COMP90038
Algorithms and Complexity

Lecture 20: Prim and Dijkstra

(with thanks to Harald Søndergaard & Michael Kirley)

Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

Review from Lecture 19: Warshall's Algorithm

- Assume the nodes of graph G are numbered from 1 to n.
- Ask the question: Is there a path from node i to node j using only nodes that are no larger than some k as "stepping stones"?
- Such a path either uses node k as a stepping stone, or it doesn't.
- So an answer is: There is such a path if and only if we can

  step from i to j using only nodes $\leq k-1$, or

  step from i to k using only nodes $\leq k-1$, and then
  step from k to j using only nodes $\leq k-1$.

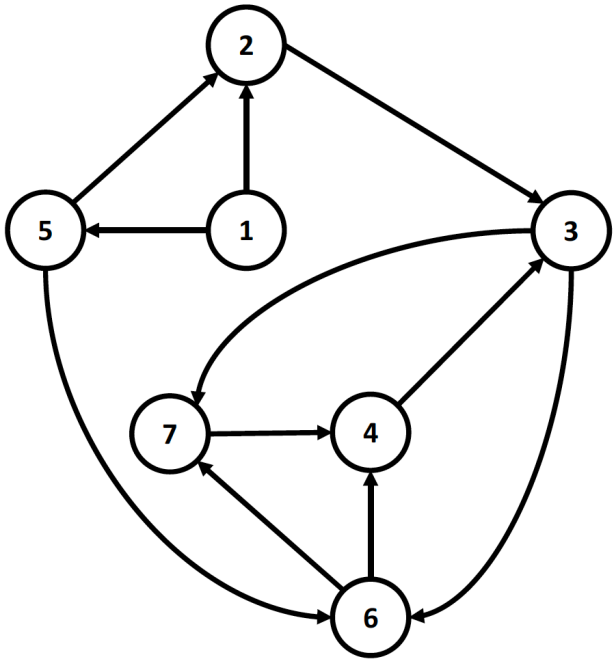
Review from Lecture 19: Warshall's Algorithm

- This leads to a better version of Warshall's algorithm:

```
for k ← 1 to n do
    for i ← 1 to n do
        if A[i, k] then
            for j ← 1 to n do
                if A[k, j] then
                    A[i, j] ← 1
```

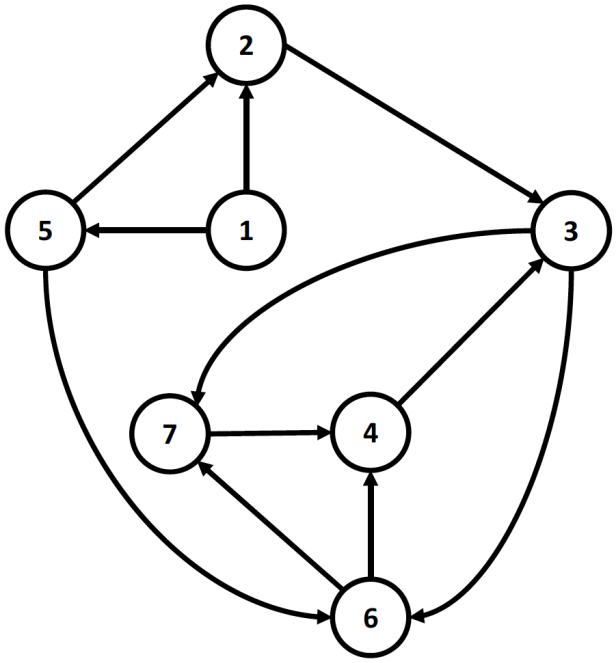
- If each row in the matrix is represented as a bit-string, the innermost for loop (and *j*) can be gotten rid of–instead of iterating, just apply the "bitwise or" of row k to row i.

Review from Lecture 19: Warshall's Algorithm



	1	2	3	4	5	6	7
1	0	1	0	0	1	0	0
2	0	0	1	0	0	0	0
3	0	0	0	0	0	1	1
4	0	0	1	0	0	0	0
5	0	1	0	0	0	1	0
6	0	0	0	1	0	0	1
7	0	0	0	1	0	0	0

Review from Lecture 19: Warshall's Algorithm



	1	2	3	4	5	6	7
1	0	1	1	1	1	1	1
2	0	0	1	1	0	1	1
3	0	0	1	1	0	1	1
4	0	0	1	1	0	1	1
5	0	1	1	1	0	1	1
6	0	0	1	1	0	1	1
7	0	0	1	1	0	1	1

Review from Lecture 19: Floyd's Algorithm

- We can use the same problem decomposition as we used to derive Warshall's algorithm. Again assume nodes are numbered 1 to n.

- This time ask the question: What is the shortest path from node i to node j using only nodes $\leq k$ as "stepping stones"?

- We either use node k as a stepping stone, or we avoid it. So again, we can

  step from i to j using only nodes $\leq k-1$, or

  step from i to k using only nodes $\leq k-1$, and then
  step from k to j using only nodes $\leq k-1$.

Review from Lecture 19: Floyd's Algorithm

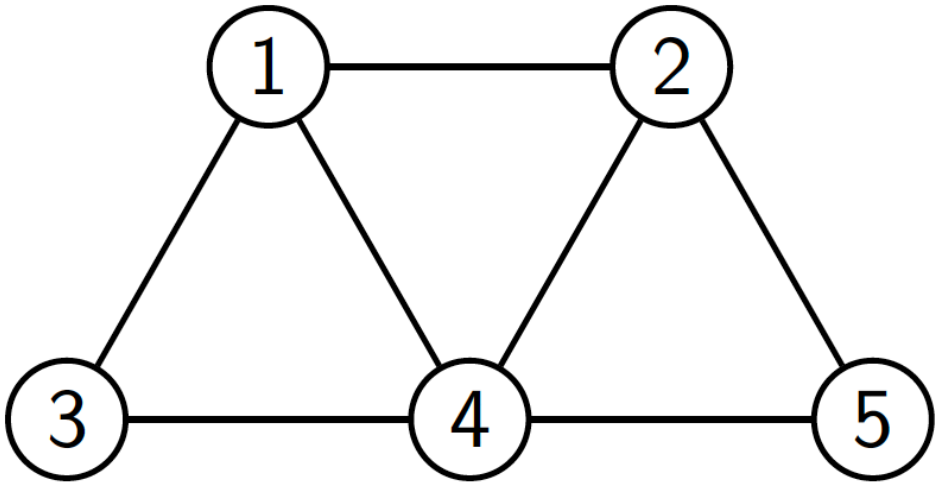
- If G's weight matrix is W then we can express the recurrence relation for minimal distances as follows:

$$D_{ij}^{0} = W[i,j]$$
$$D_{ij}^{k} = \min\left(D_{ij}^{k-1},\ D_{ik}^{k-1} + D_{kj}^{k-1}\right)$$

