COMP90038 Algorithms and Complexity

Greedy Algorithms: Prim and Dijkstra

Michael Kirley

Lecture 21

Semester 1, 2019

Greedy Algorithms

A natural strategy to problem solving is to make decisions based on what is the locally best choice.

Suppose we have coin denominations 25, 10, 5, and 1, and we want to change 30 cents using the smallest number of coins.

In general we will want to use as many 25-cent pieces as we can, then do the same for 10-cent pieces, and so on, until we have reached 30 cents. (In this case we use 25+5 cents.)

This greedy strategy will work for the given denominations, but not for, say, 25, 10, 1.

Greedy Algorithms

In general we cannot expect locally best choices to yield globally best outcomes.

However, there are some well-known algorithms that rely on the greedy approach, being both correct and fast.

In other cases, for hard problems, a greedy algorithm can sometimes serve as an acceptable approximation algorithm (we will discuss approximation algorithms in Week 12).

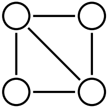
Here we shall look at

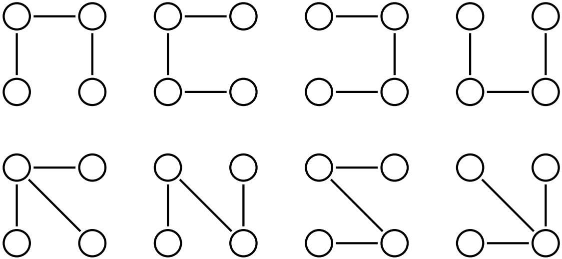
- Prim's algorithm for finding minimum spanning trees
- Dijkstra's algorithm for single-source shortest paths

Spanning Trees

Recall that a tree is a connected graph with no cycle.

A spanning tree of a graph $\langle V, E \rangle$ is a tree $\langle V, E' \rangle$ with $E' \subseteq E$.

The graph  has eight different spanning trees:

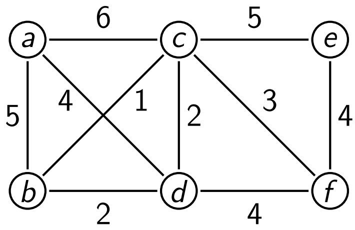


Minimum Spanning Trees of Weighted Graphs

In applications where the edges correspond to distances, or cost, some spanning trees will be more desirable than others.

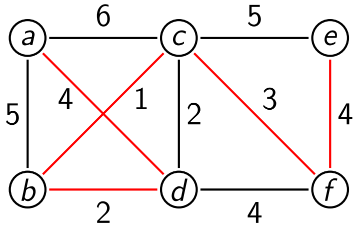
Suppose we have a set of ‘stations’ to connect in a network, and also some possible connections, together with the cost of each connection.

Then we have a weighted graph problem, of finding a spanning tree with the smallest possible cost.



Minimum Spanning Trees

Given a weighted graph, a sub-graph which is a tree with minimal weight is a minimum spanning tree for the graph.



Minimum Spanning Trees: Prim's Algorithm

Prim's algorithm is an example of a greedy algorithm.

It constructs a sequence of subtrees T, each adding a node together with an edge to a node in the previous subtree. In each step it picks a closest node from outside the tree and adds that. A sketch:

function $\textsc{Prim}(\langle V, E \rangle)$
    $V_T \leftarrow \{v_0\}$
    $E_T \leftarrow \emptyset$
    for $i \leftarrow 1$ to $|V| - 1$ **do**
        find a minimum-weight edge $(v, u) \in V_T \times (V \setminus V_T)$
        $V_T \leftarrow V_T \cup \{u\}$
        $E_T \leftarrow E_T \cup \{(v, u)\}$
    return E_T

Prim's Algorithm

Note that in each iteration, the tree grows by one edge.

Or, we can say that the tree grows to include the node from outside that has the smallest cost.

But how to find the minimum-weight edge (v, u)?

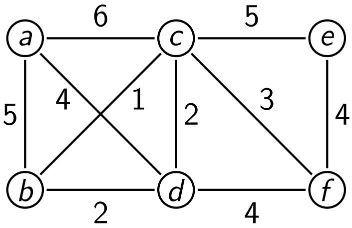
A standard way to do this is to organise the nodes that are not yet included in the spanning tree T as a priority queue, organised in a min-heap by edge cost.

The information about which nodes are connected in T can be captured by an array *prev* of nodes, indexed by V. Namely, when (v, u) is included, this is captured by setting $prev[u] = v$.

