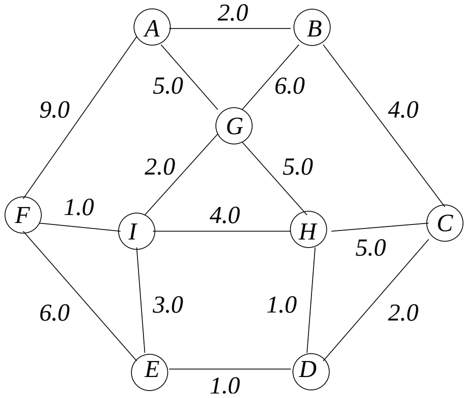
A *fringe edge* for a subtree $\mathcal{T}$ of a graph is an edge with exactly one endpoint in $\mathcal{T}$ (so $e = (u, v)$ with $u \in \mathcal{T}$ and $v \notin \mathcal{T}$).

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

$\mathcal{T} \leftarrow$ one vertex tree with arbitrary vertex of $\mathcal{G}$
while there is a fringe edge **do**
    add fringe edge of minimum weight to $\mathcal{T}$
return $\mathcal{T}$

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

Example



Correctness of Prim's algorithm

1. Throughout the execution of PRIM, $\mathcal{T}$ remains a tree.

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

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

   (ii) We never add any edge which has *both* endpoints in $\mathcal{T}$ (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 $\mathcal{T}$.

   Proof: by *contradiction* ... (depends on our assumption that the graph $\mathcal{G}$ 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 $v = v_0 e_1 v_1 e_2 \ldots v_n = w$ be a path from some vertex v inside $\mathcal{T}$ to w (we know such a path must exist, because $\mathcal{G}$ is connected). Let v_i be the first vertex on this path that never got added to $\mathcal{T}$.
   - After v_{i-1} was added to $\mathcal{T}$, $e_i = (v_{i-1}, v_i)$ would have become a fringe edge. Also, it would have remained as a fringe edge unless v_i was added to $\mathcal{T}$.
   - 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 $\mathcal{T}$ for every vertex w.

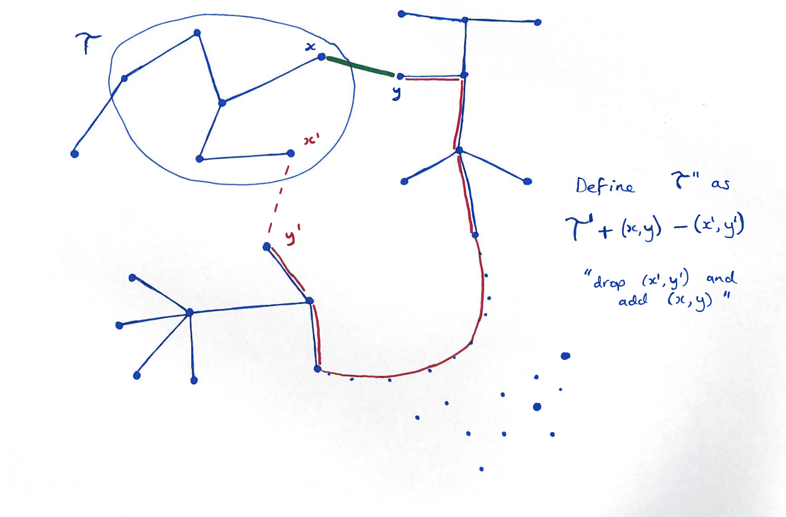
Correctness of Prim's algorithm (cont'd)

3. Throughout the execution of PRIM, $\mathcal{T}$ is contained in some MST of $\mathcal{G}$.

   Proof: (by Induction)

   - Suppose that $\mathcal{T}$ is contained in an MST $\mathcal{T}'$ and that fringe edge $e = (x, y)$ is then added to $\mathcal{T}$ by PRIM. We shall prove that $\mathcal{T} + e$ is contained in some MST $\mathcal{T}''$ (not necessarily $\mathcal{T}'$).
   - case (i): If e is contained in $\mathcal{T}'$, our proof is easy, we simply let $\mathcal{T}'' = \mathcal{T}'$.
   - case (ii): Otherwise, if $e \notin \mathcal{T}'$, consider the unique path $\mathcal{P}$ from x to y in $\mathcal{T}'$. Then $\mathcal{P}$ contains *exactly one* fringe edge $e' = (x', y')$.