- And then the algorithm follows easily:

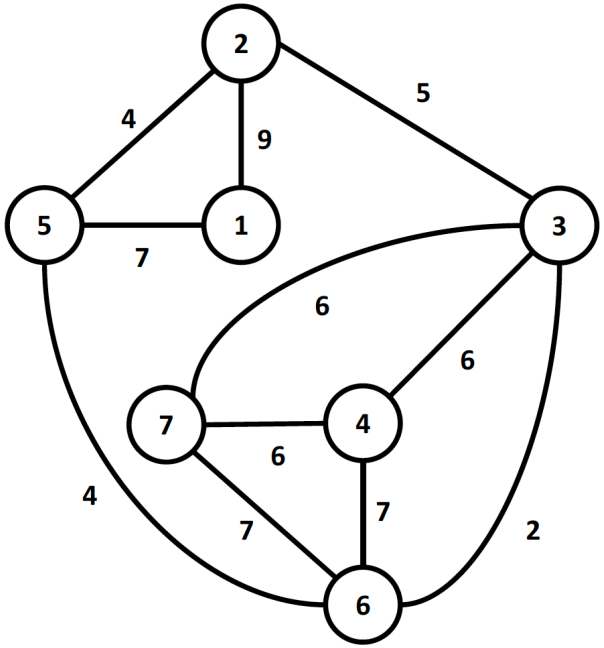
```
function FLOYD(W[1..n, 1..n])
    D ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min(D[i, j], D[i, k] + D[k, j])
    return D
```

Review from Lecture 19: Floyd's Algorithm



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & \infty \\ 1 & 0 & \infty & 1 & 1 \\ 1 & \infty & 0 & 1 & \infty \\ 1 & 1 & 1 & 0 & 1 \\ \infty & 1 & \infty & 1 & 0 \end{bmatrix} \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 & 1 \\ 1 & 2 & 0 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 2 & 1 & 0 \end{bmatrix} \end{array}$$

Review from Lecture 19: Floyd's Algorithm



	1	2	3	4	5	6	7
1	0	9	∞	∞	7	∞	∞
2	9	0	5	∞	4	∞	∞
3	∞	5	0	6	∞	2	6
4	∞	∞	6	0	∞	7	6
5	7	4	∞	∞	0	4	∞
6	∞	∞	2	7	4	0	7
7	∞	∞	6	6	∞	7	0

	1	2	3	4	5	6	7
1	0	9	13	18	7	11	18
2	9	0	5	11	4	7	11
3	13	5	0	6	6	2	6
4	18	11	6	0	11	7	6
5	7	4	6	11	0	4	11
6	11	7	2	7	4	0	7
7	18	11	6	6	11	7	0

Greedy Algorithms

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

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

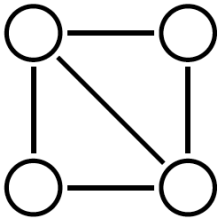
- In general, we will want to use as many 25-cent pieces as we can, then do the same for 10-cent pieces, and so on, until we have reached 30 cents. (In this case we use 25+5 cents.)

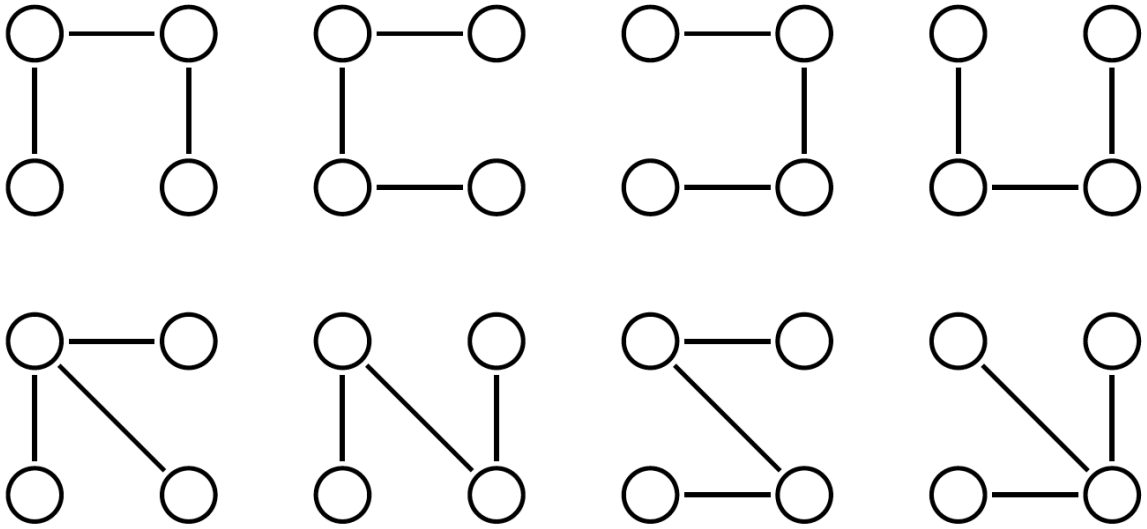
- This greedy strategy will work for the given denominations, but not for, say 25, 10, 1. (compare 25+1+1+1+1+1 with 10+10+10).