Prim's Algorithm

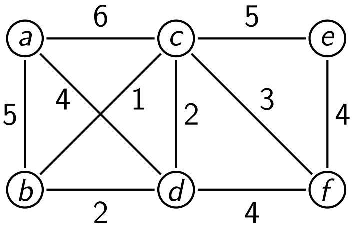
$$
\begin{array}{l}
\textbf{function } \text{PRIM}(\langle V, E\rangle) \\
\quad \textbf{for } \text{each } v \in V \textbf{ do} \\
\quad\quad cost[v] \leftarrow \infty \\
\quad\quad prev[v] \leftarrow nil \\
\quad \text{pick initial node } v_0 \\
\quad cost[v_0] \leftarrow 0 \\
\quad Q \leftarrow \text{INITPRIORITYQUEUE}(V) \qquad \triangleright \text{ priorities are cost values} \\
\quad \textbf{while } Q \text{ is non-empty } \textbf{do} \\
\quad\quad u \leftarrow \text{EJECTMIN}(Q) \\
\quad\quad \textbf{for } \text{each } (u, w) \in E \textbf{ do} \\
\quad\quad\quad \textbf{if } weight(u, w) < cost[w] \textbf{ then} \\
\quad\quad\quad\quad cost[w] \leftarrow weight(u, w) \\
\quad\quad\quad\quad prev[w] \leftarrow u \\
\quad\quad\quad\quad \text{UPDATE}(Q, w, cost[w]) \quad \triangleright \text{ rearranges priority queue}
\end{array}
$$

Prim's Algorithm: Example



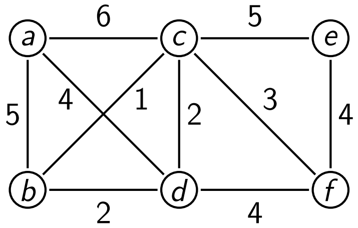
Tree T	a	b	c	d	e	f
–	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil

Prim's Algorithm: Example



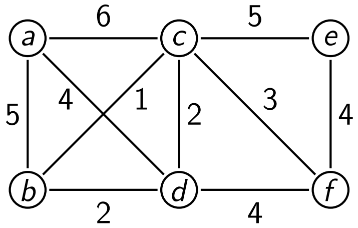
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil

Prim's Algorithm: Example



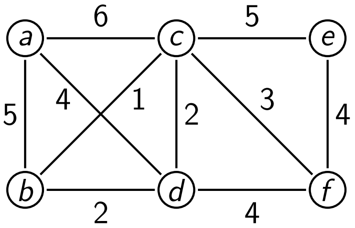
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$

Prim's Algorithm: Example



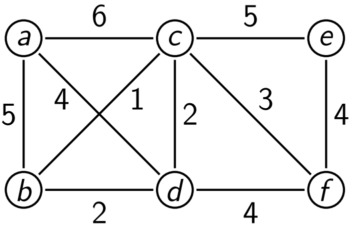
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$

Prim's Algorithm: Example



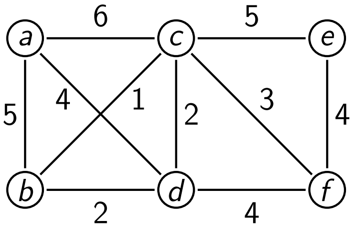
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$
a, d, b, c, f					$4/f$	

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$
a, d, b, c, f					$4/f$	
a, d, b, c, f, e						

Analysis of Prim's Algorithm

First, a crude analysis: For each node, we look through the edges to find those incident to the node, and pick the one with smallest cost. Thus we get $O(|V| \cdot |E|)$. However, we are using cleverer data structures.

Using adjacency lists for the graph and a min-heap for the priority queue, we perform $|V| - 1$ heap deletions (each at cost $O(\log |V|)$) and $|E|$ updates of priorities (again, each at cost $O(\log |V|)$).

Altogether $(|V| - 1 + |E|)O(\log |V|)$.

Since, in a connected graph, $|V| - 1 \leq |E|$, this is $O(|E| \log |V|)$.

Kruskal's Algorithm

An alternative minimal-spanning-tree algorithm, also greedy, is Kruskal's algorithm.

The algorithm is explained in Levitin's Section 9.2, which we skip.

For sparse graphs, if properly implemented, Kruskal's algorithm will generally do better than Prim's.

Dijkstra's Algorithm

Another classical greedy weighted-graph algorithm is Dijkstra's algorithm, whose overall structure is the same as Prim's.

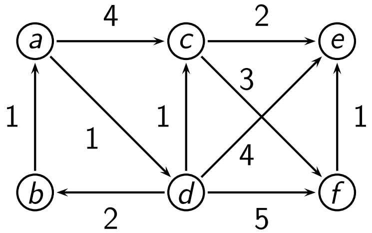
Recall that Floyd's algorithm gave us the shortest paths, for every pair of nodes, in a (directed or undirected) weighted graph. It assumed an adjacency matrix representation and had complexity $O(|V|^3)$.

