

COMP20007 Design of Algorithms

Greedy Algorithms: Prim and Dijkstra

Lars Kulik

Lecture 8

Semester 1, 2020

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**.

Here we shall look at

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

The Priority Queue

A priority queue is a **set** (or **pool**) of elements.

An element is injected into the priority queue together with a **priority** (often the key value itself) and elements are ejected according to priority.

As an abstract data type, the priority queue supports the following operations on a “pool” of elements (ordered by some linear order):

- **find** an item with maximal priority
- **insert** a new item with associated priority
- test whether a priority queue is empty
- **eject** the **largest** element

Stacks and Queues as Priority Queues

Special instances are obtained when we use **time** for priority:

- If “large” means “late” we obtain the **stack**.
- If “large” means “early” we obtain the **queue**.

Possible Implementations of the Priority Queue

Assume priority = key.

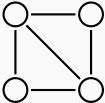
	INJECT(<i>e</i>)	EJECT()
Unsorted array or list		
Sorted array or list		

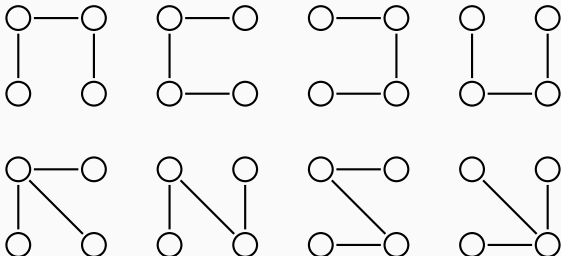


Spanning Trees

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:

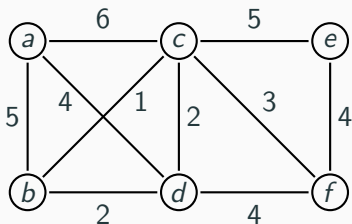


Minimum Spanning Trees of Weighted Graphs

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

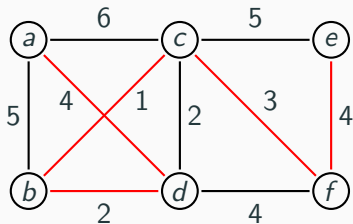
Suppose we have a set of 'stations' to connect in a network, and also some possible connections, together with the **cost** of each connection.

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 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 do we find the minimum-weight edge (v, u) ?

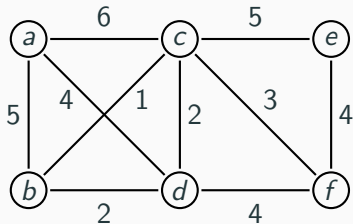
A standard way to do this is to organise the nodes that are not yet included in the spanning tree T as a **priority queue** organised by edge **cost**.

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

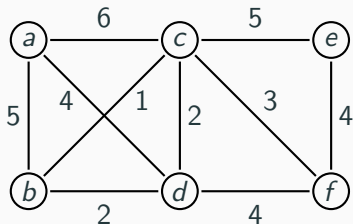
```
function PRIM( $\langle V, E \rangle$ )  
  for each  $v \in V$  do  
     $cost[v] \leftarrow \infty$   
     $prev[v] \leftarrow nil$   
  pick initial node  $v_0$   
   $cost[v_0] \leftarrow 0$   
   $Q \leftarrow \text{INITPRIORITYQUEUE}(V)$  ▷ priorities are cost values  
  while  $Q$  is non-empty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

Prim's Algorithm: Example



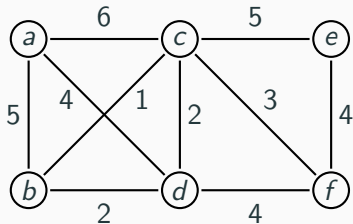
Tree T	a	b	c	d	e	f
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>

Prim's Algorithm: Example



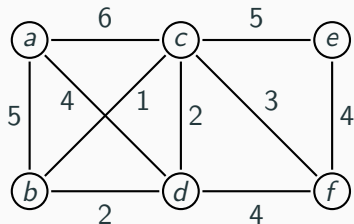
Tree T	a	b	c	d	e	f
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
a		5/ a	6/ a	4/ a	∞ / <i>nil</i>	∞ / <i>nil</i>

Prim's Algorithm: Example



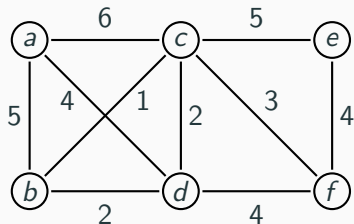
Tree T	a	b	c	d	e	f
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
a		5/ a	6/ a	4/ a	∞ / <i>nil</i>	∞ / <i>nil</i>
a, d		2/ d	2/ d		∞ / <i>nil</i>	4/ d

Prim's Algorithm: Example



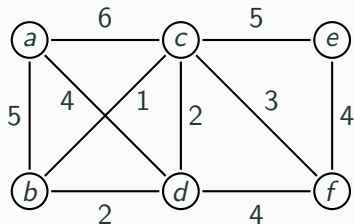
Tree T	a	b	c	d	e	f
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
a		5/ a	6/ a	4/ a	∞ / <i>nil</i>	∞ / <i>nil</i>
a, d		2/ d	2/ d		∞ / <i>nil</i>	4/ d
a, d, b			1/ b		∞ / <i>nil</i>	4/ d

Prim's Algorithm: Example



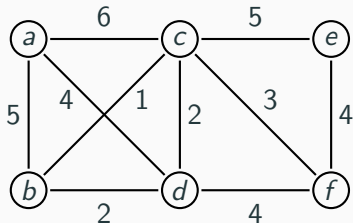
Tree T	a	b	c	d	e	f
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
a		5/ a	6/ a	4/ a	∞ / <i>nil</i>	∞ / <i>nil</i>
a, d		2/ d	2/ d		∞ / <i>nil</i>	4/ d
a, d, b			1/ b		∞ / <i>nil</i>	4/ d
a, d, b, c					5/ c	3/ c

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
a		5/ a	6/ a	4/ a	∞ / <i>nil</i>	∞ / <i>nil</i>
a, d		2/ d	2/ d		∞ / <i>nil</i>	4/ d
a, d, b			1/ b		∞ / <i>nil</i>	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/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
a		5/ a	6/ a	4/ a	∞ / <i>nil</i>	∞ / <i>nil</i>
a, d		2/ d	2/ d		∞ / <i>nil</i>	4/ d
a, d, b			1/ b		∞ / <i>nil</i>	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 can do better! We will discuss this later.

Dijkstra's Algorithm

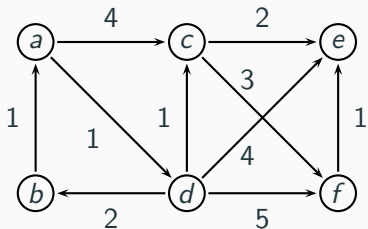
Another classical greedy weighted-graph algorithm is **Dijkstra's algorithm**, whose overall structure is the same as Prim's.

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

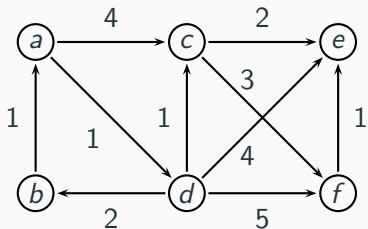
```
function DIJKSTRA( $\langle V, E \rangle, v_0$ )  
  for each  $v \in V$  do  
     $dist[v] \leftarrow \infty$   
     $prev[v] \leftarrow nil$   
  