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:

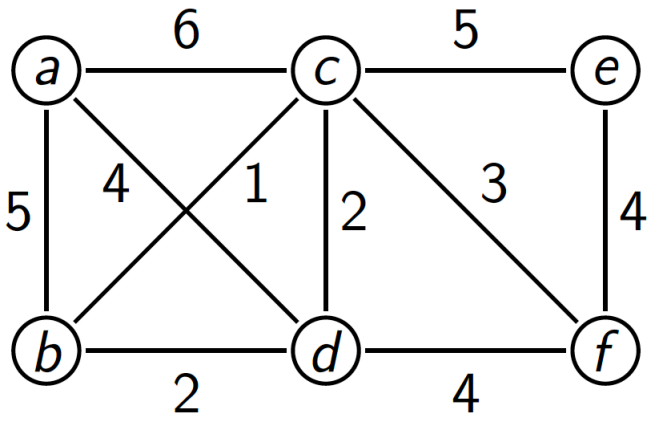


Minimum Spanning Trees of Weighted Graphs

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

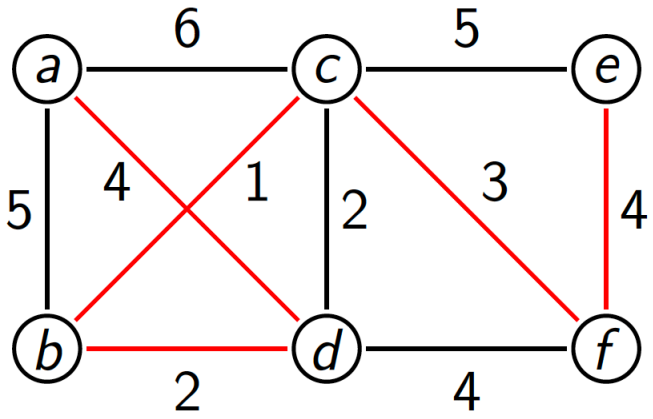
- Suppose we have a set of 'stations' to connect in a network, and also some possible connections, together with the cost of each connection.

- 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 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

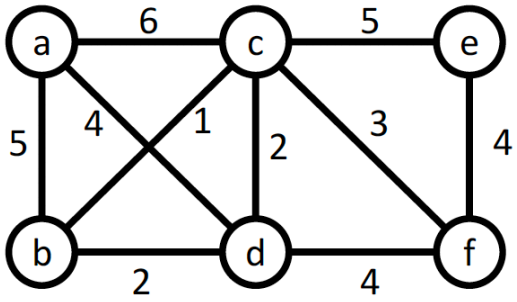
Prim's Algorithm

- 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 $\text{prev}[u] = v$.

Prim's Algorithm


```
function PRIM(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v_0
    cost[v_0] ← 0
    Q ← INITPRIORITYQUEUE(V)        ▷ priorities are cost values
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, cost[w])   ▷ rearranges priority queue
```

Prim's Algorithm: Example

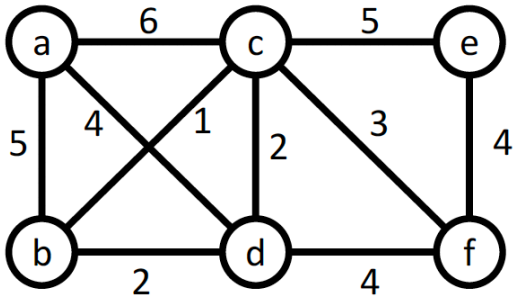


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- On the first loop, we only create the table.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil

Prim's Algorithm: Example

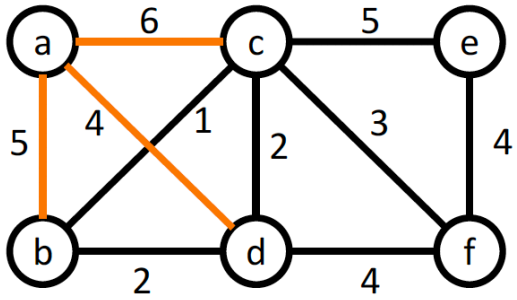



```
function PRIM(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v0
    cost[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)        ▷ priorities are cost values
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, cost[w])   ▷ rearranges priority queue
```

- Then we pick the first node as the initial one.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
	$cost$	0	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil

Prim's Algorithm: Example

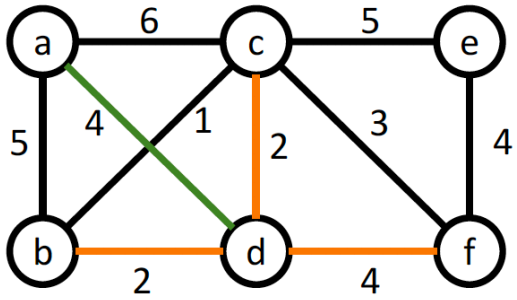


- We take the first node out of the queue & update the costs.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
	$cost$	0	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
a	$cost$		5	6	4	∞	∞
	$prev$		a	a	a	nil	nil

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
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

Prim's Algorithm: Example

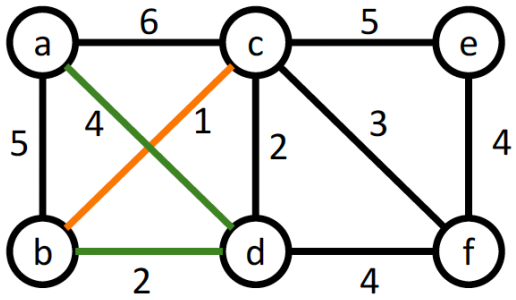


- We eject the node with the lowest cost & update the queue.

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

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
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

Prim's Algorithm: Example

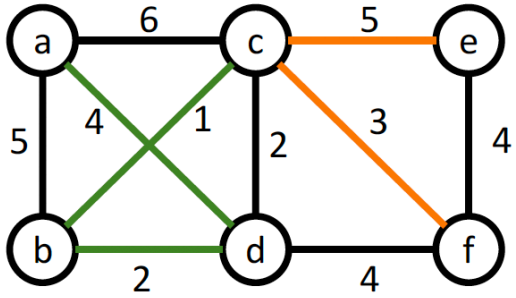


- We eject the next node based on alphabetical order.

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

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
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

Prim's Algorithm: Example

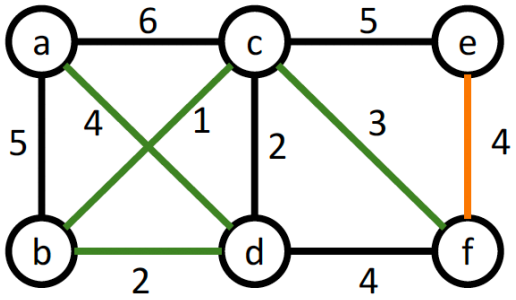


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- We now update (f).