Correctness of Prim's algorithm (cont'd)



Correctness of Prim's algorithm (cont'd)

3. case (ii) cont'd
   - Then $W(e) \leq W(e')$.
     (otherwise e' would *definitely* have been added before e)
   - Let $\mathcal{T}'' = \mathcal{T}' + e - e'$.
   - $\mathcal{T}''$ is a tree.
     We drop $e' = (x', y')$, which splits the MST into two components: $\mathcal{T}'_{x'}$ and the other subtree $\mathcal{T}'_{y'} = \mathcal{T}' \setminus \mathcal{T}'_{x'}$. We know x and y are now in different components after this split, because we have broken the unique path $\mathcal{P}$ between x and y in $\mathcal{T}'$. Hence we can add (x, y) to re-join $\mathcal{T}'_{x'}$ and $\mathcal{T}'_{y'}$ without making a cycle.
     $\mathcal{T}''$ has the same vertices as $\mathcal{T}'$, thus it is a spanning tree.
   - Moreover, $W(\mathcal{T}'') \leq W(\mathcal{T}')$, thus $\mathcal{T}''$ 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:

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

To update the keys during the execution of $\textsc{Prim}$, we need priority queues supporting the following additional method:

- $\textsc{Decrease-Key}(e, k)$: Set the key of e to k and update the priority queue. It is assumed that k is smaller than of equal to the old key of e.

Implementation of Prim's Algorithm

Algorithm $\textsc{Prim}(\mathcal{G}, W)$

```
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.  weight[v] = 0;
8.  while not(Q.IS-EMPTY()) do
9.          y ← Q.EXTRACT-MIN()
10.         inMST[y] ← true
11.         for all z adjacent to y do
12.                 RELAX(y,z)
13. return π
```

Algorithm $\textsc{Relax}(y, z)$

```
w ← W(y, z)
if weight[z] = ∞ then
        weight[z] ← w
        π[z] ← y
        Q.INSERT(z, w)
else if (w < weight[z] and
             not (inMST[z])) then
        weight[z] ← w
        π[z] ← y
        Q.DECREASE KEY(z, w)
```


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 $\mathcal{G}$ 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

$$\Theta\big(n \cdot T_{\textsc{Extract-Min}}(n)\big)$$

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

$$\Theta\big(m \cdot T_{\textsc{Relax}}(n, m)\big).$$

Analysis of Prim's algorithm (Relax)

- Decreasing the time needed to execute $\textsc{Insert}$ and $\textsc{Decrease-Key}$, the execution of $\textsc{Relax}$ requires time $\Theta(1)$.
- $\textsc{Insert}$ is executed once for every vertex, which requires time

$$\Theta\big(n \cdot T_{\textsc{Insert}}(n)\big)$$

- $\textsc{Decrease-Key}$ is executed at most once for every edge. This can require time of size

$$\Theta\big(m \cdot T_{\textsc{Decrease-key}}(n)\big)$$

Overall, we get

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

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)

method	running time		
	Array	Heap	Fibonacci Heap
$\textsc{Insert}$	$\Theta(1)$	$\Theta(\lg n)$	$\Theta(1)$
$\textsc{Extract-Min}$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
$\textsc{Decrease-Key}$	$\Theta(1)$	$\Theta(\lg n)$	$\Theta(1)$ (amortised)

Running-time of Prim

$$T_{\text{PRIM}}(n,m) = \Theta\left(n\big(T_{\text{EXTRACT-MIN}}(n) + T_{\text{INSERT}}(n)\big) + mT_{\text{DECREASE-KEY}}(n)\right)$$

Which Priority Queue implementation?

- With array implementation of priority queue:

$$T_{\text{PRIM}}(n,m) = \Theta(n^2).$$

- With heap implementation of priority queue:

$$T_{\text{PRIM}}(n,m) = \Theta((n+m)\lg(n)).$$

- With Fibonacci heap implementation of priority queue:

$$T_{\text{PRIM}}(n,m) = \Theta(n\lg(n) + m).$$

(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 Θ-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 $m \in O(\frac{n^2}{\lg n})$, 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).

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?