   $dist[v_0] \leftarrow 0$   
   $Q \leftarrow \text{INITPRIORITYQUEUE}(V)$  ▷ priorities are distances  
  while  $Q$  is non-empty do  
     $u \leftarrow \text{EJECTMIN}(Q)$   
    for each  $(u, w) \in E$  do  
      if  $w \in Q$  and  $dist[u] + weight(u, w) < dist[w]$  then  
         $dist[w] \leftarrow dist[u] + weight(u, w)$   
         $prev[w] \leftarrow u$   
         $\text{UPDATE}(Q, w, dist[w])$  ▷ rearranges priority queue
```

Dijkstra's Algorithm: Example



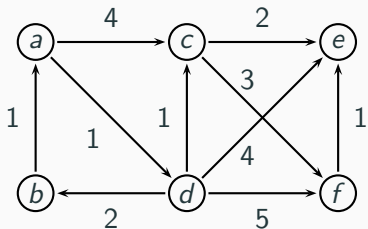
Covered	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>

Dijkstra's Algorithm: Example



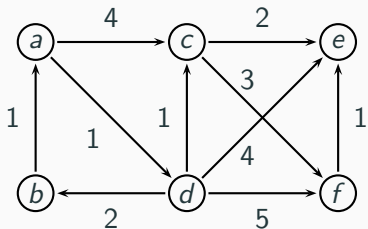
Covered	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a</i>		∞ / <i>nil</i>	4/ <i>a</i>	1/ <i>a</i>	∞ / <i>nil</i>	∞ / <i>nil</i>

Dijkstra's Algorithm: Example



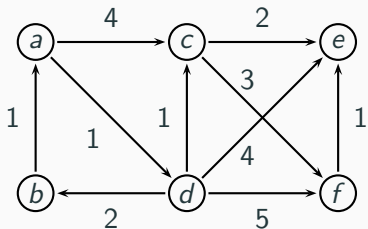
Covered	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a</i>		∞ / <i>nil</i>	4/ <i>a</i>	1/ <i>a</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a, d</i>		3/ <i>d</i>	2/ <i>d</i>		5/ <i>d</i>	6/ <i>d</i>

Dijkstra's Algorithm: Example



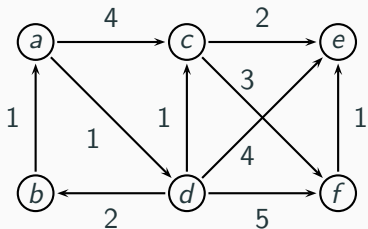
Covered	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a</i>		∞ / <i>nil</i>	4/ <i>a</i>	1/ <i>a</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a, d</i>		3/ <i>d</i>	2/ <i>d</i>		5/ <i>d</i>	6/ <i>d</i>
<i>a, d,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

Dijkstra's Algorithm: Example



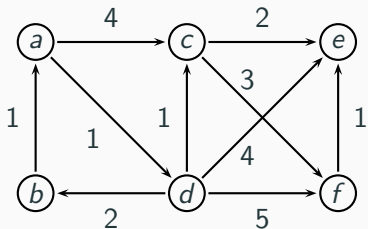
Covered	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
—	0/ <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a</i>		∞ / <i>nil</i>	4/ <i>a</i>	1/ <i>a</i>	∞ / <i>nil</i>	∞ / <i>nil</i>
<i>a, d</i>		3/ <i>d</i>	2/ <i>d</i>		5/ <i>d</i>	6/ <i>d</i>
<i>a, d,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

Dijkstra's Algorithm: Example



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

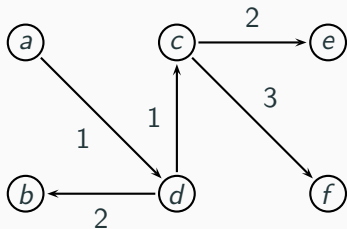
Dijkstra's Algorithm: Example



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

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.