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

Prim's Algorithm: Example

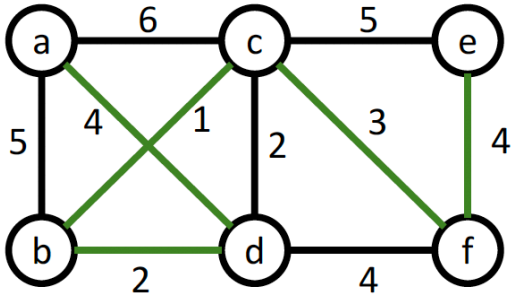


- We have reached the last choice.

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
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

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

Prim's Algorithm: Example



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
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

- The resulting tree is $\{a, d, b, c, f, e\}$.

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

Analysis of Prim's Algorithm

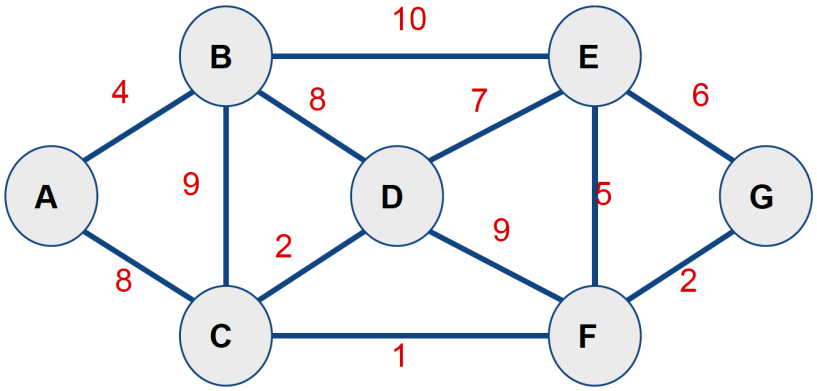
- First, a crude analysis: For each node, we look through the edges to find those incident to the node, and pick the one with the smallest cost. Thus we get $O(|V| \cdot |E|)$. However, we are using clever data structures.

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

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

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

Prim's Algorithm: Example

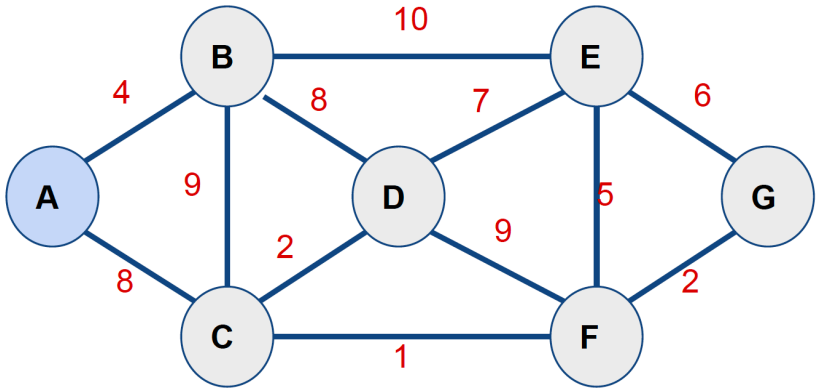



```
function PRIM(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v0
    cost[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)        ▷ priorities are cost values
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, cost[w])   ▷ rearranges priority queue
```

- Pick the first node as the initial one.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil

Prim's Algorithm: Example

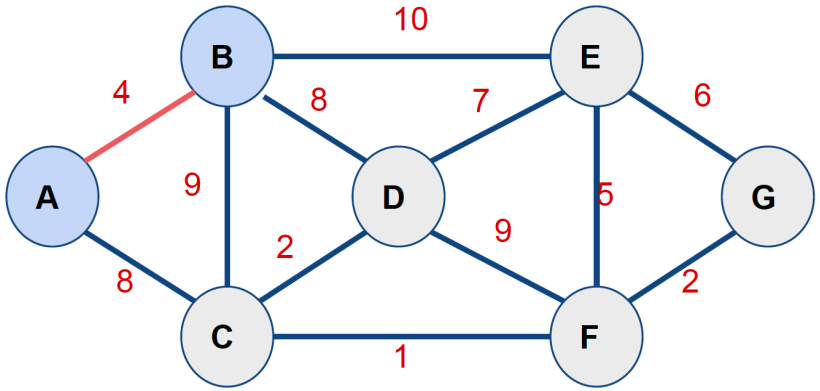


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- We take the first node out of the queue & update the costs.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil

Prim's Algorithm: Example

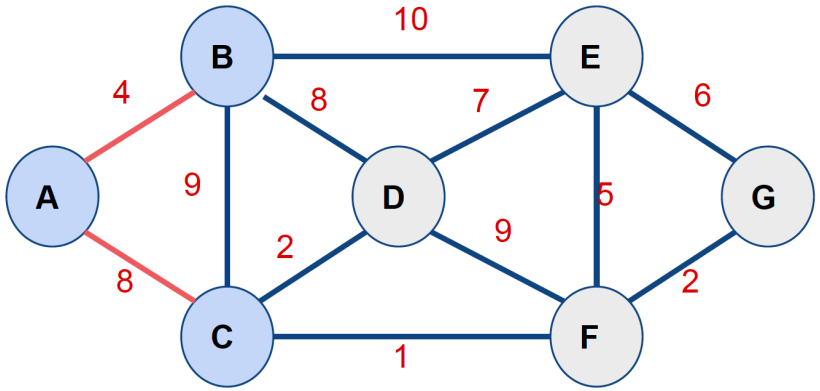



```
function PRIM(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v_0
    cost[v_0] ← 0
    Q ← INITPRIORITYQUEUE(V)        ▷ priorities are cost values
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, cost[w])   ▷ rearranges priority queue
```

- We eject the node with the lowest cost & update the queue.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*
a, b	*cost*			8	8	10	∞	∞
	prev			a	b	b	*nil*	*nil*