Dijkstra's algorithm is also a shortest-path algorithm for (directed or undirected) weighted graphs. It finds all shortest paths from a fixed start node. Its complexity is the same as that of Prim's algorithm.

Dijkstra's Algorithm

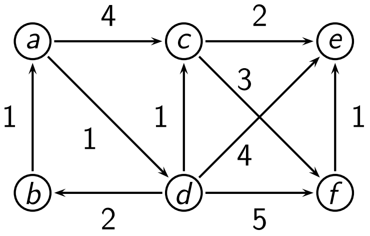
$$
\begin{array}{l}
\textbf{function } \textsc{Dijkstra}(\langle V, E\rangle, v_0) \\
\quad \textbf{for } \text{each } v \in V \textbf{ do} \\
\qquad dist[v] \leftarrow \infty \\
\qquad prev[v] \leftarrow nil \\
\quad dist[v_0] \leftarrow 0 \\
\quad Q \leftarrow \textsc{InitPriorityQueue}(V) \qquad \triangleright \text{ priorities are distances} \\
\quad \textbf{while } Q \text{ is non-empty } \textbf{do} \\
\qquad u \leftarrow \textsc{EjectMin}(Q) \\
\qquad \textbf{for } \text{each } (u, w) \in E \textbf{ do} \\
\qquad\quad \textbf{if } dist[u] + weight(u, w) < dist[w] \textbf{ then} \\
\qquad\qquad dist[w] \leftarrow dist[u] + weight(u, w) \\
\qquad\qquad prev[w] \leftarrow u \\
\qquad\qquad \textsc{Update}(Q, w, dist[w]) \qquad \triangleright \text{ rearranges priority queue}
\end{array}
$$

Dijkstra's Algorithm: Example



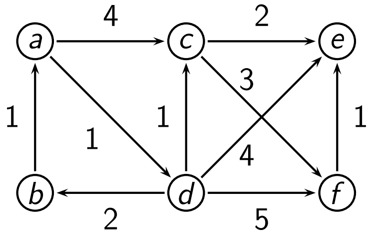
Covered	a	b	c	d	e	f
–	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil

Dijkstra's Algorithm: Example



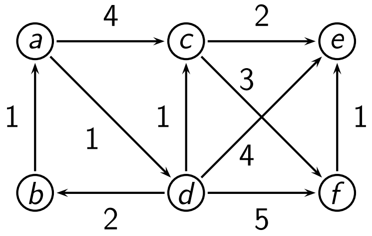
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil

Dijkstra's Algorithm: Example



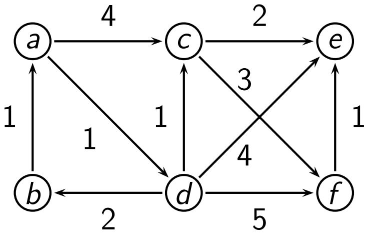
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$

Dijkstra's Algorithm: Example



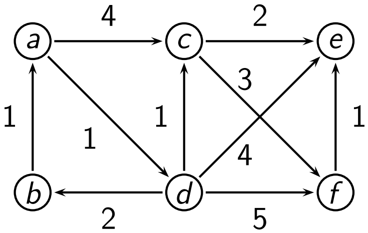
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$

Dijkstra's Algorithm: Example



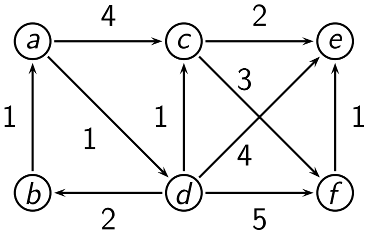
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$

Dijkstra's Algorithm: Example



Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$
a, d, c, b, e						$5/c$

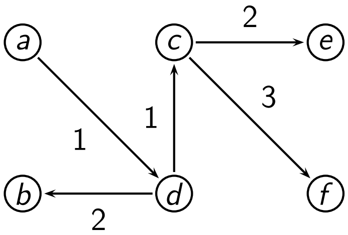
Dijkstra's Algorithm: Example



Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$
a, d, c, b, e						$5/c$
a, d, c, b, e, f						

Dijkstra's Algorithm: Tracing Paths

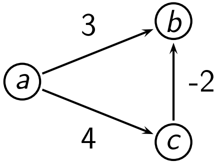
The array `prev` is not really needed, unless we want to retrace the shortest paths from node *a*:



Negative Weights

In our example, we used positive weights, and for a good reason: Dijkstra's algorithm may not work otherwise!

In this example, the greedy pick—choosing the edge from a to b—is clearly the wrong one.



The Bellman-Ford algorithm will find single-source shortest paths in arbitrary weighted graphs (but is also more costly to run).

And if the graph has a cycle with accumulated negative weight then none of these algorithms work—they don't really make sense.

Why?

Coming Up

We will look at a final example of a useful greedy algorithm:

Huffman encoding for data compression.