Prim's Algorithm: Example

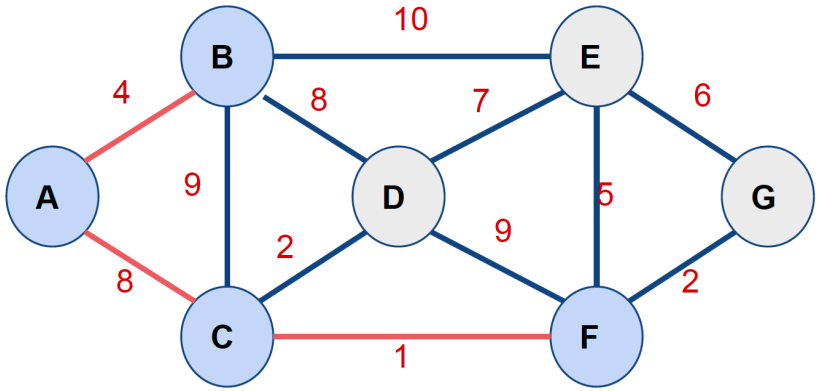


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- We eject the node with the lowest cost & update the queue.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil
a, b	$cost$			8	8	10	∞	∞
	$prev$			a	b	b	nil	nil
a, b, c	$cost$				2	10	1	∞
	$prev$				c	b	c	nil

Prim's Algorithm: Example

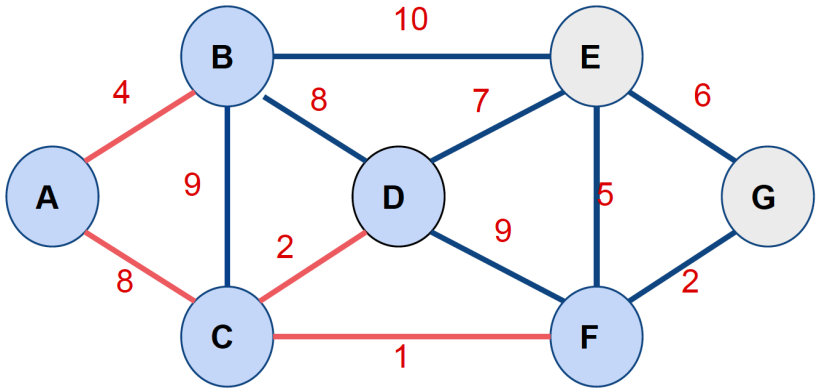


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- We eject the node with the lowest cost & update the queue.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil
a, b	$cost$			8	8	10	∞	∞
	$prev$			a	b	b	nil	nil
a, b, c	$cost$				2	10	1	∞
	$prev$				c	b	c	nil
a, b, c, f	$cost$				2	5		2
	$prev$				c	f		f

Prim's Algorithm: Example

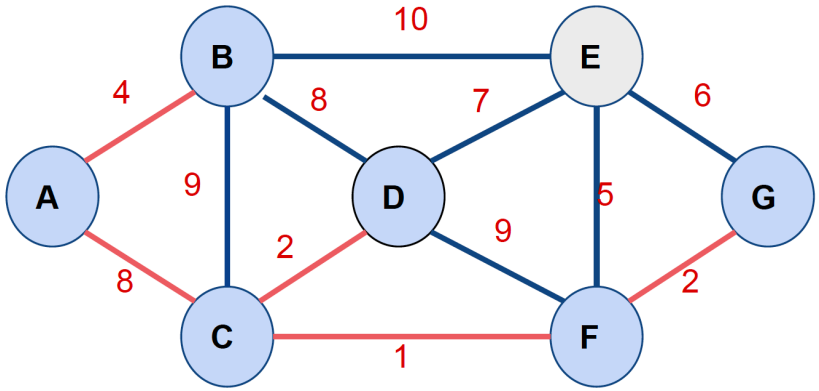


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- We eject the node with the lowest cost & update the queue.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil
a, b	$cost$			8	8	10	∞	∞
	$prev$			a	b	b	nil	nil
a, b, c	$cost$				2	10	1	∞
	$prev$				c	b	c	nil
a, b, c, f	$cost$				2	5		2
	$prev$				c	f		f
a, b, c, f, d	$cost$					5		2
	$prev$					f		f

Prim's Algorithm: Example

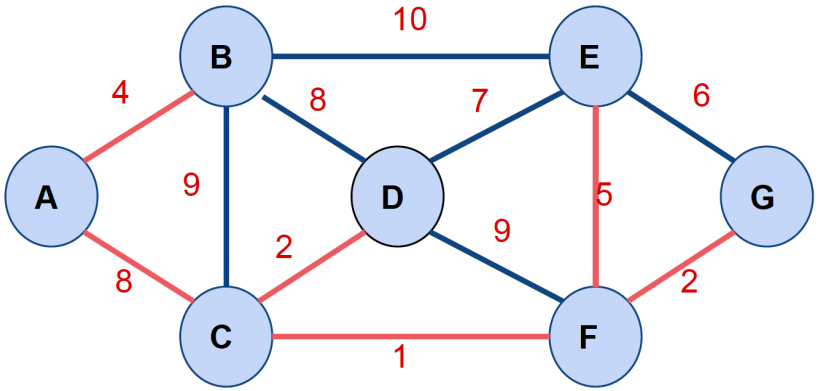


function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)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

- We eject the node with the lowest cost & update the queue.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil
a, b	$cost$			8	8	10	∞	∞
	$prev$			a	b	b	nil	nil
a, b, c	$cost$				2	10	1	∞
	$prev$				c	b	c	nil
a, b, c, f	$cost$				2	5		2
	$prev$				c	f		f
a, b, c, f, d	$cost$					5		2
	$prev$					f		f
a, b, c, f, d, g	$cost$					5		
	$prev$					f		

Prim's Algorithm: Example



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
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

- We eject the node with the lowest cost & update the queue.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil
a, b	$cost$			8	8	10	∞	∞
	$prev$			a	b	b	nil	nil
a, b, c	$cost$				2	10	1	∞
	$prev$				c	b	c	nil
a, b, c, f	$cost$				2	5		2
	$prev$				c	f		f
a, b, c, f, d	$cost$					5		2
	$prev$					f		f
a, b, c, f, d, g	$cost$					5		
	$prev$					f		
a, b, c, f, d, g, e	$cost$							
	$prev$							

Dijkstra's Algorithm

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

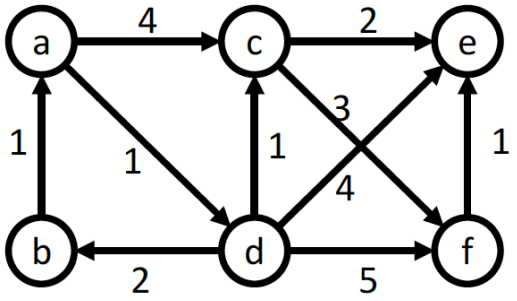
- Recall that Floyd's algorithm gave us the shortest paths, for every pair of nodes, in a (directed or undirected) weighted graph. It assumed an adjacency matrix representation and had a complexity of $O(|V|^3)$.

- 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


```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

Dijkstra's Algorithm: Example

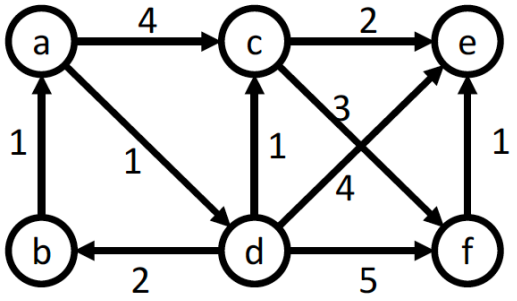



```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

- On the first loop, we only create the table.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil

Dijkstra's Algorithm: Example

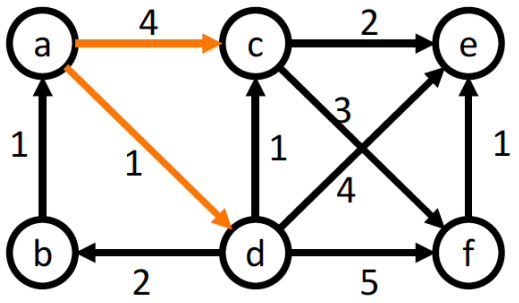


- Then we pick the first node as the initial one.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
	$cost$	0	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil

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

Dijkstra's Algorithm: Example

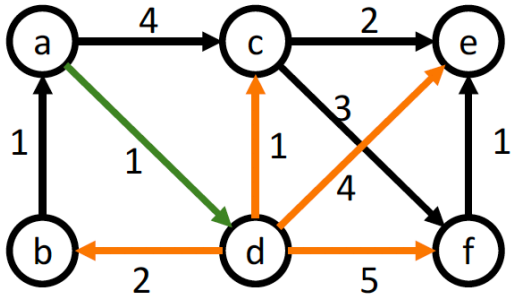


- We take the first node out of the queue & update the distance.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
	$cost$	0	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
a	$cost$		∞	4	1	∞	∞
	$prev$		nil	a	a	nil	nil

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

Dijkstra's Algorithm: Example

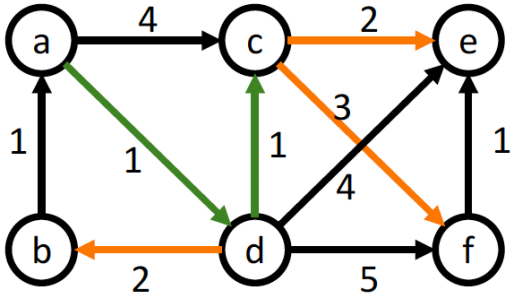


- Eject node with shortest distance from queue; update all path by +1

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

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

Dijkstra's Algorithm: Example

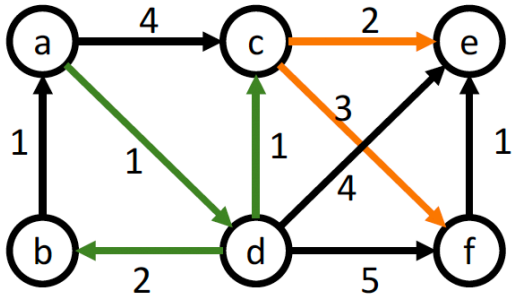


- The next node will be the one with the shortest path overall.

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

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

Dijkstra's Algorithm: Example

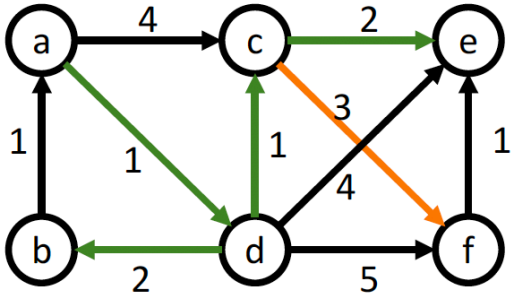



```
function DIJKSTRA(⟨V, E⟩, v_0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v_0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

- Now we continue evaluating from (c).

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

Dijkstra's Algorithm: Example

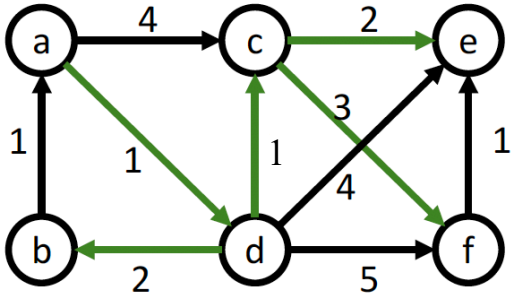



```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

- We arrive at our last decision.

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

Dijkstra's Algorithm: Example



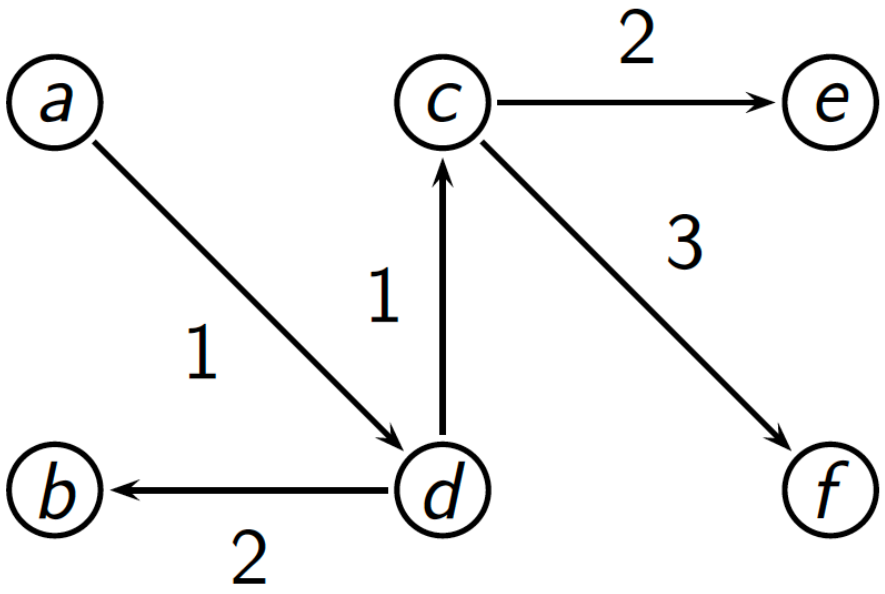

```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

- The complete tree is $\{a, d, c, b, e, f\}$.

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

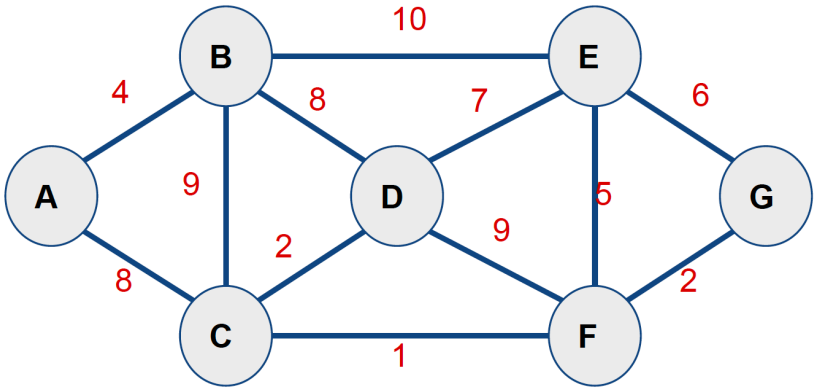
Dijkstra's Algorithm: Tracing Paths

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



- This is referred to as the shortest-path tree.

Dijkstra's Algorithm: Example

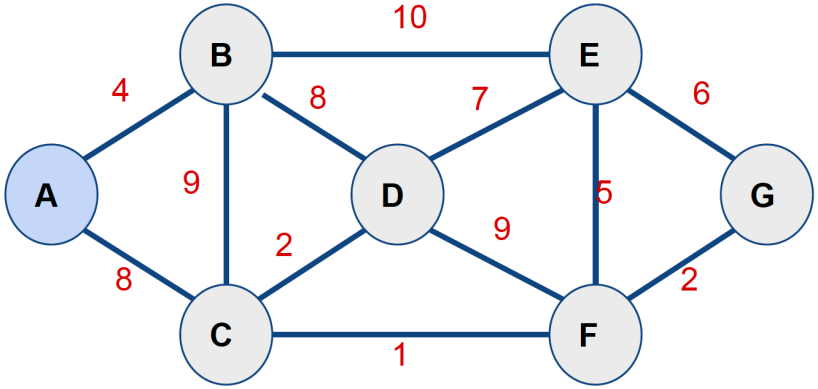


function DIJKSTRA($\langle V, E \rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are distances
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each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

- Pick the first node as the initial one.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil

Dijkstra's Algorithm: Example

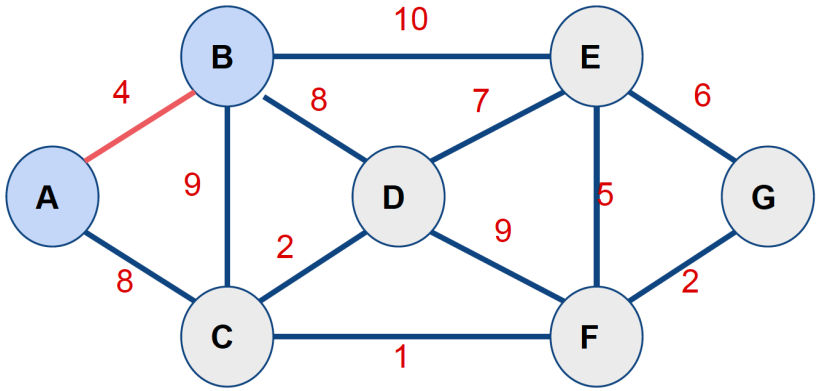



```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

- We take the first node out of the queue & update the distances.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*

Dijkstra's Algorithm: Example

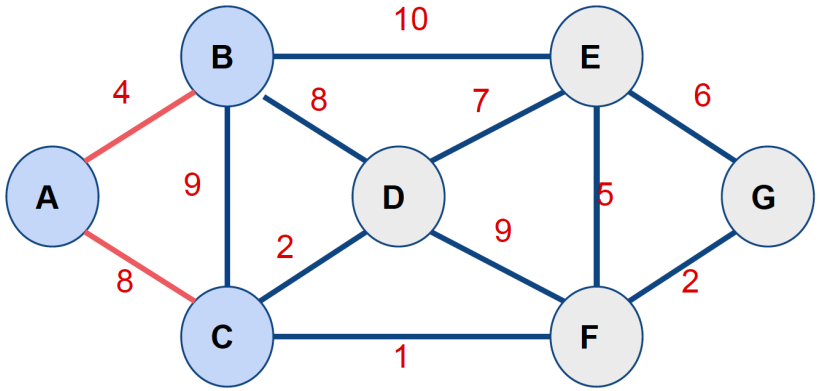


function DIJKSTRA($\langle V, E\rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are distances
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each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

- We eject the node with the lowest distance & update the queue.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*
a, b	*cost*			8	12	14	∞	∞
	prev			a	b	b	*nil*	*nil*

Dijkstra's Algorithm: Example

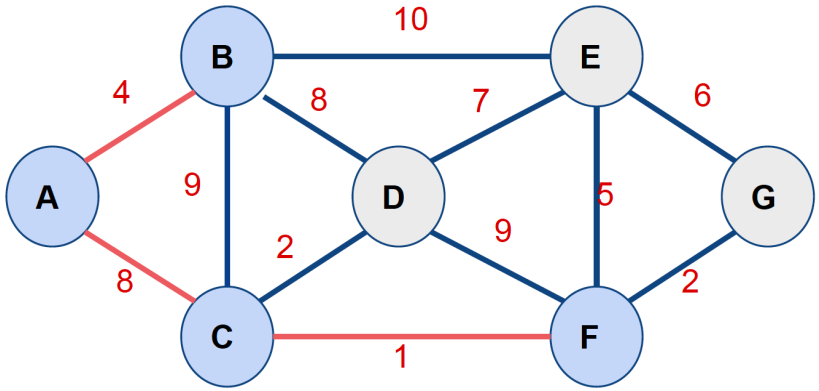



```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

- We eject the node with the lowest distance & update the queue.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*
a, b	*cost*			8	12	14	∞	∞
	prev			a	b	b	*nil*	*nil*
a, b, c	*cost*				10	14	9	∞
	prev				c	b	c	*nil*

Dijkstra's Algorithm: Example

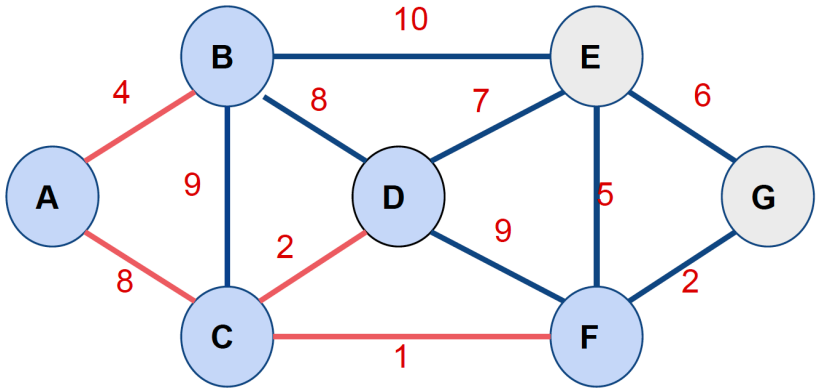


function DIJKSTRA($\langle V, E\rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are distances
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each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

- We eject the node with the lowest distance & update the queue.

Tree T		a	b	c	d	e	f	g
	$cost$	0	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil	nil
a	$cost$		4	8	∞	∞	∞	∞
	$prev$		a	a	nil	nil	nil	nil
a, b	$cost$			8	12	14	∞	∞
	$prev$			a	b	b	nil	nil
a, b, c	$cost$				10	14	9	∞
	$prev$				c	b	c	nil
a, b, c, f	$cost$				10	14		11
	$prev$				c	b		f

Dijkstra's Algorithm: Example

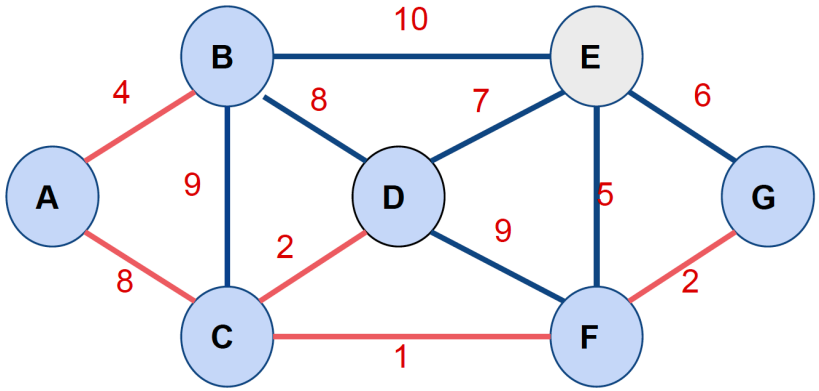


function DIJKSTRA($\langle V, E\rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are distances
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each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

- We eject the node with the lowest distance & update the queue.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*
a, b	*cost*			8	12	14	∞	∞
	prev			a	b	b	*nil*	*nil*
a, b, c	*cost*				10	14	9	∞
	prev				c	b	c	*nil*
a, b, c, f	*cost*				10	14		11
	prev				c	b		f
a, b, c, f, d	*cost*					14		11
	prev					b		f

Dijkstra's Algorithm: Example

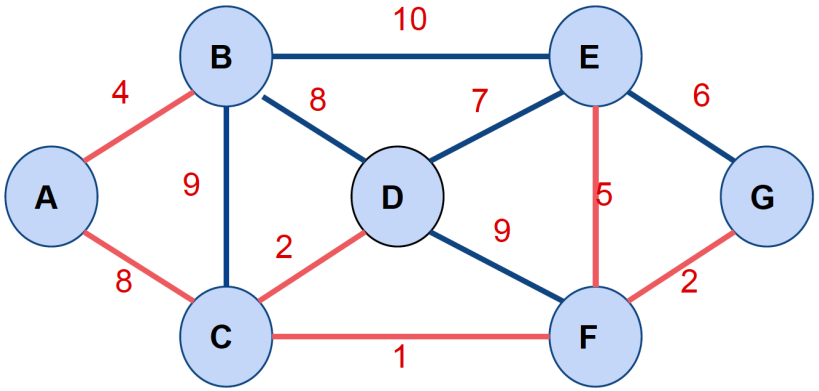


function DIJKSTRA($\langle V, E\rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are distances
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each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

- We eject the node with the lowest distance & update the queue.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*
a, b	*cost*			8	12	14	∞	∞
	prev			a	b	b	*nil*	*nil*
a, b, c	*cost*				10	14	9	∞
	prev				c	b	c	*nil*
a, b, c, f	*cost*				10	14		11
	prev				c	b		f
a, b, c, f, d	*cost*					14		11
	prev					b		f
a, b, c, f, d, g	*cost*					14		
	prev					b		

Dijkstra's Algorithm: Example



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

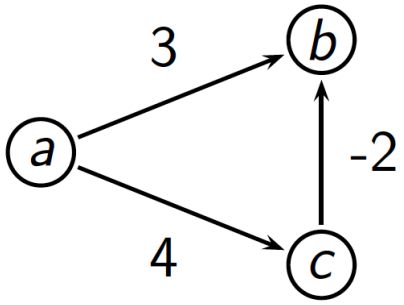
- We eject the node with the lowest distance & update the queue.

Tree T		a	b	c	d	e	f	g
	cost	0	∞	∞	∞	∞	∞	∞
	prev	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*	*nil*
a	*cost*		4	8	∞	∞	∞	∞
	prev		a	a	*nil*	*nil*	*nil*	*nil*
a, b	*cost*			8	12	14	∞	∞
	prev			a	b	b	*nil*	*nil*
a, b, c	*cost*				10	14	9	∞
	prev				c	b	c	*nil*
a, b, c, f	*cost*				10	14		11
	prev				c	b		f
a, b, c, f, d	*cost*					14		11
	prev					b		f
a, b, c, f, d, g	*cost*					14		
	prev					b		
a, b, c, f, d, g, e	*cost*							
	prev							

Negative Weights

- In our example, we used positive weights, and for 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.

Coming Up Next

- Huffman encoding for data compression (Levitin Section 9.4)